

MineSet™ 3.0 Enterprise Edition Interface Guide

Document Number 007-3993-001

CONTRIBUTORS

Written by Helen Vanderberg

Production by Linda Rae Sande

Engineering contributions by Barry Becker, Amit Bleiweiss, Jeff Brainerd, Cliff Brunk, Eben Haber, Ara Jerahian, Andy Kar, Ed Karrels, Eser Kandogan, Alex Kozlov, Alan Norton, Peter Rathmann, Mario Schkolnick, Dan Sommerfield, Peter Welch, and Brett Zane-Ulman.

St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

© 1999, Silicon Graphics, Inc.— All Rights Reserved

The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED AND RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Data clause at FAR 52.227-14 and/or in similar or successor clauses in the FAR, or in the DOD, DOE or NASA FAR Supplements. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy., Mountain View, CA 94043-1351.

Silicon Graphics, IRIX, and OpenGL are registered trademarks, and SGI, the Silicon Graphics logo, and MineSet are trademarks, of Silicon Graphics, Inc. INFORMIX is a registered trademark of Informix Software, Inc. Microsoft, Windows, Windows NT and Internet Explorer are registered trademarks, and ActiveX is a trademark, of Microsoft Corporation. Netscape and Netscape Navigator are registered trademarks of Netscape Communications Corporation. Oracle is a registered trademark, and SQL*Net is a trademark of Oracle Corporation. Sybase is a registered trademark, and SQL Server is a trademark of Sybase Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. X Window System is a trademark of the Massachusetts Institute of Technology.

The Tree Visualizer is patented under United States Patents No. 5,528,735, 5,555,354 5,671,381, and 5,861,885. The Splat Visualizer is patented under United States Patent No. 5,861,891. Patent pending for the 2D slider in the Map Visualizer, Scatter Visualizer and Splat Visualizer. Patent pending for the Evidence Visualizer.

MineSet™ 3.0 Enterprise Edition Interface Guide
Document Number 007-3993-001

Contents

	List of Tables	xi
0.	About This Guide	xiii
	Audience for This Guide	xiii
	Finding the Information	xiii
	Structure of This Document	xiv
	Typographical Conventions	xv
1.	Getting Started With MineSet	1
	Putting MineSet on Your System	1
	System Requirements	2
	Software Licence Agreement	2
	64-Bit Support for IRIX	2
	MineSet Tools Suite Overview	3
	About the Tool Manager	4
	Understanding DataMover	5
	MineSet Plug-In Capability	5
	Plug-In Functions	5
	Plug-In Transformations	6
	Plug-in Mining Tools	6
	Basic Tool Execution Scenario	6
	Installing Sample Datasets	9
	Connecting to a Database	9
2.	Setting Up MineSet	11
	Configuring MineSet	11
	Setting Up MineSet on IRIX Systems	12
	Location of Server Files on Windows Systems	12

	Configuring the DataMover Server	13
	User Configuration File	13
	Global Configuration File	16
	Using MineSet With Existing Data Files	19
	Importing Data Files	21
	Exporting Files	23
	Loading Sample Datasets	24
	Logging In To a DBMS	27
3.	File Exchange between MineSet and SAS (IRIX only)	29
	Overview	29
	Converting MineSet Data Files to SAS Data Sets	29
	-names <i>namefile</i> Command Line Option	30
	-svsc Option	31
	Converting SAS Data Sets Into MineSet Data Files	31
	The -nolabel Option	32
	The -names <i>namefile</i> Option	32
	The -nodata Option	32
	The -svsc Option	32
4.	MineSet Web Extensions	33
	Overview	33
	MineSet Web Extension Files	34
	MineSet Web Installation (Client)	35
	MineSet <i>mtr</i> Files	35
	Creating <i>mtr</i> Files	35
	Publishing on the Web	36
5.	Flat File Support for MineSet	37
	The Data File	37
	Data Types	38
	Arrays	39

The <i>.schema</i> File	41
Variable Names	41
Strings and Characters	42
Comments	42
File Statements	42
Data Statements	42
Input Options	45
Exceptions	45
6. Creating Data and Configuration Files for the Tree Visualizer	47
Tree Visualizer Data File	47
Tree Visualizer Configuration File	49
Sections	49
Options Files	49
Statements	50
Variable Names	50
Option Statements	51
Include Statements	51
Sinclude Statements	51
Strings and Characters	52
Keywords	52
Expressions	53
Input Section	55
Input Options	56
Expression Section	57
Hierarchy Section	58
Levels Statements	59
Key Statements	60
Aggregate Subsection	62
Aggregate Base Subsection	63
Expressions Subsection	64
Sort Statements	64
Hierarchy Options	65

	View Section	66
	Height Statements	67
	Base Height Statements	70
	Disk Height Statements	70
	Color Statements	71
	Base Color Statements	74
	Disk Color Statements	74
	Label Statements	75
	Message Statements	75
	View Options	77
7.	Creating Data, Configuration, Hierarchy, and GFX Files for the Map Visualizer	85
	The Data File	85
	Data Types	86
	Fixed Arrays	87
	The Configuration File	88
	Overview	88
	Keywords	91
	Expressions	92
	The Input Section	93
	The Expressions Section	100
	The View Section	101
	The Hierarchy File	109
	The <i>.gfx</i> File	110
8.	Creating Data and Configuration Files for the Scatter Visualizer	115
	The Data File	115
	Data Types	116
	Arrays	117
	Null Values	117

The Configuration File	118
Sections	118
Defaults Files	118
Statements	119
Variable Names	119
Options Statements	119
Include Statements	120
Sinclude Statements	120
Strings and Characters	120
Comments	120
Keywords	121
Expressions	122
The Input Section	123
File Statements	124
Enumeration Statements	124
Data Statements	126
Input Options	128
The Expressions Section	129
The View Section	130
Slider Statement	131
Entity Statement	131
Size Statement	133
The Max Clause	133
Color Statement	134
Axis Statement	137
Summary Statement	139
Drillthrough Statement	140
Message Statement	140
Execute Statement	141
Filter Statement	142
View Options	143

- 9. Creating Data and Configuration Files for the Splat Visualizer 145**
 - The Data File 145
 - Data Types 146
 - Null Values 147
 - The Configuration File 147
 - Sections 148
 - Defaults Files 148
 - Statements 148
 - Variable Names 149
 - Options Statements 149
 - Include Statements 149
 - Sinclude Statements 150
 - Strings and Characters 150
 - Comments 150
 - Keywords 150
 - The Input Section 151
 - File Statements 152
 - Enumeration Statements 152
 - Data Statements 154
 - Input Options 155
 - The View Section 155
 - Slider Statement 156
 - Opacity Statement 157
 - Color Statement 158
 - Axis Statement 161
 - Summary Statement 162
 - View Options 163
- 10. Creating Data and Configuration Files for the Decision Table Visualizer 165**
 - Sample File 166
- 11. Format of the Evidence Visualizer's Data File 167**

-
- 12. **Nulls in MineSet** 173
 - Semantics of Nulls 173
 - Representation of Nulls 174
 - Operations on Nulls 174
 - Arithmetic Expressions 174
 - Boolean Expressions 174
 - Relational Operations 175
 - Testing for Nulls 175
 - Aggregations in the Presence of Nulls 176
 - Sort Order for Nulls 177
 - Bins and Arrays With Nulls 177
 - 13. **ActiveX Visualization Control API for MineSet Visualizers** 179
 - Overview 179
 - Basics of Component Object Model 179
 - ActiveX Architecture 180
 - How MineSet's Visualization Controls Work 180
 - Recommended Requirements 181
 - ActiveX Controls 181
 - IVizCommon Methods 182
 - IScatterviz 189
 - ISplatviz 191
 - IMapviz 193
 - IEviviz 194
 - IDtableviz 196
 - ITree Visualizer 197
 - 1. **Further Reading and Acknowledgments** 201
 - Further Reading 201
 - Acknowledgments 205
 - Index** 207

List of Tables

Table 1-1	sysune Parameters	3
Table 2-1	Importable Data File Types	21
Table 6-1	Keywords for the Tree Visualizer	52
Table 7-1	Keywords for the Map Visualizer	91
Table 7-2	Operators Used With Expressions	92
Table 7-3	Characters That Can Follow the Percent Symbol in the Format String	96
Table 8-1	Scatter Visualizer Keywords	121
Table 8-2	Operators Used With Expressions	122
Table 8-3	Characters That Can Follow the Percent Symbol in the Format String	126
Table 9-1	Splat Visualizer Keywords	150
Table 9-2	Characters That Can Follow the Percent Symbol in the Format String	153

About This Guide

The *MineSet 3.0 Enterprise Edition Interface Guide* deals with technical features and advanced capabilities of the MineSet suite of data mining and visualization tools. Current information about the MineSet product can also be found on the World Wide Web at <http://mineset.sgi.com>.

Audience for This Guide

This guide is written for someone who is already familiar with the operation of MineSet from the Tool Manager. The *MineSet 3.0 Enterprise Edition Interface Guide* is written to help you make interconnections with MineSet and the outside world, export MineSet work to other applications, build custom applications using the ActiveX API to the 3D visualizers, and to help in some basic system administration tasks. Windows users will find familiar methods and pathnames given. For IRIX users a familiarity with UNIX commands is assumed.

Finding the Information

Much of this guide deals with the tasks involved with installing and running MineSet from the command-line or configuration files. In some cases programmatic interfaces are described. A chapter-by-chapter summary can be found in “[Structure of This Document](#).”

For background details and explanations, refer to the *MineSet 3.0 Enterprise Edition Reference Guide*. For information on how to use the MineSet tools, refer to the *MineSet 3.0 Enterprise Edition User's Guide for Windows*.

- MineSet provides a means for third parties such as AcPro to plug in to the application. If AcPro is installed, documentation can be found in `/usr/acpro/doc`. Late-breaking information can be found on the MineSet Web page.
- MineSet provides a means for third parties to integrate the 3D visualization tools into their own Windows applications using an ActiveX API (see [Chapter 13](#).)

Structure of This Document

The guide begins with installing MineSet. Subsequent chapters concentrate on specific tools and processes as shown:

[Chapter 1, “Getting Started With MineSet”](#)

This chapter answers questions raised at installation and provides an overview of MineSet operations.

[Chapter 2, “Setting Up MineSet”](#)

This chapter provides directions on installing and setting up the appropriate configuration files for MineSet, and how to connect to the database server.

[Chapter 3, “File Exchange between MineSet and SAS \(IRIX only\)”](#)

This chapter describes how to convert MineSet data files into SAS data sets, and the reverse.

[Chapter 4, “MineSet Web Extensions”](#)

This chapter describes how to interact with MineSet over the Web, and how to set up the necessary files to achieve secure remote viewing.

[Chapter 5, “Flat File Support for MineSet”](#)

This chapter describes the files that are necessary to run MineSet, and gives examples of the *.data* and *.schema* files.

[Chapter 6, “Creating Data and Configuration Files for the Tree Visualizer”](#)

This chapter provides a description of the necessary files for the Tree Visualizer and gives appropriate examples.

[Chapter 7, “Creating Data, Configuration, Hierarchy, and GFX Files for the Map Visualizer”](#)

This chapter provides a description of the files needed to enable the Map Visualizer to run, together with examples.

[Chapter 8, “Creating Data and Configuration Files for the Scatter Visualizer”](#)

This chapter describes the Scatter Visualizer configuration and data files with examples.

[Chapter 9, “Creating Data and Configuration Files for the Splat Visualizer”](#)

This chapter describes and gives examples of the files necessary for the Splat Visualizer.

Chapter 10, “Creating Data and Configuration Files for the Decision Table Visualizer”

This chapter provides a description of the Decision Table data and its data and configuration files.

Chapter 11, “Format of the Evidence Visualizer’s Data File”

This chapter provides a description of the Evidence Visualizer and its data and configuration files.

Chapter 12, “Nulls in MineSet”

This chapter describes how nulls are handled in MineSet.

Chapter 13, “ActiveX Visualization Control API for MineSet Visualizers”

This chapter describes how application developers can use ActiveX controls to create custom data mining solutions for Windows platforms with MineSet’s visualizers.

Appendix A, “Further Reading and Acknowledgments”

This appendix acknowledges the resources engaged in establishing MineSet, and provides a resource for further reading.

Typographical Conventions

The following type conventions and symbols are used in this guide:

Italics Executable names, filenames, program variables, tools, utilities, variable command-line arguments, and variables to be supplied by the user in examples, code, and syntax statements.

Bold Keywords

Fixed-width type
On-screen command-line text and prompts.

Bold fixed-width type
User input, including keyboard keys (printing and non-printing); literals supplied by the user in examples, code, and syntax statements.

Getting Started With MineSet

This chapter provides a brief overview of the quickest way to install MineSet and start working. A summary of the basic tool execution scenario helps you understand which parts of MineSet should be installed. The sections are:

- [“Putting MineSet on Your System” on page 1](#)
- [“MineSet Tools Suite Overview” on page 3](#)
- [“About the Tool Manager” on page 4](#)
- [“Understanding DataMover” on page 5](#)
- [“MineSet Plug-In Capability” on page 5](#)
- [“Basic Tool Execution Scenario” on page 6](#)
- [“Installing Sample Datasets” on page 9](#)
- [“Connecting to a Database” on page 9](#)

Note: Before using any of the MineSet tools, follow the licensing instructions in the MineSet release notes. If you are installing on an IRIX system, you must set up the DataMover configuration file, and also can choose to enable various options. The setup details are described in [Chapter 2, “Setting Up MineSet.”](#)

Putting MineSet on Your System

You can install MineSet either from the Web or the CD. If you install over the Web, you are asked if you want to accept transmission in sections. If you accept this option, the sections are reassembled on your system, and installation proceeds in the normal manner. Whether you install from the Web or the CD, you will be asked to answer some basic questions, the details of which are discussed here. If you prefer, you can simply respond to the online questions and skip this section.

MineSet runs in a client server manner. The client and server can both be on your system, or on different systems. When client and server are on different systems MineSet seeks the name of the remote system.

On Windows systems you can run both server and client on Windows NT version 4.0 and above. Installing MineSet on a Windows system is simple—click “Install,” and follow the series of screen prompts. Installation proceeds automatically.

On IRIX systems you have a choice of which version of the MineSet server is installed—single or parallel. Choose single if you are working with a single CPU, multiple if you can spread the work among several CPUs. Parallelization is offered through the multiprocessor (n64) version of MineSet only.

System Requirements

System requirements depend on which components of MineSet you choose to install. The details of disk space availability and requirements are shown on the installation screens. If this your first experience with MineSet, it is recommended that you install everything.

Software Licence Agreement

You must accept the license agreement to run MineSet permanently. You can run a demonstration version for 30 days without a license. These details are covered in the release notes, and the Web site: <http://mineset.sgi.com>.

64-Bit Support for IRIX

Large memory (64-bit) is supported on IRIX 6.4 and later releases. If you have IRIX 6.2, you can still use the 32-bit data mining utility, but you must upgrade to IRIX 6.5 in order to obtain 64-bit support and pthreads. To get the full advantage of 64-bit addressing you may also need to change the systune resource parameters, depending on your system configuration.

The systune parameters determine the default limits on the available system resources. [Table 1-1](#) lists the systune parameter values that Silicon Graphics recommends (for more details see the [systune\(1M\)](#) reference page):

Table 1-1 systune Parameters

Parameter	Definition	Recommended Value
<code>rlimit_pthread_cur</code>	Current limit on the number of threads	1024
<code>rlimit_rss_cur</code>	Current limit on memory usage	The amount of physical memory on your machine
<code>rlimit_vmem_cur</code>	Current limit on virtual memory usage	The size of the logical swap space on your machine or about twice the physical memory
<code>rlimit_nofile_cur</code>	Current limit on number of open file	1024 or the limit on the number of threads

Note: You must reboot your machine after installing the new parameters.

MineSet Tools Suite Overview

The MineSet suite of tools lets you mine and graphically display quantitative information in ways that can help you better visualize, explore, and understand your data. These tools provide a highly interactive, three-dimensional (3D) visual interface that lets you manipulate visual objects on the screen, as well as search, filter and perform animations.

The MineSet suite consists of three basic components:

- **a centralized control module**, consisting of a graphical user interface tool called the *Tool Manager*, and a process called the *DataMover*, which runs on the server part of MineSet's client/server architecture.
- **analytical data mining**, with nine data mining tools:
 - Association Rules Generator
 - Automatic Binning
 - Cluster Generator
 - Column Importance

- Decision Table Inducer and Classifier
- Decision Tree Inducer and Classifier
- Evidence Inducer and Classifier
- Option Tree Inducer and Classifier
- Regression Tree Inducer and Regressor
- **visualization tools**, which let you view your data using ten different visual metaphors:
 - Cluster Visualizer
 - Decision Table Visualizer
 - Evidence Visualizer
 - Map Visualizer
 - Record Viewer
 - Scatter Visualizer
 - Splat Visualizer
 - Statistics Visualizer
 - Tree Visualizer

For a complete description and directions for using the tools see the *MineSet 3.0 Enterprise Edition User's Guide for Windows*, the *MineSet 3.0 Enterprise Edition Reference Guide* or the *MineSet 3.0 Enterprise Edition Tutorial for Windows*.

About the Tool Manager

Each of the mining and visualization tools can be configured and started using the consistent graphical user interface called the Tool Manager. The Tool Manager:

- connects you to the server on which the analytical mining and transformations are performed
- lets you access, query and transform data
- creates configuration files for each tool

The Tool Manager is installed on the MineSet client.

Understanding DataMover

The DataMover is a process that runs on the server. The server performs the following actions:

- connects to databases, or flat files (ASCII or binary), and retrieves the data
- invokes the mining tools
- performs additional data manipulation such as binning and aggregation
- returns the data to the Tool Manager for distribution to the visualization tools
- stores the data in files on the server or client for future operations.

You can run MineSet in client-only mode with limited functionality, provided you have access to the server.

MineSet Plug-In Capability

Mineset 3.0 supports a new plug-in architecture that allows user to add new mining tools, functions, and data transformations to MineSet. These plug-ins are installed and executed on the MineSet server, and any user interface for the plug-ins is downloaded by the Mineset client. The MineSet Plug-in API is identical on IRIX and Windows, so plug-ins can be installed on IRIX or Windows servers.

Plug-In Functions

Plug-in functions are added to the list of available functions when forming Add Column or Filter expressions. They are implemented by writing and implementing a C function with a particular prototype, compiling that into a shared library, and putting the shared library on the MineSet server computer.

Plug-In Transformations

Plug-in transformations are accessible from the “Plug-in Ops” button in the Data Transformations pane of the Tool Manager. They are also visible in the View History panel. A plug-in transformation is implemented with a standalone executable that is given a data set as input, and produces a data set as output. The MineSet Plug-in SDK will ship with a library of functions that will assist in reading and writing MineSet data sets. Plug-in transformations also require a Java GUI that will be sent to the MineSet client, and incorporated into the Tool Manager.

Plug-in Mining Tools

Plug-in mining tools are added as new tabs in the “Mining Tools” tabbed deck in the Tool Manager. They are implemented as standalone executables on the MineSet server that are given data sets or model files as input, and produce data sets or model files as output. Like plug-in transformations, plug-in mining tools also include a Java GUI that will be added to the Tool Manager.

Basic Tool Execution Scenario

Each of the MineSet tools is started, configured, and run in a consistent manner. The sequence of actions followed by the MineSet client and the MineSet server is shown schematically in [Figure 1-1](#). A description of the steps inherent in this figure follows.

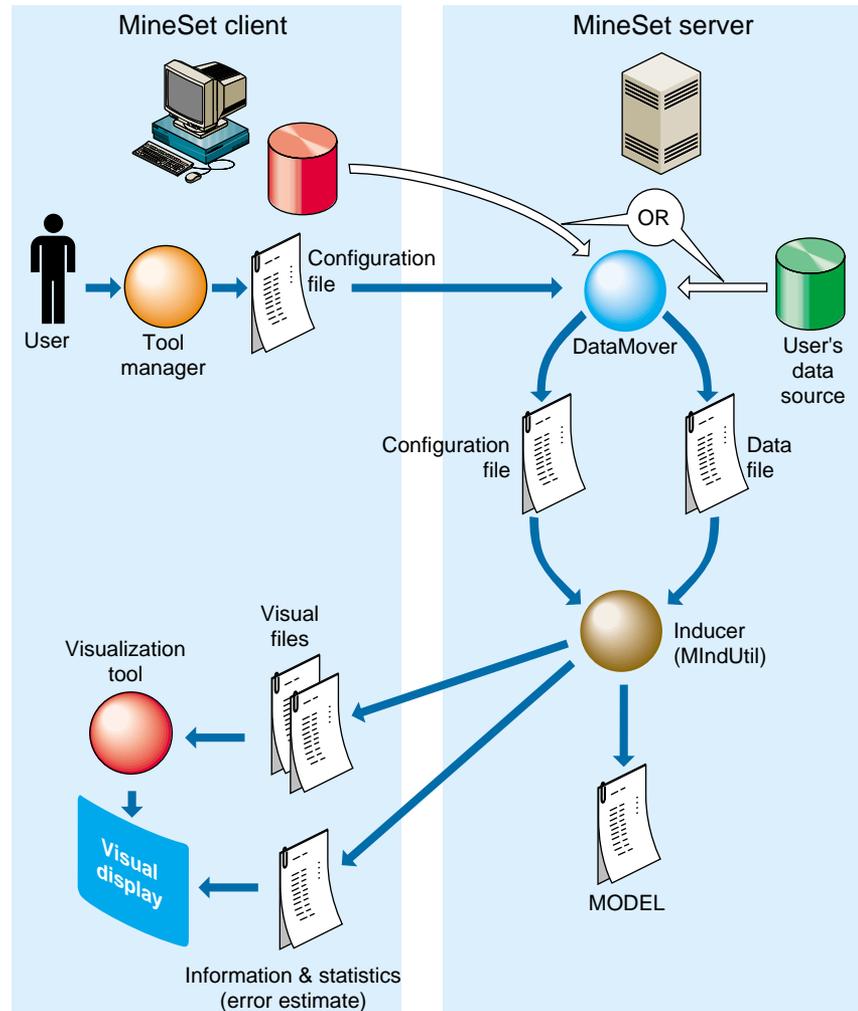


Figure 1-1 Tool Execution Sequence

The following steps describe a “typical” interaction with a MineSet tool, and the sequence of the tool’s actions. Depending on your requirements, some steps might be skipped (for instance, if the data and configuration files have been generated in a previous work session).

1. Start the Tool Manager, which is the graphical interface for generating and specifying the configuration file, data file, and tools to be used. The Tool Manager runs on your MineSet client.
2. The Tool Manager opens a network connection to the DataMover, which runs on the MineSet server, which in some cases may be the same as your client workstation, and in others is a separate machine.
3. Use the Tool Manager to specify
 - the database and table, or a binary or ASCII flat file containing the data on either the client or the server
 - which mining or visualization tools are to be applied
 - how that data is to be displayed, through tool options
 - a session file to save the history of your work

Information retrieved using the DataMover guides this interaction. As a result, the Tool Manager generates a configuration file. This file contains the user-defined parameters that determine the execution of the following steps.

4. The Tool Manager transmits a copy of the configuration file from step 3 to the DataMover. The DataMover processes the file by
 - accessing the database or flat file
 - performing the specified data transformations
 - running the mining tools when requested
 - generating the visualization files when requested

These visualization files consist of your data in a specific format readable by the MineSet tool. Then a copy of these visualization files is transferred to the MineSet client.

5. The Tool Manager invokes the appropriate MineSet visualization tool.
6. The tool accesses the visualization files and displays the data.
7. If you generated a model, that model can be applied to additional data.

Note: The MineSet client and server can run on different machines, using a network to communicate. Because network bandwidth is often scarce, you should be cautious about transferring large files between client and server regularly. If you are doing mining operations on a large database or file, you can achieve greater efficiency by storing that file on the server where the DataMover runs, rather than on the client.

Installing Sample Datasets

When you install MineSet you have the choice of installing the sample datasets and visualization examples. If you are new to MineSet, it is strongly suggested that you install these sample files, as well as the complete set of documentation including the Tutorial. The Web site can provide online videos that can be helpful in understanding how some of the mining tools can be used.

Connecting to a Database

Your next requirement is preparing to connect to a database to use your own data. This means certain configuration files must be modified to identify the source of your data. The data files themselves may also require modification to conform with MineSet's expectations. Both of these steps can be done using any familiar text editor. Instructions are provided in [Chapter 2, "Setting Up MineSet."](#)

Setting Up MineSet

This chapter describes how to set up MineSet on either Windows or IRIX systems. The subjects discussed are:

- “Configuring MineSet” on page 11
- “Configuring the DataMover Server” on page 13
- “Importing Data Files” on page 21
- “Exporting Files” on page 23
- “Loading Sample Datasets” on page 24
- “Logging In To a DBMS” on page 27

In addition, examples of loading the sample files into Oracle, Sybase and INFORMIX are given.

Configuring MineSet

There are two major pieces of configuration information in MineSet: in which directory to store the user’s temporary and persistent files on the server machine, and which databases the server can connect to.

On IRIX systems this configuration is defined by global and per-user configuration files. On Windows these configuration files are still available, but most information is stored in the registry, hence Windows users have less need to edit configuration files.

Setting Up MineSet on IRIX Systems

Setting up MineSet on IRIX systems requires modifying *dm_config* and the DataMover configuration file *.datamove*. The DataMover process runs on the server, and is not directly accessible to users. The DataMover provides access to databases and data stored in flat files, and transforms data for the mining and visualization tools. The configuration has two parts:

- configuring the user's account on the server, and
- a global configuration by the system administrator.

The next section describes these files and how they may be modified. The last section of this chapter describes how to load sample datasets into the supported relational databases.

Location of Server Files on Windows Systems

The MineSet server uses a hybrid strategy to determine where on the server to store model, schema, and data files. At installation time, the Installer prompts for a base directory for all user's server files. Each user then is assigned a subdirectory of this base directory, according to username. For example, if *ServerFilesBase* is left at its installation default of *C:\MineSet Files*, user `guest` will have MineSet server files located in *C:\MineSet Files\,uest*.

An individual may override this machine-level default by setting the registry entry *HKEY_CURRENT_USER\SOFTWARE\SGI\MineSet\3.0\ServerFilesDirectory*. (To set a registry entry, right-click on the My Computer icon on your desktop, and choose Preferences.) If this entry exists, its value will set the server files directory for that user only.

The different versions of Windows support a number of different filesystems, the most common of which are NTFS and FAT. In general, it is better to locate the server files directory on an NTFS filesystem, if possible. The MineSet server allows multiple instances of the MineSet client to connect to the same server files directory. This can be convenient, if one user wants to try several experiments in parallel, or if two or more individuals want to share data and model files. On FAT filesystems, the DataMover uses a coarse-grained locking mechanism to maintain consistency, which limits at most one client to be running a history at one time from within a given server files directory. Other users attempting to run histories will get a "Failed to get a lock" error message. On NTFS, the DataMover's locking implementation is finer grained, and users will rarely collide in this way.

Finally, whichever way the server files directory is determined, the MineSet server will then look for a file *.datamove* in that directory, which can contain additional configuration information, as described in the next section. The format and entries for the *.datamove* file are the same on Windows and IRIX. However, the most common use of the *.datamove* file on is setting the location of server files, and this is already handled by the registry on Windows. Hence, most users of the Windows MineSet server will have little need to create a *.datamove* file.

Configuring the DataMover Server

To use the MineSet tools, two configuration files may be created on the server: one by the user—the *.datamove* file, the other by the system administrator—the *dm_config* file.

User Configuration File

Note: Each user must have an account on every server to be accessed.

The DataMover configuration file, *.datamove*, creates files on the server system for each user. This *.datamove* file lets you control where these user files are created and whether different classes of files are saved or discarded. On IRIX systems, the *.datamove* file is located on the server, in the user's home directory. A sample *.datamove* file called *datamove.sample* is located on the server in the following directories:

- Windows users: *MINESET_HOME\config\datamove*
- IRIX users: */usr/lib/MineSet/datamove*

If the *.datamove* file is absent, or if a particular entry is not present in the *.datamove* file, the DataMover uses a default value for that entry.

Each entry in the DataMover's configuration file must be on a separate line. For example:

```
file_cache = directory_name
```

where *file_cache* specifies the location in which the DataMover stores its output data files and models resulting from mining algorithms. If the *file_cache* directory does not exist, the DataMover attempts to create it on its first invocation. On Windows the default *file_cache* directory is determined by the registry (see above); on UNIX, the default *file_cache* directory is *./mineset_files/%U*. The *%U* is a wildcard that is filled in with the user's login name on the client machine.

The *file_cache* should be a directory in a filesystem with sufficient room to hold all of a user's output and temporary files. DataMover will create this directory if it does not already exist. Intermediate files are deleted when the DataMover no longer needs them, unless one of the following **keep** options is set:

```
keep_client_upload
keep_client_download
keep_classifier_files
keep_mlc_input
use_ascii_mlc_input
```

Each of these entries is described below.

```
keep_client_upload (default no)
```

Keep files uploaded from the client for processing. If kept, they will be in the *client_upload* subdirectory.

```
keep_client_download (default no)
```

Retain on the server a copy of data files and visualizations after they are downloaded to the client. If kept, the files will be in the *client_download* subdirectory.

```
keep_classifier_files (default yes)
```

Keep the persistent classifiers (decision trees and so forth) generated by mining operations. The method is generally useful.

```
keep_classifier_options_files (default no)
```

Keep the options file that is used when generating, or inducing, the classifier. This tactic is not useful. If kept, the files will be in the *mlc_work* subdirectory.

```
keep_mlc_input (default no)
```

Keep input files used for mining (MIndUtil or associations) operations. If kept, the files will be in the *mlc_work* subdirectory.

```
use_ascii_mlc_input (default no)
```

Normally the DataMover creates MineSet binary files for MIndUtil input. If this option is set, create ascii files instead.

```
aggregation_memory_limit (default 2147483647)
```

Memory limit (in bytes) for aggregation operations. This can be no larger than the system-wide limit set in the *dm_config* file.

```
optimize_history=yes
```

The DataMover is able to rewrite histories to remove redundant computations. The *optimize_history* parameter controls whether or not to do this. Since this rewriting can speed up processing considerably, it is normally turned on.

How Files Are Handled

A file in the *file_cache* directory is the result of a successful operation. If an operation returns an error (that is, the Tool Manager reports a message beginning *fatal error on server*) nothing should be changed in the *file_cache* directory. Two examples help illustrate the point:

- Example 1: A user's *file_cache* directory contains the files *cars.data* and *cars.schema*, both the result of a previous database query. The user then selects the same table, and sets the output to *server_file*, filtering for examples with *mpg*>55. Since no records in the dataset have *mpg* values this high, when the history executes, it returns no rows, which is flagged as a fatal error. After this happens, the user's *file_cache* directory will still contain the old *cars.schema* and *cars.data* files.
- Example 2: A user's *file_cache* directory contains the files *cars.data* and *cars.schema*, both the result of a previous database query. The user then selects the same table, and sets the output to a visualization. The operation completes and the visualization launches successfully. Once again, the user's *file_cache* directory still contains the old *cars.schema* and *cars.data* files. The *file_cache* directory is not updated unless the user specifically chooses *server_file* as the output.
- Example 3: Two client users are connected to the same server account, and the same server files directory. The first user starts a long-running mining operation on the *adult94.schema* and *adult94.data* server files. Before that mining operation completes, another user executes a database query and replaces the *adult94.schema* and *adult94.data* files with new versions. Neither user will see any errors; the first user's model will complete based on the earlier version of the data file. If the users were connecting to a Windows system with the server files directory on a FAT filesystem, the second user's operation would exit with a "failed to get a lock" error.

Global Configuration File

On IRIX, the DataMover works with Oracle versions 7.2 or later, INFORMIX and Sybase; on Windows, the DataMover supports ODBC and OLE-DB data sources, and native connections to Oracle.

If you are using relational databases, you may need to configure the server to find these databases. ODBC and OLE-DB data sources are listed in the Windows system registry and not in the *dm_config* file. Hence a Windows server using only ODBC and OLE-DB data sources will not need a *dm_config* file. Windows users will only need to create a *dm_config* file if they want to use a native connection to a local or remote Oracle database. Oracle may also be accessed with ODBC or OLE-DB, but the native connection will generally give better performance. On IRIX, the *dm_config* file may be omitted if the server is not intended to access any database.

This section details the DataMover configuration file, *dm_config*, which is read by the DataMover server during start up. File location is:

- Windows users: from the directory in which MineSet is installed
`\config\datamove\dm_config`
- IRIX users: `/usr/lib/MineSet/datamove/dm_config`.

This file is not created automatically during installation. It must be created by the system administrator, who must log in as root to edit this file. It can be created using any simple text editor such as Notepad or Emacs. An example file, *dm_config.sample*, can be found in the *datamove* directory. The format of this file is as follows:

```
Oracle {
"ORACLE_SID", "ORACLE_HOME";
}

Oracle_Remote {
"SERVICE_NAME", "ORACLE_HOME";
}

Informix {
"INFORMIXSERVER", "INFORMIXDIR";
}

Sybase {
"DSQUERY", "SYBASE";
}
```

The line "ORACLE_SID", "ORACLE_HOME" is filled in with the specific information and repeated once for each Oracle database to be accessed via the DataMover. ORACLE_SID and ORACLE_HOME are Oracle specific parameters defining an Oracle instance.

The Oracle_Remote section is for accessing remote Oracle databases via Oracle Networking. The SERVICE_NAME entry is a logical name for the remote database. Such logical names are defined by editing an Oracle-defined configuration file (for example, *tnsnames.ora*), or, depending on the Oracle version and operating system, with an Oracle-supplied applet, for instance Net8 Easy Config. The second ORACLE_HOME parameter is the directory of your client installation. On IRIX systems only, it is often possible to do without an Oracle client installation. In this case the second parameter is interpreted as the directory where DataMover searches for the *tnsnames.ora* file. This file is described in Oracle's Networking documentation.

Each line in the Informix section defines a database server that, in turn, can contain several databases. The server is checked at runtime to determine which databases it contains, so there is no need to record the individual databases in the *dm_config* file. The first entry is the INFORMIX server (corresponding to the *INFORMIXSERVER* environment variable), and the second is the INFORMIX directory (corresponding to the *INFORMIXDIR* environment variable).

Each entry in the Sybase section defines a database server (or, in Sybase terminology, an SQL Server). The first entry is the Sybase SQL Server name (corresponding to the *DSQUERY* environment variable); the second is the Sybase home directory (corresponding to the *SYBASE* environment variable). This may be a remote database.

Note: A Windows configuration file defines different pathnames, but the structure follows conventions similar to the following example.

An IRIX example configuration file shows:

```
Oracle {
"v81", "/ora/app/oracle//product/8.1.5";
"wrhse", "/opt/oracle";
}

Oracle_Remote {
    "lifeseq", "/usr/lib/MineSet2/datamove/";
#native connections to Informix and Sybase supported on IRIX only
}

Informix {
"learn_online", "/u5/informix";
}
```

```
Sybase {  
  "MINESET", "/usr/sybase/11.5.1";  
}  
  
# Following parameter is relevant only if your database is in a  
# non-English locale. Default of TRUE allows database to perform  
# automatic translation of character encoding (if necessary) to  
# compensate for any mismatch in locale between the database and the  
# MineSet client. Setting parameter to FALSE disables this  
# adjustment, which is sometimes necessary with very old or  
# misconfigured databases.  
  
enable_database_locale_adjustments = TRUE;
```

The `enable_database_locale_adjustments` is common to both platforms.

This configuration file lets the DataMover access:

- three Oracle databases, one named *v73* (installed in */usr/people/oracle/v73*), another named *wrhse* (installed in */opt/oracle*), and a remote database named *lifeseq*
- an INFORMIX Server
- a Sybase SQL Server

Each of the INFORMIX and Sybase servers can, in turn, contain multiple databases.

