ConvexAVS

First Edition

CONVEX

CONVEX COMPUTER CORPORATION

ConvexAVS

Order No. DSW-300

First Edition
March 1991

CONVEX Computer Corporation
Richardson, Texas USA

ConvexAVS

Order No. DSW-300

Copyright 1991 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

All or some portion of this document has been reprinted with the permission of Stardent Computer Inc. Copyright 1989 by Stardent Computer Inc.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation (CONVEX) does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

Unless provided otherwise in writing with CONVEX Computer Corporation (CONVEX), the program described herein is provided as is without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Some states do not allow the exclusion of implied warranties. The above exclusion may not be applicable to all purchasers because warranty rights can vary from state to state. In no event will CONVEX be liable to anyone for special, collateral, incidental or consequential damages, including any lost profits or lost savings, arising out of the use or inability to use this program. CONVEX will not be liable even if it has been notified of the possibility of such damage by the purchaser or any third party.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
ConvexAVS and ConvexOS are trademarks of CONVEX Computer Corporation.
STARDENT is a trademark of Stardent Computer Inc.
Iris and Silicon Graphics are trademarks of Silicon Graphics, Inc.
LaserWriter is a trademark of Apple Computer, Inc.
Mathematica is a trademark of Wolfram Research, Inc.
NeWS is a trademark of Sun Microsystems, Inc.
PostScript is a trademark of Adobe Systems, Inc.
X Window System is a trademark of M.I.T.

AVS was created and developed by, and is a trademark of, Stardent Computer, Inc.

Printed in the United States of America

Acknowledgements

The authors wish to thank all the people at CONVEX and the users who contributed their ideas and time in the development of this book. In particular:

- The team who developed ConvexAVS and made this book possible: Rick Franklin, Steve Gardner, Chris Goodman, Pete Levinthal, and Bob Miller.
- The team who created tests to pinpoint conflicts in the functionality of ConvexAVS and helped affirm the accuracy of this book: Randy Hall and Robert Mingee.
- The people who provided vision and management for the product and documentation: Dave Holt, Duane Gustavus, Kathy Harris, Mike Turner, Paul Lambert, and Frank Marshall.
- For their assistance in creating and improving the online tutorials and online help system: Dan Connolly, Tom Christiansen, and Phil Barr.
- For her editorial comments that help make the book more consistent and readable: Robin Sowton.

—Neal W. Johnston and Henry C. Smith II

Revision Information for

ConvexAVS

Edition	Document No.	Description
First	710-012830-001	Initially released with ConvexAVS V1.0, March 1991.

Contents

Preface

Purpose and Audience xvii
Organization xvii
Notational Conventions xviii
Command Syntax xviii
General Conventions xviii
Associated Documents xviv
Ordering Documentation xviv
Technical Assistance xx
The `contact` Utility xx

Using ConvexAVS 1

What Is ConvexAVS?

Introduction 3
Scientific and Engineering Data 5
Other Data Formats 6
Pixel-Based Visualization 6
Colormap Lookup 7
Further Pixel Processing 7
High-Quality Pixel-Based Visualization 7
Geometry-Based Visualization 8
ConvexAVS Subsystems 9
Image Viewer 9
Volume Viewer 9
Geometry Viewer 9
Network Editor 9
ConvexAVS Modules 10
Modules: Ports and Parameters 10
Data Inputs 11
Input Parameters 12
Widget Types 13

Data Outputs 13
Subroutine and Coroutine Modules 13
Module Libraries 14
User-Written Modules 14
ConvexAVS Networks 15
Data Flow in an ConvexAVS Network 15
Network Control Panel 17
ConvexAVS Display Windows 18
Preparing Your Data for Use With ConvexAVS 18
Supported File Formats 19
ConvexAVS-Specific File Formats 19
Geometry 19
Image 20
Field 20
Volume Data 20
Geometry Data File Format 20
Image Data File Format 20
Field Data File Format 21
Volume Data File Format 29
Converting Your Data to a ConvexAVS Format 30
Writing Your Own Conversion Utility 30
Writing a ConvexAVS Module 31
Creating a Data Directory 31

Starting ConvexAVS

Requirements and Setup Information 33
PseudoColor X Servers 33
Environment Variables 34
Command and Command-Line Options 35
ConvexAVS Startup File 38
The Main Menu 40
Switching Among the Subsystems 41
On-line Help 41

Image Viewer Subsystem

Features 43
Using the Image Viewer 44
Entering and Leaving the Image Viewer 45
Using the Data Preprocessors 46
Selecting Visualization Techniques 48
Lookup Table Techniques 49
Image Processing Techniques 49
Running the Network 50
Working with the Display Window 52
Deactivating or Restarting a Technique 53
Using More Than One Technique 53

Duplicating a Technique 54
Additional Image Viewer Features 55

Volume Viewer Subsystem

Features 57
Using the Volume Viewer 58
Entering and Leaving the Volume Viewer 59
Selecting Visualization Techniques 60
    Imaging Techniques 60
    Geometric Techniques 61
    Volume Rendering 61
    Data Preprocessing 62
Running the Network 63
    Working with the Display Window 65
    Deactivating or Restarting a Technique 65
Using More Than One Technique 66
Duplicating a Technique 66
Additional Volume Viewer Features 67

Geometry Viewer Subsystem

Entering and Leaving the Geometry Viewer 69
Using the Geometry Viewer 72
    Scenes: Objects, Lights, Cameras 72
    Data Formats 72
Menu Choices, Sliders, and Function Keys 75
Transformations and the Transform Selection Area 76
    Transforming Objects 77
    Transforming Lights 78
    Transforming Cameras 79
    Recovering Transformations 80
    Function Key Usage 80
Geometry Viewer Menu 81
    Objects 82
        Read Object 82
        Scroll Bars 83
        Save Object 85
        Delete Object 85
        Edit Property 85
        Object Info 89
        Show Object/Hide Object 89
    Rendering Methods 90
    Lights 91
    Cameras 94
    Labels 97
        Creating Labels 97
        Labeling the Top-Level Object 98

Picking and Moving a Label 98
Attaching a Label to a Vertex 99
Making a Label Into a Title 99
Editing a Label 99
Label Menu Selection 99
Font Selection Submenu 99
Label Attributes Submenu 100
Action 101
Adding Frames 101
Playing Back the Frames 103
Deleting Frames 103

Network Editor Subsystem

Starting the Network Editor 105
Closing the Network Editor 106
Switching to the Geometry Viewer 107
Overview of Network Editor Usage 107
Using the Module Palette and the Workspace 108
Data Input Modules 108
Filter Modules 108
Mapper Modules 108
Renderer Modules 108
Moving Icons into the Workspace—Left Button 110
Moving Modules within the Workspace 111
Deleting Modules from the Workspace 111
Connecting Modules—Middle Button 111
Disconnecting Modules— Right Button 115
The Module Editor and Parameter Editor Windows 116
Show Module Documentation 116
Disable Module 116
The Parameter Editor 116
Finding the Module You Want 117
Scrolling a Module List 117
Incremental Search Through a Module Category 117
Changing the Construction Window 118
Module Libraries 118
Completing a Network 119
Controlling the Execution of a Network 119
Using Control Widgets 121
Using Type-In Controls 121
Using Dial Controls 121
The Dial Editor 122
Using Slider Controls 123
Using a Set of Choices (Radio Buttons) 123
Using Toggle Controls 124
Using Tri-state Controls 124
Using One-shot Controls 124

Using File Browser Controls 125
Using the Colormap Control 126
Organizing a Network's Display Windows 128
Picture Size and Window Size 128
Images 128
Pixmaps 129
Using the Window Manager 129
Using the Network Editor Menu System 130
Network Tools 131
Read Network 131
Write Network 131
Clear Network 131
Print Network 131
Disable Flow Executive (toggle) 132
Save Parameters 132
Restore Parameters 132
Module Tools 132
Read Module(s) 132
Read Module Library 132
Write Module Library 133
Select Module Library 133
Flash Active Modules (toggle) 133
Verbose Mode (toggle) 133
Layout Editor 134
Elements of a Layout 134
Working with the Layout Editor 135
Including Display Windows 137

Developing Applications 139

Fundamentals

Modules 141
Data Types 142
Networks 142
Data Flow 142

Data Types

Primitive and Aggregate 145
Byte 147
Integer 147
Real 148
String 148
Field 148
Mapping Computational Space to Coordinate Space 148

Uniform Fields 149
Rectilinear Fields 149
Irregular Fields 149
Mapping Information 150
Examples of Field Mappings 150
Example 1 - Uniform Field 151
Example 2 - Uniform Field 151
Example 3 - Rectilinear Field 151
Example 4 - Rectilinear or Irregular Field 152
Example 5 - Irregular Field 153
Example 6 - Irregular Field 155
Field Components 155
Declaring Fields 158
Manipulating Fields from C 158
Manipulating Fields from FORTRAN 160
Creating Fields 162
Scatter Data 162
Image Data 163
Volume Data 163
Colormap 163
Geometry 165
What Is an Edit List? 165
Why Use Edit Lists? 165
How an Edit List Is Used 166
Design Factors 166
Manipulating Edit Lists 168
Pixel Map 169

Modules

Components 171
Name 171
Type 171
Data 171
Filter 171
Mapper 172
Renderer 172
Ports 172
Parameters 173
Functions 174
Description Function 174
Computation Function 177
Subroutines and Coroutines 178
Subroutine Modules 179
Coroutine Modules 180
Module Examples 180
Handling Errors 180
Creating Online Help 182

Selective Computation 182
Building and Linking Modules 183
Include Files 183
C Language Include Files 183
FORTRAN Include Files 183
Compiling and Linking Modules 184
Converting Applications into Modules 184
Coroutine Modules 184
Subroutine Modules 185
Module Internals 186
Subroutine Modules 186
Coroutine Modules 187

Module Reference 189

Appendixes 339

Geometry Conversion Programs
Automatic Data Filtering 342
Shell-Level Usage of Filter Utilities 343
Templates for New Filter Utilities 345
Writing a New Filter Utility 346

The Geometry Viewer Script Language
Scene Files and Object Files 349
Script Language Commands 350

ConvexAVS File Formats
Start-up File 357
Module Library File Formats 358
Image File Format—.x 359
Field File Format—.fld 360
Volume Data File Format—.dat 368
Geometry File Format—.geom 369

Memory Management

Module Routines
Grouped by Function 377
Module Description Functions 377
Modifying and Interpreting Parameters 377
Coroutine Modules 378
Selective Computation 378
Creating Fields 378

Accessing Fields 378
Initializing Modules 379
Handling Errors 379
Defined Alphabetically 380
AVSadd_float_parameter 380
AVSadd_parameter 380
AVSadd_parameter_prop 383
AVSautofree_output 386
AVSbuild_2d_field 387
AVSbuild_3d_field 387
AVSbuild_field 388
AVSchoice_number 389
AVSconnect_widget 390
AVScorout_exec 392
AVScorout_init 392
AVScorout_input 392
AVScorout_output 393
AVScorout_wait 393
AVScreate_input_port 394
AVScreate_output_port 395
AVSdata_alloc 395
AVSdebug 396
AVSerror 397
AVSfatal 397
AVSfield_alloc 398
AVSfield_copy_points 399
AVSfield_data_offset 399
AVSfield_data_ptr 400
AVSfield_free 400
AVSfield_get_dimensions 401
AVSfield_get_int 401
AVSfield_make_template 402
AVSfield_points_offset 402
AVSfield_points_ptr 403
AVSinformation 403
AVSinit_from_module_list 404
AVSinit_modules 404
AVSinitialize_output 405
AVSinput_changed 405
AVSmark_output_unchanged 406
AVSmessage 406
AVSmodify_float_parameter 409
AVSmodify_parameter 410
AVSmodule_from_desc 411
AVSparameter_changed 412
AVSset_compute_proc 412
AVSset_destroy_proc 412
AVSset_init_proc 413

AVSset_module_flags 413
AVSset_module_name 413
AVSwarning 414

C Language Field Macros

Grouped by Function 415
Defined Alphabetically 416
COORD_X_3D 416
COORD_Y_3D 416
COORD_Z_3D 416
I1DV 417
I2D 417
I2DV 417
I3D 418
I3DV 418
I4D 418
I4DV 419
MAXX 419
MAXY 419
MAXZ 419
RECT_X 420
RECT_Y 420
RECT_Z 420

Geometry Routines

Grouped by Topic 423
Object Creation Routines 423
Generic - GEOMobj 423
Lines and Polytriangles - GEOM_POLYTRI 423
Quadrilateral Meshes - GEOM_MESH 423
Polygons & Polyhedrons - GEOM_POLYHEDRON 423
Spheres - GEOM_SPHERE 423
Text Labels - GEOM_LABEL 424
Object Modification Routines 424
Color - All GEOMobj except GEOM_POLYTRI 424
Normals - GEOM_MESH, _POLYHEDRON 424
Vertices - GEOM_MESH , _POLYHEDRON 424
Type Conversion - GEOM_POLYTRI 424
Object Transformation Routines 424
Automatic Placement - All GEOMobj 424
Extent - All GEOMobj 425
Edit List Manipulation 425
Creation - GEOMedit_list 425
Object Insertion - All GEOMobj 425
Object Transformation - All GEOMobj 425
Object Hierarchy - All GEOMobj 425
Object Properties - All GEOMobj 425
Light Source - Light 425

Geometry File Utilities 426
Input and Output - All GEOMobj 426
Object management- All GEOMobj 426
Listed Alphabetically 427
GEOMadd_disjoint_line 427
GEOMadd_disjoint_polygon 427
GEOMadd_float_colors 428
GEOMadd_int_colors 429
GEOMadd_label 429
GEOMadd_normals 430
GEOMadd_polygon 431
GEOMadd_polygons 431
GEOMadd_polyline 432
GEOMadd_polytriangle 433
GEOMadd_radii 433
GEOMadd_vertices 434
GEOMadd_vertices_with_data 434
GEOMauto_transform 435
GEOMauto_transform_list 435
GEOMauto_transform_non_uniform 435
GEOMauto_transform_non_uniform_list 436
GEOMcreate_label 436
GEOMcreate_label_flags 436
GEOMcreate_mesh 437
GEOMcreate_mesh_with_data 438
GEOMcreate_normal_object 439
GEOMcreate_obj 439
GEOMcreate_polyh 440
GEOMcreate_polyh_with_data 440
GEOMcreate_scalar_mesh 441
GEOMcreate_sphere 442
GEOMcvt_mesh_to_polytri 443
GEOMcvt_polyh_to_polytri 444
GEOMdestroy_edit_list 444
GEOMdestroy_obj 445
GEOMedit_color 445
GEOMedit_concat_matrix 446
GEOMedit_geometry 446
GEOMedit_light 447
GEOMedit_parent 447
GEOMedit_picked 448
GEOMedit_properties 448
GEOMedit_render_mode 449
GEOMedit_set_matrix 449
GEOMedit_visibility 450
GEOMflip_normals 450
GEOMgen_normals 450
GEOMinit_edit_list 451

GEOMnormalize_normals 451
GEOMread_obj 451
GEOMset_computed_extent 452
GEOMset_extent 452
GEOMset_object_group 452
GEOMset_pickable 453
GEOMunion_extents 453
GEOMwrite_obj 453
GEOMwrite_text 454
Programming Information 454
Object Creation Routines 454
Creating An Object 455
Flags 455
Extents 455
Object Utility Routines 456
Edit List Manipulation 456
FORTRAN Binding 458
Compiling and Linking 458

Module Code Examples

C Language Subroutine 459
C Language Coroutine 462
FORTRAN Subroutine 466

Creating Online Help for Your Modules

Format and Naming Conventions 469
Integrating Your Help Files 470
ConvexAVS Help 470
Man Command 471

Index

Preface

Purpose and Audience

The *ConvexAVS* manual describes how to use ConvexAVS subsystems and develop networks.

ConvexAVS V1.0 is the CONVEX implementation of the Stardent Application Visualization System (AVS) V2.0 for Convex C-series supercomputers.

This guide addresses scientists, engineers, and software developers who want to visualize their data or need to develop modules using ConvexAVS.

Organization

This manual is organized into the following sections:

- The Using ConvexAVS part introduces general concepts necessary to use the software and provides details about the subsystems.
- The Developing Applications part provides fundamental information about data types, module construction, and data flow.
- The Module Reference part provides man pages for every module.
- The Appendixes part contains useful reference information.

Notational Conventions

This section discusses notational conventions used in this book.

Command Syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
   ①         ②        ③      ④         ⑤
```

1. **COMMAND** must be typed as it appears.
2. **input_file** indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. **output_file** indicates an optional file name.

General Conventions

In general, the following conventions are used in this guide:

- **`Bold constant-width font`** identifies user input in examples.
- *Italics*
  - Designate user-supplied variables in a command-line example.
  - Introduce new and important terms.
  - Identify variables in mathematical equations.
  - Indicate document titles.
- `Constant-width font` designates input and output, including:
  - Command names and options.
  - System calls.
  - Data structures and types.
  - Directives, program statements, display examples, printout examples, and error messages returned.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.

- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **Return** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press **Return**.
- The ConvexOS prompt is printed as a percent sign (`%`).

Note

A Note highlights supplemental information.

Caution

A Caution highlights procedures or information necessary to avoid damage to equipment, software, or data.

Associated Documents

Using this software may require information not specific to the tasks described in this document.

For more information on the operating system, you can order the following manuals:

- *ConvexOS Primer* (DSW-133). This book introduces users to the CONVEX operating system.
- *Managing ConvexOS: Operations* (DSW-031). This guide provides instructions for managing, monitoring, and controlling system resources.

Ordering Documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Include the order number listed on the back cover.

Technical Assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384
- Outside the continental U.S., contact local CONVEX office.

The `contact` Utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it to the TAC electronically. The TAC notifies you within 48 hours that your report has been received.

To use `contact` requires:

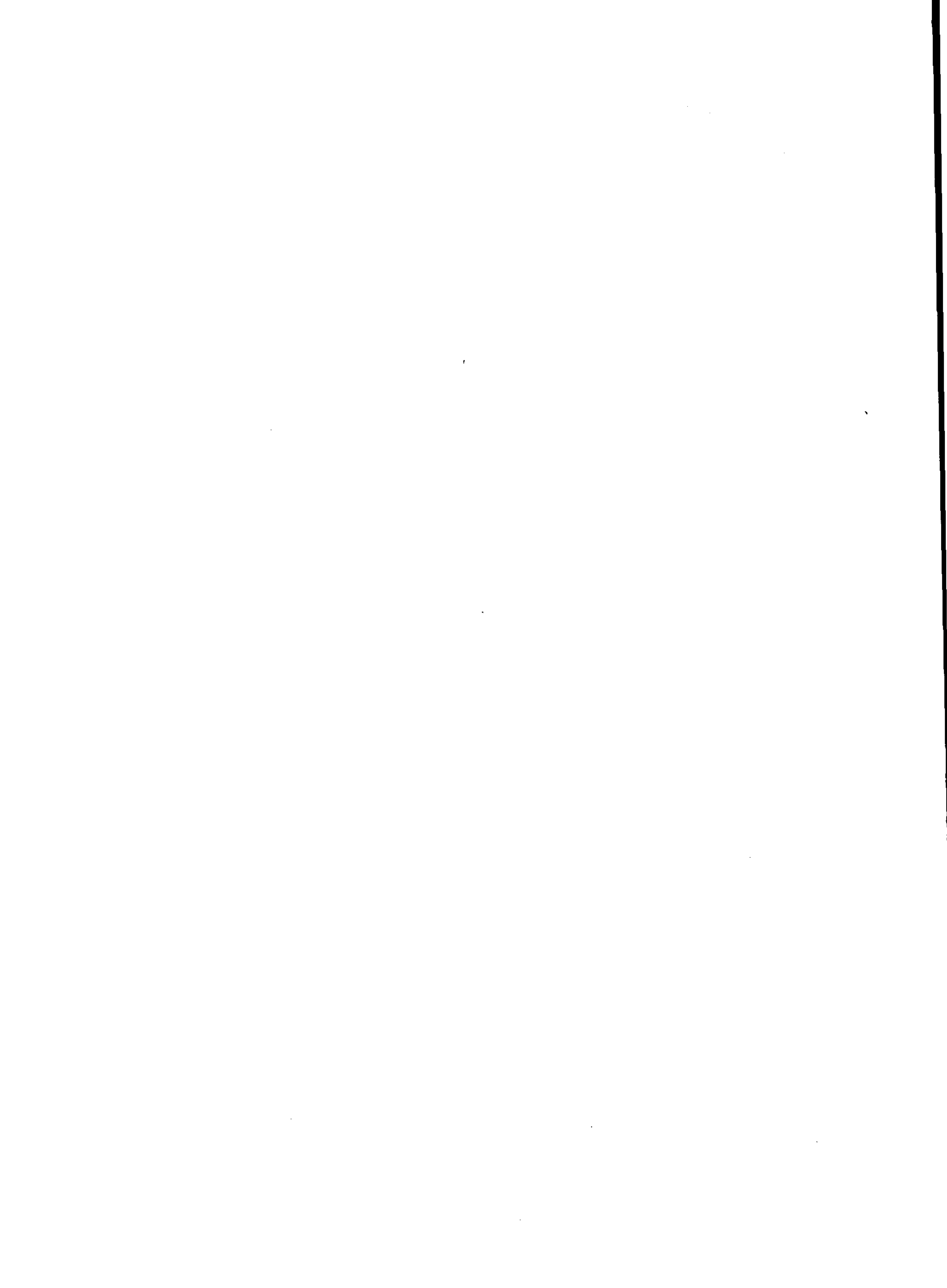
- UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC.
- Full path name of the program or utility in question.
- Version number of the program or utility in question.

Refer to the `contact`(1) man page for complete details.

Using ConvexAVS

This part introduces general concepts necessary to using the software and provides details about the subsystems.

- What is ConvexAVS?
- Starting ConvexAVS
- Image Viewer
- Volume Viewer
- Geometry Viewer
- Network Editor



What Is ConvexAVS?

1

Introduction

The increasing power of supercomputers and graphics systems has made it possible for the scientific and engineering communities to gain new insight into their disciplines. In areas as diverse as fluid dynamics, computer-aided engineering, molecular modeling, and geophysics, researchers are applying these powerful systems to analyze and view their data, producing real-time interactive displays.

A limiting factor in this growing field has been the existing software tools that require specialized programming expertise and great expense, both in time and in money. The Convex Application Visualization System (ConvexAVS) addresses this problem, allowing researchers to apply the hardware power to their problems without requiring programming expertise or a great investment of time.

ConvexAVS includes a substantial number of visualization techniques which you can invoke simply by selecting them from a menu. These image-viewing and volume-viewing techniques will satisfy many users' first-level needs in turning data into pictures.

For situations in which these standard techniques do not suffice, you can construct your own visualization applications by combining software components into *executable flow networks*. The components, called modules, implement specific functions in the visualization cycle:

Filtering the basic data into a more usable form (more informative, smaller, etc.)

Mapping the filtered data into geometric primitives (triangles, lines, spheres, etc.)

Rendering the geometric primitives into pictures on the display screen or a file.

The flow networks are built from a menu of modules by using a direct-manipulation interface. You produce an application by selecting a group of modules and drawing connections between them. In many cases, you can construct an entire visualization application in this way, using standard modules and without resorting to any traditional procedural programming.

ConvexAVS includes a rich set of modules for construction of networks. Given the nature of scientific visualization and the need for extensibility, ConvexAVS also supports the creation and dynamic loading of new modules. You need not have detailed knowledge of the ConvexAVS implementation or expertise in disciplines outside their areas of interest. Modules are software building blocks with well-defined interfaces, written either in FORTRAN or in C. The overall structuring of the application is handled on the ConvexAVS level; the computational details are handled within modules as FORTRAN or C procedures.

Modules receive typed data as inputs and produce typed data as outputs. The basic data types in the system are oriented toward scientific data manipulation and graphic display. These types include 1D, 2D, and 3D vectors of floating-point values, 2D and 3D grids with vectors of floating-point values at each grid point, geometric data, and images. Byte and integer data types are also supported.

In addition to input and output data, modules also have parameters that control the module's computation. Once the structure of the application has been established, ConvexAVS executes the network, allowing you to interact with the application by navigating through the network diagram and interacting with various modules through their individual parameters. ConvexAVS generates the control panel user interface to a module automatically by associating parameters with graphical control panels (buttons, sliders, etc.).

The remainder of this chapter presents an overview of the ConvexAVS approach to the challenge of scientific visualization.

Scientific and Engineering Data

In the engineering and scientific arena, a set of data to be processed by computer typically takes the form of a sequence of numbers. Sometimes, the numbers are generated as a real-time data stream. Many measurement instruments can produce streams of digital output (perhaps aided by an analog-to-digital converter). Often, the data is being produced by a running computational process. Sometimes, the numbers have been generated at some previous time, and are stored in a file on disk. ConvexAVS has facilities for handling both real-time data streams and disk-based data.

Each number in a data set can be represented in various ways. ConvexAVS handles the following integer and floating-point numerical formats:

byte (8 bits)

A single byte can represent an unsigned integer in the range 0..255 or a signed integer in the range -128..127.

integer (32 bits)

A single machine word can represent an unsigned integer in the range $0..2^{32}-1$ (0..4294967295) or a signed integer in the range $-2^{31}..2^{31-1}$ (-2147483648..2147483647).

single-precision (32 bits)

A single machine word can also be used to represent a floating-point quantity in IEEE 754 single format.

double-precision (64 bits)

Two machine words can be used to represent a floating-point quantity in IEEE 754 double format.

In many cases, the sequence of numbers in a data set has an implied or explicit structure. For instance, a sequence of 40,000 numbers may represent a 2D square grid (a 200x200 matrix). Similarly, a sequence of 500,000 numbers might represent a 100x200x25 lattice of data points.

Presumably, a numerical grid corresponds to a physical grid with a particular distance between grid points. In some cases, the grid may be non-regular— the distance between grid points is variable, rather than constant. It is also possible for the grid to describe a curved or arbitrarily deformed space, instead of a rectangular space. For more on this subject, refer to the section "Preparing Your Data for Use with ConvexAVS" later in this chapter.

In ConvexAVS, data files always begin with a header that specifies the overall structure of the data. Additional structural information (for instance, the real-world coordinates that correspond to the numerical data grid) can also be included in the data file.

Other Data Formats

ConvexAVS can also use data that is in a format other than a simple stream of numbers. At many sites, purely numerical data has already been processed into a structured form by a user-written program or by an application software package. For instance, Table 1-1 shows part of a file written in the Brookhaven Protein Data Bank format. This file defines the structure of a particular protein molecule called *crambin*. There is a ConvexAVS data input module to read files in this format. You can supply your own data input modules for other data formats.

Table 1-1
Data File in Brookhaven Protein Data Bank Format

ATOM	1	HN1	THR	1	17.017	14.972	4.068
ATOM	2	HN2	THR	1	16.297	13.912	2.883
ATOM	3	N	THR	1	16.982	14.095	3.587
ATOM	4	HN3	THR	1	17.707	14.470	3.008
ATOM	5	CA	THR	1	16.949	12.808	4.348
ATOM	6	C	THR	1	15.686	12.779	5.142
ATOM	7	O	THR	1	15.236	13.827	5.603
...							

ConvexAVS implements two basic strategies for translating numerical data into color images. In the pixel-based method, data points become pixels, more or less directly. In the geometry-based method, the numerical data is converted to descriptions of 3D objects. These are, in turn, turned into color images by the machine's low-level graphics software and rendering hardware.

These two strategies are described further in the sections that follow.

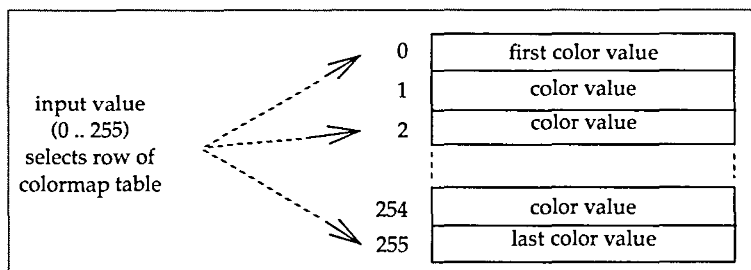
Pixel-Based Visualization

The essence of the pixel-based visualization strategy is simple: take a raw data value and translate it into a number that represents a color. In ConvexAVS, this translation is accomplished with a simple table lookup, called a colormap. You can define, save, and retrieve your own colormaps. ConvexAVS includes an interactive drawing tool for generating colormaps conveniently and quickly.

Colormap Lookup

A ConvexAVS colormap is a 256-row table; each row specifies a 24-bit true-color value (and, optionally, an 8-bit auxiliary field), as shown in Figure 1-1. A colormap lookup consists of using an input value to select a particular row of the table. The color value in that row is the result of the lookup.

Figure 1-1
ConvexAVS
Colormap Lookup



In general, ConvexAVS colormaps accept byte data as input values. Each byte is considered to be an unsigned integer (0..255) that specifies a particular row of the table. ConvexAVS colormaps are independent of the hardware colormaps used by low-level graphics software. All ConvexAVS colormaps produce 24-bit true color output. If necessary, further translation takes place automatically— for instance, to produce images on a machine with only 12 color planes.

Further Pixel Processing

If multi-dimensional data is converted to pixels, the results must somehow be reduced to 2D before they can be displayed as an image on screen. ConvexAVS provides several ways to perform such reductions. One example is slicing in which a 2D cross-section can be made through a 3D block of pixels.

High-Quality Pixel-Based Visualization

In simple pixel-based visualization, each data point corresponds to a single pixel. When you zoom in on a particular portion of the image, the magnification is performed by pixel replication. (For instance, a single pixel value may be used throughout a 6x6 patch in a zoomed image.)

A variety of techniques can be used to improve image quality, such as high-order interpolation of data values. In addition, 3D graphics techniques such as lighting, shading, and perspective viewing can be used to compute the interpolated pixel values.

Geometry-Based Visualization

The other ConvexAVS strategy for turning numbers into pictures brings all the power and flexibility of interactive 3D graphics to the visualization arena. The raw data values (or, more likely, a subset of the values) are mapped into the vertices (points) of geometric objects. The values are used to assign colors to the vertices, using ConvexAVS colormaps. Then, the graphics subsystem creates color images from the geometric descriptions.

There are many techniques for creating geometric descriptions, or geometries, from raw data. For instance:

- Represent each atom of a molecule as a sphere. Assign color and transparency to the sphere based on the type of atom.
- Given a set of data that specifies the temperature at many points within a volume, use all the points at a given temperature to define an isosurface.
- Given a set of data that specifies the wind velocity at many points within a volume, use arrows to represent the velocity at each point on an arbitrary plane within the volume.
- Given window velocity data as above, construct flow lines to represent the motion of an object through the field.

ConvexAVS Subsystems

The following subsystems are included in ConvexAVS:

Image Viewer

The Image Viewer subsystem is a high-level tool for manipulating and viewing 2-dimensional images.

Volume Viewer

The Volume Viewer subsystem is a high-level tool for visualizing 3-dimensional (volume) data. It can handle data sets that convey information for a sampling of points in a 3D space.

Geometry Viewer

The Geometry Viewer subsystem allows you to compose scenes that contain geometrically-defined objects created by programs or ConvexAVS modules that use the ConvexAVS GEOM programming library. You can transform the objects themselves (move, rotate, scale), change the viewing parameters (move the eye point, perspective view, etc.), and control the way in which the graphical images are rendered (lighting and shading, etc.).

Network Editor

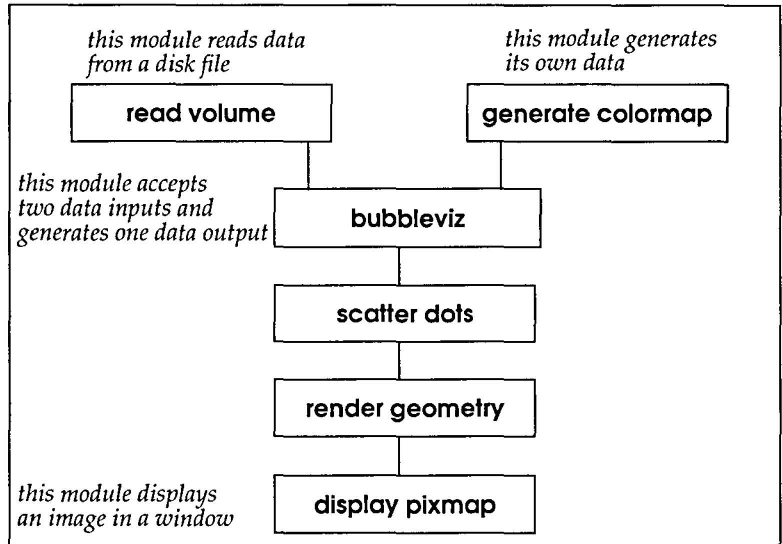
The Network Editor subsystem is a tool for connecting computational modules together into networks that perform visualization functions. Modules and networks are discussed in the sections that follow.

When using ConvexAVS for the first time, familiarize yourself with the product by selecting **About AVS** from the main menu.

ConvexAVS Modules

The module is the ConvexAVS computational unit. Each module accepts data as input and generates other data as output. To create a ConvexAVS application, you connect together a group of modules into a network. The connections represent the flow of data among the modules. Typically, the data originates in one or more disk files, but it can also be supplied by an external program, running on the same machine or on another machine in the local network. The data is transformed into one or more images by a collection of modules, and finally is displayed in a window on screen. Figure 1-2 shows a network of modules.

Figure 1-2
Network of ConvexAVS Modules



Modules: Ports and Parameters

Each ConvexAVS module is designed to be a powerful, flexible, easy-to-use processing component. A module is general in its functionality so that you can use it in a variety of application contexts. Each module does a substantial amount of processing so that networks need contain only a few modules to do work.

You can include a particular module in any number of ConvexAVS applications (networks). You can even include the same module more than once in a single network.

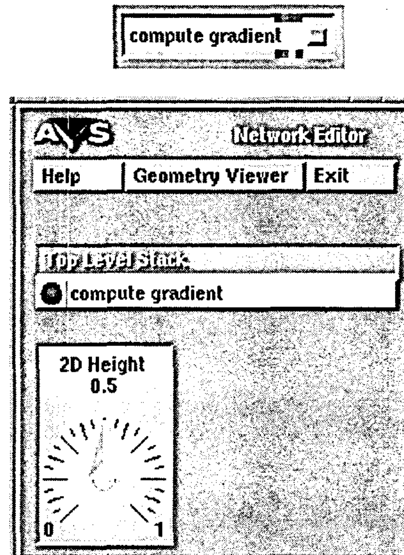
The key to the modular approach to application building is that each module has a simple, consistent interface that includes:

- A set of *data inputs*
- A set of *input parameters* that controls the way the module processes its input data or determines which data to use
- A set of *data outputs*

One of the most powerful ConvexAVS features is that you can change parameter values interactively as a network executes.

When you use ConvexAVS to create a network, each module's interface is represented visually by a *module icon* and a *control panel* (Figure 1-3). The *module icon* is a rectangle labeled with the module's name. Each data input is represented by an input port along the top edge. Each data output is represented by an output port along the bottom edge. Each input parameter is represented by a control widget (slider, dial, etc.); the controls are assembled in a separate control panel window.

Figure 1-3
A Module's Interface: Icon and Control Panel



Data Inputs

A module accepts one or more data sets as input. Each data set must be of a particular ConvexAVS data type: field, colormap, etc.

Each data input is represented on the module icon by a color-coded input port, along the top edge of the icon. The color indicates the type of data that the port accepts as shown in Table 1-2.

Table 1-2
Module Input Ports/Convex-AVS Data Types

Port Color	Data Types
red	geometry
yellow	colormap
light blue	pixmap
multi-color	field

ConvexAVS helps you to match data types as you interactively build a network. When you begin to establish a module-to-module connection, ConvexAVS shows you the valid possibilities.

Some modules have no input ports at all. Such modules create their own data or read data in from a source that is external to the ConvexAVS network (for example, a disk file).

Input Parameters

A module's data inputs determine the type of data it processes, while its input parameters determine how the data is to be processed.

The following examples use the modules shown in Figure 1-2 to illustrate several types of parameters:

read volume brings a 3D block of byte values into a network. Its input parameter specifies the file from which the values are to be read.

generate colormap creates and outputs a colormap that transforms byte values into color values. Its input parameter is implemented as an interactive colormap editor, with which you specify the 256-entry colormap.

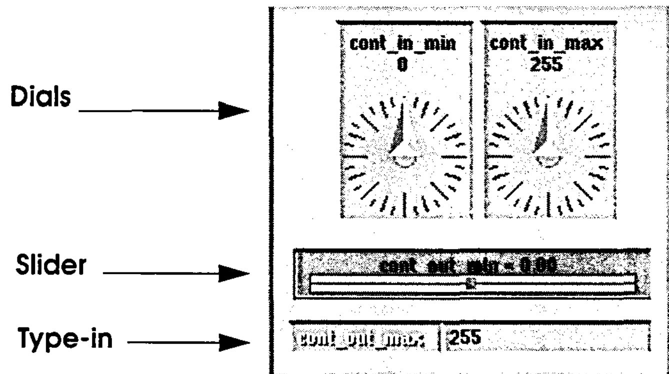
bubbleviz module generates spheres of various radii and colors at the element locations of a 3D field.

You can control the way in which a module processes its data by adjusting the parameters associated with each module. In ConvexAVS, control knob widgets give you an easy way to make changes to the parameters. You can use knobs for the following kinds of actions:

- change the angle of a cross-section plane or a rotation
- change a coloring scheme, change the way values are sampled from a large data set
- blow up an image to examine some detail

Each of a module's parameters is represented by a unique on-screen control widget. Figure 1-4 presents examples of control widgets.

Figure 1-4
Module Control Widgets



Widget Types

ConvexAVS implements the following types of control widgets:

Dials and sliders indicate integers or floating point values.

Typeins specify a character string: title, label, file name, etc. Typeins can also be used to specify numeric values: integers or floating-point numbers.

Toggles are on/off switches for various parameters.

Radio buttons provide sets of mutually exclusive choices.

File browsers allow you to specify a file to be read or written (also called choices).

Data Outputs

Data outputs for modules are analogous to data inputs. Each data output is represented on the module icon by a color-coded output port along the bottom edge of the icon. The color-coding is the same for input ports.

Subroutine and Coroutine Modules

There are two types of ConvexAVS modules: subroutines and coroutines. For more information, refer to *Developing Applications*.

- Subroutine modules are like subroutines in a standard program. When you execute a network, each subroutine module initializes itself (a process is created). But the module does not perform any work (the process sleeps) until the ConvexAVS

Flow Executive signals it. In addition to waking up the module, the Flow Executive passes its input data to it. When the module finishes computing its output data, it passes the data back to the Flow Executive, and then returns to its dormant state.

- Coroutine modules are active, cooperative processes that continually execute. They pass data to the Flow Executive on their own initiative instead of doing so only when signalled. Coroutine modules typically implement computational simulations, such as repeatedly releasing particles to flow through a field.

Module Libraries

The modules are grouped into module libraries; each library contains a set of modules designed to be used together. During a ConvexAVS session, you can switch back and forth among module libraries. You can also rearrange the libraries or create new ones simply by creating lists of modules with a text editor program.

User-Written Modules

One of the most important aspects of the ConvexAVS system is its extensibility. Many installations have already developed computer programs to process the raw data. ConvexAVS makes it easy to turn such user-supplied programs into ConvexAVS modules. Once this is accomplished, the user-written module can be combined with other modules— ConvexAVS-supplied or user-written— to implement visualization applications.

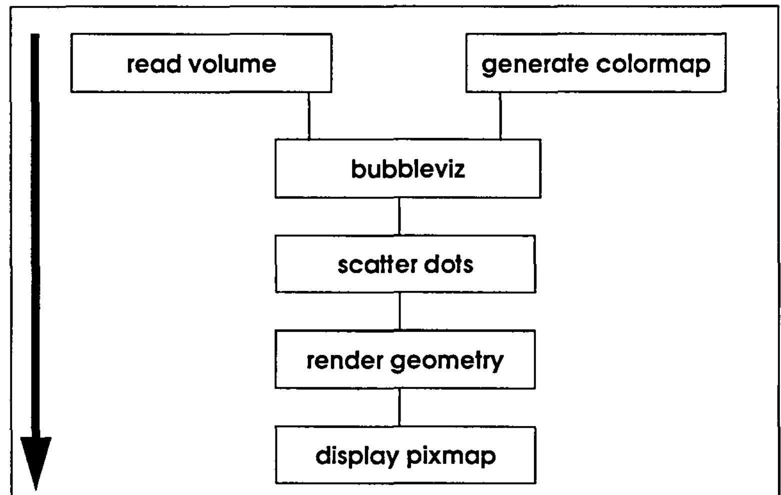
ConvexAVS Networks

As modules are the computational units in ConvexAVS, networks are the operational units. Given data that you wish to view in a particular manner, you select the modules that perform the appropriate computations and combine them into a network. You can save the network on disk, then repeatedly use it to visualize the same data, or any other data set of the same form. After using ConvexAVS for some time, you will likely maintain a group of networks that satisfy your visualization needs.

Data Flow in an ConvexAVS Network

Figure 1-5 repeats the network shown in Figure 1-2. This time, the figure emphasizes the way data flows through a typical ConvexAVS network.

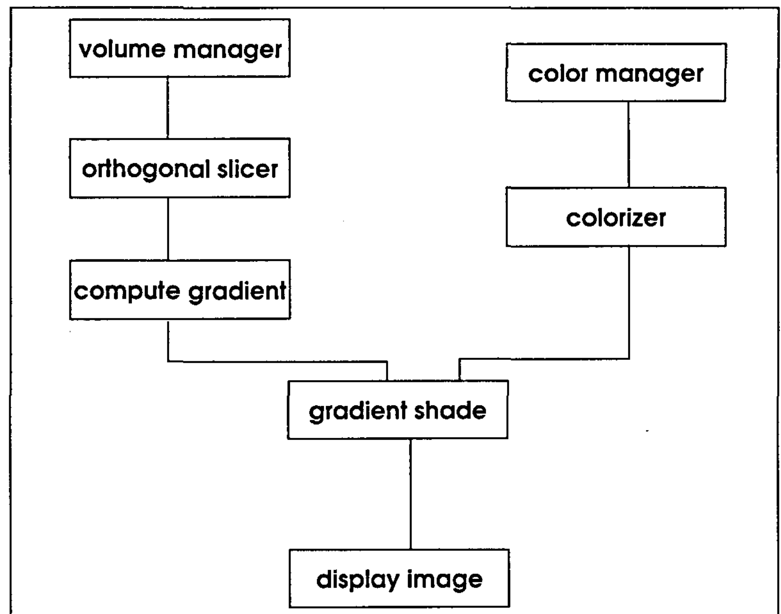
Figure 1-5
Data Flow in a Network



The data-flow diagram reflects the scientific visualization process, which begins with data and ends with on screen images. Networks that use data stored on disk begin with a read data module. (There are several such modules, to accommodate the variety of ConvexAVS data types.) These modules allow you to specify the name of a file containing the raw data. By selecting different files, you can use the same network to visualize different data sets.

The network illustrated above has a simple structure and performs a (relatively) simple task— reading a single data set and constructing a single image. More complex networks can use multiple data sets that create independent images or composite images. A network can consist of any number of independent sub-networks. Figure 1-6 illustrates a more complex network.

Figure 1-6
Complex Network Structure

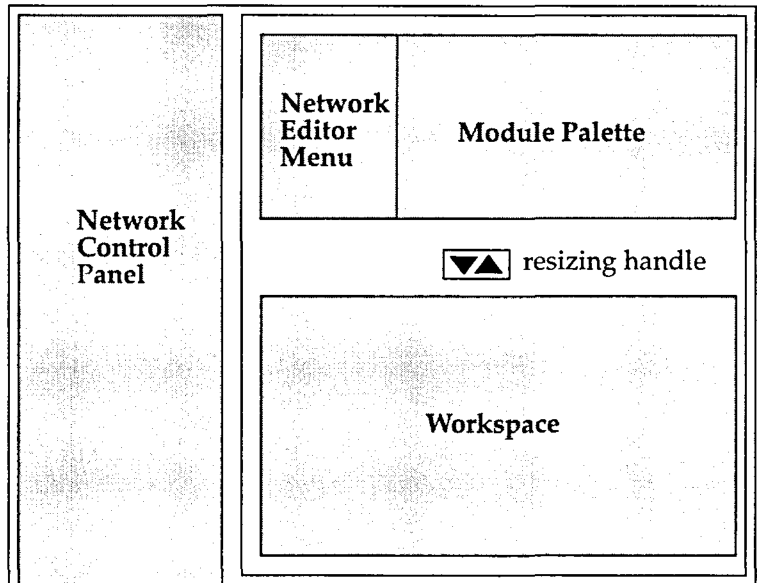


There are limits to network complexity, however. Networks are inherently flat— ConvexAVS provides no support for creating hierarchical structures. And networks may not contain cycles: a module's output data cannot subsequently be fed back into the module as input, directly or indirectly.

You create networks using the ConvexAVS Network Editor subsystem. The mouse-driven interface allows you to interactively construct network diagrams, like those illustrated above. To select a module, you drag its icon from a Palette into a Workspace (Figure 1-7). To make and break connections between modules, you click-and-drag the middle mouse button.

At any time, you can save a network in a disk file for later retrieval. Only the network structure and the current settings of the input parameters are saved—the data to be visualized is not part of the network but is loaded when the network executes.

Figure 1-7
ConvexAVS Network Editor Windows



Network Control Panel

A network's data-flow diagram omits one very important aspect of network execution: the settings of the module's input parameters. As you construct a network, the control widgets that represent the parameters (and allow you to control their values) are automatically assembled in the Network Control Panel window along the left edge of the screen.

By default, the control widgets are collected into pages, one page for each module. You can redesign the layout of control widgets, however, to create simpler and more convenient user interfaces to your networks. This allows developers of networks to package their work so that even the most sophisticated visualization tasks can be performed easily and reliably by users.

Image and Volume Viewer Networks

Two of the ConvexAVS subsystems, the Image Viewer and the Volume Viewer, make use of networks transparently. These subsystems are entirely menu-driven: through menu choices, you select the data to be processed along with one or more visualization techniques to be applied to the data. Each technique is implemented with a pre-existing ConvexAVS network. You can control the execution of the networks using control panels.

Both the Image Viewer and the Volume Viewer allow you to view the networks that implement the visualization techniques and to switch to the Network Editor in order to revise or enhance them.

ConvexAVS Display Windows

ConvexAVS creates its visualization images in display windows on the screen. (There is also a provision for saving images in PostScript files for printing, storage, or transfer to another site.) Each display window is an X Window System window. This integration of ConvexAVS with X means that you can move, resize, iconify, and otherwise manipulate display windows using the X window manager. ConvexAVS also provides some window oriented functions, such as zoom. You can integrate display windows into the control panels of the visualization networks you build that allow you to build predictable and space-efficient user interfaces.

Preparing Your Data for Use With ConvexAVS

ConvexAVS is designed for users who already have large numerical data sets waiting to be visualized. Inevitably, the data sets will encompass a wide range of file formats. Some formats may be highly structured, comprising many types of data records. Other formats may be essentially unstructured: a small amount of header information followed by a stream of data.

In its current implementation, ConvexAVS can directly read several file formats, which are described below. Some of these formats (for example, the Brookhaven Protein Data Bank format) are already in general use in the scientific/engineering community. If your data is already in one of these formats, you can start immediately to visualize it using ConvexAVS. Other formats are ConvexAVS-specific. This means you will need to do some data-conversion programming work to make such data directly usable by ConvexAVS.

Supported File Formats

The file formats listed in Table 1-3 can be used directly by ConvexAVS:

Table 1-3
ConvexAVS-Readable File Formats: General Use

File Format	ConvexAVS File Name Suffix
Mathematica ThreeScript	*.ts*
Movie BYU	*.byu*
Brookhaven Protein Data Bank	*.ent*
Polygen Protein Data Bank	*.pdb*
UNC	*.ppoly*

Each of these formats embodies a description of one or more geometric objects. If a file is named with the appropriate suffix from the table above, a single command in the Geometry Viewer subsystem reads the file, automatically converts its geometric descriptions to the ConvexAVS geometry format, and displays the object(s) in a window. (The converted data is also stored on disk in a file with a *.geom* suffix.)

Protein Data Bank format

There is also a ConvexAVS module, **pdb to geom**, that reads a file in the Protein Data Bank format and outputs the data as a ConvexAVS geometry. (None of the other conversion routines are implemented as modules.)

For more information about conversion of these data formats to the geometry format, see the "Geometry Conversion Programs" appendix.

ConvexAVS-Specific File Formats

ConvexAVS has several data formats of its own, which are designed to accommodate a wide variety of scientific/engineering data sets with a minimum of conversion effort. These formats are summarized here and described in more detail in the following sections.

Geometry

As described in the preceding section, ConvexAVS has its own data format for the specification of display output in terms of geometric primitives: lines, triangles, spheres, etc. Many (but not all) numerical data sets are converted to geometries during the visualization process.

Image

ConvexAVS can read an image of any size. The file format is essentially a stream of pixel values: each pixel is specified by a red-green-blue triple.

Field

The most flexible ConvexAVS data format is the field, a generalization of the n-dimensional array construct that is commonly used to represent scientific data sets.

Volume Data

The ConvexAVS field construct is extremely general and powerful. As a convenience, ConvexAVS also supports a simpler format that handles one commonly-used type of field, volume data.

Geometry Data File Format

The ConvexAVS Geometry Viewer subsystem (also implemented as the read geometry module) reads files in the GEOM file format. Such files can be created with routines in the special GEOM programming library *(libgeom.a)*, which is included with ConvexAVS. For a description of this library and the file format, refer to Appendix G, *"Geometry Routines."* Geometry data files should have names that end with a .geom suffix.

Image Data File Format

The read image and image manager modules can read a file that contains an image— a 2D array of pixel values. In ConvexAVS, such files should have names that end with a *.x* suffix.

The file must begin with a two-word header that specifies the dimensions of the image:

first word: number of pixels in *horizontal* direction (32-bit integer)

second word: number of pixels in *vertical* direction (32-bit integer)

There is no explicit limit on the size of an image.

The remainder of the file is a sequence of 4-byte (32-bit) words, one for each pixel of the image. The pixels are arranged in rows; there is no padding at the end of a row.

The four bytes of a pixel are interpreted as four component values in the range 0..255. Three of the bytes are the red, green, and blue color components. The fourth byte is an auxiliary field, which is used by some ConvexAVS modules to represent an opacity/transparency value:

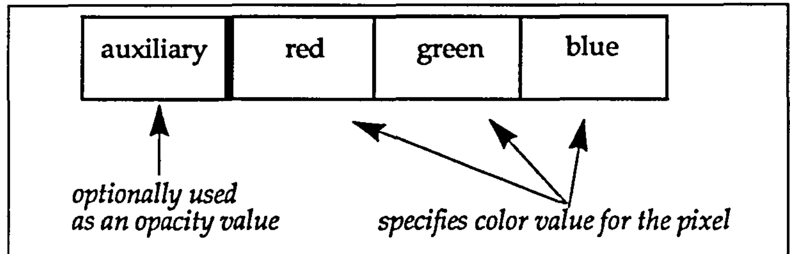
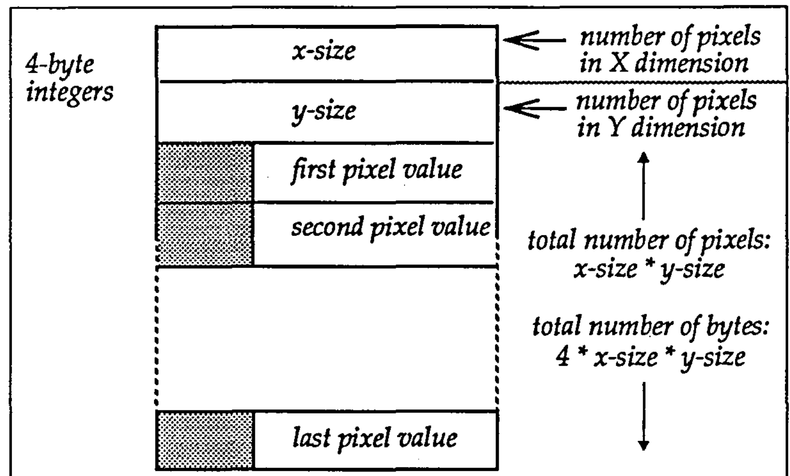


Figure 1-8 illustrates the ConvexAVS image data file format. Image data files should have names that end with a *.x* suffix.

Figure 1-8
Image Data File Format



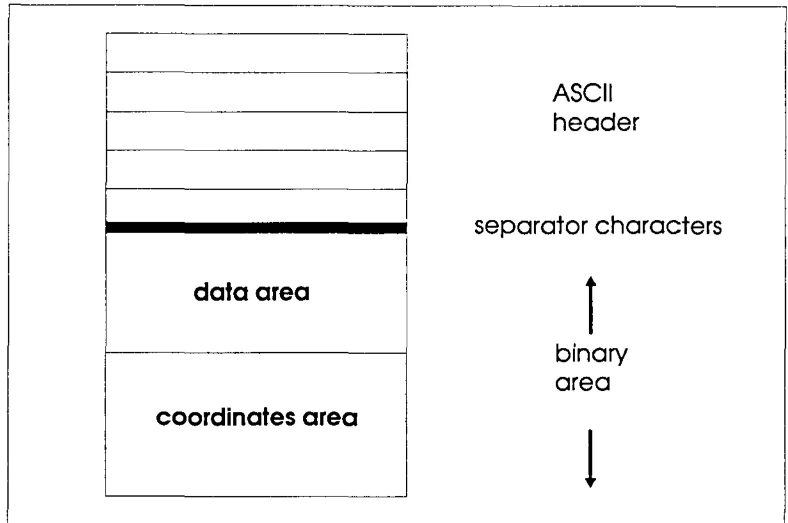
Field Data File Format

A field is a generalization of the familiar array structure. Whereas each element of an ordinary array has a single data value (for example, byte or integer), each element of a ConvexAVS field can have a list of data values. Thus, a field can be described as an *n-dimensional* array with an *m-dimensional* vector of values at each array location (where *n* and *m* are any integers).

Moreover, the field can include coordinate data so that each field element is mapped to a real-world location.

Figure 1-9 illustrates the top-level structure of a ConvexAVS field:

Figure 1-9
ConvexAVS Field Structure



Field data files should have names that end with a *.fld* suffix.

Before describing the field file format in more detail, let's look at several examples of fields.

Example 1: Uniform 2D Field

Consider the following 2D integer-valued array (using a FORTRAN-style notation):

DATA(I, J)	I=1,2	J=1,5
DATA(1,1)	=	12
DATA(2,1)	=	17
DATA(1,2)	=	4
DATA(2,2)	=	0
DATA(1,3)	=	10
DATA(2,3)	=	-5
DATA(1,4)	=	16
DATA(2,4)	=	16
DATA(1,5	=	16
DATA(2,5)	=	8

This array describes a 2D computational space, with I and J dimensions. The size of the I dimension is 2; the size of the J dimension is 5. The data is of type *integer.*

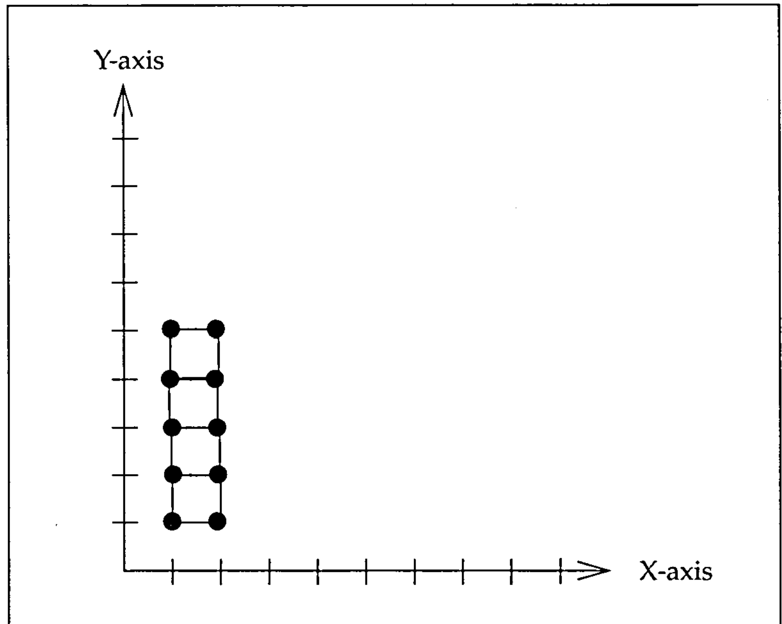
Because there is only one data value for each field element, this is said to be a scalar field. The following notation might be used to indicate the values of a vector field:

```
DATA(2,3)    =       (2.51, 1.09, 5.73)

      or

DATA(2,3)    =       (2.51, 1.09, 5.73, 0, 1)
```

In the first case, the field is still 2-dimensional, but the data value is said to be a 3-vector. Such a data value might be used to represent a velocity vector. The 5-vector in the second case might represent the temperature-pressure-humidity measurements at each location in space, along with two boolean values to indicate the presence/absence of other atmospheric conditions.

In the absence of any additional information, there is a natural mapping between the computational space and a 2D physical space, the X-Y coordinate plane as shown in Figure 1-10:

Figure 1-10
Mapping Computational and 2-D Physical Space



The physical space is a uniformly-spaced lattice. Accordingly, a field with no coordinate data is said to have the field type *uniform*.

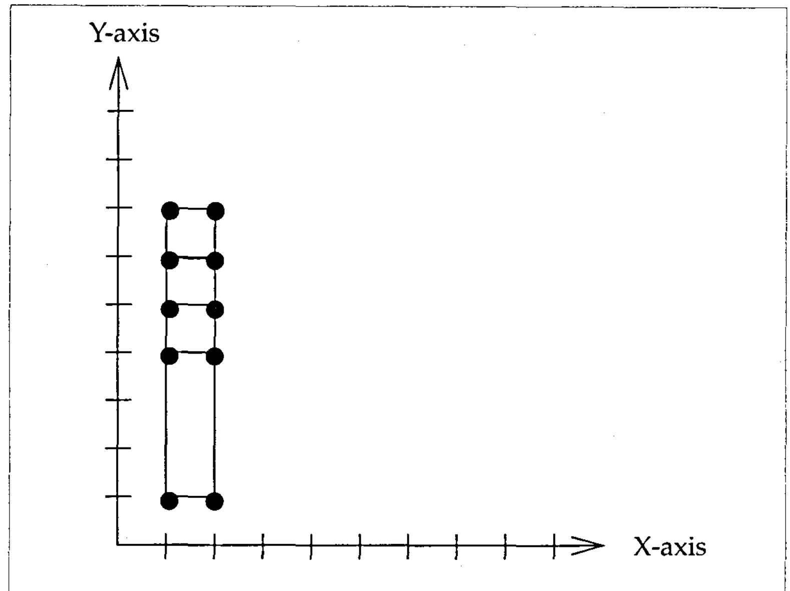
Example 2: Rectilinear 2D Field

Continuing the preceding example, we can establish an explicit mapping between the computational and physical spaces by specifying coordinate data:

X-coordinates:	0, 3, 6, 9, 12
Y-coordinates:	20*log(1), 20*log(2), 20*log(3), 20*log(4), 20*log(5)

For example, array element IDATA(1,3) is mapped to physical location (0, 20*log(3)) according to this scheme. This mapping from computational space to the X-Y plane can be pictured as follows:

Figure 1-11
Mapping a Rectilinear 2D Field



Note

In a *rectilinear* field, lines connecting the lattice points are always mutually orthogonal— all the angles are right angles.

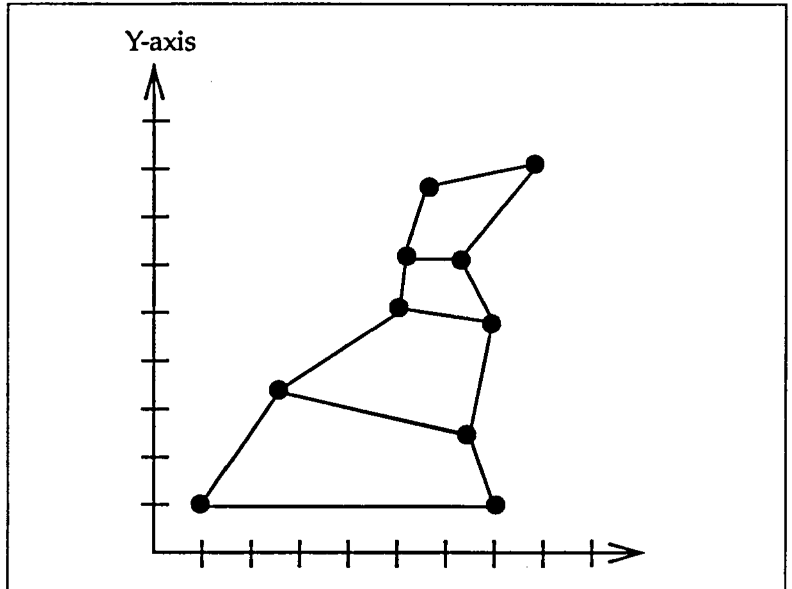
Example 3: Irregular 2D Field

Continuing the example once more, there is another way of establishing a mapping between the computational and physical spaces. Instead of mapping the array indices, we can map individual field elements to arbitrary points in physical space:

```
DATA(1,1) ->    (1, 1)
DATA(2,1) ->    (7, 1)
DATA(1,2) ->    (3, 3)
DATA(2,2) ->    (6, 2.5)
DATA(1,3) ->    (4, 4.5)
DATA(2,3) ->    (5.5, 4.5)
DATA(1,4) ->    (3.5, 6)
DATA(2,4) ->    (4.5, 5.5)
DATA(1,5) ->    (3.5, 7.5)
DATA(2,5) ->    (6,8)
```

This mapping from computational space to the X-Y plane can be pictured as follows:

Figure 1-12
Mapping an Irregular 2D Field



There is nothing in this scheme that restricts the physical space to having the same number of dimensions as the computational space. For example, the field element DATA(2,3) could be mapped to the physical point (4.5, 5.5, -8.1) in 3D space. This kind of mapping can be used to wrap a plane (computational space) around a sphere (physical space) or to warp a flat plane into a 3D manifold.

For additional examples, including some involving non-2D fields, refer to Chapter 8, "Data Types," in the *Developing Applications* section.

ASCII Header

Every field file must begin with a header in ASCII text format, as shown in Figure 1-13.

Figure 1-13
ASCII Header for AVS Field File

```
# AVS field file
#
ndim   = 2    # number of computational dimensions
dim1   = 512
dim2   = 480
nspace = 2    # number of physical dimension
veclen = 4
data   = byte
field  = uniform
```

This header includes a number of keyword=value pairs, one per line; it also may include comment lines.

The keywords are described below, with reference to the three examples in the preceding sections.

ndim

This value specifies the number of computational dimensions in the field. That is, it specifies the number of dimensions in the field element array. In all the examples above, *ndim* has the value 2.

dim1, dim2, ...

The values for these keywords specify the size of the computational space. For a 2D field, you would specify *dim1* and *dim2* values. In all the examples above, *dim1* = 2 and *dim2* = 5. For a 4D field, specify *dim1*, *dim2*, *dim3*, and *dim4* values.

nspace

This value specifies the dimensionality of the physical space that corresponds to the computational space. In all the examples above, *nspace* = 2. At the end of Example 3, the following mapping to a 3D physical space is suggested:

DATA(2,3)-> (5.5, 4.5, -8.1)

In this case, nspace = 3.

veclen

This value specifies the number of data values for each field element. Example 1 above discussed two possibilities:

```
DATA(2,3)= -5                                   veclen = 1
DATA(2,3)= (2.51, 1.09, 5.73, 0.0, 1.0)veclen = 5
```

data

This keyword takes one of the following values:

- byte
- integer
- real
- double

It indicates the type of data that is supplied for each field element. ConvexAVS fields have the restriction that all of the *veclen* data values must be of the same type.

field

This keyword takes one of the following values: *uniform, rectilinear, irregular.* The main purpose of the three examples above is to illustrate these three field types.

A **uniform field** (as in Example 1) has no computational-to-physical space mapping. The field implicitly takes its mapping from the organization of the computational array of field elements.

For a **rectilinear field** (as in Example 2), each array index in each dimension of the computational space is mapped to a physical coordinate. This produces a physical space whose axes are orthogonal, but the spacing among elements is not necessarily equal.

For an **irregular field** (as in Example 3), there is no restriction on the correspondence between computational space and physical space. Each element in the computational space is assigned its own physical coordinates.

Separator Characters

The ASCII header must be followed by two formfeed characters, in order to separate it from the binary area. A *formfeed* is expressed variously as Ctrl-L, octal 14, decimal 12, or hex 0C.

This scheme allows you to use the *more(1)* command to examine the header. When more stops at the formfeeds, press q to quit. This avoids the problem of the binary data garbling the screen.

Binary Area

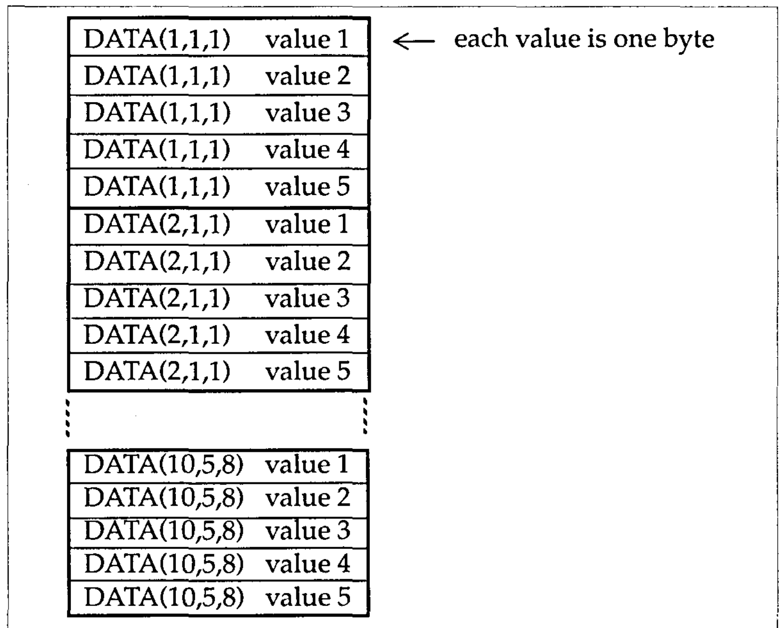
The binary area consists of all the data that is associated with the field elements, along with all the coordinates. (For uniform fields, the coordinates area is null.)

The data area begins with one or more data values for the first field element. All the data values for a field element are stored together. The first array index varies most quickly (FORTRAN-style). For example, suppose the ASCII header is as follows:

```
ndim = 3
dim1 = 10
dim2 = 5
dim3 = 8
nspace=3
veclen=5
data=byte
field=uniform
```

The data ordering can be illustrated as follows:

Figure 1-14
Data Ordering



Volume Data File Format

Measurement data often takes the form of a 3-dimensional array, which corresponds to a uniform lattice in 3D space. Each array value indicates one measurement (temperature, pressure, etc.) at the corresponding lattice point. Such data can be represented as a *uniform 3D field,* as described in the preceding section. For convenience, ConvexAVS also provides a simpler volume data format to accommodate this type of data.

The ConvexAVS volume data format requires that each value in the data array be a byte. For other data types such as single-precision, you must use the more general field construct. Volume data files should have names that end with a *.dat* suffix.

Note

The volume data and field file formats are not compatible.

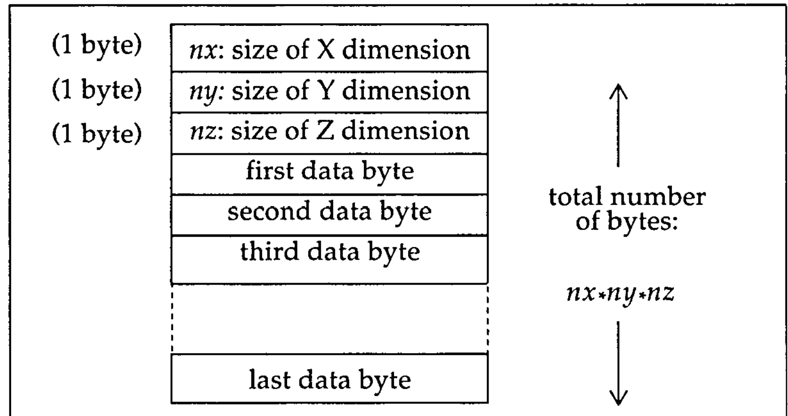
A volume data file begins with a 3-byte header, which specifies the size of the array in the first (X), second (Y), and third (Z) dimensions. Because each dimension's size must be expressed as a 1-byte number, the largest array supported by this file format is 255x255x255.

The remaining contents of the file are the values of a 3D array of bytes, in column-major order (FORTRAN-style). For example, the values in a 50x20x10 array would be stored as follows (using FORTRAN notation):

```
DATA(1,1,1)
DATA(2,1,1)
DATA(3,1,1)
...
DATA(50,1,1)
DATA(1,2,1)
DATA(2,2,1)
DATA(3,2,1)
...
DATA(1,20,1)
DATA(2,20,1)
DATA(3,20,1)
...
DATA(1,20,10)
DATA(2,20,10)
DATA(3,20,10)
...
DATA(50,20,10)
```

Figure 1-15 illustrates the volume data file format.

Figure 1-15
Volume Data File Format



Converting Your Data to a ConvexAVS Format

The previous section contains descriptions of the file formats that are directly readable by the standard ConvexAVS modules. This section discusses strategies for converting your data to these formats.

The ConvexAVS data formats are quite simple. Each involves a short header followed by a stream of data values. There are two basic strategies for getting your data into these formats.

Writing Your Own Conversion Utility

You can use FORTRAN or C (or any other language) to write a program that converts your data to ConvexAVS's field format or volume data format. In addition to the basic conversion task, the program may need to perform some data filtering in order to prevent problems in the ConvexAVS environment. Here are some issues that such a conversion utility should address:

- holes in the data set
- special flag values
- size of the data set

Ideally, your conversion utility will filter out all data that could confuse ConvexAVS or seriously stress its memory allocation system. ConvexAVS includes many useful filter modules, but it is unlikely that they will cover every user's needs.

A drawback of this approach is that you may have to maintain two versions of every data set: the original one (presumably used by other analysis software) and the new one for use by ConvexAVS. With large data sets, this can have a significant impact on your system's data storage capacity.

Writing a ConvexAVS Module

A more elegant approach to converting your data is to write a ConvexAVS module that reads a data set in its original form and outputs a ConvexAVS field. For more information, refer to the *Developing Applications* section.

Creating a Data Directory

You can store the data to be used with ConvexAVS anywhere in the directory hierarchy. By default, ConvexAVS initially reads its data from directory */usr/avs/data*. During program execution, you can make any other directory the current data directory. You can also use command-line options or the ConvexAVS start-up file to specify any directory as the initial data directory. This is explained further in Chapter 2.

Starting ConvexAVS

2

The following sections discuss how to run AVS from any of several displays including the color X server display.

Requirements and Setup Information

The following requirements apply to this release of ConvexAVS:

- Your system must be running ConvexOS V9.0 or later.
- Your system must have IEEE hardware support.
- Your system must be running Convex Network Utilities V9.0 or later.

PseudoColor X Servers

ConvexAVS can be run as an X Window System client on any Convex machine from any color display hardware that meets the following requirements:

- Runs an X server version X11R3 or later
- Has a screen resolution of at least 1024x768
- Its X server has a defined X Window System "visual" that supports one of the following:
  - 8-bit PseudoColor
  - 24-bit TrueColor

  DirectColor is not supported. Monochrome is also not supported at this time.

  This visual must be the default visual. AVS only communicates with an X server's default visual.

Note

ConvexAVS as an X Windows application has control over the client side of the interface only, and sends out the same protocol requests to every X server. Different X servers may respond to the same protocol requests with different behaviors. If unexpected events occur, try the same sequence of actions on a different X server to help isolate the source of the problem.

Caution

Never use an X Window manager to exit from a ConvexAVS window. This will cause the parent program (ConvexAVS) to exit as well. You are free to use the X window manager functions to move, resize, and iconify ConvexAVS windows, but create and close windows with the ConvexAVS functions and buttons.

ConvexAVS is a large application that makes heavy demands on your X server. Besides opening several windows, ConvexAVS will make use of a variety of fonts in different sizes if they are available. This will require that a substantial amount of memory be available for use by the X server. With X terminals that do not have virtual memory access, it is possible to crash the server by making requests that exceed available memory. As a rule of thumb, a minimum of 12 megabytes for workstations and 4 megabytes for X terminals should be available.

To conserve screen real estate, you may prefer to minimize the border decorations added by many window managers on ConvexAVS windows.

If your X server screen size is less than 1280x1024 pixels, use initialization options set in a *.avsrc* file to change the window size. Smaller windows imply smaller fonts, however, some X servers do not have a large set of fonts sizes and may degrade the displayed menus and text windows.

Environment Variables

Before starting, make sure that the following environment variables are properly set:

DISPLAY is used by the X Window System to indicate the display screen at which you are working.

AVS_HELP_PATH
specifies one or more locations in the file system for ConvexAVS to use when searching for online help files. This is a colon-separated list of complete path names. If the variable is set, the **-reindex** option (re)creates *.topic* files in the directory tree under each path name in **AVS_HELP_PATH**. If this variable is not set, **-reindex** (re)creates *.topic* files in the standard directory tree under */usr/avs/runtime/help*.

Command and Command-Line Options

The basic command to start ConvexAVS is:

```
avs
```

There are quite a few options that you can use when issuing the **avs** command. All option keywords begin with a hyphen (for example, **-data**). In many cases, the keyword is followed by an additional word (for example, a directory name). You must separate the keyword and the additional word with white space (SPACE and/or TAB characters).

All options keywords can be abbreviated as long as there is no ambiguity. For example, **-data** can be abbreviated to **-da**. But you cannot abbreviate it to **-d** because this might indicate either **-data** or **-display**.

In several cases, you can use an entry in the ConvexAVS start-up file as an alternative to a command-line option. For example, a **DataDirectory** entry in the start-up file is equivalent to a **-data** option. See the next section for details on the start-up file.

Note

The -geometry option and any associated sub-options must be entered as the last argument on the command line.

-data *directory*

(startup file equivalent: DataDirectory) Specifies the directory in which the ConvexAVS Network Editor subsystem initially will look for data files (files used as input to computational modules). The default data directory is */usr/avs/data*.

-netdir *directory*

(start-up file equivalent: NetworkDirectory) Specifies the directory in which the ConvexAVS Network Editor subsystem initially will look for network files (Read Network and Write Network functions).
The default network directory is */usr/avs/networks*.

-path *directory*

(start-up file equivalent: Path) Specifies the directory tree in which ConvexAVS itself is installed. The default path is */usr/avs*. If you specify another path, then the default data directory and network directory are modified accordingly. For example:

If:	`path`	= */usr/local/avs*
Then:	`data directory`	= */usr/local/avs/data*
	`network directory`	= */usr/local/avs/networks*

-display *display-name*

Specifies the X Window System display on which ConvexAVS is to execute. This overrides the current setting of the **DISPLAY** environment variable.

-geometry *[geom-option(s)]*

Invokes the Geometry Viewer subsystem at startup. You can include the following options that are specific to this subsystem:

-dir *pathname*

Specifies the default directory used by the following **Edit Property** window functions:

Read Object
Save Object
Read Scene
Save Scene
Read
Save

The default data directory is */usr/avs/data* (same as the Network Editor).

-filter *pathname*

Specifies the directory to search for geometry conversion utilities. See the Geometry Conversion Programs appendix.

The default directory for these programs is */usr/avs/filter.*

-scene *scene-file*

Loads a scene from disk storage. This option executes the Geometry Viewer's Read Scene function, using the file *scene-file.scene.*

-geometry *geom_spec*

Specifies an X Window System geometry (for example, 500x500-5-5) for the initial window created by the Geometry Viewer.

-defaults *filename*

Specifies a Geometry Viewer defaults file. The format of this file is described in the Geometry Viewer Script Language Appendix.

-usage

Displays a usage message for the Geometry Viewer options. No ConvexAVS session is started if you type the following line:

```
avs -geometry -usage
```

The ConvexAVS Geometry Viewer options correspond to the command-line options recognized by ConvexAVS.

-image

Invokes the Image Viewer subsystem at startup.

-volume

Invokes the Volume Viewer subsystem at startup.

-network *network-file*

Automatically invokes the Network Editor subsystem at startup and loads the specified network file using the Read Network function.

-modules *directory*

Specifies the directory in which the ConvexAVS Network Editor subsystem initially will look for executable modules. All executable files in the directory are examined to determine whether they contain one or more modules (start-up file equivalent: **ModulesDirectory**).

You can use more than one **-modules** option to have ConvexAVS search through multiple directories for modules. The default modules directory is */usr/avs/avs_library*.

-reindex *[pathname:pathname]* *(Does not start a ConvexAVS session.)*

Causes ConvexAVS to recreate the *.topic* file in the directory tree under each path name in **AVS_HELP_PATH**. If this environment variable is not set, **-reindex** (re)creates *.topic* files in the standard directory tree under */usr/avs/runtime/help*. The optional colon-separated list of path names overrides the **AVS_HELP_PATH** value.

-usage *(Does not start a ConvexAVS session.)*

Displays a usage message for the ConvexAVS options (except for the Geometry Viewer option— see above).

-version *(Does not start a ConvexAVS session.)*

Displays the AVS version number:

```
AVS Version: 2.0P (1.0 CONVEX)
```

ConvexAVS Startup File

When it begins execution, ConvexAVS searches for a start-up file, which specifies the locations of various directories. ConvexAVS looks for the following files, in the order listed:

./.avsrc	(current directory)
HOME/.avsrc	(home directory)
/usr/avs/runtime/avsrc	(system directory)

Only one of these start-up files is read. If ConvexAVS finds one of them, it ignores the others. A */usr/avs/runtime/avsrc* file is included on the ConvexAVS distribution tape.

Startup File Format

Each line of the ConvexAVS start-up file consists of keyword-value pair, with white space separating the keyword and the value. For example:

```
NetworkWindow        867x567+407+2
NetworkDirectory     /usr/henry/avs/nets
DataDirectory        /usr/smith/avs/data
```

In most cases, the keyword corresponds to one of the command-line options described in the preceding section. If you use a command-line option, it overrides the specification, if any, in the start-up file.

The ConvexAVS startup file keywords are:

DataDirectory (command-line equivalent: **-data**)

Specifies the directory in which the various read data modules (read field, read geometry, etc.) initially will look for data files.

DisplayGeometryWindow

Specifies the X Window system geometry of the Geometry Viewer display window.

ModuleLibraries

Specifies the directory in which ConvexAVS initially will look for module files.

ModulePanelHeight

Specifies the initial size of the Network Editor Module Pallet above the Workspace.

NetworkDirectory (command-line equivalent: **-netdir**)

Specifies the directory in which the ConvexAVS Network Editor subsystem initially will look for network files (**Read Network** and **Write Network** functions).

NetworkWindow

Specifies the X Window system geometry of the Network Construction Window that includes the Network Editor menu, the Module Palette, and the Workspace in which you construct networks of modules.

Path (command-line equivalent: **-path**)

Specifies the directory tree in which the ConvexAVS executable or script is installed.

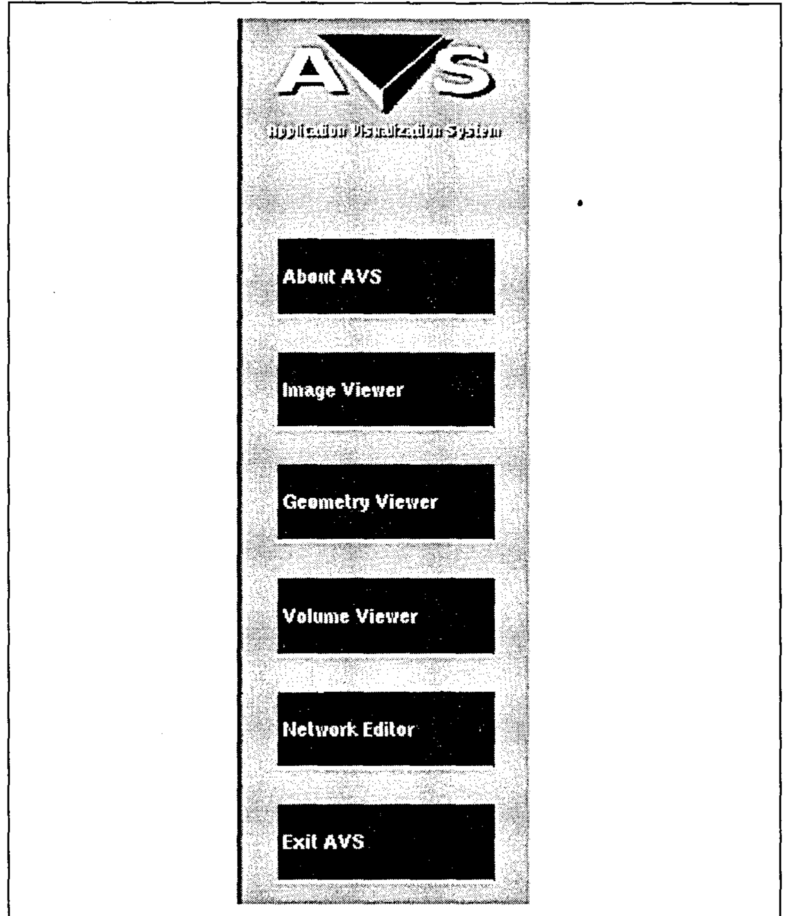
ScreenSize

Specifies a reference screen size, in pixels, for the default visual.

The Main Menu

When you start ConvexAVS, the main menu appears within a control panel along the left edge of the screen (Figure 2-1).

Figure 2-1
ConvexAVS Main Menu



Each of the subsystems has its own control panel (usually, along the left edge of the screen). When you click the **Exit** button at the top of a subsystem's control panel, the ConvexAVS main menu reappears.

Switching Among the Subsystems

You cannot go directly from one ConvexAVS subsystem to another. You must click Exit on the current subsystem menu to return to the ConvexAVS main menu, then select another subsystem.

Note

When you exit a subsystem, your work is not automatically saved. (A dialog box will appear to remind you of this fact.)

If you wish to save your work for later use, be sure to use the appropriate function first (for example, **Write Network** in the Network Editor subsystem).

There are some exceptions to this rule: for instance, you can go directly to the Geometry Viewer subsystem from other subsystems, and return again.

Online Help

ConvexAVS online help consists of **About AVS** tutorials, module editor help, and shell-level help.

Click the Help button in the upper -left corner of each control panel to bring up the online help text viewer. User information for each of the major subsystems and the network editor is provided.

Tutorials

The ConvexAVS main menu includes an About AVS button. Clicking on the button provides information to get you started with ConvexAVS through step-by-step online tutorials.

When you select About AVS, an intial banner dialog is displayed. Click Continue to bring up the Topics menu.

Note

This tutorial viewer is intended to run on an X11R4 version window manager. If you are using an X window manager based on the earlier X11R3 version, the windows used with this tutorial may not stack properly and cannot be moved.

Once you have chosen a tutorial, a reader pop-up is displayed. The top half of the menu displays the major topics for the current session. We suggest that you run through them in sequence if this is the first time you have used ConvexAVS.

To move to the next step, click the Next Step button. Use the Previous Step button to move backwards through the session steps. Click Close to end the current session topic. Click Quit to end the Tutorial Browser.

Module Editor

Each computational module in ConvexAVS is represented on-screen by an icon. Clicking on the small square at the right side of the icon with the middle or right mouse button opens a Module Editor window that displays minimal information about the module. Clicking on the **Show Module Documentation** box pops up a **Help Browser** that displays the complete manual page for that module.

Shell-Level Help

The manual pages are also available through the shell command *man(1)*. For example, typing `man colorize` at a shell prompt displays the manual page for the colorize module.

Image Viewer Subsystem

3

The ConvexAVS Image Viewer subsystem provides access to a set of visualization networks specifically created for manipulating 2D images. You don't need to do any network construction at all— just select the network you want from the Image Viewer menu system.

Features

The Image Viewer implements the following features:

Fast Start

Because the networks have already been created, you can transform your data into a visualization display quickly. In most cases, just a few menu-choice mouse clicks is all it takes.

Replication and Comparison

It is easy to view different data sets side-by-side, using the same visualization technique. Similarly, you can view the same data set in several windows, each using different input parameter settings.

Resource Sharing among Networks

When you invoke several networks to perform different types of visualization (or the same type on different data sets), the Image Viewer can multiplex resources among the networks. For example, a single mouse click can take the data set from one network and plug it into another network. This capability extends to the networks' display windows for visualization output and their colormaps, too.

Connections to Other ConvexAVS Subsystems

It is easy to make the transition from using the Image Viewer's pre-existing visualization networks to creating your own. At any time, you can pop up a window that shows the network

being executed. Moreover, you can instantly switch to the Network Editor subsystem in order to refine or extend the network. In addition, you can instantly switch from the Image Viewer to the Geometry Viewer subsystem.

Using the Image Viewer

This section presents a quick overview of the Image Viewer's typical pattern of usage. It assumes that your data is already in the ConvexAVS image format that is directly readable by ConvexAVS data modules. Be sure to consult the Image Data File Format section in Chapter 1 before attempting to use your data with the Image Viewer.

Typical usage of this subsystem includes the following steps:

1. **Entering the Image Viewer Subsystem.** You can enter the Image Viewer either from the shell or from the ConvexAVS main menu.
2. **Preprocessing the Data.** In some cases, you may want to perform some transformation on your input data (for example, extracting a subset) before visualizing it. This step reads a data file and creates a new file containing the transformed data.
3. **Selecting a Visualization Technique.** The Image Viewer offers a wide variety of techniques for processing images. When you select one by clicking its menu choice, ConvexAVS automatically loads a network that implements the technique. (An empty display window appears, in which the visualization output will be drawn later.)
4. **Selecting the Data.** Each image viewer network begins by reading an image format data file from disk. When you select a data file (perhaps one you created in step #2 above), the data flows through the network and an image appears in the display window.
5. **Adjusting the Parameters.** In general, each module in the underlying network has input parameters whose values you can adjust with on-screen control widgets. As you work with these dials, sliders, etc., the visualization image in the display window changes accordingly.

The following sections discuss these steps in more detail. This pattern of usage is not the only way in which you can use the Image Viewer, as explained in further sections.

Entering and Leaving the Image Viewer

There are two ways to enter the Image Viewer subsystem:

From the shell

The following command line invokes the Image Viewer when ConvexAVS starts execution:

```
avs -image
```

From the ConvexAVS main menu

Click the Image Viewer choice on the ConvexAVS main menu.

The Image Viewer starts by displaying its control panel along the left edge of the screen (Figure 3-1).

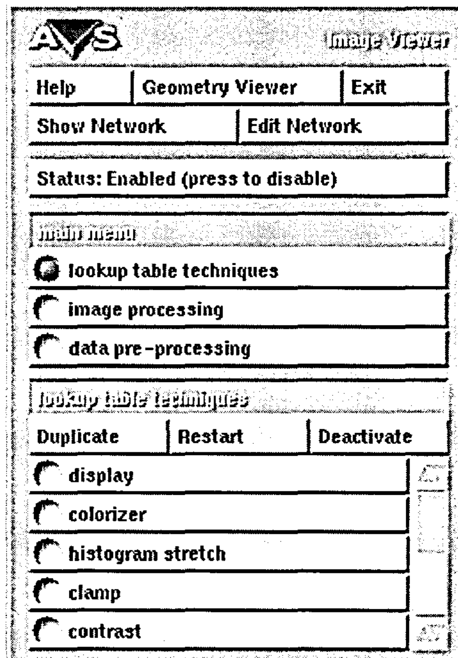


Figure 3-1
Image Viewer Control Panel

To leave the Image Viewer, click the **Exit** button at the top of the control panel. Control reverts either to the ConvexAVS main menu or to the shell, depending on how you entered the subsystem.

Using the Data Preprocessors

In some situations, you may want to preprocess your data before visualizing it. With the Image Viewer, the strategy is to take a data file, preprocess it, and create another data file. The second file can then be used as input to one or more of the visualization techniques.

In the Network Editor subsystem, you can create networks that preprocess data on the fly, so that you need not create additional disk files containing the preprocessed data.

To start, click **data preprocessing** on the Image Viewer top-level menu. Each of the preprocessing functions provides access to a ConvexAVS filter module with the same name:

crop

Extracts a range of elements from a field. This is useful for discarding unwanted data. You can also use this to perform a quick analysis on a subset of the data in preparation for a more time-consuming analysis on the complete data set.

downsize

Reduces the size of a data set by sampling every *n*th element in each dimension. Like crop, it is useful for performing quick analyses on subsets.

mirror

Produces a mirror image of a data set.

transpose

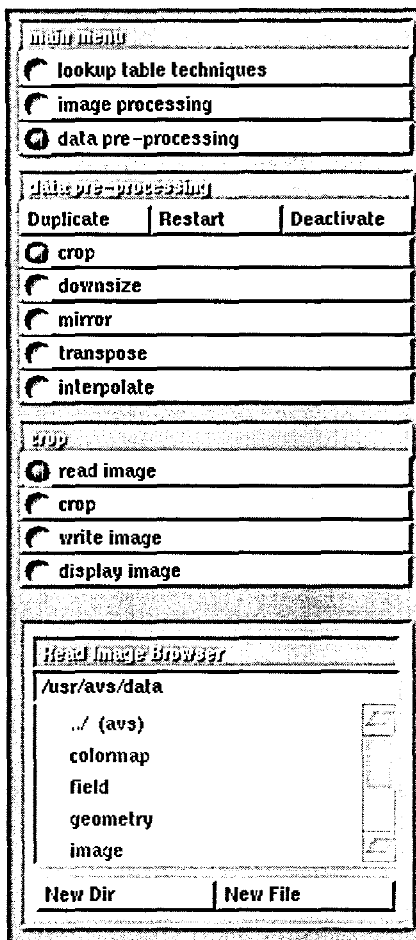
Exchanges the dimensions in a 2D or 3D data set.

interpolate

Reduces the size of a 2D or 3D data set by performing an interpolation.

When you select one of these functions, a four-choice menu appears in the control panel as shown in Figure 3-2.

Figure 3-2
Crop Menu



The **Read Image** choice (automatically selected initially) pops up a File Browser window in which you can specify the data set to be preprocessed.

Click the second choice (for example, **crop**) to display a set of control widgets that interactively control the preprocessing operation (for example, to select which part of the data set is to be cropped). As you work the controls, the potential results of the preprocessing operation are displayed in a display window.

When you have adjusted the controls to your liking, click the **Write Image** choice to pop up another File Browser. Specify a file in which to store the preprocessed data.

Note

If you have any problems figuring out how to use the browsers, see the "Network Editor" chapter.

Selecting Visualization Techniques

The Image Viewer's visualization techniques are organized into two submenus:

- **Lookup Table Techniques:**
  - display
  - colorizer
  - histogram stretch
  - clamp
  - contrast
- **Image Processing:**
  - gradient shade
  - image transform
  - 3D mesh

Initially, the **Lookup Table Techniques** category is selected so its five choices are visible. You can click the **Image Processing** button in the Image Viewer main menu to display the three choices in this category. To select one of the visualization techniques, just click on it.

Selecting a technique invokes a ConvexAVS network that includes several modules. In general, however, there is a single main component at the heart of the network— the module that does the real work. For example, the colorizer and histogram stretch techniques invoke networks that are essentially packages for the like-named modules.

Lookup Table Techniques

display

Displays the input data, a ConvexAVS image, unchanged.

colorizer

The input data is passed through a color lookup table whose contents are controlled by a colormap control widget.

histogram stretch

Equalizes the distribution of values between the parameter-controlled minimum and maximum values. Values outside these bounds are left unchanged.

clamp

Establishes parameter-controlled minimum and maximum values for the data. Pixel values below the minimum value are set to *min*; pixels above the maximum value are set to *max*.

contrast

Stretches the range of the input data linearly between parameter-controlled min and max output values. It can be used to increase the contrast of an image or to invert an image.

Image Processing Techniques

gradient shade

First determines the gradient of the surface of the image at each pixel. (This is done by taking the X and Y values between neighboring pixels and using a parameter-controlled Z value.) Then shades the pixel values according to the gradient. This produces an image with 3D characteristics.

image transform

Maps the input data onto a rectangle that can be arbitrarily transformed in three dimensions.

3D mesh

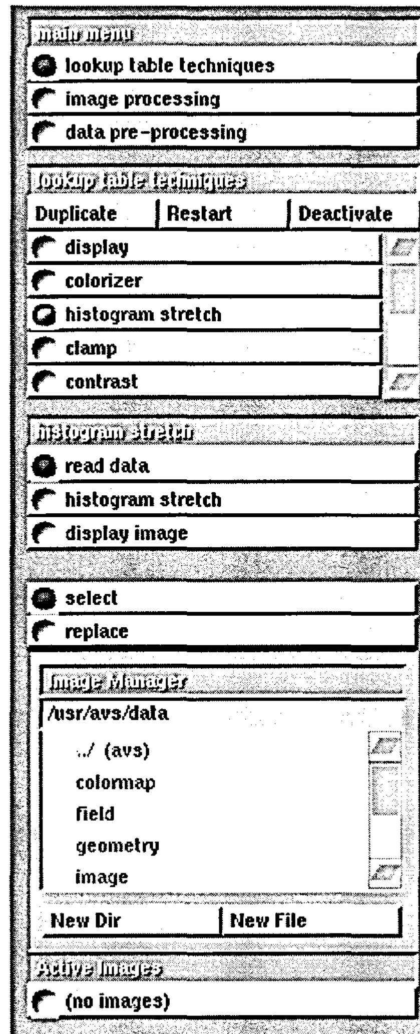
Creates a 3D mesh from the input data with X and Y values determined by the pixel ordering and Z values proportional to the value of the pixel.

Running the Network

This section discusses in more detail how you work with any one of the Image Viewer's visualization techniques. (You can also use several techniques at once— this subject is discussed in a subsequent section.)

When you select one of the visualization techniques, a choice menu (radio buttons) appears in the control panel and an empty display window is created to the right of the control panel. For example, if you select **histogram stretch**, the control panel appears (Figure 3-3).

Figure 3-3
Control Panel for Histogram Stretch



The choice menu has an entry for each module in the network that has input parameters. You can click the choices to switch back and forth among the pages of control widgets. (The empty display window created when you select a technique is not represented in the choice menu. Its controls are in a pull-down menu, accessed via the small square in the window's title bar—see next section.)

Note

If you wish to see the network that implements a technique, click the Show Network button at the top of the control panel. This creates a window showing the network. You may want to leave this window open as you use the visualization technique. Each module icon flashes as the module executes; if a module fails, its icon turns black. If you deactivate or switch visualization techniques (described later), this window is updated accordingly.

Typically, you start by selecting the data to be visualized. As shown in Figure 3-3, the read data choice is initially selected (corresponding to the Image Manager module in the network), and its File Browser control widget appears below the menu. When you specify an image to be read, it flows through the network and the image appears in the display window. This window automatically resizes itself to accommodate the image you select.

Note that whenever data is flowing through the network, the **Status bar** near the top of the control panel turns red and displays a message that describes the current activity. If you wish to disable this feature, just click on the status bar with any mouse button; another click re-enables this feature.

The next typical step is to exercise the controls of the module that is the main component of the network. For the histogram stretch technique, click on the **histogram** choice to bring up its page of control widgets.

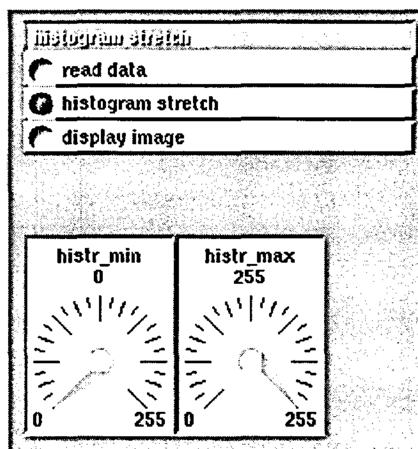


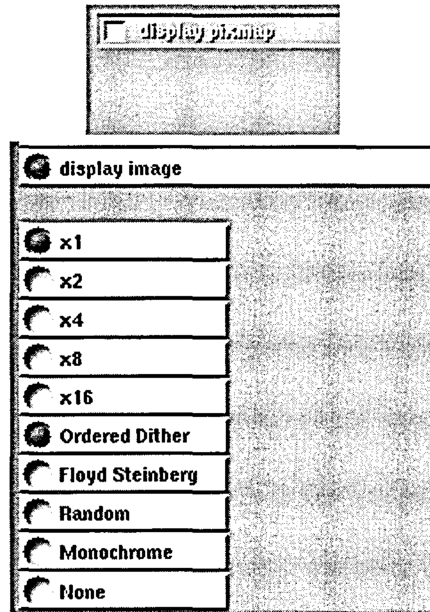
Figure 3-4
Control Widgets for histogram

As you adjust the dials (or other control widgets) with the mouse, the image in the display window changes accordingly. If you need guidance using any of the control widgets, consult the on-line manual page for the associated module.

Working with the Display Window

The display window for a visualization technique is created by the last module in the network, either **display image** or **display pixmap**. You can move, resize, and iconify this window using the X Window System window manager. There are also ConvexAVS-level controls for the pixmap window; they are in a pull-down menu attached to the small square in the window's title bar (Figure 3-5).

Figure 3-5
Display Window Pull-down button and Radio Button Controls



Scaling controls are available through radio buttons that appear in the control panel when an image manager display is created. The display window is designed for flexibility: it automatically resizes itself when the image size changes. This can occur when you explicitly magnify an image or zoom it to full-screen size. It can also occur when you switch data sets to produce a different output image.

Deactivating or Restarting a Technique

Just above the top-level techniques menu are three buttons: **Duplicate**, **Restart**, and **Deactivate**. When you are finished using a technique, but you don't want to exit the Image Viewer, click **Deactivate** to close down the currently active network. All its control panels and display windows disappear.

If you don't plan to return to a technique, it is advisable to deactivate it. This saves memory usage and processes, both of which are finite resources.

Note

When you deactivate a technique, no other technique becomes current (no technique gets the light blue ball). You must make another technique current by clicking on its submenu entry.

Clicking **Restart** returns the current network to its initial state: no input data selected, all controls in their initial positions, and an empty display window.

The **Restart** button can be very useful in situations where one of the network's modules fails to load, hangs, or crashes. Pressing this button does not allow you to recover your work, however.

Using More Than One Technique

One of the most powerful and flexible aspects of the Image Viewer is that you can invoke several visualization techniques at once. For instance, you might select **colorizer** and then select **contrast**.

Note

All selected techniques are marked with a dark blue ball. One of the selected techniques is current and is marked with a light blue ball.

When two or more techniques are selected, each can have its own input data set. But all the data sets that you have read in are accessible by all the techniques. The Active Images list includes every data set that you have read. With any technique, you can switch to another active data set simply by clicking on it in this list. This makes it easy to apply several different techniques to the same data set.

To switch back and forth among two or more active techniques, just click on the technique name in the submenu. Remember that the light blue ball always indicates the currently selected technique. There is no problem with having techniques from different submenus active at the same time.

Duplicating a Technique

In some situations, you may want to invoke the same technique more than once. For example, you might want to create three windows side by side, each showing the same image, but with different contrast settings. Make sure that the technique you wish to invoke more than once is currently selected (is marked with a light blue ball). Then click the **Duplicate** button once or more. For each click, a copy of the technique is entered in the submenu. (A digit is appended to the name so that you can distinguish the different copies.) You can now switch back and forth among the copies and share data among them.

Example

The following sequence shows how you might use three different images with two of the visualization techniques:

1. Click **colorizer** to invoke that technique.
2. Use the read image file browser to load the first image from file *mandrill.x*. The image appears in the display window and the name *mandrill* is placed in the Active Images list at the bottom of the control panel
3. Click **contrast** to invoke that technique.
4. Use the **read image** file browser to load a second image from file *marble.x*. Because the **Select** button just above the file browser is highlighted, this image is added to the Active Images list. The image appears in a second display window; there are now two display windows.
5. Click **colorizer** again to switch back to the control panel for that technique. (Note how the light blue ball always indicates the currently selected technique.) The **Active Images** list is the same as that for contrast, containing both *marble* (currently selected) and *mandrill*.
6. Click *marble* to make it the current image for the **colorizer**. Now, both display windows show the same image.
7. Click the **Replace** button just above the file browser, then select the third image file, *convex.x*. In the **Active Images** list, convex replaces the current image, *marble*, instead of being added to the list. Because *marble* had been the current image for both techniques, the *convex* image takes its place in both display windows.

Additional Image Viewer Features

The following function buttons appear at the top of the Image Viewer control panel:

Help

Pops up a Help Browser, allowing you to view a collection of help files for the Image Viewer.

Show Network

Click this button after selecting a visualization technique to pop up a window that shows the ConvexAVS network used to implement the technique. Click this button again to close the window.

Edit Network

Clicking this button goes one step further than Show Network: it actually invokes the Network Editor on the network that implements the currently-selected technique.

Geometry Viewer

Switches immediately to the Geometry Viewer subsystem.

Volume Viewer Subsystem

4

The ConvexAVS Volume Viewer application provides access to a set of visualization networks specifically created for 3D scalar volume data. No special network construction is required. Just select the network you want from the Volume Viewer menu.

Features

The Volume Viewer implements the following features:

Fast Start

Because the networks have already been created, you can transform your data into a visualization display quickly. In most cases, just a few menu-choice mouse clicks is all it takes.

Replication and Comparison

It is easy to view different data sets side by side, using the same visualization technique. Similarly, it is easy to view the *same* data set in several windows, each using different input parameter settings.

Resource Sharing among Networks

When you invoke several networks to perform different types of visualization (or the same type on different data sets), the Volume Viewer can multiplex resources among the networks. For example, a single mouse click can take the data set from one network and plug it into another network. This capability extends to the networks' display windows for visualization output and their colormaps, too.

Connections to Other ConvexAVS Subsystems

It is easy to make the transition from using the Volume Viewer's pre-existing visualization networks to creating your own. At any time, you can pop up a window that shows the network being executed. You can also instantly switch to the Network Editor in order to refine or extend the network.

In addition, you can instantly switch from the Volume Viewer to the Geometry Viewer subsystem.

Using the Volume Viewer

This section presents a quick overview of the Volume Viewer's typical pattern of usage. It assumes that your data is already in the ConvexAVS volume format that is directly readable by ConvexAVS data modules. Be sure to consult the "Volume Data File Format" section in Chapter 1 before attempting to use your data with the Volume Viewer.

Typical usage of this subsystem includes the following steps:

1. **Entering the Volume Viewer Subsystem.** You can enter the Volume Viewer either from the ConvexOS shell or from the ConvexAVS main menu.
2. **Preprocessing the Data.** In some cases, you may want to perform some transformation on your input data (for example, extracting a subset) before visualizing it. This step reads a data file and creates a new file containing the transformed data.
3. **Selecting a Visualization Technique.** The Volume Viewer provides a variety of techniques for processing volumes. When you select one by clicking its menu choice, ConvexAVS automatically loads a network that implements the technique. (An empty display window appears, in which the visualization output will be drawn later.)
4. **Selecting the Data.** All the volume viewer networks begin by reading an volume format data file from disk. When you select a data file (perhaps one you created in step #2 above), the data flows through the network and an volume appears in the display window.
5. **Adjusting the Parameters.** In general, each module in the underlying network has input parameters whose values you can adjust with on-screen control widgets. As you work with these dials, sliders, etc., the visualization volume in the display window changes accordingly.

The sections below discuss these steps in more detail. Because this pattern of usage is not the only way in which you can use the Volume Viewer, there is also a section that discusses features that are not covered in the above recipe.

Entering and Leaving the Volume Viewer

There are two ways to enter the Volume Viewer subsystem:

From the ConvexAVS main menu

Click the **Volume Viewer** choice on the ConvexAVS main menu.

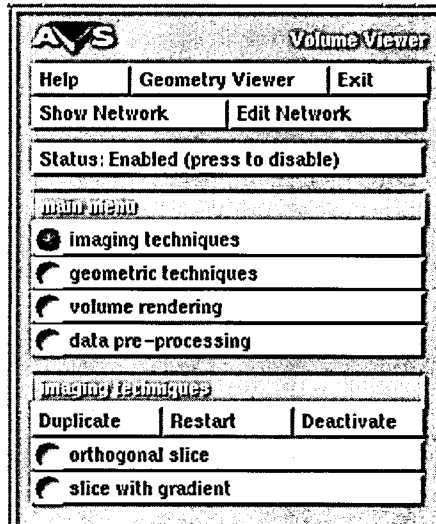
From the ConvexOS shell

The following command line invokes the Volume Viewer automatically when ConvexAVS starts execution:

```
avs -volume
```

The Volume Viewer starts by displaying its control panel along the left edge of the screen (Figure 4-1).

Figure 4-1
Volume Viewer Control Panel



To leave the Volume Viewer, click the **Exit** button at the top of the control panel. Control reverts either to the ConvexAVS main menu or to the shell, depending on how you entered the subsystem.

Selecting Visualization Techniques

The Volume Viewer's visualization techniques are organized into the following submenus:

- Imaging Techniques:
  - orthogonal slice
  - slice with gradient
- Geometric Techniques:
  - arbitrary slice
  - dot surface
  - isosurface tiler
  - volume bounds
- Volume Rendering:
  - vbuffer
- Data Preprocessing:
  - crop
  - downsize
  - mirror
  - transpose
  - interpolate

Imaging Techniques

orthogonal slice

Takes a 2D slice from the 3D input data array by holding the array index in one dimension constant and letting the other indices vary. The resulting 2D array of values is converted to colors by passing it through a colormap.

slice with gradient

Same as orthogonal slice but with additional processing of the output image. The gradient of the surface of the image is determined at each pixel. (This is done by taking the X and Y values between neighboring pixels and using a parameter-controlled Z value.) Then the pixel values are shaded according to the gradient. This produces an image with 3D characteristics.

Geometric Techniques

arbitrary slice

Maps the input data to a uniform 3D lattice then slices through this volume with a plane. The data values at the (approximated) intersection points are used to calculate a 3D mesh. The orientation of the slice plane is parameter-controlled allowing you to position it arbitrarily within the volume. Also parameter-controlled are the resolution of the grid of points that define the slice plane and the sampling technique: nearest neighbor or trilinear interpolation.

dot surface

Creates an isosurface all the elements that have a parameter-controlled data value -- representing this surface as a mesh of dots.

isosurface tiler

The input data is processed with a marching-cubes algorithm to create a surface showing the elements with a parameter-controlled data value.

volume bounds

Draws lines that show the 3D bounding box for the input data.

Volume Rendering

vbuffer

Performs volumetric rendering of a 3D uniform scalar field using interpolation, color shading, and transparency-processing algorithms.

Data Preprocessing

In some situations, you may want to preprocess your data before visualizing it. With the Volume Viewer, the strategy is to take a data file, preprocess it, and create another data file. The second file can then be used as input to one or more of the visualization techniques.

In the Network Editor subsystem, you can create networks that preprocess data on the fly, so that you need not create additional disk files containing the preprocessed data.

To start, click **data preprocessing** on the Volume Viewer top level menu. Each of the preprocessing functions provides access to a ConvexAVS filter module with the same name:

crop

Extracts a range of elements from a field. This is useful for discarding unwanted data. You can also use this technique to perform a quick analysis on a subset of the data in preparation for a more time-consuming analysis on the complete data set.

downsize

Reduces the size of a data set by sampling every *n*th element in each dimension. Like crop, it is useful for performing quick analyses on subsets

mirror

Produces a mirror image of a data set.

transpose

Exchanges the dimensions in a 2D or 3D data set.

interpolate

Reduces the size of a 2D or 3D data set by performing an interpolation.

These same data preprocessing techniques are also available in the Image Viewer subsystem.

Running the Network

This section discusses in more detail how you work with any one of the Volume Viewer's visualization techniques. (You can also use several techniques at once— this subject is discussed in a subsequent section.)

When you select one of the visualization techniques, a choice menu (radio buttons) appears in the control panel and an empty display window is created to the right of the control panel. For example, if you select **Orthogonal slice**, the control panel appears (Figure 4-2).

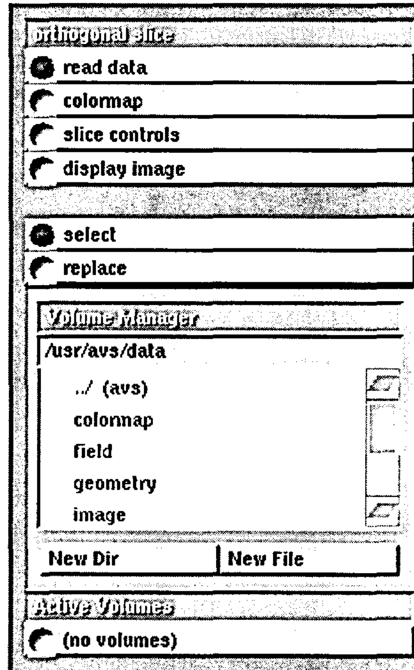


Figure 4-2
Control Panel for Orthogonal Slice

The choice menu has an entry for each module in the network that has input parameters. You can click the choices to switch back and forth among the pages of control widgets. (The empty display window created when you select a technique is not represented in the choice menu. Its controls are in a pull-down menu, accessed via the small square in the window's title bar— see next section.)

Note

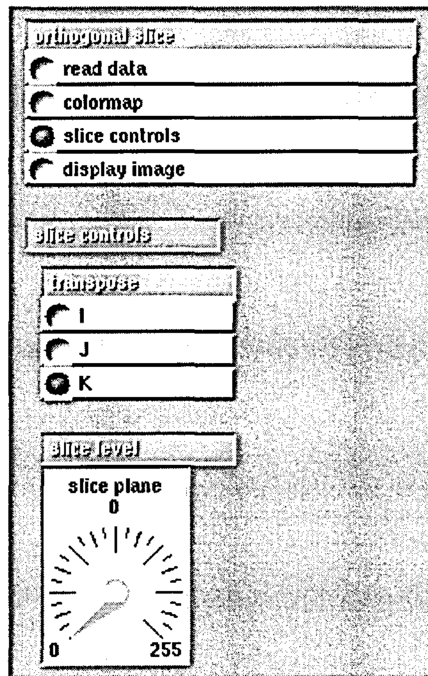
If you wish to see the network that implements a technique, click the *Show Network* button at the top of the control panel. This creates a window showing the network. You may want to leave this window open as you use the visualization technique. Each module icon flashes as the module executes; if a module fails, its icon turns black. If you deactivate or switch visualization techniques (described later), this window is updated accordingly.

Typically, you start by selecting the data to be visualized. As shown in Figure 4-2, the **read data** choice is initially selected (corresponding to the Volume Manager module in the network), and its File Browser control widget appears below the menu. When you specify an volume to be read, it flows through the network and the volume appears in the display window. This window automatically resizes itself to accommodate the volume you select.

Note that whenever data is flowing through the network, the status bar near the top of the control panel turns red and displays a message that describes the current activity. If you wish to disable this feature, just click on status bar with any mouse button; another click enables this feature again.

The next typical step is to exercise the controls of the module that is the main component of the network. For the Orthogonal slice technique, click on the **orthogonal slice** choice to bring up its page of control widgets.

Figure 4-3
Control Widgets for orthogonal slice



As you adjust the dials (or other control widgets) with the mouse, the picture of the volume in the display window changes accordingly. If you need guidance using any of the control widgets, consult the on-line manual page for the associated module.

Working with the Display Window

The display window for a visualization technique is created by the last module in the network, either **display image** or **display pixmap**. You can move, resize, and iconify this window using the X Window System window manager.

The display window is designed for flexibility: it automatically resizes itself when the image size changes. This can occur when you explicitly magnify an image or zoom it to full-screen size. It can also occur when you switch data sets to produce a different output volume.

There may be cases in which the image does not exactly fit inside the display window. When the volume is too big for the window, you can move the image simply by grabbing-and-dragging, using any mouse button.

Deactivating or Restarting a Technique

Just above the top-level techniques menu are three buttons: **Duplicate**, **Restart**, and **Deactivate**. When you are finished using a technique, but you don't want to exit the Volume Viewer, click **Deactivate** to close down the currently active network. All its control panels and display windows disappear.

If you don't plan to return to a technique, it is advisable to deactivate it. This saves memory usage and processes, both of which are finite resources.

Note

When you deactivate a technique, no other technique becomes current (no technique gets the light blue ball). You must make another technique current by clicking on its submenu entry.

Clicking **Restart** returns the current network to its initial state: no input data selected, all controls in their initial positions, and an empty display window.

The **Restart** button can be very useful in situations where one of the network's modules fails to load, hangs, or crashes. Pressing this button does not allow you to recover your work, however.

Using More Than One Technique

One of the most powerful and flexible aspects of the Volume Viewer is that you can invoke several visualization techniques at once. For instance, you might select **colorizer** and then select **contrast**. Note that all selected techniques are marked with a *blue* ball. One of the selected techniques is current and is marked with a *light blue* ball.

When two or more techniques are selected, each can have its own input data set. But all the data sets that you have read in are accessible by all the techniques. The Active Volumes list includes every data set that you have read. With any technique, you can switch to another active data set simply by clicking on it in this list. This makes it easy to apply several different techniques to the same data set.

To switch back and forth among two or more active techniques, just click on the technique name in the submenu. Remember that the light blue ball always indicates the currently selected technique. There is no problem with having techniques from different submenus active at the same time.

Duplicating a Technique

In some situations, you may want to invoke the same technique more than once. For example, you might want to create three windows side by side, each showing the same volume, but with different contrast settings. Make sure that the technique you wish to invoke more than once is currently selected (is marked with a light blue ball). Then click the **Duplicate** button once or more. For each click, a copy of the technique is entered in the submenu. (An extension digit is appended to the name so that you can distinguish the different copies.) You can now switch back and forth among the copies and share data among them.

Additional Volume Viewer Features

The following function buttons appear at the top of the Volume Viewer control panel:

Help

Pops up a Help Browser, allowing you to view a collection of help files for the Volume Viewer.

Show Network

Click this button after selecting a visualization technique to pop up a window that shows the ConvexAVS network used to implement the technique. Click this button again to close the window.

Edit Network

Clicking this button goes one step further than Show Network: it actually invokes the Network Editor on the network that implements the currently-selected technique.

Geometry Viewer

Switches immediately to the Geometry Viewer subsystem.

Geometry Viewer Subsystem

5

The ConvexAVS Geometry Viewer subsystem enables you to manipulate and view one or more 3D objects that have been created through geometry library primitives.

Entering and Leaving the Geometry Viewer

This subsystem can be invoked in several ways:

From the shell directly

The following command line invokes the Geometry Viewer automatically when ConvexAVS starts execution:

```
avs -geometry
```

See the "Starting ConvexAVS" chapter for additional command-line options that affect the manner in which the Geometry Viewer is invoked.

From the ConvexAVS main menu

Click the Image Viewer choice on the ConvexAVS main menu.

From another subsystem

The Geometry Viewer is the one subsystem to which there is direct access from all the other subsystems. There is a Geometry Viewer button at the top of each subsystem's control panel.

In a network

There are two ways to go directly from the Network Editor to the Geometry Viewer:

- Click the **Geometry Viewer** button at the top of the Network Construction Window. You can return to the Network Editor by clicking the **Close** button in the Geometry Viewer menu.
- If a network includes the **render geometry** module, use the left mouse button to click the small square box in the icon for

The Geometry Viewer starts by displaying its control panel at the left edge of the screen (Figure 5-1). Only one menu is displayed, even though there may be multiple **render geometry** modules in a network.

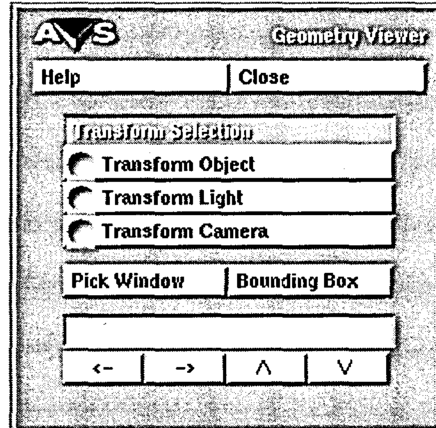


Figure 5-1
Geometry Viewer Control Panel

Help

Select the **Help** button for on-line help information about the Geometry Viewer.

Close

Select the **Close** button to close the Geometry Viewer Menu.

Pick Window

The **Pick Window** button toggles the display area in the menu for each object that you have currently selected in the Display area. A miniature version of each object is displayed; clicking in the menu display area brings each object forward, one at a time.

Hierarchical Picking Controls

Use the arrow buttons located above the pick window select the current object. Click the **up** arrow to cycle through the various parent objects. Click the **left** and **right** arrows to cycle through any children of a parent object. Use the **down** arrow to bring up the last active child object (if one was previously selected).

Bounding Box

Real-time rendering of moving 3D objects is very compute-intensive. If you try to rotate or move a 3D object, light, or camera in the Geometry viewer with the middle mouse button in "real-time" on an X Server, depending on the complexity of the object or scene, you can experience a noticeable delay while the program catches up with the flood of data.

The **Bounding Box** switch disables the compute intensive real time rendering. With **Bounding Box** turned on, when you place the mouse cursor over the current object and press the middle or right button, a wireframe box enclosing the volume of the object appears. As you move the mouse, the bounding wireframe box moves -- the object does not. You move the bounding box to the destination position/rotation/scale, then release the mouse button. Only then is the object rendered at its new location and orientation. This option is **on** by default.

Geom Format Utilities for Objects

Whichever way you invoke the Geometry Viewer, you work with objects that are represented in the ConvexAVS **geom** format. Such objects must have previously been created and stored in a file, using the ConvexAVS *libgeom* programming library. (If you invoke the **render geometry** module in a network, it can input geometries created and/or revised by upstream modules.) ConvexAVS includes a number of utility programs that read standard data formats and create *geom*-format files. The standard formats include the following types:

- Mathmatica ThreeScript
- Wavefront
- MovieBYU
- Brookhaven Protein Data Bank
- Polygen Protein Data Bank
- Convex internal format

Refer to Appendix A, "Geometry Conversion Programs" for more information.

Using the Geometry Viewer

The following sections provide an overview of the way the Geometry Viewer works.

Scenes: Objects, Lights, Cameras

Using the Geometry Viewer, you can work with one or more scenes. Each scene consists of:

- A collection of 3D objects, assembled into a single coordinate system (world coordinate space). Objects have attributes, such as surface color, light reflectance characteristics, and a rendering method. You can selectively hide objects, so that they are temporarily invisible, although still part of the scene.
- A collection of lights, defined in the same world coordinate space. Each light can be a different color.
- One or more view windows, each of which provides its own view of the collection of objects, as they are illuminated by the collection of lights. Each view window is considered to be a camera viewing the objects.

  Different cameras can produce different views, because each can have its own position in world coordinates. In addition, cameras can vary in the way they display lines.

Several scenes may be displayed at the same time. You can manipulate the various view windows with Geometry Viewer functions, such as **Create Camera**. You can also manipulate the windows with an X Window System window manager program.

Note

If you invoke the Geometry Viewer as the read geometry module in a ConvexAVS network, you may want to integrate the view window(s) into the network's overall user interface. For details, see Including Output Windows in a Reorganized Layout in the Network Editor chapter.

Within each view window, you can translate, rotate, and scale objects, using the system mouse. If a scene has several view windows (cameras), then the objects move in all of them. Likewise, changing the lighting affects all the windows that belong to a scene.

Data Formats

The following sections describe the data formats used by the Geometry Viewer.

Geometries

The Geometry Viewer's fundamental data structure is the geometry: a collection of points in 3D space, along with additional information (typically, indicating connectivity). The geometry defines a simple or complex 3D object, with the specified points as its vertices. (A geometry can also include a color and/or *normal* for each vertex.)

Each geometry is stored in its own file, with a *.geom* extension. ConvexAVS includes a small library of *.geom* files. It also includes a data filter facility, with which you can create your own *.geom* files, either from scratch or from geometric data in a number of commonly-used formats (e.g. *Movie.BYU*).

Note

There is no way to define a new geometry or modify an existing one from within the viewing application.

Objects

Using the Geometry Viewer, you can start with a simple object and adjust your view of it by specifying various attributes, or properties:

- The position and orientation in 3D space of the object
- The surface color of the object
- The way in which the surface reflects light (including specular highlights) on the object
- The rendering method to be used in drawing the geometry

A geometry that is customized with these property specifications is called an *object*. Each object is stored in its own file, with a *.obj* extension. This is an ASCII-format file, expressed in the Geometry Viewer Script Language. An object file is created when you customize a geometry, and save it with the **Save Object** function. (You can also store the properties separately, in a file with a *.prop* extension.)

You can create equivalent object files with a text editor, using the Script Language directly.

You can also create hierarchical composite objects, which include several or many geometries. You can specify the properties listed above for the individual-geometry level or at various levels of the hierarchy. If a geometry does not have its own properties set, it inherits them from the next level up (or some higher level).

With the Geometry Viewer, you create only a two-level hierarchy— a top-level object with a number of objects as its children. The Script Language gives you more flexibility (at the expense of interactivity). A script can define multiple-level hierarchies, using the **group** command. Refer to Appendix B, "The Geometry Script Language."

Scenes

Just as you build up basic geometries into objects, you compose objects into scenes. Like objects, scenes are represented in the Script Language. You can build a scene interactively with the viewing application:

- Adding and manipulating the positions of several objects
- Adding and positioning lights
- Defining one or more cameras to view what you've built

When you select **Save Scene**, a script is written to a file with a *.scene* extension. You can also create equivalent scene files with a text editor, using the Script Language directly.

Note

The binary-format geometry files, which specify the vertices of objects, are the basic building blocks for the Geometry Viewer. These are the only files in which geometric definitions are stored. The ASCII-format object and scene files, written in the Script Language, include references to geometry files, along with other specifications. This means you must be careful not to disturb geometry files that are used as building blocks for objects and scenes. If, for instance, you rename or move the file teapot.geom, it will invalidate all objects and scenes that include the teapot defined therein.

File Type Summary

The following table summarizes the types of files that the Geometry Viewer can read directly:

Table 5-1
Geometry Viewer File Types

File Type	Extension	Format	Contents of File
Geometry	*.geom*	binary	Describes a single geometric object (simple or complex). Can include per-vertex data normals and/or colors.
Property	*.prop*	ASCII	Specifies a set of surface attributes, including color and light-reflectance characteristics.
Object	*.obj*	ASCII	Specifies the names of one or more geometry files, along with surface attributes and rendering methods. Can also define a hierarchy that includes geometries and other objects.
Scene	*.scene*	ASCII	Specifies objects (as in a .obj file), along with light(s) and camera(s).

Menu Choices, Sliders, and Function Keys

You can use any mouse button to make menu choices. A single click of the mouse button anywhere within a choice's rectangle makes the selection.

The **Edit Property** window includes a number of sliders, as do the **Light and Camera** menu selection areas. You can move a slider either by dragging it with any mouse button, or by clicking once at the place to which you want the slider to move.

ConvexAVS makes use of several of the keyboard's function keys. This is described under "Function Key Usage" in a later section.

Transformations and the Transform Selection Area

One of the most powerful features of the Geometry Viewer is that it allows you to interactively transform aspects of the scenes you create. For example, you can change the positions and sizes of objects, the positions and types of lights, the placement of the camera in the scene, and so on. You control these transformations with the mouse. At any moment, the mouse is attached to the current transformable, which is one of the following:

- An object (possibly a composite object)
- A light
- A camera

The **Transform Selection** menu, in the upper part of the control panel, allows you to change the type of the current transformable. A one-line label just below the Transform Selection menu indicates what the current transformable is. The label type displayed depends on the current selection in the Transform Selection menu. For example:

top

The top-level object, typically consisting of several individual objects retrieved from disk storage with **Read Object**.

Directional *light_number* (*Transform Light active*)

Identifies the *light_number* on the lighting panel currently specified as being a directional light.

AVS-*view_number* (*Transform Camera active*)

Identifies the view window you've created during the current AVS session. (The first one is named *AVS-0*.) You can use the start-up file to define your own names for the windows that ConvexAVS creates. See "Start-up File" above. You can also specify view window names when you create a *.scene* file with the Geometry Viewer Script Language.

jet1.17 (*Transform Object active*)

An object retrieved from file *jet.obj*.

Mouse Usage - A mouse button has approximately the same function whether you are transforming an object, a camera, or a light. Since the exact functions vary, a full listing is contained in the following sections.

Transforming Objects

To transform objects, click the **Transform Object** button (or, equivalently, press function key **F1**). This sets the mouse buttons to perform the following functions:

Left Mouse

Selects a particular object, making it the current object. The selected object appears in the Current Object Indicator in the control panel. Repeated clicks on the same object move up the object hierarchy, progressively expanding the selection. For example, given the jet object:

One click: Selects wing, part of one wing of the jet object.

Two clicks: Expands the selection to jet, the entire jet object.

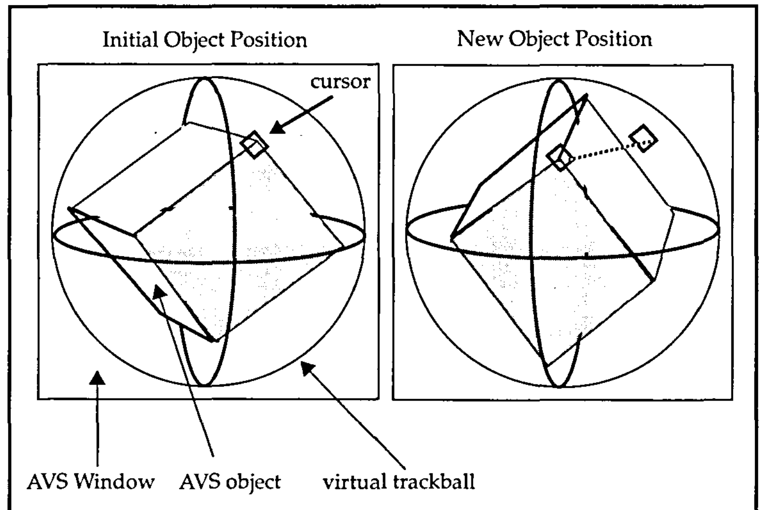
Three clicks: Expands the selection to the entire top-level object, which includes the jet and, perhaps, several other objects.

Four clicks: Having reached the top level, returns to wing, as selected with a single click.

Middle Mouse

A dragging action (holding down the middle mouse button and then moving the mouse) rotates the current object in 3D space. The object behaves as if it were attached to a trackball whose center is at the center of the view window. The mouse cursor is attached to the part of the trackball that protrudes above the surface of the window.

Figure 5-2
Rotating an Object with the Virtual Trackball



Right Mouse

A dragging action translates the selected object in the plane of the window (that is, moves the object left-right and/or up-down).

Right Mouse (with SHIFT button held down)

A dragging action translates the selected object perpendicular to the plane of the window (that is, moves the object in-out). Dragging upward or to the right moves the object away from your eye; dragging downward or to the left moves the object toward your eye.

Eventually, this translation may cause the object to cross the front or back clipping plane. This causes the object to partially disappear. (The location of the clipping planes in world coordinates is not currently controllable by the user.)

If the scene is not drawn in perspective (see "Cameras" section below), there will be no change in the object's appearance until it gets clipped.

Middle Mouse (with SHIFT button held down)

A dragging action scales the object: dragging downward or to the left makes the object smaller; dragging upward or to the right makes the object larger.

Transforming Lights

To transform lights, click the **Transform Light** button (or, equivalently, press function key **F2**). This sets the mouse buttons to perform the following functions:

Left Mouse

Still selects the current object. This button always selects objects, no matter what the Transform Selection is. To select the current light, click one of the boxes in the lighting panel (which appears when the Menu Selection is Lights).

Middle Mouse

A dragging action rotates the direction (directional light) of the current light using the trackball paradigm (see above).

Right Mouse

A dragging action translates the selected light in the plane of the window. With (bi-)directional lights, this changes the position of the symbol that represents the light source (**Show Lights**), but has no effect on the light source itself.

Middle Mouse (with SHIFT button held down)

Scales the **Show Lights** representation of the light source.

Transforming Cameras

To transform cameras, click the **Transform Camera** button (or, equivalently, press function key **F3**). This sets the mouse buttons to perform the following functions:

Left Mouse

Still selects the current object. This button always selects objects, no matter what the Transform Selection is. To select the current camera (view), just click in the desired window.

Middle Mouse

A dragging action rotates the position of the current camera (the camera in the current window) using the trackball paradigm (see above).

Note that the object seems to rotate, rather than the camera.

Right Mouse

A dragging action translates the camera in the plane of the window.

Right Mouse (with SHIFT button held down)

Translates the camera perpendicular to the plane of the window. If the scene is not drawn in perspective, this will have no effect. The **perspective** function must be enabled for this to work.

Middle Mouse (with SHIFT button held down)

Scales the 3D view volume for the camera. Only objects within this view volume appear in the window.

Recovering Transformations

Just below the **Current Object Indicator** are two additional buttons:

Reset

Restores the current transformable (object, light, or camera) to its default position. It does not also reset the color, surface properties, or rendering mode of the object. (You can also use the **F6** key).

Normalize

Scales the current object so that it fills its view window. (You can also use the **F7** key).

Function Key Usage

Most of the choices in the Transform Selection Areas can be made by pressing function keys instead of using the mouse. This can save you the overhead of moving the mouse cursor back and forth between the view window and the Transform Selection Area.

F1

Selects Transform Object, attaching the mouse to the (composite) object shown in the Current Object Indicator window.

F2

Selects Transform Light, attaching the mouse to the current light, as indicated on the lighting panel under the Lights menu selection.

F3

Selects Transform Camera, attaching the mouse to the camera in the current window. If you move to a different window, the mouse automatically switches to the camera in that window.

F5

Cycles the current object, as shown in the Current Object Indicator window. This is the same as clicking the mouse in the Current Object Indicator window.

F6

Performs a Reset, returning the current object, light, or camera to its original position and orientation.

F7

Performs a Normalize, resizing the current object so that it fills the current window.

Geometry Viewer Menu

The Menu Selection part of the Geometry Viewer control panel provides access to most functions for creating 3D scenes. Each scene includes a combination of objects, lights, and cameras. (Each camera you define shows the scene in a different window.) The Menu Selection area provides functions for moving data between disk storage and Geometry Viewer windows.

The following top-level menu choices are always visible in the Menu Selection area:

- Objects
- Lights
- Cameras
- Labels
- Action

One of these choices is selected at any particular moment. For instance, when you start the Geometry Viewer, Objects is selected automatically. The area below this top-level menu changes, depending on which choice is currently selected.

Objects

Selecting Objects causes the Menu Selection area to appear as follows.

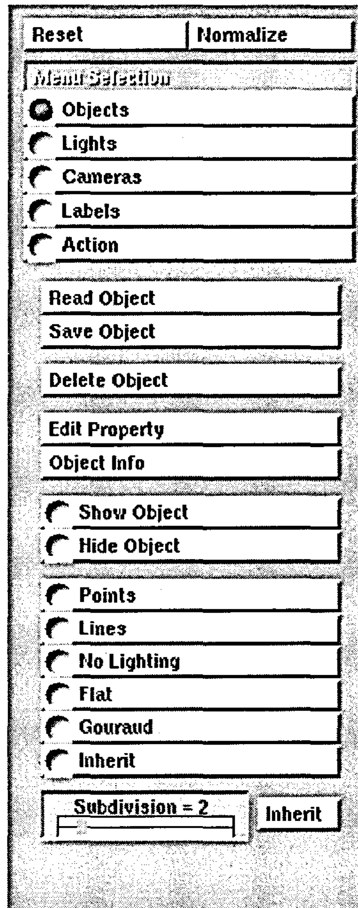


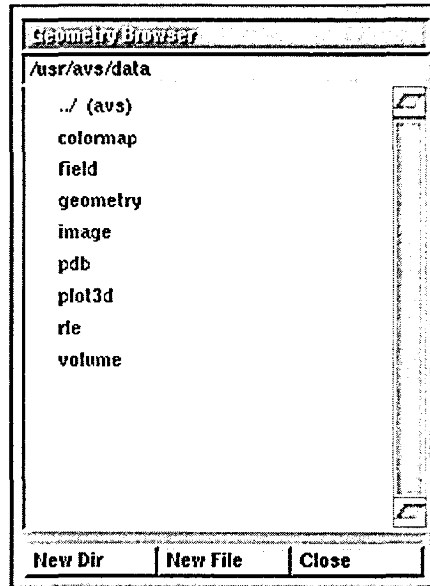
Figure 5-3
Objects Menu Selections

Read Object

This function allows you to retrieve one or more objects from disk files, placing them in the current window. As you select each object, it becomes the current object, as shown by the Current Object Indicator in the upper part of the control panel.

When you select **Read Object**, a small window (the File Browser) filled with file names from the current directory appears near the control panel.

Figure 5-4
The Geometry File Browser



The File Browser is sticky— it remains on-screen until you explicitly remove it by clicking on **Close**. This makes it convenient to retrieve multiple objects consecutively. You can also cancel Read Object by clicking on **Close** before you've read any objects at all.

The entries on the Geometry file browser are color-coded:

- **black** entries are files that contain Geometry Viewer objects;
- **red** entries are subdirectories - the topmost **red** entry is the parent directory

To select one of the entries, click on it with any mouse button.

Scroll Bars

Because a directory might contain a large number of entries, the File Browser has a scroll bar along its right edge. Clicking inside the scroll bar makes additional entries appear:

- The left mouse button scrolls upward.
- The middle button effect depends on where the cursor is:
  - In the top arrow box: Click to scroll the list to the very top.
  - In the elevator shaft: Click and hold down the button to grab the elevator bar. Moving the bar up or down causes the list

  to scroll accordingly.
  - In the arrow box at the bottom: Click to scroll the list to the very bottom.

- The right mouse button scrolls downward.

Selecting an object adds it to the current window. You can then use the mouse to move, rotate, and resize the object.

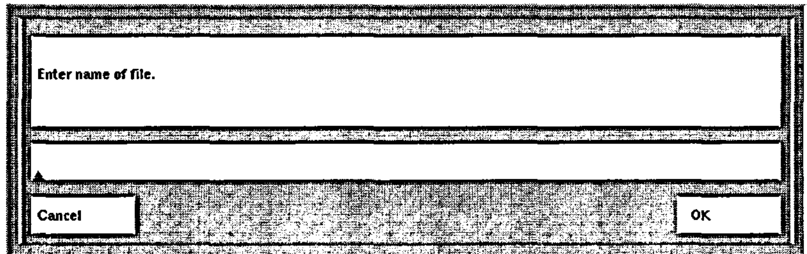
Selecting one of the red (directory) entries changes the working directory. The names of the Geometry Viewer object files in that directory are displayed, along with the names of any subdirectories.

You can also change the working directory by clicking on **New Dir** at the bottom of the File Browser. A window pops up so that you can type the name of another directory. (If you change your mind, click **Cancel** with the mouse.) Be sure the mouse cursor is in the one-line text-entry area before you start typing the directory name

Similarly, you can click the **New File** button to enter the full or partial path name of a file. Be sure to include the file name extension.

When typing a file name or directory name, you can use the **Backspace** key to erase the last character. Pressing Ctrl-U erases the entire line you've typed.

Figure 5-5
Entering a file name



You can type a full path name (starting with /) or a path name relative to the current directory; the name of the current directory is displayed above the text-entry area. For instance, to go two levels up the directory hierarchy, you would enter ../.. as the new directory.

To finish entering the new directory name, press the **RETURN** key or click **OK** with the mouse.

Note

File browser windows are also used in many places throughout the other ConvexAVS subsystems.

Save Object

This function saves the current object (shown by the Current Object Indicator) in a *.obj* file. This can be a composite object, consisting of two or more of the simple objects defined in *.geom* files. Any properties you have assigned with the **Edit Property** window are also saved, as are the rendering method(s) for the simple object(s). The *.obj* file contains references to one or more .geom files (which define simple objects). That is, the *.obj* file does not contain copies of geometries, but merely contains pointers to them. For this reason, be careful not to disturb *.geom* files that store the building blocks for your objects.

A window pops up so that you can type a file name. Be sure the mouse cursor is in the one-line text-entry area before you start typing the file name.

Note

You don't need to type the .obj extension when you enter a file name— ConvexAVS adds this extension automatically (unless you type it yourself). To finish entering the file name, press the Return key or click OK with the mouse.

Click on **Cancel** to cancel the save operation.

After you've saved an object, its file name appears in the File Browser. You can later bring the object back into the same window, or a different one, using **Read Object.**

ConvexAVS actually uses a special Script Language to create the object file (see Appendix B, "The Geometry Viewer Script Language").

Delete Object

This function removes the current object (shown by the Current Object Indicator) from the current window. It also removes the object from all other windows that show the same scene.

Note

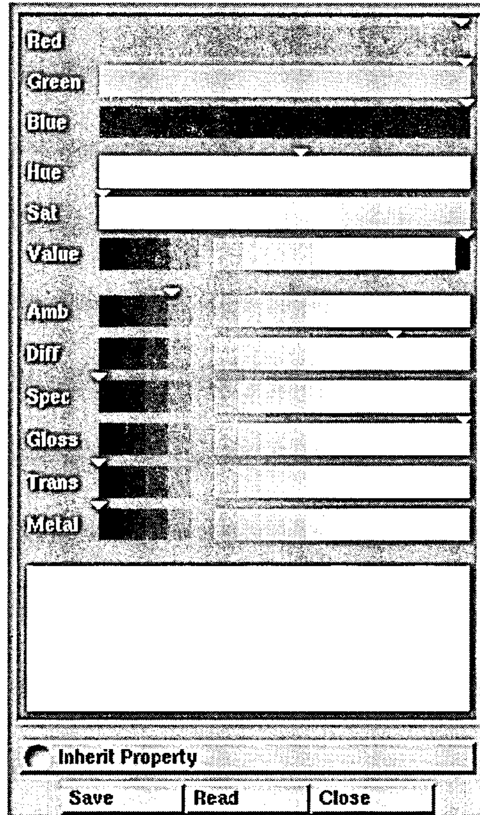
Since there is no way to undo deleting an object, you may want to perform a *Save Object* before deleting something that might be useful later on.

Edit Property

This function allows you to change the reflectance properties of the current object. The way in which an object in the real world reflects light depends on the characteristics of its surface: color, material (e.g., plastic, metal, fabric), smoothness, etc. From an intuitive point of view, then, this function allows you to specify the material from which the object is constructed.

When you select Edit Property, a window appears next to the control panel. The Edit Property window contains sliders that control the various surface properties:

Figure 5-6
The Edit Property Window



When the window first appears, the sliders show the current settings for the current object. As you move the sliders, the image of the object changes as soon as you release the mouse button.

Note

You can move a slider with any mouse button. You can either drag a slider by holding down the mouse button, or just click once at the spot where you want the slider to move.

Like the File Browser, the Edit Property window is sticky —it remains on-screen until you explicitly remove it by clicking on Close. This makes it convenient to change several properties of an object, or to change the properties of several different objects.

The sliders in the Edit Property window are as follows:

RGB Color

The top three sliders control the object's color by adjusting the amount of red, green, and blue. To make an object white, move all three sliders all the way to the right. To make an object black, move all three sliders all the way to the left.

HSV Color

The next three sliders provide an alternative way to specify the object's color: hue-saturation-value. White is specified by a zero saturation (slider all the way to the left). Black is specified by a zero value.

The RGB slider set and the HSV slider set provide two ways of controlling the object's surface color. Whenever you make an RGB change, the HSV sliders automatically adjust to reflect the change, and vice-versa.

Ambient Light Reflectance (Amb)

The proportion of the available ambient light that the object reflects. Ambient light is non-directional, affecting all parts of all surfaces equally.

This setting determines how much ambient light the object reflects. To control what ambient light there is in the scene, select the **AM** light on the lighting panel. This is described under the top-level menu choice **Lights**.

Diffuse Light Reflectance (Diff)

The proportion of the available non-ambient light that the object reflects equally in all directions. Non-ambient light emanates from directional light sources, which you specify with the lighting panel.

This setting is used in the calculations for Flat, and Gouraud shading.

Specular Highlight Intensity/Gloss/Metal

Specular highlights of a particular color and brightness are created when the direction of incoming light (from a directional light source) is sufficiently close to the viewing direction.

The Intensity (**Spec**) determines the brightness of such highlights. It corresponds to the specular coefficient in the lighting calculations.

The **Gloss** setting determines what sufficiently close means. The greater the sharpness, the smaller (more focused) is the size of the specular highlight. This setting corresponds to the specular exponent in the lighting calculations.

The **Metal** setting specifies the color of the specular highlight. ConvexAVS constrains the color to be somewhere between the color of the light source (left-most) and the color of the object (right-most).

Transparency

This setting controls the degree to which you can see through the front of an object, allowing you to see the back of the object and other objects behind it.

The Edit Property window also contains these buttons:

- The **Save** and **Read** buttons enable you to maintain a library of properties settings on disk. Each time you Save, ConvexAVS creates a file containing the current settings of all the sliders. It prompts you to enter a file name, and automatically adds the file name extension .prop to the name you enter. Be sure the mouse cursor is in the one-line text-entry area before you start typing the file name.
- The **Inherit Property** button replaces the current slider settings with those of the parent of the current object. The current settings are not lost, however. To restore them, just click on **Inherit** again. For longer-term storage of properties settings, use the **Save** and **Read** buttons.

Object Info

Clicking this button displays a window of information pertaining to the current object:

- Number of child objects
- Number of triangles in the object
- Number of lines in the object
- Number of triangle strips in the object
- Number of polylines in the object
- Number of disjoint lines in the object
- Number of spheres in the object
- Additional object data: vertex normals, vertex colors