For Sybase, DataMover uses vendor-supplied shared libraries as its connection to the databases. One of the purposes of the *dm_config* file is to specify where DataMover must look for its shared libraries.

DataMover looks for the shared libraries *libct.so*, *libcs.so*, *ibcomn.so*, *libintl.so*, *libtcl.so*, and *libinsck.so* on IRIX systems in the *\$\$SYBASE/lib/* directory.

The IRIX DataMover is compiled as an n32 program and hence needs to load the n32 versions of the Sybase client libraries. n32 client libraries come with Sybase versions 11.5 or later. If there is a need to use MineSet 3.0 with an earlier version of Sybase, the user will need to install a recent version of the Sybase client.

A Windows example of the *dm_config* file shows:

```
# Connection to a local Oracle database on Windows NT.  First entry is
# the Oracle sid, and second entry is the ORACLE_HOME.  Note that if
# '\' is used as a directory separator, it must be doubled.

Oracle{
    "mse", "F:\\Oracle\\Ora81";
}

# 3-tier access to remote Oracle databases via Oracle networking.
# First argument is the name of the data source, as created by the
# Net8 configuration assistant or listed in the tnsnames.ora file.
# Second argument is the local ORACLE_HOME directory.  At least a
# client install of Oracle is required to connect to remote Oracle
# databases.

Oracle_Remote {

    "uci_megamine", "F:/Oracle/Ora81";
    "MSE_NET8.ENGR.SGI.COM", "F:/Oracle/Ora81";
    "DEMO_MEGA", "F:/Oracle/Ora81";
    "mega_v72", "F:/Oracle/Ora81";
}
```

Using MineSet With Existing Data Files

Sometimes it is convenient to use MineSet with data that is already stored as a file, but requires further processing before it can be mined or visualized. In this case, the data file can be made available (with a modest effort) to the Tool Manager/DataMover.

First, the data file must be in a tab-delimited format, with the same number of fields in each line. A numeric or string field with a single “?” character appearing between delimiters is loaded as a Null value. Importing data files automatically using the MineSet Tool Manager is detailed in [“Importing Data Files” on page 21](#). A detailed example of the MineSet .schema file format can be found in [Chapter 5, “Flat File Support for MineSet.”](#)

The contents of the data file must be described to the Tool Manager/DataMover using a file with the *.schema* extension. The format of the *.schema* file is shown next:

```
#
# A line beginning with a "#" is a comment
#
input {

# The first line lists the data file which is described. It
# must be a simple filename, not a path.

    file "carmodels.data";

# Fields are listed left to right in the line, legal
# types are float, double, int, string, date, fixedString and
# dataString
# Be sure to end every line with a semicolon ";"

    float mpg;
    int cylinders;
    float cubicinches;
    int horsepower;
    int weightlbs;
    double timeaccelerate;
    date when_introduced;
    string origin;
    fixedString(3) manufacturer_code;
    dataString model;
}
```

The schema and data files must be located in the same directory. If you prepare a dataset in this fashion on the client machine, it can be opened with the Tool Manager's *Find File* dialog. If the file requires any additional processing, it is copied to the server. Sometimes this is not convenient, especially if the file already exists on the server, or is large. In this case, the *.schema* and *.data* files must be copied (or symbolically linked) into your *file_cache* directory on the server.

Importing Data Files

This section shows you how to import files from other formats using the MineSet Tool Manager.

To import a file, choose the “Import” option from the “File” menu. A file dialog appears allowing you to choose the file you wish to import, along with a number of import options.

The format of the imported file may be selected from a box at the top of the options portion of the file dialog. The supported file formats are:

Table 2-1 Importable Data File Types

File Type	Description of Source
ASCII File - Delimited	A text file with fields separated by a delimiter character.
ASCII File - Fixed Format	A text file where each value for a particular column has the same width.
1-2-3	Lotus 1-2-3
DBASE III	DBase II, III+, IV, and compatible systems such as Clipper or Alpha Four.
Epi Info	Epi Info through Version Six.
Excel '97	
Microsoft Excel	
FoxPro	
Gauss	Either Gauss 89 or Gauss 96.
JMP	SAS JMP statistics files.
Limdep	
LIMDEP version 7 for Windows	
Matlab	Matlab matrices.
Minitab	Minitab versions 8 through 12.

Table 2-1 (continued) Importable Data File Types

File Type	Description of Source
Osiris	OSIRIS Type 1 data sets.
Paradox	
Quattro Pro	
SAS Data File	SAS version 6.08 and above.
SAS Transport File	
SigmaPlot	
S-Plus	32-bit S-PLUS data sets.
SPSS Data File	
SPSS Portable File	
Stata	Any version of Stata.
Statistica	Statistica version 5.
Systat	Either double or single precision SYSTAT files.

When importing ASCII files (both delimited and fixed format), the following options may be specified:

- **Maximum Line Width (default 4096):** this is the maximum width for lines in the file to be imported. If the file is wider than 4096 characters, you must specify the width here.
- **Maximum # Columns (default 255):** this is the maximum number of columns in the file. If the file has more than 255 columns, specify how many there are here.
- **Two digit year cutoff (default 69):** When processing date values from an ASCII file, if a year is specified with only two digits, years greater than or equal to this number are considered to be in the 20th century. For example, given the default value of 69, the date 2/3/69 becomes 2/3/1969, but 2/3/68 becomes 2/3/2068.

- **Decimal separator:** the character used as a decimal point when processing numbers. The default value is taken from the current locale, e.g., ' in the USA and ',' in most European countries.
- **Separator every 3 zeros:** sometimes numbers have punctuation every 3 digits, e.g., 1,000,000.345. This field allows you to specify that character. The default value is taken from the current locale, e.g., ',' in the USA and '.' in most European countries.

When importing delimited ASCII files, it is also possible to specify the delimiter between columns. If you select the “Outshines delimiter” radio button, the delimiter will be automatically detected from either comma, tab, semicolon, or space. If you select the “Specify delimiter” radio button, you can enter a different delimiter character.

When importing fixed-width ASCII files, it is necessary to provide a .SCH file to indicate where the fields begin and end. An example of the format follows:

```
[alcoholSample]
Filetype=Fixed
Date-Order=MDY
Date-Punctuation=/
Dec-Point=.
CharSet-ascii
Field1=household_id,Float,16,0,0
Field2=pdate, Date,10,0,16
Field3=name,Char,255,0,26
Field4=age,Float,16,0,281
```

It starts with the name of table, followed by the file type, date order, date punctuation, decimal point, character set, and a list of field names, types, length, zero, and the starting location for the field.

Exporting Files

Exporting files is possible from the Tool Manager Data File Destination panel, which has three radio buttons:

- MineSet ASCII File
- MineSet Binary File
- Other file types

The first two buttons allow the user to specify whether the output file (on the client or server) should use the ASCII or Binary formats.

The third button, available only when creating client files, also converts the output client file to a different data format specified in the combo box next to the “Other file types” button. Available file types are the same as those in the Import dialog, with the exception of Osiris.

Loading Sample Datasets

This section describes how to load the sample datasets included with the MineSet distribution into one of the supported relational databases.

Installed on the server are:

- all the sample data, along with a brief description of what it contains
- directions on how to load the data using the provided scripts

Load the sample datasets into a database that has been set up on your server. The data and directions (*README.server*) are installed in the directory in which MineSet is installed under *\Examples\DBexamples* on Windows systems. IRIX users find them in */usr/lib/MineSet/DBexamples* on the server.

This directory contains scripts for loading the complete set of data files into one of the supported databases. To load the complete set of data, run one of the following loader scripts, depending on which database you have. (This assumes your database and environment are already set up.)

```
sh load_all_Oracle.sh userid passwd
```

```
sh load_all_Sybase.sh userid> <passwd
```

If you are going to work with an INFORMIX database, use the `dbaccess` interface to select

```
create_all_Informix.sql
```

followed by

```
load_all_Informix.sql
```

Loading Individual Datasets

Alternatively, you can load, or reload, the sample data separately. Each data directory in *DBexamples* on the server contains files necessary to load the data into any of the supported databases. Windows users find these files in the directory in which MineSet is installed under *\Examples\DBexamples*. IRIX users find them in */usr/lib/MineSet/DBexamples*. These files are:

README - explains the data

**.sql* - sets up an Oracle table

**.ctl* - control file for loading into Oracle

**_syb.sql* - sets up a Sybase table

**.bcf.fmt* - Sybase format file

**_inf.sql* - sets up an INFORMIX table

**_load.sql* - loads the data into the INFORMIX table

In the **.ctl* file, the separator is declared in the line

```
" fields terminated by X'20' "
```

The separator is specified in ASCII hexadecimal; thus:

X'20' is used for ' '

X'2c' is used for ','

X'09' is used for '\t'

Loading Into Oracle

Perform the following steps on the server with an Oracle database:

1. Ensure the following environment variables are set correctly:

```
ORACLE_HOME
```

```
ORACLE_SID
```

2. Type

```
sqlplus <userid>/<passwd>  
SQL> @<dataset>.sql
```

Where *dataset* is the name of the dataset being loaded, and *userid/passwd* are your assigned username and password for the Oracle database.

To delete an already existing table, type

```
SQL> drop table <dataset>;
```

3. Type

```
sqlload control = <dataset>.ctl userid = <userid>/<passwd>  
log = /tmp/<dataset>.log direct = true
```

4. Check the resulting *dataset.log* to ensure the data was loaded correctly.

Loading Into Sybase

Perform the following steps on the server with a Sybase database:

1. Ensure that the following environment variables are set:

```
SYBASE  
DSQUERY
```

2. To create the table, type

```
isql -U<userid> -P<passwd> -i <dataset>_syb.sql
```

Where *dataset* is the name of the dataset being loaded, and *userid/passwd* are your assigned username and password for the Sybase database.

To delete an already existing table, type

```
isql -U<userid> -P<passwd>  
drop table <dataset>  
go
```

3. To load the data, type

```
bcp <dataset> in <dataset>.data -U<userid> -P<passwd> -f  
<dataset>.bcp.fmt
```

where *dataset* is the table name (created using *<dataset>_syb.sql*), *in* means "load into the dbms," *<dataset>.data* refers to the name of the ASCII data file, and *-f* points to the already-created format file. (When reading in from a file, the data types are character.)

Loading Into INFORMIX

Perform the following steps on the server with an INFORMIX database:

1. Ensure the following environment variables are set:

```
ONCONFIG  
INFORMIXSERVER  
INFORMIXTERM
```
2. To create the table, type

```
dbaccess
```
3. If necessary, log into the appropriate database.
4. Choose *Query-language*, then choose the appropriate database from those listed.
5. Choose `<dataset>_inf.sql`, and run it.
6. Choose `<dataset>_load.sql`, and run it (where `<dataset>` is the name of the dataset being loaded).

Logging In To a DBMS

To verify that you have the proper configuration, start MineSet, and from the Tool Manager's File menu, choose Open New DBMS Table.

1. Verify you have the correct server or change servers if necessary.
2. Click *Change DBMS*.
3. Select demo-Oracle from the DBMS menu.

```
login: demo  
Password: demo
```

4. Click OK.

This loads the table from your selected list. You can then manipulate the table to develop a classifier or visualization as detailed in the *MineSet 3.0 Enterprise Edition User's Guide*.

File Exchange between MineSet and SAS (IRIX only)

This chapter describes the support for file exchanges between MineSet and SAS on IRIX systems. The subjects discussed are:

- [“Converting MineSet Data Files to SAS Data Sets”](#) on page 29
- [“Converting SAS Data Sets Into MineSet Data Files”](#) on page 31

Systems other than IRIX must use third-party solutions for file exchange.

Overview

Exchanging data sets between MineSet and SAS is done through two utilities: *mineset2sas* and *sas2mineset*. To convert a MineSet *.schema* and *.data* file pair into a SAS data set, use *mineset2sas*. To convert a SAS data set into MineSet *.schema* and *.data* files, use *sas2mineset*. Both *mineset2sas* and *sas2mineset* invoke the SAS executable. SAS must be installed on the machine on which these conversion utilities are used.

Converting MineSet Data Files to SAS Data Sets

Use *mineset2sas* to convert MineSet data files into SAS data sets. The syntax for this is

```
mineset2sas MineSet file SAS libref.datafile [options]
```

Options are

- **-svsc** to save the script sent to SAS. The script normally is deleted after use.
- **-names** *namefile* to save trimmed column names in *namefile*. The script normally is deleted after use.

For example:

```
mineset2sas cars sasuser.cars -svsc -names cars.names
```

mineset2sas converts the MineSet *.schema* and *.data* files (in this case, *cars.schema* and *cars.data*) into a SAS data file. Currently, only string and numeric data types are supported. The MineSet *.data* file must be in ASCII format; binary format is not supported. To save MineSet data files in ASCII, deselect Use binary data files in Tool Manager Preferences.

SAS column names (or, in SAS terminology, variable names) can consist of only letters, and underscore characters. The first character in a column name cannot be a number. Furthermore, SAS column names can be up to eight characters long. Since any character string can be a legal MineSet column name, *mineset2sas* maps MineSet column names to legal SAS column names. The rules for this mapping are:

- Any invalid character is replaced with an underscore.
- If the first character is a digit, an underscore is prepended to the column name.
- Column names are truncated to eight characters. If this truncation results in non-unique column names, the ends of the conflicting column names are replaced with sequential numbers, thus creating unique column names.

To preserve as much of the full column names as possible, *mineset2sas* also saves the first 40 characters of each column name as the column label.

-names *namefile* Command Line Option

To get a listing of the column names before and after conversion into SAS format, specify the **-names *namefile*** command line option. When *mineset2sas* executes, it writes out a mapping of the column name changes to the specified file. For example:

```
`date of birth` -> `date_of_`  
`92census` -> `_92censu`  
`# of days to end of quarter` -> `__of_da0`  
`# of days to end of year` -> `__of_da1`  
`# of davenports` -> `__of_da2`
```

-svsc Option

The *mineset2sas* utility reads the schema for the specified data file, and writes a customized SAS script. SAS is invoked with this script to read and convert the data. The script sent to SAS is normally deleted after use. With the **-svsc** option, the script is saved as the file *mineset2sas.sas*. If there is an error in the script processing, the SAS error log is saved as *mineset2sas.log*.

- For Windows users the SAS script must be installed in the directory in which MineSet is installed under *Programs\sas*.
- For IRIX users the SAS script must be installed in */usr/sbin/sas*.

Converting SAS Data Sets Into MineSet Data Files

Use *sas2mineset* to convert SAS data sets into MineSet data files. The syntax for this is `sas2mineset <SAS libref.datafile> <MineSet file> [options]`

Options are

- **-nodata** creates only a *.schema* file, no *.data* file.
- **-svsc** saves the scripts sent to SAS.
- **-nolabel** indicates that you do not want labels used for column names.
- **-names <namefile>** restores long column names from *<namefile>*, created by *mineset2sas*.

For example,

```
sas2mineset sasuser.houses -svsc -names houses.names
```

The *sas2mineset* utility converts a SAS data file into MineSet *.schema* and *.data* files. Currently, this utility supports only string, numeric, and date data types.

The **-nolabel** Option

SAS only supports eight-character long column names, but allows optional 40 character labels for each column. MineSet sets no limit on the column name length, so, by default, *sas2mineset* uses the column labels to name the columns in the output file, if labels have been defined. To force *sas2mineset* to use the SAS column name for each column, even if a label is specified, add the **-nolabel** option to the command line.

The **-names *namefile*** Option

If a MineSet data file is converted into SAS with *mineset2sas* and then back to MineSet format with *sas2mineset*, a column name map file can be created to keep track of the original column names. To have *sas2mineset* use a name map file created by *mineset2sas*, add the **-names *<namefile>*** option to the command line, and specify the same map file as specified when the file was converted into SAS format with *mineset2sas*. This option is useful only for data files with column names longer than 40 characters, since *mineset2sas* saves up to 40 characters in the column label.

Note that the **-name *<namefile>*** option overrides the **-nolabel** option.

The **-nodata** Option

To create just a MineSet schema file without downloading the data from a SAS data file, add the **-nodata** option to the command line.

The **-svsc** Option

The *sas2mineset* utility writes two customized SAS scripts to retrieve the specified data file. The first script extracts column descriptions; the second extracts the data. The scripts are normally deleted after use. With the **-svsc** option, the scripts are saved as *getschema.sas* and *getdata.sas*, respectively. If there is an error in the script processing, the SAS error logs are saved as *getschema.log* and *getdata.log*, respectively.

MineSet Web Extensions

This chapter describes the MineSet extensions that are provided to let you create or view visualizations or interact with MineSet over the web. The following subjects are discussed:

- [“MineSet Web Extension Files” on page 34](#)
- [“MineSet Web Installation \(Client\)” on page 35](#)
- [“MineSet mtr Files” on page 35](#)
- [“Publishing on the Web” on page 36](#)

Overview

MineSet Web extension allows visualizing files and data generated by MineSet software over the Web. This can be done using the MineSet *mtr* extension. MineSet *mtr* extension lets you place MineSet visualization, schema and data files into an archive file, which can be embedded in a web page as an HTML tag.

Once the user clicks on the hyperlink in Internet Explorer, the browser automatically invokes the MineSet ActiveX control which brings up the visual tool within the browser’s window. Any *viz* or *mtr* file embedded within an HTML document will seamlessly launch the ActiveX control within the rest of the web page. The user can then click on the hyperlink in Internet Explorer, or type the filename in the URL box, and ActiveX control will be launched as well.

The machine that the browser is running on must have the MineSet client software installed.

MineSet Web Extension Files

All MineSet Web Extension files are located in the */MineSet 3.0/examples* subdirectory.

File	Purpose
<i>adult-salary.eviviz.mtr</i>	mtr file of <i>adult-salary.eviviz</i> . Windows users find the file in the directory in which MineSet is installed under <i>Examples\eviviz\examples</i> . IRIX users find the file in <i>/usr/lib/MineSet/eviviz/examples</i> .
<i>nl.births.mapviz.mtr</i>	mtr file of <i>nl.births.mapviz</i> . Windows users find the file in the directory in which MineSet is installed under <i>Examples\mapviz\examples</i> . IRIX users find the file in <i>/usr/lib/MineSet/mapviz/examples</i> .
<i>company.scatterviz.mtr</i>	mtr file of <i>company.scatterviz</i> . Windows users find the file in the directory in which MineSet is installed under <i>Examples\scatterviz\examples</i> . IRIX users find the file in <i>/usr/lib/MineSet/scatterviz/examples/</i>
<i>cars-dt.treeviz.mtr</i>	mtr file of <i>cars-dt.treeviz</i> . Windows users find the file in the directory in which MineSet is installed under <i>Examples\treeviz\examples</i> . IRIX users find the file in <i>/usr/lib/MineSet/treeviz/examples</i> .
<i>cars-odt.treeviz.mtr</i>	mtr file of <i>cars-odt.treeviz</i> . Windows users find the file in the directory in which MineSet is installed under <i>Examples\treeviz\examples</i> . IRIX users find the file in <i>/usr/lib/MineSet/treeviz/examples</i> .
<i>churn-dt.treeviz.mtr</i>	mtr file of <i>churn-dt.treeviz</i> . Windows users can find the file in Windows users find the file in the directory in which MineSet is installed under <i>Examples\treeviz\examples</i> . IRIX users find the file in <i>/usr/lib/MineSet/treeviz/examples/</i>

MineSet Web Installation (Client)

During the MineSet client installation the client part of the web extension is automatically installed. The system must have Internet Explorer 4.0 or greater installed for full Web Support. Netscape will launch the visualizer tool externally, not within its browser window.

MineSet *mtr* Files

MineSet *mtr* files are archives of MineSet files generated by the Viz tool. Once created, it can be used as a hyperlink in an html page. The *mtr* files are very effective in sharing multiple visualizations over the web, eliminating the need for attaching huge files in mails, remote copies, or file transfers (ftp). The *mtr* files can also be launched directly by dragging them into a browser, or setting "File" property in the HTML PARAM tag.

Since *mtr* files are in a compressed format and use the underlying http protocol, the transfer of an *mtr* file is very fast and does not require a cumbersome setup on the part of administrators.

Creating *mtr* Files

An *mtr* file can be created from any MineSet visualizer by selecting File > Publish on the Web. This creates an *mtr* file for the current visualization.

Publishing on the Web

Having created an *mtr* file, the next step is to make it available on the Web. This can be done in the following ways:

1. Creating a hyperlink to the *mtr* file

After the *mtr* file is created, move it to the directory containing all your *.html* files. For Internet Explorer or Netscape to launch an *mtr* file, you can invoke it directly by entering

```
http://yourserver/directory/foo.treeviz.mtr
```

in the Internet Explorer or Netscape Location window; or you can make a link to it from a page by adding the following line in the *.html* file for that page.

```
<A HREF="foo.treeviz.mtr">foo.treeviz.mtr </A>
```

2. Also, in Internet Explorer, you can set the OBJECT tag in the following manner:

- PARAM tag with CLASSID

You can set the ActiveX control's "File" property directly:

```
<OBJECT CLASSID =  
clsid:clsid:911D5D2F-906D-11D2-9A61-00104BD33DDB  
WIDTH = 804 HEIGHT = 627 ID=vizCompositeCtrl>  
<PARAM NAME = "File" VALUE = "foo.scatterviz.mtr">  
</OBJECT>
```

- Alternatively, use MIME Type instead of CLASSID

```
<OBJECT TYPE = "application/x-mineset-tar"  
WIDTH = 804 HEIGHT = 627 ID=vizCompositeCtrl>  
<PARAM NAME = "File" VALUE = "foo.scatterviz.mtr">  
</OBJECT>  
</P>
```

3. The ActiveX control can also be embedded using an EMBED tag.

```
<EMBED SRC="cars.scatterviz">
```

These techniques can also be used for embedding *.viz* files as well as *.mtr* files.

Flat File Support for MineSet

This chapter describes the *.schema* and the *.data* files that are required to define the MineSet flat files. The following subjects are discussed:

- [“The Data File” on page 37](#)
- [“The *.schema* File” on page 41](#)
- [“Exceptions” on page 45](#)

The Tool Manager also generates *.schema* files to include the *.schema* files for Tree Visualizer, Map Visualizer, Scatter Visualizer, and Splat Visualizer.

The Data File

In its simplest form, the data file consists of a list of lines, each containing a set of fields, each separated by one tab. (Other separators are also allowed—see [“Input Options” on page 45](#)—but only one, per file, may separate each field). All lines must contain the same fields. (The interpretation of the fields is specified by the *.schema* file, described in the next section.) For example, the first few lines of retail store data might look like this:

```
Eastern Maryland Baltimore 1816 appliances 72 115 138
Eastern Maryland Baltimore 1816 clothing 355 344 395
Eastern Maryland Baltimore 1816 electronics 156 182 209
Eastern Maryland Baltimore 1816 furniture 78 75 82
Eastern Massachusetts Boston 1331 appliances 48 68 81
Eastern Massachusetts Boston 1331 clothing 307 258 296
Eastern Massachusetts Boston 1331 electronics 38 183 210
Eastern Massachusetts Boston 1331 furniture 52 69 75
Eastern Massachusetts Boston 1220 appliances 37 63 75
Eastern Massachusetts Boston 1220 clothing 233 240 276
Eastern Massachusetts Boston 1220 electronics 175 208 239
Eastern Massachusetts Boston 1220 furniture 35 53 58
```

In this example, the first five columns are strings: region, state, city, store ID, and product. These are followed by three numbers, representing current sales, last year's sales, and the sales target.

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; doing so generates blank fields. It is possible to use other characters, such as a colon (:), as a separator. In this case, the first line appears as:

```
Eastern:Maryland:Baltimore:1816:appliances:72:115:138
```

The order of the columns must match the format of the *.schema* file. For some visual tools, the order of the rows can affect the layout of the final graphic. See the tool-specific chapters for details.

Any field in the data can also be a "?", indicating that the data is null (unknown). See [Chapter 12, "Nulls in MineSet."](#)

Note: MineSet also supports a binary format, which currently is not documented.

Data Types

MineSet supports integer, floating-point number, dates, and string data types, as well as arrays of these types. The following data types are supported:

- **int** represents a 32-bit signed integer.
- **float** represents a single-precision floating point number. The decimal point is optional. Numbers in exponential "e" notation are also accepted.
- **double** represents a double-precision floating point number. The decimal point is optional when representing a floating point number. Numbers in exponential "e" notation are also accepted. The superior precision of **double** can be useful for accurately representing large numbers, since **float** can represent only seven or eight significant digits accurately. This superior accuracy, however, consumes twice the memory space of **float**.

- **dataString** represents a string that is unlikely to appear multiple times. If it appears multiple times, several copies are made. A **dataString** can be used to store a memory address. Addresses are unlikely to be compared, and each record can have a different address.
- **string** represents a string of characters that can appear multiple times in the data file. Unlike a **dataString**, only a single copy of a given string is stored in memory, no matter how many times it appears in the data. This saves memory for strings appearing many times.

Comparing **strings** is also much quicker than comparing **dataStrings**. Reading in **strings** can be slower than reading in **dataStrings** because it is necessary to look for duplications. An example of **string** use is a division name that appears once for each department in the division. If you are unsure whether to use a **string** or a **dataString**, use a **string**.

- **fixed string** represents a string of fixed length. Like a **dataString**, if a **fixed string** appears multiple times, multiple copies are made. In general, **fixed strings** are used internally for representations of data from data bases, and are generally better to use than **strings** or **dataStrings**.
- **date** represents a date and time. In the data file, a **date** appears with the format MM/DD/YYYY HH:MM:SS. Output from MineSet always represents dates with four-digit years, although two-digit years are acceptable for input. MineSet follows the X-OPEN standard for interpreting two-digit years. Fields with values 69 or greater are considered to be from the 20th century (1969-1999), and values from 0 to 68 are considered to be from the 21st century (2000 - 2068).

Arrays

In MineSet, you can use one-dimensional or two-dimensional arrays of fixed or variable size.

In a fixed-sized array, all entries of the given type have the same number of values. For example, the budgets of the 50 United States, can be represented by a separate float column for each state, or by a single array with 50 floats.

A special form of a fixed array is an “enumerated array.” Like the normal fixed array, there are a fixed number of values in the array; however, the values are associated with an enumeration. For each value in the enumeration, there is a single entry in the array. For example, if there is an enumeration representing the 50 states, an enumerated array based on this enumeration has 50 values.

A variant of the “enumerated array” is the “null enumerated array,” which has an additional entry at the beginning for null (represented as a “?”). For example, with the enumeration of the 50 states, the null enumerated array has 51 values, one for NULL, and the remaining 50 for the 50 states. The null array element could be used for entries where the state is unknown.

The tree visualizer also supports variable length arrays (see [Chapter 6, “Creating Data and Configuration Files for the Tree Visualizer,”](#) for details).

As with the columns, arrays are represented as values separated by tabs or other separators. For a fixed-sized array, the same separator can be used for columns and for individual array elements (in which case, array elements are not visually distinguished from separate columns). You can also define a different separator. In the sales example (in [“The Data File” on page 37](#)), for example, you can treat the location as a four-element array, rather than as four columns. It then could be represented like this:

```
Eastern:Maryland:Baltimore:1816    appliances    72    115    138
```

Here, the array is separated by colons, and the columns are separated by tabs. (For clarity, the rest of this document uses tabs to separate columns, and colons to separate array elements.)

For a variable-length array, you must use different separators for the array and for the columns; otherwise, it is impossible to determine where the variable-length array ends and the other columns begin.

Null Values

Any field or array element in the data file can also have the value “?” (question mark), indicating an unknown or null value (see the discussion of nulls in [Chapter 12, “Nulls in MineSet.”](#))

The .schema File

The schema file consists of an input section, which defines the name and format of the file. (The .schema files generated by the Tool Manager can also contain a history section, which is a copy of the .mineset file. This section is used by drill-through and would normally not be present in manually generated .schema files.)

A typical input section might look like this:

```
input {
  file "store";

  string region;
  string state;
  string city;
  string storeId;
  string product;
  float sales;
  float lastYear;
  float target;

  options separator ':';
}
```

This example states that the input file is called *store*, and that there are eight fields: five of type **string**, three of type **float**.

Variable Names

A variable name can appear in two formats:

- In the first format, it is a letter followed by a number of letters, digits, or underscores. It cannot be a keyword, and should not be quoted.
- In the alternate form, the variable name should be surrounded by back quotes (`). In this form, the variable name can match a keyword, and can contain even non-alphanumeric characters. The primary purpose of this second form is for .schema files generated automatically by the Tool Manager.

There is no scoping of variable names; a given variable name can only be declared once in the .schema file.

Strings and Characters

Strings and characters in the *.schema* file follow C-language conventions. Strings are in double quotation marks ("), and characters are in single quotation marks ('). All standard backslash conventions are followed (for example, `\n` represents a new line).

Comments

Comments begin with a pound (#) symbol at the beginning of a line; anything after this symbol to the end of the line is ignored, up to the end of the line.

File Statements

The file statement names the data file to be read. This statement is required. Its syntax is:

```
file "filename";
```

Filename must be in double quote marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current *.schema* file. If it is not found in the current *.schema* file's directory, the file is sought in the current directory.

Data Statements

The data statements declare the columns in the data file. The columns must be declared in the order they appear in the data file. The format of most data statements is:

```
type name;
```

where type is **int**, **float**, **double string**, **dataString**, **date**, and **fixedString(*n*)**, where *n* is an integer representing the width of the string; *name* is the variable name. Unlike in C, only one variable can be declared per statement.

Other supported types include enumerations, fixed arrays, and enumerated arrays. These data types must be declared inside the 'input' section, before the declaration of the specific column.

Enumerations

The syntax for declaring an enumeration is:

```
enum type name { value, value...};
```

For example:

```
enum string state {  
    "Alabama",  
    "Alaska",  
    ...  
    "Wyoming"  
};
```

The word “string” indicates that the enumeration maps integers to strings; they can also be mapped to other types.

Once an enumeration is declared, a column can be declared to be of that enumeration using the following syntax:

```
enum enumname columnname;
```

For example:

```
enum state st;
```

declares `st` to be a variable of state enumeration. The input file corresponding to this column must contain values from 0-49 (or “?” representing null); however, the output shows the state name.

Enumerations also can be used to declare enumerated arrays (see [“Enumerated Arrays”](#)).

Fixed Arrays

Arrays are also declared using data declarations. The simplest form is the fixed array. The declaration syntax is

```
type name [ number ] ;
```

For example:

```
float revenue [50];
```

You can also override the separator by declaring it as

```
type name [ number ] separator `char`;
```

For example:Monitor

```
float revenue [50] separator `:`;
```

If no separator is specified, the default column separator (usually a tab) is used.

Enumerated Arrays

To declare an enumerated array, first declare the enumeration (see the “[Enumerations](#)” subsection on [page 43](#)). Then declare the array using the following syntax:

```
type name [ enum keyname ];
```

or

```
type name [enum keyname ] separator `char`;
```

For example:

```
float revenue [enum state];
```

As with the normal fixed array, you can also specify a separator. To declare a null enumerated array, use the syntax:

```
type name [ null enum keyname ];
```

or

```
type name [ null enum keyname ] separator `char`;
```

For example:

```
float revenue [null enum state];
```

indicates that the array contains one additional value at the beginning, corresponding to null.

Input Options

The input section of a data file has several options. All options statements begin with the word *options* and have one or more comma-separated options.

- The separator option defines the separator between columns in the data file. The default separator is a tab. The syntax is:

```
options separator 'char';
```

For example:

```
options separator ':';
```

Note: Arrays can override the separator.

- The backslash option controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

```
options backslash off;  
options backslash on;
```

The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than separators. Also, within strings, standard C-style backslash processing is done.

Exceptions

The following exceptions apply to the *.schema* and *.data* files:

- The Tree Visualizer supports only one-dimensional arrays.
- The Tree Visualizer supports variable-length arrays.
- The Map Visualizer and the Scatter Visualizer support a special enum format for dates.
- The Tree Visualizer and the Map Visualizer support the Monitor option.

Note: These exceptions are discussed in detail in the respective tool's chapters.

Creating Data and Configuration Files for the Tree Visualizer

This chapter describes the types and formats of data supported by the Tree Visualizer, how the Tree Visualizer reads in, and graphically displays, the data file. The topics discussed are:

- “Tree Visualizer Data File” on page 47
- “Tree Visualizer Configuration File” on page 49

Both the data and configuration files can be generated automatically by the Tool Manager. Read the Tree Visualizer chapter in the *MineSet 3.0 Enterprise Edition User's Guide*, before using this chapter.

Tree Visualizer Data File

Data input to the Tree Visualizer must be provided as a single file containing raw data, usually in ASCII text form. An example of this data file is described in [Chapter 5, “Flat File Support for MineSet”](#). Special conditions for the Tree Visualizer are noted here.

The only restrictions on data files for treeviz are in the use of arrays. With the Tree Visualizer, you can use one-dimensional arrays of fixed or variable size.

With the Tree Visualizer, variable length arrays are particularly useful for describing the tree organization. Often this can represent organizations in which different parts have different depths. For example, one department could be represented by Gomez:Shapiro:Lacy (three entries), while another is Gomez:Wong:McMartin:Singe (four entries).

A variable-length entry with zero values can also be declared by passing an empty string. This can be used to specify data for the top level of a hierarchy.

When representing an organization with variable-length arrays, be careful. The Tree Visualizer computes the height for each level of the hierarchy separately, giving the highest bar on each level a user-specified height and normalizing the other bars accordingly. For example: Imagine a U.S.-based organization with a domestic and an international sales force. Domestic sales are divided up into states, which are further divided into cities. International sales are divided into continents, which are then divided into countries and cities. You can have locations such as `domestic:California:Mountain View`, and `international:Europe:Italy:Rome`. When displaying organizational hierarchies of this type, it is best to normalize heights at each level. If this is not done, small parts of the organization (for example, Mountain View) would be dwarfed by large parts of the organization (for example, domestic).

When the system tries to match up the levels, the normalization process might introduce anomalies. Usually, this is not the case at the highest level (domestic is matched with international); however, at lower levels this correspondence is no longer valid. Domestic cities (for example, Mountain View) are at the third level, but the third level for international is a country (for example, Italy). Comparing domestic cities against foreign countries usually has little validity. In this case, it is recommended that you introduce artificial levels to balance the hierarchies (for example, `domestic:USA:California:Mountain View`), thus matching cities.

Variable-length arrays might also be useful when some of the regions being compared are subdivided further than others. For example, an organization might have `USA:California:San Francisco` and `USA:California:Los Angeles`, but only `USA:Wyoming`. There is no need to construct an artificial third level just to keep the arrays balanced, as long as each level in the array matches the same level in other arrays.

Starting up the Tree Visualizer takes longer when variable-length arrays are read in than when fixed-length arrays or individual columns are read in. Unless the data is variable length, it is best not to use variable-length arrays.

The syntax for representing variable length arrays is described in [“Arrays” in Chapter 5](#).

Tree Visualizer Configuration File

The configuration file format is flexible. Words in it must be separated by spaces, and it is case-sensitive. Except for the include statement and text within quoted strings, spacing and line breaks are irrelevant.

Comments begin with a pound (#) symbol at the beginning of a line; anything after this symbol to the end of the line is ignored.

Sections

The configuration file consists of a series of sections, each of which has this syntax:

```
sectionKeyword
{
    statements...
}
```

where *sectionKeyword* names the section. A semicolon (;) can follow the closing brace (}) but is not required. The order of the sections is significant, since sections can refer to variables defined in previous sections. The sections found in a treeviz configuration file are “[Input Section](#)”, “[Expression Section](#)”, “[Hierarchy Section](#)”, and “[View Section](#)”.