```
Name: dodec.1
Children: 0
Total Triangles: 91
Total Lines: 60
Triangles Strips: 1
Poly Lines: 0
Disjoint Lines: 60
Spheres: 0
Mesh Vertices: 0
Labels: 0
Polygons: 0
Data: normals
```

Show Object/Hide Object

The functions cause the current object to disappear from the current window (**Hide**) or reappear there (**Show**). The object also disappears or reappears in all other windows that show the same scene.

A hidden object is still part of its scene. If you perform a **Save Scene** (described under Cameras), the hidden object is saved along with all the visible ones. You can later perform a **Read Scene** followed by a **Show Object** to bring the object back onscreen.

Rendering Methods

The following menu items change the rendering method used to draw the current object.

Points

The vertices are displayed as dots.

Lines

The object is drawn as a wire-frame, using non-anti-aliased lines.

No Lighting

The object is drawn using filled polygons, using no lighting or shading at all. The only color (or colors) used is the color of the object itself.

Flat Shading

The object is drawn using filled polygons. Each is flat shaded.

Gouraud Shading

The object is drawn using filled polygons, each of which is Gouraud shaded.

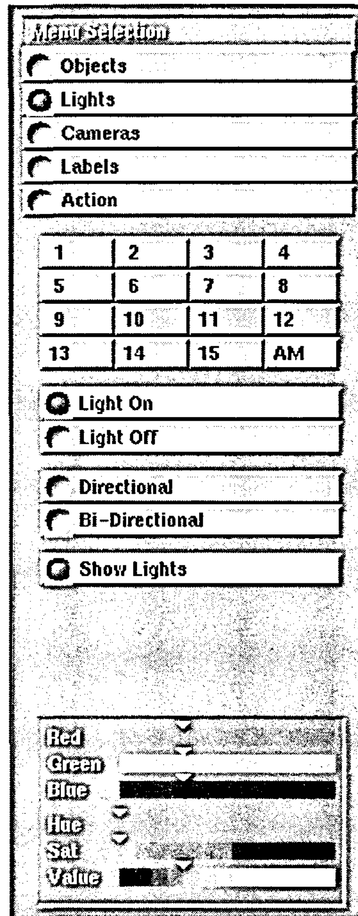
Inherit

Causes the current object to inherit the rendering mode of its parent object. Clicking again restores the previous explicit setting of the rendering mode.

Lights

Selecting Lights causes the Menu Selection area to appear as follows.

Figure 5-7
Lights Menu Selections



The grid of numbers is a lighting panel. In any scene, you can define up to 15 directional lights. In addition, you can specify the ambient light, indicated by **AM** on the lighting panel.

The original window of every scene is created with the following initial lights:

- Ambient light, with white color.
- Directional light #1, with white color. The direction of the light is parallel to your line of sight, as if a white sun were directly behind you.

To create an additional light, click the number on the lighting panel. Then, click **Light On** to turn on the light.

At any particular moment, one light in the scene is the current light. The number of the current light is always highlighted on the lighting panel. All the lights that are currently on are indicated by green numbers on the lighting panel.

Light On/Light Off

Turns the current light on or off.

Directional/Bi-Directional

Selects the type of the current light:

- **Directional** (default light type): A light source whose rays all point in the same direction (are parallel). The sun is the canonical directional light source.
- **Bi-Directional:** A pair of directional light sources that point in exactly opposite directions. This type of light can be used to correct the lighting of an object whose faces have been carelessly defined, so that the normals of some faces point outward and the normals of other faces point inward.

Note

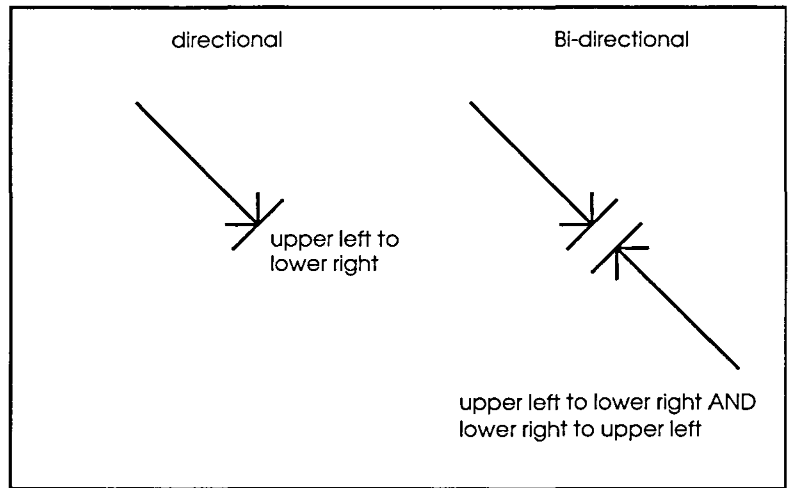
A bi-directional light is actually implemented as two lights— it occupies two positions, *n* and *n+8*, in the lighting panel. For example, if you make light #5 bi-directional, then light #13 is used as the second light. Accordingly, only lights #1 to #7 can be specified as bi-directional. If you attempt to select one of these complementary lights, an error message is displayed.

Show Lights

Displays a symbol for each light source, indicating its position and direction. The size of the symbol indicates the light source's orientation vis-a-vis the view plane (the plane of the display screen). The symbol's color is the same as the light source color, and it is depth-cued to help indicate its distance from the view plane.

Since the initial direction for the light is parallel to your line of sight, the initial display symbol is a cross-hair in the middle of your current object. Use the mouse to drag the directional light symbol.

Figure 5-8
Symbols for Light Types



To move (the direction of) a light, make sure that **Lights** is selected in both the Menu Selection and Transform Selection parts of the control panel. Then use the mouse to rotate and/or translate the light. See the Transform Selection section for details.

Color of Light

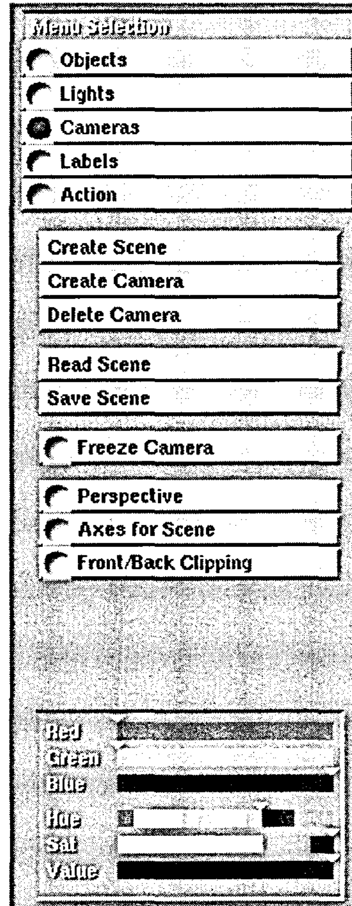
At the bottom of the Lights menu selection area, there are RGB and HSV sliders for setting the color of the light source. See the "Edit Property" section for an explanation of how to use the sliders.

The default color of all lights is white.

Cameras

Selecting **Cameras** causes the Menu Selection area to appear as displayed in Figure 5-9:

Figure 5-9
Cameras Menu Selections



These selections allow you to create additional windows to display the current collection of objects (that is, different cameras for the current scene). You can also create entirely new scenes, with different sets of objects.

Create Scene

Creates a new, empty window. The new window becomes the current window, as indicated by the bright red border.

Create Camera

Creates a new window that contains the same object(s) as the current window. This is not a new scene, but an additional window on the same scene. Each such window can have its own camera position, and its own settings for the camera parameters.

When you make a change in one window, all the windows on the same scene are affected simultaneously. This includes rotating or moving an object, changing an object's surface properties, changing the color or position of a light, and so on.

See **Freeze Camera** for a way to suppress this synchronization of windows on the same scene.

Note

In general, the new window is a different size from the original, so the images of the objects are scaled appropriately. The new window becomes the current window, as indicated by the bright red border.

Delete Camera

Deletes the current window. If you delete the last camera of a particular scene, then the scene itself is deleted, too.

You cannot delete the last camera if the display is generated from a module in a network that is still active.

Read Scene/Save Scene

These functions allow you to maintain a disk library of scenes. Each scene consists of one or more windows. Selecting **Save Scene** stores the current state of all of the scene's windows in a file. ConvexAVS prompts you to enter a file name, and automatically adds the file name extension *.scene* to the name you enter. Be sure the mouse cursor is in the one-line text-entry area before you start typing the file name.

Selecting **Read Scene** brings back all of the scene's windows to the screen.

See also the Geometry Viewer Script Language appendix. This language allows you to define scenes in ASCII files.

Note

Similarly to .obj files, .scene files contain references to objects and geometries, rather than copies of them. For this reason, be careful not to disturb .geom and .obj files that store the building blocks for your objects.

Freeze Camera

Use this function when you have several windows on a scene. When you freeze one of the windows, you can still manipulate the objects, lights, and camera in any of the other windows. The changes are reflected only in the unfrozen window(s) -- the image in the frozen window remains the same.

Perspective

This setting causes the current window to use a perspective viewing projection (the default is to use a parallel projection). The difference becomes most apparent when you scale the view volume.

Select **Transform Camera**. Then use the middle mouse button together with the **Shift** key to change the size of the view volume. For more on transformations, see the section "Transformations and the Transform Selection Area."

Axes for Scene

This choice toggles display of X-Y-Z axes in the current scene. The right-hand coordinate system indicated by these axes is the world coordinate system for the scene.

Front/Back Clipping

This choice toggles the use of front and back clipping planes. When clipping is enabled, objects disappear as they move either very close to the eyepoint or very far away. When clipping is disabled, the front and back clipping planes still exist. They are so distant, however, that for all practical purposes, no front/back clipping takes place.

Color of Window Background

At the bottom of the Cameras menu selection area, there are RGB and HSV sliders for setting the background color of the current window. See the "Edit Property" section for an explanation of how to use the sliders. The default background color for all windows is black. You can set the background colors (as well as the location and size) for a sequence of windows by specifying a defaults file with the following command-line option:

```
avs -geometry -defaults filename
```

Labels

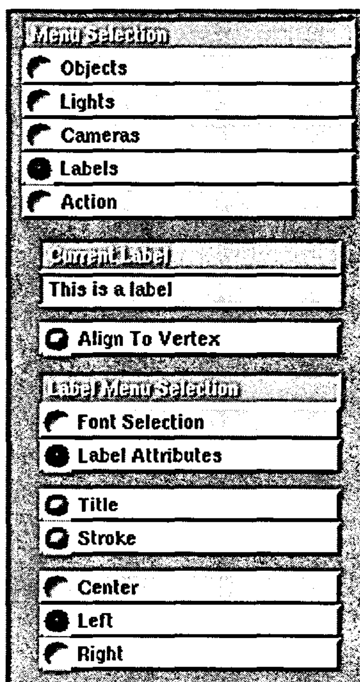
The Labels menu selection provides access to the Geometry Viewer's annotation text facility. You can attach one or more labels to any object. Each label consists of a single line of text. As you manipulate the object— move it, resize it, temporarily hide it, permanently delete it, etc.— the object's label(s) react accordingly.

You have considerable typographic control, with a wide range of fonts, type styles, sizes, and colors to choose from. You can also control the position of each label relative to its associated object; one alternative is to have the label become a title, which always appears at the same location in the window, no matter how the object is transformed.

Creating Labels

To create a label, first make sure the object to be labeled is the current object. If necessary, click on the object with the left mouse button. Then, click the **Labels** menu selection to bring up the display shown in Figure 5-10.

Figure 5-10
Labels Menu Selections



Place the cursor in the empty box below Current Label, and type any string of printable characters. Use **Backspace** (erase last character) and Ctrl-U (erase entire line) to make corrections.

Be sure to press **Return** when you've finished the label. When you do so, the label appears centered on the current object, surrounded by brackets.

Note

In some cases, part or all of the label may be obscured by the object itself. The brackets, however, will always be visible.

To create additional labels for the same object, select the object again by clicking on it with the left mouse button. This clears the Current Label box. (In addition, you may want to check that the Current Object Indicator shows the object and its name.) As before, type in a text string and press **Return**.

Labeling the Top-Level Object

Labels you create for the top-level object apply to the entire scene— they will appear in every window you create for the scene using Create Camera. The "Transformations and the Transform Selection Area" section describes the ways in which you can select the top-level object.

Picking and Moving a Label

Each of an object's labels is attached to a particular point in the object's coordinate system. Initially, this base point is the center of the object (that is, the origin of the coordinate system). You can move an existing label so that its base point is at a different X-Y-Z location:

- **Moving within the X-Y plane:** Click and hold down the left mouse button on the label. The brackets reappear to confirm that the label has been picked. Drag the cursor to any other location, then release the button. This moves the base point parallel to the plane of the display screen.
- **Moving in the Z direction:** Hold down the SHIFT key, then use the left mouse button as described above. This moves the label perpendicular to the plane of the display screen. Note that this does not change the size of the label (but see Changing Label Attributes below).

The label's new location is still defined in terms of the object's coordinate system— you have simply changed the coordinates of the base point. As you move, resize, or rotate the object, it remains attached to its base point, and so moves around the display window.

Attaching a Label to a Vertex

If you want to attach a label to one of the object's vertices, you need not worry about separate movements in the X-Y plane and the Z direction. Just click the **Align to Vertex** selection, then drag the label using the left mouse button. Before you release the mouse button, make sure the cursor is on (or very near) a vertex. This causes the vertex to become the label's new X-Y-Z base point.

Making a Label Into a Title

It is sometimes desirable to have one or more labels that are associated with an object, but which don't move around the screen as the object is transformed. Such labels are called titles. For instance, you might want a title string for an object to appear in the upper left corner of the window whenever the object is displayed. You can change any regular label into a title label by clicking the **Title** selection.

A title label lives in the window's X-Y coordinate system, rather than the object's X-Y-Z system. You can change the position of a title label using the left mouse button.

Editing a Label

To change the text of a label, first click on the label with the left mouse button to make it appear in the **Current Label** box. Then move the cursor into the box and type the changes. As when you first create a title, **Backspace** erases the last character and Ctrl-U erases the entire label.

Label Menu Selection

The annotation text facility includes a two-level function menu, which allows you to customize the appearance of each label. The top-level choices, **Font Selection** and **Label Attributes**, are always visible. The submenu of items for whichever of these choices is currently selected appears below them.

Font Selection Submenu

The Font Selection menu (Figure 5-11) lets you select the X Window System font to be used for the label.The submenu for Font Selection includes the following choices:

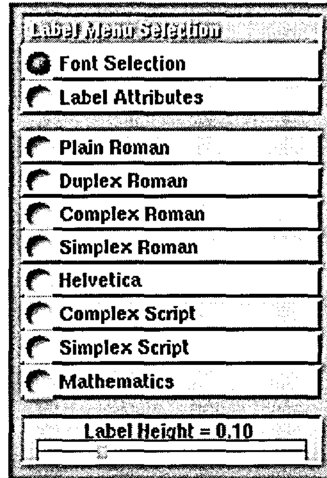
Plain Roman, Duplex Roman, Complex Roman, Triplex Roman, Helvetica, Complex Script, Simplex Script, Mathematics.

Label Height:

Selects the point size of the label. Labels do not scale continuously; instead, ConvexAVS makes best use of the available X

Window System fonts. As you move the slider to indicate a larger or smaller size (using any mouse button, by clicking or by dragging), the label size changes when a different font provides the closest fit.

Figure 5-11
Font Attributes Sub-menu



The brackets around the label does scale continuously to indicate the label's height at the requested size, whether or not a font of that size is available. The *xlsfonts(1)* utility program lists all the X Window System fonts available on your machine.

Label Attributes Submenu

The submenu for Label Attributes has the following choices:

Title:

Makes the current label into a title, whose position is defined in terms of window coordinates, rather than in relation to the object's 3D location. See "Making a Label Into a Title" above.

Stroke:

When set, causes labels to align through the Z axis. When not set, labels are aligned with the viewing plane.

Center, Left, Right:

Specifies which part of the label is placed on the base point. Initially, it is the bottom center. The alternatives are the lower left corner and the lower right corner.

Color Editor:

An RGB-HSV color editor, similar to ones used elsewhere by the Geometry Viewer, allows you to specify the color of the label.

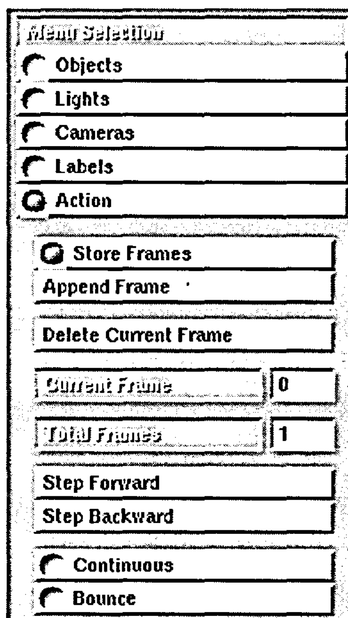
Action

The Action menu selection allows you to define animations, which take the form of a sequence of geometries (called cycles). You can append new frames to the end of the sequence, delete any frame within the sequence, and play back the sequence in a variety of ways. You can also define the sequence of geometries as a cycle in the Geometry Viewer Script Language. (See the Script Language appendix for details.)

A cycle is a group of frames consisting of different geometric descriptions. An object must be a *leaf object* to be a cycle. A leaf object is a sub-element (child) of a top-level object. You cannot use top-level objects to create frames.

When you select **Action**, the following submenu appears:

Figure 5-12
Action Menu Selections



Adding Frames

There are two modes for adding new frames to the end of the sequence:

Store Frames

If you turn on this toggle switch, every new geometry sent to the output window will be appended to the frame sequence. The Current Frame and Total Frames counters are updated automatically.

Caution

Main memory must be allocated for each frame. Make sure that your system has sufficient memory to accommodate all the frames.

Append Frame

This is a command, rather than a toggle switch. If Store Frames is turned off, you can click this button to add a frame to the sequence. The currently-displayed geometry is not added —rather, the next time a new geometry is sent to the output window, it will also be added to the sequence.

Limitations

The Add Frame feature has some restrictions.

Each time you read a new geometry, the existing animation sequence (if any) is discarded and a new sequence is begun. (Reason: when a new geometry is added to a sequence, it must have the same name as the object being animated; that is, it must be a modification of the current object.)

You cannot create an animation simply by clicking on a series of different names in the File Browser. You can create this kind of animation using the Script Language, however.

A frame contains a geometry definition only. A frame does not contain such attributes as the transformation, surface color, or material properties.

Lighting information is not captured in a frame. This means, for instance, that you can't create an animation that shows an object going through a rotation sequence. (The rotation is a transformation attribute, not part of the object's geometry.)

One way to generate a sequence is to create a network through the Network Editor that displays geometries and then invoke the Geometry Viewer and select the Store Frames function. Changes to any parameters within the network are recorded as new frames.

Playing Back the Frames

The following functions provide a variety of ways of viewing the frames in an animation sequence:

Step Forward

Displays the next object in the cycle. When the end of the cycle is reached, you automatically wrap around to the first object.

Step Backward

Displays the previous object in the cycle. When the beginning of the cycle is reached, you automatically wrap around to the last object.

Continuous

Continuously cycles forward through all the objects in the cycle. At the end of the cycle, the animation wraps around to the beginning automatically. To stop the animation, click Continuous again.

Bounce

Continuously cycles through the images, but alternates between going forward (beginning to end) and backward (end to beginning). To stop the animation, click Bounce again.

Deleting Frames

Clicking the **Delete Current Frame** button deletes one frame. (There is no way to delete a range of frames.) Typically, the current frame is the one most recently added to the sequence, but you can make any frame current. Use **Step Forward** or **Step Backward** to move to a particular frame. Alternatively, go to the **Current Frame** box, use **Backspace** or Ctrl-U to erase the number already there, type a new number, and press **Return**.

Network Editor Subsystem

The ConvexAVS Network Editor subsystem is your main tool for creating, testing, and revising ConvexAVS networks. You can also use the Network Editor to refine the user interface to a network, so that others can perform visualization tasks without having to be knowledgeable about network construction.

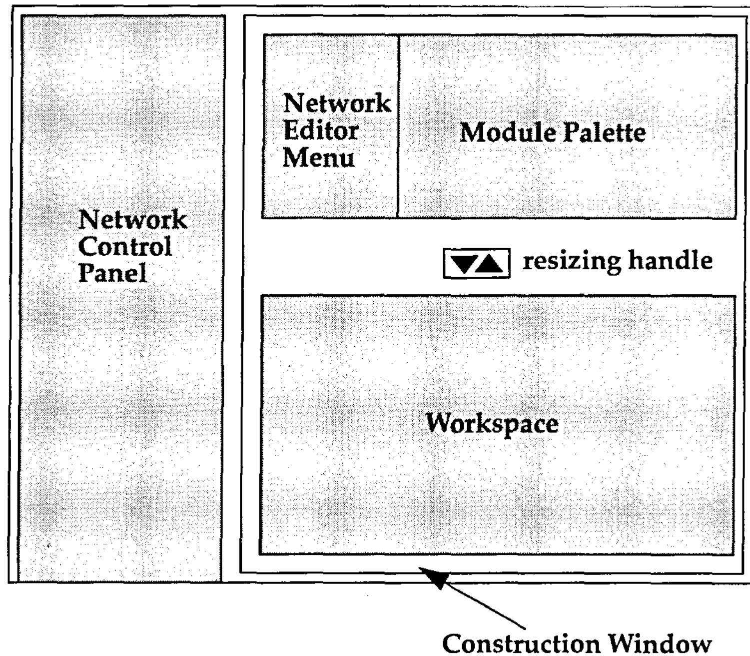
Starting the Network Editor

To start the Network Editor subsystem, click the Network Editor button on the ConvexAVS main menu. If you're not currently at the main menu, you can return there by clicking the **Exit** button at the top of the current subsystem's main control panel.

A Network Control Panel window appears along the left edge of the screen. Initially, this window is empty, since no network is currently active.

The remainder of the screen is used for the Network Construction window, which is divided into an upper part and a lower part. The upper part is shared by the Network Editor Menu and the Module Palette. The lower part is the Workspace in which you build networks (Figure 6-1):

Figure 6-1
Network Control Panel and Construction Window



Closing the Network Editor

The Network Editor subsystem has two top-level windows, which can be closed separately:

- Click the **Exit** button at the top of the Network Editor control panel window to close down the Network Editor and return to the ConvexAVS main menu. Any current work is lost, so be sure to save your work first.
- Click the **Close** button at the top of the Network Construction Window to close that window without destroying any work. This button is useful when you finish building a network and want more screen space for executing the network (for example, to manipulate the network's display windows).

  Clicking this button causes a **Display Network Editor** button to appear at the top of the Network Control Panel window. This allows you to reopen the Network Construction Window at a later time. The **Display Network Editor** button remains visible on the menu and can also be used to toggle the construction window on and off.

Switching to the Geometry Viewer

In many situations, you'll want to switch to the Geometry Viewer subsystem without losing your current Network Editor work. For example, if you create a network that displays a geometry, you may want to modify the rendering method or the lighting. There are two ways to go directly from the Network Editor to the Geometry Viewer:

- Click the **Geometry Viewer** button at the top of the Network Construction Window. You can return to the Network Editor by clicking the **Close** button in the Geometry Viewer control panel
- If a network includes the **render geometry** module, use the left mouse button to click the small square box in the icon for this module.

Overview of Network Editor Usage

In general, creating a network includes these steps:

1. Using the mouse to pull modules from the AVS Module Library Palette into the Workspace, and/or reading already-existing networks from disk storage.
2. Using the mouse to connect the modules' input and output ports. The connections define the network by specifying the flow of data among the modules.
3. Adjusting the modules' input parameters using the widgets in the Network Control Panel.

These steps are described more fully in the sections that follow.

Using the Module Palette and the Workspace

The Module Palette includes an icon for each of the ConvexAVS computational modules. The modules are partitioned into four functional categories:

Data Input Modules

These modules introduce new data into a ConvexAVS network. Some modules (e.g. **read volume**) read a data file from disk storage. Other modules (e.g **generate colormap**) create data according to the settings of their input parameters.

Filter Modules

These modules transform a numerical data set into another numerical data set. They perform such actions as sampling, creating subsets, establishing threshold values, applying a linear transformation, etc.

Mapper Modules

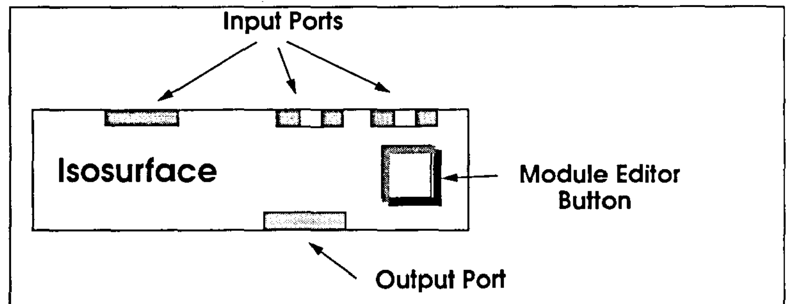
These modules perform the visualization step— converting a numerical data set to a description of one or more geometric objects. For instance, the field to mesh module creates a 2D surface in 3D space. It does so by interpreting each scalar value of a 2D array as the height of a point above a base plane. The collection of points defines (an approximation to) a 2D surface above the plane.

Renderer Modules

These modules produce the final output of the visualization process. In most cases, this is an on-screen image, displayed in its own window. Some modules store image data in image files for later display, or in PostScript files for printing.

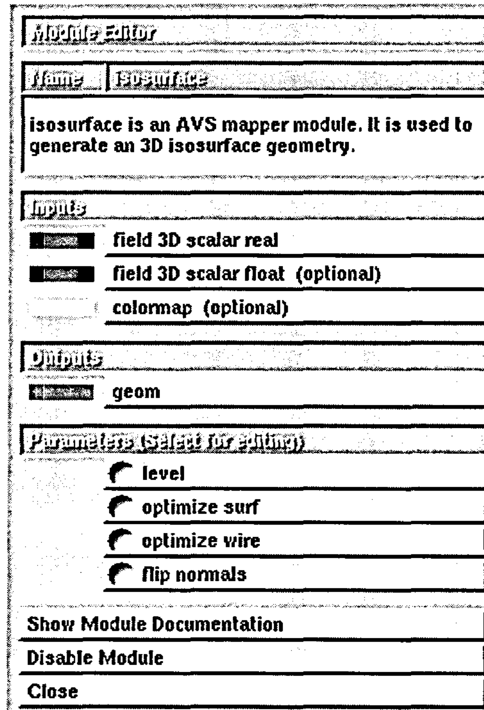
Each module icon shows the module's name, along with input ports and output ports to indicate the types of data that the module handles (Figure 6-2). The ports are color-coded to indicate the type of data that can pass through the port.

Figure 6-2
Module Icon



You need not memorize the color-coding scheme— ConvexAVS allows you to connect ports only if their data types are compati ble. You can also display the ports' data types by clicking the small square **Module Editor** button on the module icon (the dimple) with the middle or right mouse button. This pops up the Module Editor window, which displays helpful information about the module: a capsule description, the data type of each input and output port,and a list of the input parameters. If you need further information on the module, click the **Show Module Documentation** button in the Module Editor window to display the entire manual page for the module in a help browser window (Figure 6-3).

Figure 6-3
Module Editor Window



The next few sections describe how to work with module icons using the mouse. For quick reference, here's a listing of how the mouse buttons work in this context:

Left Mouse: Move one or more icons.

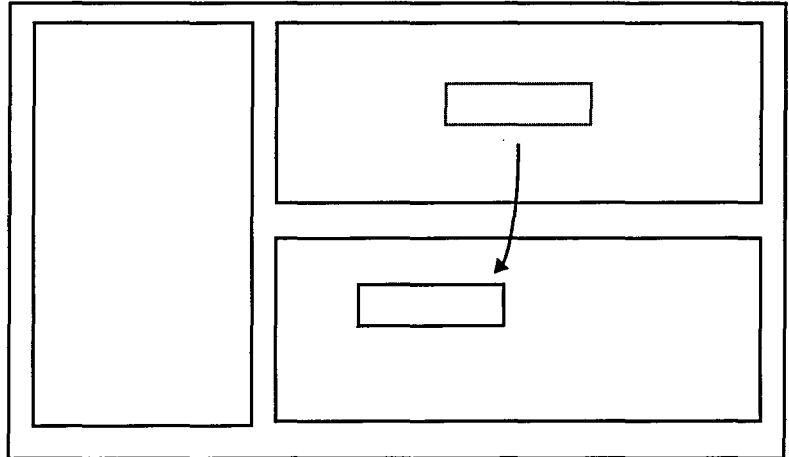
Middle Mouse: Establish a connection between two icons.

Right Mouse: Break an existing connection between two icons.

Moving Icons into the Workspace—Left Button

Use any mouse button to drag a module icon from the Palette to the Workspace. As you do so, the module's control panel—the set of widgets that control the input parameters— appears in the Network Control Panel window at the left side of the screen (Figure 6-4).

Figure 6-4
Dragging a Module From the Palette Into the Workspace



At the top of the Network Control Panel is a choice menu (radio button menu) labeled Top Level Stack, which lists all the modules currently in the Workspace. At any moment, one module's control panel is visible. You can click in the menu to bring any other module's control panel in view. (You can also bring up the control panel of any module in the Workspace by clicking the small square on the module icon with the left mouse button.)

Note

The *render geometry* module is an exception. When you drag it into the Workspace, its control panel does not appear, nor is its name added to the Network Control Panel menu. The control panel for render geometry is the entire Geometry Viewer menu system described in Chapter 5.

When you click the small square on the render geometry module icon with the left mouse button, the Geometry Viewer control panel appears, obscuring the Network Control Panel. To make it disappear, click the Close button at the top or use the left mouse button to click the small square of some other module icon in the Workspace. Another alternative is to move the Geometry Viewer control panel aside, using the X Window System window manager. This allows you to see both the Geometry Viewer/render geometry control panel and the Network Control Panel at the same time.

Moving Modules within the Workspace

Once a module icon is in the Workspace, you can move it around, using the left mouse button. You can also drag a rectangular lasso around several icons:

1. Click and hold down the left mouse button when it is not on a module icon. This places one corner of the lasso.
2. Drag the mouse to expand the lasso, fully enclosing one or more module icons.
3. Release the button to complete the lasso.
4. Press the left button again with the mouse cursor within the lasso area to drag the entire group to a different location in the Workspace.

(To remove a lasso, click the left mouse button in the background part of the lasso area.)

Deleting Modules from the Workspace

Use the left mouse button to drag a module icon onto the hammer icon in the lower right corner of the Workspace. You can also lasso several icons, drag the entire lasso area so that any part of it touches the hammer, then release the button.

Whenever you delete a module from the Workspace, any connections between its ports and those of other modules are automatically deleted, too. The deleted module's control panel disappears from the Network Control Panel.

Connecting Modules—Middle Button

The small colored bar(s) at the top edge of a module icon represent the module's input ports. (Data modules have no input port, since they introduce new data into a network, rather than process data that is already there.)

Similarly, the colored bar(s) at the bottom edge represent output ports. (Most renderer modules have no output port, since they don't pass any data to other modules. Instead, they either display an image on-screen or write data to a disk file.)

The ports are color-coded to represent the type of data that can pass through—an output port can only be connected to an input port with a matching color:

red = *geometry*

A geometry is a geometric description of one or more objects (a scene). It can be created by a module or other program using calls to the ConvexAVS *libgeom* library. Along with the definitions of the objects in terms of points, lines, triangles, spheres, etc., a geometry can include specifications for vertex and surface colors, lighting, rendering mode, transformations (translation, rotation, scaling), and transparency.

ConvexAVS includes conversion utilities that accept data in common formats and produce geometry files that can be read into a network with the read geometry module. See the Geometry Conversion Programs appendix for details.

ConvexAVS also includes modules that dynamically convert raw data into geometries (e.g. the field to mesh module).

yellow = *colormap*

A colormap is a table that converts an integer value to a pixel value (i.e. to a color). Typically, you use the generate colormap module to create colormaps dynamically. This module also allows you to maintain a set of on-disk colormaps that you can load during network execution.

light blue = *pixmap*

A pixmap is an image stored in memory allocated by the X server. When a geometric description (a geometry) is rendered to produce pixel values, the pixmap format is used to hold the resulting image. Thus, the output of the render geometry module is often sent to the display pixmap module (or another module that handles pixmap input).

multi-color = *field*

A field is a very flexible data type, more like a collection of related types. A field is a generalization of the array structure that is used to represent many kinds of scientific data. See the preceding chapter for a discussion of fields. For more information about data types, see the "Data Types" chapter.

Matching has a special meaning in the case of the multi-colored field ports. For more information, see "Connecting Field Ports" below.

To make a connection

1. Click and hold down the middle mouse button on one module's output port. ConvexAVS automatically displays thin lines that indicate all the valid connections to other modules' input ports.
2. Drag the mouse toward one of the valid destinations.
3. As soon as ConvexAVS highlights the connection you want to make (turns it white), release the button to complete the connection. It's not necessary to drag the mouse all the way to the destination.

If you release the mouse button before any of the possible paths is highlighted, no connection is made. You can also avoid making a connection by returning the mouse cursor to the original output port.

The same procedure works for making connections in the opposite direction—start on an input port and connect backward to another module's matching output port.

Connecting Field Ports

The ConvexAVS field data type is actually a general format—you can think of it as a collection of related sub-types. Fields can differ in their dimensions: 1D, 2D, 3D, etc. Fields can also differ in the type of data that is specified for each point: scalar byte, 4D vector of bytes, scalar float, etc.

The various field sub-types are incompatible. Amodule that outputs a 2D field cannot be connected to one that expects to input a 3D field; a module that outputs floating-point data cannot be connected to one that expects to input byte data.

ConvexAVS includes modules that can help you to smooth over field-level incompatibilities. For instance, the field to byte module accepts any field as input, and outputs a field whose data values are bytes. This may be necessary when you plan to use a module that accepts byte-valued fields only. Similarly, there are modules for handling dimension-based conversions. The orthogonal slicer takes a 2D slice from any 3D field. You can extract one dimension from a multi-dimension field using extract scalar; you can assemble a multi-dimension field from its component dimensions using combine scalar.

As an aid in matching field sub-types, the color bars for field input and output ports are divided into four parts (Figure 6-5 and Table 6-1):

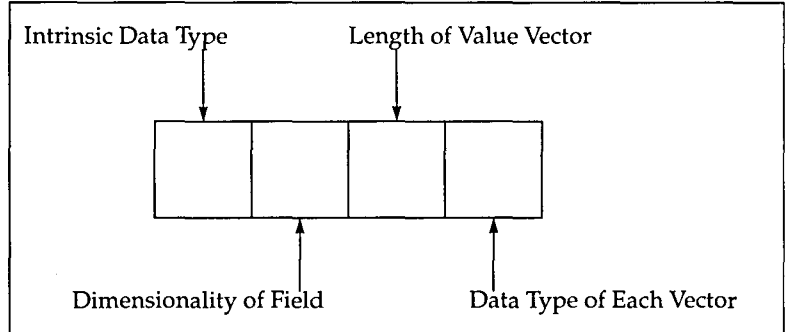


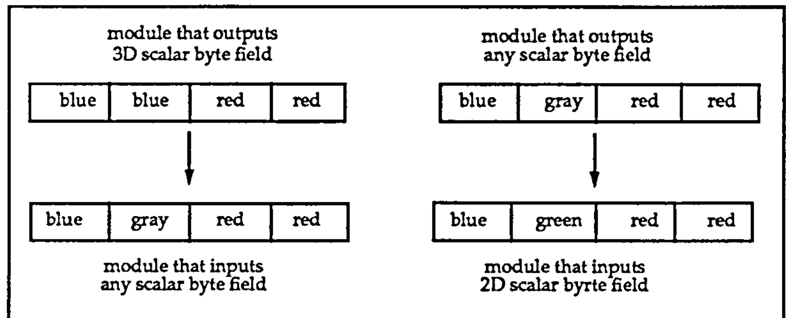
Figure 6-5
Color-Coding for Field Input/Output Ports

Table 6-1
Color-Coding for Field Input/Output Ports

	Color	Meaning
Intrinsic Data Type	blue	field
Dimensionality of Field	green blue red gray	1-dimensional 2-dimensional 3-dimensional any dimensionality
Length of Value Vector at Each Point in Field	red green blue gray	1 (i.e. value is a scalar quantity) 3 4 (or gray - wildcard) either of the above
Data Type of Each Value	red green blue yellow gray	byte integer single-precision floating point double-precision floating point any of the above

Note that the color gray is a wildcard: it indicates that the module is written to handle any of the supported alternatives. For example, if the second part of an input port color bar is gray, the module can accept fields of any dimensionality. This means that the color bars for field ports don't have to match exactly to be candidates for connection (Figure 6-6).

Figure 6-6
Gray Color-Coding as Wildcard Value



Caution

You must be careful when exploiting the flexibility illustrated in the above diagram. In fact, this diagram shows how you can connect three modules in such a way that a 3D scalar byte field is sent to a module that expects a 2D scalar byte field. Such a network will compute incorrect results.

More generally, it is important to keep in mind that the amount of type-checking performed by ConvexAVS, both when you construct a network and when you run data through a network, is quite limited. It is possible to construct networks that will pass ConvexAVS checks, but will perform incorrect or meaningless computations.

Disconnecting Modules— Right Button

The process of disconnecting modules is similar to connecting them, except that you use the right mouse button instead of the middle button. Click and hold down the right mouse button on a connected input (or output) port. Drag the mouse toward the other end of the connection, until the connection to be deleted is highlighted. Then release the button.

When you delete a module from the workspace (see above), all its connections are automatically deleted, too.

The Module Editor and Parameter Editor Windows

Each module icon has a small square **Module Editor** button at its right edge. You can click this button to bring up the Module Editor window (see Figure 6-3).

Note

When an icon is in the Palette, you can use any mouse button to bring up the Module Editor. When the icon is in the Workspace, only the middle and right mouse buttons perform this function. The left mouse button acquires a new function— raising the module's control panel to the top of the Network Control Panel stack.

This window provides a first level of documentation for the module: a one-sentence summary description, descriptions of the input and output ports, and a listing of the module's input parameters.

The Module Editor window also includes these function buttons:

Show Module Documentation

Displays the compete manual page for the module in a help browser window. This is the same page that you can view from the shell with the `man` command.

Disable Module

Temporarily disconnects the module from its network, preventing it from receiving or sending data. This often has the effect of freezing the entire network. The module icon turns red to indicate its disabled state.

To re-enable the module, click this button again.

The Parameter Editor

When you open the Module Editor window from the Workspace (not from the Palette), you can click on any of the input parameters to open its Parameter Editor window. This window allows you to change the control widget that is attached to the input parameter. For instance, you might want a parameter that currently is controlled by a dial to be attached to a type-in, instead. This would allow you to enter an exact value, such as 48.2, rather than using the mouse to fine-tune a dial setting.

Finding the Module You Want

The Network Editor's standard module library includes more than 60 modules. This number is large enough so that you may not immediately spot the module you're looking for at any given moment. And in some cases, there are too many modules in a particular category to fit in the vertical space allotted to the Palette. The following sections describe ConvexAVS facilities for handling such situations.

Scrolling a Module List

The icons in each category are listed alphabetically. If ConvexAVS cannot simultaneously display all the icons in the allotted space, it adds a scroll widget to the category's title bar:

Figure 6-7
Scroll Icon for a Module Category

One or both of the arrows are lit at any moment, indicating which way(s) the list can be scrolled. Clicking the left mouse button in the title bar scrolls toward the top of the list; clicking the right mouse button scrolls toward the bottom.

Incremental Search Through a Module Category

Each module category is organized alphabetically by module name. At any time (even when all the module icons in a category are visible), you can perform an incremental search through the names:

1. Put the cursor in the title bar of the category to be searched.
2. Type any character in the module's name. The list automatically scrolls so that the first icon containing that letter is at the top of the list.
3. Now, there are two ways to continue searching:
   - Type the next letter in the module's name. The next icon containing the pair of letters scrolls to the top. For instance, to search for the **colorize** module, you might type **c** followed by o, or you might type i followed by z.
   - Press **Return** to continue the search on the current basis—that is, search for the next icon containing the letter you typed.
4. You can repeat the preceding step as many times as you like, either adding characters to the search string, or pressing **Return** to continue the search for the same string.

There is no explicit way to end the search. Whenever you're finished searching, just stop typing. Similarly, nothing special happens if a search string fails to match any module— the list simply doesn't scroll.

Note

Any time the cursor is in the title bar of a category, you can press BACKSPACE to scroll the category back to the top.

Changing the Partitioning of the Construction Window

In some cases, you may find it desirable to change the vertical space allotment for the Module Palette. Increasing it can reduce the need for scrolling the module lists. There is a handle marked with scroll arrows at the top of the Workspace area (see Figure 6-1). When you place the cursor on this handle, a message appears alongside it, explaining how to use it: grab the handle with any mouse button and move it downward or upward. That is, click and hold down the mouse button, drag the mouse, and then release the button. This changes the partitioning of the Network Construction window between the upper area (Palette/Menu) and the lower area (Workspace).

Module Libraries

When it begins execution, the Network Editor locates a set of executable modules, looking in directory */usr/avs/avs_library* and in any directories specified with **-modules** command-line options. It does not, however, automatically install all the modules it locates in the Module Palette. Instead, the Network Editor installs only those modules specified in the default module library, which is defined in file */usr/avs/avs_library/SupportedModules*

You may find that this collection of modules is not exactly what you'd like the Palette to contain:

- There may be modules that you never use, and would like to remove.
- You may want to add some modules that you've written.
- You may want to organize the modules into subsets (perhaps overlapping) that contain modules for a specific type of visualization, a particular data type, etc.

To meet these needs, you can define you own module libraries. The **Read Module Library** function reads a module library file and replaces the current Palette contents with the new library. If you've already read in several libraries, you can switch back and forth among them instantly using the Select Module Library function.

Module library files are ASCII text files, the format of which is described in the File Formats appendix.

Completing a Network

A network is complete when it includes one or more modules that generate data, and one or more modules that display an image (or store data on disk). It can also include any number of modules that perform intermediate processing on the data.

You are now ready to control the execution of the network using the Network Control Panel window.

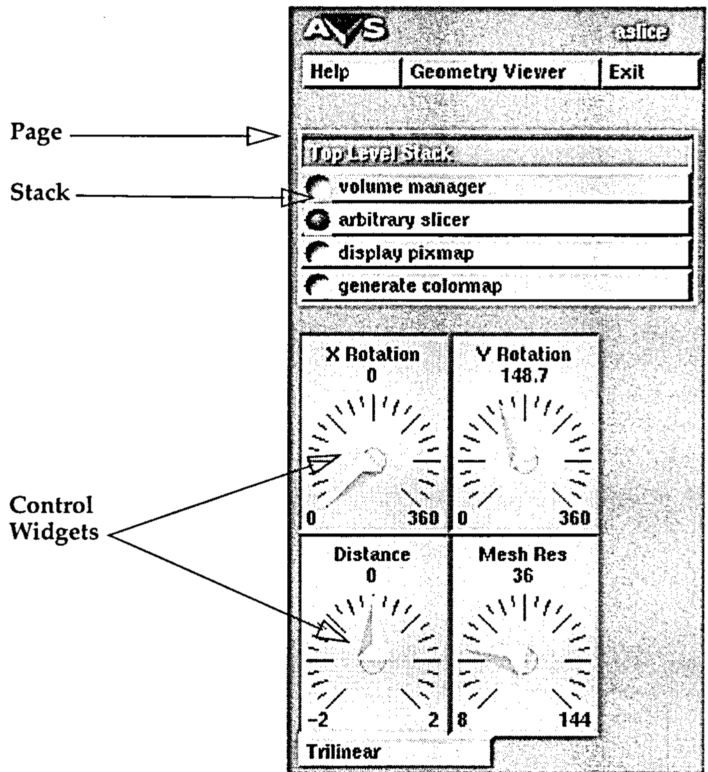
Controlling the Execution of a Network

As you build a network, its modules start to execute. In most cases, nothing useful will occur until the network is complete and you specify the input data to be visualized. Thereafter, the network re-executes each time th efollowing actions take place:.

- You specify a different data set (or the data entering the network changes in some other way)
- You change the setting of a module input parameter. (There is also a **Disable Flow Executive** function that suspends network execution, allowing you to adjust several parameters before having the network run again.)

You make these changes to the network's execution environment using the Network Control Panel window at the left edge of the screen. The Network Control Panel is organized as follows (Figure 6-8):

Figure 6-8
Organization of the Network Control Panel



- Individual *control widgets* (sometimes simply called controls or widgets) correspond to the input parameters of the modules in the network.
- Each module's controls are assembled onto a *page*. Each module has its own page, whose size depends on the number of input parameters and the control widgets attached to them.
- All of the pages are gathered into the Network Control Panel window, which has the form of a *stack*: only one page at a time is visible; you can switch among the pages by clicking in the choice menu at the top of the window. (This menu is automatically created as you add pages to the stack.)

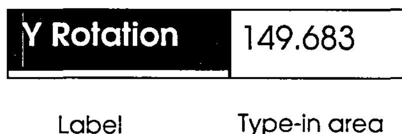
Using Control Widgets

ConvexAVS has a variety of control widgets that allow you to specify module input parameters: integers, floating point numbers, text strings, file names, mutually-exclusive choices, non-mutually-exclusive choices, and colormaps. The following sections describe how to use the various types of control widgets.

Using Type-In Controls

Figure 6-9 shows a typical type-in control widget.

Figure 6-9
Type-In Control Widget



To use a type-in, move the mouse cursor into the type-in area, so that it lights up. Then, type any printable characters and press **Return**. The existing value, if any, is not replaced— you must explicitly erase it if you want to enter a completely new value. There are two erasure keys:

Backspace: Erases the last character currently in the type-in area.

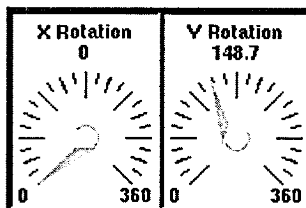
Ctrl-U: Completely erases the type-in area.

When you press **Return** to finish entering the new value, ConvexAVS checks it against the parameter's bounds and type. In some cases, the value you type is converted (e.g. decimal value converted to integer, out-of-bounds value converted to allowable maximum), and the result of the conversion is displayed.

Using Dial Controls

Figure 6-10 shows two typical dial control widgets.

Figure 6-10
Dial Control Widgets



You can use a dial either by clicking or by dragging:

- If you click with any mouse button at a location along the edge

of the dial, the needle jumps immediately to that location and the current value indicator changes accordingly.

- Alternatively, click and hold down any mouse button near the needle; then use a circular motion to drag the needle either clockwise or counter-clockwise. As you do so, the current value indicator changes. You can drag the needle any amount, from just a few degrees to many complete revolutions.

  If a dial's associated parameter has limits, attempting to drag the needle to a value outside the parameter's min-max bounds will fail— the needle stops moving when you reach the limit.

The Dial Editor

Dial control widgets are special in that they have an associated control window called the Dial Editor (Figure 6-11). To pop up the Dial Editor, move the cursor to the center of the dial, causing it to be highlighted. Then, click with any mouse button.

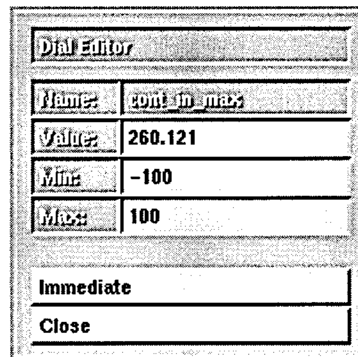


Figure 6-11
Dial Editor

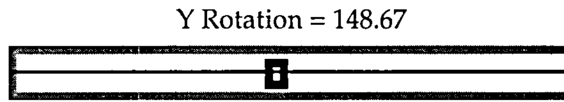
You can use this window to specify an exact value for the parameter (which may be easier than trying to move the dial needle by microscopic amounts). You can also change the dial's minimum and maximum bounds.

The **Immediate** button is a toggle switch. If you turn this feature on, the dial continuously sends values when you drag the dial needle to a new position. This causes the image that depends on the parameter to change continuously, also. (This may not be advisable for compute-intensive networks— changes to the image may lag behind the movement of the needle.)

Using Slider Controls

Figure 6-12 depicts a typical slider control widget.

Figure 6-12
Slider Control Widget



As with a dial, you can use a slider either by clicking or by dragging:

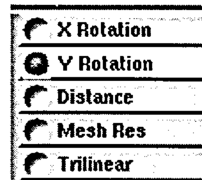
- If you click with any mouse button at a location along the slider, the cross-hair jumps immediately to that location and the current value indicator changes accordingly.
- Alternatively, click and hold down any mouse button near the cross-hair; then drag it to the left or right. As you do so, the current value indicator changes.

The parameter's minimum and maximum values are displayed only while you are adjusting the slider. Similarly, the slider's name disappears while you are adjusting it.

Using a Set of Choices (Radio Buttons)

Figure 6-13 shows a typical choice (radio buttons) control widget, which allows you to select from a mutually-exclusive set of choices.

Figure 6-13
Set of Radio Buttons



A red ball and highlighting indicates which one of the choices is currently selected. To select another choice, move the cursor anywhere within its box. (The label inside the box lights up to indicate the cursor's presence.) Then click any mouse button to move the red ball to the new selection.

Using Toggle Controls

Figure 6-14 shows a toggle control widget, in both the off state and the on state:

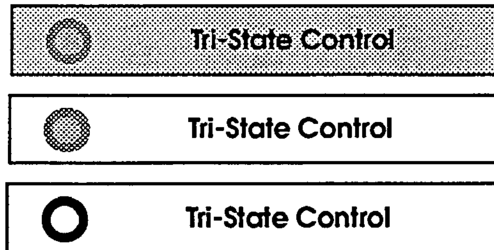
Figure 6-14
Toggle Control Widget

To change the state of a toggle switch, move the cursor anywhere within its box. (The label inside the box lights up to indicate the cursor's presence.) Then click any mouse button.

Using Tri-state Controls

Figure 6-15 shows a typical tri-state control widget, in its three states. This type of control is used for parameters that can assume three values, not just two.

Figure 6-15
Tri-state Control Widgets



To change the state, move the cursor anywhere within its box. (The label inside the box lights up to indicate the cursor's presence.) Then click any mouse button. Successive clicks cycle the widget through its three states.

Using One-shot Controls

Figure 6-16 shows a typical one-shot control widget. This type of control is used to invoke a command, rather than to change the state of a parameter.

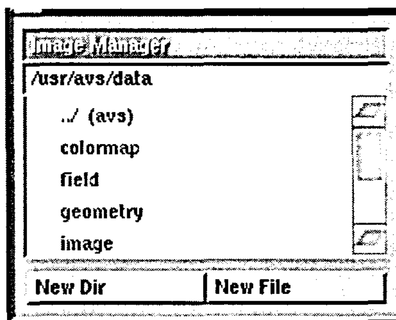
Figure 6-16
One-shot Control Widget

To use a one-shot control, move the cursor anywhere within its box. (The label inside the box lights up to indicate the cursor's presence.) Then click any mouse button to make the box flash.

Using File Browser Controls

Figure 6-17 shows a typical file browser control widget.

Figure 6-17
File Browser Control Widget



The entries in a file browser are color-coded: black entries are files; red entries are subdirectories (the topmost red entry is usually the parent directory). To select one of the entries, click on it with any mouse button. Selecting a directory entry changes the working directory, causing file names in that directory to be displayed, along with the names of any subdirectories.

Since a directory might contain a large number of entries, a file browser has a scroll bar along its right edge. Clicking inside the scroll bar makes additional entries appear:

- The left mouse button scrolls upward (or leftward).
- The effect of the middle button depends on the cursor position:
  - In the arrow box at the top: click to scroll the list to the top.
  - In the elevator shaft: click and hold down the button to grab the elevator bar. Moving the bar up or down causes the list to scroll accordingly.
  - In the arrow box at the bottom: click to scroll the list to the very bottom.
- The right mouse button scrolls downward (or rightward).

A file browser has these buttons at the bottom:

New Dir

Pops up a dialog box in which you can type the name of another directory (full path name or path relative to the current directory). Be sure the mouse cursor is within the dialog box (but not on the **OK** or **Cancel** button) before you start typing the directory name. When you click the **OK** button in the dialog box (or press the **Return** key), the directory whose name you've typed becomes current, and its file names are displayed in the browser window.

Use **Backspace** to erase the last character or **Ctrl-U** to erase the entire name. If you change your mind altogether, click the **Cancel** button.

New File

Pops up a dialog box that works the same way as the **New Dir** box. This allows you to specify the file to be processed, either with a full path name or a name relative to the current directory. Note that the current directory does not change, no matter what name you enter.

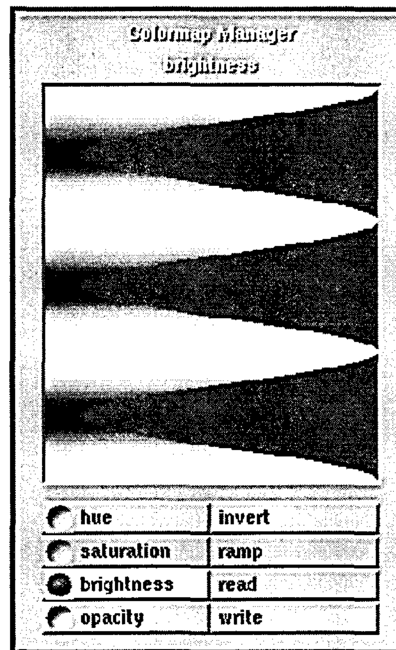
Close

(not always present) Some file browsers are sticky— they pop up in a separate window and remain on-screen until you explicitly remove them by clicking this button.

Using the Colormap Control

Figure 6-18 shows the control widget that generates a colormap. You can also use it to maintain a set of colormaps in disk files.

Figure 6-18
Colormap Control Widget



A ConvexAVS colormap is used by a number of modules to translate integers in the range 0..255 to pixel values (i.e. colors). A colormap is essentially a 256-line table, each line of which includes four fields: hue, saturation, brightness, and an auxiliary field. The colormap generator control widget allows you to create a colormap table visually.

The widget has four pages, one each for hue, saturation, value, and opacity (the auxiliary field). You switch among the pages by clicking the radio buttons in the bottom left part of the control widget.

Each page has the appearance of an area graph. It is actually a set of 256 very thin horizontal bars. On the hue page, the length of the top bar specifies the hue number for line 0 of the table. The color of this bar indicates the hue to which a data value of 0 will be mapped. The next bar specifies both the hue number for line 1 and the hue to which data values of 1 will be mapped; and so on.

Initially, the hue numbers form a linear ramp. Smaller numbers will be mapped into the blue part of the spectrum; larger numbers will be mapped into the red part. To change the set of hue numbers:

- Place the cursor near the top (but within) the square containing the 256 horizontal bars.
- Press any mouse button and drag the cursor downward along the new path. The lengths and colors of the horizontal bars change as you drag downward. The new values are reported at the top of the control widget as you drag.

The colormap generator widget also includes the following buttons:

Invert

Reverses the mapping of the range 0..255 to color values. The 0th color becomes the 255th color, the 1st color becomes the 254th color, etc. Visually, this flips the colormap over a horizontal axis.

Ramp

Restores the default colormap, which is a linear ramp starting in the blue range and ending in the red range.

Read, Write

These functions implement a system for maintaining a set of colormaps on disk. Clicking either of these buttons brings up a file browser that allows you to specify a file in which to store the current colormap (Write), or from which to reinstate a previously-stored colormap (Read).

Organizing a Network's Display Windows

In general, a ConvexAVS network produces one or more pictures as its output. (In this section, we use the word picture to refer either to an image, produced by converting data directly into pixels, or to a pixmap, produced by converting data to a geometry which is then rendered.) Each picture is displayed in its own display window (output window), although some pictures may combine data from several data sets. This section describes the way in which ConvexAVS creates display windows, and the ways in which you can manipulate these windows.

Whenever you drag a module icon from the Palette to the Workspace, ConvexAVS adds the corresponding page of control widgets to the Network Control Panel window. For modules whose output is an on-screen picture, ConvexAVS also creates a display window. Initially, this window is empty. When you complete a network and specify all the required input data, a picture appears in this window.

Complex networks may include several modules that produce pictures as output. ConvexAVS creates a separate window for each such module.

Picture Size and Window Size

When a picture first appears in a display window, ConvexAVS automatically resizes the window to fit the picture. (Since the size of the picture depends on the data being visualized, ConvexAVS cannot calculate the appropriate window size before data flows through the network.) If the window size subsequently changes, ConvexAVS automatically resizes the picture, if appropriate. In this connection, it is important to keep in mind the difference between images and pixmaps:

Images

An image is originally defined in terms of pixels. The only scaling ConvexAVS performs on images is successive doubling: x2, x4, x8, etc. These functions are available through the **display image** module.

When you resize an image window, there are several possibilities:

- If you make the window exactly two times as large (or four times, or eight times, etc.), the image is scaled and continues to fill the window exactly.
- If you make the window any other size, the window will no longer fit the image exactly. ConvexAVS chooses a scaling for the image that makes it too big for the window, rather than too small. Only part of the image will be visible; to see more of it, use any mouse button to click-and-drag the image.
- ConvexAVS refuses to scale an image as large as the new window size if such a scaling would severely strain system resources (e.g. main memory). In such cases, it chooses a smaller image size, so that part of the window remains unused.

Pixmaps

In most cases, a pixmap is originally defined as a geometry. ConvexAVS can scale pixmaps continuously. When you resize a pixmap window, the picture always resizes accordingly.

Using the Window Manager

All display windows are initially created as top-level X windows. This means that you can manipulate them using the X Window System's window manager program—move, iconify, resize, raise, lower, etc.

If you use the Edit layout function of the Network Editor to reorganize a network's control widgets, you may want to include the network's display windows in the reorganization. This topic is discussed in section Including Display Windows in a Reorganized Layout below.

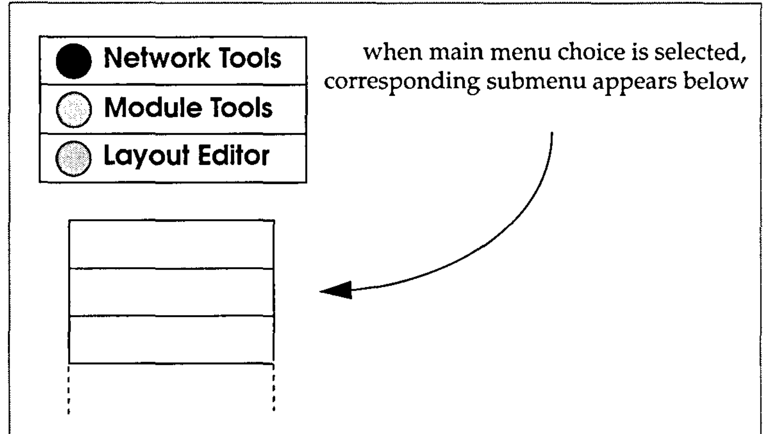
Caution

Don't use the xkill(1) program or any other means to delete a ConvexAVS display window or any other ConvexAVS window. This will cause the network to hang.

Using the Network Editor Menu System

The Network Editor has a two-level menu of functions that support your work in creating, revising, and executing networks. The top-level menu is always visible in the upper-left part of the Network Construction window (Figure 6-19). At any moment, one of the menu choices is selected, and the corresponding submenu appears below the main menu.

Figure 6-19
Network Editor Main Menu



The following sections describe the functions in the Network Editor submenus. Whatever submenu is currently active, the following two buttons always appear near the upper-left corner of the Network Construction window:

Help

Pops up a help browser window and displays the contents of the network_editor.txt help file. This file provides an overview of Network Editor usage. The browser shows the additional help topics that relate to the Network Editor.

Close

Closes the Network Construction Window, but does not delete the current network, if any. In fact, if a network is currently executing, it will continue to do so even though the Network Editor windows are closed.

A Display Network Editor button automatically appears at the top of the Network Control Panel window, providing a way to reopen the Network Construction Window at a later time.

Network Tools

The following functions are available when you select the **Network Tools** button:

Read Network

Reads an existing network from disk storage into the workspace. A file browser widget appears to help you specify the file containing the network definition.

If there is already a network (or even just a single module) in the Workspace, you must choose whether to Clear the existing network or to Merge the new one with the existing one. Merging can cause the module icons to overlap in the Workspace—use the left mouse button to rearrange them afterward.

Write Network

Writes the current network to disk storage. If you have already specified a file name for the current network with Read Network or Write Network, ConvexAVS offers to use the same name. If you choose not to, a file browser appears to help you specify the file name.

Note that performing a Read Network with the Merge does not change the name of the current network. You must select the Clear option to effect the name change or choose a new name when writing the network.

Clear Network

Deletes the current network and associated control panels. ConvexAVS displays a dialog box to have you confirm the selection.

Print Network

Create a Postscript™ file named */tmp/AVSnetwork.ps_proc* (where *proc* is the process number), which shows the layout of the current network. ConvexAVS composes a shell command to print the file, then displays a pop-up window showing this command. You must choose whether or not to issue the print command. (If you choose not to print, you may want to copy the PostScript file to another location, using an **xterm** window. The next time you select **Print Network**, the PostScript file will be overwritten.

Disable Flow Executive (toggle)

Modules perform their computations under the control of the Flow Executive, which determines when their output is required by another module and re-executes them if their most recent computation has become out of date. Disabling the Flow Executive inhibits all network execution. This is useful when you wish to change the values of several parameters, but you don't wish to have the network's modules recompute after each change.

Save Parameters

Saves the current module parameter values for the current network in a parameters file, named */tmp/avs_snapshot.parms_proc*. (*proc* is the process number.) This is useful if you want to provide yourself a checkpoint, to which you can return later in the same Network Editor session.

Restore Parameters

Resets the network's parameter values to those most recently saved. If appropriate (and if the Flow Executive is not disabled), the network recomputes. You cannot retrieve the parameters of another network, or of the same network from a previous Network Editor session.

Module Tools

The following functions are available with the **Module Tools** button:

Read Module(s)

Adds a module to one of the categories in the Palette. A file browser widget appears to help you specify the module program file. Each module specifies its category; you cannot choose a particular category when invoking this function.

It is possible for a single program file to define several modules. In this case, all the modules defined in the file are added to the Palette. You can also specify a directory, in which case the Network Editor loads all the modules defined in module program files within that directory.

Read Module Library

Empties the Module Palette, then adds all the modules in a specified library. A file browser widget appears to help you specify the library file to be read. After the library is loaded, the title bar above the Palette changes to display the name of the new library.

A library file names some combination of Convex-supplied modules, user-written modules, and directories that contain modules. For example:

```
builtin   render geometry
builtin   read geometry
builtin  display pixmap
file      /usr/johnp/avs_modules/smooth
file     /usr/johnp/avs_modules/rough
directory /usr/johnp/avs_modules/tools_dir
```

For details on creating module libraries see the "File Formats" appendix.

Write Module Library

Writes a module library file, consisting of the modules currently in the Palette.

Select Module Library

Invokes a pop-up menu, allowing you to select one module library from all the ones that have already been selected with Read Module Library during the current Network Editor session. The default library— the one automatically loaded at the beginning of the session— is listed, too. This function provides a convenient way to switch back and forth among different sets of modules.

Flash Active Modules (toggle)

If this toggle switch is on, module icons are highlighted (displayed with a black background) as the modules execute. Turning this off may speed up the execution of highly interactive networks.

Verbose Mode (toggle)

If this toggle switch is on, ConvexAVS displays debugging information as the modules execute. The information is sent to the *stderr* of the **avs** command that started the ConvexAVS session. Typically, the information is displayed in the **xterm** window from which you typed the **avs** command.

Layout Editor

Selecting Layout Editor places the Network Editor in a mode that allows you to redesign the user interface of the current network. By default, each module in the network has its own control panel, and all the control panels are assembled into the Network Control Panel window. You can switch among the various modules' panels, but you can see (and work with) only one at a time.

The facility for editing the layout of the Network Control Panel includes these features:

- Changing the widgets that provide interactive control over parameter values as a network executes. For example, you might change a dial into a slider, or into a type-in.
- Moving widgets around within their control panels.
- Moving widgets to other control panels.
- Creating new control panels. You might create a new panel, then move widgets from various existing control panels to the new one.

Any changes you make to the Network Control Panel layout are automatically saved and restored by the Write Network and Read Network functions under Network Tools.

Elements of a Layout

The user interface to a network consists of control widgets that are organized hierarchically:

- Individual control widgets (sometimes simply called controls) correspond to the input parameters of the modules in the network.
- A page is a window that contains one or more control widgets. The page construct allows you to see all of its control widgets at the same time.

  By default, all of a module's control widgets are assembled onto a single page.
- A stack is a window that contains one or more elements (typically, pages). The stack construct allows you to see just one element at a time. You can switch among them by clicking in the choice menu at the top of the stack window. (This menu is automatically created as you add elements to the stack.)

The Network Control Panel window is, itself, a stack. ConvexAVS automatically assembles all the pages of control widgets for a network—one page for each module— into this stack.

Working with the Layout Editor

When you select Layout Editor, the following submenu items appear:

Create Page

Creates a new, empty control panel page.

Create Stack

Creates a new, empty stack.

Undo

Undoes the effect of the most recent layout operation. Clicking Undo repeatedly will step back at most five actions.

This feature does not undo the creation of new pages and stacks. It does not undo the effects of X window manager actions.

ConvexAVS creates files with names of the form */tmp/avs_undoN.lyt_proc* to implement the **undo** feature. (N is a small integer and *proc* is the process number.) Don't delete these files during a Network Editor session if you want to use the **undo** feature. They are automatically deleted whenever you start ConvexAVS or perform a **Clear Network.**

Click these choices to create additional places in which to organize the network's control widgets. Then, use the mouse buttons to re-arrange the control widgets. To add a widget or page (or even another stack) to a stack, move it onto the stack's set of buttons; a new button appears for the newly-added item.

Note

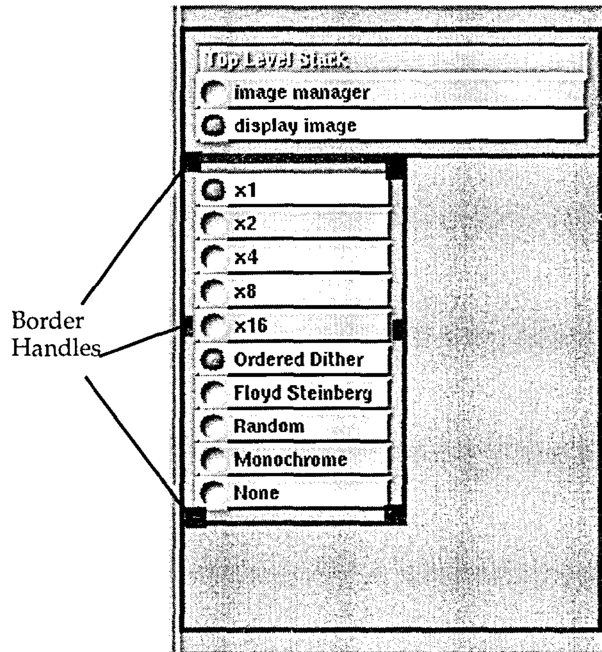
In the Layout Editor, the mouse buttons modify the layout of control widgets, pages, and stacks rather than changing the values of parameters. Red borders around the control widgets, pages, and stacks remind you that you are in this Layout Editor mode.

You can resize pages to allow them to accommodate more control widgets. You can reorganize pages into one or more new stacks. (You can also place individual control widgets, or even other stacks, within a stack.)

As you move the mouse, elements whose layout can be changed are outlined in white. A simple border means the window can be moved or deleted (Figure 6-20); a border with a series of handles (corner and side boxes) can be resized, as well.

While in the Layout Editor, most of the titles on pages and stacks become type-in widgets. You can edit the titles just as you would use any type-in.

Figure 6-20
Window Borders in the Layout Editor



You can move, resize, and delete control widgets as follows:

- **Left Button:** Move element. You can move any type of element— control widget, page, or stack. The destination can be elsewhere within the same page, to a different page or to the root window. In the latter case, the control widget becomes a top-level window, and can be manipulated using the X window manager.
- **Middle Button:** Resize a control panel or stack. (Not supported for individual control widgets— a question mark cursor appears.) Click and hold down the button, then drag the cursor through the edge or corner you want to move.
- **Right Button:** Pops up a menu appropriate to the element. The menu may include:
  - **Delete:** Delete the element. If you delete a control widget, you'll have no way to affect the value of the associated input parameter when the network executes. Deleting a page or stack effectively deletes all the control widgets it contains.

Note

If you did not mean to delete the element, select Undo immediately. You may also need to perform a Reconfigure (see below) to adjust the page size. To recover a control widget after it is too late for an Undo, you must invoke the Parameter Editor. In the Workspace, find the module whose parameter is associated with the deleted control widget. Click the small square button on the icon with the middle or right mouse button to open the Module Editor window. In the Parameter Editor section of this window, click on the desired parameter. This pops up a menu of choices (e.g. dial, slider, type-in) for the form in which the control widget is to be reinstated.

- **Add Title:** Add a title box to a page (if one doesn't already exist). To edit the title, move the cursor into the title box, and type in a new title. You'll probably want to start by using BACKSPACE (delete last character) or Ctrl-U (delete entire title). Don't forget to press Return to finish the title. The title box is itself an element that can be moved, deleted, or edited. It cannot, however, be resized or moved out of its original page.
- **Reconfigure:** Resize a page or stack to fit its contents.
- **Control widget type (radio buttons):** Change the type of control widget (e.g. change slider to dial). You can also change the type of a parameter's control widget using the Parameter Editor, as described above.

Including Display Windows in a Reorganized Layout

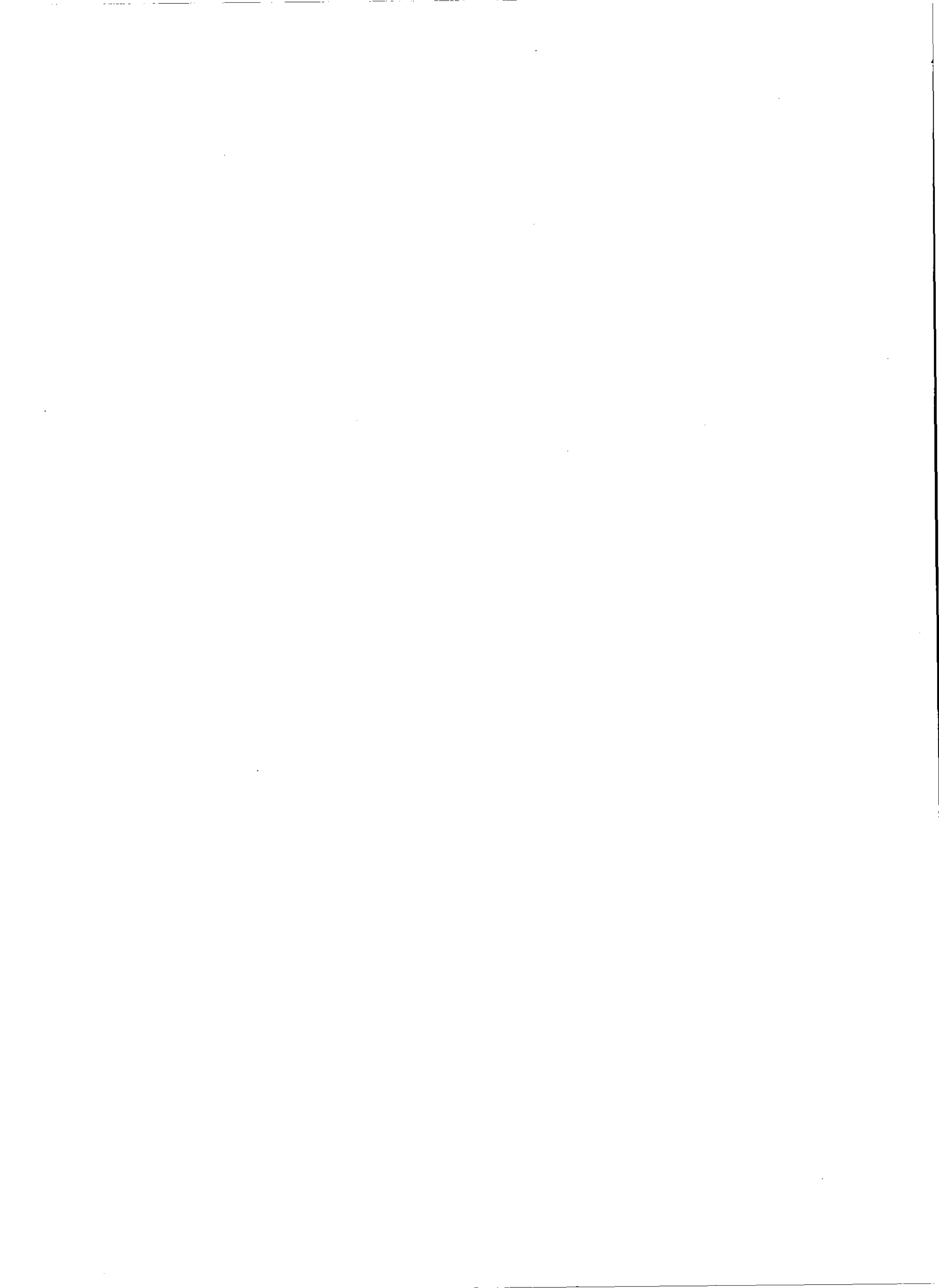
You can include the display windows created by the display image and display pixmap modules in a reorganized layout. Make sure that the page or stack into which you want to move the window is large enough. Then, move the window using the left mouse button, just as you would move any control widget. The display window is automatically subsumed under the page or stack, so that you can no longer manipulate it using the X window manager. If you subsequently move the display window out of its page or stack, it becomes a top-level X window again.

The **Zoom** function temporarily makes the window a top-level X window; **Unzoom** returns it to the page or stack whence it came.

Developing Applications

This part provides the fundemental information about the data types, module construction, and data flow.

- Fundamentals
- Data Types
- Modules



Fundamentals 7

Modules

The fundamental unit of computation in ConvexAVS is the *module*. Modules are high-level units of computation. For example, a module might be designed to simulate fluid flow through a nozzle.

You can affect the action of the computation by changing module parameters at runtime.

You can extend the capabilities of ConvexAVS by writing a new module. Because ConvexAVS operates on general data types, a new module can work with preexistent modules to perform computation.

Modules consist of two functions: the *description* function and the *computation* function. Most functions can be written in the C or FORTRAN language. The description function describes what data the module takes as input, what data it produces as output, and parameters that control its behavior. The computation function is called with its inputs, parameters, and outputs as arguments; the computation function typically operates on the inputs and parameters to produce new output.

There are two kinds of modules: *subroutines* and *coroutines*. A subroutine module is invoked by ConvexAVS, usually whenever its inputs or parameters change. A coroutine module executes independently, obtaining inputs from ConvexAVS and controlling its outputs.

Simulations and scientific applications can be converted into ConvexAVS coroutine modules. Conversion includes making the application use ConvexAVS data types, inserting calls to transmit data to and from ConvexAVS, and writing a description function.

Data Types

There are two types of data in ConvexAVS: *primitive* and *aggregate*. Primitive data items are objects, such as floating-point numbers and text strings. Aggregate data items are large chunks of data that scientific applications use. One fundamental type of aggregate data is called fields. These implement array structures (either uniform or non-uniform, scalar or vector) as well as unconnected and irregular structures. ConvexAVS also has other types of aggregate data: geometries, colormaps, and pixel maps.

Networks

You build an application by constructing a *network* of modules. An example network might consist of modules performing the following tasks:

- Importing data from outside ConvexAVS (or generating their own data) and converting it into data of one of the ConvexAVS data types. This is called a data module.
- Transforming ConvexAVS data in some way to produce output data of:
  - the same or different type (called a filter module)
  - a different type that is more complex than a filter's output (called a mapper module)
- Rendering or storing ConvexAVS data on an external device, such as the display screen or a file. This is called a renderer module.

A module can receive data through an input port and transmit data through an output port. Connecting two modules is actually connecting an output port from one module to an input port of another module. Two ports can be connected when they have matching ConvexAVS data types.

Data Flow

Networks provide a data-processing pipeline in which, at each step, the output of one module becomes the input of another. In this way, data can enter ConvexAVS, flow through the modules of a network, and be rendered on a display or stored outside ConvexAVS.

This process requires that each module in a network be invoked at the appropriate time. For a subroutine module, the computation function must be executed whenever the inputs or parameters change. ConvexAVS has a flow executive that is active during the life of the application. The flow executive supervises data movement between modules, keeps track of which parameters have changed, and invokes modules in the correct order.

ConvexAVS uses a remote procedure call to establish communication between modules. When you start a module, ConvexAVS creates a new process in which that module runs. It also sets up a connection between the module and ConvexAVS. Both sides use remote procedure calls to communicate through this connection.

ConvexAVS allows coroutine modules to execute independently. A coroutine can be a simulation or animation (an application that executes multiple times to produce a series of frames or data sets). ConvexAVS communicates with coroutine modules through the same kind of remote procedure call it uses to communicate with subroutine modules.

In ConvexAVS V1.0, only one module executes at a time; modules do not execute in parallel.

Data Types

8

Primitive and Aggregate

ConvexAVS supports primitive and aggregate data types. Primitive data types are used for parameters and aggregate types are used for data passed between modules. Data type categories are:

- Primitive:
  - *Byte* implements 8-bit bytes.
  - *Integer* implements standard 32-bit integers.
  - *Real* implements 32-bit IEEE single-precision floating-point numbers.
  - *String* implements simple text strings.
- Aggregate:
  - *Field* implements *n*-dimensional arrays with scalar or vector data at each point. Fields also can map rectilinear or irregular sets of points in arbitrary coordinate systems to scalar or vector data. Fields can contain floating-point, integer, or byte data.
  - *Colormap* implements a transfer function that can be used to map a functional value into color and opacity values.
  - *Geometry* implements geometric descriptions that can be used by the geometric renderer to view objects. Geometry objects are created using calls to subroutines in the geom library.
  - *Pixel map* refers to the X server's representation of the rendered form of an image.

Fields are the fundamental data type. They use the full generality of the type system to span a set of commonly used data types. This allows you to write modules that are as general as possible for the application while allowing optimized algorithms to be used for specific cases. Output data from a scientific simulation can be represented as a field. ConvexAVS routines allow conversion of standard arrays of data to fields.

When ConvexAVS calls a C language computational routine, it passes an element of a certain data type as a pointer to that element. Most data types are represented as structures, which are defined in type-specific include files. Primitive types, such as integers, are passed directly. C routines get direct pointers to the data for inputs and parameters, but pointers to pointers are used to allocate data for outputs. A module that takes a field as input and produces a field as output is called as follows:

```
module_compute(field_in, field_out)
/* note double indirection for field_out */
AVSfield_float *field_in, **field_out;
{
 float *data_out;
 AVSfield_float *result;

 dim[3];

 dim[0] = MAXX(field_in);
 dim[1] = MAXY(field_in);
 dim[2] = MAXZ(field_in);
 result = (AVSfield_float *) AVSdata_alloc
          ("field 3d real", dim);
 ... compute ...
 *field_out = result;
 return(1);
}
```

Because FORTRAN programs do not use structures in the same way as C programs, FORTRAN computation routines get their arguments as separate elements. For example, a subroutine that takes a 3D scalar field as input gets arguments in the following form:

FUNCTION COMPUTE(*F, NX, NY, NZ*, ...)

F is a 3D array with dimensions *NX, NY, NZ*. ConvexAVS attempts to make the arguments to the computation function a natural representation of that data type for the programmer. The implication is that the computation routine written in FORTRAN often has more formal arguments than there are inputs, outputs, and parameters, with multiple formal arguments representing a single input, output, or parameter.

The type declarations in Table 8-1 are used for arguments to module computation functions that correspond to input ports, parameters, and output ports.

A field is passed to a FORTRAN routine as multiple arguments. Refer to Section "Manipulating Fields from FORTRAN." A colormap is input to a FORTRAN routine as a series of parameters. Refer to Section "Colormaps." A FORTRAN routine cannot take a pixel map as an argument.

Table 8-1
Field Declarations for Data Types

Data Type	C Input	C Output	FORTRAN Input	FORTRAN Output
byte	char	char *	BYTE	BYTE
integer	int	int *	INTEGER	INTEGER
real	float *	float **	REAL	Pointer to REAL
string	char *	char **	CHARACTER*(*)	Pointer to CHARACTER*(*)
field	AVSfield *	AVSfield **	--	–
colormap	AVScolormap *	AVScolormap **	--	--
geometry	GEOMedit_list	GEOMedit_list *	INTEGER	INTEGER
pixel map	AVSpixdata *	AVSpixdata **	--	--

Byte

Bytes are declared using the data type "byte." A byte is passed to a computation routine in C as a char (char * for output) and to a subroutine in FORTRAN as a BYTE.

Integer

Integers are declared using the type "integer." An integer is passed to a subroutine in C as an int (int * for output) and to a subroutine in FORTRAN as an INTEGER. ConvexAVS has data types for parameters that are also represented as integers:

- "boolean"
- "tristate"
- "oneshot"

Refer to AVSadd_parameter in Appendix E, "Module Routines."

Real

ConvexAVS supports floating-point numbers in IEEE format. Single-precision floating-point numbers are declared using the type "real." This corresponds to the C type float and to the FORTRAN type REAL or REAL*4. A single-precision floating-point number is passed to a computation routine in C as a float * (float ** for output) and to a subroutine in FORTRAN as a REAL (a pointer to a REAL for output).

String

Text strings are one-dimensional character strings. A character string is declared using the type "string." It is passed to a computation routine in C as a char * (char ** for output) and to a subroutine in FORTRAN as a CHARACTER *(*) (a pointer to a CHARACTER *(*) for output).

Field

A field is a general representation for an array of data. The array can have any number of dimensions, and the dimensions can be of any size. Each data element in the array can consist of one value or a vector of values. All values in the array are of one of four primitive types:

- Character (byte).
- Integer.
- Single-precision floating point.
- Double-precision floating point.

A field represents data elements that correspond to points in space. For example, each data element of a three-dimensional field might be a vector of values representing temperature, pressure, and velocity at some point in a volume of fluid. The field has an implicit or explicit mapping of data elements to coordinates that represent their corresponding points in space. In other words, a field is a relation between two kinds of space: the computational space of the field data and the coordinate space to which the field data is mapped.

Mapping Computational Space to Coordinate Space

ConvexAVS assumes that the computational space is logically rectangular. In the computational domain, the mesh is similar to a uniformly spaced lattice in Cartesian space. This is only true for uniform fields and not necessarily all fields.

ConvexAVS supports three types of mapping: uniform, rectilinear, and irregular.

Uniform Fields

In uniform fields, coordinate mapping is direct and implicit. Coordinate space has the same number of dimensions as computational space. Each dimension of computational space is implicitly mapped to the corresponding axis of coordinate space. The first dimension of computational space is implicitly mapped to the X-axis, the second dimension is implicitly mapped to the Y-axis, etc. In each dimension, the coordinate that corresponds to a given data element is the index of that element in the data array. Data is mapped to a uniformly spaced lattice in Cartesian space. Each cell is a constant-length line segment for a 1D field, a square for a 2D field, a cube for a 3D field, or a hypercube for a field of higher dimensions. Because the coordinate mapping is implicit, the field does not need any coordinate information separate from the data array.

Rectilinear Fields

In rectilinear fields, as in uniform fields, coordinate space has the same number of dimensions as computational space and data is mapped to a lattice in Cartesian space. However, each dimension of the data array has a separate and explicit coordinate mapping. The spacing of data elements along each axis need not be uniform. Each cell is a variable-length line segment for a 1D field, a rectangle for a 2D field, a rectangular parallelepiped for a 3D field, etc. Cell dimensions can vary from one cell to the next within the field.

Irregular Fields

In irregular fields, coordinate space might not have the same number of dimensions as computational space. Each data element in computational space is explicitly mapped to a point in coordinate space. This allows for a variety of mappings. For example, a 3D computational space can be mapped to a 3D coordinate space in which each cell has curvilinear bounds. A 1D computational space can be mapped to a 2D or 3D coordinate space that has a set of scattered points with a data element at each point.

Mapping Information

For a uniform field, ConvexAVS needs no explicit mapping information. The X-coordinate for a data element is simply the subscript of the data element along the first dimension of computational space. The Y-coordinate is the subscript of the data element along the second dimension of computational space.

For a rectilinear field, ConvexAVS needs a mapping from each dimension of computational space to the corresponding axis of coordinate space. The mapping consists of one X-value for each subscript along the first dimension of computational space, one Y-value for each subscript along the second dimension of computational space, etc. The total number of values in the mapping is the sum of the dimensions of the field in computational space.

For an irregular field, ConvexAVS needs a mapping from each data element in computational space to a point in coordinate space. The mapping consists of a set of coordinates (X, Y, etc.) for each data element. The total number of values in the mapping is the product of each dimension in computational space and the number of dimensions in coordinate space. Table 8-2 summarizes these mappings.

Table 8-2
Field Mappings

Mapping	Mapping Information		Coordinates
Uniform	Implicit	Computational Dimension to Coordinate Axis	$X = i$ $Y = j$ ...
Rectilinear	Explicit	Computational Dimension to Coordinate Axis	$X = X(i)$ $Y = Y(j)$ ...
Irregular	Explicit	Computational Element to Coordinate Point	$X = X(i, j, ...)$ $Y = Y(i, j, ...)$ ...

Examples of Field Mappings

This section presents examples of fields and their mappings from computational to coordinate space.

Example 1 - Uniform Field

A two-dimensional image is represented by a mesh of data elements, each of which specifies the value of a pixel. Each data el ement is a vector of four bytes that specify the three color components and an alpha channel. The field consists of 65,536 el ements, each with four values:

$\{ V_n(i,j), i=1,256, j=1,256, n=1, 4 \}$

The field is uniform.

Following is a summary of the field characteristics:

Data type: Byte
Number of values per data element: 4
Number of computational dimensions: 2
Computational dimensions: 256 by 256
Number of computational values: 4*256*256 = 26,2144
Mapping type: Uniform
Number of coordinate dimensions: 2
Number of coordinate values: 0

Example 2 - Uniform Field

A medical imaging data set contains 100 evenly spaced scan planes, each with a resolution of 256 by 256 pixels. Each data element is a single byte. The field consists of 6,553,600 elements:

$\{ F(i,j,k), i=1,256, j=1,256, k=1,100 \}$

The field is uniform.

Following is a summary of the field characteristics:

Data type: Byte
Number of values per data element: 1
Number of computational dimensions: 3
Computational dimensions: 256 by 256 by 100
Number of computational values: 1*256*256*100 = 6,553,600
Mapping type: Uniform
Number of coordinate dimensions: 3
Number of coordinate values: 0

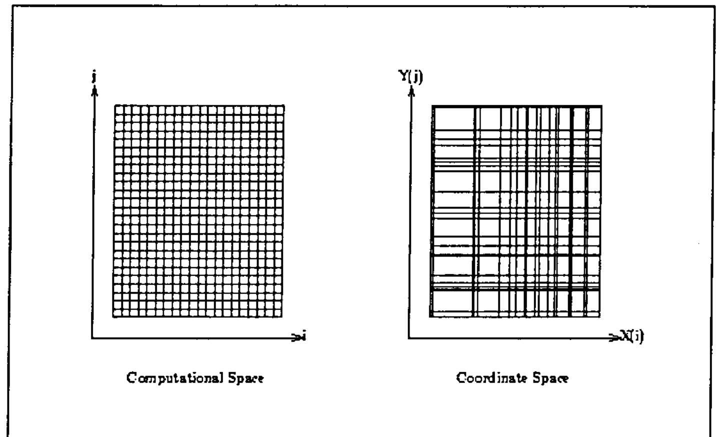
Example 3 - Rectilinear Field

A scalar field is defined as a two-dimensional mesh with non-constant spacing between both X- and Y-values. The field consists of 500 elements:

$\{ F(X(i), Y(j)), i=1,20, j=1,25 \}$

The field is rectilinear with 20 X-coordinates and 25 Y-coordinates. Each cell in coordinate space is rectangular. Figure 8-1 shows the mapping between computational and coordinate space.

Figure 8-1
Rectilinear Field



Following is a summary of the field characteristics:

Data type: Floating-point
Number of values per data element: 1
Number of computational dimensions: 2
Computational dimensions: 20 by 25
Number of computational values: 1*20*25 = 500
Mapping type: Rectilinear
Number of coordinate dimensions: 2
Number of coordinate values: 20 + 25 = 45

Example 4 - Rectilinear or Irregular Field

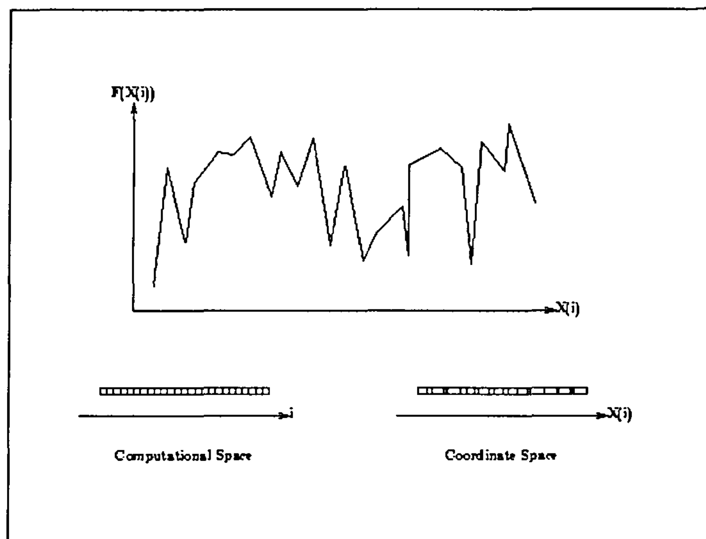
A data set consists of 25 data elements, each representing F(X) for a given value of X. The field consists of 25 elements:

{ *F(X(i)), i*=1,25 }

Computational space is one dimensional with 25 values for *F(X)*. Coordinate space is also one dimensional with 25 X-coordinates, one for each value of *F(X)*. The spacing between points in X is not constant, so the field is rectilinear or irregular.

Figure 8-2 shows the mapping between computational and coor dinate space. It also presents a line graph, *F(X(i))* versus *X(i)*, of the relation between the data elements and the coordinate values.

Figure 8-2
Rectilinear or Irregular Field



Following is a summary of the field characteristics:

Data type: Floating point
Number of values per data element: 1
Number of computational dimensions: 1
Computational dimensions: 25
Number of computational values: 1*25 = 25
Mapping type: Rectilinear or irregular
Number of coordinate dimensions: 1
Number of coordinate values: 50

Suppose that each data element in this example consisted of a two-component velocity vector. In this case the field characteristics would be:

Data type: Floating point
Number of values per data element: 2
Number of computational dimensions: 1
Computational dimensions: 25
Number of computational values: 2*25 = 50
Mapping type: Rectilinear or irregular
Number of coordinate dimensions: 1
Number of coordinate values: 50

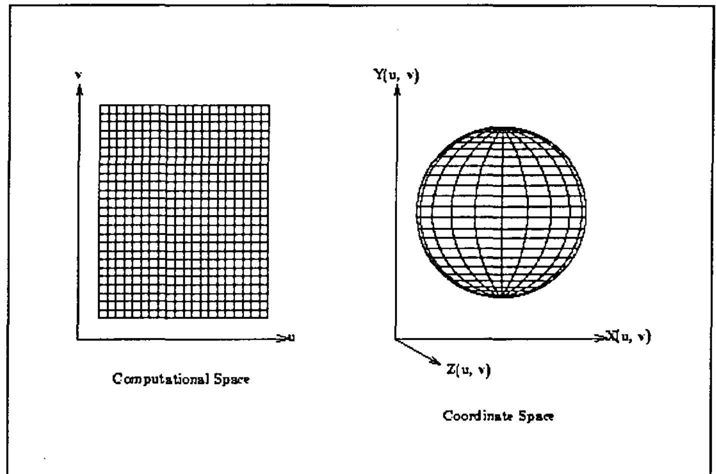
Example 5 - Irregular Field

A two-dimensional mesh is mapped to a sphere. One dimension of the mesh, *u*, corresponds to lines of equal longitude on the sphere. The other dimension of the mesh, *v*, corresponds to lines of equal latitude on the sphere. The field consists of 500 elements:

$\{ F(X(u,v), Y(u,v), Z(u,v)), u=1,20, v=1,25 \}$

The field is irregular, with 500 X-coordinates, 500 Y-coordinates, and 500 Z-coordinates. Each cell in coordinate space has curvilinear bounds. Figure 8-3 shows the mapping between computational and coordinate space.

Figure 8-3
Irregular Field



Following is a summary of the field characteristics:

Data type: Floating point
Number of values per data element: 1
Number of computational dimensions: 2
Computational dimensions: 20 by 25
Number of computational values: 1*20*25 = 500
Mapping type: Irregular
Number of coordinate dimensions: 3
Number of coordinate values: 3*20*25 = 1500

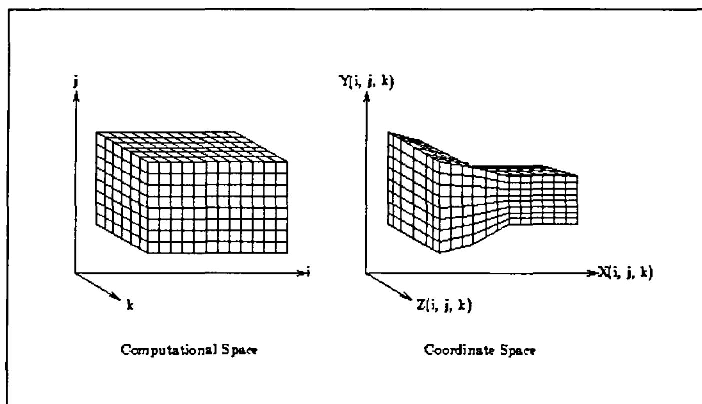
Example 6 - Irregular Field

A fluid-dynamics application is a three-dimensional simulation of fluid flow through a nozzle. Each data element has five values: a three-component velocity vector, temperature, and density. The field consists of 576 elements, each with five values:

$$\{ V_n(X(i,j,k), Y(i,j,k), Z(i,j,k)), i=1,12, j=1,8, k=1,6, n=1,5 \}$$

The field is irregular, with 576 X-coordinates, 576 Y-coordinates, and 576 Z-coordinates. Many of the cells in coordinate space have curvilinear bounds. Figure 8-4 shows the mapping between computational and coordinate space.

Figure 8-4
Irregular Field



Following is a summary of the field characteristics:

Data type: Floating point
Number of values per data element: 5
Number of computational dimensions: 3
Computational dimensions: 12 by 8 by 6
Number of computational values: 5*12*8*6 = 2,880
Mapping type: Irregular
Number of coordinate dimensions: 3
Number of coordinate values: 3*12*8*6 = 1,728

Field Components

As represented in ConvexAVS, a field has the following components:

- The number of dimensions in computational space (ndim). This is an integer.
- The dimensions in computational space (int *dimensions). This is an array of integers whose length is the number of dimensions in computational space. Each element of the array is the number of data elements along the corresponding dimension of computational space.

- The number of variables or values for each data element (veclen). This is an integer. A field with one value for each data element is a scalar field. A field with more than one value for each data element is a vector field. A field can also consist only of coordinates with no values for each data element. In this case, the field represents a list of points in coordinate space.
- The data type of each value for the data elements (type). This is an integer. The data type can be character (byte), integer, single-precision floating-point, or double-precision floating-point. ConvexAVS defines a constant to represent each data type: AVS_TYPE_BYTE, AVS_TYPE_INTEGER, AVS_TYPE_-REAL, and AVS_TYPE_DOUBLE. These constants are defined in the avs.h include file for C programs and avs.inc for FORTRAN programs.
- The array of data elements representing the computational space of the field. Each element of the array is a value for a data element of the field. For a vector field, this array has one more dimension than the number of dimensions in computational space; the extra array dimension is the number of values per data element. The size of the array is the product of each dimension in computational space and the number of values per data element. Elements of the array are stored in FORTRAN order with all values for each data element kept together. The array subscript for the value per data element varies fastest, followed by the subscript for the first dimension, the subscript for the second dimension, etc. If n_value is the subscript for the value per data element and i, j, and k are subscripts for the first, second, and third dimensions, respectively, the array is accessed in C as follows:

  ```
  data[k][j][i][n_value]
  ```

  The same array is accessed in FORTRAN as follows:

  ```
  DATA(N_VALUE, I, J, K)
  ```

  ConvexAVS has a number of macros that makes access to this array more convenient for C language programmers. Refer to Appendix F, "C Language Field Macros."
- A flag indicating the type of mapping from computational space to coordinate space (uniform). This is one of the following constants: UNIFORM, RECTILINEAR, or IRREGULAR. These constants are defined in the field.h include file for C programs and avs.inc for FORTRAN programs.
- The number of dimensions in coordinate space (nspace). This is an integer. For a uniform or rectilinear field, this is the same as the number of dimensions in computational space. For an irregular field, this can differ from the number of dimensions in computational space.

- For a rectilinear or irregular field, an array of floating-point values representing the coordinates of the field.

For a rectilinear field, this array contains one X-value for each subscript along the first dimension of computational space, one Y-value for each subscript along the second dimension of computational space, etc. The coordinate array has one dimension, and the size of the array is the sum of the dimensions in computational space. All the X-coordinates corresponding to the first dimension of computational space are stored first, all the Y-coordinates corresponding to the second dimension of computational space are stored second, etc. If i, j, and k are subscripts for the first, second, and third dimensions of computational space, and if idim1, idim2, and idim3 are the first, second, and third dimensions of computational space, the X-, Y-, and Z-coordinates are obtained in C as follows:

```
x = points[i]
y = points[idim + j]
z = points[idim + jdim + k]
```

The coordinates are obtained in FORTRAN as follows:

```
X = POINTS(I)
Y = POINTS(IDIM + J)
Z = POINTS(IDIM + JDIM + K)
```

For an irregular field, this array contains a set of coordinates (X, Y, etc.) for each data element in computational space. The coordinate array has one more dimension than the number of dimensions in computational space; the extra array dimension is the number of dimensions in coordinate space. The size of the array is the product of each dimension in computational space and the number of dimensions in coordinate space. All the X-coordinates are stored first, then all the Y-coordinates, etc. The subscript for the first dimension of computational space varies fastest, followed by the subscript for the second dimension of computational space, etc. The subscript for the dimension of coordinate space (X, Y, etc.) varies most slowly. If n_coord is the subscript for the dimension of coordinate space and i, j, and k are the subscripts for the first, second, and third dimensions of computational space, the array is accessed in C as follows:

```
points[n_coord][k][j][i]
```

The same array is accessed in FORTRAN as follows:

```
POINTS(I, J, K, N_COORD)
```

ConvexAVS has a number of macros that makes access to this array more convenient for C language programmers. Refer to Appendix F, "C Language Field Macros."

Declaring Fields

When declaring or allocating fields, a programmer uses a field-type string. This string consists of the word "field" followed by words describing each way in which the field is specialized, such as "field 3D scalar uniform float." When declaring input and output ports (with AVScreate_input_port or AVScreate_output_port), you can omit particular specifications to indicate that your module can accept or produce a more general data type. For example, a module writer can declare an input port as accepting "field scalar" to indicate that the module accepts any type of scalar field.

The ConvexAVS flow executive does not permit you to connect a module's output to another module's input if the output and input are declared to be conflicting types of fields. For example, ConvexAVS does not allow a "field 2D" output to be connected to a "field 3D" input. However, ConvexAVS does allow an output and an input to be connected if one is a subtype of another. For example, ConvexAVS allows a "field" output to be connected to a "field 2D" input.

If a module accepts some subtypes of fields but not all, it checks the inputs and signals an error by calling AVSmessage if the input is of a type it does not accept. That is, if a module accepts 2D and 3D scalar uniform fields of floating-point numbers, it should declare the input as "field scalar uniform float," then the module's computation routine should check the number of dimensions in the input field.

In a field declaration, the word "field" is mandatory and is always the first word in the string. Specializing words are optional and can appear in any order. Table 8-3 lists possible specializing words.

For the number of dimensions of coordinate space, any string beginning with "n-coord" is acceptable. For example, ConvexAVS recognizes "3-coords," "3-coordinate," and "3-coordinates."

Manipulating Fields from C

When a C language module has declared an input port, output port, or parameter to be a field, the computation routine is called with one argument corresponding to each field. If the field is an input port or parameter argument, the subroutine parameter is declared as AVSfield*. If the field is an output port, the subroutine parameter is declared as AVSfield**.

Four types of AVSfield structures are defined in <avs/field.h>. Each field structure supports a different data type:

Table 8-3
Field Declarations and Specializing Words

Field Component	Value	Specializing Words
Number of Dimensions	n	nD
Vector Length	1	scalar, 1-vector
	n	n-vector
Data Type	byte	byte, char
	integer	integer, int
	real	real, float
	double	double, real*8
Number of Coordinates	n	n-coord, n-space
Mapping Type	uniform	uniform
	rectilinear	rectilinear
	irregular	irregular

Field Type	Data Type
AVSfield_char	Byte
AVSfield_int	Integer
AVSfield_float	Real
AVSfield_double	Double

The only difference among these types is the type declaration for the data array. For the generic type AVSfield, the data is defined to be a union. Refer to the /usr/avs/field.h include file for more information.

An AVSfield structure appears in Figure 8-5 (using AVSfield_char as an example).

To illustrate the relation between field declarations and elements of the field structure, use the example of a field representing fluid flow through a nozzle. The field has three dimensions in computational space, 12 by 8 by 6. Each data element has five floating-point values. The field is irregular with a three-dimensional coordinate space. The declaration for that field is:

```
"field 3D 5-vector real 3-coordinate irregular"
```

The corresponding members of the AVSfield structure and their values are:

Figure 8-5
Example AVSfield Structure

```
typedef struct {
 int ndim;               /* Number of dimensions in the field */
 int nspace;             /* Number of physical coordinates per point */
 int veclen;             /* number of components at each point */
 int type;               /* data type (see below for values) */
 int size;               /* size of each element */
 int single_block;       /* reserved; 1 if field is a single malloc */
                         /* 2 if using shared memory */
 int uniform;            /* != 0 if field is uniform (points = null) */
 int *dimensions;        /* dimension along each axis, length is ndim */
 float *points;          /* real-world coords for non-uniform fields */
 unsigned char *data;    /* the field itself as chars */
 int shm_key;            /* shared memory key */
 char *shm_base;         /* shared memory base address */
} AVSfield_char;
```

Member	Value
ndim	3
nspace	3
veclen	5
type	AVS_TYPE_REAL
size	sizeof(float)
uniform	IRREGULAR
dimensions	dims[3] = {12, 8, 6}
points	coords[3][6][8][12]
data	data[6][8][12][5]

The field.h include file defines preprocessor macros to help C programmers gain access to components of a field, including dimensions in computational space, the data array, and the coordinate array. For more information, refer to Appendix F, "C Language Field Macros."

Manipulating Fields from FORTRAN

FORTRAN modules that manipulate field data must call the following routine from the description function:

```
CALL AVSSET_MODULE_FLAGS(SINGLE_ARG_DATA)
```

This allows ConvexAVS to distinguish the module from older modules that use an obsolete method of accessing field data.

An input or output field is represented by a single integer value in FORTRAN. The FORTRAN module then accesses field structures using accessor functions on the integer value rather than by directly accessing the structure as a C module does. For both input and output fields, the integer argument is actually a pointer to a field pointer. This is unlike C that declares input fields and output fields differently. For example, a computation routine that has one input field and one output field would be declared like this:

```
FUNCTION COMPUTE(INFIELD, OUTFIELD)
INTEGER INFIELD, OUTFIELD
```

Most of the accessor functions either return the requested information or copy it into an array. For example, instead of referencing infield->ndim as in C, FORTRAN calls AVSfield_get_int:

```
NDIM=AVSFIELD_GET_INT(INFIELD,AVS_FIELD_NDIM)
```

The avs.inc include file includes the necessary function declarations and accessor constants. Field arrays that are of predictable size, such as the dimensions array, are filled directly by the accessor functions. Ensure that arrays passed in are large enough for the maximum expected dimensions.

Accessing either data or points array in a field becomes more complicated because arrays are arbitrarily large. There are two approaches to accessing each array. The first approach returns an offset index *N* between a local FORTRAN array and the actual field data array. The *N*+1 element of the local FORTRAN array is the same as the first element of the desired array. This element reference can then be passed into a second function that declares it to be an array of a particular type and dimensionality. This approach is a little awkward but portable. An example is provided in /usr/avs/examples/fthres2.f. The routines used are AVSfield_data_offset and AVSfield_points_offset.

The second approach gets the field data array pointer back as an integer then pass the %VAL() of this value to a second function that declares it as an array of the anticipated type and dimensions. This approach is easier than the first but less portable. Some FORTRAN compiles provide %VAL, others %LOC, and some might not support this non-ANSI feature. An example is provided in /usr/avs/examples/fthres1.f. The routines used are AVSfield_data_ptr and AVSfield_points_ptr.

Creating Fields

For allocating and freeing field structures, ConvexAVS provides several routines accessible from both C and FORTRAN. These routines create a field that is internally self consistent (for example, if it contained a 20 by 30 2D computational array, the appropriate data and points arrays are automatically allocated) and takes advantage of shared memory storage. For example, to create a "field 3D 30vector real rectilinear" of size 20 by 20 by 20, the following call is made:

```
output_field=AVSdata_alloc("field 3D 3-vector real
                              rectilinear",dims)
```

where dims is a three-element integer array containing [20, 20, 20,]. If an existing field is available as a template, AVSfield_alloc may be called to make a duplicate. Fields can be freed using AVSdata_free or AVSfield_free. For example:

```
AVSdata_free('field', output_field)
```

or

```
AVSfield_free(output_field)
```

AVSbuild_field, AVSbuild_2d_field, and AVSbuild_3d_field are also provided as an alternate way to build certain special-case fields.

Scatter Data

A *scatter* is a list of points in coordinate space with an optional scalar or vector data element for each point. ConvexAVS represents scatters as 1D irregular fields. For example, a scatter with scalar real data and 3D coordinates would be declared as a "field 1D scalar real 3-coordinate irregular." The one dimension of the field in computational space is the number of points in the scatter. The length of the data array is the product of the number of points in the scatter and the number of values per data element at each point.

A module can declare a scatter to have no data by declaring the vector length to be 0. For example, a scatter with no data and 3D coordinates would be declared as "field 1D 0-vector 3-coordinate irregular." Such a field has no data array. The number of dimensions should still be declared to be 1, and the one dimension of the field in computational space is still the number of points in the scatter. This dimension is necessary to calculate the length of the coordinate array.

Image Data

ConvexAVS represents two-dimensional images as 2D uniform vector fields. Each vector contains four elements of byte data, and each byte represents one component of a pixel value. An image is declared as a "field 2D 4-vector byte." The following information shows which vector element corresponds to each component of the pixel value:

Byte	Component
0	blue
1	green
2	red
3	alpha

The information is zero-based, as in a C language vector. In FORTRAN, the vector index is one-based. The alpha byte is not used in determining color; some modules use it to convey other information, such as opacity.

Volume Data

ConvexAVS represents volumes as 3D scalar fields of bytes declared as "field 3D scalar byte." The value of each byte is between 0 and 255 inclusive. Some modules use the field data as indexes to colormaps. For many ConvexAVS modules that deal with volumes, each dimension of the field must be less than 256.

Colormap

A *colormap* is a transfer function that assigns a color to each integer between an upper and a lower bound. A colormap consists of four arrays of floating-point values, one each for:

- Hue.
- Saturation.
- Value.
- Opacity.

Each value is between 0.0 and 1.0 inclusive. A colormap also has an integer size or number of colors, which is the length of each of the four arrays. A colormap has floating-point lower and upper bounds that determine the resolution of the colormap. The lower bound is an index that maps to the first element of each array. The upper bound is an index that maps to the last element in each array.

In C, a colormap is represented by an AVScolormap structure defined in the colormap.h include file as:

```
typedef struct {
 int size;       /*number of entries in each array*/
 float lower;   /*0th entry maps to this value*/
 float upper;   /*size-th entry maps to this value*/
 float *hue;
 float *saturation;
 float *value;
 float *alpha;
} AVScolormap;
```

A C routine declares a colormap input argument as AVScolormap * and a colormap output argument as AVScolormap **. A FORTRAN computation routine can input a colormap by declaring a series of parameters:

```
INTEGER FUNCTION my_module(size, lower, upper, hue,
                                sat, val, alpha)
 INTEGER size
 REAL lower, upper
 REAL hue(size), sat(size), val(size), alpha(size)
```

A FORTRAN routine can output a colormap as:

```
INTEGER FUNCTION my_module(size, lower, upper,
                    phue, psat, pval, palpha)
 INTEGER size
 INTEGER phue
 REAL lower, upper
 REAL hue(size), sat(size), val(size), alpha(size)
 phue = MALLOC(4*size)
```

Geometry

A *geometry object* describes changes to the geometry of a particular scene that is represented by a module input or output. ConvexAVS allows your module to create geometry objects as outputs. It is possible for a module to use geometry objects as inputs, but ConvexAVS V1.0 does not support this. Geometry output is used as input to a ConvexAVS-supplied renderer module.

What Is an Edit List?

ConvexAVS geometry is based upon the concept of the *edit list.* An edit list is a sequence of geometry instructions that are packed together and sent across a red geometry connection to a render module. The render module unpacks the edit list and sequentially executes the commands it contains.

The geometry commands from the edit list are used to build an internal geometry database. This database corresponds to the graphics that are displayed on the screen.

When the render module unpacks a geometry command from the edit list, it checks to see if it has already encountered items referenced by this command. If it has, then it does not recreate the geometry but rather "edits" the item's entry in the geometry database. This means that a user module does not need to send a completely new geometry each time a parameter changes. Instead, it only has to send commands referencing specific geometry items to be modified.

Why Use Edit Lists?

Edit lists provide a *protocol* for user modules to encapsulate geometry commands and pass them on to a general renderer. They provide a separation between modules that produce geometry and those that render it.

Edit lists reduce traffic between geometry producing modules and render modules. This is because once a complete geometry has been created, shorter "edits" can be sent to modify specific portions of the representation.

Render modules can receive input from multiple geometry producing modules. The render modules are a central location where modules can mix and modify each other's geometry. They also provide a central user interface for geometric data.

How an Edit List Is Used

Currently, ConvexAVS has three geometry rendering modules; render manager, render geometry, and display geometry GL. They provide a conduit for geometry to get from a module to the internal geometry database. The *render modules* unpack the edit lists they receive and rebuild the geometry database according to the instructions contained in the edit list.

The manner in which the database is displayed is determined in two places. The first is in the program that produces the geometry. The second is in the Geometry Viewer panel where you manipulate display parameters.

The Geometry Viewer is the interface for the internal geometry database. It provides a tool for you to view and modify representations of geometry contained in the database.

Figure 8-6 shows an edit list data flow.

Design Factors

All geometry commands must be stored in an edit list for them to be passed through the red geometry connection and executed by the renderer. Once a geometry is sent to the renderer, it stays in the database until it is explicitly deleted. The deletion can occur either from the Geometry Viewer or with a command from a module.

Geometry items are referenced in the database by unique strings. This means that a module needs to remember the name under which the geometry was created to reference it later.

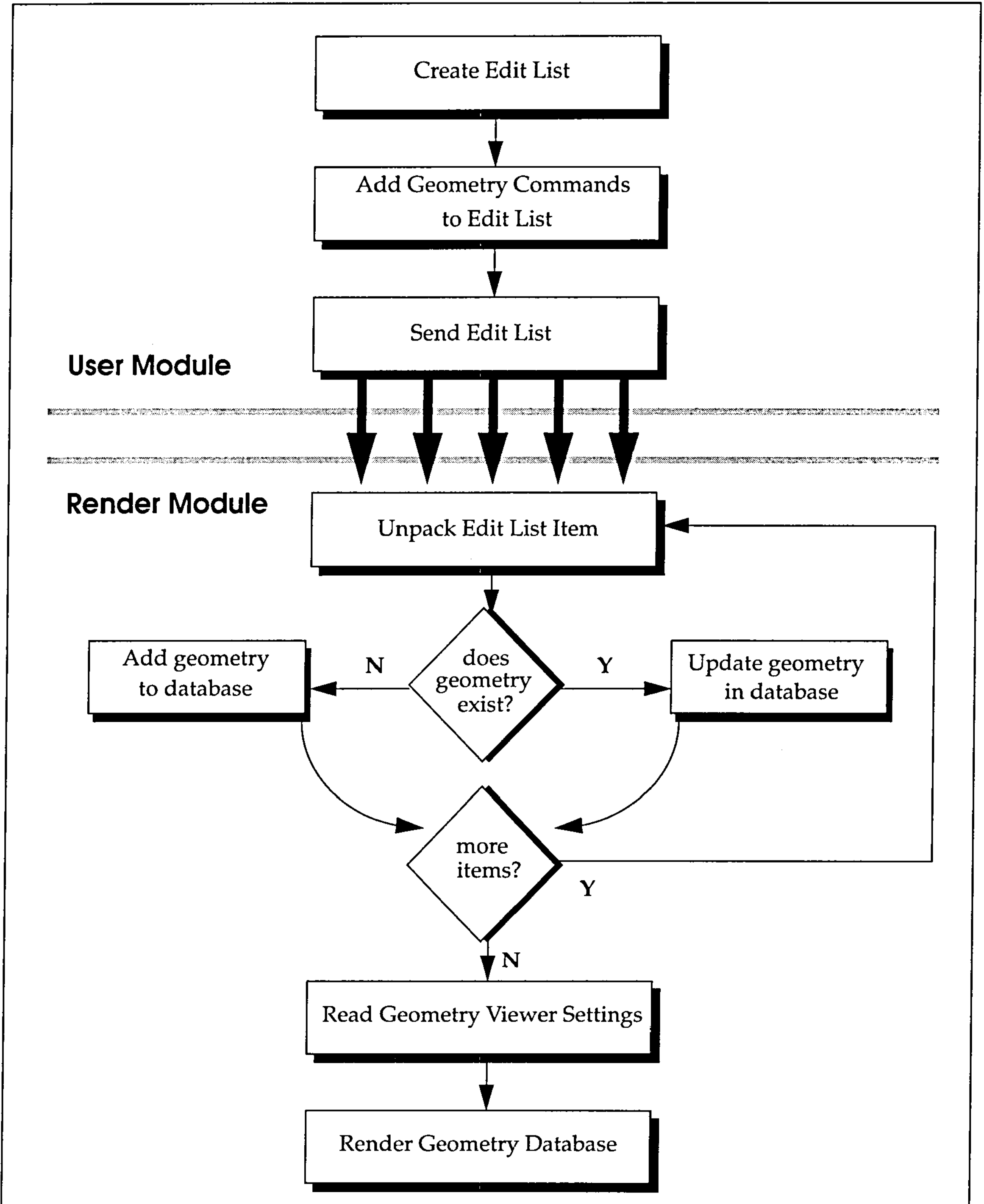
Adding a new item to the geometry database uses more internal resources than changing an attribute of an already existent one. The most efficient way to program ConvexAVS is to create a complete geometry once, then modify it later.

ConvexAVS spends a significant amount of time unpacking edit lists. You can reduce this time by using as few GEOM calls as possible when building a geometric description.

Performance can be improved by storing created edit lists and resending them later. This is only useful if you want to send the same geometry commands more than once.

There is currently no mechanism for a module to get the status of the Geometry Viewer. Therefore, geometry programs should be designed to be independent of the state of the renderer.

Figure 8-6
Diagram of an Edit List Data Flow



Manipulating Edit Lists

Each time a module is invoked, it starts with an empty edit list. It places changes into the edit list that it wants to be made for this invocation. In creating and using edit lists, geometry objects, and light sources, a module uses routines in the geometry library. Refer to Appendix G, "Geometry Routines." A module uses the following steps in preparing an edit list for output:

1. Initialize the edit list using GEOMinit_edit_list in C or GEOM_INIT_EDIT_LIST in FORTRAN. This creates a new list or empties an existing list.
2. Create and modify geometry objects, cameras, or light sources using routines in the geom library.
3. Modify the edit list using routines whose names begin with GEOMedit in C or GEOM_EDIT in FORTRAN.
4. For a coroutine module, use AVScorout_output to output the list. Then use GEOMdestroy_edit_list in C or GEOM_DESTROY_EDIT_LIST in FORTRAN to deallocate the list.

A module must deallocate an existing edit list before reusing the list. For a subroutine module, the edit list passed to the module as an output argument is the edit list the module created on its last execution. The module must deallocate this list at the start of each invocation of the module, normally by calling the GEOMinit_edit_list routine in C or GEOM_INIT_EDIT_LIST in FORTRAN before modifying the list. A C language example is:

```
/* C */
my_module(output)
GEOMedit_list *output;
{
 /*
 * Deallocate edit list from last invocation;
 * initialize edit list for this invocation.
 */
 *output = GEOMinit_edit_list(*output);
 < rest of module >
}
```

A FORTRAN example is:

```
C FORTRAN
 FUNCTION MY_MODULE(OUTPUT)
 EXTERNAL GEOM_INIT_EDIT_LIST
 INTEGER OUTPUT, GEOM_INIT_EDIT_LIST
 OUTPUT = GEOM_INIT_EDIT_LIST(OUTPUT)
 < rest of module >
```

A coroutine module can use GEOMdestroy_edit_list in C or GEOM_DESTROY_EDIT_LIST in FORTRAN to deallocate a list after calling AVScorout_output. A C language example is:

```
/* C */
 ...
 GEOMedit_list output;
 < generate edit list "output" >
 AVScorout_output(output);
 GEOMdestroy_edit_list(output);
```

A FORTRAN example is:

```
C FORTRAN
 ...
 INTEGER OUTPUT
 < generate edit list "OUTPUT" >
 CALL AVSCOROUT_OUTPUT(OUTPUT)
 CALL GEOM_DESTROY_EDIT_LIST(OUTPUT)
```

Pixel Map

A *pixel map* is a data structure that incorporates a reference to an X Window System *pixmap*. An X-pixmap is an array of pixel values that can be a destination for a rendered image. It resides in the X server. (In contrast, an image is a data structure that includes an array of pixel values and resides in client memory.)

A pixel map includes an Xlib Pixmap ID, the Xlib Window ID of the window associated with the pixmap, the Window ID of that window's parent window, and a reserved flag.

In C, a pixel map is defined as an AVSpixdata data type. A pixel map input argument is declared as AVSpixdata *, and a pixel map output argument is declared as AVSpixdata **. AVSpixdata is a structure defined in the avs_pixdata.h include file with the following components:

```
typedef struct _AVSpixdata {
 int parent;
 int window;
 int pixmap;
 int is_buffer; /* reserved; must be set to 0 */
} AVSpixdata;
```

A FORTRAN computation routine cannot take a pixel map as an argument.

Modules

9

Components

A module is a fundamental building block in a ConvexAVS network and could have one of the following purposes:

- To import data from outside ConvexAVS (or generate its own data) and convert it into a ConvexAVS data type.
- To transform ConvexAVS data and produce output data of the same or of a different ConvexAVS type.
- To render or store ConvexAVS data on an external device, such as the display screen or a file.

Name

The name of a module is a string that identifies the module. The name appears on the module icon in the module palette and workspace.

Type

A module is of one of four types, depending on its function: data, filter, mapper, and renderer. These module type distinctions affect only the presentation of the module in the ConvexAVS user interface. Module type determines in which menu the module icon appears in the module palette.

Data

A module that generates data or imports data from outside ConvexAVS and converts it into a ConvexAVS data type.

Filter

A module that transforms ConvexAVS data and produces output data of the same or of a different ConvexAVS type.

Mapper

A module that converts ConvexAVS data into a different type that is more complex than a filter's output (for example, a geometry data type).

Renderer

A module that renders or stores ConvexAVS data on an external device, such as the display screen or a file.

Ports

A module may have zero or more *input ports* and zero or more *output ports*. A port is a channel through which data passes to or from other modules. Each port has a name and a ConvexAVS data type. An input port is represented in the Network Editor by a colored bar at the top of the module icon, and an output port is represented by a colored bar at the bottom of the icon. The color of each bar indicates the port's data type:

Color	Data Type
red	geometry
yellow	colormap
light blue	pixmap
multi-color	field

Data modules usually read or generate their own input data and do not have input ports. Renderer modules often display or write their own output data and, therefore, do not have output ports.

When a module exists in ConvexAVS, each input port can be connected to an appropriate output port of another module, and each output port can be connected to an appropriate input port of another module. A pair of ports can be connected only when the data types of the ports match. The data types match when they are the same or when one is a subtype of the other. For example, a port declared to be of type "field" matches a port of type "field 2D," but a port of type "field 2D" does not match a port of type "field 3D." An output port cannot be connected to an input port of the same module.

For some input ports, a connection to an output port of another module is required before the module can be invoked. For other input ports, a connection is optional.

Parameters

A *parameter* has a name, a type, and an initial value. Some parameters also have bounding information, such as a range of allowed values. Parameter types include most primitive ConvexAVS data types along with constrained variants such as "boolean" and "choice." For example, a module that scales a field by a value between 0.0 and 1.0 would use AVSadd_parameter to add a "float" parameter input. Refer to AVSadd_parameter in Appendix E, "Module Routines."

Every parameter is connected to a *widget* that enables you to change the value of the parameter between module invocations. A widget is a virtual input device, such as a dial or a file browser. A parameter can be connected only to a widget that is compatible with the parameter's type. Each parameter type has a default widget type, but the module can override the default and attach a parameter to another compatible widget. For example, the module mentioned above could use AVSconnect_widget to generate a floating point value by attaching a slider or dial widget to the "float" parameter. Refer to AVSconnect_widget in Appendix E, "Module Routines."

A parameter can also have properties. A *property* usually determines some aspect of how the associated widget presents the parameter. By setting properties on a parameter, a module can customize how the user interface handles the parameter. Each property is meaningful only with certain widgets. For example, you could give a slider a title called "scale value" if the number produced by the slider represents a scale value. For a description of available properties, refer to AVSadd_parameter_prop in Appendix E, "Module Routines."

When appropriate, a module can alter the current value or bounds of a parameter dynamically. ConvexAVS then updates any widget associated with the parameter. Refer to AVSmodify_parameter in Appendix E, "Module Routines."

Functions

Each module has one or more of the following functions:

- *Description* (required for all modules). ConvexAVS invokes this procedure when it first learns that a module is available and again when you make an instance of it, for example, move the module icon from the Network Editor module palette to the workspace.
- *Computation* (only for subroutines). ConvexAVS invokes this procedure when the flow executive is active and the module's input data or parameters have changed. The arguments to the computation function correspond to module input ports, output ports, and parameters. This function does the computational work of the module, using input data and parameters to produce output data. A coroutine module's main program determines when to perform its computation.
- *Initialization* (optional). ConvexAVS invokes this procedure when you make an instance of the module. The initialization function may take such actions as allocating memory or creating a window but has no arguments and returns no meaningful values.
- *Destruction* (optional). ConvexAVS invokes this procedure when you destroy the module, for example, by moving the module icon from the Network Editor workspace to the hammer icon. The destruction function may take such actions as freeing memory or destroying a window but has no arguments and returns no meaningful values.

Description Function

The description function describes the module's name, type, inputs, outputs, and parameters using a set of library functions. A C language file can contain more than one module and therefore more than one description function. The file must contain a routine called AVSinit_modules that refers to all the description functions in the file. A FORTRAN file can contain only one module and, therefore, only one description function. A FORTRAN description function must be named AVSINIT_MODULES. The description function has no arguments and returns no value.

Figure 9-1 is the C language version of a description function for a module that computes the threshold of a 3-dimensional scalar field. The threshold module is created with one input port, one output port, and two parameters.

Figure 9-2 is the FORTRAN version of the same routine.

Figure 9-1
C Language Description Function

```
threshold()
{
 int thresh_compute();
 int in_port, out_port;

 AVSset_module_name("threshold", MODULE_FILTER);
 in_port = AVScreate_input_port("Input Field", "field 3D scalar",
                              REQUIRED);
 out_port = AVScreate_output_port("Output Field", "field 3D scalar");
 AVSinitialize_output(in_port, out_port);
 AVSadd_float_parameter("thresh_min", 0.0, FLOAT_UNBOUND,
                              FLOAT_UNBOUND);
 AVSadd_float_parameter("thresh_max", 255.0, FLOAT_UNBOUND,
                              FLOAT_UNBOUND);
 AVSset_compute_proc(thresh_compute);
}
```

Figure 9-2
FORTRAN Language Description Function

```
 SUBROUTINE AVSINIT_MODULES
#include <avs/avs.inc>
 EXTERNAL AVSCREATE_INPUT_PORT, AVSCREATE_OUTPUT_PORT
 INTEGER IN_PORT, AVSCREATE_INPUT_PORT
 INTEGER OUT_PORT, AVSCREATE_OUTPUT_PORT
 EXTERNAL THRESH_COMPUTE
 CALL AVSSET_MODULE_NAME('threshold', 'filter')
 IN_PORT = AVSCREATE_INPUT_PORT('Input Field',
 + 'field 3D scalar', REQUIRED)
 OUT_PORT = AVSCREATE_OUTPUT_PORT('Output Field',
 + 'field 3D scalar')
 CALL AVSINITIALIZE_OUTPUT(IN_PORT, OUT_PORT)
 CALL AVSADD_PARAMETER('thresh_min', 'real', 0.0,
 + FLOAT_UNBOUND, FLOAT_UNBOUND)
 CALL AVSADD_PARAMETER('thresh_max', 'real', 255.0,
 + FLOAT_UNBOUND, FLOAT_UNBOUND)
 CALL AVSSET_COMPUTE_PROC(THRESH_COMPUTE)
 RETURN
 END
```

The description function:

- Uses AVSset_module_name to set the module name and type. A description function must call this routine.
- Uses AVScreate_input_port and AVScreate_output_port to create the input and output ports. A description function may

have zero or more calls to each of these routines, depending on how many input and output ports it has. Each routine returns an integer port identifier for use as an argument to other routines, such as AVSinitialize_output.

- Creates parameters using AVSadd_parameter or AVSadd_float_parameter. A description function may have zero or more calls to each of these routines, depending on how many parameters it has. Each routine returns an integer parameter identifier for use as an argument to other routines, such as AVSconnect_widget.
- Uses AVSset_compute_proc to set the computation function. A description function for a subroutine module must call this routine. A description function for a coroutine module does not call this routine.

A description function can also:

- Use the AVSinitialize_output routine to allocate memory for output data before invoking the module computation function. This routine pairs an output port with an input port. Before invoking the module computation function, ConvexAVS frees data at the output port and allocates a new data structure of the same size and dimensions as the data at the input port. This frees the computation routine from the necessity of allocating memory for the data structure.
- Use the AVSautofree_output routine to free memory allocated for output data before invoking the module computation function. By default, ConvexAVS does not free the memory allocated for output data during the previous invocation of the module computation function.
- Set an initialization function using the AVSset_init_proc routine.
- Set a destruction function using the AVSset_destroy_proc routine.
- Use the AVSconnect_widget routine to declare a preference that a parameter be attached to a widget of a given type. Each type of parameter is associated with a default widget type. This routine allows the module to override the default.

  For example, a module can use a parameter of type "string" for a file path name. The default widget for a string parameter is a text type-in. The module description function can use AVSconnect_widget to connect the parameter to a file browser. A C language example follows:

```
int p;
p = AVSadd_parameter ("Data File", "string",
                      "/mydata", "", "");
AVSconnect_widget (p, "browser");
```

The FORTRAN version is:

```
EXTERNAL AVSADD_PARAMETER
INTEGER P, AVSADD_PARAMETER
P = AVSADD_PARAMETER ('Data File', 'string',
                      '/mydata', '', '')
CALL AVSCONNECT_WIDGET (P, 'browser')
```

- Use the AVSadd_parameter_prop routine to add a property to a parameter. By calling this routine, a module can customize how the user interface handles the parameter.

Computation Function

Each subroutine module must have a computation function in addition to a description function. ConvexAVS invokes the computation function when the flow executive is active and the module's inputs or parameters change.

The computation function can have any name. The module identifies the computation function to ConvexAVS by calling the AVSset_compute_proc routine in the description function. To return an integer, declare the computation function. It should return a value of 0 to indicate an error. In this case, the flow executive does not invoke any other modules whose inputs depend on outputs from the erring module.

Arguments to the computation function correspond to the module's inputs, outputs, and parameters. A C language computation function has one argument for each input port, output port, and parameter declared in the description function. In the parameter list, all input ports are represented first, then all output ports, then all parameters. Within each category, the arguments appear in the order in which ports or parameters are declared in the description function.

For a FORTRAN computation function, the arguments are presented in the same order as the arguments to a C language computation function. However, each port or parameter can generate more than one argument to the computation routine. The number of arguments for each port or parameter depends on the data type declared in the description function and, for a port, on whether the port is input or output. For example, an input port declared as "field 3D scalar uniform" in the description function generates four arguments to a FORTRAN computation routine. For more information about arguments to FORTRAN computation functions, refer to Chapter 8, "Data Types."

For a C language computation function, an argument that represents an input port or a parameter is usually passed as a pointer to an object of the C storage type that corresponds to the data type of the port or parameter declared in the description function. An argument that represents an output port is usually passed as a pointer to a pointer to an object of the appropriate data type. This double indirection is provided to allow the computation routine to allocate memory for the output data. For example, a C language computation function declares an input field argument as AVSfield * and an output field argument as AVSfield **. Arguments that represent ports or parameters of some data types, such as integers, are passed as those objects.

Because FORTRAN arguments are passed by reference, a FORTRAN computation routine usually declares an argument to be of the FORTRAN type that corresponds to the data type of the port or parameter. For example, an argument that represents a floating-point input port, output port, or parameter is declared to be of type REAL.

The computation routine usually performs some operations on the input data and parameters to produce output data. By default, the computation function is responsible for freeing memory allocated for output data on previous invocations of the module and for allocating memory for output data on the current invocation. The module can use the AVSinitialize_output and AVSautofree_output routines in the description function to eliminate the need for some of this memory management.

Subroutines and Coroutines

ConvexAVS has two types of modules:

- Synchronous *subroutines*.
- Asynchronous *coroutines*.

The difference between these modules is the way they interact with ConvexAVS to do their computational work. A subroutine module does its computation and sends output automatically, for example, when the module's input or parameters change. A coroutine module has control over its computation and when to send output.

Subroutines are used in the demand-driven portions of a network where a module needs to compute only when input data or a parameter changes. A coroutine usually performs a number of independent computations, each of which represents one iteration of a series, and sends output to ConvexAVS after each iteration. For example, the particle advector module is a coroutine.

Subroutine Modules

A basic subroutine module consists of a description function and a computation function with optional initialization and destruction functions. You do not supply a main program. Instead, the ConvexAVS library supplies the main program for a module's executable file.

A C language executable file may contain more than one module, including description and computation functions for each module, but it has only one main program. In addition to the description and computation functions, you supply a function called AVSinit_modules to invoke the description functions for all modules in the file. This routine takes no arguments and returns no meaningful value. It must make one call to AVSmodule_from_desc for each module in the file. The AVSinit_modules routine can call AVSmodule_from_desc either:

- Directly for each module in the file.
- Indirectly through a single call to AVSinit_from_module_list for a list of modules.

AVSmodule_from_desc invokes the given module's description function. Following is a simple example of an AVSinit_modules routine for a file that contains a single threshold module.

```
AVSinit_modules()
{
 /* threshold is the module description function */
 int threshold();
 /* this invokes the threshold routine */
 AVSmodule_from_desc(threshold);
}
```

Following is an example of an AVSinit_modules routine for a file that contains more than one module.