Options Files

As each section is encountered, a special configuration file (referred to as an “options file”) is also read in. Options files have names in the form:

```
sectionName.treeviz.options
```

Options files normally contain options statements. These files are read in the following order:

1. The directory containing system defaults.
 - For Windows: from the directory in which MineSet is installed `\config\treeviz`.
 - For IRIX: `/usr/lib/MineSet/treeviz`.
2. Your home directory, where you set up personal defaults.
3. The current directory. This lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section in the configuration file is read after all the options files have been read in; thus, options in the configuration file override those in the options files.

Statements

A statement has the following syntax:

```
statementKeyword info ;
```

where *statementKeyword* defines the statement, and *info* varies according to the keyword. A statement can be another section (using the brace format defined under “[Sections](#)”).

Variable Names

A variable name can appear in two formats:

- In the first format, it is a letter followed by a number of letters, digits, or underscores. It cannot be a keyword (defined later in this appendix), and should not be placed in quotation marks.
- In the alternate form, the variable name should be surrounded by back quotes (`). In this form, the variable name can match a keyword, and can contain even non-alphanumeric characters. The primary purpose of this second form is for configuration files generated automatically by the Tool Manager.

There is no scoping of variable names; a given variable name can be declared only once in the configuration file.

Option Statements

Many sections have options statements, which have this syntax:

```
options key info, key info... ;
```

where *key* defines the specific option, and *info* depends on the key. In some cases, the *key* can be more than one word. To maximize the number of allowable variable names, most option keys are meaningful only within the appropriate option statement; keys do not conflict with variable names. You can declare several options on the same line, separating them by commas or placing them in several options statements. For example, the following two examples are equivalent:

```
options home angle 30, shrinkage 10.0;
```

and

```
options home angle 30;  
options shrinkage 10.0;
```

If two conflicting values for the same option appear, the last value is taken.

Include Statements

The configuration file can contain lines of the form:

```
include "filename"
```

These lines can appear anywhere in the configuration file, but each must be on its own line. The filename must be in quotation marks; anything after the closing quotation mark is ignored. Include statements can be nested. If a relative pathname (one not beginning with a slash) is specified, the file is first sought relative to the directory containing the current configuration file. (If include statements are present, this might not be the same as the initially loaded configuration file.) If it is not found in the directory containing the current configuration file, the include file is sought in the current directory. If the file is not found, an error message appears.

Sinclude Statements

A statement similar to an include is `sinclude`, which has the syntax:

```
sinclude "filename"
```

This is identical to the include statement, except that no error appears if the file does not exist; instead, the sinclude statement is ignored.

Strings and Characters

Strings and characters in the configuration file follow C language conventions. Strings are in double quotes ("), and characters are in single quotation marks ('). All standard backslash conventions are followed (for example, \n represents a new line).

Keywords

The currently recognized keywords are listed in [Table 6-1](#). Variables cannot have these names unless they are surrounded by back quotes (`). Tokens appearing only in option statements are not keywords, and can be used for variable names.

Table 6-1 Keywords for the Tree Visualizer

aggregate	disk	int	normalize
any	divide	isSummary	off
ascending	double	key	on
average	enum	label	options
back	execute	landscape	scale
base	expressions	legend	separator
buckets	file	levels	sort
color	filter	max	string
colors	float	message	sum
count	height	min	view
dataString	hierarchy	modulus	
descending	input	none	

Expressions

The expression language used in the Filter and Add Column panels is similar to expressions in C, C++, and Java. The basic operators are the same:

+	addition
-	subtraction
*	multiplication
/	division
()	parentheses for grouping expressions
%	modulo (remainder after division)
!	logical NOT
~	logical NOT
&&	logical AND
	logical OR
^	logical exclusive OR
==	equal to
!=	not equal to
<=	less than or equal to
<	less than
>=	greater than or equal to
>	greater than
&	bitwise AND
	bitwise OR

The expression language also provides the following:

isNull()	determines if the value in parentheses is null
if () then () else ()	standard if/then/else
() ? () : ()	C syntax if/then/else
divide(x, y, z)	divide x by y, and give value z if y is 0

Also, the following functions are available:

- **modulus**(x, y, z) is similar to **divide**, but for modulus.
- **hierarchy**(string) is valid only within a hierarchy. It produces a string describing the components of the hierarchy, separated by *string*. For example:

```
hierarchy(" : " )
```

might produce

```
Western:California:Mountain View
```

The **hierarchy** function is most useful in the **execute** statement, passing the hierarchy information to the command being executed.

- **isSummary**() returns 1 if the expression is being applied to base information; otherwise, it returns zero. Often, this is useful with the **?:** operator, particularly in **message** and **execute** statements.

Type handling is similar to that in C. Expressions using **int** and **float** promote both sides to float. Expressions using **int** and **double**, or **float** and **double** promote both sides to double. The result of a relational expression (for example, **==**, **<**) is always an **int**. Type casting is also supported.

Unlike in C, strings can be compared using relational expressions; the strings are compared lexicographically.

Input Section

The first section of a configuration file is normally the input section. It defines the name and format of the file. A typical input section might look like this:

```
input {
    file "store";

    string region;
    string state;
    string city;
    string storeId;
    string product;
    float sales;
    float lastYear;
    float target;

    options separator ':';
}
```

This example states that the input file is called *store*, and that there are eight fields: five of type **string**, three of type **float**.

When the input section is entered, the options file, *input.treeviz.options*, is read in.

The syntax for the input section is the of *.schema* files and is described in [Chapter 5, “Flat File Support for MineSet”](#). For treeviz configuration files generated by the MineSet Tool Manager, the *.schema* file is used for the input section using an “include” statement.

Input Options

Treewiz does support one option not found in the standard input section of a *.schema* file. All options statements begin with the word *options* and have one or more comma-separated options.

- The monitor option allows a dynamic update of the data displayed. When the specified file is changed (for example, through the *touch* command), the data file (not the configuration file) is reread. The data file should not be used to trigger the updates. This prevents the data file being read at the same time it is being updated. The syntax of the monitor option is

```
options monitor "filename";  
options monitor "filename" timeout;
```

where *filename* is the file to watch, and the optional *timeout* specifies the number of seconds to wait after the file changes. If the user interacts with the application in any way during this timeout (via the mouse or keyboard), the timeout restarts. Updating the file can take a few seconds. By specifying a timeout, the chances of an update occurring while the user is interacting with the tool are minimized (but the update is delayed). If no timeout is specified, the update occurs immediately.

The file being monitored must exist at the start of the program. When this file is being updated, it must not be removed and re-created; instead, only its modify time should be updated (for example, through the *touch* command). If the file is deleted, subsequent updates are not shown.

Suppose a program extractor extracts data from a database into a data file. If you want the program to update the data file every 10 minutes, the script might look like this:

```
extractor > dataFile;           # create first data file  
touch trigger;                 # create the trigger file  
while (sleep 600)              # sleep 10 minutes  
do  
    extractor > dataFile;       # create new data file  
    touch trigger;             # force a reread  
done &                          # this loop goes in the  
                                # background  
treeviz configFile;           # run treeviz  
kill $!                         # when treeviz exits, kill the  
                                # update loop
```

The monitor option can be used only if the file alteration monitor */usr/etc/fam* is installed (this can be found in the subsystem *desktop_eoe.sw.fam*).

The input section of a configuration file might look like this:

```
input
{
  file "dataFile"
  #data declarations here
  options monitor "trigger" 15;
}
```

Expression Section

The expression section of a data file lets you define additional columns that are expressions of existing columns. For example, one column can be defined as the sum of two other columns. The expressions are calculated before the definition of the hierarchy. In many cases, it is more appropriate to apply the expressions after creating the hierarchy; the expressions then should be defined within the hierarchy section (described later), and the expressions section can be omitted.

The following is a sample expression section. This section assumes two existing columns of type **float**, “male” and “female”; these represent spending by males and females on various goods. Two columns are added: “total” represents the total dollars spent, and “pctFemale” represents the percentage of dollars spent by females.

```
expressions
{
float total = male+female;
float pctFemale = divide (female*100, total, 50.0);
}
```

Note: The pctFemale calculation uses “total,” defined in the previous statement. Also, note the use of the divide function rather than the / operator. This results in 50% for the case where there are no dollars spent at all; using the / operator generates a divide by zero error in such a case.

The format of the expressions section is:

```
expressions
{
    expressionDeclaration;
    ...
}
```

where *expressionDeclaration* has the following syntax:

```
type name = expression ;
```

Since the expressions section has no options, no options file is read in for it.

Hierarchy Section

The hierarchy section of a data file describes how the previously read table is converted into a hierarchy. Here is a sample hierarchy section:

```
hierarchy
{
    levels region, state, city, storeId;
    key product;
    aggregate
    {
        sum sales;
        sum lastYear;
        sum target;
    }
    expressions
    {
        float pctLastYear = divide(sales*100, lastYear, 100.0);
        float pctTarget = divide(sales*100, target, 100.0);
    }
}
```

The parts of the hierarchy section are described below.

When entering the hierarchy section, the *hierarchy.treeviz.options* options file is read in.

Levels Statements

The levels statement defines how the table is converted into a hierarchy. The format is:

```
levels name, name...;
```

where *name* represents a column previously defined in the input or the expressions section. How the hierarchy is created depends on the types of the columns defined.

If the columns represent simple types (for example, strings or numbers), each column is converted into a single level of the hierarchy. The top level of the hierarchy is a single, all-inclusive node. The next level contains one node for each unique value in the first column. The third level contains one node for each unique value in the second column, and so on. Hierarchies created in this way are always balanced: All branches in the hierarchy go to the same depth (namely one greater than the number of columns specified in the levels statement).

In the case where the column is an array, there can be only a single column specified in the levels statement. Each value in the array is mapped to one level in the hierarchy. The top level is a single node representing the total aggregation. The next level contains one node for each unique value of the first value in the array; the third level contains one node for each unique value of the first two values of the array, and so on.

If the array is of fixed type, this hierarchy is balanced. If a variable array is used, the hierarchy is not necessarily balanced (some branches can go deeper than others).

A variable-length array can be used to specify the hierarchy, even if the hierarchy is balanced to a fixed depth. When using columns or fixed arrays to specify the levels, you can specify data associated only with those levels at the bottom (or leaf) nodes. In this case, all higher nodes in the hierarchy must be aggregated. However, rather than relying on automatic aggregation, you might want to supply your own data for each level of the hierarchy (if, for example, the calculation can not be done automatically by the Tree Visualizer). In that case, use variable-length arrays to specify levels and provide separate data for each level.

For example, the data file might contain lines such as:

```
Domestic:Western 43
Domestic:Eastern 57
Domestic      85
Intl:Europe   52
Intl:Asia    39
Intl         94
           133
```

Note: The last line has an empty value for the location; the number 133 is translated to the top of the hierarchy.

Key Statements

The key statement specifies those keys that are used to select the bars at each node in the hierarchy. The key corresponds to the bars displayed in the final view. The syntax of the key statement is:

```
key name [sort [ascending|descending]];
```

where *name* is the name of one of the previously defined columns. It cannot be the name of a column used in the levels statement. Only a single key statement can be made.

By default, the bars generated by the key statement appear in the order first encountered. If the key is an enumerated array, the bars appear in the order of the enumeration; otherwise they appear in the order in which values are first encountered in the data file. Adding the word `sort` at the end of the key statement sorts the bars. Sorting depends on the type: Strings are sorted alphabetically, and numbers are sorted numerically. Enumerations are sorted on the index of the enumeration, not the string that the enumeration refers to. If, however, the key is an enumerated array, the sorting takes place according to the enumeration string (to sort based on the enumeration index, leave it unsorted). Optionally, the word `sort` can be followed by `ascending` or `descending` to specify the sort order; the default is ascending.

If the key column is a simple type (for example, a **string**), the unique values of that key are looked up in the original table. The order of the values is the same as the one in which the key values appear in the original input table. Although it is not required, the same keys are often repeated in the same order. For example, in the following table, the fifth column is the key, and has the values “appliances,” “clothing,” “electronics,” and “furniture.”

Eastern	Maryland	Baltimore	1816	appliances	72	115	138
Eastern	Maryland	Baltimore	1816	clothing	355	344	395
Eastern	Maryland	Baltimore	1816	electronics	156	182	209
Eastern	Maryland	Baltimore	1816	furniture	78	75	82
Eastern	Massachusetts	Boston	1331	appliances	48	68	81
Eastern	Massachusetts	Boston	1331	clothing	307	258	296
Eastern	Massachusetts	Boston	1331	electronics	38	183	210
Eastern	Massachusetts	Boston	1331	furniture	52	69	75
Eastern	Massachusetts	Boston	1220	appliances	37	63	75
Eastern	Massachusetts	Boston	1220	clothing	233	240	276

```
Eastern Massachusetts Boston 1220 electronics 175 208 239
Eastern Massachusetts Boston 1220 furniture 35 53 58
```

The key can also be any column of the enumerated array type. In this case, the enumeration is used as the key for specifying the bars. Other columns in the input can also be enumerated array types, as long as they use the same enumeration. For example, this table can also be input as

```
Eastern Maryland Baltimore 1816
      72:355:156:78 115:344:182:75 138:395:209:82
Eastern Massachusetts Boston 1331
      48:307:38:52 68:258:183:69 81:296:210:75
Eastern Massachusetts Boston 1200
      837:233:175:35 63:240:208:53 75:276:239:58
```

For clarity, each line has been wrapped onto two lines; however, in the file these should be on single lines. The input section for this data appears as

```
input
{
  file "...";
  key string product {
    "appliances", "clothing", "electronics", "furniture"
  }
  string region;
  string state;
  string city;
  string storeId;
  float sales [ enum product ] separator ':' ;
  float lastYear [ enum product ] separator ':' ;
  float target [ enum product ] separator ':' ;
}
```

Note: Since the arrays are fixed, the use of a colon separator for the arrays is not required; however, it might make it easier for a human to read the input.

In this example, the hierarchy section appears as follows:

```
hierarchy
{
  levels region, state, city, storeId;
  key sales;
  ...
}
```

Since *sales* is an enumerated array, it used its key type (product) as the key to generating the bars; thus, each graph in the final view has four bars. Note that *lastYear* and *target* must use the same key type for their array.

Arrays other than enumerated arrays can not be specified as the key.

Aggregate Subsection

The aggregate subsection of the hierarchy section describes how values are aggregated at higher levels of the hierarchy. An example is:

```
aggregate
{
    sum sales;
    sum lastYear;
    sum target;
}
```

This indicates that *sales*, *lastYear*, and *target* are to be summed at higher levels of the hierarchy (each level summing the values in the level below it). In addition to the **sum** aggregation, the aggregations **average**, **min**, **max**, **count**, and **any** are allowed. All are self-explanatory, except for **any**, which indicates that any of the values can be used. This aggregation is used if you expect the same value (for example, a string) to appear everywhere in the hierarchy and if you just want it to populate the entire hierarchy.

A special case is when the key is an enumerated array. Here, the key is normally also aggregated.

In the case where a variable-length array specifies data for all levels of the hierarchy simultaneously (as opposed to merely specifying the data at the leaf nodes), the aggregate section cannot be used.

The two forms an aggregate statement can take are

```
agg name;
name1 = agg name2;
```

In both cases, the aggregate (*agg*) is one of **sum**, **average**, **min**, **max**, **count**, and **any**. The first form was illustrated above; it aggregates a column, and the result is given the same name as the original column being aggregated. The second form aggregates the column *name2*, but gives the result the name *name1*. This second form is useful if the same value is being aggregated multiple times. Since using the first form creates two aggregations with the same name, the second form can be used to differentiate the aggregations.

For example, if you have a column named *expenses* and want to aggregate it to show the maximum and minimum expenses, you can use

```
aggregate
{
    maxExpenses = max expenses;
    minExpenses = min expenses;
}
```

Aggregate Base Subsection

This subsection specifies how values in the base are aggregated. It can be used only if the aggregate subsection is not present. (If the aggregate section is present, the base is aggregated using the aggregations specified in it).

A sample aggregate base subsection is:

```
aggregate base
{
    sum sales;
    sum lastYear;
}
```

An aggregate statement takes the form

```
agg name;
```

where the *aggregate (agg)* is one of **sum**, **average**, **min**, **max**, **count**, and **any**, (similar to the aggregate section). The aggregation is applied to all the bars on that base to give the appropriate value for the base. After the base is aggregated, its values correspond to all of the columns used in specifying the bars. Any column not specified in the aggregate base section has a value of zero. Because the base values correspond to the bar values, the second form of the aggregate statement (using the =), cannot be used in the aggregate base section.

Expressions Subsection

An expressions subsection of the hierarchy section is similar to the expressions section described earlier, except that it is applied after the hierarchy is created and aggregated. The syntax is identical, but it is declared within the hierarchy section, not external to it.

To give an example of the difference between calculating the expressions before and after creating the hierarchy, take the example of male and female dollars spent. Assume you want to calculate the percentage of dollars spent by women. The expressions might be:

```
expressions
{
    float total = male+female;
    float pctFemale = divide (female*100, total, 50.0);
}
```

Assume you calculated these variables before creating the hierarchy. Then, when aggregating the data up the hierarchy, summing the percentages is not useful. Averaging the percentages results in a believable number; however, it averages percentages of large dollars with percentages of small dollars, and produces incorrect results. (To make this clearer, suppose that on one product, males spent \$99, and females spent \$0. On another product, males spent \$0, and females spent \$1. On the first product females spent 0%, and on the second they spent 100%. Averaging these gives 50%, but in reality, females spent only 1% of the dollars spent on the two products combined.)

The base data should be aggregated first, then the expressions should be applied. (In the example, after aggregating, the result is a combined spending of \$99 for males, and \$1 for females; if the percentage is calculated after the aggregation, the correct value of 1% results.)

Sort Statements

By default, the order of the nodes within each level of the hierarchy is based on the order of the data in the input file. However, sometimes it is desirable to sort the hierarchy. The sort statement can appear in one of two forms:

```
sort name [, ascending|descending];
sort key [, ascending|descending];
```

In the first form, one column name (not used in the level statement) is used for sorting. The column can be the result of an aggregation or an expression. In the second form, the value used in the level statement is the one used in laying out the hierarchy.

The hierarchy can be sorted in ascending or descending order. If neither option is specified, the default is descending order if the first form of the sort is used (this places the largest columns on the left); the default is ascending order if the second form is used (this typically sorts alphabetically).

Note that sort statements affect the sorting of only the branches of the hierarchy; they do not affect the bars within each node of the hierarchy.

Hierarchy Options

There are two options in the hierarchy section: *skipMissing* and *organization*. The format for the *skipMissing* option is

```
options skipMissing;
```

If this option is off (the default) and some values of the key are not present for a given hierarchy node, dummy entries are created with values of 0. This guarantees that all graphs in the hierarchy have the same number of bars, and the same layout. If this option is on, no such entries are generated. This results in variable-length tables in the hierarchy, and bars exist only for items in the input. The position of these bars, however, is not meaningful. This option is not useful if the key is an enumerated array (for which all values are supplied).

The *skipMissing* option increases memory usage and should be avoided, if possible.

The format for the *organization* option is

```
options organization same;  
options organization contains;  
options organization unknown;
```

The *organization* option provides hints about the hierarchy organization that allow for more efficient algorithms. This option is most useful if no hierarchy aggregation is done. The *same* value specifies that all nodes in the hierarchy contain entries for the same item (for example, all nodes could contain “appliances,” “clothing,” “electronics,” and “furniture”). The *contains* value indicates that a parent node contains entries for all values that its children contain. For example, if a node contains “appliances,” its parent node must also contain “appliances,” although not all of its child nodes must contain appliances. The *unknown* value means that no assumptions are to be made regarding the contents of individual nodes.

If no organization is specified, the Tree Visualizer determines the organization as follows.

- If there is no aggregate subsection, *unknown* is used.
- If there is an aggregate section, but the *skipMissing* option is provided, *contains* is used; otherwise, *same* is used. Since this is normally correct when an aggregate subsection is provided (unless *skipMissing* is used but nothing is missing), there normally is no need to provide an organization if the aggregate subsection is present.

If the organization specified does not match the data, the results are unspecified. For example, *same* should not be specified, unless all nodes have the same entries.

View Section

The view section of a data file describes how the hierarchy is displayed, including the mapping of heights, colors, labels, and so forth. A sample view section is:

```
view hierarchy landscape
{
  height sales, normalize levels, max 2.0;
  height legend label "Height: Total sales";
  base height max 1.0;
  disk height target, legend label "Disk height: Target
  sales";
  color pctTarget, scale 0 100 200 500;
  color colors "red" "gray" "green" "blue";
  color legend label "Color: % of target" "0%" "100%"
  "200%" "500%";
  options columns 4;
  message "$%,.2f, %.0f%% of target, %.0f%% of last year",
  sales, pctTarget, pctLastYear;
}
```

The first words of the view section (before the opening brace) describe the type of view. The only view type supported is **view hierarchy landscape**; thus, these words must introduce the view section.

When entering the view section, the *viewHierarchyLandscape.treeviz.options* options file is read in. Note that there is not a simple *view.treeviz.options* options file, the full name *viewHierarchyLandscape* must be used.

Height Statements

The height statement describes how the columns are mapped to the height of objects. It consists of a series of clauses separated by commas. Alternatively, it can be specified as multiple height statements. Thus: the following three examples are equivalent:

- `height sales, normalize levels, max 2.0;`
- `height sales;`
`height normalize levels;`
`height max 2.0;`
- `height sales, normalize levels;`
`height max 2.0;`

The first clause normally contains the name of a column that is to be mapped to height (“sales,” in the example). The column must be of a number type (**int**, **float**, or **double**); **float** is the most efficient. If no height column is specified, all bars are flat, and the remaining height clauses have no effect.

Normalize Clause

The normalize clause determines the maximum value of the height variable; it normalizes all values relative to that height. Thus, if the maximum value is 30.0, and that bar was given a height of 1.0 (in arbitrary units), a value of 15.0 would be mapped to a value of 0.5.

The syntax of the normalize clause can be

`normalize` This normalizes all values against one another, throughout the hierarchy.

`normalize levels` This performs independent normalization at each level of the hierarchy.

`normalize none` This performs no normalization, and is the default.

The second form is particularly useful in cases where the data is aggregated up the hierarchy. For example, assume the sales data is aggregated up the company. Comparing the sales of the company as a whole to the sales of a single individual has little meaning; in a large company, the heights of the bars for the individuals are so small as to be indistinguishable from zero. It makes more sense to compare sales people to sales people, offices to offices, regions to regions, and so on. Normalizing levels does this.

Regardless of which form of normalization is used, the base (if shown) is always normalized independently of the bars. By default, the same normalization mechanism for the bars is used for the base.

The scale Clause

The scale clause scales the height of all objects; all values are multiplied by the scale. The syntax of the scale clause is:

```
scale float
```

where *float* is a floating point number (the decimal point is optional). For example, to double the heights, specify

```
scale 2
```

The filter Clause

Large datasets can contain many graphics. This results in poor performance. In many cases, the data values are small and of little informative value. The filter clause prefilters the data based on the height variable, so that only the nodes with the highest bars are shown. The syntax of the filter clause is:

```
filter > float%
```

The `>` and `%` characters must be typed literally. For example:

```
filter > 5%
```

This example filters out all charts containing no bars greater than 5% of the maximum bar height, except for those containing descendants in the hierarchy containing such bars. Note that if a chart contains just one bar that meets this criterion, the entire chart is shown.

The filter value can be changed interactively through the filter panel.

The filter clause is permitted only on the height statement.

The legend Clause

The legend clause defines the meaning of the height mappings. Any string can be placed in the height legend. The legend clause has the following syntaxes:

`legend off` This turns off the height legend (this is the default).

`legend on` This turns on the height legend.

`legend label string`
This changes the legend. If legend label is used, legend on is unnecessary.

By default, the legend has the following syntax:

```
height:varname
```

where *varname* is the name of the variable that is mapped to height.

It is possible to declare separate legends for the height, the base height, and the disk height.

Base Height Statements

The base height statement specifies how the height of the base is calculated. The format is similar to the height statement, except that it is preceded by the word “base.” If the base height statement is omitted, the height of the base is calculated using the same values as in the height statement (the same variable, normalization mechanism, max value, and so on). You also can specify only some of the clauses for the base, in which case everything else is the same as the height statement. For example:

```
height sales, normalize levels, max 2.0;  
base height max 1.0;
```

In this case, the base height is based on *sales*, and it is normalized by *levels*. The maximum height, however, is only 1.0 instead of 2.0. Usually, the visual effect is better if the base height max is less than the max for the bars.

The filter clause is not permitted on the base height statement.

The on and off Clauses

The initial value of the base height can be turned on and off via the on and off clauses. To turn it off, use

```
base height off
```

To turn it on, use the default:

```
base height on
```

The base height can be changed interactively using the Base Height option in the Display menu. The on and off clauses are valid only with base height. Do not use them with the height or disk height statements.

Disk Height Statements

You can place a disk on each bar to indicate an additional item of data. This is done with the disk height statement. The disk height statement’s syntax is similar to that of the height statement, but it is preceded by the word “disk.” For a disk to be displayed, there must be a clause specifying the column to be mapped to the disk. Other clauses are optional; if these are omitted, the height statement’s defaults are used.

If the height statement has a normalize clause, and the disk height statement has no normalize or max clause, then the disks are normalized with the bars (they are drawn to the same scale). If the disk height statement has either a normalize clause or a max clause, the disks are normalized independently of the bars. For example:

```
height sales, normalize levels, max 2.0;
disk height target;
```

In this case, the bars are mapped to the variable “sales,” and the disks are mapped to “target.” Both are normalized, with the maximum value of sales or target on each level mapped to a value of 2.0. If instead this example is written as

```
height sales, normalize levels, max 2.0;
disk height target, normalize levels;
```

the bars are mapped so the highest bar at each level is 2.0, and the highest disk on each level is 2.0, but the bars and disks are not mapped to the same scale. This can be used, for example, if the bars represent dollars and the disks represent head count.

The filter clause is not permitted on the disk height statement.

Color Statements

The color statement describes how values are mapped to colors. The format is similar to that of the height statement, consisting of several clauses that can be separated by commas or entered as multiple statements.

Color Naming

Color names follow the conventions of the X Window System™, except that the names must be in quotation marks. Examples of valid colors are “green,” “Hot Pink,” and “#77ff42.” The last one is in the form “#rrggbb”, in which the red, green, and blue components of the color are specified as hexadecimal values. Pure saturation is represented by ff, a lack of color by 00. For example, “#000000” is black, “#ffffff” is white, “#ff0000” is red, and “#00ffff” is cyan. (A list of available colors is found in the file *rgb.txt*.)

- Windows users should only use the “#ff0000” form, as only some of the named colors are supported (for example, *white, black, gray, red, yellow, green, etc.*.)
- IRIX users can find the color list file at */usr/lib/X11/rgb.txt*.

The Color Variable

As with height, you also can specify a single column to be mapped to a color. The column must be a number type. Unlike for height, there is no normalization of colors.

The key Clause

Instead of specifying a variable, the word `key` can be specified. This assigns a different color based on each key, normally for each bar. For example, if the 50 states were the keys, `key` assigns a different color to the bar for each state. Since the base is not keyed, when the `key` clause is used, the base is always gray.

The colors Clause

The `colors` clause specifies the colors to be used. The `colors` clause syntax is:

```
colors "colorname" "colorname"...
```

The format for `colorname` is described under [“Color Naming” on page 71](#). Note that there are no commas between the colors. This is because commas are used to separate clauses in the color statement. A sample `colors` clause is

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the above example, red is color 0, gray is color 1, and blue is color 2.

If there is no `colors` statement, colors are chosen randomly; however, if there is a `colors` statement, at least as many colors must be specified as are to be mapped. If a `key` is used, there must be one color for each key value.

The scale Clause

The `scale` clause allows assignment of values to a continuous range of colors. For example, when displaying a percentage, red can be assigned to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example, 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the `scale` clause is

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The colors statement must contain at least as many colors as are to be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the `pctFemale` column indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The colors statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

The buckets Clause

The buckets clause is similar to the scale clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the buckets clause is

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the scale clause.

If, in the `pctFemale` example, you used the buckets clause instead of the scale clause, the statement would be

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater or equal to 100 are colored red. Values greater than or equal to 50, but less than 100, are gray. All other values are blue.

The legend Clause

The legend clause creates a legend of the colors. By default, a legend is on for the bar colors, and off for base and disk colors, although separate legends are permitted for each. The legend clause syntax can be any of the following:

```
legend off
legend on
legend "string" "string" ...
legend label "string"
legend "string" "string" ... label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. It can be omitted if other legend statements are included. Specifying only **legend on** generates the default legend.

The default legend includes a single label to the left (with the name of the column that is mapped to color), and a list of colored labels on the right (with values obtained from the scale clause, the buckets clause, or from the keys). To override the strings in the colored labels, specify the strings as: `legend "string" "string"`.

To override the label on the left, specify it following the word `label`. To eliminate this label, specify an empty string; that is:

```
legend label ""
```

Base Color Statements

The base color statement controls the color of the base. Its syntax is similar to the color statement, except that it is preceded by the word “base.” If this word is omitted, the base has the same color as the bars. If the base color statement is present, any omitted clauses default to the values of the color statement.

Disk Color Statements

The disk color statement controls the color of the disk. The syntax is similar to the color statement, except that it is preceded by the word “disks.” If the disk color statement is omitted, the disk has the same color as the bars. If the statement is present, any omitted clauses default to the values of the color statement.

Since disks are drawn only if a disk height statement is present, a disk color statement has no effect without a disk height statement.

Label Statements

Label statements specify the labels used when labeling objects in the scene. Normally, these statements can be omitted. By default, each bar is labeled with its key; each base is labeled with its position in the hierarchy. The syntaxes of the label statements are:

```
label name
base label name
line label name
back label name
```

where *name* is the name of the column to be used as the label. The first form is used as the label on the bars. The second form is the label on the bases. The third form labels the lines connecting the bases. The fourth places labels behind the bases. (Note that bases often obscure the back labels, so this form is less useful; however, there might be occasions where it is appropriate.)

Message Statements

The message statement specifies the message displayed when the pointer is moved over an object or when an object is selected. The syntax is similar to that of the C **printf** statement. A sample message statement is

```
message "%s: $%f, %.0f%% of target, %.0f%% of last year",
        product, sales, pctTarget, pctLastYear;
```

This could produce the following message:

```
furniture: $2425.37, 23% of target, 87% of last year
```

The formats must match the type of data being used:

- Strings must use %s.
- Ints must use integer formats (such as %d).
- Floats and doubles must use floating point formats (such as %f).

For details of the **printf** format, see the printf (1) reference (man) page (type `man printf` at the shell prompt).

A special format type has been added to **printf**. If the percent sign is followed by a comma (for example, “%,f”), commas are inserted in the number for clarity. Currently, only the United States convention of d,ddd,ddd.dddd is supported, with the decimal point represented by a period, and commas separating every three places to the left of the decimal point. For example, if the above format were:

```
message "%s: $%,f, %, .0f%% of target, %, .0f%% of last year",  
        product, sales, pctTarget, pctLastYear;
```

it would produce the message:

```
furniture: $2,425.37, 23% of target, 87% of last year
```

The **\$**, *****, **h**, **l**, **ll**, **L**, and **n** **printf** format options are not supported.

All values, including the format string, are expressions. Thus, if you had a `pctFemale` column, but wanted a more gender-neutral message, you could use

```
message pctFemale>50?"%f%% females":"%f%% males",  
        pctFemale>50?pctFemale:100-pctFemale;
```

If `pctFemale` is 70, the message “70% females” is displayed; if `pctFemale` is 30, the message “70% males” is displayed. In this case, you can also achieve the same result with a single format string:

```
message "%f%% %s", pctFemale>50?pctFemale:100-pctFemale,  
        pctFemale>50?"females":"males";
```

By default, the same message is used for the base as for the bars. It is possible to specify a different message by using a base message statement, which has the same syntax.

If no message is specified, a default message containing the names and values of all the columns is used.

The Execute Statement

The execute statement lets you execute a shell command by double-clicking an object. The syntax is similar to that of the *message* command; however, since hierarchy information is not displayed on a separate line, it is useful to include the hierarchy information and to pass the key information as arguments.

Here is a sample execute statement that uses *xconfirm* to show a window with information about the item. (The first line, the string, is broken into multiple lines to fit into a single page. In an actual file, it should be on a single line. Multi-line strings are not supported.)

```
execute "xconfirm -t '%s' -t 'sales of %s' -t '$%,.0f'
        -t 'target $%,.0f (%.0f%% of target)'
        -t 'last year $%,.0f, %.0f%% of last year'>/dev/null",
        hierarchy(" "), isSummary()?"everything":product,
        sales, target, pctTarget, lastYear, pctLastYear;
```

This might produce a dialog with the message

```
Eastern Connecticut Milford
sales of clothing
$348
target $427 (81% of target)
last year $372 (94% of target)
```

Note the use of `hierarchy(" ")` to produce a blank-separated description of the hierarchy. Also note the `isSummary()?"everything":product`; this produces the word “everything” if the base was selected, but otherwise produces the product. An alternative to this is using separate execute and base execute statements.

If there is no execute statement, double-clicking an object has the same effect as single-clicking it.

This sample execute statement will work on Windows as well as UNIX, because a simple *xconfirm* utility is provided with the installation on Windows.

View Options

The view section has many options. Like other options statements, the options can be separated by commas, or they can appear in separate lines.

Sky and Ground Colors

The sky and ground color can be specified using the following syntax:

```
options sky color colorname
options sky color colorname colorname
options ground color colorname
options ground color colorname colorname
```

The syntax for color names is the same as that for color naming.

For both the sky and the ground it is possible to specify either one or two colors. If only one color is specified, the sky or ground is solid. If two colors are specified, the sky or ground is shaded between the colors. For the sky, the first color is for the top of the sky, the second for the bottom. For the ground, the first color is for the far horizon, the second for the near ground.

For example, to have a solid black background, specify:

```
options sky color "black", ground color "black";
```

Bar Layout

By default, bars in each chart are laid out as close to a square as possible. You can override this using either the rows or the columns option:

```
options rows number
options columns number
```

Only one of these can be specified.

Overview

Although the overview can be brought up using the Show menu, it can also be configured to come up automatically at startup. The overview syntax is:

```
options overview on
options overview off
```

The first form causes the overview to be displayed at startup. The second form (the default) turns the overview off. Regardless of the setting, the overview can be invoked from the Show menu.

Shrinkage

Hierarchies normally have a large aspect ratio, having greater width than depth. In their unaltered form, it is impossible to view the entire hierarchy, except from such a far distance that no detail would be visible. To see the hierarchy more clearly, distant objects can be shrunk more than perspective normally dictates. The shrinkage option lets you control the shrinkage for a given graph. The shrinkage option syntax is any of the following:

```
options shrinkage auto
options shrinkage float
options shrinkage off
```

The first form (the default) automatically calculates a shrinkage value. Its results are usually reasonable, but not necessarily optimal in unusual hierarchical layouts. Thus, you might want to explicitly set the shrinkage using the second form. For hierarchies in which some parts are deeper than others, automatic calculation does not work well. The best shrinkage value depends on the graph being displayed, as well as various layout options such as margins. You should experiment with each graph. Start with a value of 10.0, then make adjustments. Smaller values result in a narrower hierarchy and increased distortion. The shrinkage value must be positive; avoid values smaller than 5.0.

Shrinkage can be turned off. This is recommended only for very small hierarchies, as it produces hierarchies with very large aspect ratios.

Root Label

By default, the root node of the hierarchy gets a label based on the name of the configuration file. You can override this by using the **root label** option. The format is

```
options root label string
```

This option also affects the string displayed when an object is selected, as well as the result of the `hierarchy()` function.

Note that the root label option has no effect if the base label statement was used (that statement defines the base label for the root as well as for all other bases).

Font

The font option controls the font used for drawing the labels. The syntax is

```
options font "fontname"
```

where *fontname* can be any font in the directory */usr/lib/DPS/outline/base*.

It also can be the string *default*. This attempts to use Helvetica (if available), or the default Inventor font (if Helvetica is not available). Note that different systems can have different fonts installed.

Base Label Color

The base label color option controls the color of the labels in front of the bases. The syntax is

```
options base label color "color"
```

Bar Label Color

The bar label color option controls the color of the labels in front of the bars. The syntax is

```
options bar label color "color"
```

Line Color

The line color option controls the color of the lines connecting the nodes in the hierarchy. The syntax is

```
options line color "color"
```

Zero

The zero option lets you determine whether bars, disks, and bases of height zero are drawn solid, as an outline, or hidden completely. In the last case, space is left for the object, but it is not drawn. The default value is solid. This option can be changed at run time using the Display menu.

The syntax for the zero option is

```
options zero solid
options zero outline
options zero hidden
```

Null

The null option lets you determine whether bars, disks, and bases of height null (see [Chapter 12, “Nulls in MineSet,”](#)) are drawn solid, outline, or hidden completely. In the last case, space is left for the object, but it is not drawn. The default value is outline. This option can be changed at run time using the Display menu. The syntax is

```
options null solid
options null outline
options null hidden
```

Other Options

There are 10 other options to control the layout of the display, level of detail, and other parameters. Generally, it is not necessary to adjust these parameters. The values of many of the options are in arbitrary units. Adjust the options by increasing or decreasing the value. For the default values of these parameters, see the file *viewHierarchyLandscape*. Windows users find this file in *Program Files\Sgi\MineSet 3.0\config\treeviz*. IRIX users find this file in */usr/lib/MineSet/treeviz*.

- `options speed float`
Controls the speed during free-form (middle-mouse) horizontal navigation (forward, backward, and side to side). The larger the value, the faster the motion.
- `options climb speed float`
Controls the speed when moving up and down using Shift + middle mouse. The larger the value, the faster the motion.
- `options leaf leaf margin float`
Controls the distance between adjacent nodes in the hierarchy. Larger values move the nodes farther away.

- `options root leaf margin float`
Controls the distance between a node and its children. Larger values move the nodes farther away.
- `options leaf edge margin float`
Adds margin space next to nodes at the edge of a subhierarchy.
- `options initial position float float float`
Provides the initial *x*, *y*, and *z* position from which the scene is viewed. A value of 0 0 0 positions the viewer at the root of the hierarchy; since the user is looking forward, the root probably is not visible. Increasing *x*, *y*, and *z* moves the camera to the right, up, and back, respectively. A typical position has a zero *x*, positive *y*, and positive *z*. If unspecified, the initial position depends on the layout of the hierarchy.
- `options initial angle float`
Provides the initial angle, measured in degrees, from which the hierarchy is viewed. The value must be between 0 and 90. A value of 0 looks at the scene horizontally; a value of 90 looks straight down.
- `options bar label size float`
Specifies the size of the labels in front of the bars. Larger values result in larger labels.
- `options base label size float`
Specifies the size of the labels in front of the bases. Larger values result in larger labels.
- `options lod [bar float float] [bar label float [float]]
[base float float] [base label float [float]] [disk float]
[motion float]`
Controls the level of detail. The parameters can appear in any order, be omitted, or placed in multiple **lod** options. These options control the changing form, or disappearance of, objects, thus providing better system performance.

Except for the **motion** parameter, all float values represent the size of the object when the form change or disappearance takes place. The smaller the value specified, the smaller and farther away the object is when the change takes place. Smaller values provide nicer graphics but slower system performance. The numbers of the different parameters cannot be compared directly because the size of the object also determines when the change takes place. A value of 0.0 means no level of detail changes for that parameter. This setting can significantly slow the rendering process.

bar controls when a bar is drawn with less detail. The first value specifies when the object is drawn as a pair of planes; the second value specifies when the object is drawn as a single line.

bar label controls when the labels on the bars disappear. If two values are specified, the first value specifies when the label is drawn in a lower-quality, fast font; the second value controls when it disappears.

base controls when the bases, and the bar charts in front on top of them, disappear. The first number is based on the width of the base; the second on the height of the base plus the tallest bar on it.

base label controls when the label in front of the base disappears. If two values are specified, the first value specifies when the label is drawn in a lower-quality, fast font; the second value controls when it disappears.

initial depth controls the initial depth to which the hierarchy is viewed. When you are at the top of the hierarchy, you see only the number of hierarchical levels specified by the slider. The nodes in the rows are arranged to optimize their visibility. When navigating to nodes lower in the hierarchy, additional rows are made visible automatically. The nodes above them automatically adjust their locations to accommodate the newly added nodes; thus, some nodes might seem to move. Note that the overview shows all nodes in the hierarchy, not just the top nodes, so the layout of the overview might not match the layout of the main view. The X in the overview approximates the corresponding location in the main view; there is no exact mapping between the two layouts.

An initial depth of zero, or one greater than the depth of the hierarchy, shows the entire hierarchy.

Once the Tree Visualizer is running, the depth can be changed through the filter panel.

disk controls when the disk disappears.

motion controls changes in some of the level of detail calculations when the scene is animated. A value greater than 1.0 defaults to 1.0. A value of 1.0 specifies that motion has no effect on the level of detail. Smaller values change the level of detail at a proportional distance. For example, a value of 0.5 means that during animation, level of detail changes occur at half the normal distance.

Creating Data, Configuration, Hierarchy, and GFX Files for the Map Visualizer

This chapter describes the data and configuration files that are required for the Map Visualizer. These files can also be generated automatically using the Tool Manager. The subjects discussed are:

- [“The Data File” on page 85](#)
- [“The Configuration File” on page 88](#)
- [“The Hierarchy File” on page 109](#)
- [“The .gfx File” on page 110](#)

Read the Map Visualizer chapter of the *MineSet 3.0 Enterprise User’s Guide for Windows* before using this chapter.

The Data File

Data input to the Map Visualizer must be provided as a single file containing raw data, usually in ASCII text form. In its simplest form, the data file consists of a list of lines, each containing a set of fields separated by one tab. (Other separators are also allowed, but only one can separate each field. See [“Input options” on page 98](#).) All lines must contain the same fields. The interpretation of the fields is specified by the configuration file, described in [“The Configuration File” on page 88](#). Using the U.S. population data in the *examples* directory, the *population.usa.data* file, which is provided as part of the Map Visualizer package, the first few lines of this input file appear as shown below:

```
AL      0 0 0 1000 9000 127901 309527 590756 771623 964201 996992 1262505
1513401 1828697 2138093 2348174 2646248 2832961 3061743 3266740 3444354
3894025 4040587      51705
AR      0 0 0 0 1000 14000 30000 98000 210000 435000 484000 803000
1128000 1312000 1574000 1752000 1854000 1949000 1910000 1786000 1923000
2286000 2351000      53187
AZ      0 0 0 0 0 0 0 0 0 10000 40000 88000 123000 204000 334000 436000
499000 750000 1302000 1775000 2717000 3665000      114000
```

```
CA    0 0 0 0 0 0 0 0 93000 380000 560000 865000 1213000 1485000
2378000 3427000 5677000 6907000 10586000 15717000 19971000 23668000
29760021    158706
```

In this example, the first column is a two-character string identifying the graphical object—the state. (This string locates a record in a *.gfx* file containing information about the shape of the graphical object.) The tab separator is followed by a grouping of 23 numeric values, which represent the state’s population from 1770 through 1990, in 10-year increments. The next tab separator is followed by a single numeric value, which specifies the state’s area in square miles.

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; this generates blank fields. The order of the columns must match the format specified by the configuration file.

Any field in the data can also be a “?”, indicating that the data is null (unknown). See [Chapter 12, “Nulls in MineSet.”](#)

Data Types

The Map Visualizer supports integer, floating point number, and string data types, as well as arrays of these types. The following data types are supported:

- **int** represents a 32-bit signed integer.
- **float** represents a single-precision floating point number. The decimal point is optional. Numbers in exponential “e” notation are also accepted.
- **double** represents a double-precision floating point number. The decimal point is optional when representing a floating point number. Numbers in exponential “e” notation are also accepted. The superior precision of **double** can be useful for accurately representing large numbers, since **float** can represent only seven or eight significant digits accurately. This superior accuracy, however, consumes twice the memory space of **float**.

- **dataString** represents a string that is unlikely to appear multiple times. If it appears multiple times, multiple copies are made.
- **string** represents a string of characters that can appear multiple times in the data file. Unlike a **dataString**, only a single copy of a given string is stored in memory, no matter how many times it appears in the data. This saves memory for strings appearing many times.

Comparing **strings** is also much quicker than comparing **dataStrings**. Processing is somewhat slower when looking for duplicate strings as they are read in. An example of **string** use is for a division name that appears once for each department in the division. If you are unsure whether to use a **string** or a **dataString**, use a **string**.

- **fixed string** represents a string of fixed length. Like a **dataString**, if a **fixed string** appears multiple times, multiple copies are made. In general, **fixed strings** are used internally for representations of data from data bases, and are generally better to use than **strings** or **dataStrings**.
- **date** represents a date and time. In the data file, **date** must appear in the format MM/DD/YY HH:MM:SS. Output from MineSet always represents dates with four-digit years, although two-digit years are acceptable for input. MineSet follows the X-OPEN standard for interpreting two-digit years. Fields with values 69 or greater are considered to be from the 20th century (1969-1999), and values from 0 to 68 are considered to be from the 21st century (2000 - 2068).

Fixed Arrays

With the Map Visualizer, you can use one- or two-dimensional arrays of fixed size. In a fixed-sized array, all entries of the given type have the same number of values. Arrays contain the data values across one or two independent variables, that is, those dimensions controlled by the sliders.

A variant of the “enumerated array” is the “null enumerated array.” This is a variant of the enumerated array with an additional entry at the beginning for null, which is represented by “?”.

The Configuration File

The configuration file format is flexible. Words in it must be separated by spaces, and it is case-sensitive. Except for the include statement and text within quoted strings, spacing and line breaks are irrelevant.

Overview

The configuration file's structure and grammar are explained in the following sections.

Sections

The configuration file consists of a series of sections, each of which has the following syntax:

```
sectionKeyword
{
    statements...
}
```

where *sectionKeyword* names the section. A semicolon (;) can follow the closing brace (}) but is not required. The order of the sections is significant, since sections can refer to variables defined in previous sections.

Defaults Files

As each section is encountered, a special configuration file (referred to as a *defaults file*) is also read in. The defaults file has the same name as the section. Defaults files contain options statements. These files are searched in the following order:

- Windows searches for all the files in the directory in which MineSet is installed under `\config\mapviz`.
 - IRIX searches for the files as specified by the X-resource `Mapviz*configPath` in the `Mapviz` file, in the directory `X11/app-defaults`, for example in `/usr/lib/MineSet`.
1. The `mapviz` directory. This directory contains system defaults.
 2. The `~/.MineSet` directory (where the tilde, `~`, indicates your home directory). You can set up personal defaults in this directory.
 3. The current directory. This lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section in the configuration file is read after all the defaults files; thus, options in the configuration file override those in the defaults files.

Statements

A statement has the following syntax:

```
statementKeyword info ;
```

where *statement***Keyword** defines the statement, and *info* varies according to the keyword. A statement can be another section (using the brace format defined under “[Sections](#)” on [page 88](#)).

Variable Names

A variable name can appear in two formats:

- In the first format, it is a letter followed by a number of letters, digits, or underscores. It cannot be a keyword, and should not be placed in quotation marks.
- In the alternate form, the variable name should be surrounded by back quotes (`). In this form, the variable name can match a keyword, and can contain even non-alphanumeric characters. The primary purpose of this second form is for configuration files generated automatically by the Tool Manager.

There is no scoping of variable names; a given variable name can be declared only once in the configuration file.

Option Statements

Many sections have options statements, which have the syntax:

```
options key info, key info... ;
```

where *key* defines the specific option, and *info* depends on the key. In some cases, the *key* can be more than one word. To maximize the number of allowable variable names, most option keys are meaningful only within the appropriate option statement; keys do not conflict with variable names. You can declare several options on the same line, separating them by commas or placing them in several options statements. If two conflicting values for the same option appear, the last value is taken.

Include Statements

The configuration file may contain lines of the form

```
include "filename"
```

These lines can appear anywhere in the configuration file, but each must be on its own line. The filename must be in quotes; anything after the closing quote is ignored. The number of nested includes is unlimited. If a relative pathname (one not beginning with a slash) is specified, the file is first sought in the directory containing the current configuration file. If include statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the directory containing the current configuration file, the include file is sought in the current directory. If the file is not found, an error message appears.

Sinclude Statements

A statement similar to an include is `sinclude`, which has the syntax:

```
sinclude "filename"
```

This is identical to the include statement, except that no error is given if the file does not exist; instead, the `sinclude` statement is ignored.

Strings, Characters, and Comments

Strings and characters in the configuration file follow C conventions. Strings are in double quotation marks ("), and characters are in single quotation marks ('). All standard backslash conventions are followed (for example, `\n` represents a new line).

Comments begin with a pound (#) symbol at the beginning of a line; anything after this symbol to the end of the line is ignored.

Keywords

The currently recognized keywords are listed below. Variables can not have these names unless they are surrounded by back quotes (`). Tokens appearing only in option statements are not keywords, and can be used for variable names.

Table 7-1 Keywords for the Map Visualizer

buckets	expressions	level	outlines
color	file	map	scale
colors	float	message	separator
datapoints	from	modulus	slider
dataString	height	monitor	string
date	input	null	summary
divide	int	objects	title
double	key	off	to
enum	label	on	view
execute	legend	options	

Expressions

Expressions are accepted in several places in the input. Expressions follow standard C-language syntax. The operations and their expressions are listed in [Table 7-2](#):

Table 7-2 Operators Used With Expressions

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
==	Equals
!=	Not equals
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
&&	AND
	OR
!	NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
A?B:C	If (A), then B else C

Also, the following functions are available:

- **divide**(*x*, *y*, *z*) divides *x* by *y*, unless *y* is zero. If *y* is zero, the result is *z*; this is equivalent to $y=0 ? z : x/y$.
- **modulus**(*x*, *y*, *z*) is similar to **divide**, but for modulus.

Type handling is similar to that in C. Expressions using **int** and **float** promote both sides to float. Expressions using **int** and **double**, or **float** and **double** promote both sides to double. The result of a relational expression (for example, ==, <) is always an **int**. Type casting is also supported.

Unlike in C, strings can be compared using relational expressions; the strings are compared lexicographically.

The following sections explain the use and syntax of the Map Visualizer configuration file's input, expression, and view geography sections.

The Input Section

The first section of a data file is normally the **input** section. It defines the name and format of the data file. A typical IRIX input section might look like this:

```
input
{
  file
    "/usr/lib/MineSet/mapviz/examples/population.usa.data";
  enum int Year from 1770 to 1990 by 10;
  string states;
  float population[enum Year] separator ' ';
  float sqMiles;
}
```

This example specifies that the input data file is called *population.usa.data*, and that there are three tab-separated (the default) fields as follows:

- one of type **string**
- one a fixed-length vector of type **float**, with each value separated by a space
- one a scalar value of type **float**

When the input section is entered, the defaults file, *input.mapviz.options*, is read in.

- For Windows users this file is in the directory in which MineSet is installed in `\config\mapviz\input.mapviz.options`.
- For IRIX users the file is in `/usr/lib/MineSet/mapviz/input.mapviz.options`.

File Statements

The **file** statement names the data file to be read. This statement is required. Its syntax is

```
file "filename";
```

The file name must be in double quotation marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current configuration file. If include statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the current configuration file's directory, the file is sought in the current directory.

Enum Statements

Enum statements declare enumeration variables that index into array fields. The enum statement has three forms.

- The first form is

```
enum type name from value1 to value2 by increment;
```

This declares an enum with values starting at *value1* and incremented by *increment* until they reach or exceed *value2*. For example, the statement:

```
enum int age from 20 to 70 by 10;
```

declares age as an array dimension with the values 20, 30, 40, 50, 60, and 70.

Type must be a number type (**int**, **float**, or **double**) or **date** (see [“Dates” on page 95](#)).

- The second enum statement form is

```
enum type name from value1 to value2 across numberOfValues;
```

This declares an enum with values ranging from *value1* to *value2*. The *numberOfValues* is an integer specifying the number of values. For example, the statement

```
enum int age from 20 to 70 across 6;
```

declares age as an enum with the values 20, 30, 40, 50, 60, and 70.

Type must be a number type (**int**, **float**, or **double**) or **date** (see “Dates” on page 95).

- The third enum statement explicitly lists the enumeration values. Its form is

```
enum type name { value1, value2, ..., valueN };
```

Type can be any type or date (see “Dates” on page 95).

Dates

The enum statement includes special support for a date type that handles date and time values starting Jan 1, 1753. The date type is valid only within enum statements. A date enum statement can have the following syntaxes:

```
enum date "format" name from "value1" to "value2" across
    numberOfValues;
enum date "format" name { value1, value2, ..., valueN };
enum date "format" name from "value1" to "value2" by
    "increment";
```

The *format* string specifies the format of the values; it is useful for controlling how dates are displayed in the animation control panel. The syntax of the *format* string is similar to the *scanf* function in C. Various units of time are represented by special characters preceded by the percent symbol (%). For example:

```
enum date cq "Calendar Q%Q, %Y" from "Calendar Q1, 1980" to "Calendar
Q3, 1985" by "1 quarter";
```

The “Calendar Q” in the *format* string matches the “Calendar Q” in *value1* and *value2*. The %Q in the *format* string indicates that the next number in *value1* and *value2* is the calendar quarter. The comma and space in the *format* string match the commas and spaces in the values. Finally, the %Y in the *format* string specifies that the year values are next.

[Table 7-3](#) lists the characters that can follow the percent symbol and the units of time they represent.

Table 7-3 Characters That Can Follow the Percent Symbol in the Format String

Character	Time Unit	Precision
Y	year	4
Q	calendar quarter	1
M	month	2
N	month name	>= 3
D	day	2
h	hour	2
m	minute	2
s	second	2

With the exception of N, each character matches an integer of the specified precision. N matches 3 or more characters giving the English name of the month.

The from-to-by form of the enum statement includes an increment value. For dates, the increment is a quoted string containing an integer, an optional space, and one of the special characters in [Table 7-3](#) or one of the symbols **year**, **quarter**, **month**, **day**, **hour**, **minute**, and **second**. The plural forms of these symbols are also accepted. Note that these symbols are not keywords, since they have special meaning only in the increment string. The following are examples of valid increments:

```
"1 year"
"7 days"
"4h"
```

Data Statements

The data statements declare the columns in the data file. The columns must be declared in the order they appear in the data file. The format of most data statements is

```
type name;
```

where *type* is **int**, **float**, **double**, **string**, **dataString**, **date**, and **fixedString(*n*)**, where *n* is an integer representing the width of the string; *name* is the variable name. Unlike in C, only one variable can be declared per statement.

Fixed Arrays

Fixed arrays can also be declared using simple numeric data declarations; however, if you also are going to declare a slider, you must use the enum declaration form. The declaration syntax is

```
type name [ number ] ;
```

For example:

```
float revenue [50];
```

You can also override the separator by declaring it as

```
type name [ number ] separator 'char';
```

For example:

```
float revenue [50] separator ':';
```

If no separator is specified, the default separator (usually a tab) is used.

Fixed arrays can also be two-dimensional, such as

```
enum string products { "bread", "milk", "cheese", "cereal",  
"apples", "lettuce", "juice", "toothpaste", "soap", "eggs" };  
enum year from 1985 to 1994 by 1;  
float prices[enum products][enum year];
```

or

```
float prices[10][20];
```

which might be used for an array of prices for a set of 10 products over a 20-year period.

Using the *prices* array, for example, if you specified in the Tool Manager that data was to be retrieved from the database in "wide" mode (with a bin for null values), the enumerated *products* are declared as:

```
float prices[null enum products][enum year];
```

and the first column contains the prices for unknown products (products not in the enumerated list of ten known products) declared in the *enum string products* statement.

Input options

The input section of a data file has several options. All **options** statements begin with the word "options" and have one or more comma-separated options.

- The separator option defines the separator between columns in the data file. The default separator is a tab. The syntax is

```
options separator 'char';
```

For example:

```
options separator ':';
```

Note: Arrays can override the separator.

- The monitor option allows a dynamic update of the data displayed. When the specified file is changed (for example, through the UNIX *touch* command), the data file (not the configuration file) is reread. Note that although the data file could be used to trigger the updates, it is better to use a different file so that the data file is not read while it is being updated. The syntax of the monitor option is:

```
options monitor "filename";  
options monitor "filename" timeout;
```

where *filename* is the file to watch, and the optional *timeout* specifies the number of seconds to wait after the file changes. If the user interacts with the application in any way during this timeout (via the mouse or keyboard), the timeout restarts. Updating the file can take a few seconds. By specifying a timeout, the chances of an update occurring while the user is interacting with the tool are minimized. This might delay the update. If no timeout is specified, the update occurs immediately.

On Windows, the timeout value has no effect. In other words, monitor always occurs immediately.

The file being monitored must exist at the start of the program. When this file is being updated, it must not be removed and re-created; instead, only its modify time should be updated (for example, through the *touch* command). If the file is deleted, subsequent updates are not shown.

Suppose a program `extractor` extracts data from a database into a data file. If you want the program to update the data file every 10 minutes, the script you write might look like this:

```
extractor > dataFile;           # create first data file
touch trigger;                  # create the trigger file
while (sleep 600)                # sleep 10 minutes
do
extractor > dataFile;           # create new data file
touch trigger;                  # force a reread
done &                          # this loop goes in the
# background
mapviz configFile;              # run mapviz
kill $!                          # when mapviz exits, kill
                                # the update loop
```

The `monitor` option on UNIX systems can be used only if the file alteration monitor `/usr/etc/fam` is installed (this can be found in the subsystem `desktop_eoe.sw.fam`).

The input section of configuration file might look like this:

```
input
{
    file "dataFile:
    #data declarations here
    options monitor "trigger" 15;
}
```

- The `backslash` option controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

```
options backslash off;
options backslash on;
```

The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than separators. Also, within strings standard C-style backslash processing is done.

The Expressions Section

The expressions section of a data file lets you define additional columns that are expressions of existing columns. For example, one column can be defined as the sum of two other columns. The following is a sample expression section. This section assumes two existing fixed-length columns of type **double**: “male” and “female”; these represent spending by males and females on various goods across time (one independent dimension). Two columns are added: “total” represents the total dollars spent, and “pctFemale” represents the percentage of dollars spent by females.

```
expressions
{
double total[enum month] = male+female;
double pctFemale[enum month] = divide(female*100,total,50.0);
}
```

Note: The pctFemale calculation uses “total,” defined in the previous section. Also, note the use of the divide function rather than the / operator. This results in 50% for the case where there are no dollars spent at all; using the / operator generates a divide by zero error in such a case. (The divide function is described in the “Expressions” section.)

The format of the expressions section is

```
expressions
{
    expressionDeclaration;
    ...
}
```

where *expressionDeclaration* has the following syntax:

```
type name = expression ;
```

The format of *expression* has already been described.

Since the expressions section has no options, no defaults file is read in for it.

The View Section

The view section of a data file describes how the graphic objects are displayed, including the mapping of heights, colors, labels, and so forth. A sample view section is

```
view map
{
  map objects "usa.states.hierarchy";
  slider Year;
  height population;
  height legend label "Height: U.S. Population (1770-1990)";
  color density, scale 0 250 500 750 1000;
  color colors "white" "#ffc0c0" "#ff8080" "#ff4040" "red";
  color legend label "Color: Pop. Density" "0/sq-mile"
    "250/sq-mile" "500/sq-mile" "750/sq-mile"
    "1000/sq-mile";
  message "population %,.0f  %,.1f per sq mile",
    population, density;
  execute "xconfirm -t 'Population %,.0f'
    -t 'averaging %,.1f per sq mile'
    -t 'across %,.0f sq-miles' > /dev/null",
    population, density, sqMiles;
}
```

The first words of the view section (before the opening brace) describe the type of view. The only view type supported is **view map**; thus, these words must introduce the view section.

When entering the view section, the *viewMap.mapviz.options* defaults file is read in. Note that there is no simple view defaults file, so the full name *viewMap.mapviz.options* must be used.

Title Statement

The **title** statement inserts a title string at the bottom of the main window. The syntax is

```
title string;
```

where *string* is a string enclosed by double- quotation makes.

Map Statement

The map statement specifies how the graphical objects are to be drawn in the main window. The map statement has three possible syntaxes: one required, the other two optional. The required syntax is

```
map objects hierarchy_filename;
```

where “objects” is a keyword, and *hierarchy_filename* is a filename enclosed in double quotation marks. This statement names the *.hierarchy* file describing the 3D graphical objects that exhibit heights and colors.

The following **map** statements are optional:

- `map outlines hierarchy_filename;`

Declares graphical objects that are drawn as flat lines on which the **map objects** objects are placed. See the samples provided in the *examples* directory, *population.usa.cities.mapviz*.

- `map level column_name;`

Specifies an alternative level of the geographical hierarchy for initial display. For example, in the *examples* directory, the *population.usa.mapviz* file, the unstated default is

```
map level states;
```

and the main window initially displays individual states. If, instead, the configuration file specified

```
map level eastWest;
```

the main window initially displays the United States as two halves: East and West.

Slider Statement

The **slider** statement identifies a key to be used as a slider dimension. Its syntax is

```
slider [enum] enumName;
```

where *enumName* is the name of an enum variable declared in the input section. Note that the **enum** keyword is optional.

There can be 0, 1, or 2 slider statements. The first slider statement applies to the horizontal slider. The second slider statement applies to the vertical slider. If there is no slider statement, the resulting display does not include animation.

No slider statement is required if “height” and “color” map to non-array variables. One slider statement can be included if “height” and “color” map to one-dimensional arrays. Two slider statements can be included if “height” and “color” map to:

- two-dimensional arrays, or
- one-dimensional arrays, where dimensions are enum variable names that one of the sliders controls.

Height Statement

The **height** statement describes how the columns of data are mapped to the height of objects. It consists of a series of clauses separated by commas. The first clause normally contains the name of a column to be mapped to height (“population,” in the example in the section [“The View Section” on page 101](#)). The column must be of a number type (**int**, **float**, or **double**), of which **float** is the most memory-efficient. If the column is a fixed-length array, the **view** section also must contain at least one, and no more than two, **slider** statements.

If no height column is specified, all bars are flat, and the remaining height clauses have no effect.

The **scale** clause lets you scale the height values. Normally, the height variable is mapped directly to the height of the graphical objects, so that the tallest object (with the largest numeric value) rises towards the top of the view window. With the optional scale clause, all values are multiplied by the scale. The scale clause syntax is

```
scale float
```

The **legend** clause defines the meaning of the height mappings. Any string can be placed in the height legend. The legend clause has the following syntaxes:

legend off This turns off the height legend (this is the default).

legend on This turns on the height legend. The legend can be changed by using the legend label form, in which case **legend on** is unnecessary. The legend's default syntax is

```
height:varname
```

where *varname* is the name of the variable that is mapped to height.

legend label *string*

where *string* is the name of the variable that is mapped to height. The legend can be changed by using the legend label form. If **legend label** is used, **legend on** is unnecessary.

Color Statement

The color statement describes how values are mapped to colors. The format is similar to that of the height statement, consisting of several clauses that can be separated by commas or entered as multiple statements.

Color naming follows the conventions of the X Window System, except that the names must be in quotation marks. Examples of valid colors are "green," "Hot Pink," and "#77ff42." The last one is in the form "#rrggbb", in which the red, green, and blue components of the color are specified as hexadecimal values. Pure saturation is represented by ff, a lack of color by 00. For example, "#000000" is black, "#ffffff" is white, "#ff0000" is red, and "#00ffff" is cyan.

- Windows users should only use the "#ff0000" form, as only some of the named colors are supported (for example, *white, black, gray, red, yellow, green, etc.*)
- IRIX users can find the color list file at */usr/lib/X11/rgb.txt*.

The color variable lets you specify a single column to be mapped to a color (as with height). The column must be a number type.

The colors clause specifies the colors to be used. The colors clause's syntax is

```
colors "colorname" "colorname"...
```

The format for *colorname* is described above. Note that there are no commas between the colors. This is because commas are used to separate clauses in the color statement. A sample colors clause is

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the above example, red is color 0, gray is color 1, and blue is color 2.

If there is no colors statement, colors are chosen randomly; however, if there is a colors statement, at least as many colors must be specified as are to be mapped.

The scale clause allows assignment of values to a continuous range of colors. For example, when displaying a percentage, red can be assigned to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the scale clause is

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The colors statement must contain at least as many colors as are to be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the pctFemale column indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The colors statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

The buckets clause is similar to the scale clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the buckets clause is

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the scale clause.

If, in the `pctFemale` example, you used the `buckets` clause instead of the `scale` clause, the statement would be:

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater or equal to 100 are colored red. Values greater than or equal to 50, but less than 100, are gray. All other values are blue.

The **normalize** clause controls a form of color normalization, analogous to height normalization. By default, color normalization is off. The syntax is

```
normalize off;  
normalize on;
```

When color normalization is on, the color scale (or buckets) list of values must range between 0 and 100. These color values then represent relative percentages of the range from the minimum to the maximum for a given viewed scene. For example,

```
color totalSales;legend off  
color scale 0 100, colors "white" "red", normalize on;
```

generates colors in the range of “white” to “red,” where “white” corresponds to the minimum “totalSales” and “red” corresponds to the maximum “totalSales” for the particular set of graphical objects being viewed. See the file *variations.articles.france.mapviz* for a more elaborate example.

- Windows users look in the directory in which MineSet is installed under *Examples\mapviz\examples*.
- IRIX users look in */usr/lib/MineSet/mapviz/examples/variations.articles.france.mapviz*.

The **legend** clause creates a legend of the colors. By default, the color legend is off. The legend clause syntax can be any of the following:

```
legend off  
legend on  
legend "string" "string" ...  
legend label "string"  
legend "string" "string" ... label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. It can be omitted if other legend statements are included. Specifying only **legend on** generates the default legend.

The default legend includes a single label to the left (with the name of the column that is mapped to color), and a list of colored labels on the right (with values obtained from the scale clause, the buckets clause, or from the keys). To override the strings in the colored labels, specify the strings as:

```
legend "string" "string"
```

To override the label on the left, specify it following the word `label`. To eliminate this label, specify an empty string; that is

```
legend ""
```

Message Statement

The **message** statement specifies the message displayed when an object is selected. The syntax is similar to the C **printf** statement. A sample message statement is

```
message "%s: $%f, %.0f%% of target, %.0f%% of last year",
        product, sales, pctTarget, pctLastYear;
```

This could produce the following message:

```
furniture: $2425.37, 23% of target, 87% of last year
```

The formats must match the type of data being used:

- Strings must use `%s`.
- Ints must use integer formats (such as `%d`).
- Floats and doubles must use floating point formats (such as `%f`).

For details of the **printf** format, on UNIX systems see the `printf (1)` man page (type `man printf` at the shell prompt).

A special format type has been added to **printf**. If the percent sign is followed by a comma (for example, `%,f`), commas are inserted in the number for clarity. Currently, only the United States convention of `d,ddd,ddd.dddd` is supported, with the decimal point represented by a period, and commas separating every three places to the left of the decimal point. For example, if the above format were:

```
message "%s: $%,f, %, .0f%% of target, %, .0f%% of last year",
        product, sales, pctTarget, pctLastYear;
```

it would produce the message:

```
furniture: $2,425.37, 23% of target, 87% of last year
```

The `$`, `*`, `h`, `l`, `ll`, `L`, and `n` **printf** format options are not supported.

All values, including the format string, are expressions. Thus, if you had a `pctFemale` column, but wanted a more gender-neutral message, you can use:

```
message pctFemale>50?"%f%% females":"%f%% males",
      pctFemale>50?pctFemale:100-pctFemale;
```

If `pctFemale` is 70, the message “70% females” is displayed; if `pctFemale` is 30, the message “70% males” is displayed. In this case, you can also achieve the same result with a single format string:

```
message "%f%% %s", pctFemale>50?pctFemale:100-pctFemale,
      pctFemale>50?"females":"males";
```

If no message is specified, a default message containing the names and values of all the columns is used.

Execute Statement

The **execute** statement lets you execute a shell command by double-clicking an object. The syntax is similar to that of the *message* command.

Here is a sample UNIX execute statement that uses *xconfirm* to show a window with information about the item. Note that the command line (string) is shown as three lines. In an actual file, this should be on a single line. Multi-line strings are not supported.

```
execute "xconfirm -t '%s' -t 'population %, .0f' -t '%, .0f per
      sq mile' -t '%, .0f sq-miles' > /dev/null", states,
      population, density, sqMiles;
```

This might produce a dialog with the message:

```
CA
64 per sq mile
266,807 sq-miles
```

If there is no execute statement, double-clicking an object has the same effect as single-clicking it.

This sample execute statement will work on Windows as well as UNIX, because a simple *xconfirm* utility is provided with the installation on Windows.

Summary Statement

The **summary** statement specifies the initial setting of the Show Data Points pulldown menu option. The syntax is

```
summary datapoints on;
```

or

```
summary datapoints off;
```

The **summary** statement is optional, and the default setting is *off*.

The Hierarchy File

The hierarchy file defines the object hierarchy, allowing objects to be displayed at different levels of aggregation. It enables the drill up and drill down capabilities of the Map Visualizer. The hierarchy file is specified in the *.mapviz* configuration file with the `map object hierarchy_filename` statement (see [“The View Section” on page 101](#) and [“Map Statement” on page 102](#)).

Here are the first few lines of the *usa.states.hierarchy* file:

```
states      regions      eastWest      USA
usa.states.gfx      usa.states.gfx
      usa.states.gfx      usa.states.gfx
AL      E_S_CENTRAL      USA_E      USA_ALL
AR      W_S_CENTRAL      USA_W      USA_ALL
AZ      MOUNTAIN      USA_W      USA_ALL
CA      PACIFIC      USA_W      USA_ALL
CO      MOUNTAIN      USA_W      USA_ALL
CT      NEW_ENGLAND      USA_E      USA_ALL
DE      MID_ATLANTIC      USA_E      USA_ALL
```

This defines how states combine into regions, sectors, and into a single object encompassing all states.

The first record is a list of column names of the hierarchy; each name must be separated by a single tab ('\t') character. One of the column names must match a type **string** column in the **data file**, as declared in the configuration file's **input** section in [“The Input Section” on page 93](#). In this example, the first column name, *states*, is also the name of a data column in the example *population.usa.mapviz*. The number of column names in this record must be the same as the number of columns of hierarchy data, beginning at the third record of the *.hierarchy* file. If there is only one column name (for example, *gfx_files/canada.provinces.hierarchy*), then there are only two records in the *.hierarchy* file.

The second record is a list of *.gfx* file pathnames, where each pathname is separated by a single tab ('\t') character. Each column name in the first record must have a matching *.gfx* file pathname.

If there is a single column name (and *.gfx* file pathname), then only these two records must be in the file. If there are multiple column names and pathnames, then starting at the third record in the *.hierarchy* file is an N-column table of keywords of graphical objects, where N is the number of column names in the first record. Looking at the sample file, the first column contains “states” keywords, the second column “regions” keywords, the third the “eastWest” keywords, and the fourth the “USA” keyword. The matching *.gfx* files contain the positions and shapes of each of the column's graphical objects.

The third and remaining records in the hierarchy file are the hierarchy data. These records define how objects at one level correspond to objects at other levels.