```
int ((*mod_list[])()) = {
 module_1_desc,
 module_2_desc,
 module_3_desc
};

#define NMODS (sizeof(mod_list) / sizeof(char *))

AVSinit_modules()
{
 AVSinit_from_module_list(mod_list, NMODS);
}
```

A FORTRAN executable file has only one module and one main program. A FORTRAN module does not have a separate AVSINIT_MODULES function. Instead, its description function is named AVSINIT_MODULES.

Coroutine Modules

A basic coroutine module consists of a main program and a description function with optional initialization and destruction functions. Each executable file can contain only one module. The description function can have any name.

Module Examples

Appendix H, "Module Code Examples," contains source code examples for:

- C language subroutine and coroutine.
- FORTRAN subroutine.

The s ource code for these and other examples is available in the /usr/avs/examples directory.

Handling Errors

ConvexAVS provides a mechanism for modules to report errors. The AVSmessage routine presents information about the module and function sending the message.

ConvexAVS treats error reports differently depending on their severity. The severity that the module declares determines how ConvexAVS presents the message to you and whether or not you must acknowledge the message before ConvexAVS can continue. If the message appears in a dialog box, the border of the dialog box is color coded to indicate the severity. Following are the possible levels of severity in increasing order:

AVS_Information The message does not indicate an error. The message is written to stderr, and ConvexAVS continues executing. No choices are presented.

AVS_Debug The message does not indicate an error. It conveys information during module testing. The message is written to stderr, and ConvexAVS continues executing. No choices are presented.

AVS_Warning	The message indicates a problem that is not fatal to module execution. The message and choices are presented in a dialog box with a yellow border. You must make a choice before ConvexAVS can continue.
AVS_Error	The message indicates a serious problem that may cause the module to produce erroneous results but is not permanently fatal to module execution. The message and choices are presented in a dialog box with a red border. You must make a choice before ConvexAVS can continue.
AVS_Fatal	The message indicates a problem that is permanently fatal to module execution. The message and choices are presented in a dialog box with a black border. You must make a choice before ConvexAVS can continue. The module is marked as dead, and the module icon in the Network Editor workspace turns black. The flow executive no longer executes the module.

When a subroutine module computation function encounters an error that produces erroneous output, the computation function should return a value of 0. A coroutine module should not call AVScorout_output. The flow executive does not execute downstream modules that depend on output from the module that encounters the error.

If a module encounters an error likely to be fatal, such as a failure to allocate memory, it usually should not terminate its process by calling `exit`(2). Instead, it should call AVSmessage with a severity of AVS_Fatal. A subroutine computation function should then return a value of 0. A coroutine module should call AVScorout_wait and should not call AVScorout_input or AVScorout_output again.

If a module exits or dies unexpectedly and ConvexAVS tries to communicate with that module, ConvexAVS automatically generates a fatal error message.

ConvexAVS provides simple interfaces to AVSmessage for reporting errors of a given severity. These routines are called AVSinformation, AVSdebug, AVSwarning, AVSerror, and AVSfatal. Refer to Appendix E, "Module Routines."

Creating Online Help

You can supplement the online help facility with documentation for your own modules and networks. You can create a series of help files and have them accessible through the **Help** buttons and the **Show Module Documentation** button in the Module Editor window. You can also arrange for your help files to be visible to the `man(1)` shell command.

For details on this, refer to Appendix I, "Creating Online Help for Your Modules."

Selective Computation

When a module has more than one input port or parameter, it is likely that when the module computation function is executed, some ports or parameters have not changed since the previous execution of the computation function. The module might be able to avoid some computation for ports or parameters that have not changed.

ConvexAVS provides two routines, AVSinput_changed and AVSparameter_changed, to determine whether a given input port or parameter has changed since the previous invocation of the computation function. These routines return 1 if the input or parameter has changed and 0 if it has not. For a coroutine module, these routines determine whether the input or parameter has changed since the previous call to AVScorout_input.

When a module has more than one output port, it is possible that after the module computation function is executed some ports have not changed since the previous execution of the computation function. By default, ConvexAVS assumes that all output ports have changed after each invocation of a module computation function. This can cause ConvexAVS to invoke downstream modules whose input depends on the output of the current module, even if some output ports have not changed.

ConvexAVS provides AVSmark_output_unchanged to declare that a given output port has not changed since the previous invocation of the computation function. For a coroutine module, this routine declares that the output port has not changed since the previous call to AVScorout_output.

Building and Linking Modules

Each ConvexAVS module is a program that resides in a single executable file. That file can contain more than one C language subroutine module. The source code can be in either C or FORTRAN. The routines that you provide depend on the source language and whether the module is a subroutine or a coroutine.

Include Files

ConvexAVS supplies a number of *include files* for both C language and FORTRAN programs. Some include files are needed for nearly all modules, while others are needed only if the module uses data of a particular type. Appendix E, "Module Routines," lists any include files needed for each ConvexAVS routine.

ConvexAVS include files are in the /usr/avs/include directory. The file /usr/include/avs is a link to this directory so that both C language and FORTRAN programs can refer to an include file using the following syntax:

C:	`#include` <avs/*filename*>
FORTRAN:	`#include "avs/`*filename*`"`

C Language Include Files

Most C language modules should include a single file, avs.h. This file contains definitions not specific to particular data types. The following files are needed when a module uses data of specific types:

avs_pixdata.h	Definitions for pixel maps.
colormap.h	Definitions for colormaps.
field.h	Definitions for fields.
geom.h	Definitions for geometries.

FORTRAN Include Files

Most FORTRAN modules should include a single file, avs.inc. This file contains definitions not specific to particular data types as well as definitions needed when using data of most ConvexAVS types. FORTRAN modules that use geometries should include the file geom.inc.

Compiling and Linking Modules

To compile and link a module, use cc(1) for C language modules and fc(1) for FORTRAN modules. ConvexAVS supplies four basic module libraries in the /usr/avs/lib directory. Each module must be linked with one of these libraries. The library to use depends on the source language and whether the module is a subroutine or a coroutine. Refer to Table 9-1.

Table 9-1
Module Libraries

Module Type	Language	Library
Subroutine	C	libflow_c.a
Subroutine	FORTRAN	libflow_f.a
Coroutine	C	libsim_c.a
Coroutine	FORTRAN	libsim_f.a

Modules need IEEE floats, either the -fi flag when you compile or having IEEE as your machine default. Modules also need to be linked with the libraries libgeom.a and libutil.a. An example for a C subroutine would be:

```
% cc -o threshold threshold.o -L/usr/avs/lib -lflow_c
                                    -lgeom -lutil
```

Converting Applications into Modules

Many existing simulations, batch data converters, and other scientific applications can be converted into modules.

Coroutine Modules

Following are some of the essential steps in converting an application into a coroutine module:

- Determine what data the application needs to obtain from ConvexAVS as inputs or parameters and what data it needs to send to ConvexAVS as outputs.
- Choose the ConvexAVS data type that is most appropriate for each input, output, and parameter.
- Write a description function to declare the module and its inputs, outputs, and parameters.
- In the application's main program, insert a call to AVScorout_init and calls to other coroutine functions such as AVScorout_input, AVScorout_output, and AVScorout_wait, as appropriate.

- Convert the program's data structures to the corresponding ConvexAVS data types for inputs, outputs, and parameters.
- Ensure that the program allocates and frees memory for ConvexAVS outputs where necessary. Using AVSinitialize_output and AVSautofree_output make this task easier.
- Use AVSmessage or its variants to handle errors in the program.
- Ensure that the program uses appropriate ConvexAVS include files. Most C language programs should include avs.h and any files needed for particular data types. Most FORTRAN programs should include avs.inc.
- Compile and link the program with the ConvexAVS coroutine module archive library that is appropriate for the program's source language.

Subroutine Modules

Converting an existing application into a subroutine module is similar to a coroutine module, with these differences:

- Convert the application's main program to a computation function. A subroutine module does not supply its own main program.
- Ensure that the computation function returns 1 if successful and 0 if unsuccessful.
- Do not insert calls to coroutine functions. Instead, ensure that arguments to the computation function are the module inputs, outputs, and parameters.
- For a C language subroutine module, supply an AVSinit_modules routine. For a FORTRAN subroutine module, name the description function AVSINIT_MODULES.
- Compile and link the module with the ConvexAVS subroutine module archive library that is appropriate for the module's source language. ConvexAVS has different archive libraries for subroutine and coroutine modules.
- Call AVSset_compute_proc in description function.

Module Internals

This section presents information about the internal workings of ConvexAVS modules.

Subroutine Modules

ConvexAVS invokes the module's main program twice:

- When you read the module into ConvexAVS.
- When you make an instance of the module, for example, by moving the module icon from the Network Editor palette to the workspace.

In both cases, ConvexAVS creates a new process and invokes the module executable file in that process.

When ConvexAVS invokes the module's main program the first time, it does so for identification. The module's main program then performs the following:

- Sets up a connection to ConvexAVS.
- Invokes the AVSinit_modules routine. This routine in turn invokes the description functions of all modules in the executable file.
- Conveys to ConvexAVS the module declarations for all modules in the executable file.
- Terminates the module's process.

When ConvexAVS receives the module declarations, it adds the module icons to the Network Editor palette.

When ConvexAVS invokes the module's main program a second time, it does so for instantiation. The module's main program then does the following:

- Sets up a connection to ConvexAVS.
- Invokes the AVSinit_modules routine. This routine, in turn, invokes description functions of all modules in the executable file.
- Conveys to ConvexAVS module declarations for all modules in the executable file.
- Sets up an instance of the module that can receive data from and send data to ConvexAVS.
- Invokes the module initialization function, if any.
- Enters a server routine that loops indefinitely waiting for remote procedure calls from ConvexAVS, then executing the requests.

When the flow executive is active, ConvexAVS issues a remote procedure call when any of the module's input ports or parameters change. When the module's server routine receives a computation request, it reads the module's inputs and parameters from ConvexAVS, invokes the module's computation function, and conveys the module's outputs to ConvexAVS. If another module's input port is connected to the current module's output port, ConvexAVS marks the other module's input port as having changed data. This may cause ConvexAVS to send a remote procedure call to the second module.

ConvexAVS may issue remote procedure calls other than computation requests during the lifetime of the module. For example, you may destroy the module by dragging the module icon to the hammer icon. ConvexAVS then issues a remote procedure call that causes the module server routine to invoke the module's destruction function, if any, and then terminate the module's process. The module's computation function may also issue callbacks to ConvexAVS, for example, when reporting errors via the AVSmessage routine.

Coroutine Modules

Similar to subroutine modules, ConvexAVS invokes the coroutine module's main program twice: once when you read the module into ConvexAVS and once when you make an instance of the module. In both cases, ConvexAVS creates a new process and invokes the module executable file in that process.

When ConvexAVS invokes the module's main program the first time, it does so for identification. Because ConvexAVS does not supply the main program, you are responsible for ensuring that the main program responds properly to this invocation. The main program must call the AVScorout_init routine before attempting to do any computation. The AVScorout_init routine performs the following during the identification phase:

- Sets up a connection to ConvexAVS.
- Invokes the module's description function.
- Conveys to ConvexAVS the module declarations for the module.
- Terminates the module's process.

When ConvexAVS receives the module declarations, it adds the module icon to the Network Editor palette.

When ConvexAVS invokes the module's main program a second time, it does so for instantiation. When the main program invokes AVScorout_init during the instantiation phase, that routine performs the following:

- Sets up a connection to ConvexAVS.
- Invokes the module's description function.
- Conveys to ConvexAVS module declarations for the module.
- Sets up an instance of the module that can receive data from and send data to ConvexAVS.
- Invokes the module initialization function, if any.
- Return.

The main program can then interact with ConvexAVS at any time it wants. For example, the main program can behave like a subroutine module by looping indefinitely, taking the following steps on each iteration:

- Call the AVScorout_wait routine. This routine waits until one of the module's inputs or parameters changes, then returns.
- Call the AVScorout_input routine. This routine obtains the module's inputs and parameters from ConvexAVS.
- Perform the module's computation.
- Call the AVScorout_output routine. This routine conveys the module's outputs to ConvexAVS.

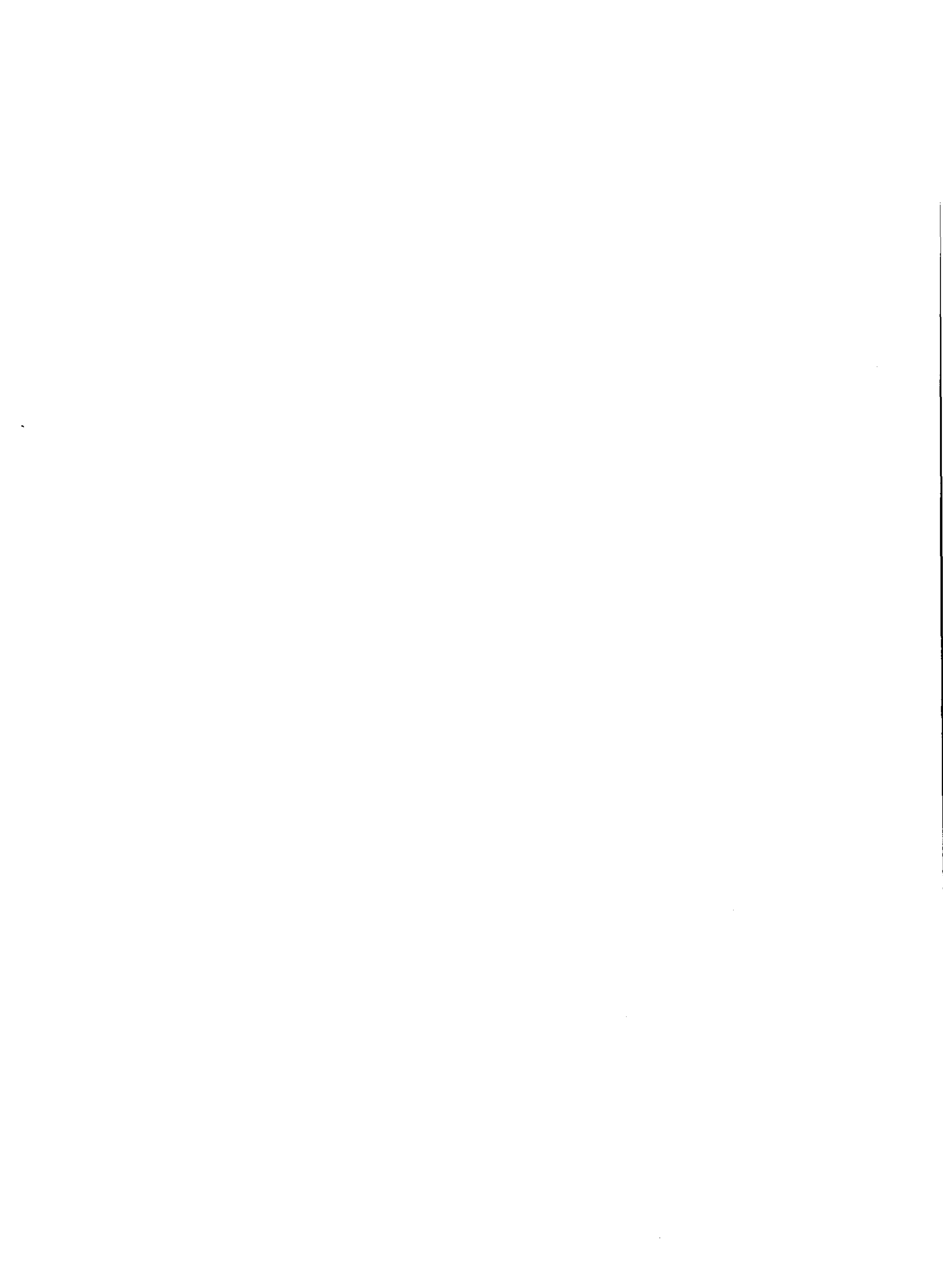
A coroutine module performs a series of independent computations, sending output to ConvexAVS after each iteration. The main program can accomplish this by using the loop described above, except that in order to compute continuously, it omits the call to AVScorout_wait. If a coroutine module computes continuously, it should provide a parameter that allows you to stop the computation. The module should check the value of this parameter after the call to AVScorout_input.

After calling AVScorout_init, a coroutine module might ensure that input is available before beginning computation. It can do this in a loop, calling AVScorout_wait, then AVScorout_input until the input data is not null.

A coroutine module can also use the AVScorout_exec routine. This routine waits until the flow executive has stopped running, then returns. This allows the module to ensure that the network has processed the output of each computational iteration before sending more output so that no data is lost.

Module Reference

This part provides man pages for each of the modules. These pages are arranged alphabetically for ease of reference, and they contain detailed information that is not covered in other parts of this book.



NAME

avs_modules – list of ConvexAVS module manual pages

DESCRIPTION

This section includes a manual page for each module in the ConvexAVS distribution. The manual pages are also available on the system; you can view them either within ConvexAVS itself or from a shell.

Within ConvexAVS

Click on the small square in any module icon to open its Module Editor window. Then click the **Show Module Documentation** button.

From a shell

Enter a command in the form:

% man -S avs *module_name*

Note that in *module_name*, you should replace any SPACE characters that appear on the module icon with underscore characters. For instance, to display the manual page for the field to mesh module, enter this command:

% man -S avs field_to_mesh

MODULE LISTING

The supported modules included with ConvexAVS V1.0 are:

arbitrary slicer	map 3D scalar field to arbitrary plane
bubbleviz	generate spheres to represent values of 3D field
clamp	restrict values in data field
colorizer	convert field of data values to color values
colormap manager	share colormaps among subnetworks
combine scalars	combine scalar fields into a vector field
compute gradient	compute gradient vectors for 2D or 3D data set
contrast	perform linear transformation on range of field values
crop	extract range of elements from a field
density PLOT3D	strip out the plot3d density
display geometry GL	display geometric image on Silicon Graphics Iris
display image	show image in a display window
display image GL	show image on Silicon Graphics Iris
display pixmap	show pixmap in a display window
dot surface	generate points that define an isosurface
downsize	reduce size of data set by sampling
export_PLOT3D	convert plot3d files into formatted form
extract scalar	extract scalar field(s) from a vector field
field to byte	transform any field to a byte-valued field
field to double	transform any field to a field of double-precision floating point values
field to float	transform any field to a field of single-precision floating point values

field to int	transform any field to an integer-valued field
field to mesh	transform a 2D scalar field to a surface in 3D space
flip normal	change direction of each vertex normal
generate colormap	output colormap
geom to scatter	convert geometry to point list
gradient shade	apply lighting and shading to colored data set
hedgehog	show vectors on a slice of a 3D 3-vector field
histogram stretch	balance the histogram of a data set
image manager	share images among subnetworks
image to pixmap	convert image to pixmap
import_PLOT3D	convert plot3d files into unformatted form (refer to export_PLOT3D)
interpolate	change the size of the data
isosurface	generate an isosurface for a volume of data
mirror	reverse array indices in a 2D or 3D data set
momentum PLOT3D	strip out the plot3d momentum vector
offset	translate vertices along vertex normals
orthogonal slicer	slice through volume data with plane perpendicular to coordinate axis
output postscript	convert pixmap to PostScript and store in file
particle advector	release grid of particles into velocity field
pdb to geom	create molecule geometry from PDB file
pdb viewer	view and manipulate information in Brookhaven PDB format
pixmap to image	transform pixmap to image
read HDF volume	read volume file in HDF format from disk into a field
read PLOT3D	read plot3d files
read RLE image	read image file in RLE format from disk into a field
read field	read field from a disk file
read geom	reads a data file containing a geometry
read image	read image file from disk into a field
read volume	read volume file from disk into a field
render geometry	convert geometric description to image (Geometry Viewer)
render manager	share geometries among subnetworks
scalar PLOT3D	calculate derived plot3d scalar functions
scatter dots	generate spheres at points in 3D space
shrink	make an object smaller
stagnation PLOT3D	strip out the plot3d stagnation energy
stream lines	generate stream lines for a vector field
stream mesh	generate stream lines for a vector field as a polygonal mesh
threshold	restrict values in data field

transpose	exchange dimensions in a 2D or 3D data set
tube	convert lines to cylindrical tubes
vbuffer	perform volumetric rendering on volume data
vector PLOT3D	calculate derived plot3d vector functions
vector curl	compute the curl of a vector field
vector div	compute the divergence of a vector field
vector grad	compute the vector gradient of a scalar field
vector mag	compute the magnitude of a vector field
vector norm	normalize a vector field
volume bounds	generate bounding box of 3D 3-vector field
volume manager	share volumes among subnetworks
wireframe	convert object from surface to wireframe representation
write field	write a field description to disk
write image	store image data in a file
write volume	write volume data to a file

NAME

arbitrary_slicer – map 3D scalar field to arbitrary plane

SUMMARY

Name arbitrary slicer

Type mapper

Inputs field 3D scalar byte
colormap

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
X rotation	dial	0	0	360
Y rotation	dial	0	0	360
distance	dial	0	–2	2
mesh res	integer	36	8	144
trilinear	toggle	off		

DESCRIPTION

The arbitrary slicer module extracts a 2D slice from a 3D volume of data. The slice place can be oriented arbitrarily — it need not be parallel to any of the coordinate axes.

The volume of data is represented as a 3D scalar field. The slice plane is represented as a 2D grid, with a parameter-controlled resolution. The intersection of the volume and the grid is a mesh of vertices in 3D space.

Each vertex in the mesh is assigned a color that corresponds to one or more values of the 3D scalar field. Because the mesh vertices do not coincide with the original lattice points, an interpolation method can be used — refer to the **trilinear** input parameter.

By default, the volume is placed at the origin and the slice plane is the X-Y plane. You can control the resolution of the mesh using the mesh res parameter. At lower resolutions, fewer original data points are used in the computations; at higher resolutions, more points are used.

The optimal way to use this module is to start off with a low resolution mesh, position it as desired, then increase the resolution and turn on trilinear mapping.

INPUTS

Data Field (required; field 3D scalar byte uniform)
The input data must be a 3D field, with a byte value at each location in the field. The field must be uniform.

Colormap (optional; colormap)
The values of a colormap are transformed to the range 0..255, and are then used as indexes into the colormap.

OUTPUTS

Geometry (geometry)
The output is a ConvexAVS geometry.

PARAMETERS

X rotation
Rotates the mesh along the first axis of the volume.

Y rotation
Rotates the mesh along the second axis of the volume.

distance Moves the mesh along its normal.

mesh res Controls the resolution of the slice plane mesh. Higher resolution meshes result in higher quality representations, but take longer to compute and render. The default mesh is 8 by 8.

trilinear Controls the way in which each vertex of the output mesh is assigned a color:

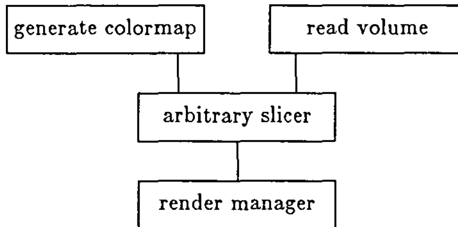
If **OFF**, a nearest-neighbor algorithm is used. Each mesh vertex is assigned the byte value of the nearest point in the lattice.

If **ON**, a trilinear interpolation is performed. The value at each vertex depends on the byte values at the eight lattice points that are the corners of the "enclosing cube."

The trilinear interpolation method is more accurate but takes longer to compute, particularly with larger meshes.

EXAMPLE

This example shows a common usage of arbitrary slicer:



LIMITATIONS

The arbitrary slicer module does not handle rectilinear or irregular field data in an optimal way. The slice plane is planar in computational space, not physical space. This can result in curved surfaces that are difficult to manipulate with control widgets.

NAME

bubbleviz – generate spheres to represent values of 3D field

SUMMARY

Name bubbleviz

Type mapper

Inputs field 3D scalar byte
colormap

Outputs
field irregular 1D 3-coord 4-vector real

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Radius	float	1.0	0.0	100.0

DESCRIPTION

The bubbleviz module generates spheres of various radii and colors at the element locations of a 3D field. This is a "cuberille" style of volume visualization, except that it uses spheres rather than cubes.

This module can be used for non-uniform input fields (rectilinear or irregular).

INPUTS

Data Field (required; field 3D scalar byte)
The input data for the bubbleviz module can be of type byte, real, double, or integer, but the range is limited by the bounds on the colormap used. Values falling outside this range are clamped to the nearest colormap bound.

Colormap (optional; colormap)
The optional colormap may be of any size. Because each input datum is a byte, the natural size for the colormap is 256. If you specify a larger colormap, its entries beyond the 256th are unused.

OUTPUTS

Data Field (field irregular 1D 3-coord 4-vector real)
The output is a list of points in 3D space, with a 4-vector of reals at each point:

The first element is the sphere's radius. If the radius value is 0.0, no sphere is generated as output.

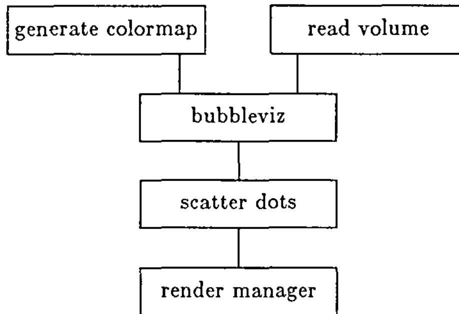
The last three elements specify the red-green-blue components of the sphere's color (0.0 = no color; 1.0 = maximum color).

PARAMETERS

Radius A multiplier factor for the sphere radii. This is particularly useful for irregular fields, for which the computational-to-physical mapping often makes the default spheres too small.

EXAMPLE

The following network uses bubbleviz:



RELATED MODULES

colorizer, gradient shade, read volume, scatter dots

LIMITATIONS

The bubbleviz module can generate extremely large databases (one sphere per voxel for volume data). Use 0.0 values in the last field of the input colormap ("opacity" field) to eliminate unnecessary data.

NAME

clamp – restrict values in data field

SUMMARY

Name clamp

Type filter

Inputs field *any-dimension any-data*

Outputs
field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
clamp_min	float	0.0	none	none
clamp_max	float	255.0	none	none

DESCRIPTION

The clamp module transforms the values of a field as follows:

Any value less than the value of the **clamp_min** parameter is set to **clamp_min**.

Any value greater than the value of the **clamp_max** parameter is set to **clamp_max**.

All values within the **clamp_min**-to-**clamp_max** range are not changed.

After being clamped, a data set's values are all in this range:

clamp_min ≤ *value* ≤ **clamp_max**

Note the difference between the clamp and threshold modules:

The threshold module sets values outside the specified range to be zero.

The clamp module sets values outside the specified range to be the range's minimum and maximum values.

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

Data Field (field *any-dimension any-data*)
The output field has the same dimensionality and type as the input field.

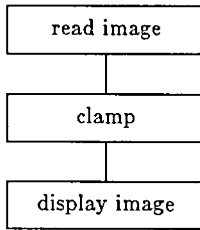
PARAMETERS

clamp_min
A floating-point number that specifies the minimum output value.

clamp_max
A floating-point number that specifies the maximum output value.

EXAMPLE

This example shows a common usage of clamp:



RELATED MODULES

colorizer, read volume, threshold

NAME

colorizer – convert field of data values to color values

SUMMARY

Name colorizer

Type filter

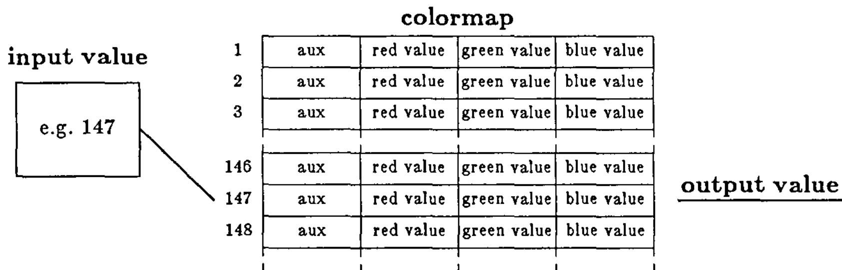
Inputs field *any-dimension* scalar byte
colormap

Outputs
field *any-dimension* 4-vector byte

Parameters
none

DESCRIPTION

The colorizer module converts the data at each point of a scalar field from a byte to a color (4-vector of bytes). The conversion is accomplished by using the byte value (0-255) as an index into a colormap:



INPUTS

Data Field (required; field *any-dimension* scalar byte)
The principal input data for the colorizer module is a field, which can be of any dimensionality. The data at each point of the field must be a single byte. The byte values will be interpreted as numbers in the range 0..255.

Colormap (optional; colormap)
The optional colormap may be of any size. Because each input datum is a byte, the natural size for the colormap is 256. If you specify a larger colormap, its entries beyond the 256th are unused. If this input is omitted, a gray-scale colormap is used (0 = black; 255 = white).

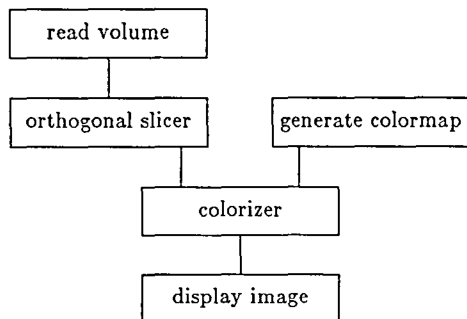
OUTPUTS

Field of Colors (field *any-dimension* 4-vector byte)
Each input byte is transformed into a color value that is structured as four bytes, as illustrated above. The red, green, and blue bytes specify a true-color pixel value. The auxiliary byte is typically used to specify an opacity value (0 = completely transparent; 255 = completely opaque).

The dimensionality of the output field is the same as that of the input field. The output field is four times as large as the input field because each byte (8 bits) is converted to a color value (32 bits).

EXAMPLE

This example shows how colorizer might be used:



RELATED MODULES

arbitrary slicer, display image, field to byte, generate colormap, gradient shade, read volume

NAME

colormap_manager – share colormaps among subnetworks

SUMMARY

Name colormap manager

Type data

Inputs none

Outputs colormap

Parameters

Name	*Type*
Colormap Manager	colormap
Colormap Choices	choice

DESCRIPTION

The colormap manager module produces a colormap data structure for use by modules that transform input data into color values. These modules include arbitrary slicer, bubbleviz, colorizer, field to mesh, and isosurface. colormap manager works exactly like generate colormap with one exception: separate active subnetworks, each with its own colormap manager module, share a single "pool" of colormaps.

A menu of all the active colormaps appears in a choice menu below each colormap manager's editing widget. All the menus have the same entries — different maps can be selected in different managers.

OUTPUTS

Colormap (colormap)
The output is a ConvexAVS colormap.

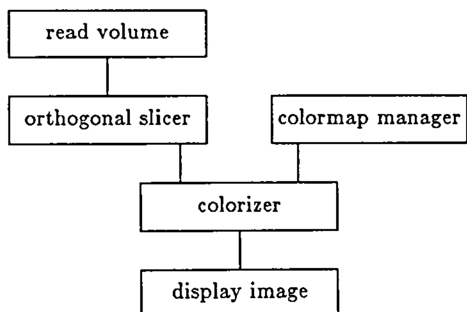
PARAMETERS

Colormap Manager
A colormap generator widget. Refer to the generate colormap manual page for details on using this widget.

Colormap Choices
A set of choices, listing each of the currently active colormaps.

EXAMPLE

This example shows a common usage of colormap manager:



NAME

combine_scalars – combine scalar fields into a vector field

SUMMARY

Name combine scalars

Type filter

Inputs field *any-dimension* scalar *any-data* (channel 0)
field *any-dimension* scalar *any-data* (channel 1)
field *any-dimension* scalar *any-data* (channel 2)
field *any-dimension* scalar *any-data* (channel 3)

Outputs
field *same-dimension* 1D/2D/3D/4D *same-data*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Vector Length	Dial	4	1	4

DESCRIPTION

The combine scalars module combines up to four fields with scalar data values into a field whose data values are vectors. The input field must be of like dimension and the scalar values must be of the same type. This module is most useful for constructing images or gradient fields from separately computed components.

The four input ports on the module are processed right to left. The right port contributes a value to the first element (lowest memory location) of each output vector while the left port contributes a value to the last element (highest memory location).

INPUTS

No inputs are required, but at least one must be present. If an input channel is omitted, zero values are output in the corresponding element of each output vector.

Channel 0 (optional; field *any-dimension* scalar *any-data*) A set of values to be output in the fourth dimension of the output vectors.

Channel 1 (optional; field *any-dimension* scalar *any-data*) A set of values to be output in the third dimension of the output vectors.

Channel 2 (optional; field *any-dimension* scalar *any-data*) A set of values to be output in the second dimension of the output vectors.

Channel 3 (optional; field *any-dimension* scalar *any-data*) A set of values to be output in the first dimension of the output vectors.

OUTPUTS

Field (field *same-dimension* 1D/2D/3D/4D *same-data*) The scalar input streams are assembled into a single output stream consisting of vectors, whose dimension is specified by **Vector Length.**

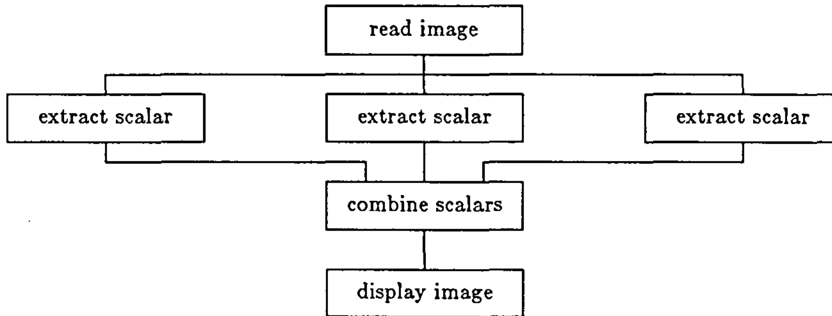
PARAMETERS

Vector Length

Specifies the dimension of the output vectors (1-4). The number of input ports that appears on the module icon varies according to this parameter.

EXAMPLE

The following network shows an example of combine scalars:



RELATED MODULES

extract scalar

NAME

compute_gradient – compute gradient vectors for 2D or 3D data set

SUMMARY

Name compute gradient

Type filter

Inputs field 2D/3D scalar byte

Outputs
field *same-dimension* 3-vector real

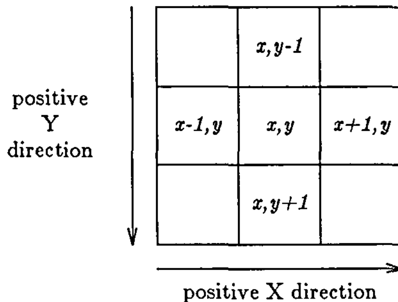
Parameters

Name	*Type*	*Default*	*Min*	*Max*
2D Height	float	0.5	0.0	1.0

DESCRIPTION

The compute gradient module computes the gradient vector at each point in a 2D or 3D field of data. The gradient is can be used (for example, by gradient shade) as a "pseudo surface normal" at each point.

A "nearest neighbor" approach is used to compute the gradient: in each direction, the component of the gradient vector is the difference of the *next* data and the *previous* data. In two dimensions, this can be pictured as follows:



$$\Delta x_{x,y} = data_{x+1,y} - data_{x-1,y}$$

$$\Delta y_{x,y} = data_{x,y+1} - data_{x,y-1}$$

$$\Delta z_{x,y,z} = data_{x,y,z+1} - data_{x,y,z-1} \quad \text{(<— for 3D data)}$$

$$\Delta z_{x,y} = 2D\ height \quad \text{(<— for 2D data)}$$

INPUTS

Data Field (required; field 2D/3D scalar byte) The input field may be either 2D or 3D. The data at each point of the field must be a single byte. The byte values will be interpreted as integers in the range 0..255.

OUTPUTS

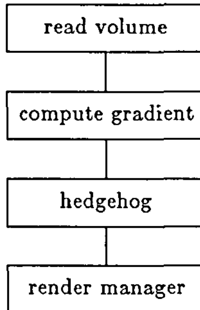
Data Field (field *same-dimension* 3-vector real)
The output field has the same dimensionality as the input field. For each element, the output data is a 3D vector of reals, representing the 3D gradient.

PARAMETERS

2D Height (used for 2D data only) Supplies the Z-coordinate of the gradient. It can be used the change the apparent height of the surface. A value of 1.0 is generally a very "rough" or "noisy" surface, whereas values approaching 0.0 will show little effect for shading.

EXAMPLE

This network shows an example of compute gradient:



RELATED MODULES

display image, extract scalar, gradient shade

LIMITATIONS

There may be algorithms better than "nearest-neighbor" for computing the gradient.

This module produces 12 bytes per pixel (voxel). For example, a 128 x 128 x 128 byte volume is about 2.1 MB before the gradient is computed. The compute gradient module produces a 25.2 MB internal data set from this data. This will have an adverse performance effect on systems whose physical memory is 32 MB or less.

NAME

contrast – perform linear transformation on range of field values

SUMMARY

Name contrast

Type filter

Inputs field *any-dimension* scalar *any-data*

Outputs
field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
cont_in_min	float	0.0	*none*	*none*
cont_in_max	float	255.0	*none*	*none*
cont_out_min	float	0.0	*none*	*none*
cont_out_max	float	255.0	*none*	*none*

DESCRIPTION

The contrast module transforms all the values in a scalar field. Two different types of transformation take place:

Linear transform:
All values that fall within the "input range" specified by the **cont_in_min** and **cont_in_max** parameters are transformed linearly to the "output range" **cont_out_max .. cont_in_max**.

$$new_value = \frac{(cont_out_max - cont_out_min) * (value - cont_in_min)}{cont_in_max - cont_in_min}$$

(More precisely, this is an *affine transformation.*) In essence, this transformation "stretches" or "compresses" one specified range of data to fit another specified range.

Clamped values:
All values that fall outside the specified input range are "clamped" to the limit values of the output range.

The contrast module typically is used to remove low-level noise from images and volumes or to increase the contrast in faded images and volumes.

INPUTS

Data Field (required; field *any-dimension* scalar *any-data*)
The input data may be a field of any dimensionality.

OUTPUTS

Data Field
The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced.

PARAMETERS

cont_in_min
Specifies the bottom of the range of input values that will be transformed linearly.

cont_in_max
Specifies the top of the range of input values that will be transformed linearly.

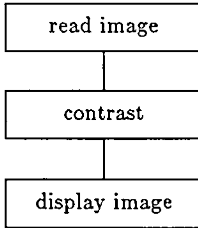
cont_out_min

Specifies the bottom of the range of output values. All values ≤ **cont_in_min** will be transformed to this value.

cont_out_max
Specifies the top of the range of output values. All values ≥ **cont_in_max** will be transformed to this value.

EXAMPLE

The following network shows how contrast could be used:



RELATED MODULES

read volume, threshold

NAME

crop – extract range of elements from a field

SUMMARY

Name crop

Type filter

Inputs field 1D/2D/3D *any-data*

Outputs field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
min x	int	first indx	first indx	last indx
max x	int	last indx	first indx	last indx
min y	int	first indx	first indx	last indx
max y	int	last indx	first indx	last indx
min z	int	first indx	first indx	last indx
max z	int	last indx	first indx	last indx

DESCRIPTION

The crop module changes the size of a field by extracting the data within a specified range of elements. This process is analogous to "cropping" a photographic image.

This module is useful for subsampling the data without changing it (for example, by interpolation). It preserves the resolution of the data, but may change its aspect ratio. Typical uses are to eliminate uninteresting portions of the data and to increase processing speed by reducing the amount of data.

INPUTS

Data Field (required; field 1D/2D/3D *any-data*)
The input data may be any field with a dimension of 1D, 2D, or 3D.

OUTPUTS

Data Field
The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced.

PARAMETERS

All the input parameters are normalized to the range 0.0–1.0. Thus, specifying a parameter value of 0.75 means "three-quarters of the way" along a particular dimension of the input field. Note that the parameters indicate *positions* of elements in the field — they have nothing to do with the *values* of field elements.

min x Specifies the lower bound array index in the field's first dimension.

max x Specifies the upper bound array index in the field's first dimension.

min y Specifies the lower bound array index in the field's second dimension.

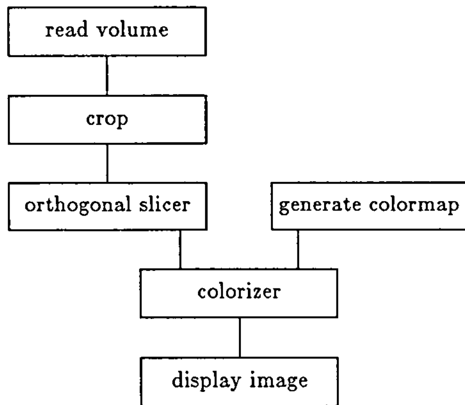
max y Specifies the upper bound array index in the field's second dimension.

min z Specifies the lower bound array index in the field's third dimension. (Has no effect for 2D input data sets.)

max z Specifies the upper bound array index in the field's third dimension. (Has no effect for 2D input data sets.)

EXAMPLE

The crop module could be used as follows:



RELATED MODULES
arbitrary slicer, colorizer, downsize, gradient shade, interpolate, orthogonal slicer, read volume

LIMITATIONS
The crop module only works for 2D and 3D data sets.

NAME

density_PLOT3D – strip out the plot3d density

SUMMARY

Name density PLOT3D

Type filter

Inputs field 3D 3-space irregular 5-vector real

Outputs field 3D 3-space irregular scalar real

Parameters none

DESCRIPTION

This module allows you to visualize the density field in the plot3d file. It merely converts the 5-vector into the density scalar field.

INPUTS

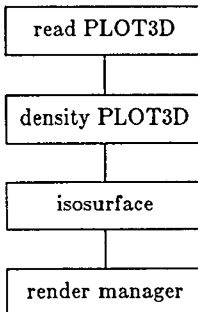
Data Field (required; field 3D 3-space irregular 5-vector real)
This input data is the 5-vector field output by read PLOT3D. It is the most important field because it contains the mesh and solution data.

OUTPUTS

Scalar Field (field 3D 3-space irregular scalar real)
A scalar field representing the density field from the plot3d file read by read PLOT3D.

EXAMPLE

This example reads in a plot3d data set and does an isosurface on the density field. Notice how only the plot3d field itself is used. We do not use the blanking records because we are extracting a part of the raw plot3d field.



For additional examples, refer to the read PLOT3D and vector PLOT3D manual pages.

RELATED MODULES

momentum PLOT3D, stagnation PLOT3D, read PLOT3D, scalar PLOT3D, vector PLOT3D

RELATED PROGRAMS

export_PLOT3D and import_PLOT3D

NAME

display_geometry_GL – display geometric image on Silicon Graphics Iris

SUMMARY

Name	display geometry GL
Type	renderer
Inputs	geometry geometry geometry
Outputs	none
Parameters	*Name* *Type* show control panel oneshot

DESCRIPTION

The display geometry GL module displays geometry data on a Silicon Graphics, Inc. Iris-4D series workstation, using the GL graphics library to access the 4D graphics hardware. It has capabilities similar to ConvexAVS' Geometry Viewer subsystem.

Other modules that produce geometry data can be connected to any of the three geometry input ports. The three ports are identical; there are several of them so that scenes can be composed with multiple objects in them. You can also invoke display geometry GL with no inputs so that the scene is initially empty.

When you select the **show control panel** button, a panel of controls similar to the Geometry Viewer's control panel appears. The controls include a current transformation button pad and **Objects**, **Lights**, **Cameras**, **Labels**, and **Action** menus.

Objects can be added to a scene either by upstream modules or by the **Read Object** selection on the display geometry GL control panel. Geometries and descriptions sent by upstream modules can be saved to files using the **Save Object** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Object**.

SPECIAL CONSIDERATIONS

In some circumstances, it is useful to access both the display geometry GL control panel and the Network Control Panel simultaneously. They both occupy the same screen position along the left edge. In these cases, use the X Window System window manager to move one of these menu windows.

The display geometry GL module is intended to replace all of the Geometry Viewer functionality on Iris workstations. You can still invoke the Geometry Viewer, but it will not take advantage of the Iris' rendering hardware to produces higher quality graphics quickly.

INPUTS

Geometry (optional; geometry)

Geometry (optional; geometry)

Geometry (optional; geometry)

The input data can be any ConvexAVS geometry. All geometries are combined into the same scene.

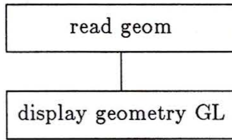
PARAMETERS

show control panel

A oneshot that makes the main control panel visible when it is activated.

EXAMPLE

This network shows how display geometry GL can be used:



RELATED MODULES

pdb to geom, read geom, render geom, render manager

LIMITATIONS

The display geometry GL module has no current object view showing the currently selected object.

It is not possible to use the left mouse button to drag text labels in display geometry GL.

Viewing windows must be square. If you read a scene description that contains a non-square viewing window, you get a square viewing window with dimensions equal to the larger value of the specification. For example, if you try to read a scene that contains a 50 by 200 viewing window, the result is a viewing window with dimensions of 200 by 200.

NAME

display_image – show image in a display window

SUMMARY

Name display image

Type renderer

Inputs field 2D 4-vector byte

Outputs none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Magnification	choice	x1	x1	x16
Automag_Size	integer	256	50	1024
Max Image Dimension	integer	1280	100	4096
			Choices	
Approximation Technique (16-plane system only)	choice	none	Ordered Dither Floyd Steinberg Random Monochrome None	

DESCRIPTION

The display image modules takes an input image and displays it in a display window. This window has a pulldown menu, accessed via the small square in the window's title bar. The menu allows you to control image magnification, window resizing, and other options relating to the display window.

When the image is larger than the display window, you can scroll it with the mouse, either by "dragging" the image itself or by using horizontal and vertical scrollbars.

You can resize the display window manually, using the X Window System window manager. You can also have the window resize itself automatically in response to a change in the image contents or a magnification selected from the display window's pulldown menu.

INPUTS

Data Field (required; field 2D 4-vector byte)

The input field must be in the ConvexAVS image format.

PARAMETERS

Magnification

A choice to specify a power of 2 (1,2,4,8,16) by which to multiply each dimension of the image.

Automag_Size

(for internal use only) This is used as a communications port to handle resizing of the image. Do not change this parameter.

Maximum Image Dimension

(for internal use only) Controls resource allocation — refer to the description below.

Approximation Technique (16-plane systems only)

Controls the conversion of color values to pixel values. There are four approximation techniques:

Ordered Dither: fast dithering technique

Floyd Steinberg: slower dithering technique that produces better results for computer-generated imagery

Random: uses a random number dither to approximate each color

Monochrome: uses the luminance of the color as an index into a greyscale ramp

None: takes the closest approximation for each color

MAGNIFICATION

You can magnify an image for closer examination, although the magnified image will provide no new detail. Magnification is implemented by duplicating the pixels in the original image. The result is "blockier" but provides a closer look at the image. There are several magnification levels (x1,x2,x4,x8,x16) in the pulldown menu.

Magnification may result in views that require more pixmap resources than the X server can provide. An "invisible" **Maximum Image Dimension** parameter provides a limit to how large an overall view can be safely produced. If the image size times the magnification exceeds this parameter in one of its dimensions, the module will present a warning and automatically reduce the magnification to produce a size within the limits. While intended as an internal limit, you could attach a widget to this parameter to increase this limit when required; exceeding the available resources of the X server will cause a crash of ConvexAVS as an X client process.

RESIZING

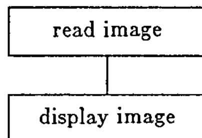
The display window can be resized in several ways. You can use the X window manager's resize window operation to enlarge or shrink the display window. An approximate image magnification is automatically chosen that makes the image at least as large as the window.

SCROLLING

Whenever the image is larger than the display window, only a portion of the image is visible. You can "pan" over the entire image by dragging the image itself. Place the mouse cursor anywhere in the image, click and hold down any mouse button, and drag the mouse. The image moves continuously, and the scrollbars are updated when you release the mouse button. The image automatically stops scrolling when it hits its borders.

EXAMPLE

The following network shows an example of display image:



RELATED MODULES

display pixmap

LIMITATIONS

This module can fail for large images that require more resources than the X server process can provide. While the **Maximum Image Dimension** protects against this in many cases, it may fail if the original image itself is too large. For large images, the downsize module may be useful in producing a more manageable original image size.

NAME

display_image_GL – display image on Silicon Graphics Iris

SUMMARY

Name display image GL

Type renderer

Inputs field 2D 4-vector byte

Outputs none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Zoom	integer	1	1	10
Monochrome	toggle			

DESCRIPTION

The display image GL module displays an input image in a window on a Silicon Graphics, Inc. Iris-4D series workstation using 24-bit color resolution.

INPUTS

Data Field (required; field 2D 4-vector byte)
The input field must be in the ConvexAVS image format.

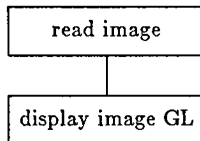
PARAMETERS

Zoom This control specifies a magnification factor to apply to the image. The window grows automatically to accomodate the magnification.

Monochrome
When this toggle is set, the image is displayed in gray scale.

EXAMPLE

This network shows how you could use display image GL:



RELATED MODULES

display image, display pixmap

LIMITATIONS

The workstation must have at least 24 color bitplanes.

There is no way to make the window smaller than the image being displayed.

The Silicon Graphics X server, Release 3.3, only supports 8 bitplane visuals, so it is not possible to take advantage of the Iris' full color resolution through the X Window System.

Because the display window is not an X window, you cannot use an X window manager to position it. You have to use the NeWS window manager.

NAME

display_pixmap – show pixmap in a display window

SUMMARY

Name display pixmap

Type renderer

Inputs pixmap

Outputs none

Parameters

Name	*Type*	*Default*	*Choices*
Store Frames	toggle		
Append Frame	oneshot		
Delete Current	oneshot		
Replay	choice	Off	Continuous Bounce Off
Current Frame	integer slider		
Max Frames	integer typein		
Replace Speed	integer slider		
Save Image	string typein		

DESCRIPTION

The display pixmap module displays its input pixmap in a display window. It automatically sizes the pixmap to fit the window.

In addition, you can:

Save the pixmap as a ConvexAVS image in a file.

Create and play back a "flipbook" of consecutive images.

INPUTS

Pixmap (required; pixmap)
The input data must be a ConvexAVS pixmap, typically created by the render geometry module.

PARAMETERS

Store Frames
This toggle controls whether all new frames are automatically added to the animation sequence.

Append Frame
Explicitly adds the currently displayed pixmap to the animation sequence. (Use when **Store Frames** is off.)

Delete Current
Deletes the currently displayed pixmap from the animation sequence.

Replay This choice widget controls how the animation sequence is to be played back: The choices are **Continuous, Bounce**, and **Off**.

Current Frame
The number of the current frame in the animation sequence (first frame = 0). This field is a typein — change the number to jump directory to another frame.

Max Frames
A typein field that specifies the ceiling for the number of frames that you can place in an animation sequence.

Replay Speed
Controls the rate at which an animation is played back. The larger the value, the greater the delay between frames.

Save Image
This is a typein field. If you enter a filename or pathname into this field, the current pixmap is written to a file.

SAVING AN IMAGE

To save an image in a file, enter the filename as the value of the **Save Image** parameter. To save another image under the same name, you can move the mouse cursor to the **Save Image** input area and press the **Return** key again.

ANIMATION

By changing the input data or by adjusting the parameters of upstream modules, you can have the display pixmap window show a sequence of images. You can create an animation ("flip book" by designating certain images to be "frames." Then, you can play back the images, adjusting the speed with a control widget.

Because each of the images in a flip book takes up a significant amount of system memory, there is a **Max Frames** parameter. Be sure that its value is low enough so that your system can comfortably keep all of the images in memory at the same time. ConvexAVS requires roughly four bytes of memory per pixel of each image. The larger the display window, the greater the memory requirements.

There are two ways to create a flip book:

To save *all* the images that appear in the window (actually, just the last **Max Frames** that are produced), turn on the **Store Frames** toggle. As each image is drawn, it will be appended to the end of the flip book. If **Max Frames** images have already been saved, this new pixmap will replace the oldest pixmap in the cycle.

If you want to selectively add images to the flip book, modify the image until it is as you want it, then select the oneshot **Append Frame**. This appends the image to the end of the existing flip book. This method allows you to carefully construct a flipbook animation.

The **Replay** parameter controls the way in which the flip book is displayed. It has three selections:

Continuous plays through all of the frames in the animation, wrapping around when it reaches the end.

Bounce plays forward through the last **Max Frames** or fewer frames. When it reaches the end, it plays backwards through those frames.

Off turns off the animation facility

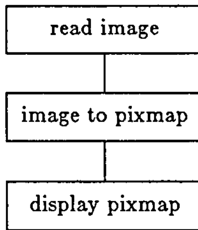
The **Replay Speed** parameter controls the rate at which flip book frames are displayed.

The **Current Frame** parameter allows you to select a particular frame "manually." It is normally updated to display the current frame, but for cases in which such updating would impact animation performance, it is not updated. Because only the last **Max Frames** frames are stored, the animation can begin at a frame other than 0.

After you select a particular frame, you can delete it with the oneshot **Delete Frame**.

EXAMPLE

The following network shows display pixmap in use:



RELATED MODULES

render geometry

LIMITATIONS

There is no way to store the "first **Max Frames**" frames of an animation loop.

NAME

dot_surface – generate points that define an isosurface

SUMMARY

Name dot surface

Type filter

Inputs field 3D scalar *any-data*

Outputs field 1D scalar irregular 3-space

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Stepsize	real	.01	1.0E–5	1.0
Threshold	real	.02	–100	100

DESCRIPTION

The dot surface module accepts a 3D scalar field as input and generates a list of points that defines an isosurface. The input field is composed of cells, where each cell is defined as a subvolume composed of six faces. Each cell is processed checking for a possible intersection of the surface. If the cell does contribute to the surface it is then subdivided until the maximum physical dimension of the resulting subcell is ≤ the value of the **Stepsize** parameter. A smooth surface can be generated in this manner, given a sufficiently small **Stepsize** value.

The running time of this module is directly proportional to the number of cells processed and the number of cells that contribute to the surface. It is inversely proportional to the **Stepsize** value.

If the input field is uniform, then a physical grid is generated mapping the data volume into a canonical size. The largest dimension of the volume is mapped into the interval: [-1.0, +1.0]. Other dimensions are scaled accordingly, thus if a uniform volume consisting of 100 nodes in the X-direction, 50 in the Y-direction and 20 in the Z-direction will have a bounding volume of: x=[-1.0, +1.0], y=[-0.5,+0.5], z=[-0.2,+2.0]. The distance between each node is then approximately equal to 0.02. The **Stepsize** parameter is relative to this length scale.

INPUTS

Data Field (required; field 3D scalar *any-data*)
This module uses a scalar data value for each field element. If the input is a vector-valued field, then the first component of the vector is used as the scalar value.

OUTPUTS

Point List (field 1D scalar irregular 3-space). The scalar data value for each output field element is unused. The only useful information is the 3D coordinate data.

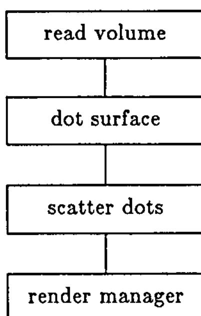
PARAMETERS

Stepsize A floating-point value that determines the resolution of the isosurface. The smaller this value, the smoother the surface.

Threshold A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the field element's data value equals the **Threshold** value.

EXAMPLE

This example shows you how you can use the dot surface module:



LIMITATIONS

The number of points may be inadequate to represent areas of small surface curvature with respect to the cell's local coordinate system.

A maximum of 80,000 points will be generated. Once the module calculates this number of points, it returns leaving all other cells unprocessed.

RELATED MODULES

combine scalars, isosurface, read volume, scatter dots, vbuffer

NAME

downsize – reduce size of data set by sampling

SUMMARY

Name downsize

Type filter

Inputs field *any-dimension any-data*

Outputs field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
downsize	integer	1	1	16

DESCRIPTION

The downsize module changes the size of the input data set by subsampling the data. It extracts every *N*th element of the field along each dimension, where *N* is the value of the **downsize** factor parameter. This technique preserves the aspect ratio of the input data.

This module is useful for operating on a reduced amount of data, in order to adjust other processing parameters interactively. After the parameter values have been set, you can remove the downsize module so that the full data set is used for final processing.

Alternatively, retain the downsize module in the network so that you can interactively choose between image quality (**downsize** factor = 1 for highest-resolution data) and execution speed (**downsize** factor > 1 for lower-resolution data).

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

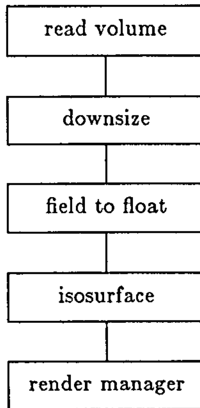
Data Field
The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced by the **downsize** factor.

PARAMETERS

downsize Determines how data elements from the field are sampled. Increasing this parameter causes more elements to be skipped over, thus decreasing the size of the output.

EXAMPLE

The following network shows downsize in use:



RELATED MODULES

arbitrary slicer, colorizer, crop, gradient shade, interpolate, orthogonal slicer, read volume

LIMITATIONS

The downsize module works for 1D, 2D, and 3D data sets only.

NAME

export_PLOT3D, import_PLOT3D – convert plot3d files into and out of FORTRAN formatted and unformatted form

SYNOPSIS

export_PLOT3D <command_file

import_PLOT3D <command_file

DESCRIPTION

export_PLOT3D converts one or more plot3d files from Convex FORTRAN unformatted form into formatted form (ascii). This allows them to be used on non-Convex machines.

import_PLOT3D converts one or more plot3d files from Convex FORTRAN formatted form (ascii) into unformatted form. This allows them to be used with the ConvexAVS plot3d reader.

IMPORTING FILES

import_PLOT3D is designed to obtain a list of files to convert from unit 6 (stdin). Input to import_PLOT3D is a set of lines of the following format:

<input file pair> <flags> <output files> <flags>

The *<input file pair>* field consists of the mesh (x) and solution (q) file names in that order.

The *<flags>* field is a two-character field that tells import_PLOT3D whether the file in question is whole or plane and whether it contains blanking records or not. Thus a regular expression for the flag field would be:

(w|p)(b|n).

An example input line to import_PLOT3D might be:

WhlBlnkx.dat WhlBlnkq.dat wb WhlBlnkx.fmt WhlBlnkq.fmt wb

It is also possible to convert a file from whole to plane data in the process of importing it. An example of this functionality would be:

WhlBlnkx.dat WhlBlnkq.dat wb PlnBlnkx.fmt PlnBlnkq.fmt pb

EXPORTING FILES

export_PLOT3D is designed to obtain a list of files to convert from unit 6 (stdin). Input to export_PLOT3D is a set of lines of the following format:

<input file pair> <flags> <output files> <flags>

The *<input filename pair>* is made up of the mesh (x) and solution (q) file names in that order.

The *<flags>* field is a two-character field that tells export_PLOT3D whether the file in question is whole or plane and whether it contains blanking records or not. Thus a regular expression for the flag field would be:

(w|p)(b|n).

An example input line to export_PLOT3D might be:

WhlBlnkx.dat WhlBlnkq.dat wb WhlBlnkx.fmt WhlBlnkq.fmt wb

It is also possible to convert a file from whole to plane data in the process of exporting it. An example of this functionality would be:

WhlBlnkx.dat WhlBlnkq.dat wb PlnBlnkx.fmt PlnBlnkq.fmt pb

LIMITATIONS

It is important to remember that the export and import utilities have no independent way of knowing whether the flags you give it are appropriate for the files you are passing it. If you import a file with flags that do not reflect the true nature of that file (for example, you give an input plane file the w flag), unpredictable and undesirable results will be generated. Examine the input file and ensure whether it contains blanking records or not and whether it is organized in planes or whole. Be careful not to pass the mesh file in as a q file and vice versa.

NAME

extract_scalar – extract scalar field(s) from a vector field

SUMMARY

Name extract scalar

Type filter

Inputs field *any-dimension* n-vector *any-data*
($n = 1..9$)

Outputs
field *same-dimension* scalar *same-data*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	int	0	0	9

DESCRIPTION

The extract scalar module inputs a field whose data values are vectors (1D to 10D) and outputs one of the dimensions ("channels") as a scalar-valued field. The output field has the same structure as the input field, except that its data values are scalars (vector length of 1).

This module is useful for performing operations on individual channels of vector fields. It is frequently used with the combine scalars module, which composes vector fields from individual scalar fields.

INPUTS

Data Field (required; field *any-dimension* n-vector *any-data*)
The input data may be any field whose data values are vectors with 10 or fewer dimensions. Even scalar fields may be used because their data values are considered to be 1D vectors.

OUTPUTS

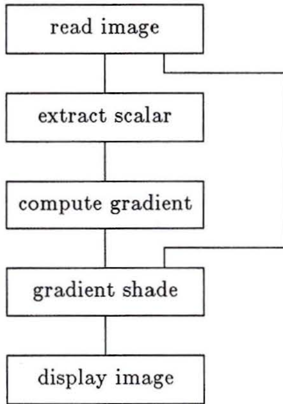
field (*same-dimension* scalar *same-data*) The output field has the same dimensionality as the input field. The data for each element is reduced from a vector to a scalar.

PARAMETERS

Channel Selects the dimension of the input data values to be output. The bounds of the control widget are automatically adjusted according to the vector length of the input field.

EXAMPLE

This example shows extract scalar in a network:



RELATED MODULES
combine scalars

NAME

field_to_byte – transform any field to a byte-valued field

SUMMARY

Name field to byte

Type filter

Inputs field *any-dimension any-data*

Outputs
field *same-dimension* byte

Parameters

Name	*Type*	*Default*	*Choices*
byte normalize	toggle	off	on, off

DESCRIPTION

The field to byte module takes a field of data (byte, real, double, or integer) and converts it to a byte field. It can be used in conjunction with volume visualization modules that have a bias towards byte fields (colorizer, compute gradient, arbitrary slicer, etc.).

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

Data Field (field *same-dimension* byte)
The output field has the same dimensionality as the input field, but each scalar value is forced to be a byte.

PARAMETERS

byte normalize
This is a toggle parameter:

If ON:
The data is transformed linearly into the range 0..255:

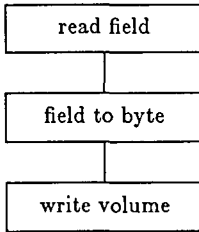
$$new_value = \frac{(value - min) * 255}{(max - min)}$$

If OFF:
The data is "clamped" so that no value falls outside the range 0..255:

If $value < 0$	$new_value = 0$
If $0 \leq value \leq 255$	$new_value = value$
If $value > 255$	$new_value = 255$

EXAMPLE

This network shows one way to use field to byte:



RELATED MODULES

field to double, field to float, field to int, read volume

NAME

field_to_double – transform any field to a field of double-precision floating point values

SUMMARY

Name field to double

Type filter

Inputs field *any-dimension any-data*

Outputs field *same-dimension* double

Parameters

Name	*Type*	*Default*	*Choices*
double normalize	toggle	off	on, off

DESCRIPTION

The field to double module takes a field of data (byte, real, double, or integer) and converts it to a double field. This may be useful for computing fields at greater data resolutions.

By default, the input data is simply cast (re-typed) to be double-precision floating point. If the toggle parameter is turned on, the data is also normalized to the range 0..1.

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

Data Field (field *same-dimension* double)
The output field has the same dimensionality as the input field, but each scalar value is forced to be a double-precision number.

PARAMETERS

double normalize
This is a toggle parameter:

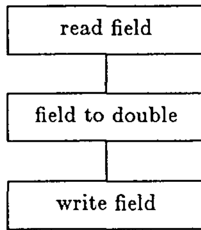
If ON:
The data is transformed linearly into the range 0..1:

$$new_value = \frac{(value - min)}{(max - min)}$$

If OFF:
The data is converted to double-precision floating point format (IEEE 754).

EXAMPLE

This network is one way you can use field to double:



RELATED MODULES
read volume

NAME

field_to_float – transform any field to a field of single-precision floating point values

SUMMARY

Name field to float

Type filter

Inputs field *any-dimension any-data*

Outputs field *same-dimension* float

Parameters

Name	*Type*	*Default*	*Choices*
float normalize	toggle	off	on, off

DESCRIPTION

The field to float module takes a field of data (byte, real, double, or float) and converts it to a float field. It can be used in conjunction with volume visualization modules that have a bias towards float fields (isosurface, dot surface).

By default, the input data is simply cast (re-typed) to be single-precision floating point. If the toggle parameter is turned on, the data is also normalized to the range 0..1.

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

Data Field (field *same-dimension* float)
The output field has the same dimensionality as the input field, but each scalar value is forced to be a single-precision number.

PARAMETERS

float normalize
This is a toggle parameter:

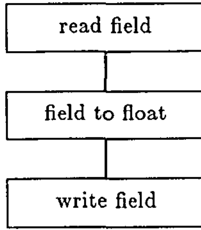
If ON,
the data is transformed linearly into the range 0..1:

$$new_value = \frac{(value - min)}{(max - min)}$$

If OFF,
the data is converted to single-precision floating point format (IEEE 754).

EXAMPLE

This network is one way you can use field to float:



RELATED MODULES

dot surface, isosurface, read volume

NAME

field_to_int – transform any field to an integer-valued field

SUMMARY

Name field to int

Type filter

Inputs field *any-dimension any-data*

Outputs
field *same-dimension* integer

Parameters

Name	*Type*	*Default*	*Choices*
int normalize	toggle	off	on, off

DESCRIPTION

The field to int module takes a field of data (byte, real, double, or integer) and converts it to an integer field. This may be useful for performing integer math with greater precision (–65536..65535) than that offered by byte fields (0..255).

By default, the input data is "clamped" to the range 0..65535. If the toggle parameter is turned on, the data is normalized to that range instead.

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

Data Field (field *same-dimension* integer)
The output field has the same dimensionality as the input field, but each scalar value is forced to be an integer.

PARAMETERS

int normalize
This is a toggle parameter:

If **ON**,
the data is transformed linearly into the range 0..65535:

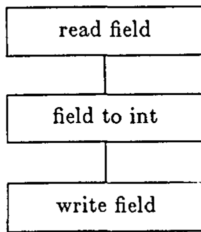
$$new_value = \frac{(value - min) * 65535}{(max - min)}$$

If **OFF**,
the data is "clamped" so that no value falls outside the range 0..65535:

If $value < 0$	$new_value = 0$
If $0 \leq value \leq 65535$	$new_value = value$
If $value > 65535$	$new_value = 65535$

EXAMPLE

This network is one way you can use field to int:



RELATED MODULES

read volume

NAME

field_to_mesh – transform a 2D scalar field to a surface in 3D space

SUMMARY

Name field to mesh

Type mapper

Inputs field 2D 2-space/3-space scalar
colormap

Outputs
geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Z scale	float	1.0	none	none

DESCRIPTION

For 2D 2-space:

The field to mesh module maps the 2D 2-space plane into 3-space by using the X- and Y-coordinates of the input field as the X- and Y-coordinates of a 3-space mesh and the value of the field at that point as the z-coordinate of the mesh. It maps the X,Y-plane to a surface in X,Y,Z-space by using the values defined on the plane as the height of the surface above the plane. This is the usual way of graphing a two dimensional function.

For 2D 3-space:

The field to mesh module takes the input field and creates a mesh from it without using the value of the field. It is important to realize that the values of the field do not affect the geometry of the mesh. field to mesh then uses the value of the field to "paint" the surface (mesh). This painting is done by driving a color lookup table with the functional values of the field. This mode of operation is useful when using the orthogonal slicer to visualize a 3D 3-space field.

INPUTS

Data Field (required; field 2D 2-space/3-space scalar)
The input data must be a 2D field with a scalar data value at each element. The data value may be of any primitive type: byte, integer, float, or double.

Colormap (optional; colormap)
Colors each vertex of the mesh, according to the data value (0..255) at that point. (Non-byte data values are first mapped to the range 0..255, then translated to colors.) If no colormap is supplied, the vertices are colored white.

OUTPUTS

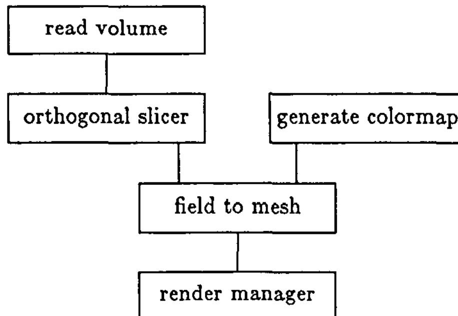
Geometry (geometry)
The output is a ConvexAVS geometry.

PARAMETERS

Z scale With uniform input fields, determines the height of the mesh. With rectilinear and irregular input fields, this parameter is unused.

EXAMPLE

This example shows field to mesh in a network:



LIMITATIONS

This module can output meshes that are too big for the render geometry module to handle, causing ConvexAVS to crash. Use the downsize filter module to reduce the size of the input data.

NAME

flip_normal – change direction of each vertex normal

SUMMARY

Name flip normal

Type filter

Inputs geometry

Outputs
geometry

Parameters
none

DESCRIPTION

The flip normal module transforms a ConvexAVS geometry so that all the vertex normals point in the opposite direction. This is most often used to correct normals that have been calculated incorrectly.

When its normals are backwards, a 3D object appears unaffected by light sources; it frequently appears as a grey silhouette.

INPUTS

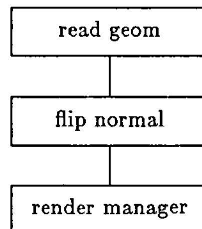
Geometry (required; geometry)
The output can be any ConvexAVS geometry.

OUTPUTS

Geometry (geometry)
The output is a ConvexAVS geometry that represents the same object.

EXAMPLE

This example shows you how to use flip normal:



RELATED MODULES

offset, read geom, render geometry, shrink, tube

NAME

generate_colormap – output colormap

SUMMARY

Name generate colormap

Type data

Inputs none

Outputs
colormap

Parameters

Name	*Type*	*Default*	*Min*	*Max*
colormap	colormap		(-- NOT APPLICABLE --)	
lo value	float	0	none	none
hi value	float	255	none	none

DESCRIPTION

The generate colormap module produces a ConvexAVS colormap data structure, for use by modules that transform input data into color values. These modules include arbitrary slicer, bubbleviz, colorizer, field to mesh, and isosurface

This module bases its output colormap on the state of the colormap editor control widget. The colormap editor uses a **hue-saturation-brightness** (HSB) color space model:

hue	0.00 = red 0.16 = yellow 0.33 = green 0.50 = cyan 0.66 = blue 0.83 = magenta
saturation	0.00 = white 1.00 = **hue**
brightness	0.00 = black 1.00 = **hue**

The HSB color space can be thought of as an inverted cone:

The **hue** axis runs circularly around the cone.

The **saturation** axis runs from the center of the cone (white) to its perimeter (fully saturated color).

The **brightness** axis runs from the tip of the cone (black) to the base (white).

OUTPUTS

Colormap (colormap)
The output is a ConvexAVS colormap.

PARAMETERS

colormap The state of the colormap editor control widget specifies the colormap to be generated. This widget has an editing panel and eight buttons:

hue Raises the hue editing panel. The default panel is a linear ramp: 0=blue

through 255=red.

saturation Raises the saturation editing panel. The default panel has all colors fully saturated: 0–255 = 1.0.

brightness Raises the brightness editing panel. The default panel has all colors at full brightness: 0–255 = 1.0.

opacity Raises the opacity editing panel. (The **opacity** value is placed in the auxiliary field of the colormap.) The default panel is a linear ramp: 0=0.0 through 255=1.0.

invert Inverts the currently raised editing panel, swapping high values with low values: the value at 0 becomes the value at 255, the value at 1 becomes the value at 254, and so on.

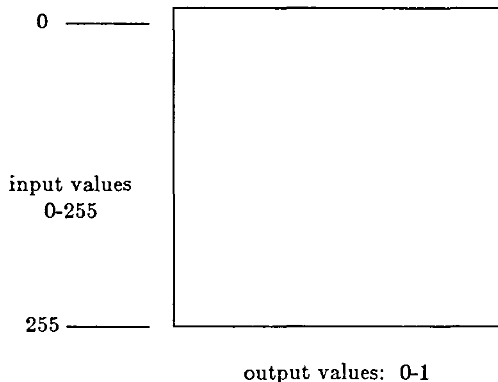
ramp Generate a linear ramp on the currently raised editing panel: 0=0.0 through 255=1.0.

read Reads a colormap from disk storage. Selecting this button pops up a File Browser widget, allowing you to specify a filename. You can also change the working directory.

write Write the current colormap to a disk file. Selecting this button pops up a File Browser widget, allowing you to specify a filename. You can also change the working directory.

You can change an editing panel from its current setting by scribing a curve with the mouse. Place the mouse cursor anywhere within the editing panel, hold down any mouse button, and drag upward or downward.

Each editing panel is organized as follows:



lo value Specifies the minimum data value that can be used as input to the colormap (the value at the top of the editing panel). The default low value is 0.

hi value Specifies the maximum data value that can be used as input to the colormap (the value at the bottom of the editing panel). The default high value is 255.

COLORMAP FILE FORMAT

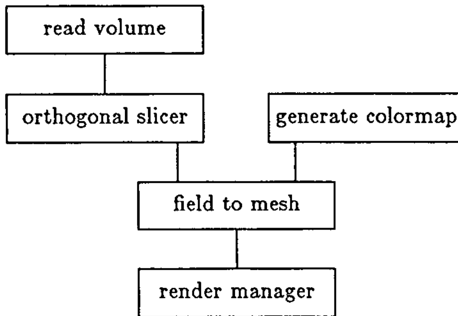
Colormaps are stored on disk as ASCII files, in the following format:

```
number_of_entries
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
 .       .           .          .
 .       .           .          .
 .       .           .          .
low_value high_value
```

The hue, saturation, brightness, and opacity values are normalized to the range 0.0-1.0. The default colormap has 256 entries, with the hue, saturation, brightness, and opacity default values as described above.

EXAMPLE

This network shows an example of generate colormap:



LIMITATIONS

The generate colormap module can only generate colormaps with 256 entries.

NAME

geom_to_scatter – convert geometry to point list

SUMMARY

Name geom to scatter

Type mapper

Inputs geometry

Outputs
field 1D 3-coordinate real irregular

Parameters
radius

DESCRIPTION

The geom to scatter module converts a ConvexAVS geometry to a point list of its vertices (a "scatter"). This is represented as a "field 1D 3-coordinate float irregular."

INPUTS

Geometry (required; geometry)
The input can be any ConvexAVS geometry.

OUTPUTS

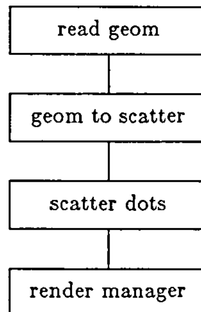
Data Field (field 1D 3-coordinate real irregular)
The output field is a list of points in 3D space — the vertices of the input geometry.

PARAMETERS

radius A value to be placed in the output field as the data value of each element.

EXAMPLE

This example shows geom to scatter:



RELATED MODULES

hedgehog, offset, particle advector, read geom, render geometry, shrink, tube, wireframe

NAME

gradient_shade – apply lighting and shading to colored data set

SUMMARY

Name gradient shade

Type filter

Inputs field 4-vector byte
field 3-vector real

Outputs
field 4-vector byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
ambient		0.1	0.0	1.0
diffuse		0.8	0.0	1.0
specular	float	0.0	0.0	1.0
gloss		20.0	0.0	50.0
lt theta	float	0.0	none	none
lt off-ctr	float	0.0	-90.0	90.0

DESCRIPTION

The gradient shade module accepts a colored 2D or 3D data set, along with its gradients (supplied by the compute gradient module). It applys a single light source to the colored data, then shades it.

The gradient at each location in the data field substitutes for the surface normal, which is used in traditional algorithms for lighting and shading surfaces. (A surface normal at a particular point on a surface is a vector perpendicular to the surface.)

Various shading styles are achievable using the lighting controls. These include creating shiny and matte surfaces and controlling the location of the light source.

INPUTS

Data Field (required; field 4-vector byte)
The input field is an image (2D pixel array) or a block of voxels (3D pixel array).

Gradient (required; field 3-vector real)
This field is the gradient of the **Data Field**.

OUTPUTS

Data Field (field 4-vector byte)
The output field has the same form as the **Data Field** input.

PARAMETERS

ambient The contribution of ambient (uniform background) lighting to the color. When this is set to 0.0, all surfaces facing away from the light source are black. When this is set to 1.0, surfaces appear in their own colors, with no shading information present.

diffuse The contribution of diffuse (directional) lighting to the color.

specular The contribution of specular lighting to the color.

gloss The sharpness of the specular highlight. The larger this value, the smaller and sharper the specular highlights.

lt theta The angle between (1) the projection of the light source on the X-Y plane and (2) the positive Y-axis. This value measures how much an off-center light source "swings around" the Z-axis.

lt off-ctr The angle between the light source and the positive Z-axis (which comes out of the screen at a right angle).

With *lt theta* = 0.0 and *lt off-ctr* = 0.0, the light source is coming straight from the eye perpendicular to the data. A positive *off-ctr* value moves the light source "up" (in the positive Y direction); a negative value moves it "down."

The equation for calculating the intensity of light reflected by a spot of surface is:

$$(int_{amb} * ambient) + (int_{diff} * diffuse * \cos(phi)) + (int_{diff} * specular * \cos^{gloss}(lt\ off\text{-}ctr))$$

In performing this computation, gradient shade:

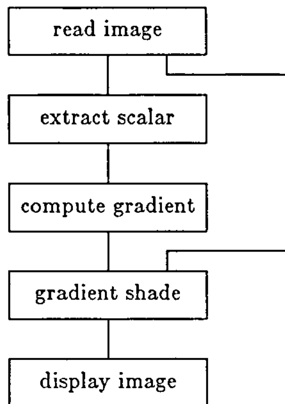
Assumes that int_{amb} and int_{diff} are both maximal (1.0).

Uses *lt theta* and *lt off-ctr* to compute *phi*, the angle between the surface normal (gradient vector) and the light source. The quantity cos(*phi*) is the attenuation (reduction) factor for the directional (diffuse) light.

Computes the quantity $\cos^{gloss}(\alpha)$, the attenuation factor for the specular highlight.

EXAMPLE

The following network uses gradient shade:



RELATED MODULES

colorizer, compute gradient, display image, extract scalar, read volume

NAME

hedgehog – show vectors on a slice of a 3D 3-vector field

SUMMARY

Name hedgehog

Type mapper

Inputs field 3D 3-vector float

Outputs
geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Vector Scale	float	1.0	0.01	10.0
Mesh Res	integer	16	2	128
X Rotation	float	0.0	0.0	360.0
Y Rotation	float	0.0	0.0	360.0
Distance	float	0.0	-2.0	2.0

DESCRIPTION

The hedgehog module takes as input a 3D field whose values are 3-vectors of floats. That is, the data represents a volume of lattice points, each point having a 3D vector of float values. This 3D-vector value can be visualized as a small line segment with a particular length and direction.

The hedgehog module takes an arbitrarily-oriented (parameter-controlled) slice through the volume, outputting the line segments that fall on the slice plane. The collection of line segments resembles the coat of a hedgehog.

The slice plane is represented as a 2D grid, with a parameter-controlled resolution. The intersection of the data volume and the grid is a mesh of vertices in 3D space.

For each vertex in the mesh, a small line segment is created. If a mesh point falls outside of the volume, no segment is created. The length and orientation of the segment depends on the nearby 3D vector values in the volume. Because the mesh vertices do not coincide with the lattice points in the data volume, a trilinear interpolation method is used to average the 3D-vector values of nearby lattice points. Refer to the arbitrary slicer man page for information on trilinear interpolation.

INPUTS

Volume Data (required; field 3D 3-vector float)
The input data must be a 3D field, representing a volume of points. The data value for each point must be a 3D vector of floats.

OUTPUTS

Geometry (geometry)
The output geometry is a collection of line segments that represent the 3D-vector values along the slice plane.

PARAMETERS

Vector Scale
The lengths of the line segments output by this module are proportional to this value.

Mesh Res Controls the resolution of the slice plane mesh. Higher resolution meshes result in higher quality representations, but take longer to compute and render. The default mesh is 16 by 16.

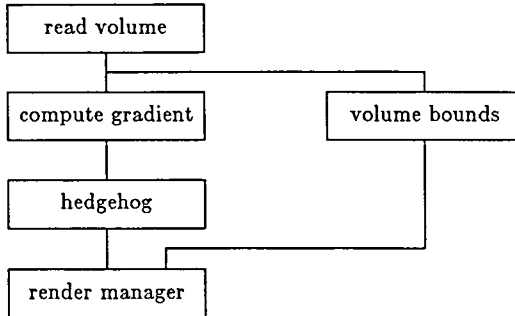
X Rotation
Rotates the mesh along the X-axis (first dimension) of the volume.

Y Rotation
Rotates the mesh along the Y-axis (second dimension) of the volume.

Distance Performs a translation perpendicular to the mesh plane.

EXAMPLE

The following network shows one way to use hedgehog:



RELATED MODULES

compute gradient, display pixmap, isosurface, read volume, render geometry, render manager, vector curl, vector div, vector grad, vector mag, vector norm volume bounds, volume manager

LIMITATIONS

The hedgehog module is useful for visualizing the magnitude of wind velocities. No supported modules generate this data type, however. This module can produce misleading geometries when run with irregular fields. The geometry produced by irregular fields can also be incorrectly registered with the volume limits of the field being viewed.

NAME

histogram_stretch – balance the histogram of a data set

SUMMARY

Name histogram stretch

Type filter

Inputs field *any-dimension* scalar byte

Outputs
field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
histr_min	int	0	0	255
histr_max	int	255	0	255

DESCRIPTION

The histogram stretch module processes images and volumes to balance the "histogram" of a data set between specified values. This operation combines histogram balancing (also called "histogram normalization" or "histogram equalization") and contrast stretching.

Finding the histogram of an image (or volume) consists of tallying the number of pixels (voxels) of each value into "bins." Byte data typically generates 256 bins (1 bin for each possible data value).

The histogram equalization process consists of trying to establish the same number of pixels (voxels) per bin by translating the pixel (voxel) values, using a well-chosen lookup table. This has the effect of creating an even distribution of values throughout the data set. It typically is used to enhance low-contrast images (volumes) or images in which the data is skewed towards one end of the spectrum.

Equalization is applied only to values within the range specified by the parameters **histr_min** and **histr_max**. Data outside this range is not included in the histogram generation and is eliminated.

INPUTS

Data Field (required; field *any-dimension* scalar byte)
The input data may be a ConvexAVS field of any dimensionality.

OUTPUTS

Data Field
The output field has the same form as the input field.

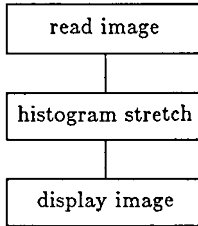
PARAMETERS

histr_min Specifies the bottom of the range of input values that will be histogrammed then transformed.

histr_max Specifies the top of the range of input values that will be histogrammed then transformed.

EXAMPLE

This network shows one use for histogram stretch:



RELATED MODULES

field to byte, field to double, field to float, field to integer, read volume

LIMITATIONS

This module works for byte fields only. (For other data types, there is no general way to determine the correct number of bins to generate.) To apply this module to non-byte data, use the field to byte module to pre-process the data.

NAME

image_manager – share images among subnetworks

SUMMARY

Name image manager

Type data

Inputs none

Outputs field 2D 4-vector byte (image)

Parameters

Name	*Type*	*Choices*
IMAGMGR Select	choice	Select, Replace
Image Manager	browser	
Image Choices	choice	

DESCRIPTION

The image manager module reads an image file from disk and outputs the image as a "field 2D 4-vector byte." It works like the read image module, except that it has both a cacheing mechanism and a way of sharing data among image manager modules in separate subnetworks.

Refer to the read image manual page for a description of the image format.

OUTPUTS

Data Field (field 2D 4-vector byte)
The output is a ConvexAVS image.

PARAMETERS

IMAGMGR Select

A choice that determines how newly-read images will be placed to the list of currently active images:

If **Select** is chosen, a new image is added to the end of the list.

If **Replace** is chosen, a new image replaces the currently selected member on this list.

In either case, the change is reflected in all the image manager modules in all active subnetworks.

Image Manager

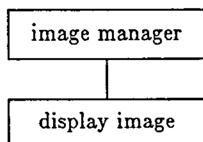
A file browser that allows you to select an image file to read.

Image Choices

A set of choices that lists each of the currently active images.

EXAMPLE

The following network is used to display an image:



RELATED MODULES

clamp, combine scalars, contrast, display image, display pixmap, extract scalar, histogram stretch, image to pixmap, interpolate, threshold

LIMITATIONS

The cached images are not freed until all image manager modules are destroyed. This is not the case with read image—the old data is freed whenever a new file is read.

NAME

image_to_pixmap – convert image to pixmap

SUMMARY

Name image to pixmap

Type mapper

Inputs field 2D 4-vector byte

Outputs pixmap

Parameters

Name	*Type*	*Default*	*Choices*
Approximation Technique (16-plane system only)	choice	none	Ordered Dither Floyd Steinberg Random Monochrome None

DESCRIPTION

The image to pixmap module takes as input an image ("field 2D 4-vector byte") and outputs the same image as a pixmap. It is useful for converting the output of modules that produce images into modules that require pixmaps.

The image and pixmap data types differ in these ways:

> Images allow for efficient direct manipulation by a module, whereas pixmaps allow for efficient manipulation by the display device.
>
> Pixmaps are directly usable by a display device (under control of the X server). In X terminology, pixmaps contain "pixel values" and images contain "colors." This difference is important only for 16-plane pseudo-color systems in which pixmap values are interpreted as indices into the system's color lookup table. An image contains 24-bit color values, which cannot be used on such systems that have only 12 color planes.
>
> A pixmap is represented by an X Window System resource ID (an integer). This means that transferring a pixmap from one module to another is more efficient than transferring all the data that defines an image.

Refer to the read image manual page for a description of the ConvexAVS image format.

INPUTS

Data Field (required; field 2D 4-vector byte)
The input field must be a ConvexAVS image.

OUTPUTS

Pixmap (pixmap)
The output is a ConvexAVS pixmap.

PARAMETERS

Approximation Technique (16-plane systems only)
Controls the conversion of color values to pixel values. The approximation techniques are:

> **Ordered Dither**: fast dithering technique
>
> **Floyd Steinberg**: slower dithering technique that produces better results for computer-generated imagery

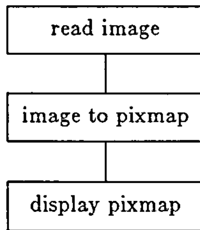
Random: uses a random number dither to approximate each color

Monochrome: uses the luminance of the color as an index into a greyscale ramp

None: takes the closest approximation for each color

EXAMPLE

This network shows image to pixmap:



RELATED MODULES

display pixmap, pixmap to image

NAME

interpolate – change the size of the data

SUMMARY

Name interpolate

Type filter

Inputs field 2D/3D scalar byte

Outputs
field *same-dimension* scalar byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Choices*
interp_sx	float	1.0	0.0	4.0	
interp_sy	float	1.0	0.0	4.0	
interp_sz	float	1.0	0.0	4.0	
sampling	choice	Point			Point, Bi/Trilinear

DESCRIPTION

The interpolate module arbitrarily changes the size of its input data, either by subsampling or interpolating it. This module is useful for smoothly scaling the data arbitrarily up and down.

The interpolation algorithm first selects for each output point its real (floating-point) position in the input data set:

New X = Old X * interp_sx
New Y = Old Y * interp_sy
New Z = Old Z * interp_sz

With the point sampling method, it then selects the closest pixel (voxel) to the computed one. With bilinear (in 2D) or trilinear (in 3D) sampling, it finds the four pixels (2D) or eight voxels (3D) around the computed point and does a linear sampling for in-between pixels.

The point sampling mode is much quicker than the linear sampling and should be used when interactivity is more important than image quality.

Some advantages to using this module are: it can scale non-uniformly in each dimension; it can do high-quality linear sampling; and it can scale data up instead of only down.

INPUTS

Data Field (required; field 2D/3D scalar byte)
The input field may be 2D or 3D. The data for each element must be a single byte.

OUTPUTS

Data Field (field *same-dimension* scalar byte)
The output field has the same form as the input field.

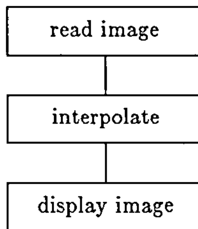
PARAMETERS

interp_sx
interp_sy
interp_sz The interpolation factors for the coordinate dimensions.

sampling This choice determines the sampling method -- **Point, Bilinear**, or **Trilinear.**

EXAMPLE

This example shows interpolate:



RELATED MODULES

crop, downsize

LIMITATIONS

This module performs incorrectly when down-sampling (going from a large image to a small one) in the Bi/Trilinear mode. It chooses the four pixels around the center of that region and interpolates among them. It should "average" appropriately chosen regions down to each pixel.

NAME

isosurface – generate an isosurface for a volume of data

SUMMARY

Name	isosurface	
Type	mapper	
Inputs	field 3D scalar real field 3D scalar real colormap	
Outputs	geometry	
Parameters	*Name*	*Type*
	Isosurface Level	float
	Optimize Surface Description	toggle
	Optimize Wireframe Description	toggle
	Flip Normals	toggle

DESCRIPTION

The isosurface module inputs a volume data set (3D field of floating-point values, either curvilinear, rectilinear, or uniform). It produces a geometric object that represents an isosurface of this object. An isosurface is a 3D generalization of a 2D contour line — it connects all field elements that have the same parameter-controlled data value.

The most important parameter is the **Isosurface Level** (threshold), which is defined in the unbounded floating-point data space of the volume. It is not always easy to know in what range the data is defined. Often, the data is defined by some well-known, real-world domain (for example, temperature in degrees). In some cases, the data has been converted from byte data and therefore must lie within the range 0.0-255.0.

Because isosurface can take a long time to compute, it is often advisable to include a downsize module in the network. This allows you to quickly select a proper isosurface level before generating one at full resolution.

Another technique is to use the **Action** capability of the render geometry module to save and play back a sequence of isosurfaces at different value levels.

INPUTS

Data Field (required; field 3D scalar real)

The input data must represent a volume, with a single floating-point value for each field element.

Auxiliary Data Field (optional; field 3D scalar real)

This port can be used to generate a colored isosurface; the color at each point on the surface indicates the value of another attribute of the volume.

For instance, you could generate a pressure isosurface with colors indicating the temperature at each point on the surface. In this case, the **Data Field** would be used to input the pressure data, and the **Auxiliary Data Field** would be used to input the temperature data.

In all cases, both volume data sets must have the same dimensions.

Colormap (optional; colormap)

If you use an **Auxiliary Data Field**, you must also specify a colormap. Because the auxiliary volume data is floating-point, you must adjust the **lo value** and **hi value**

parameters of the generate colormap module to correspond to the minimum and maximum data values of the auxiliary field.

For the pressure-temperature example described above, the temperature data set might have data values in the range 0.0-100.0 degrees. In this case, set the **lo value** to 0 and **hi value** to 100 in generate colormap.

OUTPUTS

Geometry (geometry)

A shaded surface, optionally with an associated wireframe representation.

PARAMETERS

Isosurface Level

A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the field element's data value equals the **Isosurface Level** value.

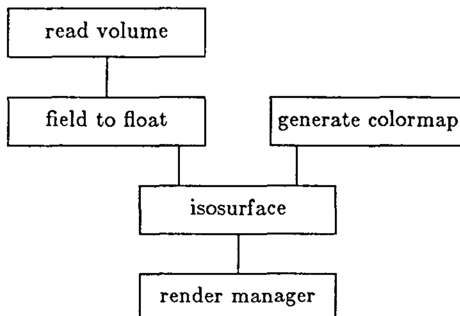
Optimize Surface Description
Optimize Wireframe Description

These two toggle parameters allow you to control a tradeoff between how efficiently the isosurface is computed and how efficiently it can be rendered. If you turn on **Optimize Surface**, extra time will be spent generating a more optimal surface description containing fewer triangles. Turn on **Optimize Wireframe** to generate a more optimal wireframe representation for the isosurface along with the shaded surface representation.

Flip Normals

Reverses the direction of each surface normal in the generated isosurface. If the normals point in the wrong direction, the outside of the isosurface will appear at the ambient light intensity. In this case, click this button or specify bi-directional lighting in the render geometry control panel (**Lights** selection).

EXAMPLE



RELATED MODULES

downsize, field to float, generate colormap, read field, read volume, render geometry

LIMITATIONS

In some circumstances, the generated isosurface may have some of its normals pointing inward and some outward. There is no way to correct this situation, but using bi-directional lighting may be helpful.

NAME

mirror – reverse array indices in a 2D or 3D data set

SUMMARY

Name mirror

Type filter

Inputs field 2D/3D *any-data*

Outputs
field of same type as input

Parameters

Name	*Type*	*Default*	*Choices*
axis	choice	Original	Original, X, Y, Z

DESCRIPTION

The mirror module reverses the array indexes along one dimension of a 2D or 3D field. This has the effect of creating a mirror image of the data set. In a 50 by 100 field, applying mirror to the X-dimension does the following (in FORTRAN array notation):

```
INPUT(1,i) ---> OUTPUT(50,i)   (for all 100 values of i)
INPUT(2,i) ---> OUTPUT(49,i)
INPUT(3,i) ---> OUTPUT(48,i)
INPUT(4,i) ---> OUTPUT(47,i)
  . . . . . . .
  . . . . . . .
  . . . . . . .
INPUT(50,i) ---> OUTPUT(1,i)
```

The mirror module can be used to change the orientation of the data for display and/or processing purposes.

To perform a reversal in two or more dimensions, use two or more mirror modules in succession.

INPUTS

Data Field (required; field 2D/3D *any-data*)
The input may be any ConvexAVS field.

OUTPUTS

Data Field
The output field as the same form as the input field.

PARAMETERS

axis The choices for exchanging the data are:

Original Copies the input to the output; no transformation is performed.

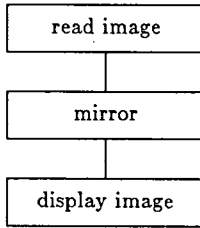
X Reverses the array indices in the X dimension (first dimension).

Y Reverses the array indices in the Y dimension (second dimension).

Z Reverses the array indices in the Z dimension (third dimension). (Equivalent to **Original** for a 2D field.)

EXAMPLE

This network uses one mirror module:



RELATED MODULES

transpose

NAME

momentum_PLOT3D – strip out the plot3d momentum vector

SUMMARY

Name momentum PLOT3D

Type filter

Inputs field 3D 3-space irregular 5-vector real

Outputs field 3D 3-space irregular 3-vector real

Parameters none

DESCRIPTION

This module allows you to visualize the momentum field in the plot3d file. It merely converts the 5-vector into the momentum vector field.

INPUTS

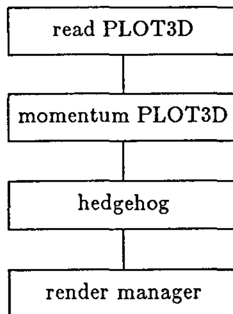
Data Field (required; field 3D 3-space irregular 5-vector real)
This input data is the 5-vector field output by read PLOT3D. It is the most important field because it contains the mesh and solution data.

OUTPUTS

Vector Field (field 3D 3-space irregular vector real)
A vector field representing the momentum field from the plot3d file read by read PLOT3D.

EXAMPLE

This example reads in a plot3d data set and does an isosurface on the momentum field. Notice how only the plot3d field itself is used. We do not use the blanking records because we are extracting a part of the raw plot3d field.



For additional examples, refer to the read PLOT3D and vector PLOT3D manual pages.

RELATED MODULES

density PLOT3D, stagnation PLOT3D, read PLOT3D, scalar PLOT3D, vector PLOT3D

RELATED PROGRAMS

export_PLOT3D and import_PLOT3D

NAME

offset – translate vertices along vertex normals

SUMMARY

Name offset

Type filter

Inputs geometry

Outputs
geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
offset	float	0.0	none	none

DESCRIPTION

The offset module transforms a ConvexAVS geometry, so that each vertex of each polygon is translated along its vertex normal. It is useful for emphasizing surface discontinuities (for example, cusps).

INPUTS

Geometry (required; geometry)
A ConvexAVS geometry, created with the libgeom library or by another module.

OUTPUTS

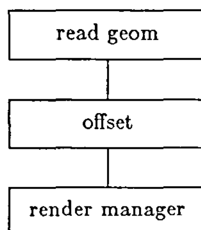
Geometry (geometry)
A geometry that represents that same object(s) as the input data.

PARAMETERS

offset The amount by which each vertex is translated along its normal. Positive values enlarge the geometry. Negative values collapse it.

EXAMPLE

This network shows the offset module:



RELATED MODULES

flip normal, read geom, render geometry, tube

LIMITATIONS

This module works only for polytriangle strips and meshes but not for polyhedra. It has no effect on objects that do not have surface normals.

NAME

orthogonal_slicer – slice through volume data with plane perpendicular to coordinate axis

SUMMARY

Name orthogonal slicer

Type mapper

Inputs field 3D *any-values*

Outputs field 2D *same-values*

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Choices*
slice plane	int	0	0	255	
axis	choice	K			I, J, K

DESCRIPTION

The orthogonal slicer module takes a 2D slice from a 3D array. It does so by holding the array index in one dimension constant and letting the other indices vary. For example, a data set might include a volume of 5000 points arranged as follows (using FORTRAN notation):

```
DATA(I,J,K)        I = 1,10
                   J = 1,20
                   K = 1,25
```

You can take a 2D "I-slice" from this data set by setting I = 4 and letting the other indices vary:

```
DATA(4,J,K)        J = 1,20
                   K = 1,25
```

The notation used in the example above assumes that the field's data values are scalars (in FORTRAN, DATA(4,5,6) must be a scalar). However, the orthogonal slicer module can also takes slices of vector-valued fields. It passes through whatever data type is presented to it. For example, if the input is a "field 3D 3-vector float," the output is a "field 2D 3-vector float."

INPUTS

Data Field (required; field 3D *any-values*)
The input may be any 3D field.

OUTPUTS

Data Field (field 2D *same-values*)
The output field is 2D instead of 3D and has the same type of data as the input field.

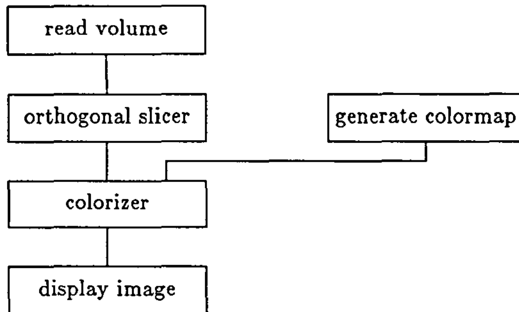
PARAMETERS

slice plane Determines the value of the array index to be held constant.

axis Selects the dimension (X, Y, or Z) in which the array index is to be held constant.

EXAMPLE

The following network shows orthogonal slicer:



The colorizer module is necessary because the output of orthogonal slicer is a "field 2D scalar byte" that must be cast into a ConvexAVS image field for display.

NAME

output_postscript – convert pixmap to PostScript and store in file

SUMMARY

Name output postscript

Type renderer

Inputs pixmap

Outputs
none

Parameters

Name	*Type*
filename	browser
mode	choice
monochrome	toggle
8 bit	toggle
compress	toggle
dither	toggle

DESCRIPTION

The output postscript module converts its input pixmap to the PostScript page description language and stores it in a file.

After the file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

Two types of PostScript output are supported:

Suitable for sending to a PostScript-compatible laser printer.

Mathematica compatible.

INPUTS

Pixmap (required; pixmap)
Any ConvexAVS pixmap.

PARAMETERS

filename A file browser that allows you to specify the name of the PostScript file to be created. After the file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

mode Selects the type of PostScript output: **laserwriter** or **mathematica**.

monochrome
If **ON**, produces monochrome output. If **OFF**, produces color output.

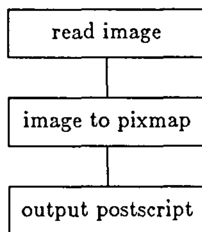
8 bit If **ON**, produces 8-bit output. If **OFF**, produces 4-bit output.

compressed
If **ON**, produces compressed output. If **OFF**, produces uncompressed output.

dither If **ON**, produces dithered output. If **OFF**, produces undithered output.

EXAMPLE

This example uses output postscript:



RELATED MODULES

image to pixmap, render geometry

LIMITATIONS

This module does not work on a 16-plane system.

The compress option is not supported in any released version of Mathematica.

The dither option produces visual artifacts on some images.

NAME

particle_advector – release grid of particles into velocity field

SUMMARY

Name particle advector

Type mapper

Inputs field 3D 3-vector float

Outputs
geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Mesh Res	integer	16	2	128
Time Step	float	0.0	0.0	1.0
Size	float	0.0	0.0	1.0
X Rotation	float	0.0	0.0	360.0
Y Rotation	float	0.0	0.0	360.0
Distance	float	0.0	–2.0	2.0
Advect Batch	oneshot	(-- NOT APPLICABLE --)		
Reset Particles	oneshot	(-- NOT APPLICABLE --)		
Color	toggle	off		
Show Bounds	toggle	on		
Surface	toggle	off		
Stop Advection	toggle	off		

DESCRIPTION

The particle advector module takes as input a 3D 3-vector field of floats (for example, fluid flow simulation data) and treats it as a velocity field. A 2D mesh of zero mass particles is "advected" (placed into the field at various initial positions with no initial direction or speed). The particles move through the velocity field according to the magnitude and direction of the vectors at the nodes in the volume. A forward differencing method is used to estimate the next position of each particle as a function of the current position and velocity.

This module is a ConvexAVS coroutine.

INPUTS

Data Field (required; field 3D 3-vector float)
The input data must be a 3D field, representing a volume of points. The data value for each point must be a 3D vector of floats.

OUTPUTS

Geometry (geometry)
The output is a ConvexAVS geometry that represents the mesh of particles.

PARAMETERS

Mesh Res Controls the number of particles.

Time Step Adjusts a scalar that multiplies the magnitude of the vector along which each particle is traveling. This causes successive positions of particles to be more widely spaced. (Refer to the **Color** parameter.)

Size Controls the radius of the particles that are rendered as spheres.

X Rotation
Rotates the mesh along the X-axis (first dimension) of the volume.

Y Rotation
Rotates the mesh along the Y-axis (second dimension) of the volume.

Distance Performs a translation perpendicular to the mesh plane.

Advect Batch
Triggers the release of a batch of particles. The number of particles is controlled by the **Mesh Res** parameter. The total number in each batch is **Mesh Res * Mesh Res.**

Reset Particles
Sets the total number of particles to zero.

Color If **ON**, colors the line segments to indicate how fast the particles are traveling through the velocity field:

red	fast backward
yellow	backward
green	slow backward
cyan	not moving
blue	slow forward
magenta	forward
red	fast forward

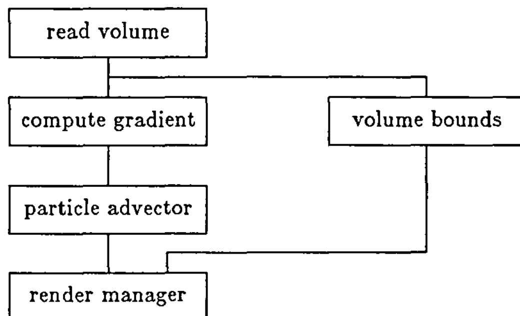
Show Bounds
Controls the visibility of the mesh of particles.

Surface Creates a solid shaded mesh. The coloring scheme is the same as that used with the **Color** parameter.

Stop Advection
Temporarily halts this module.

EXAMPLE

The following network shows you how to use particle advector:



RELATED MODULES

arbitrary slicer, display pixmap, isosurface, render geometry, render manager, vector curl, vector div, vector grad, vector mag, vector norm, volume bounds

LIMITATIONS

The particle advector module can produce misleading geometries when run with irregular fields. The geometry produced by irregular fields can also be incorrectly registered with the volume limits of the field being viewed.

NAME

pdb_to_geom – create molecule geometry from PDB file

SUMMARY

Name pdb to geom

Type data

Inputs none

Outputs
geometry

Parameters

Name	*Type*	*Choices*
Representation	choice	ball and stick, ball, stick, colored stick, colored residue

DESCRIPTION

The pdb to geom module reads the description of a molecule from a file in the Protein Data Bank (PDB) data format. Such files have a .pdb filename suffix for Polygen or a .ent suffix for Brookhaven. The output is a ConvexAVS geometry description of the molecule.

OUTPUTS

Geometry (geometry)
A ConvexAVS geometry description of the molecule.

PARAMETERS

Representation
The type of geometry produced:

ball and stick
Small spheres represent the atoms and white lines represent the bonds.

ball
Large spheres represent the atoms.

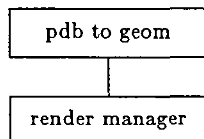
stick
White lines represent the bonds.

colored stick
Colored lines represent the atoms and their bonds.

colored residue
Colored lines represent the atoms and their bonds. The color of the lines represents the type of amino acid that the bond belongs to.

EXAMPLE

This example shows an application of pdb to geom:



RELATED MODULES
render geometry

LIMITATIONS
If you read in the same PDB file twice, you get only one instance of the geometry.

Because pdb to geom determines bonding only by the distance between atoms, large molecules take a long time to process, and some bonds are drawn incorrectly.

NAME

pdb_viewer – view and manipulate information in Brookhaven PDB format

SUMMARY

Name pdb viewer

Type data

Inputs none

Outputs
geometry

Parameters
none

DESCRIPTION

show control panel Display the control panel for the pdb viewer.

File Select The file browser is set to a directory containing PDB data files. This directory may be set in your .avsrc file as follows:

PdbDataDirectory /your/directory/path

The default data directory is /usr/avs/data/pdb.

Clicking on the data file name causes it to be loaded into the pdb viewer. The **file browser** button turns the file browser on and off. Its default is on.

If a data file is compressed by the UNIX 'compress' utility, it will be decompressed, loaded into the pdb viewer, and recompressed. You must have write permission in the directory the file is in for the decompression to occur.

Active Selection When a file is loaded it becomes active and visible. The previous active file becomes invisible. The buttons on the pdb viewer panel control attributes of the active file. To change visibilities of files which are not active, use the Geometry Viewer. To rebuild an active molecule's geometry, click its icon in the **Active Selection** menu.

Delete Active Cause the file which is currently active to be removed from the active select list. The previous file in the list is made active and visible. Ten files may be available at one time.

Information The textual data associated with a PDB file. The menu names and their corresponding PDB records are as follows:

Header - HEADER, COMPND, SOURCE, AUTHOR
Revision - OBSLTE, REVDAT, SPRSDE
Journal - JRNL
Remarks - REMARK
Footnote - FTNOTE
Nonstandard Group - HET, FORMUL
Symmetry - CRYST1, ORIGX, SCALE, MTRIX, TVECT
Statistics - Display counts of structural elements.
Data File - The entire data file in raw format is displayed.

Sequence Display the residue sequence.

Atom Select the manner in which atoms are displayed:

Atom Size Atomic radius constants come from College Chemistry by Holtzclaw, Robinson, and Nebergall (1984).

Ball and Stick Display atoms at 1/4 of their Van Der Waals radius. This is to produce a ball-and-stick representation.

Ionic Display atoms according to their ionic radius.

Covalent Display atoms according to their covalent radius.

Occupancy Display atoms according to the occupancy field in the data file. Atom records with no occupancy field default to the value of 1 angstrom.

VanDerWahls Display atoms according to their Van Der Waals radius. This is also known as CPK and is used to see a space-filled representation.

Atom Color The **Backbone Atoms** and **Side Chain Atoms** buttons correspond to their respective atom colors with the exception of carbon whose buttons are white.

Residue Type Color the atom according to which residue it belongs to and that residue's current color.

Atom Type Color the atom according its type. Carbon is grey, oxygen is red, nitrogen is blue, sulphur is yellow, hydrogen is white, deuterium is light green, and phosphorus is light blue.

Residue Atom The same as coloring by atom type, but with the carbons colored by their residue type.

Backbone Atoms Control the visibility of atom types belonging to the backbone. In the case of oxygen (O), the visibility of the bond to the backbone is also toggled.

Side Chain Atoms Control the visibility of atom types belonging to side chains.

Residue Select the manner in which residues are displayed:

Residue Color The **Display Amino Acid** buttons correspond to each residue's respective color according to the currently selected color mode.

Unique Each residue type has a unique color.

Hydro-icity Color according to the residue hydrophobicity or hydrophilicity.

Display Amino Acid Select which residues to display by their type.

The **ALL** button toggles the visibility of all of the residues. A useful technique is to turn **ALL** of the residues off, and then

turn individual ones on.

The **SET** button "clocks in" the current values of the **Affect Backbone** and **Affect Side Chain** buttons.

Affect Backbone This determines whether residue backbone atoms and bonds are affected by changes in the display parameters. By default, the buttons are off and thus residue backbones are not affected. This allows entire backbones to be displayed with only selected side chains. For changes in Affect Backbone to be seen, you must follow them by using the **SET** button.

Affect Side Chain This determines whether residue side chain atoms and bonds are affected by changes in the display parameters. By default, the buttons are on, so residue side chains are affected. This allows you to toggle the visibility of side chains of specific residue types. For changes in Affect Side Chain to be seen, you must follow them by using the **SET** button.

Motif Select display of the residues belonging to structural motifs defined in the data. The motif visibilities default to **ON** when a new molecule is loaded.

Residues belonging to **Helix**, **Sheet**, and **Turn** motifs are mutually exclusive.

Sulphur Bridge, and **Site** may share residues amongst themselves and with the previous three motifs.

Other displays the set of all residues not belonging to any other motif.

All toggles the visibility of all of the motifs. A useful technique is to turn **All** off and then turn on the individual motifs you want to view.

Chain Select the manner of displaying chains:

Backbone

Atoms Toggle the visibility of all of the atoms in the backbone.

Bonds Toggle the visibility of all of the bonds in the backbone.

Side Chain

Atoms Toggle the visibility of all side chain atoms.

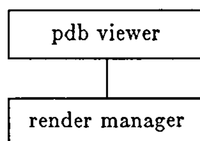
Bonds Toggle the visibility of all side chain bonds.

OUTPUTS

Geometry data which can be rendered by the geometry manager or the geometry viewer.

EXAMPLE

This example shows a simple application of the pdb viewer:



RELATED MODULES

display geometry GL, render geometry, render manager, tube

LIMITATIONS

When switching between molecules with different display parameters, the display buttons will get out of sync. To rebuild the active molecule according to the current display parameters, click its icon in the **Active Selection** menu.

Bond colors will not change until the molecule is rebuilt by pressing its icon in the Active Selection menu, or by changing atom radius or visibility. This is due to the way geometry handles color modification, and allows for maximum performance in atom color changes.

Nucleic Acid sequences are not fully supported. Bonds are not currently drawn, nor are the residue and atom types available from the menus. They can be accessed through the Geometry Viewer.

The data file browser does not work with compressed files.

NAME

pixmap_to_image – transform pixmap to image

SUMMARY

Name pixmap to image

Type mapper

Inputs pixmap

Outputs
field 2D 4-vector byte

Parameters
none

DESCRIPTION

The pixmap to image module takes a ConvexAVS pixmap as input and outputs a ConvexAVS image ("field 2D 4-vector byte"). The pixmap is an X Window System resource used to store image data in the X server. This reduces the amount of data ConvexAVS must pass between modules: a pixmap ID and window ID.

The 4-vector byte representation for the image consists of pixels that look like this:

auxiliary	red	green	blue

this field interpreted as pixel's opacity value

these three fields make up pixel's color value

The high-order byte field (auxiliary) is generally unused, but sometimes contains alpha (opacity) information on a per-pixel basis.

INPUTS

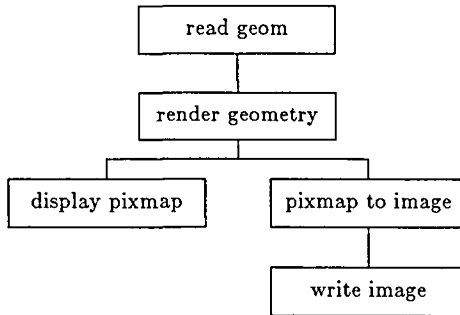
Pixmap (required; pixmap)
The input is any ConvexAVS pixmap.

OUTPUTS

Image (field 2D 4-vector byte)
The output data is a 2D block of pixels. The data set at each point of the 2D field will be a 4-vector of bytes in the ConvexAVS image format.

EXAMPLE

This module is useful for converting the output of renderers into images for writing to a file.



RELATED MODULES

clamp, colorizer, contrast, display image, display pixmap, generate colormap, histogram stretch, interpolate, render geometry, threshold, vbuffer

LIMITATIONS

This module will not work for 16-plane systems.

NAME

read_HDF_volume – read volume file in HDF format from disk into a field

SUMMARY

Name read HDF volume

Type data

Inputs none

Outputs
field 3D scalar byte

Parameters

Name	*Type*
Read HDF Volume Browser	browser

DESCRIPTION

The read HDF volume module reads a disk file in the National Center for Supercomputing Applications' (NCSA) Hierarchical Data Format (HDF) and outputs the data as a "field 3D scalar byte." It is used to read data files containing scalar-valued volume data (for example, CAT scan data and NMR data).

OUTPUTS

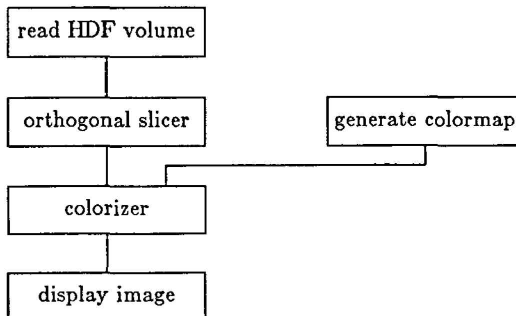
Data Field (field 3D scalar byte)
The output is the byte data cast as the scalar data in a 3D field.

PARAMETERS

Read HDF Volume Browser
A file browser allows you to specify the name of the file that contains the HDF volume data set.

EXAMPLE

This example shows read HDF volume:



RELATED MODULES

arbitrary slicer, bubbleviz, clamp, colorizer, compute gradient, contrast, crop, display image, downsize, dot surface, field to byte, field to double, field to float, field to int, field to mesh, generate colormap, gradient shade, histogram stretch, interpolate, isosurface, mirror, offset, orthogonal slicer, read volume, render geometry, transpose, vbuffer, volume bounds

REFERENCE

Information about the HDF file format is available from NCSA by anonymous FTP to ftp.ncsa.uiuc.edu (128.174.20.50). Log in by entering "anonymous" for the name. Enter your local login name for the password. Change into the HDF directory and enter "get README.FIRST" to obtain a copy of the README files that will tell you how to get and compile the most recent version of HDF.

LIMITATIONS

The HDF file format allows more than one volume description to be stored in a single HDF file. The current implementation of the read HDF volume module allows access to only the first volume data set in each file.

NAME

read_PLOT3D – read plot3d files

SUMMARY

Name　read PLOT3D

Type　data

Inputs　none

Outputs　field 3D 3-space irregular 5-vector real
field 3D scalar uniform integer
field 1D scalar uniform float

Parameters

Name	*Type*
Read Plot3D Data Browser (q)	browser
Read Plot3D XYZ Browser (x)	browser

DESCRIPTION

The read PLOT3D module reads a pair of plot3d files and builds three fields from it. The first field is a ConvexAVS floating point 5-vector irregular 3D 3-space field used to store the solution and mesh data itself. The second field is a 3D uniform scalar integer field used to store blanking records. The third field is a 4-element scalar real 1D uniform field used to store the four grid parameters (free stream Mach number, angle-of-attack (alpha), Reynolds number, and time from the solutions file). This last data is not used in all plot3d calculations but is made available for vector and scalar function routines.

This module is the most basic module in any plot3d network. It provides data for the system and other plot3d modules.

OUTPUTS

Data Field (field 3D 3-space irregular 5-vector real)
This field combines the mesh and solution data into a single 3D 3-space irregular 5-vector real field.

Blanking Data Field (field 3D scalar uniform integer)
This field contains the blanking records for the plot3d data set in the form of a 3D scalar uniform integer field. This data can be used as an optional input for the function modules (scalar and vector plot3d q.v.).

Global Parameters (field 1D scalar uniform float)
This field contains the four grid parameters (free stream Mach number, angle-of-attack (alpha), Reynolds number, and the time) that are present in the solution data as a separate record.

PARAMETERS

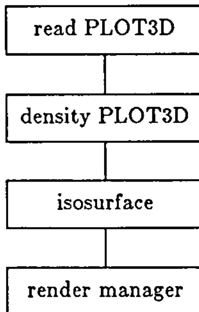
Read Plot3D Data Browser
Selects the file name of the data file. The data file, or solutions file as it is sometimes called, contains the values of the five plot3d solution elements (density, xmomentum, ymomentum, zmomentum, and stagnation energy) for each point at which they are defined. It also contains the four grid parameters.

Read Plot3D XYZ Browser
Selects the file name of the mesh file. This file contains the location in physical space of the points at which the solution data, Q(1-5), is available.

EXAMPLE

This example reads in a plot3d data set and does an isosurface on the density field. Notice how only the plot3d field itself is used. We do not use the blanking records because we are extracting

a part of the raw plot3d field.



For additional examples, refer to the scalar PLOT3D and vector PLOT3D manual pages.

RELATED MODULES

density PLOT3D, momentum PLOT3D, stagnation PLOT3D, vector PLOT3D, scalar PLOT3D

RELATED PROGRAMS

export_PLOT3D and import_PLOT3D

LIMITATIONS

The ConvexAVS plot3d reader module only reads Convex FORTRAN unformatted plot3d records. This allows the ConvexAVS plot3d reader to avoid overhead associated with ascii conversion at run time. It also allows the plot3d reader software to determine the file characteristics without user intervention. This is possible because Convex FORTRAN brackets the records it writes with record-byte-lengths when doing unformatted i/o. This record length information combined with the structure of a plot3d file allows the plt3dlib software to analyze the file to determine the existence of blanking records and whether the file is whole or plane. Unlike most plot3d software, you need not specify these characteristics to read PLOT3D.

There are two conversion utilities provided. Refer to export_PLOT3D and import_PLOT3D for details.

NAME

read_RLE_image – read image file in RLE format from disk into a field

SUMMARY

Name read RLE image

Type data

Inputs none

Outputs

field 2D 4-vector byte

Parameters

Name	*Type*
Read RLE Image Browser	browser

DESCRIPTION

The read RLE image module reads an image in the University of Utah's Run Length Encoded (RLE) image format from disk and outputs the image as a "field 2D 4-vector byte." Each field element represents a pixel. The data value for each element is a 4D vector of bytes, laid out as follows:

0	red	green	blue

this field interpreted as pixel's opacity value — these three fields make up pixel's color value

OUTPUTS

Data Field

The output data is a 2D block of pixels. The data set at each point of the 2D field will be a 4-vector of bytes in the ConvexAVS image format.

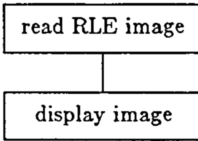
PARAMETERS

Read RLE Image Browser

A file browser window that allows you to specify the name of the RLE image file to be read.

EXAMPLE

This example shows read RLE image in a network:



RELATED MODULES

clamp, combine scalars, contrast, display image, display pixmap, extract scalar, histogram stretch, image to pixmap, interpolate, read image, threshold

REFERENCE

Information about the RLE file format can be found in the Utah Raster Toolkit available by anonymous FTP to cs.utah.edu (128.110.4.21) in the file pub/urt-3.0.tar.Z.

NAME

read_field – read field from a disk file

SUMMARY

Name read field

Type data

Inputs none

Outputs
field

Parameters

Name	*Type*
Read Field Browser	browser

DESCRIPTION

The read field module reads a ConvexAVS field data structure from a disk file and sends the field to its output port. The on-disk field format includes two parts, an ASCII header and a binary area, with a special separator between them.

OUTPUTS

Data Field (field)
The output data is a ConvexAVS field.

PARAMETERS

Read Field Browser
A parameter of data type "string" used to tell read field the name of a file to read. Its default widget is a file browser.

ASCII HEADER

The ASCII header contains a series of text lines, each of which is either a comment or a TOKEN=VALUE pair. For example, the following header defines a field of type "field 2D 4-vector byte," which is the ConvexAVS image format:

```
# AVS field file
#
ndim=2          # number of computational dimensions
dim1=512
dim2=480
nspace=2        # number of physical dimensions
veclen=4
data=byte
field=uniform
```

The first two lines are comments, indicated by the # character. Note that the first line of the header must begin with:

```
# AVS
```

Comments also occur at the end of the 3rd and 6th lines. Any characters following (and including) # in a header line are ignored.

The third through last lines of the header consist of TOKEN=VALUE pairs. This example shows all the token names, and all of them are required. An ASCII header that is missing one or more of these lines causes read field to generate an error. The tokens and their permitted values are

described below. Note that ASCII header specifications are not case-sensitive. You can surround the = character with any amount of white space (including none at all).

ndim = *value*
: The number of computational dimensions in the field. For an image, **ndim** = 2. For a volume, **ndim** = 3.

dim1 = *value*
dim2 = *value*
dim3 = *value*
...
: The dimension size of each axis (the array bound for each dimension of the computational array). The number of **dim***x* entries must match the value of **ndim**. For instance, if you specify a 3D field (**ndim**=3), you must specify the length of the X dimension (**dim1**), the length of the Y dimension (**dim2**), and the length of the Z dimension (**dim3**).

  Note that counting is 1-based, not 0-based.

nspace = *value*
: The dimensionality of the physical space that corresponds to the computational space (number of physical coordinates per field element).

  In many cases, the values of **nspace** and **ndim** are the same — the physical and computational spaces have the same dimensionality. But you might embed a 2D computational field in 3D physical space to define a manifold. Or you might embed a 1D computational field in 3D physical space to define an arbitrary set of points (a "scatter").

veclen = *value*
: The number of data values for each field element. All the data values must be of the same primitive type (for example, integer, so that the collection of values is conceptually a **veclen**-dimensional vector. If **veclen**=1, the single data value is a scalar. Thus, the term "scalar field" is often used to describe such a field.

data = byte
data = integer
data = float
data = double
: The primitive data type of all the data values.

field = uniform
field = rectilinear
field = irregular
: The field type. A uniform field has no computational-to-physical space mapping. The field implicitly takes its mapping from the organization of the computational array of field elements.

  For a rectilinear field, each array index in each dimension of the computational space is mapped to a physical coordinate. This produces a physical space whose axes are orthogonal, but the spacing among elements is not necessarily equal.

  For an irregular field, there is no restriction on the correspondence between computational space and physical space. Each element in the computational space is assigned its own physical coordinates.

SEPARATOR CHARACTERS

The ASCII header must be followed by two formfeed characters (that is, **Ctrl-L**, octal 14, decimal 12, hex 0C), in order to separate it from the binary area. This scheme allows you to use

more(1) to examine the header. When more(1) stops at the formfeeds, press **q** to quit. This avoids the problem of the binary data garbling the screen.

BINARY AREA

The size (in bytes) of the binary area depends on the field type:

For uniform fields, the binary area contains
data values only (no coordinates). Thus, the size is the product of the following numbers:

value of **dim1**	(product of sizes of computational dimensions
value of **dim2**	yields total number of field elements)
...	
value of **dim*x***	
value of **veclen**	(number of data values per field element)
size of **data**	(byte size of primitive data type)

In the stream of data values:

All the data values for a field element are stored together.

The first array index varies most quickly (FORTRAN style).

For rectilinear fields, the binary area contains
both data values and coordinates. The data values occupy the same amount of space as for a uniform field. Each coordinate is a single-precision floating-point number (4 bytes), and there is one coordinate for each array index in each dimension of computational space. Thus, the size of the coordinates area is:

$$(dim1 + dim2 \ldots + dimx) * 4$$

All of the X-coordinates are stored together at the beginning of the coordinates area. Following these are all the Y-coordinates, and so on.

For irregular fields, the data area contains
both data values and coordinates. The data values occupy the same amount of space as for a uniform field. Each coordinate is a single-precision floating-point number (4 bytes), and each field element is mapped to a point in *nspace*-dimensional physical space. Thus, the size of the coordinates area is:

$$(dim1 * dim2 \ldots * dimx) * nspace * 4$$

As with rectilinear field, all of the X-coordinates are stored together at the beginning of the coordinates are. Following these are all the Y-coordinates, and so on.

EXAMPLE 1

The following ASCII header describes a volume (3D uniform field) with a single byte of data for each field element. This format might be used to represent CAT scan data.

```
# AVS field file
ndim=3              # number of dimensions in the field
dim1=64             # dimension of axis 1
dim2=64             # dimension of axis 2
dim3=64             # dimension of axis 3
nspace=3            # number of physical coordinates per point
veclen=1            # number of components at each point
data=byte           # data type (byte, integer, float, double)
field=uniform       # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies the following amount of space:

(64 * 64 * 64) * 1 * 1 = 262,144 bytes

The coordinates area is null.

EXAMPLE 2

The following ASCII header describes a volume (3D uniform field) whose data for each field element is a 3D vector of single-precision values. This format might be used to represent the wind velocity at each point in space.

```
# AVS field file
ndim=3              # number of dimensions in the field
dim1=27             # dimension of axis 1
dim2=25             # dimension of axis 2
dim3=32             # dimension of axis 3
nspace=3            # number of physical coordinates per point
veclen=3            # number of components at each point
data=float          # data type (byte, integer, float, double)
field=uniform       # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies the following amount of space:

(27 * 25 * 32) * 4 * 3 = 259,200 bytes

The coordinates area is null.

EXAMPLE 3

The following ASCII header describes an irregular volume (3D irregular field) with one single-precision value for each field element. The binary area includes an (X,Y,Z) coordinate triple for each field element, indicating the corresponding point in physical space. This format might be used to represent fluid flow data.

```
# AVS field file
ndim=3              # number of dimensions in the field
dim1=40             # dimension of axis 1
dim2=32             # dimension of axis 2
dim3=32             # dimension of axis 3
nspace=3            # number of physical coordinates per point
veclen=1            # number of components at each point
data=float          # data type (byte, integer, float, double)
field=irregular     # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies the following amount of space:

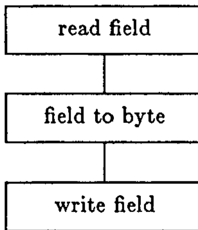
(40 * 32 * 32) * 4 * 1 = 163,840 bytes

The coordinates area occupies the following amount of space:

(40 * 32 * 32) * 4 * 3 = 491,520 bytes

EXAMPLE 4

This example shows read field in a network:



RELATED MODULES

arbitrary slicer, bubbleviz, clamp, colorizer, combine scalars, compute gradient, contrast, crop, display image, dot surface, downsize, extract scalar, field to byte, field to double, field to float, field to int, field to mesh, gradient shade, hedgehog, histogram stretch, image to pixmap, interpolate, isosurface, mirror, orthogonal slicer, particle advector, scatter dots, stream lines, stream mesh, threshold, transpose, vector curl, vector div, vector grad, vector mag, vector norm, volume bounds, write image, write volume

ERROR CHECKING

The read field module performs a significant amount of error checking. If an error is detected while reading the field, an error dialog box appears on the screen indicating the line in which the error occurred (if it was in the ASCII header) along with the type of error.

NAME

read_geom – reads a data file containing a geometry

SUMMARY

Name read geom

Type data

Inputs none

Outputs
geometry

Parameters

Name	*Type*
filename	browser

DESCRIPTION

The read geom module reads a file containing a ConvexAVS geometry and outputs the geometry to one or more modules connected to its output port. The resulting object will be named after the file from which it was read. Because ConvexAVS replaces geometries based on the object name, if you read in the same filename twice, you will only get one representation of the object.

Because the Geometry Viewer subsystem (also accessible as the render geometry module) has a built-in **Read Object** function, you rarely need to use this module. It is most useful in conjunction with a filter module that processes geometric data (for example, shrink).

OUTPUTS

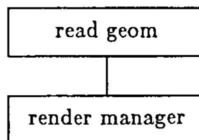
Geometry (geometry)
The output is the geometry that was read from the specified file.

PARAMETERS

filename A file browser allows you to specify the name of the file that contains a ConvexAVS geometry.

EXAMPLE

This example shows how to use read geom in a network:



RELATED MODULES

offset, render geometry, shrink, tube, wireframe

LIMITATIONS

This module reads geometry files (.geom) only. It cannot read script files (.obj) or scene files (.scene) that can be created with the Geometry Viewer Script Language.

The object is always named after the file from which it is read. This makes it awkward to create animation loops, for which you might want to direct multiple files to the same name or to read in multiple instances of the same object.

NAME

read_image – read image file from disk into a field

SUMMARY

Name read image

Type data

Inputs none

Outputs

field 2D 4-vector byte

Parameters

Name	*Type*
Read Image Browser	browser

DESCRIPTION

The read image module reads an image file from disk and outputs the image as a "field 2D 4-vector byte." Each field element represents a pixel. The data value for each element is a 4D vector of bytes, laid out as follows:

auxiliary	red	green	blue

this field interpreted as pixel's opacity value — these three fields make up pixel's color value

The auxiliary field is sometimes used to store opacity information on a per-pixel basis.

OUTPUTS

Data Field

The output data is a 2D block of pixels. The data set at each point of the 2D field will be a 4-vector of bytes in the ConvexAVS image format.

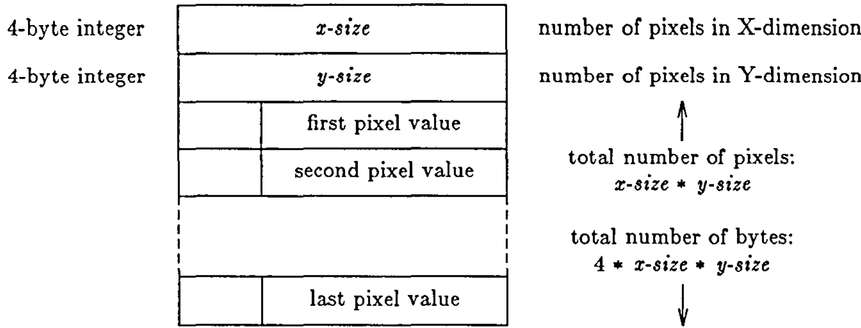
PARAMETERS

Read Image Browser

A file browser window that allows you to specify the name of the image file to be read.

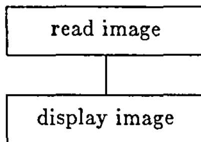
IMAGE FILE FORMAT

The read image module expects its input file to be in the following format:



EXAMPLE

This example shows read image in a network:



RELATED MODULES

clamp, combine scalars, contrast, display image, display pixmap, extract scalar, histogram stretch, image to pixmap, interpolate, read RLE image, threshold

NAME

read_volume – read volume file from disk into a field

SUMMARY

Name read volume

Type data

Inputs none

Outputs

field 3D scalar byte

Parameters

Name	*Type*
Read Volume Browser	browser

DESCRIPTION

The read volume module reads a disk file in volume data format and outputs the data as a "field 3D scalar byte." It is used to read data files containing scalar-valued volume data (for example, CAT scan data and NMR data).

OUTPUTS

Data Field (field 3D scalar byte)

The output is the byte data cast as the scalar data in a 3D field.

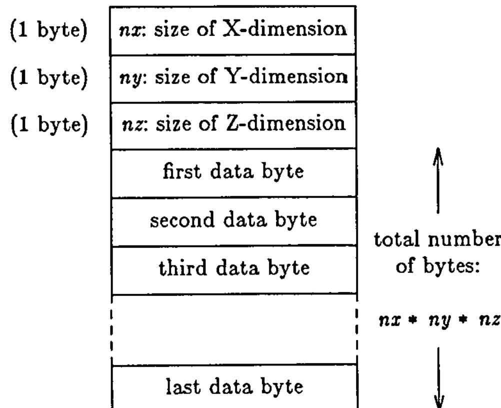
PARAMETERS

Read Volume Browser

A file browser allows you to specify the name of the file that contains the volume data set.

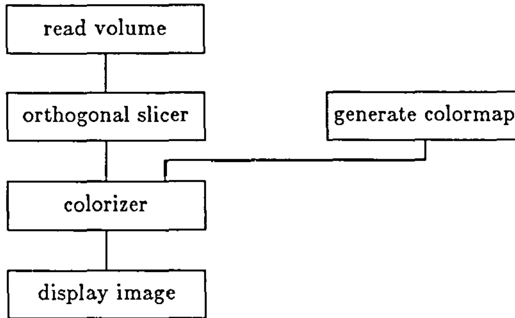
VOLUME DATA FILE FORMAT

The read volume module expects its input file to be in the following format:



EXAMPLE

This example shows read volume:



RELATED MODULES

arbitrary slicer, bubbleviz, clamp, colorizer, compute gradient, contrast, crop, display image, downsize, dot surface, field to byte, field to double, field to float, field to int, field to mesh, generate colormap, gradient shade, histogram stretch, interpolate, isosurface, mirror, offset, orthogonal slicer, read HDF volume, render geometry, transpose, vbuffer, volume bounds

NAME

render_geometry – convert geometric description to image (Geometry Viewer)

SUMMARY

Name render geometry

Type renderer

Inputs geometry

Outputs
pixmap

Parameters
Name
add to object transform

DESCRIPTION

The render geometry module provides access within a ConvexAVS network to the complete Geometry Viewer subsystem. Many different modules can supply input geometries. That is, many geometry-format outputs can be connected to render geometry's geometry input port. All the objects will be combined into a single scene. Each module providing input to render geometry can define attributes and geometries for any number of objects. Each of these modules can also define a hierarchical relationship among its objects.

You can also invoke render geometry with no inputs, so that the scene is initially empty. Objects can be added to a scene either by upstream modules or by the **Read Object** selection on the render geometry control panel. Geometries and descriptions sent by upstream modules can be saved to files using the **Save Object** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Object**.

SPECIAL CONSIDERATIONS

Instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level application menu of the Geometry Viewer subsystem. Thus, when you add the geometry viewer icon to a network, no page is added to the Network Control Panel. There are two ways to access the Geometry Viewer menu:

Click the small square in the render geometry icon with the left mouse button.

Click the **Geometry Viewer** button located at the top of the Network Control Panel. This button is always visible even when there is no active network.

In some circumstances, it is useful to be able to access both the Geometry Viewer control panel and the Network Control Panel simultaneously. They both occupy the same position, along the left edge of the screen. In these cases, use the X Window System window manager to move one of these menu windows out of the way.

The Geometry Viewer's control panel also differs from that of other modules in these ways:

The Network Editor's **Layout Editor** cannot be used to rearrange the Geometry Viewer.

If a network includes more than one instance of render geometry, ConvexAVS does not create a separate control panel for each instance. Each render geometry sends its output to a different window, but the same Geometry Viewer application menu controls all the windows. The module whose output window is currently highlighted in red is the one being controlled. To switch the focus to another render geometry output window, just click in it with any mouse button.

INPUTS

Geometry (required; geometry)

The input data can be any ConvexAVS geometry. More than one geometry can be input to this port. All the geometries will be combined into the same scene.

OUTPUTS

Pixmap (pixmap)

The output is a pixmap containing a scene that includes all the input objects.

PARAMETERS

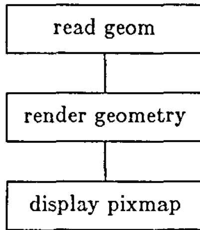
add to object transform

This parameter can be attached to the dialbox allowing these devices to control object transformations. In such cases, you can still control transformations using the mouse:

Mouse	**Transform**
middle	rotate
right	translate in plane of screen
middle with SHIFT key	scale
right with SHIFT key	translate perpendicular to plane of screen

EXAMPLE

This network shows render geometry:



RELATED MODULES

display pixmap, pdb to geom, read geom, render manager

NAME

render_manager – share geometries among subnetworks

SUMMARY

Name render manager

Type renderer

Inputs geometry

Outputs none

Parameters

Name	*Type*
Create New Window	oneshot
Active Windows	choice

DESCRIPTION

The render manager module takes geometries as input, uses the ConvexAVS Geometry Viewer to render them, and displays the results in one or more windows. This module is very similar to the render geometry module, with these differences:

> The render manager module creates its own pixmap and window on the screen rather than relying on the display pixmap module. An initial window is created by default.
>
> The render manager module has a built-in mechanism for creating and selecting output windows. A set of windows is shared among render manager modules in separate subnetworks. At any moment, one of them (the **Current Output** window) is shared by all the render manager modules in all subnetworks. This window displays the combined results of all these modules.

It is possible to create a new output window, which automatically becomes the shared current output window. This provides a powerful capability for exploring differences between datasets or different mappings of the same dataset.

The output window(s) are not destroyed until all render manager modules are destroyed.

INPUTS

Geometry (required; geometry)
Any ConvexAVS geometry.

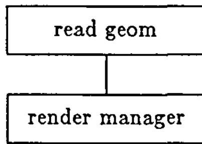
PARAMETERS

Create New Window
Click this button to create a new output window, which becomes the current output window. Subsequent geometric input is rendered into this window until such time as you change the current output window again (perhaps by creating yet another window).

Active Windows
A choice menu that lists all the output windows, showing which one is current. You can also make an output window current by pressing any mouse button in the window itself.

EXAMPLE

This example shows render manager:



RELATED MODULES

display pixmap, pdb to geom, read geom, render manager

NAME

scalar_PLOT3D – calculate derived plot3d scalar functions

SUMMARY

Name scalar PLOT3D

Type filter

Inputs field 3D 3-space irregular 5-vector real
field 3D scalar uniform integer
field 1D scalar uniform float

Outputs field 3D 3-space irregular scalar real

Parameters	*Name*	*Type*
	Plot3d derived function	choice

DESCRIPTION

This module allows you to visualize some of the scalar fields that can be derived from the basic plot3d data. The source code is designed so that each of the functions is implemented separately from the ConvexAVS flow. Each of the functions and their names (for the select button) are stored in a table. In this way, you can extend the functionality of this module beyond a list of functions implemented in the standard plot3d viewer by adding new functions and names. The source for this module is in the /usr/avs/examples/convex/plot3d/modules directory.

INPUTS

Data Field (required; field 3D 3-space irregular 5-vector real)
This input data is the 5-vector field output by read_PLOT3D. It is the most important field because it contains the mesh and solution data.

Blanking Data Field (optional; field 3D scalar uniform integer)
This field contains the blanking records. If a plot3d data set contains blanking records and this is not connected, unpredictable results may occur.

Global Parameters (required; field 1D scalar uniform float)
This field contains the four global parameters (free stream Mach number, angle-of-attack (alpha), Reynolds number, and the time). Not all derived scalar functions make use of this data, but it must be connected so that those derived scalar functions that do require the information can be calculated.

OUTPUTS

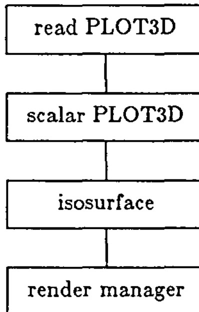
Scalar Field (field 3D 3-space irregular scalar real)
A scalar field representing the function selected.

PARAMETERS

Plot3d derived function
A choice parameter that selects the desired scalar function of the plot3d values to calculate. Implemented functions are: pressure, temperature, enthalpy, internal energy, Mach number, and entropy.

EXAMPLE

This example reads in a plot3d data set and does an isosurface on one of the derived scalar fields:



For additional examples, refer to the read PLOT3D and vector PLOT3D manual pages.

RELATED MODULES
density PLOT3D, momentum PLOT3D, stagnation PLOT3D, read PLOT3D, vector PLOT3D

RELATED PROGRAMS
export_PLOT3D and import_PLOT3D

NAME

scatter_dots – generate spheres at points in 3D space

SUMMARY

Name scatter dots

Type mapper

Inputs field 1D real 3-space irregular

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Choices*
Connect the dots	toggle	off			on, off
Radius	Real	0.01	0.0	1.0	

DESCRIPTION

The scatter dots module generates spheres at the coordinate locations in a specified field. For a scalar field, the radii of the spheres are determined only by the value of the sphere-radius widget, and the spheres are all colored white. If the field is a 4-vector float (such as that produced by the bubbleviz module), only the first element of the vector determines the sphere's radius. The other three elements are interpreted as red-green-blue color values (normalized to the range 0..1).

INPUTS

Point List (required; field 1D real 3-space irregular)
The input field must be a list of points in 3D space with a float value specified at each point (a "scatter" field).

OUTPUTS

Geometry (geometry)
The output is a ConvexAVS geometry.

PARAMETERS

Connect the dots

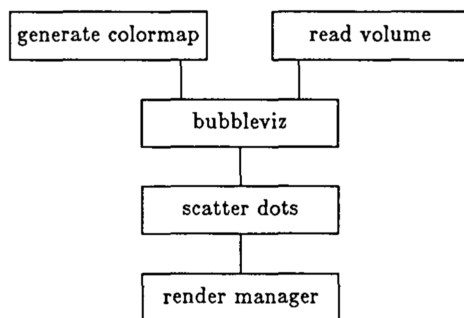
If **OFF**, a sphere is drawn at each point in the field. The radius of the sphere is specified by the field element's scalar data value. (If the field has vector data, the value of the first vector element is used and the other values are discarded.)

If **ON**, the points are represented as dots, connected with a single polyline (in the order specified by the 1D array). If the input field has 4-vector float data, the last three vector elements are interpreted as red-green-blue values, and the dots are assigned colors. No spheres are drawn.

Radius A proportional scaling value for each sphere's radius.

EXAMPLE

This network shows scatter dots:



RELATED MODULES

geom to scatter, read geom, render geometry, tube, wireframe

NAME

shrink – make an object smaller

SUMMARY

Name shrink

Type filter

Inputs geometry

Outputs geometry

Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	offset	float	0.0	none	none

DESCRIPTION

The shrink module transforms a ConvexAVS geometry, so that each vertex of each polygon is translated towards (or away from) the geometry's centroid (center of gravity). It is useful for visualizing the internal geometry of an object.

INPUTS

Geometry (required; geometry)
A ConvexAVS geometry, created with the libgeom library or by another module.

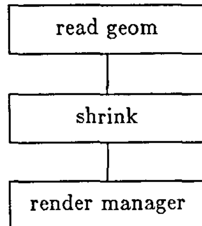
OUTPUTS

Geometry (geometry)
A geometry that represents the same object(s) as the input data.

PARAMETERS

offset The amount by which each vertex is translated. Positive values collapse the geometry inward. Negative values enlarge the geometry.

EXAMPLE



RELATED MODULES

flip normal, read geom, render geometry, tube

LIMITATIONS

This module works only for polytriangle strips and meshes; it does not work for polyhedra.

This module doesn't copy UV data.

NAME

stagnation_PLOT3D – strip out the plot3d stagnation energy

SUMMARY

Name stagnation PLOT3D

Type filter

Inputs field 3D 3-space irregular 5-vector real

Outputs field 3D 3-space irregular scalar real

Parameters none

DESCRIPTION

This module allows you to visualize the stagnation energy field in the plot3d file. It merely converts the 5-vector into the stagnation energy scalar field.

INPUTS

Data Field (required; field 3D 3-space irregular 5-vector real)

This input data is the 5-vector field output by read PLOT3D. It is the most important field because it contains the mesh and solution data.

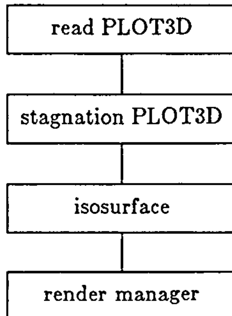
OUTPUTS

Scalar Field (field 3D 3-space irregular scalar real)

A scalar field representing the stagnation energy field from the plot3d file read by read PLOT3D.

EXAMPLE

This example reads in a plot3d data set and does an isosurface on the stagnation field. Notice how only the plot3d field itself is used. We do not use the blanking records because we are extracting a part of the raw plot3d field.