The *.gfx* File

The *.gfx* files define the geometry of each object used by the Map Visualizer when displaying the objects. Each *.gfx* file contains multiple records, one for each object being displayed. Each record contains:

- the *gfx* keyword name
- the *gfx* full name
- the vertex pair count
- the shape hint
- the vertex pairs

The following steps guide you through the procedure for building .gfx files on UNIX systems. On Windows, an application for generating your own .gfx files, including instructions for using the utility to create maps, is downloadable from the MineSet Web page: <http://mineset.sgi.com>.

1. Using a digitizing scanner, convert a geographical image into an RGB image file format. Note that the image itself is not used by the Map Visualizer; it is just used as a template for defining the graphical objects in [Step 6 on page 111](#).
2. Launch the i3dm application in `/usr/demos/bin/`. (If this application is not currently installed, it can be installed from the IRIX 5.3 or 6.2 distribution, in the subsystem `demos.sw.tools`.) This creates windows on your screen: a Menu window on the left, an Input window across the bottom, and four windows (labeled *TOP*, *Pers*, *Front*, and *Right*) on the right. All *i3dm* windows must remain displayed (not iconified) for i3dm to work.
3. Move the cursor to the Front window.
4. Press the right mouse button to display options. Continue holding the right mouse button, and scroll to the Image Background option, then to the Load Image option. The Input window (at the bottom of your screen) prompts you for a name to apply to this image.
5. Enter the name of the RGB image file. The image appears in the Front window.
6. Delineate the shape of each object in the image by pointing and clicking at significant points on the boundary of each object. Do this in a clockwise sequence for each object. Each identified point is called a “vertex” and is represented by numeric x- and y-axis values. These values are assigned by the i3dm application and exist in a relative frame of reference for that RGB image file. The following procedure is used to delineate each object’s shape:
 - Use the middle mouse button to drag the image in the Front window so that the object you are going to delineate is completely exposed. If this is not possible, see step 8.
 - Go to the Menu window, and click the right mouse button on the Create pulldown menu.
 - Choose the Line option.
 - Start the point-and-click process of selecting vertices with the left mouse button in the Front window. Note that the greater the number of vertices you identify, the more accurate the resulting graphical image is.

- Note the red line crosshairs as you move the cursor over the image. As you click the left mouse button to declare each vertex, a small red box appears at that point. The box of the previous vertex changes to a small “x,” and a yellow line connects the new vertex to the previous vertices. As you move clockwise around the object, stop selecting vertices immediately before you are about to close the shape (that is, before clicking on the first vertex you selected when starting to delineate the object).
 - Go to the Menu window, and click the right mouse button on the *Attrib* pulldown menu.
 - Scroll to the *Name* option. The Input window (at the bottom of your screen) prompts you for a name.
 - Enter a unique identifier for the object you have just delineated. Do not use spaces. This becomes the object’s *gfx* keyword name. For example, in *population.usa.mapviz* the *gfx* column is specified as the first column in the data file. This first column contains strings such as “CA” and “NY.” These are the keyword names for the states. These keyword names are the *gfx* keyword names in the associated *gfx* file.
 - Go to the Menu window, and click the right mouse button on *Done*.
7. Repeat Step 6 for every other object in the same image. If the object adjoins a previously identified object, you must reuse common vertices by selecting them with the middle mouse button instead of the left mouse button. Using the middle mouse button while the crosshairs are positioned close to a previously selected vertex ensures that the newly selected vertex is identical to the previously selected one.
- Caution:** If a graphical object is too large to fit into the Front window, you must identify the vertices in sections. After all the objects are declared and the vertex information written to an ASCII file, you must edit this output file to join the sections of each subdivided object.
8. When all objects are identified, save the recorded vertices in a file. To do this:
- Go to the Menu window and press the right mouse button on the *File* pulldown menu.
 - Scroll down to the *File i3dm* format option and choose it. The Input window (at the bottom of your screen) prompts you for a filename.
 - Enter a filename, specifying the *.i3dm* suffix.

9. Exit the i3dm application. To do this
 - Go to the Menu window, and choose the *File* pulldown menu.
 - Scroll to the Exit option, and choose it.
10. Convert the i3dm format file into a gfx file format by using the convert.i3dm utility, using the following syntax:

```
/usr/lib/MineSet/mapviz/convert.i3dm inputFilename  
outputFilename.gfx
```

For each object, the utility prompts you to

- confirm the object's keyword name (which defaults to the *Attrib* name you supplied in Step 6, substep 6, above, when identifying the vertices)
- declare the object's full name (which is the name the user sees in the Map Visualizer's Selection window when using the mouse to select a geographical object)
- declare if the object has a concave shape that requires special handling

Note: Declaring an object to be concave results in an accurate graphical display, but at the cost of slower performance. One strategy is to declare no objects as concave, examine the display to determine which objects are inaccurately drawn, then manually edit the gfx files for those objects, changing the string "convex" to "concave." Another strategy is to declare all objects as "concave" (assuming there are few objects), then determine if the resulting performance is acceptable.

Creating Data and Configuration Files for the Scatter Visualizer

The first part of this chapter describes the types and formats of data supported by the Scatter Visualizer. Data input to the Scatter Visualizer must be provided as a single file containing raw data, usually in a tab-separated ASCII text form.

The second part discusses the configuration file, which describes how the Scatter Visualizer reads in, and displays, the data file.

Both the data and configuration files can be generated automatically by the Tool Manager. Read about the Scatter Visualizer in the *MineSet 3.0 Enterprise Edition User's Guide*, before using this chapter.

The Data File

In its simplest form, the data file consists of a list of lines, each containing a set of fields, each separated by one tab. (Other separators are also allowed, but only one per file can separate each field. See “[Input Options](#)” on page 128.) All lines must contain the same fields. The interpretation of the fields is specified by the configuration file, described in the next section. The store sales data is provided as part of the Scatter Visualizer package: for Windows in the directory in which MineSet is installed, under *Examples\scatterviz\store-type.data*. For IRIX in */usr/lib/MineSet/scatterviz/examples/store-type.data*. In this input file the first few lines appear as:

```
LIQUOR STORE      4300,4460,4800,4900,4700,4200,4250,4200
2700,2800,2750,3000,2900,2600,2500,2650
1600,1650,1900,1950,2000,2200,2300,2300
GROCERY STORE     700,900,600,800,877,755,800,600
3000,2900,3100,2800,2899,2950,3400,3300
10000,11000,9000,9800,9700,9650,9770,9700
```

In this sample file listing, each line consists of four fields, separated by tabs. The first field is a string that identifies a store type. The second field is an array of eight numbers, separated by commas, which might be sales of alcohol over an eight-day period. The third and fourth fields are also arrays of eight numbers that could represent sales of tobacco and food, respectively, over the same eight-day period.

The sample data file has other fields in the same format, but these are not shown. These additional fields correspond to sales of other products (see the configuration file *store-type.scatterviz* for a listing of all the fields). Windows users find this file in the directory in which MineSet is installed, under *\Examples\scatterviz*. IRIX users find the file in */usr/lib/MineSet/scatterviz/examples/store-type.scatterviz*

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; this generates blank fields. The order of the fields must match the format specified by the configuration file.

Data Types

The Scatter Visualizer supports integer, floating-point number, and string data types, as well as arrays of these types. The following data types are supported:

- **int** represents a 32-bit signed integer.
- **float** represents a single-precision floating point number. The decimal point is optional. Numbers in exponential “e” notation are also accepted.
- **double** represents a double-precision floating point number. The decimal point is optional when representing a floating point number. Numbers in exponential “e” notation are also accepted. The superior precision of **double** can be useful for accurately representing large numbers, since **float** can represent only seven or eight significant digits accurately. This superior accuracy, however, consumes twice the memory space of **float**.

- **dataString** represents a string that is unlikely to appear multiple times. If it appears multiple times, several copies are made. A **dataString** is typically used to store a memory address. Addresses are unlikely to be compared, and each record can have a different address.
- **string** represents a string of characters that can appear multiple times in the data file. Unlike a **dataString**, only a single copy of a given string is stored in memory, no matter how many times it appears in the data. This saves much memory for strings appearing many times.

Comparing **strings** is also much quicker than comparing **dataStrings**. Processing is somewhat slower when looking for duplicate strings as they are read in. An example of **string** use is for a division name that appears once for each department in the division. If you are unsure whether to use a **string** or a **dataString**, use a **string**.

- **fixed string** represents a string of fixed length. Like a **dataString**, if a **fixed string** appears multiple times, multiple copies are made. In general, **fixed strings** are used internally for representations of data from data bases, and are generally better to use than **strings** or **dataStrings**.
- **date** represents a date and time. In the data file, **date** must appear as in the format MM/DD/YYYY HH:MM:SS. Output from MineSet always represents dates with four-digit years, although two-digit years are acceptable for input. MineSet follows the X-OPEN standard for interpreting two-digit years. Fields with values 69 or greater are considered to be from the 20th century (1969-1999), and values from 0 to 68 are considered to be from the 21st century (2000 - 2068).

Arrays

With the Scatter Visualizer, you can use fields that are one- or two-dimensional arrays of fixed size. In a fixed-sized array field, all entries of the given field are arrays with the same number of values. Arrays contain the data values across one or two independent variables (those dimensions controlled by the sliders). In the listing from the file *store-type.data*, the second, third, and fourth fields are arrays.

Null Values

Any field or array element in the data file can also have the value “?” (question mark), indicating an unknown or null value.

The Configuration File

The configuration file format is flexible. Words in it must be separated by spaces, and it is case-sensitive. Except for the include statement and text within quoted strings, spacing and line breaks are irrelevant.

Sections

The configuration file consists of a series of sections, each of which has the form:

```
sectionKeyword
{
statements...
}
```

where *sectionKeyword* names the section. The order of the sections is significant, since sections can refer to variables defined in previous sections.

Defaults Files

As each section is encountered, a special configuration file (referred to as a *defaults file*) is also read in. Defaults files normally contain options statements. These files are read in the following order:

1. The *scatterviz* directory which usually contains system defaults. Windows users can find this in the directory in which MineSet was installed, under `\config\scatterviz`. IRIX users can find the directory in `/usr/lib/MineSet/scatterviz`.
2. The `~/.MineSet` directory (where the tilde, `~`, indicates your home directory). You can set up personal defaults in this directory.
3. The current directory. This lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section in the configuration file is read after all the defaults files; thus, options in the configuration file override those in the defaults files.

Statements

A statement has the following form:

```
statementKeyword info ;
```

where *statementKeyword* defines the statement, and *info* varies according to the keyword.

Variable Names

A variable name can appear in two formats:

- In the first format, it is a letter followed by a number of letters, digits, or underscores. It cannot be a keyword, and should not be quoted.
- In the alternate form, the variable name should be surrounded by back quotes (`). In this form, the variable name can match a keyword, and can contain even non-alphanumeric characters. The primary purpose of this second form is for configuration files generated automatically by the Tool Manager.

There is no scoping of variable names; a given variable name can only be declared once in the configuration file.

Options Statements

Many sections have options statements, which have the form

```
options optionName info, optionName info... ;
```

where *optionName* defines the specific option, and *info* depends on the option. In some cases, *optionName* can be more than one word. To maximize the number of allowable variable names, most option names are meaningful only within the appropriate options statement; option names do not conflict with variable names. You can declare several options on the same line, separating them by commas or placing them in several options statements. If two conflicting values for the same option appear, the last value is taken.

Include Statements

The configuration file can contain lines of the form

```
include "filename"
```

These lines can appear anywhere in the configuration file, but each must be on its own line. The filename must be in quotation marks; anything after the closing quote is ignored. The number of nested includes is unlimited. If a relative pathname (one not beginning with a slash) is specified, the file is first sought in the directory containing the current configuration file. If include statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the directory containing current configuration file, the include file is sought in the current directory.

Sinclude Statements

A statement similar to an include is `sinclude`, which has the form

```
sinclude "filename"
```

This is identical to the include statement, except that no error is given if the file does not exist; instead, the `sinclude` statement is ignored.

Strings and Characters

Strings and characters in the configuration file follow C conventions. Strings are in double quotation marks ("), and characters are in single quotation marks ('). All standard backslash conventions are followed (for example, `\n` represents a new line).

Comments

Comments begin with a pound (#) symbol at the beginning of a line; anything after this symbol to the end of the line is ignored, up to the end of the line.

Keywords

The keywords recognized by the Scatter Visualizer are listed in [Table 8-1](#). Variables cannot have these names unless they are surrounded by back quotes (`). Tokens appearing only in option statements are not keywords, and can be used for variable names.

Table 8-1 Scatter Visualizer Keywords

across	average	axis	background
buckets	by	color	colors
dataString	date	divide	double
entity	execute	expressions	file
float	from	include	input
int	key	label	legend
max	message	min	modulus
monitor	off	on	options
orientation	perspective	scale	separator
sinclude	size	slider	string
sum	summary	time	to
upvector	view	alpha	front
top	right	orderbt	

Currently, the keywords **execute**, **min**, **monitor**, and **time** are not used by the Scatter Visualizer.

Expressions

Expressions are accepted in several places in the input. Expressions follow the syntax of C. The following operations and their symbols are listed in [Table 8-2](#).

Table 8-2 Operators Used With Expressions

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
==	Equals
!=	Not equals
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
&&	AND
	OR
!	NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
A?B:C	If (A), then B else C

Also, the following functions are available:

- **divide**(*x*, *y*, *z*) divides *x* by *y*, unless *y* is zero. If *y* is zero, the result is *z*; this is equivalent to $y=0 ? z : x/y$.
- **modulus**(*x*, *y*, *z*) is similar to **divide**, but for modulus.

Type handling is similar to that in C. Expressions using **int** and **float** promote both sides to float. Expressions using **int** and **double**, or **float** and **double** promote both sides to double. The result of a relational expression (for example, `==`, `<`) is always an **int**. Type casting is also supported.

Unlike in C, strings can be compared using relational expressions; the strings are compared lexicographically.

The Input Section

The first section of a configuration file is normally the input section. It defines the name and format of the data file. A typical input section might look like this:

```
input {
    file "company.data";
    string company;
    slider int income from 20000 to 60000 by 10000;
    slider date "%N %Y" purchaseDate from "Jan 1990" to "Dec
1992" by "1 month";
    options array separator `,';
    float lifeSales[income][purchaseDate];
    float autoSales[income][purchaseDate];
    float homeSales[income][purchaseDate];
    string location;
}
```

This example states that the input file is called *company.data*, and that there are five fields: *company*, *lifeSales*, *autoSales*, *homeSales*, and *location*. The *company* and *location* fields are of type **string**, while the other three fields are two-dimensional arrays of type **float**. Two slider dimensions are declared:

- *income*, which is of type **int**, ranges from 20000 to 60000 in increments of 10000; and
- *purchaseDate*, which is of type **date** and ranges from January 1990 to December 1992 in increments of 1 month.

The arrays *lifeSales*, *autoSales*, and *homeSales* contain values for each income and purchase date. Individual values within the arrays are separated by commas.

When the **input** section is entered, the defaults file *inputDefaults* is read in.

File Statements

The **file** statement names the data file to be read. This statement is required. Its form is:

```
file "filename";
```

filename must be in double quotation marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current configuration file. If include statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the current configuration file's directory, the file is sought in the current directory.

Enumeration Statements

Enumeration statements declare enumerations, or enums, that index into array fields. The enum statement has three forms.

- The first enum statement form is

```
enum type name from value1 to value2 by increment;
```

This declares an enum with values starting at *value1* and incremented by *increment* until they reach or exceed *value2*. For example, the statement

```
enum int age from 20 to 70 by 10;
```

declares age as an enum with the values 20, 30, 40, 50, 60, and 70.

Type must be a number type (**int**, **float**, or **double**) or **date** (see [“Dates” on page 125](#)).

- The second enum statement form is

```
enum type name from value1 to value2 across numberOfValues;
```

This declares an enum with values ranging from *value1* to *value2*. The *numberOfValues* is an integer specifying the number of values. For example, the statement:

```
enum int age from 20 to 70 across 6;
```

declares age as an enum with the values 20, 30, 40, 50, 60, and 70.

Type must be a number type (**int**, **float**, or **double**) or **date** (see [“Dates” on page 125](#)).

- The third enum statement explicitly lists the enum values. Its form is:

```
enum type name { value1, value2, ..., valueN };
```

Type can be any type or date (see [“Dates” on page 125](#)).

Dates

The enum statement includes special support for a date type that handles date and time values starting Jan 1, 1753. The date type is valid only within enum statements. A date enum statement can have the following syntaxes:

```
enum date "format" name from "value1" to "value2" across
    numberOfValues;
enum date "format" name { value1, value2, ..., valueN };
enum date "format" name from "value1" to "value2" by
    "increment";
```

The *format* string specifies the format of the values; it is useful for controlling how dates are displayed in the animation control panel. The syntax of the *format* string is similar to the **scanf** function in C. Various units of time are represented by special characters preceded by the percent symbol (%). For example,

```
enum date cq "Calendar Q%Q, %Y" from "Calendar Q1, 1980" to "Calendar
Q3, 1985" by "1 quarter";
```

The “Calendar Q” in the *format* string matches the “Calendar Q” in *value1* and *value2*. The %Q in the *format* string indicates that the next number in *value1* and *value2* is the calendar quarter. The comma and space in the *format* string match the commas and spaces in the values. Finally, the %Y in the *format* string specifies that the year values are next.

[Table 8-3](#) lists the characters that can follow the percent symbol and the units of time they represent.

Table 8-3 Characters That Can Follow the Percent Symbol in the Format String

Character	Time Unit	Precision
Y	year	4
Q	calendar quarter	1
M	month	2
N	month name	>= 3
D	day	2
h	hour	2
m	minute	2
s	second	2

With the exception of N, each character matches an integer of the specified precision. N matches 3 or more characters giving the English name of the month.

The from-to-by form of the enum statement includes an increment value. For dates, the increment is a quoted string containing an integer, an optional space, and one of the special characters in [Table 8-3](#) or one of the symbols **year**, **quarter**, **month**, **day**, **hour**, **minute**, and **second**. The plural forms of these symbols are also accepted. Note that these symbols are not keywords, since they have special meaning only in the increment string. The following are examples of valid increments:

```
"1 year"  
"7 days"  
"4h"
```

Data Statements

The data statements declare the fields in the data file. The fields must be declared in the order they appear in the data file. The format of most data statements is

```
type name;
```

where *type* is **int**, **float**, **double**, **string**, **dataString**, **date**, and **fixedString(*n*)**, where *n* is an integer representing the width of the string; *name* is the variable name. Unlike in C, only one variable can be declared per statement.

A data field can also be based on an enumeration. The syntax is

```
enum enumName name;
```

The field must contain **ints** corresponding to the values of the enum. For example, if the enum `ageGroup` is declared as

```
enum string ageGroup { "below 30", "30-39", "40-49", "50-59",
    "60 or above" };
```

the field `age` can be declared as

```
enum ageGroup age;
```

The field should contain **ints** between 0 and 4, where 0 is displayed as “below 30,” 1 as “30-39”, and so forth.

Only one variable can be declared per statement.

Arrays

Arrays are also declared using data declarations. The declaration syntax for one-dimensional arrays is one of the following:

```
type name [ number ] ;
type name [ enumName ] ;
type name [ null enumName ] ;
```

For example:

```
float revenue [50];
```

The declaration syntax for two-dimensional arrays is one of the following:

```
type name [ number1 ][ number2 ] ;
type name [ enumName1 ][ enumName2 ] ;
type name [ null enumName1 ][ null enumName2 ] ;
```

For example:

```
float revenue [50][10];
```

When enums are used, the number of values in the array is taken from the declaration of the enum. For example, given the statements

```
enum int age from 20 to 70 by 10;
float clothingPurchases[age];
```

the array *clothingPurchases* must have six values, corresponding to the enum values 20, 30, 40, 50, 60, and 70.

The keyword **null** indicates an extra value at the beginning of the array, corresponding to null. Thus, the statements

```
enum int age from 20 to 70 by 10;
float clothingPurchases[null age];
```

declare *clothingPurchases* as an array with seven values: the first value corresponding to null or unknown age values, and the remaining six values corresponding to age values 20, 30, 40, 50, 60, and 70.

You can override the separator between values in an array by declaring it as:

```
type name [ number ] separator 'char';
```

For example:

```
float revenue [50][10] separator ':';
```

If no separator is specified, the default separator (usually a tab) is used.

Input Options

All **options** statements begin with the word “options” and have one or more comma-separated options.

- The separator option defines the separator between fields in the data file. The default separator is a tab. The syntax is

```
options separator 'char';
```

For example:

```
options separator ':';
```

Note: The separator is used also to separate values within arrays; however, arrays can override the separator.

- The backslash option controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

```
options backslash off;
```

```
options backslash on;
```

The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than separators. Within strings, this causes standard C-style backslash processing.

The Expressions Section

The **expressions** section of a configuration file lets you define additional fields that are expressions of existing fields. For example, one field can be defined as the sum of two other fields.

The format of the expressions section is

```
expressions
{
expressionDeclaration;
...
}
```

where *expressionDeclaration* has the following form:

```
type name = expression ;
```

The following is a sample expression section. This section assumes two existing array fields of type **double**: “male” and “female”; these represent spending by males and females on various goods across time (one independent dimension). Two fields are added: “total” represents the total dollars spent, and “pctFemale” represents the percentage of dollars spent by females.

```
expressions
{
double total[36] = male+female;
double pctFemale[36] = divide (female*100, total, 50.0);
}
```

Note: The `pctFemale` calculation uses “total,” defined in the previous statement. Also, note the use of the divide function rather than the `/` operator. This results in 50% for the case where there are no dollars spent at all; using the `/` operator generates a divide by zero error.

The expressions section has no options; thus, no defaults file is read in for it.

The View Section

The **view** section of a configuration file describes how the data is displayed, including the mapping of sizes, colors, axes, and so on. The default values for these options are in `view.scatterviz.options`. The Windows options are in the directory in which MineSet is installed under `\Programs\scatterviz\view.scatterviz.options`. IRIX users find the file in `/usr/lib/MineSet/scatterviz/view.scatterviz.options`. Its form is:

```
view
{
viewStatement;
...
}
```

A sample view section is

```
view {
  slider month;
  entity brand;
  axis male$, color "blue";
  axis female$, color "red";
  size total$, max 5;
  color pctFemale, scale 0 50 100, colors "blue" "gray"
  "red";
  message "brand %s, total sales %,.0f",brand, total$;
}
```

When entering the view section, the `viewDefaults` file is read in.

Slider Statement

The **slider** statement identifies an enum to be used as a slider dimension. Its syntax is one of the following:

```
slider enumName ;  
slider null enumName ;
```

The enum name is declared in the input section. If the keyword **null** is present, the slider includes a position at the beginning corresponding to null or unknown values of the enum. Arrays indexed by the slider must be declared to match the *null* in the slider statement.

There can be 0, 1, or 2 slider statements. The first slider statement applies to the horizontal slider, the second to the vertical slider. If there is no slider statement, the resulting display does not include animation.

Entity Statement

The **entity** statement lets you specify a variable that uniquely identifies the entities in the display. The entity statement consists of a series of clauses, separated by commas:

```
entity clause1, clause2, ...
```

Alternatively, the clauses can be given in separate entity statements.

The Entity Variable

The first clause of the **entity** statement normally contains the name of the entity variable (*brand* in the example on [page 130](#)).

The Label Clause

This clause defines how the entities are labeled. It has the following forms:

- `label off`
This turns off the labels.
- `label on`
This turns on the labels. The default labels use the entity variable as the label for each entity.
- `label variable`
This turns on the labels and uses the given variable to label the entities. When this form is used, it is not necessary to specify *label on*.

The Label Color Clause

This clause turns on the labels and specifies their color. It has the form:

```
label color "colorname"
```

where *colorname* is the name of a color in a special format. (Color naming is explained in [“Color Statement” on page 134.](#)) The default label color is gray.

The Legend Clause

The legend clause explains what the entities are. Any string can be placed in the entity legend. The legend clause has the following forms:

- `legend off`
This turns off the entity legend.
- `legend on`
This turns on the entity legend (this is the default). The default legend is
`Entity: varname`
where *varname* is the name of the entity variable.
- `legend label "string"`
This turns the legend on and explicitly sets the legend string. If this form is used, *legend on* is unnecessary.

Size Statement

The **size** statement describes how a field of data is mapped to the sizes of entities. The size statement consists of a series of clauses, separated by commas:

```
size clause1, clause2,...
```

Alternatively, the clauses can be given in separate size statements.

The Size Variable

The first clause normally contains the name of a field to be mapped to size (*total\$*, in the view example in [“The View Section” on page 130](#)). The field must be of a number type (**int**, **float**, or **double**), of which **float** is the most efficient. The field can be an array that is indexed by slider dimensions. If no size field is specified, all entities are the same size.

The Max Clause

Normally, the size variable is mapped to the size of the entities, so that the biggest entity has a size of 5. This size can be changed by specifying a different value. If there is no size variable, the default maximum size is 5. The **max** clause has the form:

```
max float
```

The Scale Clause

Instead of using the **max** clause to affect size values, the **scale** clause can be used to scale these values; all values are multiplied by the scale. The scale clause’s syntax is

```
scale float
```

The Legend Clause

The **legend** clause defines the meaning of the size mappings. Any string can be placed in the size legend. The legend clause has the following forms:

- `legend off`

This turns off the size legend.

- `legend on`

This turns on the size legend (this is the default). The default legend is:

```
size:varname
```

where *varname* is the name of the variable that is mapped to size.

- `legend label "string"`

This turns the legend on and explicitly sets the legend string. If this form is used, *legend on* is unnecessary.

Color Statement

The **color** statement describes how values are mapped to colors. The format is similar to the size statement, consisting of several clauses that can be separated by commas, or entered as multiple statements. The syntax is:

```
color clause1, clause2,...
```

Color Naming

Color names follow the conventions of the X window system, except that the names must be in quotes. Examples of valid colors are "green," "Hot Pink," and "#77ff42." The latter is in the form "*#rrggbb*", in which the red, green, and blue components of the color are specified in hexadecimal value. Pure saturation is represented by ff, a lack of color by 00. For example, "#000000" is black, "#ffffff" is white, "#ff0000" is red, and "#00ffff" is cyan.

- Windows users should only use the "#ff0000" form, as only some of the named colors are supported (for example, *white, black, gray, red, yellow, green, etc.*).
- IRIX users can find the color list file at `/usr/lib/X11/rgb.txt`.

The Color Variable

As with size, you also can specify a single field to be mapped to an entity color. The field can be an array that is indexed by slider dimensions. If the field is an array, it must be a number type. If the field is a number type, the scale and buckets clauses described below can be used to map a range of colors to the values of the field. If the field is not a number type, it is sorted, and each unique value is assigned a color.

The colors Clause

The **colors** clause specifies the colors to be used. The colors clause's syntax is:

```
colors "colorname" "colorname"...
```

The format for *colorname* is described in [“Color Naming” on page 134](#). Note that there are no commas between the colors, because commas are used to separate clauses in the color statement. A sample colors clause is:

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the above example, red is color 0, gray is color 1, and blue is color 2.

If there is no colors statement, colors are chosen randomly. If there is a colors statement, at least as many colors must be specified as are to be mapped.

The scale Clause

The **scale** clause allows assignment of values to a continuous range of colors. For example, when displaying a percentage, red can be assigned to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the scale clause is

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The colors statement must contain at least as many colors as are to be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the `pctFemale` field indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The colors statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

Use the `scale` clause only in conjunction with a numeric color variable.

The buckets Clause

The **buckets** clause is similar to the `scale` clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the buckets clause is

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the `scale` clause.

If, in the above example, you used the buckets clause instead of the `scale` clause, the statement would be:

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater than or equal to 100 are colored red. Values greater than, or equal to, 50 but less than 100, are gray. All other values are then blue.

Use the buckets clause only with a numeric color variable.

The legend Clause

The **legend** clause creates a legend of the colors. The legend clause syntax can be any of the following:

```
legend off  
legend on  
legend "string" "string" ...  
legend label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. It can be omitted if other legend statements are included. Specifying only **legend on** generates the default legend.

The default legend includes a single label to the left (with the name of the field that is mapped to color), and a list of colored labels on the right (with values obtained from the scale clause, the buckets clause, or from the field). To override the strings in the colored labels, specify the strings as:

```
legend "string" "string"
```

To override the label on the left, specify it following the word *label*. To eliminate this label, specify an empty string; that is:

```
legend label ""
```

Axis Statement

The **axis** statement causes a variable to be used as an axis in the 3D landscape. The variable's values determine where the entities are positioned on the axis. There can be up to three axis statements. Like the size and color statements, the axis statement contains a series of comma-separated clauses, but all of them must be specified in a single statement.

```
axis clause1, clause2,...
```

The Axis Variable

As with size and color, you can specify a field to be used as an axis. The field can be an array that is indexed by slider dimensions. If the field is an array, it must be of type **number**. If the field is not of type number, it is sorted, and each unique value is assigned a position along the axis.

The Label Clause

The **label** clause has the form:

```
label "string"
```

The string is used to label the axis. It appears in the landscape, at the end of the axis line. The default label is the name of the axis variable.

The Max Clause

Normally, the axis variable is mapped directly to the position of the entities along the axis0. The **max** clause lets you normalize the values of the axis variable, so that the maximum value is mapped to the specified max. The max clause's syntax is:

```
max float
```

The Scale Clause

Instead of using the **max** clause to affect position values, the **scale** clause can be used to scale the values. All values are multiplied by the scale. The scale clause syntax is

```
scale float
```

The Color Clause

The **color** clause specifies the color used for the axis line and label. It has the form:

```
color "colorname"
```

The Extend Clause

The **extend** clause specifies whether the axis should be extended automatically to include the value zero. It has the form:

```
extend on  
extend off
```

The Orderby Clause

The **orderby** clause forces a particular order by string values on an axis. By default the strings are arranged to correspond with the color variable. The only alternative ordering offered currently is alphabetical (**orderby** alpha).

Summary Statement

The **summary** statement specifies a summation to be calculated over all the entities. The summary is used to color the drawing window in the animation control panel. Like the size and color statements, the summary statement has several clauses that can be specified in one statement, separated by commas, or in separate statements.

```
summary clause1, clause2,...
```

The Summary Variable

You can specify the variable to be used in the summary. This variable must be of number type. Typically, the summary variable is an array indexed by slider dimensions, so that the summary value varies across the slider dimensions.

The Color Clause

The **color** clause specifies the color used to display the summary values in the drawing window. It has the form

```
color "colorname"
```

Various shades of the color, from white to the specified color, are used to represent summary values. The minimum summary value is mapped to white, while the maximum summary value is mapped to the specified color. The default summary color is red.

The Legend Clause

The **legend** clause creates a legend of the summary colors. The legend clause syntax can be any of the following:

```
legend off  
legend on  
legend label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. It can be omitted if other legend statements are included. Specifying only **legend on** generates the default legend.

The legend includes a single label to the left (which defaults to the aggregation function and variable used in the summary), and two colored labels on the right (with the minimum and maximum summary values). To override the label on the left, specify it following the word *label*. To eliminate this label, specify an empty string; that is

```
legend label ""
```

Drillthrough Statement

The Drillthrough Statement specifies a column to be used to define the drill through expression. This column must be string valued, and should specify an expression which defines the object when selected (assuming each row defines a object).

The syntax for this statment is of the form:

```
drillthrough stringColumnName;
```

Drill through mapping is used in the *.scatterviz* file generated by the associations mining algorithm. It is used here because the table in the datafile contains rules and properties of the rules, not the columns in the original dataset.

The Selections > Drill Through Columns menu in the Scatter Visualizer has no effect when a specific drill through column has been mapped.

Message Statement

The **message** statement specifies the message displayed when an entity is selected. The syntax is similar to that of the C **printf** statement. A sample message statement is

```
message "%s: $%f, %.0f%% of target, %.0f%% of last year",  
        product, sales, pctTarget, pctLastYear;
```

This could produce the following message:

```
furniture: $2425.37, 23% of target, 87% of last year
```

The formats must match the type of data being used:

- Strings must use %.
- Ints must use integer formats (such as %d).
- Floats and doubles must use floating point formats (such as %f).

For details of the **printf** format, see the `printf (1)` reference (man) page (type `man printf` at the shell prompt).

A special format type has been added to **printf**. If the percent sign is followed by a comma (for example, “%,f”), commas are inserted in the number for clarity. Only the United States convention of d,ddd,ddd.dddd is supported, with the decimal point represented by a period, and commas separating every three places to the left of the decimal point. For example, if the above format were:

```
message "%s: $%,f, %, .0f%% of target, %, .0f%% of last year",
        product, sales, pctTarget, pctLastYear;
```

it would produce the message:

```
furniture: $2,425.37, 23% of target, 87% of last year
```

The `$`, `*`, `h`, `l`, `ll`, `L`, and `n` **printf** format options are not supported.

All values, including the format string, are expressions. Thus, if you had a `pctFemale` field, but wanted a more gender-neutral message, you could use:

```
message pctFemale>50?"%f%% females":"%f%% males",
        pctFemale>50?pctFemale:100-pctFemale;
```

If `pctFemale` is 70, the message “70% females” is displayed; if `pctFemale` is 30, the message “70% males” is displayed. In this case, you can also achieve the same result with a single format string:

```
message "%f%% %s", pctFemale>50?pctFemale:100-pctFemale,
        pctFemale>50?"females":"males";
```

If no message is specified, a default message containing the names and values of all the fields is used.

Execute Statement

The **execute** statement lets you execute a shell command by double-clicking an object. The syntax is similar to that of the *message* command.

Here is a sample execute statement you can use on UNIX that uses *xconfirm* to show a window with information about the item. Note that the command line (string) is shown as three lines. In an actual file, this should be on a single line. Multi-line strings are not supported.

```
execute "xconfirm -t '%s' -t 'population %,.0f' -t '%,.0f per
      sq mile' -t '%,.0f sq-miles' > /dev/null", states,
      population, density, sqMiles;
```

This might produce a dialog with the message:

```
CA
64 per sq mile
266,807 sq-miles
```

A Windows sample execute statement follows:

```
execute "execute "echo -e '%s \n population %,.0f \n %,.0f per sq mile
      \n %,.0f sq-miles'>temp; notepad temp; rm temp", states,
      population, density, sqMiles;
echo -e '%s \n population %,.0f \n %,.0f per sq mile \n %,.0f
      sq-miles'>temp; notepad temp; rm temp", states, population,
      density, sqMiles;
```

If there is no execute statement, double-clicking an object has the same effect as single-clicking it.

The sample execute statement will work on Windows as well as UNIX, because a simple *xconfirm* utility is provided with the installation on Windows.

Filter Statement

The **filter** statement specifies that only entities meeting certain filter criteria are displayed initially. The filter criteria are in the form of expressions whose values must all be true or nonzero for an entity to be displayed (expressions are described in [“Expressions” on page 122](#)).

The syntax of the **filter** statement is

```
filter expression, expression,...
```

For example, the statement

```
filter state == "CA" || state == "WA", sales > 9000, pctTarget >= 90;
```

specifies that only records from California or Washington state, with sales greater than 9000 and a *pctTarget* value greater than or equal to 90 should be displayed initially.

After the Scatter Visualizer is invoked, the filter criteria can be changed or removed interactively using the filter panel.

View Options

The **view** section of the configuration file has several options for controlling parameters of the display. These options can appear in a single options statement, separated by commas, or in separate options statements. The syntax of the options statement is

```
options option, option,...
```

The following options are available:

- `entity label size float`
controls the size of the entity labels.
- `axis label size float`
controls the size of the axis labels.
- `hide entity label distance float`
controls the distance at which entity labels become invisible. Smaller distances might improve performance, but the labels disappear more quickly.
- `grid color "colorname"`
controls the color of the grid.
- `grid size float float float`
controls the spacing between grid lines. It applies the three values to grid lines along the x, y, and z axes, respectively.
- `entity shape shapeName`
specifies the shape used to display entities. *shapeName* can be "cube," "bar," or "diamond."
- `background color "colorname"`
specifies the initial background color.
- `orientation top right front float float float`
specifies the initial orientation of the scene. You may use three floating point values to specify an arbitrary orientation vector.

- `perspective` *on off*
specifies whether or not to use perspective initially. The default is to use it.
- `upvector` *float float float*
specifies a camera upvector for defining a frame of reference. The default is 0.10.1. This option combined with the `orientation` option allows any initial viewing orientation.

Creating Data and Configuration Files for the Splat Visualizer

The first part of this chapter describes the types and formats of data supported by the Splat Visualizer. Data input to the Splat Visualizer must be provided as a single file containing raw data, in a tab-separated ASCII text.

The second part discusses the configuration file, which describes how the Splat Visualizer reads in, and displays, the data file.

Both the data and configuration files can be generated automatically by the Tool Manager. Read the about the Splat Visualizer in the *MineSet 3.0 Enterprise Edition User's Guide* before using this chapter.

The Data File

In its simplest form, the data file consists of a list of lines, each containing a set of fields, each separated by one tab. (Other separators are also allowed, but only one per file can separate each field. See “[Input Options](#)” on page 155.) All lines must contain the same fields. The interpretation of the fields is specified by the configuration file, described in the next section. It uses the *adultJobs* data file provided as part of the Splat Visualizer package. Windows users find the file in the directory in which MineSet is installed under *Examples\splatviz\examples*. IRIX users find the file in */usr/lib/MineSet/splatviz/examples/adultJobs.data*.

the first few lines of the input file appear as:

Bachelors	Adm-clerical	3	3	51189.4869565217	115
Bachelors	Exec-managerial	2	5	70722.6271186441	59
Bachelors	Adm-clerical	2	3	37876.328358209	134
Bachelors	Exec-managerial	3	0	34436.8	5
Bachelors	Tech-support	1	2	37583.66667	3
Bachelors	Tech-support	1	3	13711.33333	3
Bachelors	Tech-support	1	4	29878.74193	31

In this sample file listing, each line consists of six fields, separated by tabs. The first field is a string that identifies level of education. The second field is a string which identifies occupation. The third field identifies the age bin. The fourth field identifies the number of hours per week worked bin. The fifth field quantifies the average gross income. The sixth field is the weight of records in the aggregate (i.e. the record count, unless record weighting has been used). This data file was derived from *adult94.data* by performing Tool Manager operations (specifically binning and aggregation). Windows users find the file in the directory in which MineSet was installed under `\Examples\data\adult94.data`. IRIX users find the file in `/usr/lib/MineSet/data/adult94.data`.

The data file cannot contain blank lines or comments. Missing or extra data on a line causes an error.

Note: One tab (the default separator) separates each field. Do not insert multiple tabs to line up the fields visually; this generates blank fields. The order of the fields must match the format specified by the configuration file.

Data Types

The Splat Visualizer supports the following seven data types:

- **int** represents a 32-bit signed integer.
- **float** represents a single-precision floating point number. The decimal point is optional. Numbers in exponential “e” notation are also accepted.
- **double** represents a double-precision floating point number. The decimal point is optional when representing a floating point number. Numbers in exponential “e” notation are also accepted. The superior precision of **double** can be useful for accurately representing large numbers, since **float** can represent only seven or eight significant digits accurately. This superior accuracy, however, consumes twice the memory space of **float**.
- **dataString** represents a string that is unlikely to appear multiple times. If it appears multiple times, several copies are made. A **dataString** is typically used to store an address. Addresses are unlikely to be compared, and each record can have a different address.

- **string** represents a string of characters that can appear multiple times in the data file. Unlike a **dataString**, only a single copy of a given string is stored in memory, no matter how many times it appears in the data. This saves much memory for strings appearing many times.

Comparing **strings** is also much quicker than comparing **dataStrings**. Processing is somewhat slower when looking for duplicate strings as they are read in. An example of **string** use is for a division name that appears once for each department in the division. If you are unsure whether to use a **string** or a **dataString**, use a **string**.

- **fixed string** represents a string of fixed length. Like a **dataString**, if a **fixed string** appears multiple times, multiple copies are made. In general, **fixed strings** are used internally for representations of data from data bases, and are generally better to use than **strings** or **dataStrings**.
- **date** represents a date and time. In the data file, **date** must appear in the format MM/DD/YYYY HH:MM:SS. Output from MineSet always represents dates with four-digit years, although two-digit years are acceptable for input. MineSet follows the X-OPEN standard for interpreting two-digit years. Fields with values 69 or greater are considered to be from the 20th century (1969-1999), and values from 0 to 68 are considered to be from the 21st century (2000 - 2068).

Null Values

Any field element in the data file can also have the value “?” (question mark), indicating an unknown or null value.

The Configuration File

The configuration file format is flexible. Words in it must be separated by spaces, and it is case-sensitive. Except for the include statement and text within quoted strings, spacing and line breaks are irrelevant.

Sections

The configuration file consists of a series of sections, each of which has the form:

```
sectionKeyword
{
statements...
}
```

where *sectionKeyword* names the section. The order of the sections is significant, since sections can refer to variables defined in previous sections.

Defaults Files

As each section is encountered, a special configuration file (referred to as a *defaults file*) is also read in. Defaults files normally contain options statements. These files are read in the following order:

1. The *splatviz* directory which usually contains system defaults. Windows finds the directory in the directory in which MineSet is installed under `\config\splatviz`. IRIX users find the directory in `/usr/lib/MineSet/splatviz`.
2. The `~/.MineSet` directory (where the tilde [~] indicates your home directory). You can set up personal defaults in this directory.
3. The current directory. This lets you set up defaults for each directory.

Files with the same name can appear in more than one of the above-named directories; in this case, the order given is the one in which the directories are read. If the same option is found in multiple files, the last option read is used. Note that the appropriate section in the configuration file is read after all the defaults files; thus, options in the configuration file override those in the defaults files.

Statements

A statement has the following form:

```
statementKeyword info ;
```

where *statementKeyword* defines the statement, and *info* varies according to the keyword.

Variable Names

A variable name can appear in two formats:

- In the first format, it is a letter followed by a number of letters, digits, or underscores. It cannot be a keyword, and should not be quoted.
- In the alternate form, the variable name should be surrounded by back quotes (`). In this form, the variable name can match a keyword, and can contain even non-alphanumeric characters. Configuration files generated automatically by the Tool Manager use this form.

There is no scoping of variable names; a given variable name can only be declared once in the configuration file.

Options Statements

Many sections have options statements, which have the form

```
options optionName info, optionName info... ;
```

where *optionName* defines the specific option, and *info* depends on the option. In some cases, *optionName* can be more than one word. To maximize the number of allowable variable names, most option names are meaningful only within the appropriate options statement; option names do not conflict with variable names. You can declare several options on the same line, separating them by commas or placing them in several options statements. If two conflicting values for the same option appear, the last value is taken.

Include Statements

The configuration file can contain lines of the form

```
include "filename"
```

These lines can appear anywhere in the configuration file, but each must be on its own line. The filename must be in quotation marks; anything after the closing quote is ignored. The number of nested includes is unlimited. If a relative pathname (one not beginning with a slash) is specified, the file is first sought in the directory containing the current configuration file. If include statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the directory containing the current configuration file, the include file is sought in the current directory.

Sinclude Statements

A statement similar to an include is `sinclude`, which has the form

```
sinclude "filename"
```

This is identical to the `include` statement, except that no error is given if the file does not exist; instead, the `sinclude` statement is ignored.

Strings and Characters

Strings and characters in the configuration file follow C conventions. Strings are in double quotation marks (`"`), and characters are in single quotation marks (`'`). All standard backslash conventions are followed (for example, `\n` represents a new line).

Comments

Comments begin with a pound (`#`) symbol at the beginning of a line; anything after this symbol to the end of the line is ignored, up to the end of the line.

Keywords

The keywords recognized by the Splat Visualizer are listed in [Table 9-1](#). Variables cannot have these names unless they are surrounded by back quotes (```). Tokens appearing only in option statements are not keywords, and can be used for variable names.

Table 9-1 Splat Visualizer Keywords

<code>across</code>	<code>average</code>	<code>axis</code>	<code>buckets</code>
<code>by</code>	<code>color</code>	<code>colors</code>	<code>weight</code>
<code>dataString</code>	<code>date</code>	<code>divide</code>	<code>double</code>
<code>execute</code>	<code>expressions</code>	<code>file</code>	<code>float</code>
<code>from</code>	<code>include</code>	<code>input</code>	<code>int</code>
<code>key</code>	<code>label</code>	<code>legend</code>	<code>max</code>
<code>message</code>	<code>min</code>	<code>modulus</code>	<code>monitor</code>

Table 9-1 (continued) Splat Visualizer Keywords

off	on	options	opacity
scale	separator	sinclude	size
slider	string	sum	summary
time	to	view	background
orientation	front	top	right

Currently, the keywords **execute**, **monitor**, **weight**, and **time** are not used by the Splat Visualizer.

The Input Section

The first section of a configuration file is normally the input section. It defines the name and format of the data file. A typical input section might look like this:

```
input {
    file "adultJobs.data";
    enum string `age_bin_k` {"- 20", "20-30", "30-40",
"40-50", "50-60", "60-70", "70+"};
    enum string `hours_per_week_bin_k` {"- 20", "20-25", "25-30",
"30-35", "35-40", "40-45", "45-50", "50-55", "55-60", "60-65", "65-70",
"70+"};
    string `education`;
    string `occupation`;
    enum `age_bin_k` `age_bin`;
    enum `hours_per_week_bin_k` `hours_per_week_bin`;
    double `avg_gross_income`;
    int `count_gross_income`;
}
```

This example states that the input file is called *adultJobs.data*, and that there are six fields: *education*, *occupation*, *age_bin*, *hours_per_week_bin*, *avg_gross_income*, and *count_gross_income*. The *education* and *occupation* fields are of type **string**. The *age_bin* and *hours_per_week_bin* are of type **enum**, where the values of these enums is defined by *age_bin_k* and *hours_per_week_bin_k* respectively. The column *avg_gross_income* is of type **double** and the field *count_gross_income* is of type **int**.

When the **input** section is entered, the defaults file *inputDefaults* is read in.

File Statements

The **file** statement names the data file to be read. This statement is required. Its form is:

```
file "filename";
```

filename must be in double quotation marks. If it is a relative pathname (no leading slash), it is first sought in the directory containing the current configuration file. If include statements are present, this might not be the same as the initially loaded configuration file. If it is not found in the current configuration file's directory, the file is sought in the current directory.

Enumeration Statements

Enumeration statements declare enumerations, or enums. The enum statement has three forms.

- The first enum statement form is

```
enum type name from value1 to value2 by increment;
```

This declares an enum with values starting at *value1* and incremented by *increment* until they reach or exceed *value2*. For example, the statement

```
enum int age from 20 to 70 by 10;
```

declares age as an enum with the values 20, 30, 40, 50, 60, and 70.

Type must be a number type (**int**, **float**, or **double**) or **date** (see [“Dates” on page 153](#)).

- The second enum statement form is

```
enum type name from value1 to value2 across numberOfValues;
```

This declares an enum with values ranging from *value1* to *value2*. The *numberOfValues* is an integer specifying the number of values. For example, the statement:

```
enum int age from 20 to 70 across 6;
```

declares age as an enum with the values 20, 30, 40, 50, 60, and 70.

Type must be a number type (**int**, **float**, or **double**) or **date** (see [“Dates” on page 153](#)).

- The third enum statement explicitly lists the enum values. Its form is:

```
enum type name { value1, value2, ..., valueN };
```

Type can be any type or date (see [“Dates” on page 153](#)).

Dates

The enum statement includes special support for a date type that handles date and time values starting Jan 1, 1753. The date type is valid only within enum statements. A date enum statement can have the following syntaxes:

```
enum date "format" name from "value1" to "value2" across
    numberOfValues;
enum date "format" name { value1, value2, ..., valueN };
enum date "format" name from "value1" to "value2" by
    "increment";
```

The *format* string specifies the format of the values; it is useful for controlling how dates are displayed in the animation control panel. The syntax of the *format* string is similar to the **scanf** function in C. Various units of time are represented by special characters preceded by the percent symbol (%). For example,

```
enum date cq "Calendar Q%Q, %Y" from "Calendar Q1, 1980" to "Calendar
Q3, 1985" by "1 quarter";
```

The “Calendar Q” in the *format* string matches the “Calendar Q” in *value1* and *value2*. The %Q in the *format* string indicates that the next number in *value1* and *value2* is the calendar quarter. The comma and space in the *format* string match the commas and spaces in the values. Finally, the %Y in the *format* string specifies that the year values are next.

[Table 9-2](#) lists the characters that can follow the percent symbol and the units of time they represent.

Table 9-2 Characters That Can Follow the Percent Symbol in the Format String

Character	Time Unit	Precision
Y	year	4
Q	calendar quarter	1
M	month	2
N	month name	>= 3
D	day	2
h	hour	2
m	minute	2
s	second	2

With the exception of N, each character matches an integer of the specified precision. N matches 3 or more characters giving the English name of the month.

The from-to-by form of the enum statement includes an increment value. For dates, the increment is a quoted string containing an integer, an optional space, and one of the special characters in [Table 9-2](#) or one of the symbols **year**, **quarter**, **month**, **day**, **hour**, **minute**, and **second**. The plural forms of these symbols are also accepted. Note that these symbols are not keywords, since they have special meaning only in the increment string. The following are examples of valid increments:

```
"1 year"  
"7 days"  
"4h"
```

Data Statements

The data statements declare the fields in the data file. The fields must be declared in the order they appear in the data file. The format of most data statements is

```
type name;
```

where *type* is **int**, **float**, **double string**, **dataString**, **date**, and **fixedString(*n*)**, where *n* is an integer representing the width of the string; *name* is the variable name. Unlike in C, only one variable can be declared per statement.

A data field can also be based on an enumeration. The syntax is

```
enum enumName name;
```

The field must contain **ints** corresponding to the values of the enum. For example, if the enum `ageGroup` is declared as

```
enum string ageGroup {"below 30", "30-39", "40-49", "50-59",  
    "60 or above"};
```

the field `age` can be declared as

```
enum ageGroup age;
```

The field should contain **ints** between 0 and 4, where 0 is displayed as “below 30,” 1 as “30-39”, and so forth.

Only one variable can be declared per statement.

Input Options

All **options** statements begin with the word “options” and have one or more comma-separated options.

- The separator option defines the separator between fields in the data file. The default separator is a tab. The syntax is

```
options separator 'char';
```

For example:

```
options separator ':';
```

- The backslash option controls whether backslashes in the input data are treated specially or like other characters. The syntax is:

```
options backslash off;
```

```
options backslash on;
```

The default is off. If backslash processing is on, separators in the input data preceded by backslashes are treated as regular characters rather than separators. Within strings, this causes standard C-style backslash processing.

The View Section

The **view** section of a configuration file describes how the data is displayed, including the mapping of sizes, colors, axes, and so on. The default values for these options are in *view.splatviz.options*. The Windows options are in the directory in which MineSet is installed under *config\scatterviz\view.splatviz.options*. The IRIX options are in */usr/lib/MineSet/splatviz/view.splatviz.options*. Its form is

```
view
{
viewStatement;
...
}
```

A sample view section is

```
view {
  slider `age_bin`;
  opacity `count_gross_income`;
  color `avg_gross_income`;
  axis `education`, color "grey";
  axis `occupation`, color "grey";
  axis `hours_per_week_bin`, max 100, color "grey";
  options grid size 0 0 0;
  summary `count_gross_income`, color "red";
}
```

When entering the view section, the *viewDefaults* file is read in.

Slider Statement

The **slider** statement identifies an enum column to be used as a slider dimension. Its syntax is one of the following:

```
slider columnName;
```

The *columnName* name is declared in the input section. If this column contains nulls, the slider includes a beginning position corresponding to those null values.

There can be 0, 1, or 2 slider statements. The first slider statement applies to the horizontal slider, the second to the vertical slider. If there is no slider statement, the resulting display does not include animation.

Opacity Statement

In the Splat Visualizer, the opacity is based on counts, or more generally, record weights.

If a column is mapped to this requirement, it is used to weight each record (rather than using 1) when computing a value for the opacity. Thus, if you had a column with values for population, density, or the result of a count aggregation, you might want to map this column to the opacity (weight) requirement. If you had no such column, the requirement can be left unmapped, and a column of 1's is used by default.

The **opacity** statement describes how a field of data is mapped to the opacity of the splats. The **opacity** statement consists of a series of clauses, separated by commas:

```
opacity clause1, clause2,...
```

Alternatively, the clauses can be given in separate opacity statements.

The Opacity Variable

The first clause normally contains the name of a field to be mapped to opacity. The field must be of a number type (**int**, **float**, or **double**), of which **float** is the most efficient.

The Max Clause

The **max** clause allows you to alter the initial opacity setting for the scene. The most opaque splat in the scene will match the value specified in this **max** clause. The default is 1. The **max** clause has the form:

```
max float
```

The Legend Clause

The **legend** clause defines the meaning of the opacity mapping. The legend clause has the following forms:

- `legend off`

This turns off the opacity legend.

- `legend on`

This turns on the opacity legend (this is the default). The default legend is:

```
opacity:count
```

where *count* is a column that the tool has created by counting the number of records in each aggregate. If a column was mapped to opacity, the name of this column prepended with "sum_" is shown in the legend. This new column is computed by sum aggregating the column mapped to the opacity requirement.

- `legend label "string"`

This turns the legend on and explicitly sets the legend string. If this form is used, *legend on* is unnecessary.

Color Statement

The **color** statement describes how values are mapped to colors. The format is similar to the opacity statement, consisting of several clauses that can be separated by commas, or entered as multiple statements. The syntax is:

```
color clause1, clause2,...
```

Color Naming

Color names follow the conventions of the X window system, except that the names must be in quotes. Examples of valid colors are "green," "Hot Pink," and "#77ff42." The latter is in the form "*#rrggb*", in which the red, green, and blue components of the color are specified in hexadecimal value. Pure saturation is represented by ff, a lack of color by 00. For example, "#000000" is black, "#ffffff" is white, "#ff0000" is red, and "#00ffff" is cyan.

- Windows users should only use the "#ff0000" form, as only some of the named colors are supported (for example, *white, black, gray, red, yellow, green, etc.*).
- IRIX users can find the color list file at `/usr/lib/X11/rgb.txt`.

The Color Variable

As with opacity, you also can specify a column to be mapped to splat color. If the column is a number type, the scale and buckets clauses described below can be used to map a range of colors to the values of the field.

The colors Clause

The **colors** clause specifies the colors to be used. The colors clause's syntax is:

```
colors "colorname" "colorname" ...
```

The format for *colorname* is described in [“Color Naming” on page 158](#). Note that there are no commas between the colors, because commas are used to separate clauses in the color statement. A sample colors clause is:

```
colors "red" "gray" "blue"
```

Colors in the list are subsequently referred to by their index, starting at zero. In the above example, red is color 0, gray is color 1, and blue is color 2.

If there is no colors statement, colors are chosen randomly. If there is a colors statement, at least as many colors must be specified as are to be mapped.

The scale Clause

The **scale** clause allows assignment of values to a continuous range of colors. For example, when displaying a percentage, red can be assigned to 0%, gray to 50%, and blue to 100%. Intermediate values are interpolated; for example 25% is pinkish, and 55% is a slightly bluish gray.

The syntax for the scale clause is

```
scale float float ...
```

The first value is mapped to color 0, the second to color 1, and so forth. The colors statement must contain at least as many colors as are to be mapped to the largest index.

Values in this statement must be in increasing order. Any value less than the first color is assigned the value of the first color. Any value greater than the last value is assigned the last color. Intermediate values are interpolated.

For example, assume the `pctFemale` field indicates what percentage of the group is female, and you want to map a group that is 100% female to red, 100% male to blue, and 50% each to gray. The colors statement for this is:

```
colors pctFemale, colors "blue" "gray" "red", scale 0 50 100;
```

Use the `scale` clause only in conjunction with a numeric color variable.

The buckets Clause

The **buckets** clause is similar to the `scale` clause without interpolation. All values are rounded down to the highest value in the clause, and that exact color is used. Values less than the first value use the first color.

The syntax for the buckets clause is

```
buckets float float ...
```

The syntax and assignment of colors is the same as for the `scale` clause.

If, in the above example, you used the buckets clause instead of the `scale` clause, the statement would be:

```
colors pctFemale, colors "blue" "gray" "red", buckets 0 50 100;
```

All values greater than or equal to 100 are colored red. Values greater than, or equal to, 50 but less than 100, are gray. All other values are then blue.

Use the `buckets` clause only with a numeric color variable.

The legend Clause

The **legend** clause creates a legend of the colors. The legend clause syntax can be any of the following:

```
legend off  
legend on  
legend "string" "string" ...  
legend label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. It can be omitted if other legend statements are included. Specifying only **legend on** generates the default legend.

The default legend includes a single label to the left (with the name of the field that is mapped to color), and a list of colored labels on the right (with values obtained from the scale clause, the buckets clause, or from the field). To override the strings in the colored labels, specify the strings as:

```
legend "string" "string"
```

To override the label on the left, specify it following the word *label*. To eliminate this label, specify an empty string; that is:

```
legend label ""
```

Axis Statement

The **axis** statement causes a variable to be used as an axis in the 3D landscape. The variable's values determine where the entities are positioned on the axis. There can be up to three axis statements. Like the size and color statements, the axis statement contains a series of comma-separated clauses, but all of them must be specified in a single statement.

```
axis clause1, clause2,...
```

The Axis Variable

As with size and color, you can specify a field to be used as an axis. The field can be an array that is indexed by slider dimensions. If the field is an array, it must be of type **number**. If the field is not of type number, it is sorted, and each unique value is assigned a position along the axis.

The Label Clause

The **label** clause has the form:

```
label "string"
```

The string is used to label the axis. It appears in the landscape, at the end of the axis line. The default label is the name of the axis variable.

The Color Clause

The **color** clause specifies the color used for the axis line and label. It has the form:

```
color "colorname"
```

Summary Statement

The **summary** statement specifies aggregate information to be calculated for all data defined by the slider position. The summary is used to color the drawing window in the animation control panel. Like the opacity and color statements, the summary statement has several clauses that can be specified in one statement, separated by commas, or in separate statements.

```
summary clause1, clause2,...
```

The Summary Variable

You can specify the variable to be used in the summary. This variable must be of number type. If no summary variable is specified, *sum of counts* is used. If a variable is specified, then the weighted average of that variable (for all the data at the slider location) is used.

The Color Clause

The **color** clause specifies the color used to display the summary values in the drawing window. It has the form

```
color "colorname"
```

Various shades of the color, from white to the specified color, are used to represent summary values. The minimum summary value is mapped to white, while the maximum summary value is mapped to the specified color. The default summary color is red. If no slider variable is specified, this statement has no effect.

The Legend Clause

The **legend** clause creates a legend of the summary colors. The legend clause syntax can be any of the following:

```
legend off  
legend on  
legend label "string"
```

The **legend off** clause turns the legend off. The **legend on** clause turns the legend on. It can be omitted if other legend statements are included. Specifying only **legend on** generates the default legend.

The legend includes a single label to the left (which defaults to the aggregation function and variable used in the summary), and two colored labels on the right (with the minimum and maximum summary values). To override the label on the left, specify it following the word *label*. To eliminate this label, specify an empty string; that is

```
legend label ""
```

View Options

The **view** section of the configuration file has several options for controlling parameters of the display. These options can appear in a single options statement, separated by commas, or in separate options statements. The syntax of the options statement is

```
options option, option,...
```

The following options are available:

- `axis label size float`
controls the size of the axis labels.
- `background color "colorname"`
controls the initial color of the background..
- `hide label distance float`
controls the distance at which axis labels become invisible. Smaller distances might improve performance, but the labels disappear more quickly.
- `grid color "colorname"`
controls the color of the grid.
- `grid size float float float`
controls the spacing between grid lines. It applies the three values to grid lines along the x, y, and z axes, respectively.
- `orientation top right front float float float`
specifies the initial orientation of the scene. You may use three floating point values to specify an arbitrary orientation vector.

- `perspective` *on off*
specifies whether or not to use perspective initially. The default is to use it.
- `shape` *splatType*
specifies the type of splat used. The *shapeName* can be “constant,” “linear,” “gaussian”, “texture”, or “sphere.”

Creating Data and Configuration Files for the Decision Table Visualizer

This chapter describes the Decision Table Visualizer's data and configuration files. The **.dtbleviz* file contains a schema and optional history section, and the **.data* file contains the data. The format of these two files is almost exactly that described in [Chapter 5, "Flat File Support for MineSet."](#)

The Decision Table Visualizer's **.dtbleviz* file must contain a schema of the form: some number of columns of type string or enum; followed by a float column containing the weight of records; followed by a vector column, indexed by an enum containing possible classes, and containing the proportion of the weight of records in each class. The **.dtbleviz* and **.data* files are automatically generated by the Tool Manager, and you should not normally need to modify them.

There are two extra key-words that the schema in the **.dtbleviz* file uses that are not present in other schemas. They are "auto", and "nominal". The Decision Table Visualizer must be able to differentiate between different types of enums. For example, enums that have been automatically rendered discrete by mining are handled differently in drill-through, than enums that have been binned by the user in Tool Manager. Hence the keyword "auto" is used to distinguish columns that have been binned by mining.

The label may be a binned column, or it may be string column. Either way it appears as an enum in the schema, so that the final probs[] column can be indexed by it. If the label is string valued, the "enum" keyword is predicated with "nominal" to distinguish it.

Sample File

An example configuration file created from the adult dataset follows:

```
MineSet 3.0
input {
    file "adult-tmbin-dtab.dtableviz.data";
    enum string `hours_per_week_bin_k` {"- 20", "20-28", "28-36",
"36-44", "44-52", "52-60", "60+"};
    enum `hours_per_week_bin_k` `hours_per_week_bin`;
    auto enum string `gross_income_k` {"- 9598", "9598-14579",
"14579-24794.5", "24794.5-24806", "24806-29606", "29606-42049.5",
"42049.5-46306.5", "46306.5-64885", "64885+"};
    enum `gross_income_k` `gross_income`;
    auto enum string `final_weight_k` {"- 223033", "223033+"};
    enum `final_weight_k` `final_weight`;
    float `weight`;
    nominal enum string `label` {"Female ", "Male "};
# the label values
    float `probs[]`[ enum `label` ]; # and probabilities
}
history {
:
:
}
```

In this example, the name of the data file is *adult-tmbin-dtab.dtableviz.data*, and # denotes a comment line. The first three columns are attributes from the data. The first one, *hours_per_week_bin* has been binned by the user using Tool Manager. The second two, *gross_income*, and *final_weight* were binned automatically by the mining. Note the use of the **auto** keyword in their enum definition. The fourth column gives the weight of records having the values given by the first three columns.

The last column, *probs[]*, is a vector valued column which gives the probability of each class, computed by dividing the number of records for each class by the value in the weight column. The sum of the entries of this vector must add to 1. Note that the definition of the index label includes the keyword **nominal** to show that the class values are not numeric, and hence do not have an implied ordering. The **nominal** keyword would not be present had the label been a binned numeric attribute.

Format of the Evidence Visualizer's Data File

This chapter describes the Evidence Visualizer's input data file. This file is a textual representation of the Evidence classifier. The data file is generated automatically through the Tool Manager. In some instances one may wish to edit this file in order to alter label, attribute, or value names.

The Evidence Visualizer requires a data file containing the label and attributes, along with weights and probabilities. These are used to create the graphics. It is output as a result of running the Evidence Inducer through the Tool Manager. The format of the data file is:

```
#MineSet 3.0

<type> "<label>" <L>
"<label1>" <weight1> <probability1>
"<label2>" <weight2> <probability2>
:
"<labelL>" <weightL> <probabilityL>

<M>

<type> "<attrib1>" <N1> <importance1>
"<value1_1>" <weight1_1_1> <prob1_1_1> ... <weight1_1_L> <prob1_1_L>
"<value1_2>" <weight1_2_1> <prob1_2_1> ... <weight1_2_L> <prob1_2_L>
:
"<value1_N1>" <weight1_N1_1> <prob1_N1_1> ... <weight1_N1_L> <prob1_N1_L>

<type> "<attrib2>" <N2> <importance2>
"<value2_1>" <weight2_1_1> <prob2_1_1> ... <weight2_1_L> <prob2_1_L>
"<value2_2>" <weight2_2_1> <prob2_2_1> ... <weight2_2_L> <prob2_2_L>
:
"<value2_N2>" <weight2_N2_1> <prob2_N2_1> ... <weight2_N2_L> <prob2_N2_L>

:
:
:
```

```
"<attribM>" <NM> <importanceM>
"<valueM_1>" <weightM_1_1> <probM_1> ... <weightM_1_L> <probM_1_L>
"<valueM_1>" <weightM_2_1> <probM_2_2> ... <weightM_2_L> <probM_2_L>
:
"<valueM_NM>" <weightN1_NM_1> <probM_NM_1> ... <weightM_NM_L> <probM_NM_L>

history {
:
:
}
```

Where L is the number of label values, M is the number of attributes, and N is the number of values or bins for attribute i . The $\langle \rangle$'s indicate variables. The actual file has numbers or strings. A NULL is considered a unique value if it is present in an attribute. If NULLs exist for an attribute they always appear as the first value (i.e., the first line following the attribute header) and are represented by "?".

The $\langle \text{type} \rangle$ can be

- NOMINAL, which currently implies a string valued attribute (or an integer attribute which is used for the label).
- ENUM, which is used for attributes binned in Tool Manager.
- AUTO-ENUM, which is used for attributes that have been discretized automatically by the inducer. If a type is not present, AUTO-ENUM is assumed.

Lines beginning with # are comments (and ignored by the program).

An optional history section can be included at the end of the file. It is used by Tool Manager for drill-through. Without this section, drill-through is not possible (in the Evidence Visualizer or any other MineSet tool).

The *weights* are the number of records (or sum of weights) in the table with that particular attribute value (or range of values); hence, the sum of the weights for each attribute equals the total number of records in the table (unless record weighting was used). The *probability* is the number of weights for that attribute value divided by the total number of weights. If the data file was generated with *Laplace correction* turned on, the probability is only approximately the number of weights for that attribute value divided by the total number of weights. Thus, the probability value indicates the proportion of records with a particular label that have this attribute value instead of another value.

Data files must have a *.eviviz* extension. When starting the Evidence Visualizer, or when opening a file, you must specify the data file.

A sample Evidence Visualizer data file, *cars.eviviz*, follows. Windows users can find the file in the directory in which MineSet is installed under `\Examples\eviviz\examples`. IRIX users find the file in `/usr/lib/MineSet/eviviz/examples/cars.eviviz`.

```
#MineSet 3.0
#automatically generated
NOMINAL "origin" 3
"Europe" 73 0.179803
"Japan" 79 0.194581
"US" 254 0.625616
6
AUTO_ENUM "mpg" 5 25.448
"? " 3 0.0410959 0 0 5 0.019685
"- 16.1" 0 0 0 0 87 0.34252
"16.1-21.05" 10 0.136986 5 0.0632911 77 0.30315
"21.05-30.95" 43 0.589041 28 0.35443 67 0.26378
"30.95+" 17 0.232877 46 0.582278 18 0.0708661
NOMINAL "cylinders" 5 29.1759
"8" 0 0 0 0 108 0.425197
"4" 66 0.90411 69 0.873418 72 0.283465
"6" 4 0.0547945 6 0.0759494 74 0.291339
"3" 0 0 4 0.0506329 0 0
"5" 3 0.0410959 0 0 0 0
AUTO_ENUM "horsepower" 4 22.3514
"? " 2 0.0273973 0 0 4 0.015748
"- 78.5" 40 0.547945 46 0.582278 25 0.0984252
"78.5-134" 31 0.424658 33 0.417722 131 0.515748
"134+" 0 0 0 0 94 0.370079
AUTO_ENUM "weightlbs" 4 28.5157
"- 2379.5" 43 0.589041 57 0.721519 30 0.11811
"2379.5-2959.5" 18 0.246575 22 0.278481 57 0.224409
"2959.5-3274" 9 0.123288 0 0 29 0.114173
"3274+" 3 0.0410959 0 0 138 0.543307
AUTO_ENUM "time_to_60" 3 10.0055
"- 13.45" 3 0.0410959 3 0.0379747 78 0.307087
"13.45-19.45" 52 0.712329 75 0.949367 162 0.637795
"19.45+" 18 0.246575 1 0.0126582 14 0.0551181
AUTO_ENUM "year" 1 2.84217e-14
"ignore" 73 1 79 1 254 1
```

Note that the sum of the probabilities corresponding to a particular label value for a given attribute always equals 1. Consider the attribute *weightlbs*, for label value US (the first one), we have $.11811+.224409+.114173+.543307=1.0$. Also note that attributes *mpg* and *horsepower* have NULL values.

The *eviviz* datafile (**.eviviz*) also accommodates loss matrices and optionally included Laplace factor. The next example *.eviviz* file contains both of these.