For additional examples, refer to the read PLOT3D and vector PLOT3D manual pages.

RELATED MODULES

density PLOT3D, momentum PLOT3D, read PLOT3D, scalar PLOT3D, vector PLOT3D

RELATED PROGRAMS

export_PLOT3D and import_PLOT3D

NAME

stream_lines – generate stream lines for a vector field

SUMMARY

Name stream lines

Type mapper

Inputs field 3D 3-vector uniform float

Outputs
geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
width	integer	12	4	32
length	integer	12	4	128
step	float	0.02	0.0	1.0
position	trackball			

DESCRIPTION

The stream lines module generates stream lines based on a field that is a volume of 3D vectors. It places a straight line — a set of collinear points — at a starting location in the volume. (Both the location of the number of points are parameter-controlled.) Then, for every time step, it advances each point through space based on the interpolated value of the vector field at its present position. The result is a set of stream lines showing the progress of massless particles through a vector field.

This module is similar to the particle advector module, except that its source points define a line instead of a square region, and the result is a static set of lines instead of a dynamically updated set of spheres.

This module is also similar to stream mesh except that it produces lines instead of a polygonal mesh.

INPUTS

Data Field (required; field 3D 3-vector uniform float)
The input field must be 3D, and the data for each field element must be 3-vector uniform floats representing the components of a velocity vector.

OUTPUTS

Geometry (geometry)
A set of disjoint lines.

PARAMETERS

width The number of points in the line placed within the field. This number of stream lines will result.

length A scale factor, which multiplies the length of the streamline segments generated during each time step.

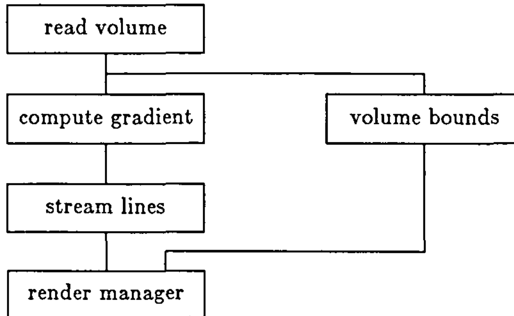
step Determines the time step for the interactive computation. The larger the value, the greater the interval.

position Determines the initial orientation of the line of points within the volume. The mouse controls the position using the same "virtual trackball" paradigm used by the Geometry Viewer (render geometry module):

Mouse	Transform
middle	rotate
right	translate in plane of screen
middle with SHIFT key	scale
right with SHIFT key	translate perpendicular to plane of screen

EXAMPLE

This network shows one way to use stream lines:



RELATED MODULES

hedgehog, particle advector, stream mesh

NAME

stream_mesh – generate stream lines for a vector field as a polygonal mesh

SUMMARY

Name stream mesh

Type mapper

Inputs field 3D 3-vector uniform float

Outputs
geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
width	integer	12	4	32
length	integer	12	4	128
step	float	0.02	0.0	1.0
position	trackball			

DESCRIPTION

The stream mesh module generates stream lines based on a field that is a volume of 3D vectors. It connects the stream lines into a geometric primitive called a polygonal mesh, which is optimized for fast rendering.

This module is essentially similar to stream lines, except that the output is a polygonal mesh rather than a set of disjoint lines.

INPUTS

Data Field (required; field 3D 3-vector uniform float)
The input field must be 3D, and the data for each field element must be 3-vector uniform floats representing the components of a velocity vector.

OUTPUTS

Geometry (geometry)
A polygonal mesh.

PARAMETERS

width The number of points in the line placed within the field. This number of stream lines will result.

length A scale factor, which multiplies the length of the streamline segments generated during each time step.

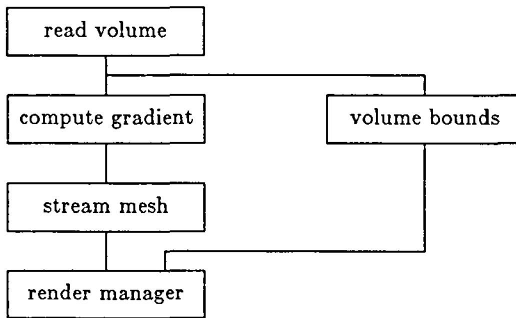
step Determines the time step for the interactive computation. The larger the value, the greater the interval.

position Determines the initial orientation of the line of points within the volume. The mouse controls the position using the same "virtual trackball" paradigm used by the Geometry Viewer (render geometry module):

Mouse	**Transform**
middle	rotate
right	translate in plane of screen
middle with SHIFT key	scale
right with SHIFT key	translate perpendicular to plane of screen

EXAMPLE

This network shows one way to use stream mesh:



RELATED MODULES

hedgehog, particle advector, stream lines

NAME

threshold – restrict values in data field

SUMMARY

Name threshold

Type filter

Inputs field *any-dimension any-data*

Outputs
field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
thresh_min	float	0.0	none	none
thresh_max	float	255.0	none	none

DESCRIPTION

The threshold module transforms the values of a field as follows:

Any value less than the value of the **thresh_min** parameter is set to 0.

Any value greater than the value of the **thresh_max** parameter is set to 0.

All values within the **thresh_min-to-thresh_max** range are not changed.

Note the difference between the clamp and threshold modules:

The threshold module sets values outside the specified range to be zero.

The clamp module sets values outside the specified range to be the range's minimum and maximum values.

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

Data Field
The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced.

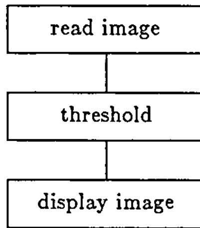
PARAMETERS

thresh_min
The minimum threshold value.

thresh_max
The maximum threshold value.

EXAMPLE

This example shows threshold in a network:



RELATED MODULES

clamp, colorizer, read volume

NAME

transpose – exchange dimensions in a 2D or 3D data set

SUMMARY

Name transpose

Type filter

Inputs field *any-dimension any-data*

Outputs field of same type as input

Parameters

Name	*Type*	*Default*	*Choices*
Axis	choice	Original	Original, YZ, XZ, XY

DESCRIPTION

The transpose module exchanges the data in two dimensions of a 2D or 3D field. It can be used to change the orientation of the data for display and/or processing purposes.

INPUTS

Data Field (required; field *any-dimension any-data*)
The input data may be any ConvexAVS field.

OUTPUTS

Data Field
The output field has the same dimensionality and type as the input field.

PARAMETERS

Axis The choices for exchanging the data are:

Original Copies the input to the output; no transformation is performed.

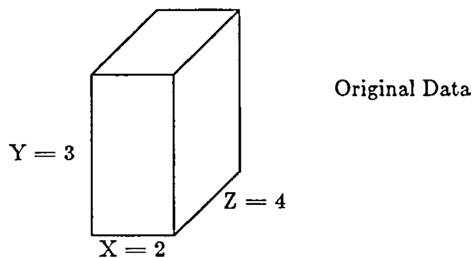
YZ Swaps the Y and Z dimensions. (Equivalent to "Original" for a 2D field.)

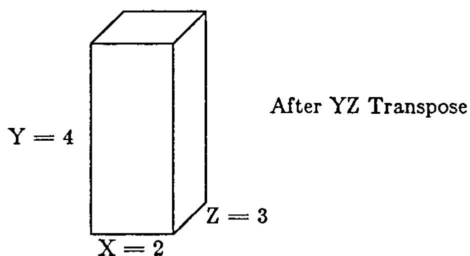
XZ Swaps the X and Z dimensions. (Equivalent to "Original" for a 2D field.)

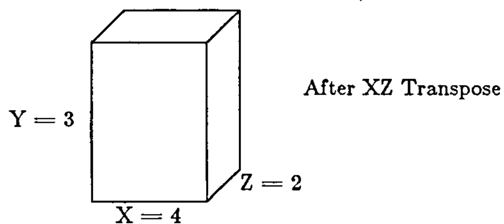
XY Swaps the X and Y dimensions.

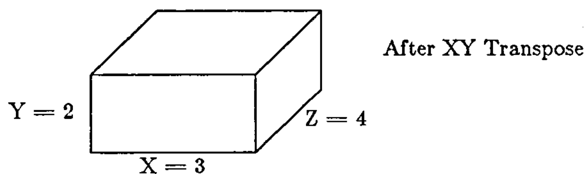
EXAMPLE 1

The following drawings illustrate the transposition choices:



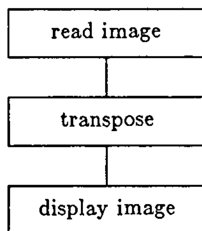






EXAMPLE 2

This example shows transpose in a network:



RELATED MODULES

mirror

NAME

tube – convert lines to cylindrical tubes

SUMMARY

Name tube

Type filter

Inputs geometry

Outputs
geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
radius	float	0.0	none	none

DESCRIPTION

The tube module replaces a set of disjoint lines with "tubes" constructed out of eight polygons.

INPUTS

Geometry (required; geometry)
A ConvexAVS geometry, created with the libgeom library or by another module.

OUTPUTS

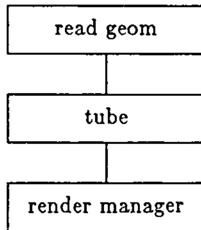
Geometry (geometry)
The output is a ConvexAVS geometry with each input line represented as a set of polygons.

PARAMETERS

radius The radius to be used for the tube.

EXAMPLE

This network shows one use for tube:



RELATED MODULES

flip normal, offset, read geom, render geometry, shrink, wireframe

LIMITATIONS

Only **radius** values in the range 0.0 - 0.4 produce acceptable results.

The cylinders are not capped and adjacent line segments are not joined. For thick cylinders, there may be quite a bit of surface intersections at the joins.

NAME

vbuffer – perform volumetric rendering on volume data

SUMMARY

Name vbuffer

Type renderer

Inputs field 3D scalar uniform
colormap

Outputs
field 2D 4-vector byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Xrotation	real	0.0	-180	180
Yrotation	real	0.0	-180	180
Zrotation	real	0.0	-180	180
ScaleX	real	0.5	0.0	5.0
ScaleY	real	0.5	0.0	5.0
ScaleZ	real	0.5	0.0	5.0
Ztrans	real	-2.5	-10	10
Imagesize	int	200	10	1024
Background	real	0.0	0.0	1.0
Cell-Based	logical	off	off	on

DESCRIPTION

The vbuffer module creates a volumetric rendering of a 3D uniform scalar field, using RGB color and opacity transfer functions. The technique employed uses two levels of interpolation:

A cell-averaged or voxel representation produces lower quality and lower apparent resolution images at moderate execution speeds.

A trilinear or cell-based interpolation results in very high quality images with a slower execution time.

You can choose between two rendering algorithms:

Trilinear Interpolation:
This algorithm uses an inverse-mapping scheme to generate images. The 3D field data is decomposed into 8-node *cells*, with one field value at each vertex of the cell. Each cell is processed by accumulating color and opacity along an integration volume which maps into one or more pixels.

At several locations through this volume, both the value of the scalar field and its gradient are interpolated. The sampled scalar field value is used to map into transfer functions, which determine the color and opacity at the point. The color for the pixel is determined by a product of the diffuse illumination (due to the dot product of the light source and gradient vectors), the opacity, and the sampled color as accumulated along the volume. After all the pixels that the cell projects into are processed, the algorithm moves on to the next cell. Partial pixel contributions are saved into an in-memory frame buffer.

Voxel Approximation (default):
The 3D field is decomposed into cells, as described above. But no interpolation is performed within the cell. Each cell has a single opacity, color, texture color, surface gradient, and set of

shading parameters. This method is much faster than the trilinear interpolation method. Use it to get a quick (albeit ragged) look at the data. It is most useful for selecting the opacity and color transfer functions.

INPUTS

Data Field (required; field 3D scalar uniform)
The input field must be 3-dimensional. The data for each field element must be a scalar.

Colormap (required; colormap)
Any ConvexAVS colormap.

OUTPUTS

Data Field (field 2D 4-vector byte)
Outputs a 2D image rendered by vbuffer.

PARAMETERS

Xrotation The X-rotation of the data field in degrees.

Yrotation The Y-rotation of the data field in degrees.

Zrotation The Z-rotation of the data field in degrees.

ScaleX The X-scale factor of the data field. By default the highest resolution axis of the data is scaled between -1.0 and 1.0, and other axes are scaled accordingly by this.

ScaleY The Y-scale factor of the data field.

ScaleZ The Z-scale factor of the data field.

Ztrans The Z-translation of the field. The coordinate system is right handed, so only data that has a negative Z coordinate will be visible.

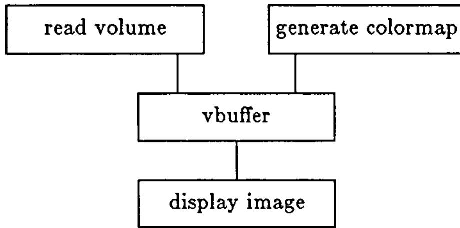
Imagesize The resolution in pixels of the computed image. The image is always square. The algorithm complexity scales with the number of pixels and the number of cells which have a non-zero opacity.

Background
The background brightness. Zero corresponds to a black background, 1.0 to a white background.

Cell-Based
A toggle switch between the voxel approximation algorithm (default) and the trilinear interpolation algorithm (cell-based and substantially slower).

EXAMPLE

This example shows a simple network with vbuffer:



RELATED MODULES

clamp, contrast, dot surface, histogram stretch, interpolate, isosurface, read volume, threshold

LIMITATIONS

Only uniform fields can be rendered by vbuffer.

Some image anomalies can occur along cell boundaries. This is view-orientation dependent.

The execution time can be dramatically reduced by limiting the number of cells that contain a non-zero amount of opacity.

NAME

vector_PLOT3D – calculate derived plot3d vector functions

SUMMARY

Name vector PLOT3D

Type filter

Inputs field 3D 3-space irregular 5-vector real
field 3D scalar uniform integer
field 1D scalar uniform float

Outputs field 3D 3-space irregular 3-vector real

Parameters

Name	*Type*
Plot3d derived function	choice

DESCRIPTION

This module allows you to visualize some of the vector fields that can be derived from the basic plot3d data. The source code is designed so that each of the functions is implemented separately from the ConvexAVS flow. Each of the functions and their names (for the select button) are stored in a table. In this way, you can extend the functionality of this module beyond a list of functions implemented in the standard plot3d viewer by adding new functions and names. The source for this module is in the /usr/avs/examples/convex/plot3d/modules directory.

INPUTS

Data Field (required; field 3D 3-space irregular 5-vector real)
This input data is the 5-vector field output by read PLOT3D. It is the most important field because it contains the mesh and solution data.

Blanking Data Field (optional; field 3D scalar uniform integer)
This field contains the blanking records. If a plot3d data set contains blanking records and this is not connected, unpredictable results may occur.

Global Parameters (required; field 1D scalar uniform float)
This field contains the four global parameters (free stream Mach number, angle-of-attack (alpha), Reynolds number, and the time). Not all derived vector functions make use of this data, but it must be connected up so that those derived vector functions that do require the information can be calculated.

OUTPUTS

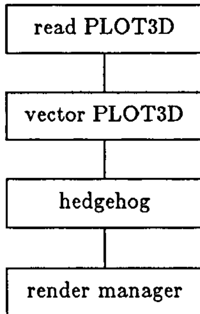
Vector Field (field 3D 3-space irregular 3-vector real)
A vector field representing the function selected.

PARAMETERS

Plot3d derived function
A choice parameter that selects the desired vector function of the plot3d values to calculate. Implemented functions are: velocity and perturbation velocity.

EXAMPLE

This example reads in a plot3d data set and does a hedgehog on one of the derived vector fields.



For additional examples, refer to the read PLOT3D and scalar PLOT3D manual pages.

RELATED MODULES

density PLOT3D, momentum PLOT3D, stagnation PLOT3D, read PLOT3D, scalar PLOT3D

RELATED PROGRAMS

export_PLOT3D and import_PLOT3D

NAME

vector_curl – compute the curl of a vector field

SUMMARY

Name vector curl

Type filter

Inputs field 3D 3-vector float

Outputs
field 3D 3-vector float

Parameters
none

DESCRIPTION

The vector curl module accepts a vector field as input and computes the curl of that field as output. This is related to the divergence as follows:

curl = (*DEL* × *F*)
div = (*DEL* • *F*)

where *F* is the vector input field.

The equation used to compute the curl is:

```
new_dx[X][Y][Z] = (dz[X][Y+1][Z] - dz[X][Y-1][Z]) -
                  (dy[X][Y][Z+1] - dy[X][Y][Z-1])

new_dy[X][Y][Z] = (dx[X][Y][Z+1] - dx[X][Y][Z-1]) -
                  (dz[X+1][Y][Z] - dz[X-1][Y][Z])

new_dz[X][Y][Z] = (dy[X+1][Y][Z] - dy[X-1][Y][Z]) -
                  (dx[X][Y+1][Z] - dx[X][Y-1][Z])
```

INPUTS

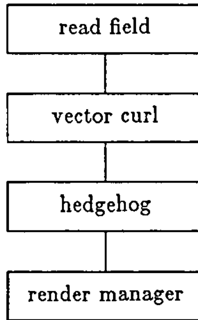
Data Field (required; field 3D 3-vector float)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

OUTPUTS

Data Field (field 3D 3-vector float)
The output field is in the same format as the input field.

EXAMPLE

This example uses vector curl in a network:



RELATED MODULES
gradient shade, hedgehog, particle advector, stream lines, stream mesh, vector div, vector grad, vector norm, vector mag

LIMITATIONS
This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis where each 3-vector of floats represents the components of a velocity or a gradient.

NAME

vector_div – compute the divergence of a vector field

SUMMARY

Name vector div

Type filter

Inputs field 3D 3-vector float

Outputs
field 3D scalar float

Parameters
none

DESCRIPTION

The vector div module accepts a vector field as input and computes the divergence of that field as output. This is related to the curl as follows:

$$curl = (DEL \times F)$$
$$div = (DEL \bullet F)$$

where F is the vector input field.

The equation used to compute the divergence is:

```
divergence[X][Y][Z] = (dx[X+1][Y][Z] - dz[X-1][Y][Z]) +
                      (dy[X][Y+1][Z] - dy[X][Y-1][Z]) +
                      (dz[X][Y][Z+1] - dz[X][Y][Z-1])
```

INPUTS

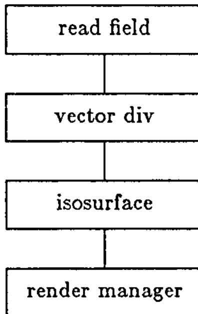
Data Field (required; field 3D 3-vector float)
The input field must represent a volume of elements with a 3D vector of floating-point data values for each element.

OUTPUTS

Data Field (field 3D scalar float)
The output field has a single floating-point value for each input field element.

EXAMPLE

This example uses vector div in a network:



RELATED MODULES

gradient shade, hedgehog, particle advector, stream lines, stream mesh, vector curl, vector grad, vector norm, vector mag

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis where each 3-vector of floats represents the components of a velocity or a gradient.

NAME

vector_grad – compute the vector gradient of a scalar field

SUMMARY

Name vector grad

Type filter

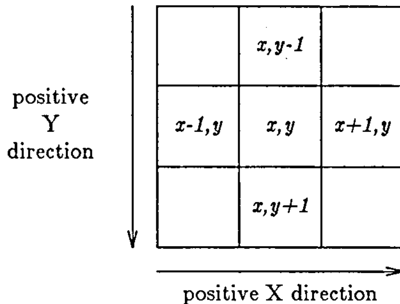
Inputs field 3D scalar float

Outputs
field 3D 3-vector float

Parameters
none

DESCRIPTION

The vector grad module computes the gradient of a 3D field. The gradient is treated by some other modules as a "pseudo-normal" to the "surface" for each data element. A "nearest neighbor" algorithm is used to compute the gradient: the difference between the next data value (in each direction) and the previous data value. In two dimensions, this can be represented as follows:



Delta_x[X][Y][Z] = data[X+1][Y][Z] - data[X-1][Y][Z]
Delta_y[X][Y][Z] = data[X][Y+1][Z] - data[X][Y-1][Z]
Delta_z[X][Y][Z] = data[X][Y][Z+1] - data[X][Y][Z-1]

This module is identical to the compute gradient module, except that it does not normalize the output. The compute gradient module is designed for gradient shading fields, whereas this module is designed for input into the other vector field modules. Note that vector grad followed by vector norm produces the same results as compute gradient.

INPUTS

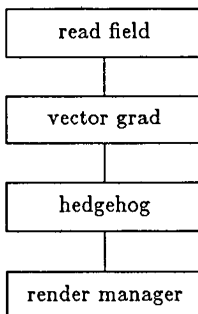
Data Field (required; field 3D scalar float)
The input field must represent a volume of elements with a 3D scalar of floating-point data values for each element.

OUTPUTS

Data Field (field 3D 3-vector float)
The output field has three floating-point values for each input field element and represents the gradient of the input field.

EXAMPLE

This example uses vector curl in a network:



RELATED MODULES

hedgehog, particle advector, stream lines, stream mesh, vector curl, vector div, vector mag, vector norm

LIMITATIONS

There may be algorithms better than "nearest-neighbor" for computing the gradient.

This module produces 12 bytes per pixel (voxel). For example, a 128 by 128 by 128 byte volume is about 2.1 MB before the gradient is computed. The compute gradient module produces a 25.2 MB internal data set from this data. This will have an adverse performance effect on systems whose physical memory is 32 MB or less.

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis where each 3-vector of floats represents the components of a velocity or a gradient.

NAME

vector_mag – compute the magnitude of a vector field

SUMMARY

Name vector mag

Type filter

Inputs field 3D 3-vector float

Outputs

field 3D scalar float

Parameters

none

DESCRIPTION

The vector mag module accepts a vector field as input and computes the magnitude of each vector data value. The output is a scalar field consisting of the magnitudes.

The magnitude equation is:

```
Magnitude[X][Y][Z] = sqrt((dx[X][Y][Z]*dx[X][Y][Z]) +
                          (dy[X][Y][Z]*dy[X][Y][Z]) +
                          (dz[X][Y][Z]*dz[X][Y][Z]) )
```

INPUTS

Data Field (required; field 3D 3-vector float)

The input field must represent a volume of elements with a 3D vector of floating-point data values for each element.

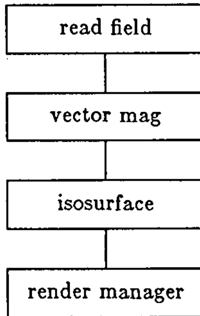
OUTPUTS

Data Field (field 3D scalar float)

The output field has a single floating-point value for each input field element.

EXAMPLE

This example uses vector mag in a network:



RELATED MODULES

gradient shade, hedgehog, particle advector, stream lines, stream mesh, vector curl, vector div, vector grad, vector norm

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis where each 3-vector of floats represents the components of a velocity or a gradient.

NAME

vector_norm – normalize a vector field

SUMMARY

Name vector norm

Type filter

Inputs field 3D 3-vector float

Outputs
field 3D 3-vector float

Parameters
none

DESCRIPTION

The vector norm module accepts a vector field as input and produces a normalized version of that vector field as output. The normalization equation looks like:

```
Magnitude  = sqrt((dx*dx) +  (dy*dy) + (dz*dz))
New_dx     = dx / Magnitude
New_dy     = dy / Magnitude
New_dz     = dz / Magnitude
```

INPUTS

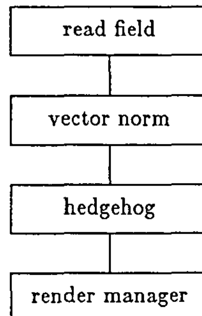
Data Field (required; field 3D 3-vector float)
The input field must represent a volume of elements with a 3D vector of floating-point data values for each element.

OUTPUTS

Data Field (field 3D 3-vector float)
The output field is in the same format as the input field.

EXAMPLE

This example uses vector norm in a network:



RELATED MODULES

gradient shade, hedgehog, particle advector, stream lines, stream mesh, vector curl, vector div, vector grad, vector mag

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis where each 3-vector of floats represents the components of a velocity or a gradient.

NAME

volume_bounds – generate bounding box of 3D 3-vector field

SUMMARY

Name volume bounds

Type mapper

Inputs field 3D *any-data*

Outputs
geometry

Parameters

Name	*Type*
Bounding Box	toggle
Hull	toggle
Min I	toggle
Max I	toggle
Min J	toggle
Max J	toggle
Min K	toggle
Max K	toggle

DESCRIPTION

The volume bounds module generates lines that indicate the "bounding box" of a 3D data set (field). It is frequently used in conjunction with other geometry-based volume-visualization modules (for example, bubbleviz, isosurface, hedgehog, arbitrary slicer), because it provides some volumetric context for the data.

INPUTS

Data Field (required; field 3D *any-data*)
The input data must be a 3D field but may have any kind of data at each location in the field.

OUTPUTS

Geometry (geometry)
The output geometry consists of the lines that form the bounding box. This box can take several forms.

PARAMETERS

Bounding Box
If **ON**, the edges of a rectangular solid are drawn and show the extent of the data in Cartesian space.

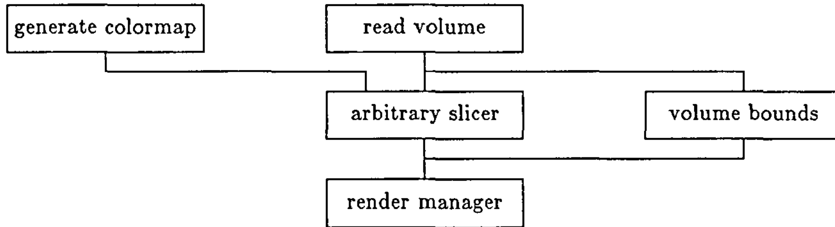
Hull (for rectilinear and irregular input fields only) If **ON**, draws the edges of the bounding box but deforms these edges in accordance with the coordinate data supplied in the input field. That is, volume bounds draws the bounding box in *physical space* rather than *computational space*.

Min I
Max I
Min J
Max J
Min K
Max K These toggle switches provide further help is visualizing the way the computational space is mapped into physical space. Each one fills in one of the six faces of the hull. For example, turning on **Min I** draws a mesh showing the 2D slice of field elements

with the minimum index value in the first dimension; turning on **Max K** draws a mesh showing the 2D slice of field elements with the maximum index value in the third dimension.

EXAMPLE

The following network shows one way to use volume bounds:



RELATED MODULES

read volume, volume manager

NAME

volume_manager – share volumes among subnetworks

SUMMARY

Name volume manager

Type data

Inputs none

Outputs field 3D scalar byte

Parameters

Name	*Type*	*Choices*
VOLUMGR Select	choice	Select, Replace
Volume Manager	browser	
Volume Choices	choice	

DESCRIPTION

The volume manager module reads an volume file from disk and outputs the volume as a "field 3D scalar byte." It works like the read volume module, except that it has both a caching mechanism and a way of sharing data among volume manager modules in separate subnetworks.

Refer to the read volume manual page for a description of the volume format.

OUTPUTS

Data Field (field 3D scalar byte)

The output is the byte data cast as the scalar data in a 3D field.

PARAMETERS

VOLUMGR Select

A choice that determines how newly-read volumes will be placed to the list of currently active volumes:

If **Select** is chosen, a new volume is added to the end of the list.

If **Replace** is chosen, a new volume replaces the currently selected member on this list.

In either case, the change is reflected in all the volume manager modules in all active subnetworks.

Volume Manager

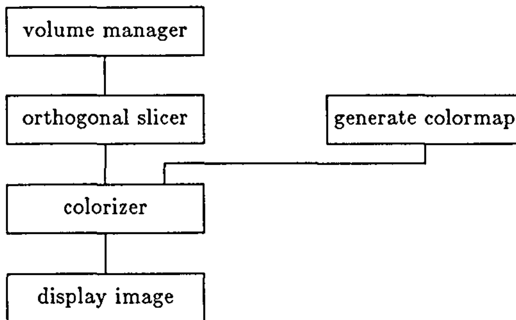
A file browser that allows you to select a volume file to read.

Volume Choices

A set of choices, listing each of the currently active volumes.

EXAMPLE

The following network shows volume manager:



RELATED MODULES

arbitrary slicer, bubbleviz, clamp, colorizer, compute gradient, contrast, crop, display image, dot surface, downsize, field to byte, field to double, field to float, field to int, field to mesh, generate colormap, gradient shade, histogram stretch, interpolate, isosurface, mirror, offset, orthogonal slicer, render geometry, transpose, vbuffer, volume bounds

LIMITATIONS

The cached volumes are not freed until all volume manager modules are destroyed. Because volume data can be large, caching multiple volume data sets can use up a lot of memory.

NAME

wireframe – convert object from surface to wireframe representation

SUMMARY

Name wireframe

Type filter

Inputs geometry

Outputs
geometry

Parameters
none

DESCRIPTION

The wireframe module replaces all surfaces defined as polytriangle strips with wireframe representations. This is useful for constructing a wireframe version of an object that has been defined as a shaded surface.

INPUTS

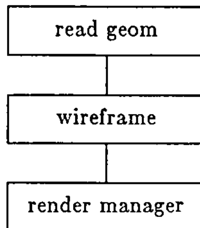
Geometry (required; geometry)
Any ConvexAVS geometry, created with the libgeom library or produced by another module.

OUTPUTS

Geometry (geometry)
A geometry that represents the same object as the input data.

EXAMPLE

This example shows the use of wireframe:



RELATED MODULES

flip normal, offset, read geom, render geometry, shrink, tube

LIMITATIONS

The wireframe module generates lines based on the order of the vertices of a polytriangle strip. Sometimes, the resulting object is not exactly what you want. It may have "cobwebs" and other data inconsistencies (usually invisible) of the original polytriangle strip. You may need to regenerate the original data in order to produce the desired wireframe representation.

NAME

write_field – write a field description to disk

SUMMARY

Name write field

Type renderer

Inputs field

Outputs none

Parameters *Name* *Type*
Write Field browser

DESCRIPTION

The write field module writes a ConvexAVS field description to disk. The field format on disk includes two parts, an ASCII header and a binary area. This format is described in detail in the read field manual page.

INPUTS

Data Field (required; field)
The input can be a ConvexAVS field.

PARAMETERS

Write Field

A file browser that allows you to specify the name of the field file to be created. The file suffix .fld is appended to the name automatically. If the file already exists, write field issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

After the field file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

EXAMPLE 1

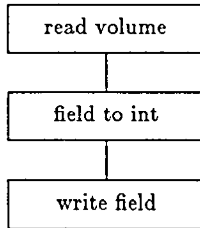
Following is an example of a file produced by write field:

```
ndim=3          # number of dimensions in the field
dim1=64         # dimension of axis 1
dim2=64         # dimension of axis 2
dim3=64         # dimension of axis 3
nspace=3        # number of physical coordinates per point
veclen=1        # number of components at each point
data=byte       # data type (byte, integer, float, double)
field=uniform   # field type (uniform, rectilinear, irregular)

262,144 bytes of data
```

The field has three dimensions: 64 by 64 by 64. There is a single byte at each point, and the field is uniform.

EXAMPLE 2

This example shows write field in a network:



RELATED MODULES

bubbleviz, clamp, colorizer, combine scalars, compute gradient, contrast, crop, dot surface, downsize, extract scalar, field to byte, field to double, field to float, field to int, geom to scatter, gradient shade, histogram stretch, image manager, interpolate, mirror, orthogonal slicer, pixmap to image, read image, read volume, threshold, transpose, vector curl, vector div, vector grad, vector mag, vector norm, volume manager

LIMITATIONS

The write field module produces an error message if it cannot open the file or if there is not enough space to write the complete file.

NAME

write_image – store image data in a file

SUMMARY

Name write image

Type renderer

Inputs field 2D 4-vector byte

Outputs
none

Parameters

Name	*Type*
Write Image	browser

DESCRIPTION

The write image module writes a ConvexAVS image data structure to a file. This structure takes the form of a "field 2D 4-vector byte." Refer to the read image manual page for a detailed description of the image format.

INPUTS

Data Field (required; field 2D 4-vector byte)
The input can be any ConvexAVS image.

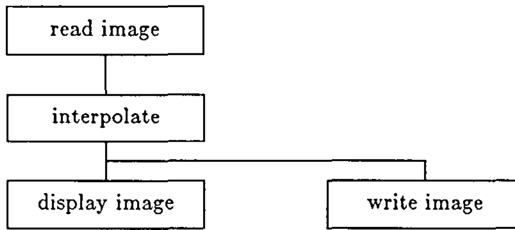
PARAMETERS

Write Image
A file browser that allows you to specify the name of the image file to be created. The file suffix .x is appended to the name automatically. If the file already exists, write image issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

After the image file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

EXAMPLE

This network shows write image:



RELATED MODULES

clamp, combine scalars, contrast, display image, extract scalar, histogram stretch, interpolate, pixmap to image, render geometry, threshold

NAME

write_volume – write volume data to a file

SUMMARY

Name write volume

Type renderer

Inputs field 3D scalar byte

Outputs
none

Parameters

Name	*Type*
Write Volume	browser

DESCRIPTION

The write volume module writes volume data to a file. The volume is in the ConvexAVS format "field 3D scalar byte." The data format on disk is:

1 byte: number of voxels in X
1 byte: number of voxels in Y
1 byte: number of voxels in Z
nx * ny * nz * 1 byte: voxel data

Each time the file is written, the filename is reset to NULL. This prevents successive changes upstream in the network to automatically trigger a volume data file to be written. A new filename must be entered each time the file is to be written out.

This module is commonly used to pre-process a volume database for later use. For example, the input data might be very low-contrast. You could construct a network that includes the contrast module and the write volume module. Once you select appropriate settings for the contrast, the data could be written to a file and used later for other types of processing.

INPUTS

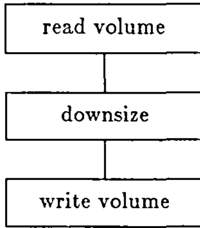
Data Field (required; field 3D scalar byte)
The input data must be a 3D field with a byte value at each location in the field.

PARAMETERS

Write Volume
A file browser that allows you to specify the name of the volume data file to be created. The file suffix .dat is appended to the name automatically. If the file already exists, write volume issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

EXAMPLE

The following example shows write volume:

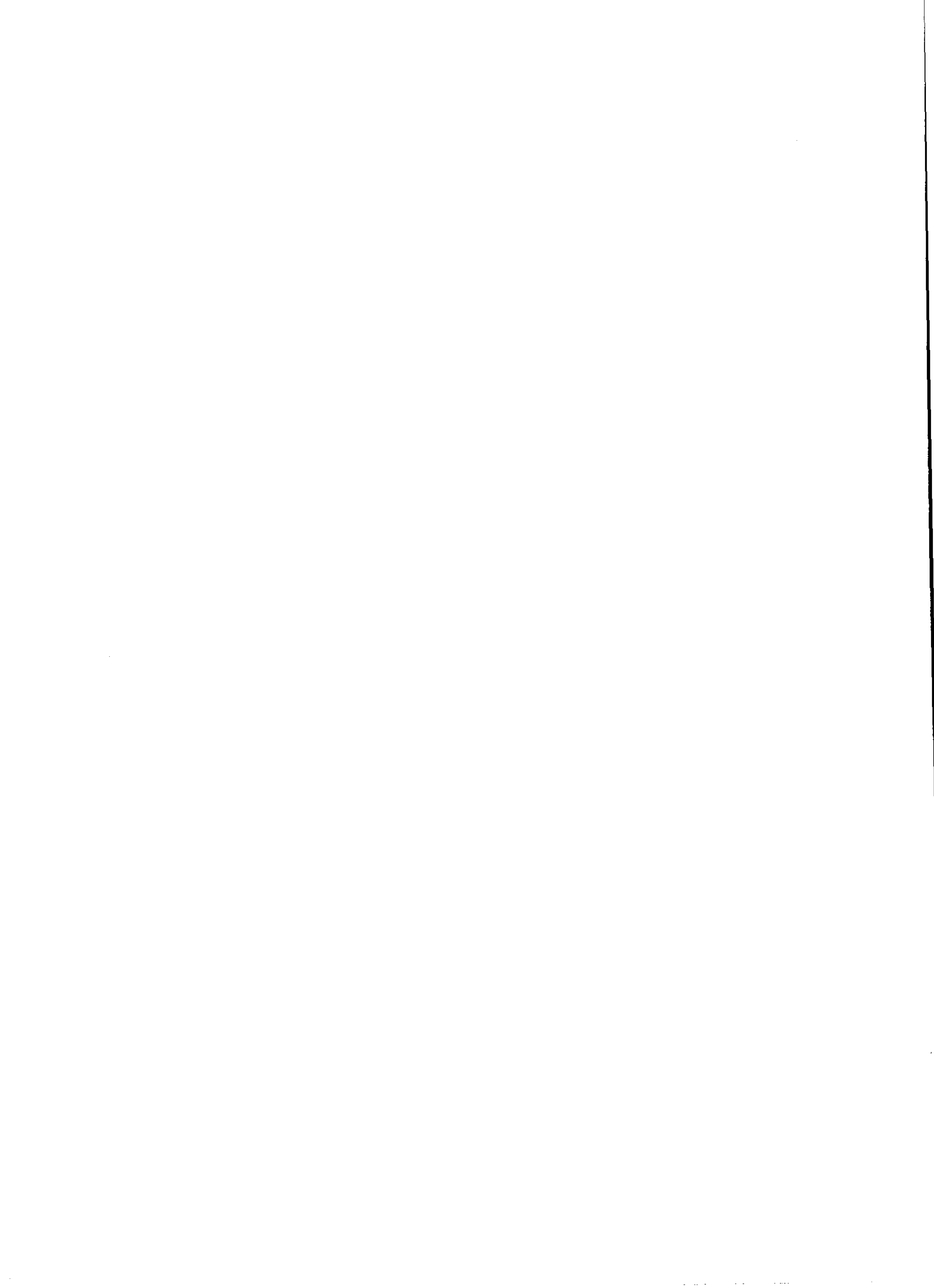


RELATED MODULES

clamp, contrast, crop, downsize, histogram stretch, interpolate, mirror, read volume, threshold, transpose

LIMITATIONS

The format of volume databases on disk is severely limiting. The dimensions are restricted to a maximum of 256 in X, Y, and Z. The data also must be in the range 0 - 255.



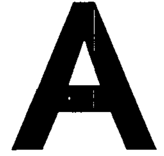
Appendixes

This part contains useful reference information.

- Geometry Conversion Programs
- The Geometry Viewer Script Language
- ConvexAVS File Formats
- Memory Management
- Module Routines
- C Language Field Macros
- Geometry Routines
- Module Code Examples
- Creating Online Help Files for your Modules

- Index

Geometry Conversion Programs



The ConvexAVS Geometry Viewer reads and writes graphical objects in a data format called *geom*. The Convex Application Visualization System includes a filter facility, which enables conversion of data in other formats to the geom format. The filter facility has several usage levels:

- Convex supplies a set of filter utilities. The ConvexAVS viewing application sometimes invokes one of these automatically when the user executes the Read Object function. Each filter converts data in a particular format to the *geom* format.

  During a Read Object, if ConvexAVS determines (using the file name extension) that the file you select is in a known data format, it invokes the appropriate filter automatically to create a corresponding geom file on disk. It then reads in the object from the *geom* file.

- You can also execute the filter utilities from the shell, outside the ConvexAVS viewing application.

- A set of source code templates for creating new filter utilities. Each template handles the conversion of a particular kind of object to the *geom* format. The templates are written in C and in FORTRAN.

  For simple data, you may be able to use one of these templates directly. For more complex data, use one of the templates as a starting point for creating a customized filter utility.

- The GEOM library, consisting of the ConvexAVS-defined object types and C-language procedures, should be used in writing new filter utilities. This library is described in *geom(3V)*.

Automatic Data Filtering

When you execute the ConvexAVS function Read Object, you select a filename in the File Browser window. If the file has a *.geom* or *.obj* extension, ConvexAVS reads it in directly.

If the file has another extension, ConvexAVS determines whether the file contains data in a known format, as follows:

- It looks in its filters directory for utility programs named *..._to_geom*. Each such filter determines a particular filename extension. For instance, the filter utility **pdb_to_geom** determines the extension *.pdb* or *.ent*.
- It compares the extension of the **Read Object** file name with its list of filter extensions.
- If there is a match, ConvexAVS automatically invokes the appropriate filter utility, creating a *geom*-format file of the same name, with a .geom extension.
- The **Read Object** function is completed by reading in the newly created *.geom* file.

You must have write permissions in the data file directory.

Table A-1 lists the filter utilities that Convex supplies in directory */usr/avs/bin*. (You can specify an alternative filters directory when you start ConvexAVS, by using the -**filter** *dirname* option.)

Table A-1
Convex- supplied Filter Utilities

Filter Name	Extension	Data Format
ts_to_geom	.ts	Mathematica ThreeScript
wfront_to_geom	.wfront	Wavefront
byu_to_geom	.byu	Movie BYU
pdb_to_geom	.pdb	Polygen protein data bank
	.ent	Brookhaven protein data bank
ppoly_to_geom	.ppoly	UNC
pobj_to_geom	.pobj	Convex internal format
objs_to_geom	.objs	Convex internal format

Shell-Level Usage of Filter Utilities

Each of the filters listed in the table above can be invoked from the shell. Each one reads a single file from *stdin* and writes a single geom-format file to *stdout*. Typically, you should redirect *stdout* to a file whose extension is *.geom*, so that it is directly readable by the AVS application:

```
pdb_to_geom < my_entry.ent > my_entry.geom
```

Command-Line Options

Most of the filters don't accept any command line options. The exceptions are described in the following sections.

ts_to_geom Filter Options

The ts_to_geom filter utility accepts the following options:

`-bbox` *`xmin xmax ymin ymax zmin zmax`*

Define a bounding box to scale the object down non-uniformly.

`-bratios` *`x y z`*

Alter the aspect ratio of the bounding box. x=1, y=1, z=1 is a uniform aspect ratio. x=1, y=1, z=0.5 forces the Z dimension to be half the size of the X and Y dimensions.

`-boxed`

Put a wireframe box around the object.

`-noscale`

Do not attempt to scale the object at all.

pdb_to_geom Filter Options

The pdb_to_geom filter accepts the following option:

`-balls`

Use the sphere representation instead of the default ball-and-stick representation.

Postprocessor Filters

Convex supplies an additional set of filters, which can use to postprocess the output of the geom-format converters. For instance, if a file in Movie BYU format defines an object with normals that point inward, you can make a *.geom* file with normals that point outward as follows:

```
byu_to_geom < myobject.byu | geom_flip
                             > myobject.geom
```

The **geom_flip** postprocessor reads and writes a geom-format file, flipping the normals of the object defined therein.

The following section lists the postprocessor filters supplied by Convex. Except for **send_to** and **animate_to**, each filter reads a file from *stdin* and writes a file to *stdout*.

animate_to -file *name geom-file1 geom-file2 ...*

Creates a script that defines a cycle object, stores the script as *name.obj*, and causes a currently-executing AVS program to read the script file. The cycle includes the specified *geom*-file sequence.

geom_flip

Flips normals of object

geom_pickable (-non)

Makes all objects pickable (non-pickable), so that their attributes can be set (or cannot be set) in individually.

geom_scale

Scales the object uniformly, so that it lies within the unit cube, (-1,-1,-1) to (1,1,1). Also, converts the object's normals to unit length (normalizes them).

geom_to_normals (-scale *length*)

Produces a disjoint-line object that represents the normals of the input object. The **-scale** option specifies the length of the normals. The default length is 1.

geom_to_text

Produces an ASCII version of the input *.geom* file.

geom_split (*name*)

Creates a series of *.geom* files, each of which contains one of the objects defined in the input file. This is appropriate only for input files that define complex objects (e.g. a polyhedron, which consists of several polygons). Each output file is named *name.n.geom*. If you don't specify the optional name, the files are named *split0.geom*, *split1.geom*, etc.

send_to *filename*

Causes a currently-executing AVS program to read the specified file, which should be a *.geom* file.

text_to_geom

Produces a *.geom* file from an ASCII source file containing a geometry description (generated as output from the *geom_to_text* filter). This filter, as a complement to the *geom_to_text* filter, is useful for porting geometries across different machines.

Templates for New Filter Utilities

Several C-language and FORTRAN-language templates are provided for those who wish to write their own filter utilities. (If your data format is simple enough, you may be able to use one of the templates without modifying it. The mesh format, in particular, can often be used without modification.)

Each template handles a particular type of object defined in the GEOM library. The table below lists the Convex-supplied filter templates. Each one reads a file form stdin, writes a file to stdout, and accepts no command-line options.

Table A-2
Templates for Filter Utilities

Source File Name(s)	Executable File Name	Object Type
mesh.c mesh.f	mesh_to_geom	Mesh
polygon.c polygon.f	polygon_to_geom	Disjoint polygon
polyh.c	polyh_to_geom	Polyhedron
sphere.c	sphere_to_geom	Sphere (from xmole demo)

The source for these filters is located in directory */usr/avs/filter*. Executable versions are in */usr/avs/bin*.

Writing a New Filter Utility

This section provides pointers for those who wish to create new filter utilities, using the template programs listed in the table above.

The basic procedure for creating a geom-format object is:

1. Decide which of the geom-format objects conforms most closely to the application data:

Polyhedron	A list of vertices with an indirect list of pointers into these vertices for each polygon.
Polygon	A list of vertices for each polygon.
Mesh	A 2D array of values, either scalars (for a height field) or vertices.
Sphere	A list of center points and radii.
Polytriangle	A single list of vertices representing polylines, disjoint lines, or a triangle mesh, where the connectivity is implied by the particular data type.

Note

No tools exist for direct conversion of non-linear geometries, such as spline surfaces and quadrics.

2. Create an instance of that geom-format.
3. Perform any necessary processing on the object, such as generating normals.
4. If necessary, convert this object to an optimized-format object, such as a polytriangle.
5. Write the object to a file.

The GEOM library contains routines that help with these tasks. For documentation of these routines, see *geom(3V)*. The sections below describe the steps for converting a variety of object types to geom format.

Converting a Polyhedron

Start with the template *polyh.c*, and

1. Create a polyhedron object.
2. Add vertices.
3. Add a list of polygons (as a list of pointers).
4. Generate normals (if necessary).
5. Convert to polytriangle object -- both wireframe and surface descriptions.

Converting a Polygon

Start with the template *polygon.c* or *polygon.f*, and

1. Create a polyhedron object.
2. Add disjoint polygons (either faceted or smooth).
3. Generate normals (if necessary).
4. Convert to polytriangle object -- both wireframe and surface descriptions.

Converting a Scalar Mesh

Start with the template *mesh.c* or *mesh.f*, and:

1. Create a mesh from a list of scalars.
2. Generate normals (if necessary).
3. Convert to polytriangle object -- both wireframe and surface descriptions.

Converting a Mesh

Start with the template *mesh.c* or *mesh.f*, and:

1. Create a mesh from the vertices.
2. Generate normals (if necessary).
3. Convert to polytriangle object -- both wireframe and surface descriptions.

Converting a Sphere

Start with the template *sphere.c*, and:

1. Create a sphere object from the sphere centers and radii.

Converting a Disjoint Line

There is no starting template for this case. You should:

1. Create a polytriangle object.
2. Add disjoint lines to this object.

Converting a Polyline

There is no starting template for this case. You should:

1. Create polytriangle object.
2. Add zero or more polylines to this object.

The Geometry Viewer Script Language

B

The ConvexAVS Script Language provides a simple method for creating objects with specific properties (color, reflectance characteristics, rendering method). You can define objects hierarchically, and specify multiple instances of an object in a hierarchy. You can also define entire scenes, which comprise objects, lighting, and one or more cameras (views).

A script is an ASCII file, which you can create with any text editor. You store the script under a filename with extension *.obj* or *.scene*. Such scripts can then be read by the AVS viewing application, using the **Read Object** and **Read Scene** functions.

The AVS viewing application itself creates a file using the Script Language whenever you use the **Save Object** or **Save Scene** function. It is often useful to create a file in this way, then revise it later using a text editor.

Scene Files and Object Files

The AVS Script Language can be used to represent either object information alone, or object information along with viewing and light-source information. A single file format handles both these cases, but for convenience, filename extensions are used to distinguish a scene (which contains object, viewing and light source information), from an object (which contains only object information). A scene file should always have a *.scene* extension, an object file should always have a *.obj* extension.

For both objects and scenes, the script file format specifies properties of the top-level object. Views and light sources are considered to be properties of this object. The only real difference between a file with a *.scene* and a *.obj* extension is that:

- Reading a *.scene* file creates a new top-level object, then modifies the object's properties.
- Reading a *.obj* file modifies the existing top-level object's properties.

Example: This object (.obj) file sets the color of the current top-level object to red:

```
set_color 1.0 0.0 0.0
```

Script Language Commands

The script language commands are described in the following sections: Object Commands, Viewing Commands, and Lighting Commands. The table below summarized these commands.

Table B-1 AVS Script Language Commands

Type	Command	Description
Object	read	Read object from disk file
	group	Create group of objects
	cycle	Create animation group
	set_color	Set color of object
	set_material	Set surface properties of object
	set_matrix	Set transform matrix
	set_position	Set X-Y-Z position of object
	set_render_style	Set rendering style of object
	rotate	Rotate object
	translate	Translate object
	scale	Scale object
Viewing	view	Define a new view (window)
	set_matrix	Set the viewing matrix
	set_position	Set the world coordinates origin
	inactive	Make the view inactive
	rotate	Rotate object
	translate	Translate object
	scale	Scale object
Lighting	light	Define a new light
	set_matrix	Set rotational position of light
	set_position	Set X-Y-Z position of light
	set_color	Set color of light

Object Commands

Each object command affects the properties of a particular object. Some object commands create a new object, which is added as a child of the current object. You can also specify the initial properties of the new object. This mechanism can be used to create an arbitrarily complex hierarchy of objects.

The read Command

`read` *name geom-file { object-properties }*

This command reads in a new object, making it the child of the current object. The object is given the name *name* and is read from file *geom_file*. The file must be an AVS geometry file, with a *.geom* extension.

If the *.geom* file is in the same directory as the .obj or **.scene* file being created, specify it with a simple file name. Otherwise, specify it with an absolute (complete) path name.

The new object can have its initial properties set in the optional object-properties field.

The group Command

`group` *name { object-properties }*

This command creates an object whose sole purpose is to group together a list of sub-objects. The object is given the name *name* and has a set of initial properties (including a list of sub-objects) in *object-properties*. Figure B-1 shows an example group.

Figure B-1
The group Command

```
group TheFlintStones {
    group FlintStone {
        read Fred flintstone.geom {
            set_color 0.0 0.0 0.0
            set_position 0.0 1.0 0.0
        }
        read Wilma flintstone.geom {
            set_color 1.0 0.0 0.0
            set_position 0.0 0.0 1.0
         }
    }
    group Rubble {
        read Barney rubble.geom {
             set_color 1.0 1.0 1.0
             set_position 1.0 0.0 0.0
         }
        read Betty rubble.geom {
            set_color 0.0 1.0 1.0
            set_position 0.0 1.0 0.0
         }
    }
}
```

The cycle Command

`cycle` *name { object-properties }*

This command creates an animation object, for which all of its children are considered to be mutually exclusive representations of the same geometry. This can be used to create an animation sequence. It can also be used to create a list of different representations that can be selected for a particular object (e.g. spheres vs. lines for a molecule). The object is made a child of the current object, and initial properties can be specified for the object. Figure B-2 shows an example cycle definition:

Figure B-2
The cycle Command

```
cycle Molecule {
    read balls sphere.geom {}
    read stick ball_and_stick.geom {}
    read lines line.geom {}
 }
cycle Face {
    read Smile smile.geom {}
    read Frown frown.geom {}
     read Grimace grimace.geom {}
}
```

The set_color Command

`set_color` *red green blue*

This command sets the color of the object to the specified RGB value. *red, green,* and *blue* must be a number between 0 and 1.

The set_matrix Command

`set_matrix` *4x4-matrix*

Sets the current transformation to be the 4x4-matrix specified. Supplying a transformation in this matrix allows you to alter the center of rotation of the object.

Note

The specified matrix replaces the existing transform. Contrast this with rotate, translate, and scale, which concatenate transformations with the existing one.

The set_position Command

`set_position` *x y z*

This command sets the position of the object to be the *x, y,* and *z* values specified. Setting the position does not alter the center of rotation of the object.

The set_material Command

```
set_material ambient diffuse specular\
    spec-exponent transparency spec-red \
    spec-green spec-blue
```

Sets the material properties of the object. All values except for the specular exponent vary from 0 to 1. The specular exponent, which specifies the roughness of the surface, should lie between 1 (roughest) and 200 (smoothest).

The set_render_style Command

```
set_render_style style
```

Sets the rendering method used to draw the object. style should be one of the following:

- `lines`
- `gouraud`
- `inherit`
- `flat`
- `smooth_lines`
- `no_light`

The rotate Command

```
rotate angle x y z
```

Rotates the object by angle degrees counterclockwise around the vector (x,y,z). This transformation is concatenated with the object's current transform.

The translate Command

```
translate x y z
```

Translates the object by the vector (x,y,z). This transformation is concatenated with the object's current transform.

The scale Command

```
rotate angle sx sy sz
```

Scales the object by sx, sy, and sz in the X, Y, and Z directions. This transformation is concatenated with the object's current transform.

Viewing Commands

Views (windows) can be created using the view command. Like objects, views also have properties. A view should only be specified in a file that has a *.scene* extension.

The view Command

`view` *name widthxheight+x+y bkg-red* \
bkg-green bkg-blue { view-property-commands }

The following example creates a 500x500 pixel view named Bob, at offset 100,100 from the upper left corner of the screen, and with a red background.

```
view Bob 500x500+100+100 1.0 0.0 0.0 { }
```

The view-property-commands are described below.

The set_matrix Command

set_matrix *4x4-array-of-floats*

Sets the viewing matrix.

The set_position Command

set_position *x y z*

Sets the position of origin of the world coordinate system.

The rotate Command

rotate *angle x y z*

Rotates the object by angle degrees counterclockwise around the vector (x,y,z). This transformation is concatenated with the object's current transform.

The translate Command

translate *x y z*

Translates the object by the vector (x,y,z). This transformation is concatenated with the object's current transform.

The scale Command

rotate *angle sx sy sz*

Scales the object by sx, sy, and sz in the X, Y, and Z directions. This transformation is concatenated with the object's current transform.

The inactive Command

```
inactive
```

TSets the initial state of the view to be inactive.

Geometry Viewer Defaults File

You can specify a defaults file to be read by the Geometry Viewer when you start AVS with the *-geometry* command-line option. For example:

```
avs -geometry -defaults \
 /usr/henry/avs/geom_windows.dfl
```

The defaults file defines a series of windows, assigning each one a name, a size and position (in standard X Window System notation), and an RGB background color.

The first window in the series is used for the first Geometry Viewer window that appears. Subsequent windows are used, in turn, by the **Create Scene** and **Create Camera** functions and by the render geometry module.

If the end of the series is reached, additional windows are created with the same size as the last window, but slightly offset from each other.

Figure B-3 shows a sample Geometry Viewer defaults file:

Figure B-3
Sample Geometry Viewer Defaults File

```
view HENRY01 400x400+300+100 1.0 1.0 0.0
view HENRY02 400x400+750+100 0.0 1.0 1.0
view HENRY03 300x300+400+500 0.0 1.0 1.0
```

Lighting Commands

Like viewing commands, lighting commands should only be specified in a file that has a *.scene* extension.

The light Command

`light` *type index { lighting-property-commands }*

This command turns on a light of type, where type is one of ambient, directional, or point. The light is given index index and can contain properties specified in lighting-properties (see below). Light indices should be in the range of 1 to 16. In the viewing application, a single ambient light source is assigned to index 16.

For example, this command turns on light 1, making it a directional light with the default lighting properties:

```
light directional 1 {}
```

The lighting property commands are described in the following sections.

The set_matrix Command

`set_matrix` *4x4 matrix*

This command sets the transformation matrix for the light. In the case of directional lights, only the rotation portion of the matrix is used (although the rest of the matrix can be used to affect the graphical display of light vectors). In the case of a point light, this matrix only affects the graphical representation of the point light source icon.

The set_position Command

`set_position` *x y z*

This command sets the position of point light sources. This attribute does not affect directional light sources.

The set_color Command

`set_color` *red green blue*

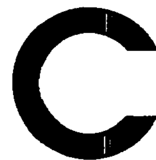
This command sets the light source color.

Example Scene File

Figure B-4
Scene File

```
view AVS 503x529+375+208 0.015686 0.078431 0.196078 {
 set_matrix0.968946 -0.201782 0.142953 0.000000
           0.246114  0.730629  -0.636879 0.000000
           0.024065  0.652280   0.757599 0.000000
           0.000000  0.000000   0.000000 1.000000
 }
light directional 1 {
 }
light ambient 16 {
}
read teapot.2 teapot.pobj {
       set_color 0.162 0.046 0.013
       set_material 0.287 0.444 0.931 10.000 0.000
0.990 0.241 0.027
     }
}
```

ConvexAVS File Formats



Start-up File

When it begins execution, ConvexAVS searches for a start-up file that specifies the locations of various directories. ConvexAVS looks for the following files, in the order listed:

./.avsrc	(current directory)
$HOME/.avsrc	(home directory)
/usr/avs/runtime/avsrc	(system directory)

If ConvexAVS finds one of these files, it reads at and ignores the others. A */usr/avs/runtime/avsrc* file is included on the Convex-AVS distribution tape.

Each line of the ConvexAVS start-up file consists of keyword-value pair, with white space separating the keyword and the value. For example:

```
NetworkWindow        867x567+407+2
NetworkDirectory     /usr/smith/avs/nets
DataDirectory        /usr/smith/avs/data
```

In most cases, the keyword corresponds to one of the command-line options described in the preceding section. If you use a command-line option, it overrides the specification, if any, in the start-up file.

The ConvexAVS start-up file keywords are:

DataDirectory

`(command-line equivalent: -data)`

Specifies the directory in which the various read data modules (read field, read geometry, etc.) initially will look for data files.

NetworkDirectory

`(command-line equivalent: -netdir)`

Specifies the directory in which the Network Editor subsystem initially will look for network files (**Read Network** and **Write Network** functions).

NetworkWindow

```
(command-line equivalent: none)
```

Specifies the X Window System geometry of the Network Construction Window that includes the Network Editor menu, the Module Palette, and the Workspace in which you construct networks of modules.

Path

```
(command-line equivalent: -path)
```

Specifies the directory tree in which ConvexAVS itself is installed.

Module Library File Formats

This section describes the format of a module library file, as you can create it with any text editor. ConvexAVS itself uses a somewhat more complex file format for the module library file it creates when you select the Write Module Library function. The more complex format allows the Network Editor to load the module library more quickly.

ID Line

The module library file begins with this line:

```
# AVS Module Library
```

Other lines that begin with # may precede this line.

Command Lines

Each subsequent line must be in one of these forms:

built-in	module_name
file	filename
directory	dirname

The built-in keyword specifies a module that is built into ConvexAVS itself, rather than being implemented as a separate executable file. The following modules are built-ins:

colormap manager	display pixmap
generate colormap	render geometry
image manager	render manager
volume manager	orthogonal slicer
write image	display image
write volume	

The file keyword specifies an executable file that includes one or more module definitions. Refer to the *Developing Applications* section for more on creating modules.

The directory keyword specifies a directory that includes executable module definition files.

Image File Format—.x

The **read image** and **image manager** modules can read a file that contains an image— a 2D array of pixel values. In ConvexAVS, such files should have names that end with a *.x* suffix.

The file must begin with a two-word header that specifies the dimensions of the image:

first word: number of pixels in *horizontal* direction (32-bit integer)

second word: number of pixels in *vertical* direction (32-bit integer)

There is no explicit limit on the size of an image.

The remainder of the file is a sequence of 4-byte (32-bit) words, one for each pixel of the image. The pixels are arranged in rows; there is no padding at the end of a row.

The four bytes of a pixel are interpreted as four component values in the range 0..255. Three of the bytes are the red, green, and blue color components. The fourth byte is an auxiliary field, which is used by some ConvexAVS modules to represent an opacity/transparency value:

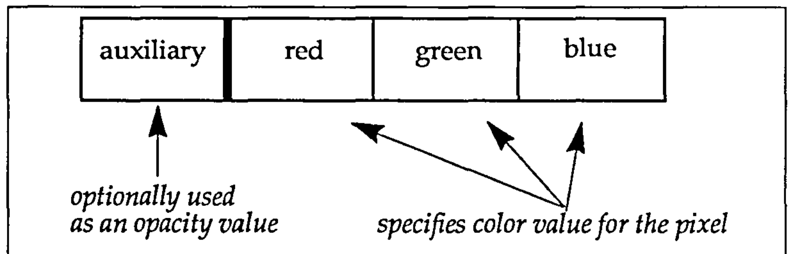


Figure C-1 illustrates the ConvexAVS image data file format. Image data files should have names that end with a *.x* suffix.

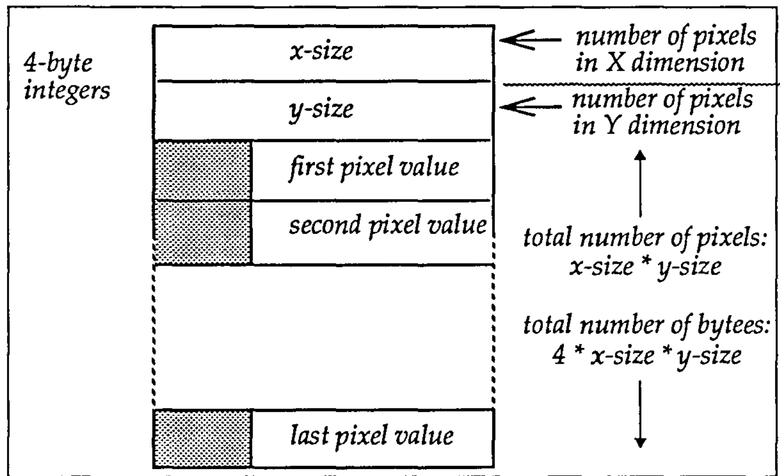


Figure C-1
Image Data File Format

Field File Format—.fld

A field is a generalized array structure. Whereas each element of an ordinary array has a single data value (for example, byte or integer), each element of an ConvexAVS field can have a list of data values. Thus, a field can be described as an n-dimensional array with an m-dimensional vector of values at each array location (where n and m are any integers).

Moreover, the field can include coordinate data so that each field element is mapped to a real-world location.

Figure C-2 illustrates the top-level structure of a ConvexAVS field:

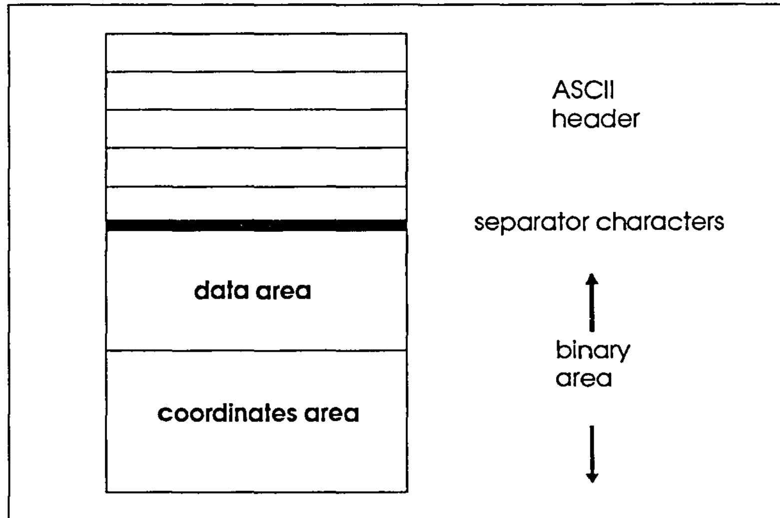


Figure C-2
ConvexAVS Field Structure

Field data files should have names that end with a *.fld* suffix.

Before describing the field file format in more detail, we present several examples of fields. This illustrates the power and flexibility of the field construct.

Example 1: Uniform 2D Field

Consider the following 2D integer-valued array (using a FORTRAN-style notation):

```
DATA(I, J)  I=1,2      J=1,5

DATA(1,1)    =         12
DATA(2,1)    =         17
DATA(1,2)    =          4
DATA(2,2)    =          0
DATA(1,3)    =         10
DATA(2,3)    =         -5
DATA(1,4)    =         16
DATA(2,4)    =         16
DATA(1,5     =         16
DATA(2,5)    =          8
```

This array describes a 2D computational space, with I and J dimensions. The size of the I dimension is 2; the size of the J dimension is 5. The data is of type integer.

Because there is only one data value for each field element, this is said to be a scalar field. The following notation might be used to indicate the values of a vector field:

```
DATA(2,3)    =         (2.51, 1.09, 5.73)

     or

DATA(2,3)    =         (2.51, 1.09, 5.73, 0, 1)
```

In the first case, the field is still 2-dimensional, but the data value is said to be a 3-vector. Such a data value might be used to represent a velocity vector. The 5-vector in the second case might represent the temperature-pressure-humidity measurements at each location in space, along with two boolean values to indicate the presence/absence of other atmospheric conditions.

In the absence of any additional information, there is a natural mapping between the computational space and a 2D physical space, the X-Y coordinate plane as shown in Figure C-3:

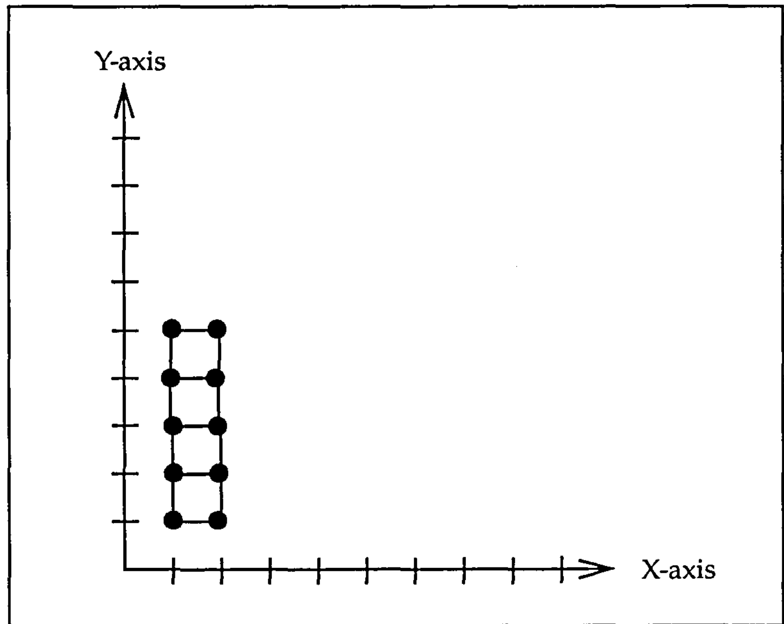


Figure C-3
Mapping Computational and 2-D Physical Space

The physical space is a uniformly-spaced lattice. Accordingly, a field with no coordinate data is said to have the field type *uniform*

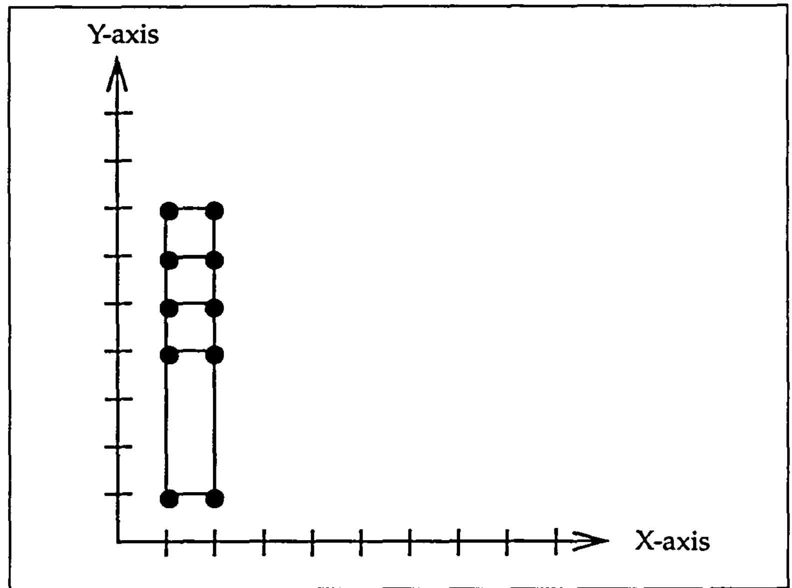
Example 2: Rectilinear 2D Field

Continuing the preceding example, we can establish an explicit mapping between the computational and physical spaces by specifying coordinate data:

X-coordinates:	0, 3, 6, 9, 12
Y-coordinates:	20*log(1), 20*log(2), 20*log(3), 20*log(4), 20*log(5)

For example, array element IDATA(1,3) is mapped to physical location (0, 20*log(3)) according to this scheme. This mapping from computational space to the X-Y plane is illlustrated in Figure C-4:

Figure C-4
Mapping a Rectilinear 2D Field



Note

In a *rectilinear* field, lines connecting the lattice points are always mutually orthogonal— all the angles are right angles.