```
MineSet 3.0
# MLC++ generated file for MineSet Evidence Visualizer.
NOMINAL "edibility" 2
"edible" 5 0.621212121212
"poisonous" 3 0.378787878788

TOTAL 8

LAPLACE yes 0

LOSS
10 0 20
10 10000 0

3

NOMINAL "cap-shape" 6 47.6129309656
"bell" 1 0.195652173913 0 0.03333333333333
"conical" 0 0.0217391304348 0 0.03333333333333
"convex" 2 0.369565217391 2 0.5666666666667
"flat" 2 0.369565217391 0 0.03333333333333
"knobbed" 0 0.0217391304348 1 0.3
"sunken" 0 0.0217391304348 0 0.03333333333333

NOMINAL "cap-surface" 4 27.8397591211
"fibrous" 2 0.386363636364 0 0.0357142857143
"grooves" 0 0.0227272727273 0 0.0357142857143
"scaly" 1 0.204545454545 2 0.607142857143
"smooth" 2 0.386363636364 1 0.321428571429

NOMINAL "cap-color" 10 47.6129309656
"brown" 1 0.18 1 0.264705882353
"buff" 0 0.02 0 0.0294117647059
"cinnamon" 0 0.02 0 0.0294117647059
"gray" 0 0.02 1 0.264705882353
"green" 0 0.02 0 0.0294117647059
"pink" 1 0.18 0 0.0294117647059
```

```
"purple" 0 0.02 0 0.0294117647059  
"red" 1 0.18 0 0.0294117647059  
"white" 1 0.18 1 0.264705882353  
"yellow" 1 0.18 0 0.0294117647059
```

The Laplace matrix follows the form shown in the GUI (see the description in the *MineSet 3.0 Enterprise Edition Reference Guide*).

Nulls in MineSet

Nulls represent unknown data. MineSet supports nulls in the data access tools, the mining tools, and the visualization tools. The purpose of this chapter is to give you a better understanding of the way MineSet handles nulls.

Semantics of Nulls

Unknown data values are often represented as nulls in data sources. While it is possible to associate different semantics with nulls, the most common is that nulls represent missing or unknown values. For example, if a data record is made up of fields representing *firstname*, *middlename*, *lastname*, and if a person's *middlename* is not known, it can be represented by the null value.

Nulls can occur in data for a variety of reasons: They can occur naturally in data as a means of representing unknown data, or they can come about as the result of doing certain kinds of aggregations. For example, if there are no flights between San Francisco and MineSet City, a query such as “find the average flight time from San Francisco to MineSet City” yields a null value.

MineSet generally follows the semantics of relational databases when dealing with nulls, and treats them as unknown values.

Some databases, such as Oracle RDBMS, do not distinguish between null and empty strings. In such a case, it is not possible to distinguish between an unknown middle name and a person who does not have a middle name. On the other hand, Sybase RDBMS distinguishes between null and empty strings. Hence MineSet can distinguish empty and null strings when reading from Sybase, but not from Oracle.

Representation of Nulls

In data files, as well as in the visual tools, nulls are represented by the string “?” (question mark). Thus, if Joe Miner’s middle name is unknown, his name is represented in our example data file (having schema *firstname, middlename, lastname*) as:

```
Joe      ?      Miner
```

In general, the color gray is often associated with null values in the visualizations. The graphical representation of nulls varies from tool to tool. See the chapters on the individual tools in the *MineSet 3.0 Enterprise Edition Reference Guide* for a discussion of how they represent them graphically.

Operations on Nulls

Given that nulls represent unknown values, it becomes straightforward to give meaning to expressions involving nulls.

Arithmetic Expressions

Arithmetic operations involving nulls always give a null result. For example:

$(5 + ?)$ evaluates to ? (adding 5 to an unknown yields yet another unknown);

$(6 / ?)$ evaluates to ?

Boolean Expressions

In addition to taking on the values of TRUE and FALSE, Boolean variables can also be null. If a Boolean valued variable has a null (unknown) value, the result of combining it with another Boolean variable in an expression is also unknown, unless it is possible to determine just from the known value what the result is. In particular:

"? AND FALSE is FALSE", because FALSE ANDed with anything is always FALSE

"? AND TRUE is ?"

"? OR FALSE is ?"

"? OR TRUE is TRUE", because TRUE ORed with anything is always TRUE

"NOT ? is ?"

Relational Operations

Relational operations (==, !=, <, >, <=, and >=) involving nulls always evaluate to null. Some particular cases worth emphasizing are:

"? == ?" evaluates to ?, not TRUE

"? != ?" evaluates to ?, not FALSE

"? != x" evaluates to ?, not FALSE

Given two unknown values, it is unknown whether the two are equal or unequal. This behavior can be confusing when using a search panel. For example, when searching for all values not equal to 0, nulls do not show up, yet neither do they show up when searching for values equal to 0. Because of this, search panels provide the ability to search explicitly for nulls. (Some search panels provide the option of treating nulls as zeros; see the individual tool discussions in the *MineSet 3.0 Enterprise Edition Reference Guide* for more information.)

Testing for Nulls

The function `isNull()` can determine whether or not a variable has the value null. For example:

`isNull(X)` evaluates to TRUE if variable *X* has the null value

`isNull(X)` evaluates to FALSE if variable *X* has a non-null value

Aggregations in the Presence of Nulls

MineSet stays close to the semantics of SQL and relational databases when aggregating columns that might have null values. Thus, null values are ignored when computing SUM, AVG, MIN, MAX, and COUNT. This is best illustrated by an example. Consider a data file having records representing the number of pets a person has. The schema of this record is *name*, *num_pets*, and null (unknown) values are represented by "?".

Name	NUM_PETS
Tesler	3
Rathmann	?
Haber	1
Bhargava	0
Sangudi	?

Then,

$$\text{SUM}(\text{NUM_PETS}) = 4$$

$$\text{COUNT}(\text{NUM_PETS}) = 3 \text{ (and not 5, even though there are 5 rows of data)}$$

$$\text{AVG}(\text{NUM_PETS}) = 1.33$$

$$\text{MAX}(\text{NUM_PETS}) = 3$$

$$\text{MIN}(\text{NUM_PETS}) = 0$$

In these aggregations, null values are basically ignored (note that the value 0 is different from ?, and is not ignored).

A special case of this is an aggregation where all the values being aggregated are themselves null. An even more specialized case is when there are no values being aggregated: for instance, when summing an empty column. In both these cases, the sum, average, min, and max are ?, while the count is 0.

Sort Order for Nulls

In an ascending sorted sequence, null values always appear before non-null values. In a descending sorted sequence, null values always appear after non-null values.

Bins and Arrays With Nulls

MineSet lets you bin numeric data into bins or discrete intervals. It also lets you (via the aggregation panel in the Tool Manager) create arrays on these bins. When a column of values is binned, all null values are put in a bin labeled "?". Such a bin label is always created, whether or not the data being binned has nulls in it. You have control over whether to use this bin for nulls in your application. You can do so by allowing arrays to ignore or keep bins for nulls by setting the desired option in the Tool Manager's Preferences dialog (on IRIX), or checking the "Use nulls in aggregation" checkbox at the bottom of the aggregation operator (on Windows.) For example, if you know that the column being binned has no nulls, or you intend to study the data corresponding to non-null values only, you can choose to ignore the bin for nulls.

ActiveX Visualization Control API for MineSet Visualizers

This chapter contains information for ActiveX application developers seeking to integrate MineSet visualization tools into their applications.

Overview

Included in the MineSet distribution are seven ActiveX controls that are automatically registered on your system when you install MineSet. Once they are registered on your system they can be browsed in the ActiveX test container in Microsoft Developer Studio.

Basics of Component Object Model

The Component Object Model, or COM provides an efficient methodology for modular application development. Developers can build stand-alone components (servers). Users, or clients of COM servers can use pre-built functionality in server objects without intimate knowledge of the server object during development. Previously, developers were required to link to a component's functionality at design time, or provide a path to the component in the source code. Instead, COM is now able to ask the registry for the object's location. This means that as long as the registry knows where to find the server object, your client application can use the object.

COM objects are either implemented within executables (EXEs) or within Dynamic Link Libraries (DLLs). COM objects implemented in EXEs are called local servers, while those implemented in DLLs are called in-process servers. An in-process COM server exposes its functions so that outside applications can use the functionality of its DLL. The visualization controls described in this chapter are of the latter type.

The COM library uses a class identifier (a CLSID value) to uniquely identify each COM object. An application then uses this identifier to determine which object the application wants to use. For example, suppose you create and register a COM object that provides functionality similar to the Microsoft Windows 95 tree-view control—showing, for instance, a Web-site hierarchy. Once you registered this object, other applications can call the functions you expose in it.

Because ActiveX controls are COM objects, this control is provided as an ActiveX control (with appropriate documentation), and can be integrated into developers' applications using a development environment and call methods, and properties can be set on the control.

ActiveX Architecture

ActiveX Controls have become the primary architecture for developing programmable software components for use in containers ranging from software development tools to end-user productivity tools. For a control to operate well in a variety of containers, it must be able to assume some minimum level of functionality that it can rely on in all containers. This minimum level of functionality is defined in the COM and ActiveX Control specifications, published by Microsoft Corporation. These guidelines have been rigorously followed in an attempt to make MineSet's visualization controls reliable and interoperable, and, ultimately, better and more usable components for building component-based solutions.

How MineSet's Visualization Controls Work

MineSet's visualization controls are comprised of seven separate ActiveX controls implemented as in-process DLLs. The only one you need to be concerned about, as a developer of a client application which will incorporate the MineSet visualizations, is the VizComposite control. The VizComposite control is an actual composite control that dynamically creates and manages the other six ActiveX controls based on the visualization file being loaded. A client application need only to create an instance of the VizComposite control, either at design-time or run-time, in order to embed MineSet visualization capabilities within their application. Once instantiated, you can call methods and set/get property values from the control to alter its behavior and to access certain functionality. These methods and properties are documented below.

Recommended Requirements

You can develop your client application using our visualization controls with any development environment, including Visual C++, Visual Basic, and Visual J++. While you can utilize Windows 95, Windows 98, or Windows NT as your development platform, we highly recommend developing on Windows NT 4.0 Workstation for any serious DCOM work. Using Windows NT, you can examine processes and shut down the system with a little bit more stability and ease than in Windows 95 or Windows 98.

If you are a novice at programming in COM, consider some entry-level books, such as:

- *Inside COM* by Dale Rogerson (Microsoft Press),
- *Understanding ActiveX and OLE* by David Chappell (Microsoft Press).

Additionally, some very good books on the details of COM and ActiveX/ATL include:

- *Essential COM* by Don Box (Addison Wesley),
- *Beginning ATL COM Programming* by Grimes and Stockton (Wrox Press),
- *Professional ATL COM Programming* by Richard Grimes (Wrox Press).

It is important to note that this document does not cover detailed semantics of the OLE interfaces; this is covered by the Microsoft Platform SDK documentation.

ActiveX Controls

The ActiveX controls names are in the form of `MineSet ControlName Control`, (for example, `MineSet VizComposite Control`.) The controls are:

- `VizComposite`—the only one of interest to the application developer.
- `VizAnimation`—use for the animation panel.
- `VizLegend`—use for the legend.
- `VizMain`—contains the 3D viewer.
- `VizPane`—auxiliary viewer used by some visualizers.
- `VizPlaneViewer`—viewer inside `VizPane`.
- `VizSelection`—shows pick/select information.

The VizComposite control contains and manages some or all the other controls as needed. This composite control implements several ActiveX interfaces.

The interface you obtain from the composite control should be one of the following:

- IVizCommon
- ITreeviz
- IMapviz
- IScatterviz
- ISplatviz
- IEviviz
- IDtableviz

IVizCommon contains methods that are common to all the other interfaces. Also, all the other interfaces include IVizCommon, so methods of IVizCommon may be called directly from any of the other interfaces.

After you have loaded a file into VizComposite, you can query the VizType property method to determine which of the specific tool interfaces you need to get, or you can just decide based on the configuration file extension.

The following describes all the interface methods and properties. Included in each description is the C++ style declaration which results when you load the control into Visual Studio and it becomes a class. If you load the control into a Java or Visual Basic environment, then the resulting class will have Java or Visual Basic style declarations instead.

Online documentation for the methods in each control may be viewed using **oleview**.

IVizCommon Methods

These methods and properties are common to all viz tools, although their behavior for a particular tool may vary. (In some rare cases, behavior has not yet been implemented for some tools, in which case E_NOTIMPL is returned. Otherwise S_OK is returned.)

The **File** property creates the right tool based on the configuration file extension. This is probably the single most important method. You can create an application using only this method. Before wrapping the `VizCompositeCtrl` in a class, this was the `File` property.

```
void SetFile(LPCTSTR fileName);
```

The **VizType** property returns an enum indicating the type of visualization tool. It returns `VIZ_UNKNOWN` if no configuration file has been opened yet. Other values are:

- `VIZ_MAP`
- `VIZ_SCATTER`
- `VIZ_SPLAT`
- `VIZ_EVI`
- `VIZ_DTABLE`
- `VIZ_TREE` (in the *VizCommon.idl* file this is the `VizType` property)

```
VIZCODE GetVizType();
```

The **Scale** property has a different meaning depending on the tool. In the Tree Visualizer, Evidence Visualizer, Decision Table Visualizer, and Map Visualizer, it scales the heights of objects in the scene. In the Scatter Visualizer it scales the size of the entities, in the Splat Visualizer it scales the opacity. For the MineSet Visualizer application the scale is adjusted by the slider on the left trim of the viewer.

```
void SetScale(float s);
```

ShowDecoration turns on or off (TRUE/FALSE) the Inventor window decoration based on the value of `on`.

```
void ShowDecoration(BOOL on);
```

ShowBackgroundColorDialog brings up the color selector dialog, so the user can select a desired background color.

```
void ShowBackgroundColorDialog();
```

The **BackgroundColor** property is used to explicitly set a specific background color. In the *.idl* file, this is the background color property. The parameters `r`, `g`, and `b` refer to the red, green and blue color components of the new background color.

```
void SetBackgroundColor(float r, float g, float b);
```

ShowFilterDialog brings up a filter panel of some type that can be used to reduce the amount of geometry in the scene. The Scatter Visualizer, Splat Visualizer, Decision Table Visualizer and Map Visualizer use the same style of filter dialog. The Tree Visualizer and Evidence Visualizer have their own variations.

```
void ShowFilterDialog();
```

DoIdle is needed to make Inventor run smoothly. It should be called in your application's OnIdle member function.

```
void Idle(long lCount);
```

For example:

```
BOOL CVizApp::OnIdle(LONG lCount) {
    if (m_bIsProcessing) {
        m_pVizView->m_vizCompositeCtrl.Idle(lCount);
    }
    return CWinApp::OnIdle(lCount);
}
```

where `m_bIsProcessing` gets set to true in the View's `OnInitialUpdate()`.

i.e.:

```
((CVizApp *)AfxGetApp())->m_bIsProcessing = TRUE;
```

ShowSelectionDialog brings up a table showing information about all the objects in the scene which have currently been selected. For the visualization application supplied with MineSet this method is called when the user selects "Show Values" from the "Selection" menu.

```
void ShowSelectionDialog();
```

The **LegendHeight** property sets the height of the legend control at the bottom of the visualization tool. If the `numLegends` argument is 3, then the height of the legend control will be enough to show 3 legends without the help of a vertical scrollbar.

```
void SetLegendHeight(short numLegends);
```

EnableSound turns the sound effects on or off (TRUE or FALSE respectively.)

```
void EnableSound(BOOL on);
```

The **ViewingMode** property refers to the inventor viewing mode: either VIEWING_PICK (viewer turned on) or VIEWING_GRASP (viewer turned off). When the viewer is turned on (pick mode), events are consumed by the viewer. When viewing is off, events are processed by the viewer's render area. This means events will be sent down to the scene graph for processing (that is, picking can occur).

In grasp mode, the cursor appears as a hand, and viewing transformations are performed. In pick mode, no navigation is possible, but you can pick and select objects in the scene. This methods has no meaning for Tree Visualizer, which does not use a modal viewer.

```
void SetViewingMode(VIEWINGCODE nNewValue);
VIEWINGCODE GetViewingMode();
```

The **SupportsAnimation** property returns TRUE if the visualization tool currently instantiated supports animation. In other words, returns TRUE if an animation slider or sliders are present. Some tools such as Evidence Visualizer, Decision Table Visualizer and Tree Visualizer never have animation in their current implementations, and hence always return FALSE.

```
BOOL GetSupportsAnimation();
```

The **SelectionExpression** property returns the selection expression which has been created based on the current selection of objects in the scene. This string is typically used to drill through to the underlying data in some manner, but there are other potential uses. If nothing has been selected the expression is the empty string (“”).

```
CString GetSelectionExpression();
```

The **HistoryString** property returns the history from the configuration file. This history contains everything that the Tool Manager needs to recreate its state at the time the current visualization was created.

```
CString GetHistoryString();
```

The **TitleString** property returns an appropriate application title comprised of the application name, and current configuration file base name.

```
CString GetTitleString();
```

EnableStereo turns on stereo rendering in the 3D viewer. It may be necessary to set hardware specific settings, and/or have certain peripherals like Crystal Eyes glasses before stereo is actually useful.

```
void EnableStereo(short on);
```

The **StereoOffset** property sets the amount of eye separation for the left and right stereo views.

```
void SetStereoOffset(BOOL offset);
```

The **SelectionsHeight** property sets the height (in pixels) of the selection control which appears at the top of all tools and displays pick information.

```
void SetSelectionsHeight(short height);
```

EnableWarnOnExecute specifies whether the user should be warned when a user-defined execute statement is invoked (this occurs when an object is double clicked and such an execute statement has been defined). The warning might be useful if you are concerned about security.

```
void EnableWarnOnExecute(short on);
```

EnableQuiet does not display all error and warning popup messages. In most cases this option should be off (FALSE). It is useful when running unit tests, not to have the popup messages appear. See also **SetErrorFile**.

```
void EnableQuiet(BOOL on);
```

The **ErrorFile** property specifies a file where all error messages should be sent. If an error file is set, error messages will still appear in dialogs on the screen. If you do not want the dialogs to appear on the screen you should use **EnableQuiet(TRUE)**. This method is primarily used for debugging and testing of applications made using the composite control.

```
void SetErrorFile(LPCSTR fileName);
```

The **HideDistance** property sets the distance at which level of detail is hidden. The most common use of this distance factor is in changing the level of detail on 3D text displayed in a scene. This option could also change detail thresholds in other circumstances. Sometimes the text has three or four levels of detail (hidden, bounding box, wire frame, full 3D), the distance parameter sets the first threshold, and the other thresholds are scalar multiples.

```
void SetHideDistance(float distance);
```

The **Null Style** property allows you to set how Null Values are displayed in the scene. Possible values are SOLID, HIDDEN and OUTLINE. Outline behaves the same as hidden for all tools except the Tree Visualizer. In the *VizCommon.idl* file there is a NullStyle property.

```
DRAW_STYLE SetNullStyle();
```

The **NullStyle** property returns current setting for how Null values are displayed.

```
void GetNullStyle(DRAW_STYLE style);
```

SetFont sets a desired font. This might be particularly useful for certain locales.

```
void SetFont(LPCSTR fontName);
```

SaveSceneGraph outputs the current scenegraph to a file in Inventor format. This *.iv* file cannot be loaded directly into another application, however, without having the right DLL's installed. This method is primarily used for debugging and testing.

```
void SaveSceneGraph(LPCSTR fileName);
```

SaveCompositeImage saves an image of the whole composite control to a file. An offscreen renderer is not used to save the scene, so a better choice might be **SaveViewerImage** if you want improved resolution.

```
void SaveCompositeImage(LPCSTR fileName, HWND hWnd = NULL);
```

SaveViewerImage saves the main viewer image to a file. An offscreen renderer is used for this, so the resolution is better. If the last three arguments are not specified (or are 0) then values are chosen based on the current screen size and resolution.

```
void SaveViewerImage(LPCSTR fileName)float desiredXinches= 0.f , float  
desiredYinches= 0.f , short printerDPI = 0 );
```

HasLegend returns TRUE or FALSE depending on whether or not a legend is present at the bottom.

```
BOOL HasLegend();
```

SaveLegendImage takes a snapshot of the legend control and save it to a file. This method is primarily used for debugging and testing.

```
void SaveLegendImage(LPCSTR fileName);
```

SaveDataTable stores the tools internal data table to a file. This method is primarily used for debugging and testing.

```
void SaveDataTable(LPCSTR fileName);
```

PrintCompositeImage prints an image of the window pointed to by hWnd to a file. An offscreen rendered is not used for saving the scene, so SaveViewerImage might be better for improved resolution. If hWnd is NULL, only the main rendering window image is saved.

```
void PrintCompositeImage(HDC hPrinterDC, HWND hWnd = NULL);
```

The **MouseButtonStyle** property is used to select the desired mouse mapping mode. Possible values are: MINESET_TWO_BUTTON_STYLE, MINESET_THREE_BUTTON_STYLE. Since many users do not have a three-button mouse, MineSet now allows for two-button operation. For those familiar with previous versions of MineSet on the IRIX platform, three-button style offers the same controls as before. The default, two-button style, is more consistent between tools and has less dependence on the use of modifier keys. However, no functionality is lost with either a three- or a two-button mouse. See the Navigation entry in the *MineSet 3.0 Enterprise Edition Reference Guide* for mouse button navigation details.

```
void SetMouseButtonStyle(MBUTTON_STYLE style);
```

The **CameraPosition** properties provide methods of manipulating the camera viewpoint. Normally the scene is interacted using the mouse, but an application may require alternative methods for setting or getting the viewpoint and orientation. Possible applications are setting up or running animations, saving or restoring interesting viewpoints, alternative input devices, and so on.

```
void SetCameraPosition(float x, float y, float z);  
void GetCameraPosition(float &x, float &y, float &z);
```

The **CameraOrientation** properties change camera orientation. In SetCameraOrientation, x,y,z are the components of the axis, and angle is the rotation of the camera around that axis of orientation in radians.

```
void SetCameraOrientation(float x, float y, float z, float angle);
```

In GetCameraOrientation, x,y,z are the components of the axis orientation, angle gives the orientation of the camera about that axis in radians.

```
void GetCameraOrientation(float &x, float &y, float &z, float &angle);
```

IScatterviz

The following methods are available to interactively modify the Scatter Visualizer.

ShowDrillThroughColumnDialog brings up a dialog that allows you to select which columns you want to be significant on drill through. In other words you use the dialog to specify which columns could potentially be in the selection Expression (see **GetSelectionExpression** above).

```
void ShowDrillThroughColumnDialog();
```

The **TrailType** properties specifies what type of trails to use if any. Possible values of TRAILCODE are:

- NO_TRAILS
- LINE_TRAILS
- FADE_OUT_TRAILS
- TUBE_TRAILS

```
void SetTrailType(TRAILCODE trailType);
```

ShowAnimationPanel indicates that if the argument is TRUE show the animation panel, otherwise hide it. If there is no animation panel, it cannot be shown.

```
void ShowAnimationPanel(BOOL on);
```

CreateBoxSelection creates a bounding box which can be used to select entities.

```
void CreateBoxSelection();
```

UseSliderOnDrillThrough specifies whether or not the animation slider position should be used in constructing the drill through expression when you drill through.

```
void UseSliderOnDrillThrough(short on);
```

The **2DSliderPositions** properties allows you to programmatically alter the slider position. If the slider is one dimensional then the second argument is ignored.

```
void Set2DSliderPositions(float slider1Pos, float slider2Pos );
```

The **Shape** property refers to the current entity shape type. Possible values for the Scatter Visualizer are:

- SHAPE_CUBE
- SHAPE_SPHERE
- SHAPE_DIAMOND
- SHAPE_BAR

```
void SetShape(SHAPECODE s);  
SHAPECODE GetShape();
```

EnableSpinAnimation specifies whether or not the automatic spin animation should be enabled in the viewer. If on is set to TRUE, then it is possible to set the animation continuously spinning by doing a rotation and releasing the left mouse before you are done dragging.

```
void EnableSpinAnimation(BOOL on);
```

The **LabelSize** property sets the scale size of object labels that appear in the scene. A bigger value of size corresponds to a bigger label.

```
void SetLabelSize(float size);
```

The **AxisLabelSize** property sets the scale size of axis labels that appear in the scene. This applies to the labels at the ends of the axes in Scatter and Splat Visualizers.

```
void SetAxisLabelSize(float size);
```

The **GridColor** property sets the color for the grid.

```
void SetGridColor(float r, float g, float b);
```

The **GridSize** property sets the grid spacing in each of the 3 dimensions. Bigger numbers correspond to wider grid line spacing, with the exception of 0 which means do not draw any grid lines in that axis dimension. If all 0's, then no grid is drawn.

```
void SetGridSize(float axis1, float axis2, float axis3);
```

The **Orientation** property sets which of three orthogonal orientations the view should be set to. Possible values are:

- ORIENTATION_FRONT
- ORIENTATION_RIGHT
- ORIENTATION_TOP
- ORIENTATION_DEFAULT

where ORIENTATION_DEFAULT uses the orientation that the viz tool thinks best.

```
void SetOrientation(ORIENTATIONCODE orientation);
```

EnablePerspective sets whether or not to use perspective in the 3D viewer. If TRUE then use perspective projection if FALSE then use orthogonal projection.

```
void EnablePerspective(BOOL on);
```

SaveAnimationPaneImage takes a snapshot of the animation control and saves it to a file. This method is primarily used for debugging and testing.

```
void SaveAnimationPaneImage(LPCSTR fileName);
```

ISplatviz

The following methods are available to interactively modify the Splat Visualizer:

ShowPickDragger If on=TRUE, the Splat Visualizer pick-dragger is shown, otherwise it is hidden.

```
void ShowPickDragger(BOOL on);
```

The **ShowingPickDragger()** property returns TRUE if the pick dragger is currently showing.

```
BOOL ShowingPickDragger();
```

ShowAnimationPanel indicates that if the argument is TRUE show the animation panel, otherwise hide it. If there is no animation panel, it cannot be shown.

```
void ShowAnimationPanel(BOOL on);
```

CreateBoxSelection creates a bounding box which can be used to select splats.

```
void CreateBoxSelection();
```

UseSliderOnDrillThrough specifies whether or not the animation slider position should be used in constructing the drill through expression when you drill through.

```
void UseSliderOnDrillThrough(BOOL on);
```

The **2DSliderPositions** property allows you to programmatically alter the slider position. If the slider is one dimensional then the second argument is ignored.

```
void Set2DSliderPositions(float slider1Pos, float slider2Pos);
```

The Shape property sets the shape type for splats:

- SHAPE_CUBE
- SHAPE_SPHERE
- SHAPE_DIAMOND
- SHAPE_BAR
- SHAPE_LINEAR
- SHAPE_GAUSSIAN
- SHAPE_TEXTURE

```
void SetShape(SHAPECODE s);
```

The **Shape** property returns the current shape type. This and the last method have no effect in Map Visualizer currently.

```
SHAPECODE GetShape();
```

EnableSpinAnimation specifies whether or not the automatic spin animation should be enabled in the viewer. If on is set to TRUE, then it is possible to set the animation continuously spinning by doing a rotation and releasing the left mouse before you are done dragging.

```
void EnableSpinAnimation(BOOL on);
```

The **AxisLabelSize** property sets the scale size of axis labels that appear in the scene. This applies to the labels at the ends of the axes in Scatter Visualizer and Splat Visualizer and similar tools.

```
void SetAxisLabelSize(float size);
```

The **GridColor** property sets the color for the grid.

```
void SetGridColor(float r, float g, float b);
```

The **GridSize** property sets the grid spacing in each of the 3 dimensions. Bigger numbers correspond to wider grid line spacing, with the exception of 0 which means do not draw any grid lines in that axis dimension. If all 0's, then no grid is drawn.

```
void SetGridSize(float axis1, float axis2, float axis3);
```

SetOrientation sets which of three orthogonal orientations the view should be set to. Possible values are:

- ORIENTATION_FRONT
- ORIENTATION_RIGHT
- ORIENTATION_TOP
- ORIENTATION_DEFAULT

where ORIENTATION_DEFAULT uses the orientation that the visualization tool thinks best.

```
void SetOrientation(ORIENTATIONCODE orientation);
```

EnablePerspective sets whether or not to use perspective in the 3D viewer. If TRUE then use perspective projection if FALSE then use orthogonal projection.

```
void EnablePerspective(short on);
```

SaveAnimationPaneImage takes a snapshot of the animation control and saves it to a file. This method is primarily used for debugging and testing.

```
void SaveAnimationPaneImage(LPCSTR fileName);
```

IMapviz

Methods relevant only for the Map Visualizer:

ShowXYCoords If on =TRUE, displays the XY Coordinate grid.

```
void ShowXYCoords(BOOL on);
```

UseRandomColors If on =TRUE, assigns a random color to each distinct geographical entity.

```
void UseRandomColors(BOOL on);
```

ShowAnimationPanel indicates that if the argument is TRUE show the animation panel, otherwise hide it. If there is no animation panel, it cannot be shown.

```
void ShowAnimationPanel(short on);
```

UseSliderOnDrillThrough specifies whether or not the animation slider position should be used in constructing the drill through expression when you drill through.

```
void UseSliderOnDrillThrough(short on);
```

Set2DSliderPositions allows you to programmatically alter the slider position. If the slider is one dimensional then the second argument is ignored.

```
void Set2DSliderPositions(float slider1Pos, float slider2Pos);
```

EnableSpinAnimation specifies whether or not the automatic spin animation should be enabled in the viewer. If on is set to TRUE, then it is possible to set the animation continuously spinning by doing a rotation and releasing the left mouse before you are done dragging.

```
void EnableSpinAnimation(short on);
```

EnablePerspective sets whether or not to use perspective in the 3D viewer. If TRUE, then use perspective projection. If FALSE, then use orthogonal projection.

```
void EnablePerspective(BOOL on);
```

SaveAnimationPaneImage takes a snapshot of the animation control and saves it to a file. This method is primarily used for debugging and testing.

```
void SaveAnimationPaneImage(LPCSTR fileName);
```

IEviz

These methods are relevant only for the Evidence Visualizer:

SetAttributeOrder allows you to programmatically change the way the attributes are listed in the evidence visualizer. Possible values are:

- ORDER_ALPHABETICALLY alphabetically is not implemented at the moment.
- ORDER_BY_IMPORTANCE—default
- ORDER_BY_DATABASE—use the database ordering

```
void SetAttributeOrder(ORDERCODE order);
```

SubtractMinEvidence whether or not to subtract the minimum amount of evidence from each value in the bars mode (see the *MineSet 3.0 Enterprise Edition Reference Guide*.)

```
void SubtractMinEvidence(BOOL on);
```

SelectAttributeValue programmatically selects one of the attribute values and resulting probabilities.

```
void SelectAttributeValue(LPCSTR attribute, LPCSTR value);
```

EnableEvidenceMode sets whether to show an evidence representation, or a probability representation. For the Evidence Visualizer, this means switching between showing the evidence cakes on the left, or the probability pies. See the *MineSet 3.0 Enterprise Edition Reference Guide* for additional information about these modes.

```
void EnableEvidenceMode(BOOL on);
```

The **EvidenceMode** property returns TRUE if the classifier visualizer is in the Evidence mode.

```
void GetEvidenceMode(BOOL on);
```

The **NominalOrder** property specifies how to order attribute values. Possible values are: ORDER_ALPHABETICALLY, ORDER_BY_WEIGHT, ORDER_BY_LABEL

```
void SetNominalOrder(ORDERCODE order);
```

The **ViewerType** property specifies whether to use the landscape style viewer or examiner viewer to display the scene. VIEWERCODE values are: LANDSCAPE.VIEWER, EXAMINER.VIEWER. Drilling up and down is not available in the landscape viewer mode.

```
void SetViewerType(VIEWERCODE type);
```

The **Laplace** property specifies whether or not to use Laplace correction. The second argument is the amount of Laplace correction to use. If the second argument is -1, then a default based on the data is used.

```
void SetLaplace(BOOL on, float value = -1.f);
```

SaveLabelPaneImage takes a snapshot of the label probability pane and saves it to a file. This method is primarily used for debugging and testing.

```
void SaveLabelPaneImage(LPCSTR fileName);
```

SelectLabelValue programmatically selects one of the label values. After this method has been invoked, one of the classes in the probability pane on the right should be selected.

```
void SelectLabelValue(LPCSTR value);
```

SetWeightThresh filters out objects with low weight. When the threshold is 0% all values are shown, no matter how small their weight; if the value is set to 100% all values are removed from the scene.

```
void SetWeightThresh(float percent);
```

SetDetailThresh changes the amount of detail in the scene. As the threshold is lowered, all attributes with lower importance values are removed from the scene.

```
void SetDetailThresh(float percent);
```

IDtableviz

EnableEvidenceMode sets whether to show an evidence representation, or a probability representation. For Evidence Visualizer, this means switching between showing the evidence cakes on the left, or the probability pies. In the Decision Table Visualizer, it determines whether the cakes show evidence (normalized conditional probabilities), or straight probabilities. See the *MineSet 3.0 Enterprise Edition Reference Guide* for additional information about these modes.

```
void EnableEvidenceMode(short on);
```

The **NominalOrder** property specifies how to order attribute values. Possible values are: ORDER_ALPHABETICALLY, ORDER_BY_WEIGHT, ORDER_BY_LABEL

```
void SetNominalOrder(ORDERCODE order);
```

The **ViewerType** property specifies whether to use the landscape style viewer or examiner viewer to display the scene. VIEWERCODE values are: LANDSCAPE.VIEWER, EXAMINER.VIEWER.

```
void SetViewerType(VIEWERCODE type);
```

The **Laplace** property specifies whether or not to use Laplace correction. The second argument is the amount of Laplace correction to use.

```
void SetLaplace(BOOL on, float value );
```

SaveLabelPaneImage takes a snapshot of the label probability pane and saves it to a file. This method is primarily used for debugging and testing.

```
void SaveLabelPaneImage(LPCSTR fileName);
```

SelectLabelValue programmatically selects one of the label values. After this method has been invoked, one of the classes in the probability pane on the right should be selected.

```
void SelectLabelValue(LPCSTR value);
```

The **WeightThresh** property filters out objects with low weight. When the threshold is 0% all values are shown, no matter how small their weight; if the value is set to 100% all values are removed from the scene.

```
void SetWeightThresh(float percent);
```

The **DetailThresh** property changes the amount of detail in the scene. As the threshold is increased, more detail is achieved through globally drilling down on all the cake charts until the maximum level of detail is reached at 100%.

```
void SetDetailThresh(float percent);
```

ITree Visualizer

Tree Visualizer specific methods.

ShowMarksDialog brings up a dialog which allows you to mark nodes in the tree with colored flags.

```
void ShowMarksDialog();
```

ShowOverviewDialog brings up a dialog which shows a two dimensional view of the entire tree.

```
void ShowOverviewDialog();
```

ShowSearchDialog brings up a dialog which lets you search for different attribute values for all the nodes in the tree hierarchy.

```
void ShowSearchDialog();
```

Navigate is the equivalent of the navigation buttons on the right-hand strip of the Tree Visualizer Viewer. TREE_GOCODE code can take on any of the following values:

- TREE_HOME—Resets the camera to the home position.
- TREE_SETHOME—Sets a new home position.
- TREE_VIEWALL—Positions the camera so the entire scene is in view.
- TREE_BACK—Go back one step in the navigation history. (undo)
- TREE_FORWARD—Go forward one step in the navigation history. (redo)
- TREE_PARENT—Update the selection to be the parent of the currently selected node, repositioning the camera to view it.
- TREE_LEFT—Update the selection to be the left sibling of current node.
- TREE_RIGHT—Update the selection to be the right sibling of current node.
- TREE_FIRST—Update selection to be the first (left) child of current node.
- TREE_LAST—Update selection to be the last (right) child of current node.

```
void Navigate(TREE_GOCODE code);
```

IsValidNavigation sets the variable 'enabled' to TRUE if the corresponding TREE_GOCODE is currently available. Otherwise sets it to FALSE.

```
void IsValidNavigation(TREE_GOCODE code, BOOL *enabled);
```

ShowBaseHeights toggles whether or not to show the base heights in the scene. If base heights are not shown, they appear simply as a flat box on the ground plane.

```
void ShowBaseHeights(BOOL on);
```

ShowMarksFlags sets whether or not to show the marks flags, if there are any.

```
void ShowMarksFlags(BOOL on);
```

SetZeroStyle allows you to set how zero values are displayed. DRAW_STYLE style can take one of the following values: SOLID, HIDDEN, or OUTLINE.

```
void SetZeroStyle(DRAW_STYLE style);
```

GetZeroStyle returns the current setting for how zero values are displayed using the variable DRAW_STYLE *style. This value will be either SOLID, HIDDEN, or OUTLINE.

```
void GetZeroStyle(DRAW_STYLE *style);
```

NormalizeSubtree recomputes the heights for all nodes in the tree using the currently selected node as the 1.0 value.

```
void NormalizeSubtree();
```

SaveInitialHierarchy is primarily used for debugging and testing.

```
void SaveInitialHierarchy(LPCSTR fileName);
```

SaveViewedHierarchy is primarily used for debugging and testing.

```
void SaveViewedHierarchy(LPCSTR fileName);
```

SaveOverviewImage saves a snapshot of the overview dialog image. This method is primarily used for debugging and testing.

```
void SaveOverviewImage(LPCSTR fileName);
```

SaveSearchPaths saves a list of all the currently shown search paths. This method is primarily used for debugging and testing.

```
void SaveSearchPaths(LPCSTR fileName);
```


Further Reading and Acknowledgments

Some datasets were taken from the UCI repository (Merz, C. J., and Murphy, P. M. (1996). UCI Repository of machine learning databases, Irvine, CA: University of California, Department of Information and Computer Science) found at <http://www.ics.uci.edu/~mlearn/MLRepository.html>.

Further Reading

Several papers describing the technology used in MineSet are available at <http://mineset.sgi.com/tech>.

An excellent, non-technical introduction to data mining techniques is:

- Michael Berry and Gordon Linoff. *Data Mining Techniques*. New York: John Wiley & Sons, 1997. ISBN 0-471-17980-9. See <http://www.data-miners.com/>.

A comparative study of data mining tools, including MineSet, was done by the Two Crows Corporation. It contains a good introduction to data mining.

- Two Crows Corporation. *Data Mining: Products, Applications & Technologies*. Ordering information is available at <http://www.twocrows.com>.

A paper describing MLC++, the underlying analytical engine used in MineSet, is described in:

- Kohavi, R., Sommerfield, D., Dougherty, J., *Data Mining using MLC++, a Machine Learning Library in C++*. International Journal of Artificial Intelligence Tools, Vol. 6, No. 4, 1997, p. 537-566. See <http://robotics.stanford.edu/users/ronnyk/>.

A general and easy-to-read introduction to machine learning is:

- Weiss, S. M., and C. A. Kulikowski. *Computer Systems that Learn*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1991.

A general comparison of algorithms and descriptions is provided in:

- Taylor, C., D. Michie, and D. Spiegelhalter. *Machine Learning, Neural and Statistical Classification*. Paramount Publishing International, 1994.

An easy-to-read introduction to decision tree induction is:

- Quinlan, J. R. *C4.5: Programs for Machine Learning*. Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1993.

An excellent book on decision trees from a statistical perspective is:

- Breiman, L., J. H. Friedman, R. A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

A good edited volume of machine learning techniques is:

- Dietterich, T. G. and J. W. Shavlik (Eds). *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990.

A summary of accuracy estimation techniques is given in:

- Kohavi, R. *A study of cross-validation and bootstrap for accuracy estimation and model selection*. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, edited by C. S. Mellish. Morgan Kaufmann Publishers, Inc., 1995. Available at <http://robotics.stanford.edu/users/ronnyk/>.

The following papers describes the decision table metaphor:

- Kohavi, R., *The Power of Decision Tables*. In The European Conference on Machine Learning, 1995.
- Becker, B. *Visualizing Decision Tables*. In IEEE's Proceedings of Information Visualization, 1998.

A good reference to a paper explaining that no classifier can be “best” is:

- Schaffer, C. *A conservation law for generalization performance*. In Machine Learning: Proceedings of the Eleventh International Conference, 259-265. Morgan Kaufmann Publishers, Inc., 1994. Available at <http://wwwcs.hunter.cuny.edu/faculty/schaffer/papers/list.html>.

Further Readings About Option Trees

MineSet uses an advanced version of the Option Trees described in:

- Ron Kohavi and Clayton Kunz. *Option Decision Trees with Majority Votes*. Machine Learning: Proceedings of the Fourteenth International Conference”, Morgan Kaufmann Publishers, Inc., 1997. (See <http://robotics.stanford.edu/users/ronnyk/>). The option trees used in MineSet average the predictions and do not simply vote them as described in this paper. Option Trees were first introduced by Wray Buntine in his thesis *A Theory of Learning Classification Rules*, 1992, School of Computing Science, University of Technology, Sydney.

Further Readings About the Evidence Inducer

The following paper describes the wrapper method used to select the features for the Evidence Classifier:

- Kohavi, R., Sommerfield, D. (1995). *Feature Subset Selection Using the Wrapper Model: Overfitting and Dynamic Search Space Topology*. The First International Conference on Knowledge Discovery and Data Mining, pp. 192-197. Available at: <http://robotics.stanford.edu/users/ronnyk/>.

An excellent introduction to the Evidence Classifier (Naive-Bayes) is:

- Kononenko, I. (1993). *Inductive and Bayesian Learning in Medical Diagnosis*. Applied Artificial Intelligence, pp. 7:317-337.

The following paper describes conditions under which the Evidence Inducer is optimal:

- Domingos, P., Pazzani, M., *Beyond Independence: Conditions for the Optimality of the Simple Bayes Classifier*. Machine Learning, Volume 29, No. 2/3, Nov/Dec 1997, pp. 103-130.

The following paper describes the use of the wrapper method in the Evidence Inducer:

- Kohavi, R., John, G., *Wrappers for Feature Subset Selection*. In Artificial Intelligence Journal, special issue on relevance, Vol. 97, Nos 1-2, pp. 273-324.

The following paper describes the Laplace correction option:

- Cestnik, B. (1990). *Estimating Probabilities: A Crucial Task in Machine Learning*. Proceedings of the Ninth European Conference on Artificial Intelligence, pp. 147-149.

The following paper describes the automatic Laplace correction used in MineSet:

- Kohavi R., Becker B., and Sommerfield D., *Improving Simple Bayes, European Conference on Machine Learning, 1997* (poster). Available at <http://robotics.stanford.edu/users/ronnyk/>.

The following paper describes the Evidence Classifier (Naive-Bayes):

- Langley, P., Iba, W., Thompson, K. (1992). *An Analysis of Bayesian Classifiers*. Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 223-228. Available at <http://www.isle.org/~langley/pubs.html>.

Simple Bayesian Classifier. To appear in *Lecture Notes in Computer Science: Issues in the Integration of Data Mining and Data Visualization*, Springer Verlag, 1998.

The following books describe the Evidence Classifier:

- Good, I. J. *The Estimation of Probabilities: An Essay on Modern Bayesian Methods*. MIT Press, 1965.
- Duda, R., Hart, P. *Pattern Classification and Scene Analysis*, Wiley, 1973.

The following paper shows that while the conditional independence assumption can be violated, the classification accuracy of the evidence classifier (called Simple Bayes in this paper) can be good:

- Domingos P., Pazzani M (1996). *Beyond Independence: Conditions for the Optimality of the Simple Bayesian Classifier*. Machine learning, Proceedings of the 13th International Conference (ICML '96), pp. 105-112. Available at <http://www.ics.uci.edu/~pedrod/>.

Further Readings About the Splat Visualizer

The following paper describes and provides further references for the technical details of the Splat Visualizer.

- Becker, Barry G, *Volume Rendering for Relational Data*, to appear in Proceedings of Information Visualization '97, IEEE Computer Society Press, Los Alamitos CA, October 19-24, 1997.

The following paper explains how to use Gaussian splats for volume rendering.

- Westover, Lee, *Footprint Evaluation for Volume Rendering* in Proceedings of SIGGRAPH '90, Vol. 24, No. 4, pages 367-376).

Acknowledgments

The iris database (described in) was originally used in Fisher, R. A. 1936. The use of multiple measurements in taxonomic problems. *Annals of Eugenics* 7(1):179-188. It is a classical problem in many statistical texts.

The breast cancer database was obtained from Dr. William H. Wolberg, L. Mangasarian, and W. H. Wolberg. Cancer diagnosis via linear programming. *SIAM News* 23(5):1 & 18. University of Wisconsin Hospitals, Madison, September 1990.

The data for the mushroom sample file comes from: *Audubon Society Field Guide to North American Mushrooms*. New York: Alfred A. Knopf, 1981.

The data on congressional voting was taken from the *Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, Volume XL*, Congressional Quarterly Inc.: Washington, D.C., 1985.

The adult dataset was derived from the US Census Bureau survey in 1994 (<http://www.census.gov/ftp/pub/DES/www/welcome.html>).

Index

Symbols

- # symbol (configuration files), 42, 49, 90, 120, 150
- % (percent) character, 69
- % symbol (configuration files)
 - enum statements, 95, 125, 153
 - message statements, 76, 107, 141
- " (double quote) vs. ' (single quote), 52
- ; symbol (configuration files), 49, 88
- > (greater than) symbol, 69
- ? character, 174
- \ (backslash) sequences, 42, 52, 90, 120, 150
- \ characters, 45
- \n sequence, 52
- } symbol (configuration files), 49, 88
- ' (single closing quotation) characters, 52

Numbers

- 2-dimensional arrays, 87, 97, 117
 - declaring, 124
- 3D charts, 137, 161
- 64-bit support, 2
 - system parameters, 3

A

- ActiveX, 179
- adding plug-ins, 6
- addresses, 39
- aggregate keyword, 62
- aggregation
 - bases, 63
 - hierarchies, 59, 62, 66, 68
 - null values and, 173, 176
- aligning fields in data files, 38
- alphabetical comparisons, 54
- AND operator, 92, 122
- animation control panel (Map Visualizer)
 - displaying dates, 95
- animation control panel (Scatter Visualizer)
 - summary window
 - coloring, 139
 - viewing dates, 125
- animation control panel (Splat Visualizer)
 - summary window
 - coloring, 162
 - viewing dates, 153
- animations, 3
- any keyword, 62, 63
- arithmetic functions, 93, 123
- arithmetic operators, 92, 122
 - null values and, 174

- arrays, 39-40, 47-48, 87, 117
 - declaring, 43, 47, 97, 124, 127
 - defining keys, 60, 62, 124, 128, 152
 - enumerating, 39-40
 - hierarchies and, 59, 62, 65
 - null values and, 40, 44, 87, 177
 - separators, 40
 - overriding, 44, 97, 128
 - zero values and, 47
 - ascending keyword, 60
 - ascending sort order
 - hierarchies, 65
 - keys, 60
 - null values, 177
 - aspect ratios, 79
 - attaching to servers, 8
 - avg keyword, 62, 63
 - null values and, 176
 - axes
 - assigning values, 137-138, 161
 - labeling, 137, 143, 161, 163
 - color options, 138, 162
 - normalizing, 138
 - scaling values, 138
 - zero values and, 138
 - axis keyword, 137, 161
 - axis statements, 137-138, 161-162
 - axis variable, 137, 161
- B**
- backslash characters, 45
 - backslash keyword, 129, 155
 - backslash sequences, 42, 52, 90, 120, 150
 - bars
 - color options
 - based on keys, 72
 - labels, 80
 - equalizing, 65
 - generating, 60
 - heights
 - adjusting, 67
 - normalizing, 67, 70
 - labeling, 75
 - colors, 80
 - size, 82
 - laying out, 78
 - null values and, 81
 - sorting, 60
 - zero values and, 80
- base color statements, 74
 - base height statements, 70
 - base keyword, 63, 74
 - base message statements, 76
- bases
 - aggregation and, 63
 - color options, 74
 - labels, 80
 - heights, 70
 - legends and, 69
 - labeling, 80
 - size, 82
 - null values and, 81
 - zero values and, 80
 - bibliography, 202
 - binary formats, 38
 - bins
 - null values and, 177
 - bitwise operators, 92, 122
 - blank fields, 38, 86, 116, 146
 - blank lines, 38, 86, 116, 146
 - Boolean expressions, 174
 - buckets keyword, 73, 105, 136, 160

C

- calculated columns, 57, 100, 129
- calendar quarters, 96, 126, 153
- character strings, 39, 87, 117, 147, 150
 - configuration files, 42, 52, 90, 120, 150
- charts
 - labeling, 137, 138, 143, 161, 162, 163
 - normalizing axes, 138
 - plotting values, 137-138, 161
 - zero values and, 138
- CLSID, 180
- color keyword, 132, 134, 158
- color mappings
 - Map Visualizer, 104
 - overriding, 105
 - Scatter Visualizer, 134-137
 - overriding, 135
 - Splat Visualizer, 158-161
 - overriding, 159
 - Tree Visualizer, 71-74
 - overriding, 73
- color names, 71, 104, 134, 158
- colors
 - bars
 - based on keys, 72
 - labels, 80
 - bases, 74
 - labels, 80
 - continuous ranges, 72, 135, 159
 - disks, 74
 - filling by key, 72
 - grids, 143, 163
 - ground, 78
 - labels, 132, 138, 162
 - bars, 80
 - bases, 80
 - legends, 73, 106, 136
 - displaying for, 160
 - nodes, 80
 - normalizing, 106
 - sky, 78
 - summary values, 139, 162
- colors keyword, 72, 104, 135, 159
- color statements
 - Map Visualizer, 104-107
 - Scatter Visualizer, 134-137
 - Splat Visualizer, 158-161
 - Tree Visualizer, 71-74
- color variable, 72, 104, 135, 159
- columns, 96, 126, 154
 - aggregation options, 62
 - calculated, 57, 100, 129
 - defining, 37, 45, 47
 - mapping to, 67, 103
- commas, adding to numbers, 76, 107, 141
- comments
 - configuration files, 42, 49, 90, 120, 150
 - data files, 38, 86, 116, 146
- comparing
 - locations, 48
 - regions, 48
 - strings, 93, 123
 - alphabetically, 54
 - dataString vs. string types, 39
- Component Object Model, 179
- configuration files, 8
 - comments, 42, 49, 90, 120, 150
 - data files and, 38
 - DataMover, 13-15
 - mandatory, 16
 - Map Visualizer, 88
 - naming variables, 41, 50, 89, 119, 149
 - option files and, 50
 - Scatter Visualizer, 118
 - formatting, 118
 - Splat Visualizer, 147
 - formatting, 147
 - Tree Visualizer, 49

configuring

- DataMover, 13-20
- Map Visualizer, 88-109
- Scatter Visualizer, 118-144, ??-164
- Splat Visualizer, 147
- Tree Visualizer, 49-83

connections, 8, 17

copying data files, 20

- count keyword, 62, 63
- null values and, 176

D

database mining tools, 3

database servers

- connecting to, 8

data exchanges, 29

data files, 37-40

- aligning fields, 38
- comments, 38, 86, 116, 146
- configuration files and, 38
- copying, 20
- Evidence Visualizer, 167
- Map Visualizer, 85-87
 - naming, 93
 - reading, 94
- null values and, 174
- pre-existing, 19
- reading, 42
- Scatter Visualizer, 115
 - naming, 123
 - reading, 124
- Splat Visualizer, 145-147
 - naming, 151
 - reading, 152
- Tree Visualizer, 47-48
 - naming, 55

.datamove files, 13

DataMover, 3, 13-20

- connecting to, 8
- pre-existing data files and, 19

datasets

- filtering, 68
- loading sample, 24
- SAS formats and, 29-32
- updating, 56, 98

data sources

- null values and, 173

data statements

- flat file support, 42
- Map Visualizer, 96-98
- Scatter Visualizer, 126-128
- Splat Visualizer, 154
- supported types, 42

dataString types, 39, 87, 117, 146

data types, 38

- Map Visualizer, 86-87, 93, 97
- Scatter Visualizer, 116, 123, 127
- Splat Visualizer, 146-147, 154
- Tree Visualizer, 54

dates, 95-96, 125-126, 153-154

- formatting, 95, 125, 153
- incrementing, 95, 125, 153

date types, 95, 125, 147, 153

days, 96, 126, 153

decimal points, 76, 107, 141

- double types, 38
- float types, 38

declaring

- arrays, 43, 47, 97, 124, 127
- data types, 97, 127, 154
- enumerations, 43
- keys, 60, 62, 124, 128, 152
 - sliders and, 102
- variables, 50, 89, 97, 127, 154

default directories, 49, 88, 118, 148

- defaults files, 49, 88, 118, 148
 - views, 101, 130, 156
- default sort order, 65
- descending keyword, 60
- descending sort order
 - hierarchies, 65
 - keys, 60
 - null values, 177
- disk color statements, 74
- disk height statements, 70
- disk keyword, 70, 74
- disks
 - color options, 74
 - heights, 70
 - legends and, 69
 - normalizing, 71
 - null values and, 81
 - zero values and, 80
- displaying
 - data
 - overhead projections, 78
 - entities, 143
 - hierarchies, 66, 79
 - labels, 132, 137, 161
 - messages
 - Map Visualizer, 107
 - Scatter Visualizer, 140
 - Tree Visualizer, 75
- display options
 - Scatter Visualizer, 143
 - Splat Visualizer, 163
 - Tree Visualizer, 77
- divide by zero errors, 57
- divide function, 93, 123
 - / operator vs., 57, 100, 130
- dm_config file, 18
- documentation, xiv
 - typographic conventions, xv

- double-precision floating-point
 - numbers, 38, 86, 116, 146
- double quotes vs. single quotes, 52
- double types, 38, 86, 116, 146
- .dtableviz file, 165
- Dynamic Link Libraries, 179

E

- empty strings, 47
 - null values vs., 173
- e notation, 38, 86, 116, 146
- entities, 131
 - displaying, 143
 - filtering, 142
 - labeling, 132, 143
 - legends, 134
 - selecting, 140
 - size, 133-134
- entity keyword, 131
- entity statements, 131-132
- entity variable, 131
- enumerated arrays, 39-40
 - declaring, 44
 - hierarchies and, 65
 - keys as, 60, 62
- enumerated values, 94, 124, 152
 - dates, 95, 125, 153
- enumerations
 - declaring, 43
 - sliders and, 131, 156
- enum keyword, 43, 94, 95, 124, 125, 152, 153
- enum statements, 94-96, 124-126, 152-154
- equality, 92, 122
- Evidence Visualizer, 167
 - history logs, 168
- .eviviz filename extensions, 169

- example files
 - loading, 24
- exceptions, 45
- exchanging data, 29
- execute keyword, 76
- execute statements
 - Map Visualizer, 108
 - Scatter Visualizer, 141
 - Tree Visualizer, 76
- executing shell commands, 76, 108, 141
- exponential notation, 38, 86, 116, 146
- expressions, 53, 92, 122
 - defining, 57, 100, 129
 - hierarchies and, 64
 - null values and, 174-175
- expressions keyword, 58, 100, 129
- expressions sections
 - Map Visualizer, 100
 - Scatter Visualizer, 129
 - Tree Visualizer, 57-58
- extend keyword, 138
- extension files (Web), 34

F

- fields, 85, 115, 145
 - aligning, 38
 - assigning colors, 72, 104, 135, 159
 - charts and, 137, 161
 - data files, 37, 45, 47
 - defining, 57, 100, 129
 - data type, 96, 126, 154
 - input sections, 41, 55
 - entity size and, 133-134
- field separators, 85, 115, 145
 - default, 38

- file_cache setting, 13
- file alteration monitor, 56, 99
- file keyword, 42, 94, 124, 152
- filenames
 - include statements, 51, 90, 120, 149
 - option files, 49
- files, including, 51, 90, 120, 149
- file statements, 42
- filtering
 - data, 68
 - entities, 142
- filter keyword, 68, 142
- filter statement, 142
- Find File dialog box, 20
- fiscal year quarters, 96, 126, 153
- fixed-sized arrays, 39, 87, 117
 - declaring, 43, 97, 124, 127
 - hierarchies and, 59
 - separators, 40
- fixed strings, 39, 87, 117, 147
- flat file support
 - data statements, 42
- floating-point numbers, 38, 86, 116, 146
- float types, 38, 86, 116, 146
- fonts, 80
- formats
 - configuration files, 49, 88, 118, 147
 - data files, 37, 47, 85, 115, 145, 167
 - numbers, 76, 107, 141
- format strings
 - dates/time, 95, 125, 153
 - messages, 75, 107, 140
- functions
 - adding, 5
- further readings, 202

G

geographic regions, 102
 displaying, 110
 legends, 104
 scaling, 103
 gfx files, 110-113
 graphs
 labeling, 137, 138, 143, 161, 162, 163
 normalizing axes, 138
 plotting values, 137-138, 161
 zero values and, 138
 greater than symbol (>), 69
 grids
 color options, 143, 163
 ground colors, 78

H

height keyword, 67
 height statements
 Map Visualizer, 103
 Tree Visualizer, 67-69
 hexadecimal color values, 71, 104, 134, 158
 hiding
 labels, 132
 hierarchies, 58
 aggregating, 62, 66, 68
 defining keys, 60, 65
 displaying, 66, 79
 getting descriptions, 77
 normalizing heights, 67
 populating, 62
 setting options, 65
 sorting, 64
 hierarchy files, 109-110
 hierarchy function, 54, 79

hierarchy keyword, 58
 hierarchy sections, 58-66
 key statements, 60-62
 levels statements, 59-60
 options, 65
 sort statements, 64
 hierarchy.treviz.options, 58
 history sections, 168
 horizontal sliders, 103, 131, 156
 hours, 96, 126, 153

I

Importing, 21
 include keyword, 51, 90, 120, 149
 include statements, 149
 Map Visualizer, 90
 Scatter Visualizer, 120
 Tree Visualizer, 51
 incrementing dates, 95, 125, 153
 incrementing numeric values, 94, 124, 152
 indexes
 color values and, 72, 73, 105, 135, 159
 defining keys, 60, 65, 124, 128, 152
 INFORMIX tables, 16, 17, 25
 loading, 27
 input keyword, 41, 55, 93, 123, 151
 input sections, 41-45
 Map Visualizer, 93-99
 data statements, 96-98
 enum statements, 94-96
 file statements, 94
 options, 98
 Scatter Visualizer, 123-128
 data statements, 126-128
 enum statements, 124-126
 file statements, 124
 options, 128

input sections (*continued*)

- Splat Visualizer, 151-155
 - data statements, 154
 - enum statements, 152-154
 - file statements, 152
 - options, 155
 - Tree Visualizer, 55-57
 - options, 56
- integers, 38, 86, 116, 146
- int types, 38, 86, 116, 146
- invoking
 - Tool Manager, 8
- isNull function, 175
- isSummary function, 54

K

- keep_classifier_files setting, 14
- keep_client_download setting, 14
- keep_client_upload setting, 14
- keep_mlc_input setting, 14
- key keyword, 60, 128
 - color statements and, 72
- keys
 - arrays, 60, 62, 65, 124, 128, 152
 - coloring bars and, 72
 - hierarchies, 60, 65
 - setting options, 51, 89
 - sliders, 102
- key statements, 60-62
- keywords, 52, 91, 121, 150, 165

L

- label keyword, 161
 - Scatter Visualizer, 132, 137
 - Tree Visualizer, 75

labels

- axes, 137, 138, 143, 161, 162, 163
 - bars, 75
 - colors, 80
 - size, 82
 - bases, 80
 - size, 82
 - color options, 132, 138, 162
 - bars, 80
 - bases, 80
 - entities, 132, 143
 - main windows, 101
 - nodes, 79
 - setting fonts, 80
 - size, 143
 - splats, 161
- label statements
 - Tree Visualizer, 75
- large memory support, 2
 - system parameters, 3
- large numbers, 38, 86, 116, 146
- legend keyword, 104, 106
 - Scatter Visualizer, 132, 134, 136, 139
 - Splat Visualizer, 158, 160, 162
 - Tree Visualizer, 69, 73
- legends
 - color values, 73, 106, 136, 160
 - entities, 132, 134
 - geographic regions, 104
 - height mappings, 69
 - splats, 158, 160, 162
 - summary, 139, 162
- levels keyword, 59
- levels statements, 59-60
- libraries, 18
- line breaks, 49
- loading example files, 24
- loading tables
 - sample, 25, 26, 27

locations
 comparing, 48
 lod options, 82

M

main windows
 Map Visualizer
 labeling, 101
 map keyword, 102
 mappings, 66
 entity size and, 133-134
 legends and, 69
 table columns, 67, 103
 Map Visualizer
 animation control panel
 displaying dates, 95
 color mappings, 104
 configuring, 88-109
 data files, 85-87
 naming, 93
 reading, 94
 data input, 85
 data types, 86-87, 93
 declaring, 97
 displaying data, 101
 gfx files and, 110
 keywords, 91
 main window
 labeling, 101
 mathematical functions, 93, 123
 max clause
 Scatter Visualizer, 133
 Splat Visualizer, 157
 syntax, 133, 157
 max keyword, 62, 63
 null values and, 176
 Scatter Visualizer, 138
 memory, 39, 87, 117, 147

message keyword, 75, 107, 140
 messages
 Map Visualizer, 107
 Scatter Visualizer, 140
 Tree Visualizer, 75
 message statements
 Map Visualizer, 107-108
 Scatter Visualizer, 140-141
 Tree Visualizer, 75-77
 MineSet, 1
 setting up, 11-19
 tools
 overview, 3-8
 mineset2sas command-line option, 29
 mineset2sas utility, 29
 running, 29
 startup options, 30
 MineSet mtr extension, 33, 35-36
 mining tools
 adding, 6
 min keyword, 62, 63
 null values and, 176
 minutes, 96, 126, 153
 missing data values, 173
 modulus function, 54, 93, 123
 monitor keyword, 56, 98
 months, 96, 126, 153
 mtr files, 35
 multiprocessor version, 2

N

-names command-line option, 32
 naming
 data files, 55, 93, 123, 151
 variables, 41, 50, 89, 119, 149
 keywords and, 52, 91, 121, 150

- nesting include statements, 51, 90, 120, 149
- network connections, 8
- new lines, 52
- nodata command-line option, 32
- nodes
 - distance between, 81
 - labeling, 79
 - line options, 80
 - populating, 66
- nolabel command-line option, 32
- normalize keyword, 68, 106
- normalizing axes values, 138
- normalizing colors, 106
- normalizing heights
 - bars, 67, 70
 - disks, 71
- normalizing trees, 48
- NOT operator, 92, 122
- null enumerated arrays, 40, 87
 - declaring, 44
- null values, 40, 173
 - arrays and, 40, 44, 87
 - binning, 177
 - defining, 174
 - display options, 81
 - empty strings vs., 173
 - in expressions, 174-175
 - sorting, 177
 - testing for, 175
- numbers, 38, 86, 116, 146
 - formatting, 76, 107, 141
 - incrementing, 94, 124, 152
 - sorting, 60

O

- objects
 - displaying messages
 - Map Visualizer, 107
 - Scatter Visualizer, 140
 - Tree Visualizer, 75
- one-dimensional arrays, 47, 87, 117
 - declaring, 127
- Opacity, 157
- opacity statement
 - Splat Visualizer, 157
 - syntax, 157
- opacity variable
 - Splat Visualizer, 157
- operators, 92, 122
- options files, 49
 - hierarchies, 58, 67
- options keyword, 45
 - Map Visualizer, 89, 98
 - Scatter Visualizer, 119, 128, 143
 - Splat Visualizer, 149, 155, 163
 - Tree Visualizer, 51, 56, 77
- options statements, 45, 49, 51, 89, 119, 149
 - defaults files and, 88, 118, 148
 - tokens and, 52, 91, 121, 150
 - views, 77, 143, 163
- Oracle tables, 16, 17, 25
 - loading, 25
- organizational hierarchies, 48
- organization option, 65
- OR operator, 92, 122
- Overview window (Tree Visualizer), 78

P

parallelization, 2

parameters

- mineset2sas command-line options, 30
- sas2mineset command-line options, 31

pathnames, 94, 124, 152

- data files, 42
- include files, 51, 90, 120, 149

percentages, 69

percent symbol (%) in configuration files

- enum statements, 95, 125, 153
- message statements, 76, 107, 141

plug-in capability, 5

Plug-in Ops button, 5

pound symbol (#) in configuration files, 42, 49, 90, 120, 150

pre-existing data files, 19

printed documentation, xiv

- typographic conventions, xv

printf manual page, 75, 107, 141

Q

quarters (calendar), 96, 126, 153

question mark (?), 174

R

random colors, 72, 105, 135, 159

raw data, 47, 85, 115, 145

reading

- arrays, 48
- strings, 39

README files, 25

references, 202

regions, comparing, 48

relational expressions, 54, 93, 123

- null values and, 175
- strings, 54, 93, 123

relative pathnames, 94, 124, 152

- data files, 42
- include files, 51, 90, 120, 149

root nodes

- labeling, 79

running shell commands, 76, 108, 141

S

sample files

- Decision Table, 166
- loading, 24

sas2mineset command-line option, 31

sas2mineset utility, 29

- running, 31

SAS datasets, 29-32

SAS executables, 29

scale keyword, 68, 103, 105

- Scatter Visualizer, 133, 135, 138
- Splat Visualizer, 159
- Tree Visualizer, 72

scaling

- axes values, 138
- colors, 72, 105, 135, 159
- entities, 133
- geographic regions, 103

Scatter Visualizer

- animation control panel

 - summary window, 139
 - viewing dates, 125

- color mappings, 134-137
- configuring, 118-144, ??-164

Scatter Visualizer (*continued*)

- data files, 115
 - naming, 123
 - reading, 124
- data input, 115
- data types, 116, 123
 - declaring, 127
- displaying data, 130
- filtering data, 142
- keywords, 121
- max clause, 133
 - syntax, 133
- .schema files, 20, 41-45
- search paths
 - data files, 42, 94, 124, 152
 - defaults files, 118, 148
 - include files, 51, 90, 120, 149
 - options files, 49, 88
- seconds, 96, 126, 153
- sections (configuration files), 49, 88, 118, 148
- selecting entities, 140
- semicolons (;) in configuration files, 49, 88
- separator keyword, 44, 45, 155
 - Map Visualiser, 97, 98
 - Scatter Visualizer, 128
- separators, 98
 - arrays, 40
 - overriding, 44, 97, 128
 - data files, 37, 45, 47, 85, 115, 145
 - default character, 38
 - fields, 85, 115, 145
 - numeric formats, 76, 107, 141
- server connections, 17
- servers
 - connecting to, 8
- shared libraries, 18
- shell commands, 76, 108, 141
- Show Data Points command
 - specifying initial settings, 109
- shrinking aspect ratios, 79
- signed integers, 38, 86, 116, 146
- sinclude keyword, 51, 90, 120, 150
- sinclude statements, 150
 - Map Visualizer, 90
 - Scatter Visualizer, 120
 - Tree Visualizer, 51
- single closing quotation marks, 52
- single quotes vs. double quotes, 52
- size keyword, 133
- size statements, 133-134
- size variable, 133
- skipMissing option, 65, 66
- sky colors, 78
- slider controls
 - Map Visualizer
 - declaring, 97
- slider keyword, 102, 131, 156
- sliders
 - assigning keys, 102
 - defining dimensions, 102, 117, 123, 131, 156
- sorting, 60
 - hierarchies, 64
- sort keyword, 65
- sort order, 60
 - null values, 177
- sort statements, 64
- Spat Visualizer
 - opacity variable, 157
- splats
 - labeling, 161
 - legends, 158, 160, 162
- Splat Visualizer
 - animation control panel
 - summary window, 162
 - viewing dates, 153
 - color mappings, 158-161
 - configuring, 147

Splat Visualizer (*continued*)

- data files, 145-147
 - naming, 151
 - reading, 152
- data input, 145
- data types, 146-147
 - declaring, 154
- displaying data, 155
- keywords, 150
- max clause, 157
 - syntax, 157
- opacity statement, 157
 - syntax, 157
- starting
 - Tool Manager, 8
- statements (configuration files), 50, 89, 119, 148
- storing
 - strings, 39, 87, 117, 147
- string function, 54
- strings, 39, 87, 117, 146, 150
 - comparing, 93, 123
 - alphabetically, 54
 - dataString vs. string types, 39
 - configuration files, 42, 52, 90, 120, 150
 - empty, 47, 173
 - hierarchies and, 54
 - sorting, 60
 - storing, 39, 87, 117, 147
 - zero values and, 47
- string types, 39, 87, 117, 147
- sum keyword, 62, 63
 - null values and, 176
- summary keyword, 109, 139, 162
- summary legends, 139, 162
- summary statements
 - Map Visualizer, 109
 - Scatter Visualizer, 139-140
 - Splat Visualizer, 162-163
- summary values
 - color options, 139, 162
 - hierarchies, 62
- summary variable, 139, 162
- svsc command-line option, 31, 32
- Sybase tables, 16, 17, 25
 - loading, 26
 - shared libraries and, 18
- syntax (configuration files), 49, 50, 88, 118, 147
 - axis statements, 137, 161
 - base color statements, 74
 - base height statements, 70
 - color statements, 71, 104, 134, 158
 - buckets clause, 73, 105, 136, 160
 - colors clause, 72, 104, 135, 159
 - key clause, 72
 - legend clause, 73, 106, 136, 160
 - normalize clause, 106
 - scale clause, 72, 105, 135, 159
 - disk color statements, 74
 - disk height statements, 70
 - entity statements, 131
 - enum statements, 124, 152
 - expressions sections, 58, 100, 129
 - filter statements, 142
 - height statements, 67, 103
 - filter clause, 68
 - legend clause, 69, 104
 - normalize clause, 68
 - scale clause, 68, 103
 - hierarchy sections, 58, 65
 - aggregate statements, 62, 63
 - key statements, 60
 - levels statements, 59
 - sort statements, 65
 - include statements, 51, 90, 120, 149
 - input sections, 41, 55, 93, 123, 151
 - data statements, 96, 126, 154
 - enum statements, 94, 95, 125, 153
 - file statements, 42, 94, 124, 152
 - options, 98

syntax (configuration files) (*continued*)
 label statements, 75
 message statements, 75, 107, 140
 execute clause, 76
 options statements, 51, 89, 98, 119, 149
 input sections, 45, 56
 sinclude statements, 51, 90, 120, 150
 size statements, 133
 summary statements, 109, 139, 162
 view sections, 66, 101, 130, 155
 map statements, 102
 options, 78, 79, 80, 81, 143, 163
 slider statements, 102, 131, 156
 title statements, 101
syntax (data files), 38, 85, 115, 145, 167
system defaults, 49, 88, 118, 148
systune parameters
 64-bit support and, 3
 rlimit_nofile_cur, 3
 rlimit_rss_cur, 3
 rlimit_vmem_cur, 3
 rlimit_pthread_cur, 3

T

tab character, 38
tables
 hierarchies and, 59, 65
 loading
 sample, 25, 26, 27
 mapping to columns, 67, 103
 variable-length, 65
three-dimensional charts, 137, 161
time, 95-96, 125-126, 153-154
 formatting, 95, 125, 153

timeout options, 56
title keyword, 101
tokens, 52, 91, 121, 150
Tool Manager, 4
 starting, 8
tools
 overview, 3-8
transforming plug-ins, 5
Tree Visualizer
 color mappings, 71-74
 configuring, 49-83
 data files, 47-48
 naming, 55
 data input, 47
 data types, 54
 displaying hierarchies, 66, 79
 filtering data, 68
 keywords, 52
 manipulating views, 81, 82
two-dimensional arrays, 87, 97, 117
 declaring, 124, 127
type casting, 54, 93, 123
typographic conventions, xv

U

unknown data values, 173
updating data, 56, 98
usa.states.hierarchy, 109
use_ascii_mlc_input, 14
usr/lib/MineSet/datamove, 13
usr/lib/MineSet/mapviz, 88
usr/lib/MineSet/scatterviz, 118
usr/lib/MineSet/splatviz, 148
usr/lib/MineSet/treeviz, 49

V

- variable-length arrays, 47-48
 - hierarchies and, 59, 62
 - separators, 40
- variables, 49
 - axes values and, 137, 161
 - axis, 137, 161
 - color, 72, 104, 135, 159
 - declaring, 50, 89, 97, 127, 154
 - entity, 131
 - naming, 41, 50, 89, 119, 149
 - keywords and, 52, 91, 121, 150
 - null values and, 175
 - size, 133
 - summary, 139, 162
- variants, 40
- vertical sliders, 103, 131, 156
- viewHierarchyLandscape.treeviz.options, 67
- view keyword, 66
- viewMap.mapviz.options, 101
- viewpoints
 - manipulating, 81, 82
- views
 - Scatter Visualizer
 - display options, 143
 - Splat Visualizer
 - display options, 163
 - Tree Visualizer, 66
 - display options, 77
 - overhead projections, 78
- view.scatterviz.options, 130
- view sections
 - Map Visualizer, 101-109
 - color statements, 104-107
 - execute statements, 108
 - height statements, 103
 - map statements, 102
 - message statements, 107-108

- slider statements, 102
 - title statements, 101
- Scatter Visualizer, 130-143
 - axis statements, 137-138
 - color statements, 134-137
 - entity statements, 131-132
 - execute statements, 141
 - message statements, 140-141
 - options, 143
 - size statements, 133-134
 - slider statements, 131
 - summary statements, 139-140
- Splat Visualizer, 155-163
 - axis statements, 161-162
 - color statements, 158-161
 - opacity statements, 157-158
 - options, 163
 - slider statements, 156
 - summary statements, 162-163
- Tree Visualizer, 66-83
 - base color statements, 74
 - base height statements, 70
 - color statements, 71-74
 - disk color statements, 74
 - disk height statements, 70
 - height statements, 67-69
 - label statements, 75
 - message statements, 75-77
 - options, 77-83
- view.splatviz.options, 155
- visualization tools, 4

W

- Web environments, 33-36
 - client installation, 35
 - extension files, 34
 - overview, 33

X

xconfirm command, 77, 108, 141

Y

years, 96, 126, 153

Z

zero values, 47
 display options, 80
 graphing, 138
 returning, 54