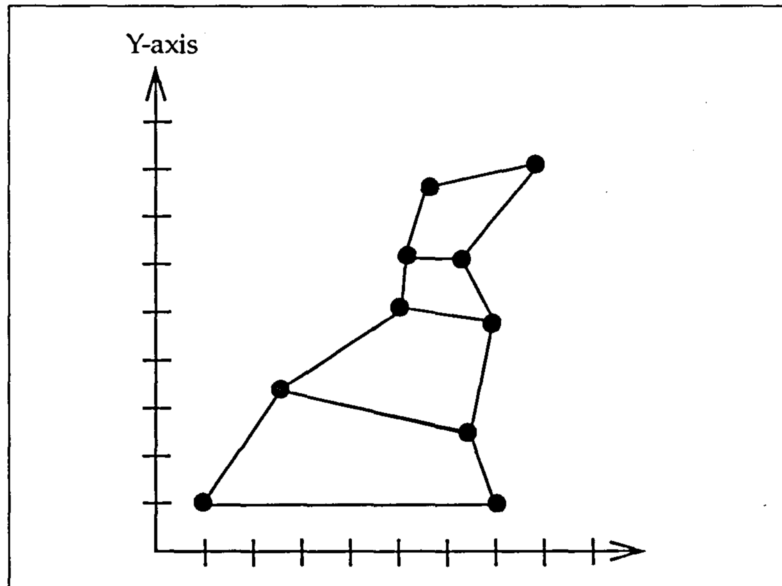
Example 3: Irregular 2D Field

Continuing the example once more, there is another way of establishing a mapping between the computational and physical spaces. Instead of mapping the array indices, we can map individual field elements to arbitrary points in physical space:

```
DATA(1,1) ->     (1, 1)
DATA(2,1) ->     (7, 1)
DATA(1,2) ->     (3, 3)
DATA(2,2) ->     (6, 2.5)
DATA(1,3) ->     (4, 4.5)
DATA(2,3) ->     (5.5, 4.5)
DATA(1,4) ->     (3.5, 6)
DATA(2,4) ->     (4.5, 5.5)
DATA(1,5) ->     (3.5, 7.5)
DATA(2,5) ->     (6,8)
```

This mapping from computational space to the X-Y plane can be pictured as follows:

Figure C-5
Mapping an Irregular 2D Field



There is nothing in this scheme that restricts the physical space to having the same number of dimensions as the computational space. For example, the field element DATA(2,3) could be mapped to the physical point (4.5, 5.5, -8.1) in 3D space. This kind of mapping can be used to wrap a plane (computational space) around a sphere (physical space) or to warp a flat plane into a 3D manifold.

For additional examples, including some involving non-2D fields, refer to Chapter 8, "Data Types," in the *Developing Applications* section.

ASCII Header

Every field file must begin with a header in ASCII text format, as shown in Figure C-6.

Figure C-6
ASCII Header for AVS Field File

```
# AVS field file
#
ndim   = 2    # number of computational dimensions
dim1   = 512
dim2   = 480
nspace = 2    # number of physical dimension
veclen = 4
data   = byte
field  = uniform
```

This header includes a number of *keyword=value* pairs, one per line; it also may include comment lines. For example:

The keywords are described below, with reference to the three examples in the preceding sections.

ndim

This value specifies the number of computational dimensions in the field. That is, it specifies the number of dimensions in the field element array. In all the examples above, *ndim* has the value 2.

dim1, dim2, ...

The values for these keywords specify the size of the computational space. For a 2D field, you would specify *dim1* and *dim2* values. In all the examples above, *dim1* = 2 and *dim2* = 5. For a 4D field, you would specify *dim1*, *dim2*, *dim3*, and *dim4* values.

nspace

This value specifies the dimensionality of the physical space that corresponds to the computational space. In all the examples above, *nspace* = 2. At the end of Example 3, the following mapping to a 3D physical space is suggested:

```
DATA(2,3)-> (5.5, 4.5, -8.1)
```

In this case, nspace = 3.

veclen

This value specifies the number of data values for each field element. Example 1 above discussed two possibilities:

```
DATA(2,3)= -5                                  veclen = 1
DATA(2,3)= (2.51, 1.09, 5.73, 0.0, 1.0)veclen = 5
```

data

This keyword takes one of the following values:

- byte
- integer
- real
- double

It indicates the type of data that is supplied for each field element. ConvexAVS fields have the restriction that all of the *veclen* data values must be of the same type.

field

This keyword takes one of the following values: *uniform, rectilinear, irregular.* The main purpose of the three examples above is to illustrate these three field types.

A **uniform field** (as in Example 1) has no computational-to-physical space mapping. The field implicitly takes its mapping from the organization of the computational array of field elements.

For a **rectilinear field** (as in Example 2), each array index in each dimension of the computational space is mapped to a physical coordinate. This produces a physical space whose axes are orthogonal, but the spacing among elements is not necessarily equal.

For an **irregular field** (as in Example 3), there is no restriction on the correspondence between computational space and physical space. Each element in the computational space is assigned its own physical coordinates.

Separator Characters

The ASCII header must be followed by two formfeed characters, in order to separate it from the binary area. A *formfeed* is expressed variously as Ctrl-L, octal 14, decimal 12, or hex 0C.

This scheme allows you to use the *more(1)* command to examine the header. When more stops at the formfeeds, press q to quit. This avoids the problem of the binary data garbling the screen.

Binary Area

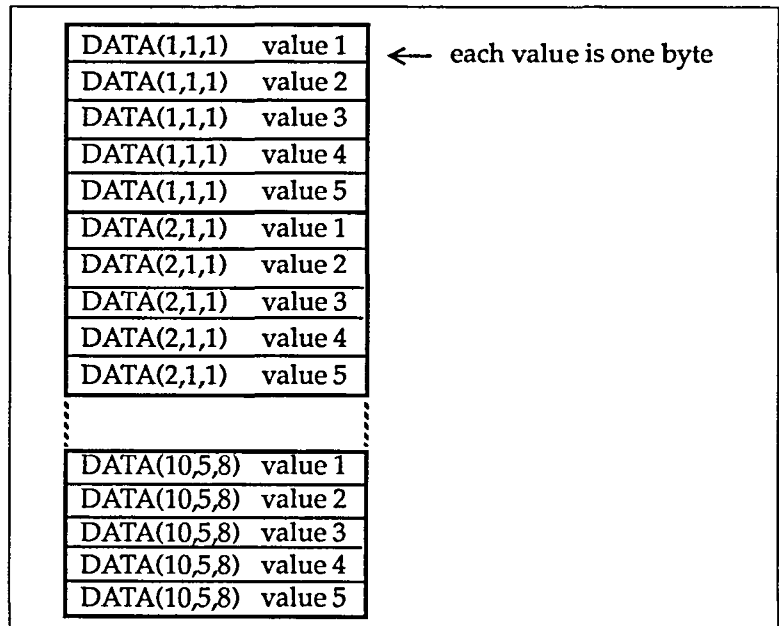
The binary area consists of all the data that is associated with the field elements, along with all the coordinates. (For uniform fields, the coordinates area is null.)

The data area begins with one or more data values for the first field element. All the data values for a field element are stored together. The first array index varies most quickly (FORTRAN-style). For example, suppose the ASCII header is as follows:

```
ndim = 3
dim1 = 10
dim2 = 5
dim3 = 8
nspace=3
veclen=5
data=byte
field=uniform
```

The data ordering can be illustrated as follows:

Figure C-7
Data Ordering



Volume Data File Format—.dat

Measurement data often takes the form of a 3-dimensional array, which corresponds to a uniform lattice in 3D space. Each array value indicates one measurement (temperature, pressure, etc.) at the corresponding lattice point. Such data can be represented as a uniform 3D field, but ConvexAVS also provides a simpler volume data format to accommodate this type of data.

The ConvexAVS volume data format requires that each value in the data array be a byte. (For other data types (for example, single-precision), you must use the more general field construct.) Volume data files should have names that end with a .dat suffix.

Note

The volume data and field file formats are not compatible.

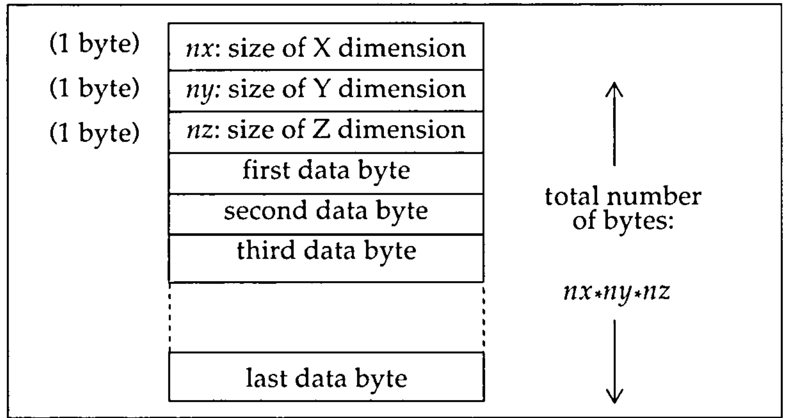
A volume data file begins with a three-byte header that specifies the size of the array in the first (X), second (Y), and third (Z) dimensions. Because each dimension's size must be expressed as a 1-byte number, the largest array supported by this file format is 255x255x255.

The remaining contents of the file are the values of a 3D array of bytes, in column-major order (FORTRAN-style). For example, the values in a 50x20x10 array would be stored as follows (using FORTRAN notation):

```
DATA(1,1,1)
DATA(2,1,1)
DATA(3,1,1)
...
DATA(50,1,1)
DATA(1,2,1)
DATA(2,2,1)
DATA(3,2,1)
...
DATA(1,20,1)
DATA(2,20,1)
DATA(3,20,1)
...
DATA(1,20,2)
DATA(2,20,2)
DATA(3,20,2)
...
DATA(1,20,10)
DATA(2,20,10)
DATA(3,20,10)
...
DATA(50,20,10)
```

Figure C-8 illustrates the volume data file format.

Figure C-8
Volume Data File Format



Geometry File Format—.geom

ConvexAVS geometric descriptions are created and written to disk files with calls to the libgeom.a programming library. Appendix G, "Geometry Routines", presents a complete description of this library.

Memory Management

D

The purpose of this document is to provide insight into memory utilization by ConvexAVS. A general explanation of how data is processed by a ConvexAVS dataflow network is followed by a more specific example. The resolution of the information is necessarily limited and will not be applicable to data sets which are small relative to the size of the executable programs themselves. An understanding of the design compromises in ConvexAVS memory usage and data management will be valuable, however, in achieving acceptable performance on your system.

First, it's important to realize that in general, each module (icon) instantiated as part of a dataflow network is a separate process under ConvexOS. The program which is executed when you enter the "`avs`" command is what is referred to as the AVS kernel. The kernel executes additional commands for you in order to build a processing pipeline which performs the functions you describe with a given dataflow network. What the user considers to be ConvexAVS then, appears to ConvexOS as a family of programs with the AVS kernel as the common ancestor.

User Perspective

From a user perspective, a dataflow network is connected, the data input module (usually at the top of the network) is instructed to obtain data, and then the data "falls" through the network with each module passing its results to the next until the final results are saved in a file or displayed in a window. The user may then change a parameter on one of the modules, with the results of that change rippling down the network from the point where the change was made to display the altered results. It is important to note that the entire processing pipeline is not usually executed each time a module parameter is changed; only the module where the change was made and all "downstream" modules needed to recompute results.

Kernel Perspective

From the perspective of the ConvexAVS kernel, this procedure looks a little different. When each module is instantiated by the kernel, it is told where to read input data, where to write output results, and in most cases when to execute.

The input module, as an example, would be instructed by the kernel to obtain data from a given source and store it in a memory cache; the next module in the dataflow network would then be instructed to process that memory cache as its input, storing results in a different memory cache.

Each module in turn processes inputs and stores results in its memory cache until the final module in the network stores its results in a data file or perhaps displays results as an image.

One obvious difference in the kernel's perspective is that the data doesn't fall from the top of the network to the bottom, but rather is transformed by each module with the interim transformations saved while only the final one is viewed. The effect of this procedure is that the memory required to run a particular network increases with the number of modules in the network. The amount of additional memory required for each module added to a network is dependent on the processing done by that module.

A fundamental goal of ConvexAVS is to allow the user to experiment interactively with an AVS network to visualize data. A processing pipeline where each module saves a copy of its results, even after the next module has processed them, compromises memory efficiency in order to maximize the possibility for interactive response. After the network has been executed one time, the input data set for each module is already available. This is the reason that changing a module parameter only executes the portion of the network affected by the change, rather than recomputing the entire network. This trade-off between memory utilization and speed is vital to understand when processing large data sets or attempting to increase interactive response.

Problem:

Since memory is a finite resource, even on virtual memory machines, a sufficiently complex network or arbitrarily large data set could require more memory for execution than the operating system can provide. Insufficient available memory is an irrecoverable error, and will interrupt the execution of a network in one of two ways, depending on whether the memory request was made by a module or the kernel:

- When a module makes a request for memory which the operating system cannot meet, the module will exit and cause the following message to be printed to the controlling terminal (that is the **xterm** window in which you executed the **avs** command):

  ```
  AVS malloc: no available memory
  ```

  If the controlling terminal window is not visible, your only indication will be a warning pop-up from ConvexAVS stating that a particular module is presumed dead (its associated icon will turn black).

  If you de-iconify or raise the controlling terminal window, you should find the *malloc* error message, probably accompanied by complaints from other modules in communication with the exited process. In this case, the remainder of the network downstream from the exited module is still functional, but there is no cached input for the top module in the remaining network.

- If the ConvexAVS kernel has a memory request denied, ConvexAVS will exit with the appropriate error message.

Data Types

There are four aggregate data types in ConvexAVS:

- Pixmaps (light blue)
- Colormaps (yellow)
- Geometries (red)
- Fields (dark blue).

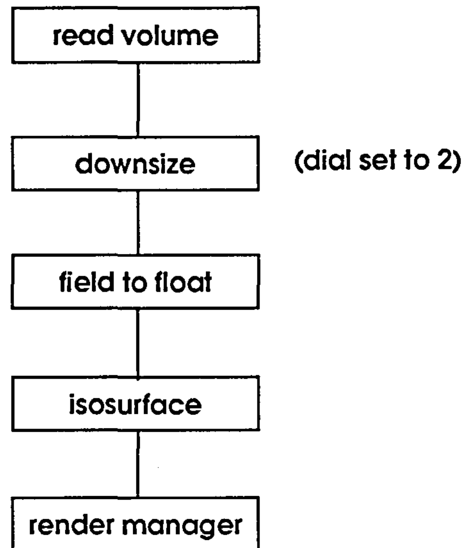
Two of these data types, Pixmaps and Colormaps, use so little memory that they can be ignored when considering memory consumption. Geometry and Field data can use large amounts of memory however.

Since fields are a major consumer of memory, ConvexAVS employs *shared memory* to avoid unnecessary duplication of field data. Shared memory allows two separate processes to "map" the same physical memory into their own individual address spaces. When a module's field output is connected to another module's field input, no field data is actually copied, rather the location of the output field is simply passed to the input of the other module.

Example Strategies

Strategies for effective utilization of system resources with ConvexAVS will vary from site to site, depending on such factors as job mix, amount of physical memory, size of virtual memory, and number of simultaneous users. ConvexAVS can make effective use of virtual memory since upstream modules that are seldom used will tend to be migrated into swap space by normal system activity. If the system begins to page heavily, however, it may be an indication that the required data sets are larger than physical memory, and some methods for decreasing the data set should be investigated.

Consider the following network:



For purposes of illustration, assume a volume data set of dimensions 250x250x250 is read into the network above.

read volume

The **read volume** obtains the data set from a file and constructs a uniform scalar 3D byte field. This requires:

`(250*250*250) = 15,625,000 bytes` *(about 15 MB)*

This field is stored in a memory cache until either the Read Volume module is destroyed, or another volume file is read. ConvexAVS then passes the field location downstream to **downsize**.

downsize

The purpose of downsize is to subsample the input field in order to reduce the size of the data set, a crucial function when dealing with large data sets. In this illustration, the downsize parameter is set to 2. Since downsize subsamples along each dimension of the input field, the reduction factor will be:

```
1/(2*2*2)
```

for a three dimensional field, resulting in an output field of 15MB/8, or **1.88 MB**.

field to float

The **field to float** module converts any type of field into a field with single precision floating point data. In this network, its purpose is to convert the volume data into a data type that Isosurface can process. Converting byte data to single precision floating point data (4 bytes) causes a factor of four increase in the size of the output field: the output field will be:

```
1.88MB * 4 = 7.5 MB
```

isosurface

Isosurface constructs a geometry output consisting of a triangle mesh that represents an approximation of a single-valued surface within the data set. Since Isosurface maps a field input to a geometry output, there is no straightforward way to predict how large the data set will be. The simplest way to determine this is to measure the difference in the size of the Isosurface module before and after it does its computation. The ConvexOS *ps(1)* utility is helpful in this regard, but be forewarned that the size reported by *ps()* includes any shared memory segments to which a process has access (for instance the 7.5 MB Field to Float output field).

render geometry

While **render geometry** does not have an output port, it does have to retain geometric data in some form in order to redisplay it in multiple viewing situations. Again, the increase in size can be measured with *ps(1)* before and after **render manager** module execution.

By this point it should be clear that this network will request a great deal more memory than the 15 MB of the original data set. If the network was reordered so that **field to float** and **downsize** were reversed, the visual results would be the same, but the memory requirements would increase enormously (the output field of the **field to float** module would be 60 MB for instance).

One way to develop a useful dataflow network is to experiment first with a small representative data set. When a network configuration appears ready for use with a "real world" data set, examine it for memory efficiency.

- How many transformations does the data go through on its journey down the network and what is the predicted memory requirement to store each transformation?
- Should data be downsized, and if so, how early in the network is this feasible?

If the expected memory demand is still too large (perhaps downsizing is not reasonable), consider breaking the network into smaller sub-networks, each of which stores results on disk. When executing a network, be aware that you can disconnect and destroy modules when their output fields are no longer required, thereby releasing the shared memory segments used for data cache. In this case, you will be required to rebuild the network if you later need the released modules.

While it is possible to build very complex, multi-function networks, the usefulness of these networks should be carefully balanced against their cost in terms of supporting system resources. By understanding the manner in which ConvexAVS attempts to implement your dataflow network, you can improve the performance and increase the capacity of the network.

Module Routines

E

This appendix is separated into two sections:

- Routines grouped by functions.
- Routines listed and defined alphabetically.

Grouped by Function

The following section groups module routines by function.

Module Description Functions

- AVSadd_float_parameter
- AVSadd_parameter
- AVSadd_parameter_prop
- AVSautofree_output
- AVSconnect_widget
- AVScreate_input_port
- AVScreate_output_port
- AVSinitialize_output
- AVSset_compute_proc
- AVSset_destroy_proc
- AVSset_init_proc
- AVSset_module_flags
- AVSset_module_name

Modifying and Interpreting Parameters

- AVSchoice_number
- AVSmodify_float_parameter
- AVSmodify_parameter

Coroutine Modules

- AVScorout_exec
- AVScorout_init
- AVScorout_input
- AVScorout_output
- AVScorout_wait

Selective Computation

- AVSinput_changed
- AVSmark_output_unchanged
- AVSparameter_changed

Creating Fields

- AVSbuild_2d_field
- AVSbuild_3d_field
- AVSbuild_field
- AVSdata_alloc
- AVSfield_alloc
- AVSfield_copy_points
- AVSfield_make_template

Accessing Fields

- AVSfield_data_offset
- AVSfield_data_ptr
- AVSfield_free
- AVSfield_get_dimensions
- AVSfield_get_int
- AVSfield_points_offset
- AVSfield_points_ptr

Initializing Modules

- AVSinit_from_module_list
- AVSinit_modules
- AVSmodule_from_desc

Handling Errors

- AVSdebug
- AVSerror
- AVSfatal
- AVSinformation
- AVSmessage
- AVSwarning

Defined Alphabetically

The following section defines module routines alphabetically.

AVSadd_float_parameter

C

```
#include <avs/avs.h>
int AVSadd_float_parameter(param_name, init, minval,
                                              maxval)
    char        *param_name;
    double      init, minval, maxval;
```

This routine declares a parameter of type "real" for the module being defined in the current description function. The routine interfaces with the AVSadd_parameter routine; it allocates space for the init, minval, and maxval arguments automatically. The calling routine should declare these arguments as float. In C, when a float is passed as an argument it is converted to a double.

FORTRAN

There is no FORTRAN equivalent for this routine. Use AVSADD_PARAMETER.

AVSadd_parameter

C

```
#include <avs/avs.h>
int AVSadd_parameter(param_name, type, init, minval,
                                          maxval)
    char        *param_name, *type;
    int         init, minval, maxval;
```

FORTRAN

```
#include <avs/avs.inc>
AVSADD_PARAMETER(NAME, TYPE, INIT, MINVAL, MAX-
VAL)
    CHARACTER*(*)    NAME, TYPE
    INTEGER          INIT, MINVAL, MAXVAL
```

This routine declares a parameter for the module being defined in the current description function. Each parameter is connected to a widget in the module control panel to allow you to modify the value of the parameter.

The param_name argument is a string that appears as the name of the widget associated with the parameter.

The init, minval, and maxval arguments are cast as ints in C and integers in FORTRAN, but the storage type depends on the parameter type. For any type of parameter, init, minval, and maxval all have the same storage type. Each value must fit into an integer-size memory slot or must be a pointer to a larger memory allocation. Values representing floats in C must be pointers to allocated memory. The routine AVSadd_float_parameter handles this allocation automatically.

For many parameter types, init is the initial or default value of the parameter, and minval and maxval are the inclusive bounds for the acceptable range of values. When this range is specified, ConvexAVS ensures that values passed to the computation routine are inside this range.

The type argument is a string that represents the parameter type. Table E-1 lists possible values for type. For each type, it lists the C and FORTRAN data types for init, minval, and maxval. These are also the data types for parameters passed as arguments to module computation routines.

Table E-1
Type Values

type **String**	**C Data Type**	**FORTRAN Data Type**
"integer"	int	INTEGER
"boolean"	int	INTEGER
"tristate"	int	INTEGER
"oneshot"	int	INTEGER
"real"	float *	real
"string"	char *	CHARACTER*(*)
"choice"	char *	CHARACTER*(*)
"colormap"	AVScolormap *	-
"field"	AVSfield *	-

ConvexAVS passes fields and colormaps to FORTRAN computation functions as multiple arguments.

Following are notes on some of these types:

integer — The minval argument is the minimum value; the maxval argument is the maximum value.

boolean — Possible values are 0 and 1. The minval and maxval arguments are ignored.

tristate — Possible values are 0, 1, and 2. The minval and maxval arguments are ignored.

oneshot	This is a command-style signal counter. The current value is incremented by 1 each time the value is set, often by means of a mouse click on a widget. This allows the module to determine how many times you set the value. Setting a value of 0, using AVSmodify_parameter, clears the counter. The minval and maxval arguments are ignored.
real	To specify an unlimited range of possible values, set both minval and maxval to the constant FLOAT_UNBOUND. Both minval and maxval must be either bounded or unbounded.
string	This is used for both simple strings and file path names. The value must be NULL in C (0 in FORTRAN) or an allocated string. Widgets often present NULL values as "$NULL." For a text browser, minval is a comment character used to suppress display of text lines that begin with that character. For a file browser, maxval is a list of acceptable file types, separated by periods. For example, if maxval is ".x.image," only path names ending with .x or .image appear in the file browser attached to this parameter.
choice	The value is one of an enumerated set of strings. The minval argument is the set of possible choices separated by a delimiter character. For example, "Alpha!Beta!Gamma". The maxval argument is the delimiter character, a "!" in this case.
colormap	The minval and maxval arguments are ignored. A FORTRAN computation routine cannot take a colormap as a parameter argument.

field	The only supported field type is "field 2D scalar real." This is used for handling 4 by 4 transformation matrices. The minval and maxval arguments are ignored.

This routine returns an integer parameter identifier that is used as an argument to some other routines, such as AVSconnect_widget.

AVSadd_parameter_prop

C

```
AVSadd_parameter_prop(param_num, prop_name,
                           prop_type, prop_value)
   int          param_num;
   char         *prop_name, *prop_type;
   int          prop_value;
```

FORTRAN

```
AVSADD_PARAMETER_PROP(PARAM_NUM,
   PROP_NAME, PROP_TYPE PROP_VALUE)
   INTEGER            PARAM_NUM
   CHARACTER*(*)      PROP_NAME, PROP_TYPE
   INTEGER            PROP_VALUE
```

This routine adds a property to a parameter for the module being defined in the current description function. A property usually determines some aspect of how the user interface presents the parameter. By calling this routine, a module can customize how the user interface handles the parameter.

The param_num argument is a parameter identifier returned by AVSadd_parameter or AVSadd_float_parameter. The prop_name argument is a string specifying the name of the property, and prop_type is a string specifying the type of property value being provided. The property type must be one of the parameter types. Each property has only one permissible property type, and ConvexAVS verifies that the prop_type is permissible for the prop_name supplied.

The prop_value argument is the value of the property. The storage type of prop_value is the storage type that corresponds to the property type. For a floating-point value, prop_type is a float rather than a float * in C.

As an example of using AVSadd_parameter_prop, assume that an integer parameter is attached to a dial widget. By default, when you manipulate the widget, ConvexAVS invokes the module only when you release the mouse button. To invoke the module continually as you manipulate the widget, the description function can use AVSadd_parameter_prop to attach an "immediate" property to the parameter. This property has a boolean value; a value of 1 causes continuous invocation as the mouse moves.

Some properties are not meaningful with all possible widgets. For example, the "immediate" property is not meaningful with a typein widget because the module should be invoked only when you have finished typing in the new value. If a call to AVSadd_parameter_prop requests a property or property value that a widget does not support, ConvexAVS ignores the request when it creates that widget. The property remains attached to the parameter, and ConvexAVS uses the property if you attach an appropriate widget at a later time.

Some widgets may allow you to change properties interactively. When you save a network after making such a change, the property settings are saved as modified. When the saved network is subsequently read, your property settings override values set by the call to AVSadd_parameter_prop.

Table E-2 lists each available property name along with its property type, the C and FORTRAN data types of the property value, and the widget types that support the property.

Following are notes on some of these types:

title — This property specifies a title label for the widget. The default title is the parameter name.

immediate — A value of 0 means that ConvexAVS invokes the module when you have finished manipulating the widget (for example, by releasing the mouse button for a dial or slider). This is the default. A value of 1 means that ConvexAVS continually invokes the module as you manipulate the widget.

Table E-2
Property Names and Types

Property Name	Property Type	C Data Type	FORTRAN Data Type	Widget Type(s)
title	string	char *	character*(*)	dial, idial, slider, islider, toggle, tristate, oneshot, radio_buttons
immediate	boolean	int	integer	dial, idial, slider, islider
accumulator	boolean	int	integer	dial, idial
editable	boolean	int	integer	text
local_range	real	float	real	dial, idial
width	integer	int	integer	toggle, tristate, oneshot, typein, text, browser, text_browser, radio_buttons
height	integer	int	integer	toggle, tristate, oneshot, typein, text, browser, text_browser, radio_buttons
columns	integer	int	integer	radio_buttons

accumulator — This property is used with dial widgets. When the parameter bounds are fixed, a value of 0 means that the parameter range should map to one complete rotation of the dial. This is the default. A value of 1 means that the parameter range may extend over multiple rotations of the dial. When the parameter is unbounded, multiple dial rotations are always allowed.

editable — This property determines whether or not a text widget is editable in the Layout Editor. A value of 1, the default, specifies that the string is editable. A value of 0 specifies that the string is not editable. Text widgets are not editable outside the Layout Editor.

local_range	This property is used with dial widgets when the parameter is unbounded or when the "accumulator" property has a value of 1, allowing the parameter range to extend over multiple rotations of the dial. The value of the "local_range" property is the range that maps to one complete dial rotation. The default is 200.0.
width	This property specifies the width of the widget. The value is an integer between 1 and 10, inclusive, and is interpreted as a multiple of the standard button width, which is approximately 60 pixels. (The application panel is just over 4 units wide.)
height	This property specifies the height of the widget. The value is an integer between 1 and 100, inclusive, and is interpreted as a multiple of the height of a text line.
columns	This property specifies the number of columns of buttons in the widget. The default is 1.

AVSautofree_output

C

```
AVSautofree_output(out_port)
   int          out_port;
```

FORTRAN

```
AVSAUTOFREE_OUTPUT(OUT_PORT)
   INTEGER          OUT_PORT
```

This routine sets a float that frees output data from the previous invocation before invoking the module being defined in the current description function. If neither this routine nor AVSinitialize_output is called, ConvexAVS does not free output data from the previous invocation when it invokes a module. The out_port argument is a port identifier returned by AVScreate_output_port.

AVSbuild_2d_field

C

```
#include <avs/field.h>
AVSfield * AVSbuild_2d_field(data, dim1, dim2)
   float        *data;
   int          dim1, dim2;
```

FORTRAN

```
AVSBUILD_2D_FIELD(DATA, DIM1, DIM2)
   REAL              DATA(DIM1, DIM2)
   INTEGER           DIM1, DIM2
```

This routine builds a two-dimensional uniform scalar real field from its components. The routine returns a pointer to an AVSfield structure. The data argument is the data array, in FORTRAN order. The subscript for the first dimension varies fastest. The dim1 and dim2 arguments are integers specifying the size of the first and second dimensions, respectively.

AVSbuild_3d_field

C

```
#include <avs/field.h>
AVSfield * AVSbuild_3d_field(data, dim1, dim2, dim3)
   float        *data;
   int          dim1, dim2, dim3;
```

FORTRAN

```
AVSBUILD_3D_FIELD(DATA, DIM1, DIM2, DIM3)
   REAL              DATA(DIM1, DIM2, DIM3)
   INTEGER           DIM1, DIM2, DIM3
```

This routine builds a three-dimensional uniform scalar real field from its components. The routine returns a pointer to an AVSfield structure. The data argument is the data array, in FORTRAN order. The subscript for the first dimension varies fastest, then the subscript for the second dimension. The dim1, dim2, and dim3 arguments are integers specifying the size of the first, second, and third dimensions, respectively.

AVSbuild_field

C

```
#include <avs/avs.h>
#include <avs/field.h>
AVSfield * AVSbuild_field(ndim, nspace, veclen, uniform,
           ncoord, type, dim1, dim2, ..., data, coords)
    int              ndim, nspace, veclen, uniform, ncoord,
                     type;
    int              dim1, dim2, ...;
    unsigned char    *data;
    float            *coords;
```

FORTRAN

```
#include <avs/avs.inc>
AVSBUILD_FIELD(NDIM, NSPACE, VECLEN, UNIFORM,
       NCOORD, TYPE, DIM1, DIM2, ..., DATA, COORDS)
    INTEGER          NDIM, NSPACE, VECLEN,
                     UNIFORM, NCOORD, TYPE
    INTEGER          DIM1, DIM2, ...
    BYTE             DATA(*)
    REAL             COORDS(*)
```

This routine constructs a field from its components. The routine returns a pointer to an AVSfield structure. The arguments are:

ndim	A positive integer specifying the number of dimensions in the computational space of the field.
nspace	A positive integer specifying the number of coordinate dimensions.
veclen	A positive integer specifying the length of the data vector at each point. For a scalar field, the value is 1.
uniform	A constant specifying whether the field is uniform, rectilinear, or irregular. Possible values are UNIFORM, RECTILINEAR, and IRREGULAR.
ncoord	An integer specifying the number of dimensions in the coordinate space of non-uniform fields. For uniform fields and rectilinear fields, the value is ignored.

type	A constant specifying the type of data in the field. Possible values are AVS_TYPE_BYTE, AVS_TYPE_INTEGER, AVS_TYPE_REAL, and AVS_TYPE_DOUBLE.
dim1, dim2, ...	For each dimension, an integer specifying the size of the dimension.
data	The data array in FORTRAN order. The subscript for vector element varies fastest, then the subscript for the first dimension, then the subscript for the second dimension, and so on. The storage type for each element depends on the data type of the field.
coords	For a non-uniform field, an array of floating-point values specifying the coordinates of the data points. For a rectilinear field, the length of the array is the sum of the dimensions of the field in computational space. For an irregular field, the length of the array is the product of the dimensions of the field in computational space and the number of dimensions in coordinate space. All the X coordinates are stored first, then all the Y coordinates, and so on. For an irregular field, the subscript for the first field dimension varies fastest. This argument is omitted for uniform fields.

AVSchoice_number

C

```
AVSchoice_number(param_name, string)
   char          *param_name, *string;
```

FORTRAN

```
AVSCHOICE_NUMBER(NAME, STRING)
   CHARACTER*(*)      NAME, STRING
```

This routine is called to interpret a value for a parameter of type "choice" passed to a module computation routine. The param_name argument is the name of the parameter as declared in the call to AVSadd_parameter in the module description function. The string argument is the string passed to the computation function as the value of the parameter.

This routine returns an integer that represents the position of the given choice in the list of choices provided in the call to AVSadd_parameter in the module description function. If the choice is the first in the list, this routine returns 1; if the choice is the second in the list, this routine returns 2; and so on. If the choice is not in the list of choices, this routine returns 0.

A module computation function can also interpret choices by means of direct string comparisons of the parameter argument with expected literal strings.

AVSconnect_widget

C

```
AVSconnect_widget(param_num, widget_type)
   int          param_num;
   char         *widget_type;
```

FORTRAN

```
AVSCONNECT_WIDGET(PARAM_NUM,
                    WIDGET_TYPE)
   INTEGER          PARAM_NUM
   CHARACTER*(*)    WIDGET_TYPE
```

This routine declares a preference that a parameter for a module being defined in the current description function be connected to a specified widget. A parameter can be connected only to a widget that is compatible with the parameter's type. If this routine is called with an impermissible widget type, ConvexAVS ignores the preference and issues a warning.

The param_num argument is a parameter identifier returned by AVSadd_parameter or AVSadd_float_parameter.

The widget_type argument is a string that indicates the type of widget to be connected to the parameter. If widget_type is "none," no widget is connected to the parameter. Table E-3 lists the available widgets for each parameter type. If a parameter type has more than one possible widget, the widget type that appears first is the default. For more information on parameter types, refer to AVSadd_parameter.

Table E-3
Widgets and Parameters

Parameter Type	Widget Type	Widget Description
[any]	none	[No widget.]
integer	idial	Round dial with pointer.
	islider	Fixed-length left-to-right slider; must be bounded.
	typein_integer	Direct typein with title.
boolean	toggle	On/off switch.
tristate	tristate	Variant of toggle that has 3 highlight states.
oneshot	oneshot	Button to request single actions.
real	dial	Round dial with pointer; may be unbounded.
	slider	Fixed-length left-to-right slider; must be bounded.
	typein_real	Direct typein with title.
string	typein	Direct typein with title.
	text	String button, useful for titling; editable only in the Layout Editor.
	browser	File browser. If the string is a path name, the initial directory is set to the directory portion of the path name.
	text_browser	ASCII file browser that displays the file specified by the string. Skips comment lines and filters out embedded nroff directives.
choice	radio_buttons	Set of radio buttons, one for each choice. The value is a copy of the selected string or NULL if no string is selected.
colormap	color_editor	Colormap editor.
field	track	Cursor-tracking virtual trackball.

AVScorout_exec

C

```
AVScorout_exec()
```

FORTRAN

```
AVSCOROUT_EXEC()
```

This routine waits until the flow executive has stopped running, then returns. The routine is useful for delaying output until the network has completely processed the output of the previous computation.

AVScorout_init

C

```
AVScorout_init(argc, argv, desc)
   int         argc;
   char        *argv[];
   int         (*desc)();
```

FORTRAN

```
AVSCOROUT_INIT(DESC)
   EXTERNAL          DESC
```

This routine recognizes and initializes the coroutine as a module and sets up the connection between the coroutine and ConvexAVS. The coroutine must call AVScorout_init before calling any other routines. If this routine is invoked during the module identification pass, it exits. If the routine is invoked during module instantiation, it returns.

In C, the argc and argv arguments are the corresponding arguments to the coroutine main program. The desc argument is a pointer to the module description function. In FORTRAN, the only argument is the module description function.

AVScorout_input

C

```
int AVScorout_input(input1, input2, ..., param1, param2, ...)
   char        **input1, **input2, ...;
   int         *param1, *param2, ...;
```

FORTRAN

```
AVSCOROUT_INPUT(INPUT1, INPUT2, ..., PARAM1,
                                     PARAM2, ...)
   INTEGER           INPUT1, INPUT2, ..., PARAM1,
                     PARAM2, ...
```

A coroutine calls this routine to obtain inputs and parameters from ConvexAVS. There is one argument for each input port and one argument for each parameter declared in the module description function. All the input arguments appear first in the arglist, followed by all the parameter arguments. For most data types, the argument is a pointer to a pointer to a data item of the appropriate type for the input or parameter declared. For some data types, such as integers, the argument is a pointer to the data item itself. When the function returns, each argument location contains a pointer to the corresponding input or parameter value (or the value itself, for data types like integers).

The routine returns 0 if a required input or parameter is missing. Otherwise, it returns the number of inputs and parameters supplied.

AVScorout_output

C

```
AVScorout_output(output1, output2, ...)
   char          *output1, *output2, ...;
```

FORTRAN

```
#include <avs/data.inc>
AVSCOROUT_OUTPUT(OUTPUT1, OUTPUT2, ...)
   INTEGER             OUTPUT1, OUTPUT2, ...
```

A coroutine calls this routine to send output data to ConvexAVS. There is one argument for each output port declared in the module description function. For most data types, the argument is a pointer to a data item of the appropriate type for the output declared. For some data types, such as integers, the argument is the data item itself.

If you have disabled the module or the flow executive, this routine may hang for an arbitrary time before returning.

AVScorout_wait

C

```
AVScorout_wait()
```

FORTRAN

```
AVSCOROUT_WAIT()
```

This routine waits until you change a parameter value or until an upstream module sends more input. It then returns.

AVScreate_input_port

C

```
#include <avs/avs.h>
int AVScreate_input_port(port_name, type, required)
   char          *port_name, *type;
   int           required;
```

FORTRAN

```
#include <avs/avs.inc>
AVSCREATE_INPUT_PORT(NAME, TYPE, REQUIRED)
   CHARACTER*(*)    NAME, TYPE
   INTEGER          REQUIRED
```

This routine declares an input port for the module being defined in the current description function. The name of the port is set to the string port_name. The type argument is a string that defines the data type of the port as shown in Table E-4.

Table E-4
Port Data Types

Data Type	*type* String	Port Color
field	"field"	multi color
colormap	"colormap"	yellow
geometry	"geom"	red
pixel map	"pixmap"	light blue

The "field" string can contain further specializing words. Refer to Table 8-3 in Chapter 8, "Data Types".

The required argument indicates whether or not a connection to the port is required before the module can be invoked. Possible values are REQUIRED, meaning that a connection is required, and OPTIONAL, meaning that a connection is not required.

This routine returns an integer identifier for the port that is used as an argument to some other ConvexAVS routines, such as AVSinitialize_output.

AVScreate_output_port

C

```
int AVScreate_output_port(port_name, type)
   char          *port_name, *type;
```

FORTRAN

```
AVSCREATE_OUTPUT_PORT(NAME, TYPE)
   CHARACTER*(*)      NAME, TYPE
```

This routine declares an output port for the module being defined in the current description function. The name of the port is set to the string port_name. The type argument is a string that defines the data type of the port. For possible values of the type argument, refer to Table 8-3.

This routine returns an integer identifier for the port that is used as an argument to some other ConvexAVS routines, such as AVSinitialize_output.

AVSdata_alloc

C

```
#include <avs/data.h>
char * AVSdata_alloc(string, dims)
   char          *string;
   int           *dims;
```

FORTRAN

```
#include <avs/data.inc>
INTEGER AVSDATA_ALLOC(STRING, DIMS)
   CHARACTER*(*)      STRING
   INTEGER(*)         DIMS
```

This routine is similar to AVSfield_alloc. It takes a character string describing the field, rather than a field template structure. It returns a pointer to a char, which should be cast as a pointer to an AVSfield. For example in C:

```
field = (AVSfield_char *)
AVSdata_alloc("field 2D 4-vector byte", dim_count);
```

AVSdebug

C

```
AVSdebug(message_format, msg1, msg2, msg3,
                    msg4, msg5, msg6)
  char        *message_format;
  char        *msg1, *msg2, *msg3, *msg4, *msg5,
              *msg6;
```

FORTRAN

```
AVSDEBUG(MESSAGE)
  CHARACTER*(*)    MESSAGE
```

This routine interfaces with the AVSmessage routine. It presents a message of severity AVS_Debug.

C language: To produce the message to be presented, ConvexAVS calls `sprintf`(3S) with message_format as the format string and msg1 through msg6 as the arguments. The msg1 through msg6 arguments can be of any type valid for `sprintf`(3S). Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented is the message argument.

This routine presents you with only the default choice, "Ok." It returns no meaningful value.

AVSerror

C

```
AVSerror(message_format, msg1, msg2, msg3,
                    msg4, msg5, msg6)
   char        *message_format;
   char        *msg1, *msg2, *msg3, *msg4, *msg5,
               *msg6;
```

FORTRAN

```
AVSERROR(MESSAGE)
   CHARACTER*(*)    MESSAGE
```

This routine interfaces with the AVSmessage routine. It presents a message of severity AVS_Error.

C language: To produce the message to be presented, ConvexAVS calls `sprintf`(3S) with message_format as the format string and msg1 through msg6 as the arguments. The msg1 through msg6 arguments can be of any type valid for `sprintf`(3S). Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented is the message argument.

This routine presents you with only the default choice, "Ok." It returns no meaningful value.

AVSfatal

C

```
AVSfatal(message_format, msg1, msg2, msg3,
                    msg4, msg5, msg6)
   char        *message_format;
   char        *msg1, *msg2, *msg3, *msg4, *msg5,
               *msg6;
```

FORTRAN

```
AVSFATAL(MESSAGE)
   CHARACTER*(*)    MESSAGE
```

This routine interfaces with the AVSmessage routine. It presents a message of severity AVS_Fatal.

C language: To produce the message to be presented, ConvexAVS calls `sprintf`(3S) with message_format as the format string and msg1 through msg6 as the arguments. The msg1 through msg6 arguments can be of any type valid for `sprintf`(3S). Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented is the message argument.

This routine presents you with only the default choice, "Ok." It returns no meaningful value.

AVSfield_alloc

C

```
#include <avs/field.h>
char * AVSfield_alloc(template, dims)
   AVSfield      *template;
   int           *dims;
```

FORTRAN

```
#include<avs/avs.inc>
INTEGER AVSFIELD_ALLOC(TEMPLATE, DIMS)
   INTEGER            TEMPLATE
   INTEGER(*)         DIMS
```

This routine creates and allocates memory for a field. It returns a pointer to a char, which should be cast as a pointer to an AVSfield.

The template argument is a pointer to a field to be used as a template for creating the new field. The dims argument is an array of integers to be used as the dimensions of the new field in computational space. The length of the array must be the same as the number of dimensions in the template field. The dims argument can also be 0. In this case, the dimensions of the template field are used to create the new field.

This routine copies the nspace, veclen, type, size, and uniform members of the template field to the new field. If the dims argument is 0, it copies the dimensions array of the template field to the new field; otherwise, it copies the dims argument to the dimensions array of the new field. This routine allocates memory for the points array of the new field. If the template field is rectilinear or irregular and if the template field has a points array, this routine copies the points array of the template field to the new field. This routine allocates memory for the data array of the new field but does not copy the data array of the template field to the new field.

The template field can be an existing field, such as an input argument to a module computation routine, or a template created from an existing field by AVSfield_make_template. A template created by AVSfield_make_template is useful when the points array of the template field is not to be copied to the new field.

AVSfield_copy_points

C

```
#include <avs/field.h>
AVSfield_copy_points(field_in, field_out)
   AVSfield      *field_in, *field_out;
```

FORTRAN

```
#include <avs/avs.inc>
AVSFIELD_COPY_POINTS(FIELD_IN, FIELD_OUT)
   INTEGER              FIELD_IN, FIELD_OUT
```

This routine copies the coordinates array from field_in to field_out. Memory must be allocated for the coordinates array in field_out before this routine is called. This routine is useful for passing the coordinates array from an input field to an output field in a module computation routine that operates only on the computational data of a field and ignores the coordinates.

AVSfield_data_offset

C

There is no C language equivalent for this routine.

FORTRAN

```
#include <avs/avs.inc>
AVSFIELD_DATA_OFFSET(FIELD, BASEVEC, OFFSET)
   INTEGER              FIELD
   REAL(*)              BASEVEC
   INTEGER              OFFSET
```

Use this routine to retrieve an offset index of the field data array relative to a given local reference array of <type>. The element *basevec(offset + 1)* is the same as the first element of the data array. For FORTRAN to handle this easier, pass this element to a second FORTRAN function that expects a variable size <type> array. The basevec array should be equivalent to the field data array being retrieved (for example, real to get real data or integer for integer data).

AVSfield_data_ptr

C

```
#include <avs/avs.h>
AVSfield_data_ptr(field)
   AVSfield      *field;
```

FORTRAN

```
#include <avs/avs.inc>
AVSFIELD_DATA_PTR(FIELD)
   INTEGER              FIELD
```

Use this routine to retrieve the direct data pointer from the field structure. To unreference the pointer, pass the %VAL() of the pointer and dimensions to a second FORTRAN routine that declares the incoming argument as a variable size array.

AVSfield_free

C

```
AVSfield_free(field)
   AVSfield      *field;
```

FORTRAN

```
AVSFIELD_FREE(FIELD)
   INTEGER              FIELD
```

This routine deallocates the memory associated with an output field. Use it on previously allocated fields that do not to have memory freed automatically by either AVSinitialize_output or AVSautofree_output.

AVSfield_get_dimensions

C

```
#include <avs/avs.h>
AVSfield_get_dimensions(field, dimensions)
   AVSfield      *field;
   int           *dimensions;
```

FORTRAN

```
#include <avs/avs.inc>
AVSFIELD_GET_DIMENSIONS(FIELD, DIMENSIONS)
   INTEGER            FIELD
   INTEGER            DIMENSIONS()
```

Use this routine to obtain the dimensions of the field data space. Only the first field in nspace elements are copied into the arrays. Ensure that the array passed is large enough for the dimensionality of the field. A return of 1 indicates valid data while a 0 indicates invalid data.

AVSfield_get_int

C

```
#include <avs/avs.h>
AVSfield_get_int(field, selector)
   AVSfield      *field;
   int           selector;
```

FORTRAN

```
#include <avs/avs.inc>
AVSFIELD_GET_INT(FIELD, SELECTOR)
   INTEGER            FIELD
   INTEGER            SELECTOR
```

Use this routine to retrieve one of the integer fields in the field structure. The selector should be one of the following:

- AVS_FIELD_NDIM
- AVS_FIELD_NSPACE
- AVS_FIELD_VECLEN
- AVS_FIELD_TYPE
- AVS_FIELD_SIZE
- AVS_FIELD_UNIFORM

AVSfield_make_template

C

```
#include <avs/field.h>
AVSfield_make_template(field_in, template)
   AVSfield        *field_in, *template;
```

This routine copies the ndim, nspace, veclen, type, size, and uniform members of field_in into template. It allocates memory for the dimensions array of the template field and copies the dimensions array of field_in to the template field. This routine does not allocate memory for the data and points arrays of the template field; it sets the value of those members of the template field to NULL.

This routine is intended to use an existing field, such as an input argument to a module computation routine, to create a template for AVSfield_alloc. The template argument can be created as follows:

```
AVSfield *template;
template = (AVSfield *) malloc(sizeof(AVSfield));
```

FORTRAN

There is no FORTRAN equivalent for this routine.

AVSfield_points_offset

C

There is no C language equivalent for this routine.

FORTRAN

```
#include <avs/avs.inc>
AVSFIELD_POINTS_OFFSET(FIELD, BASEVEC, OFFSET)
   INTEGER         FIELD
   REAL(*)         BASEVEC
   INTEGER         OFFSET
```

Use this routine to retrieve an offset index of the field points array relative to a given local reference array of <type>. The element *basevec(offset + 1)* is the same as the first element of the points array. For FORTRAN to handle this easier, pass this element to a second FORTRAN function that expects a variable size <type> array.

AVSfield_points_ptr

C

```
#include <avs/avs.h>
AVSfield_points_ptr(field)
   AVSfield      *field;
```

FORTRAN

```
#include <avs/avs.inc>
AVSFIELD_POINTS_PTR(FIELD)
   INTEGER              FIELD
```

Use this routine to retrieve the direct points array pointer from the field structure. To unreference the pointer, pass the %VAL() of the pointer and dimensions to a second FORTRAN routine that declares the incoming argument as a variable size real array.

AVSinformation

C

```
AVSinformation(message_format, msg1, msg2, msg3,
                                msg4, msg5, msg6)
   char          *message_format;
   char          *msg1, *msg2, *msg3, *msg4, *msg5,
                 *msg6;
```

FORTRAN

```
AVSINFORMATION(MESSAGE)
   CHARACTER*(*)     MESSAGE
```

This routine interfaces with the AVSmessage routine. It presents a message of severity AVS_Information.

C language: To produce the message to be presented, ConvexAVS calls `sprintf`(3S) with message_format as the format string and msg1 through msg6 as the arguments. The msg1 through msg6 arguments can be of any type valid for `sprintf`(3S). Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented is the message argument.

This routine presents you with no choices and returns no meaningful value.

AVSinit_from_module_list

C

```
AVSinit_from_module_list(AVSmodule_list, count)
   int          (**AVSmodule_list)();
   int          count;
```

AVSinit_from_module_list initializes a list of modules from their description functions. The AVSmodule_list argument is a list of pointers, one to each module description function defined in the file. The count argument is the number of pointers in the list.

For modules written in C, each file can define more than one module. AVSinit_modules should contain one call to AVSmodule_from_desc to initialize each module defined in the file. Alternately, AVSinit_modules can call AVSinit_from_module_list to initialize a list of modules defined in the file.

FORTRAN

There is no FORTRAN equivalent for this routine. In FORTRAN, the module description function itself is called AVSINIT_MODULES.

AVSinit_modules

C

AVSinit_modules()

FORTRAN

AVSINIT_MODULES()

You define this routine. ConvexAVS invokes this routine when it loads the modules defined in a file. Each executable file that defines subroutine modules should have one and only one definition for AVSinit_modules. The definition differs depending on whether the module source is C or FORTRAN:

- In C, each file can define more than one module. AVSinit_modules should contain one call to AVSmodule_from_desc to initialize each module defined in the file. Alternately, AVSinit_modules can call AVSinit_from_module_list to initialize a list of modules defined in the file.
- In FORTRAN, each file can define only one module. The module description function itself should be called AVSINIT_MODULES.

A file that defines a coroutine should not have a definition for this routine. A coroutine calls AVScorout_init from its main program instead.

AVSinitialize_output

C

```
AVSinitialize_output(in_port, out_port)
   int            in_port, out_port;
```

FORTRAN

```
AVSINITIALIZE_OUTPUT(IN_PORT, OUT_PORT)
   INTEGER              IN_PORT, OUT_PORT
```

This routine sets a float that allocates memory for output data before invoking the module being defined in the current description function. Before each invocation of the module, ConvexAVS frees output data from the previous invocation, then allocates space for an output data structure of the same size and dimensions as those of the specified input data structure. ConvexAVS does not copy the input data to the output data; this is useful for modules that transform fields, producing an output field of the same type and dimensions as the input field. The in_port argument is a port identifier returned by AVScreate_input_port. The out_port argument is a port identifier returned by AVScreate_output_port.

AVSinput_changed

C

```
int AVSinput_changed(port_name, i)
   char           *port_name;
   int            i;
```

FORTRAN

```
AVSINPUT_CHANGED(PORT_NAME, I)
   CHARACTER*(*)     PORT_NAME
   INTEGER           I
```

For a subroutine, it determines whether or not input data has changed since the previous invocation of the module. For a coroutine, it determines whether a new value was returned in the last call to AVScorout_input. The port_name argument is the name of the input port as declared in the module description function. The second argument is the number of a connection to that port; the first connection is 0 for the C routine and 1 for the FORTRAN routine. AVSinput_changed returns 1 if the input data has changed for the specified port and connection. It returns 0 if the input has not changed or if the specified connection does not exist.

AVSmark_output_unchanged

C

```
AVSmark_output_unchanged(port_name)
   char           *port_name;
```

FORTRAN

```
AVSMARK_OUTPUT_UNCHANGED(PORT_NAME)
   CHARACTER*(*)     PORT_NAME
```

By default, ConvexAVS assumes that all output data has changed after each invocation of a module. This can cause ConvexAVS to invoke downstream modules. AVSmark_output_unchanged tells ConvexAVS that output data for a port has not changed. The port_name argument is the name of the output port as declared in the module description function.

AVSmessage

C

```
#include <avs/avs.h>
char * AVSmessage(version, severity, module,
         function, choices, message_format,
         msg1, msg2, msg3, msg4, msg5, msg6)
   char           *version;
   AVS_MESSAGE_SEVERITY      severity;
   char           *module;
   char           *function_name, *choices,
                  *message_format;
   char           *msg1, *msg2, *msg3, *msg4, *msg5,
                  *msg6;
```

FORTRAN

```
#include <avs/avs.inc>
AVSMESSAGE(VERSION, SEVERITY, MODULE,
         FUNCTION_NAME, CHOICES, MESSAGE)
   CHARACTER*(*)     VERSION
   INTEGER           SEVERITY
   CHARACTER*(*)     MODULE, FUNCTION_NAME
   CHARACTER*(*)     CHOICES, MESSAGE
```

This routine presents information about the module and function sending the message.

version — A string indicating what version of the module is reporting the error. This can be any string, but it should be a meaningful identification for the code developer.

In some source code management systems, updating the version string can be handled automatically. Under the source code control system (SCCS), for example, you can insert a line into a C source file declaring a global string variable that matches SCCS ID keywords. The string is updated each time a delta is made. For example:

```
static char file_version[] = "%W% %E%";
```

severity A value indicating the relative importance of the message being sent. This determines how ConvexAVS presents the message to you and whether or not you must acknowledge the message before ConvexAVS can continue. If the message appears in a dialog box, the border of the dialog box is color coded to indicate the severity. Following are the possible values:

AVS_Information

The message does not indicate an error. The message is written to stderr, and ConvexAVS continues executing. No choices are presented.

AVS_Debug

The message does not indicate an error; it conveys information during module testing. The message is written to stderr, and ConvexAVS continues executing. No choices are presented.

AVS_Warning

The message indicates a problem that is not fatal to module execution. The message and choices are presented in a dialog box with a yellow border. You must make a choice before ConvexAVS can continue.

AVS_Error

The message indicates a serious problem that is not fatal to module execution. The message and choices are presented in a dialog box with a red border. You must make a choice before ConvexAVS can continue.

AVS_Fatal

The message indicates a problem that is fatal to module execution. The message and choices are presented in a dialog box with a black border. You must make a choice before ConvexAVS can continue. The module is marked as dead, and the module icon in the Network Editor workspace turns black. The flow executive no longer executes the module.

module
: The module sending the message. This value should always be NULL in C (0 in FORTRAN). ConvexAVS automatically identifies the module sending a message and highlights its icon in yellow.

function
: The name of the function sending the message.

choices
: A string containing the names of options to be presented to you. The choices are separated by exclamation points (!). For example, "Ok!Kill Module!Exit" is presented as three choices: "Ok," "Kill Module," and "Exit." If the value is NULL in C (0 in FORTRAN) or the empty string, ConvexAVS presents a default choice, "Ok." ConvexAVS can add choices to those specified in the choices argument.

message_format, msg1, msg2, msg3, msg4, msg5, msg6
: C: To produce the message to be presented, ConvexAVS calls `sprintf`(3S) with message_format as the format string and msg1 through msg6 as the arguments. The msg1 through msg6 arguments can be of any type valid for `sprintf`(3S). Only as many arguments as the format string requires need be supplied.

  FORTRAN: The message to be presented is the message argument.

AVSmessage returns a string containing the choice made. A C language routine can use `strcmp`(3C) to identify the choice, as shown in the following example:

```
char *answer;
answer = AVSmessage(...,"Ok!Reset!Exit", ...)
if (!strcmp(answer,"Reset")) {/*reset action*/}
else if (!strcmp(answer,"Exit") { exit(1); }
```

A FORTRAN routine should declare AVSMESSAGE to return CHARACTER*n, where n is the maximum length of the string to be returned. The string is padded on the right with spaces. The routine can use the .EQ. operator to identify the choice, as in this example:

```
    ...
    EXTERNAL AVSMESSAGE
    CHARACTER*32 AVSMESSAGE
    CHARACTER*32 RESPONSE
    RESPONSE=AVSMESSAGE('Version 1',AVS_Error,0,
    + 'MY_ROUTINE', 'Ok!Reset!Exit',
    + 'Attempt to divide by zero.')
    IF (RESPONSE(1:2) .EQ. 'Ok') THEN
C   Process 'Ok' choice
    ELSE IF (RESPONSE(1:5) .EQ. 'Reset') THEN
C   Process 'Reset' choice
    ELSE IF (RESPONSE(1:4) .EQ. 'Exit') THEN
C   Process 'Exit' choice
    ELSE
C   Process other choices added by AVS
    END IF
    ...
```

Because ConvexAVS can add choices to those supplied in the choices argument, the returned value might not be one of the substrings in choices. For messages of severity AVS_Information and AVS_Debug, no choices are presented, and the returned value is the empty string.

AVSmodify_float_parameter

C

```
#include <avs/avs.h>
AVSmodify_float_parameter(param_name, flags, init,
                                    minval, maxval)
   char       *param_name;
   int        flags;
   double     init, minval, maxval;
```

This routine is called from a module computation routine to change the value or bounds of a parameter of type "real." The routine interfaces with the AVSmodify_parameter routine; it allocates space for the init, minval, and maxval arguments automatically. The calling routine should declare these arguments as float. In C, when a float is passed as an argument it is converted to a double.

FORTRAN

There is no FORTRAN equivalent for this routine. Use AVSMODIFY_PARAMETER instead.

AVSmodify_parameter

C

```
#include <avs/avs.h>
AVSmodify_parameter(param_name, flags, init, minval,
                                              maxval)
   char        *param_name;
   int         flags, init, minval, maxval;
```

FORTRAN

```
#include <avs/avs.inc>
AVSMODIFY_PARAMETER(NAME, FLAGS, INIT,
                              MINVAL, MAXVAL)
   CHARACTER*(*)     NAME
   INTEGER           FLAGS, INIT, MINVAL,
                     MAXVAL
```

This routine is called from a module computation routine to change the value or bounds of a parameter. ConvexAVS first updates the parameter bounds, then checks the new or existing value for validity against the new bounds. If a widget is connected to the parameter, the widget is then updated to reflect the new parameter bounds and value.

The param_name argument is the name of the parameter as declared in the call to AVSadd_parameter or AVSadd_float_parameter in the module description function.

The flags argument is a bit mask indicating which combination of value, upper bound, and lower bound is to be changed. ConvexAVS defines the following constants corresponding to the three items to be changed:

AVS_VALUE	The init argument contains a new value for the parameter.
AVS_MINVAL	The minval argument contains a new minimum value for the parameter.
AVS_MAXVAL	The maxval argument contains a new maximum value for the parameter.

These constants can be combined using a bitwise OR operation to change more than one item at a time. For example, to change the value and upper bound but not the lower bound:

```
      /* C language */
      flags = AVS_VALUE | AVS_MAXVAL;
C     FORTRAN
      INTEGER FLAGS
      FLAGS = IOR(AVS_VALUE, AVS_MAXVAL)
```

ConvexAVS changes the value or a bound of a parameter only if the corresponding bit in the flags argument is on, or if a change in the bounds requires changing the current value of the parameter to be within the new bounds.

The init, minval, and maxval arguments are interpreted in the same way as the corresponding arguments to AVSadd_parameter. The meaning and type of these arguments depend on the parameter type. For more information, refer to AVSadd_parameter. If the call to AVSmodify_parameter does not change the value, lower bound, or upper bound, the corresponding init, minval, or maxval argument should be 0 in C and FORTRAN.

Note

The arguments to the module computation routine are essentially a snapshot of the parameter values at the time the computation routine is called. This means that AVSmodify_parameter affects the value and range of the parameter the next time the computation routine is called -- it does not necessarily affect the corresponding argument value within the current invocation of the routine. (It may in some cases -- floats and strings, in particular.)

If you intend to perform further computations on an argument whose corresponding parameter you change with AVSmodify_parameter: make a local copy of the argument; before calling AVSmodify_parameter, apply the same changes to the copy argument; perform further computations with the copy, not the original.

AVSmodule_from_desc

C

```
AVSmodule_from_desc(desc)
   int          (*desc)();
```

AVSmodule_from_desc initializes a module from its description function. The desc argument is a pointer to the description function.

For modules written in C, each file can define more than one module. AVSinit_modules should contain one call to AVSmodule_from_desc to initialize each module defined in the file. Alternately, AVSinit_modules can call AVSinit_from_module_list to initialize a list of modules defined in the file.

FORTRAN

There is no FORTRAN equivalent for this routine. In FORTRAN, the module description function is called AVSINIT_MODULES.

AVSparameter_changed

C

```
int AVSparameter_changed(param_name)
   char            *param_name;
```

FORTRAN

```
AVSPARAMETER_CHANGED(PARAM_NAME)
   CHARACTER*(*)     PARAM_NAME
```

For a subroutine, it determines whether or not a parameter value has changed since the previous invocation of the module. For a coroutine, it determines whether a new value was returned in the last call to AVScorout_input. The param_name argument is the name of the parameter as declared in the module description function. AVSparameter_changed returns 1 if the parameter value has changed. It returns 0 if the parameter value has not changed.

AVSset_compute_proc

C

```
AVSset_compute_proc(comp_func)
   int             (*comp_func)();
```

FORTRAN

```
AVSSET_COMPUTE_PROC(COMP_FUNC)
   EXTERNAL          COMP_FUNC
```

This routine declares the computation function for the module being defined in the current description function.

AVSset_destroy_proc

C

```
AVSset_destroy_proc(destroy_func)
   int             (*destroy_func)();
```

FORTRAN

```
AVSSET_DESTROY_PROC(DESTROY_FUNC)
   EXTERNAL          DESTROY_FUNC
```

This routine declares the destruction function for the module being defined in the current description function. ConvexAVS invokes the destruction function when the module is destroyed, usually when you move the module icon from the Network Editor workspace to the hammer icon. A destruction function might take actions such as freeing memory or destroying a window.

AVSset_init_proc

C

```
AVSset_init_proc(init_func)
   int              (*init_func)();
```

FORTRAN

```
AVSSET_INIT_PROC(INIT_FUNC)
   EXTERNAL             INIT_FUNC
```

This routine declares the initialization function for the module being defined in the current description function. ConvexAVS invokes the initialization function when the module is instantiated, usually when you move the module icon from the Network Editor module palette into the workspace. An initialization function might take actions such as allocating memory or creating a window.

AVSset_module_flags

C

There is no C equivalent for this routine.

FORTRAN

```
#include <avs/avs.inc>
AVSSET_MODULE_FLAGS(FLAGS)
   INTEGER              FLAGS
```

This routine should be called from the description function of FORTRAN modules that process fields. This routine tells ConvexAVS that fields are passed into FORTRAN computation functions as integers instead of split up into different parts (an obsolete method of handling fields in FORTRAN). Use this routine with the SINGLE_ARG_DATA flag.

AVSset_module_name

C

```
#include <avs/avs.h>
AVSset_module_name(name, type)
   char             *name;
   int              type;
```

FORTRAN

```
AVSSET_MODULE_NAME(NAME, TYPE)
   CHARACTER*(*)        NAME, TYPE
```

This routine declares the name and type of the module being defined in the current description function. The module name is set to the string name and the type to type, where type is one of the following:

Module Type	C Constant	FORTRAN String
Data	MODULE_DATA	'data'
Filter	MODULE_FILTER	'filter'
Mapper	MODULE_MAPPER	'mapper'
Renderer	MODULE_RENDER	'renderer'

The module name appears in the module icon and other portions of the Network Editor and Application Builder user interface. The module type determines the category in the Network Editor module palette in which the module icon appears.

AVSwarning

C

```
AVSwarning(message_format, msg1, msg2, msg3,
                              msg4, msg5, msg6)
   char        *message_format;
   char        *msg1, *msg2, *msg3, *msg4, *msg5,
               *msg6;
```

FORTRAN

```
AVSWARNING(MESSAGE)
   CHARACTER*(*)    MESSAGE
```

This routine interfaces with the AVSmessage routine. It presents a message of severity AVS_Warning.

C language: To produce the message to be presented, ConvexAVS calls `sprintf`(3S) with message_format as the format string and msg1 through msg6 as the arguments. The msg1 through msg6 arguments can be of any type valid for `sprintf`(3S). Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented is the message argument.

This routine presents you with only the default choice, "Ok." It returns no meaningful value.

C Language Field Macros

F

Grouped by Function

The following section groups macros by functions:

- Field Dimensions
  - MAXX
  - MAXY
  - MAXZ
- Elements of a Scalar Data Array
  - I2D
  - I3D
  - I4D
- Elements of a Vector Data Array
  - I1DV
  - I2DV
  - I3DV
  - I4DV
- Rectilinear Coordinate Arrays
  - RECT_X
  - RECT_Y
  - RECT_Z
- Coordinates for 3D Data Elements
  - COORD_X_3D
  - COORD_Y_3D
  - COORD_Z_3D

Defined Alphabetically

The following section lists and defines macros alphabetically.

COORD_X_3D

```
#include <avs/field.h>
COORD_X_3D(field, i, j, k)
    AVSfield    *field;
    int         i, j, k;
```

For a three-dimensional uniform field, COORD_X_3D returns i. For a three-dimensional rectilinear or irregular field, COORD_X_3D provides the X-coordinate from the coordinate array that corresponds to the data element specified by the indexes i, j, and k.

COORD_Y_3D

```
#include <avs/field.h>
COORD_Y_3D(field, i, j, k)
    AVSfield    *field;
    int         i, j, k;
```

For a three-dimensional uniform field, COORD_Y_3D returns j. For a three-dimensional rectilinear or irregular field, COORD_Y_3D provides the Y-coordinate from the coordinate array that corresponds to the data element specified by the indexes i, j, and k.

COORD_Z_3D

```
#include <avs/field.h>
COORD_Z_3D(field, i, j, k)
    AVSfield    *field;
    int         i, j, k;
```

For a three-dimensional uniform field, COORD_Z_3D returns k. For a three-dimensional rectilinear or irregular field, COORD_Z_3D provides the Z-coordinate from the coordinate array that corresponds to the data element specified by the indexes i, j, and k.

I1DV

```
#include <avs/field.h>
I1DV(field, i)
    AVSfield    *field;
    int         i;
```

For a one-dimensional field, I1DV provides a pointer to the first element of the vector in the data array that corresponds to index i.

I2D

```
#include <avs/field.h>
I2D(field, i, j)
    AVSfield    *field;
    int         i, j;
```

For a two-dimensional field, I2D provides the element of the data array that corresponds to index i of the first dimension and index j of the second dimension. Note that the index arguments are in order of the field dimensions. If the indexes were used directly as subscripts into the data array, they would be in reverse order.

I2DV

```
#include <avs/field.h>
I2DV(field, i, j)
    AVSfield    *field;
    int         i, j;
```

For a two-dimensional field, I2DV provides a pointer to the first element of the vector in the data array that corresponds to index i of the first dimension and index j of the second dimension. Note that the index arguments are in order of the field dimensions. If the indexes were used directly as subscripts into the data array, they would be in reverse order, with the vector index as the last subscript.

I3D

```
#include <avs/field.h>
I3D(field, i, j, k)
    AVSfield     *field;
    int          i, j, k;
```

For a three-dimensional field, I3D provides the element of the data array that corresponds to index i of the first dimension, index j of the second dimension, and index k of the third dimension. Note that the index arguments are in order of the field dimensions. If the indexes were used directly as subscripts into the data array, they would be in reverse order.

I3DV

```
#include <avs/field.h>
I3DV(field, i, j, k)
    AVSfield     *field;
    int          i, j, k;
```

For a three-dimensional field, I3DV provides a pointer to the first element of the vector in the data array that corresponds to index i of the first dimension, index j of the second dimension, and index k of the third dimension. Note that the index arguments are in order of the field dimensions. If the indexes were used directly as subscripts into the data array, they would be in reverse order, with the vector index as the last subscript.

I4D

```
#include <avs/field.h>
I4D(field, i, j, k, l)
    AVSfield     *field;
    int          i, j, k, l;
```

For a four-dimensional field, I4D provides the element of the data array that corresponds to index i of the first dimension, index j of the second dimension, index k of the third dimension, and index l of the fourth dimension. Note that the index arguments are in order of the field dimensions. If the indexes were used directly as subscripts into the data array, they would be in reverse order.

I4DV

```
#include <avs/field.h>
I4DV(field, i, j, k, l)
   AVSfield     *field;
   int          i, j, k, l;
```

For a four-dimensional field, I4DV provides a pointer to the first element of the vector in the data array that corresponds to index i of the first dimension, index j of the second dimension, index k of the third dimension, and index l of the fourth dimension. Note that the index arguments are in order of the field dimensions. If the indexes were used directly as subscripts into the data array, they would be in reverse order, with the vector index as the last subscript.

MAXX

```
#include <avs/field.h>
MAXX(field)
   AVSfield     *field;
```

MAXX provides the size of the first dimension of a field.

MAXY

```
#include <avs/field.h>
MAXY(field)
   AVSfield     *field;
```

MAXY provides the size of the second dimension of a field.

MAXZ

```
#include <avs/field.h>
MAXZ(field)
   AVSfield     *field;
```

MAXZ provides the size of the third dimension of a field.

RECT_X

```
#include <avs/field.h>
RECT_X(field)
    AVSfield      *field;
```

For a rectilinear field, RECT_X provides a pointer to the first element of the coordinate array that corresponds to the first dimension of computational space.

RECT_Y

```
#include <avs/field.h>
RECT_Y(field)
    AVSfield      *field;
```

For a rectilinear field, RECT_Y provides a pointer to the first element of the coordinate array that corresponds to the second dimension of computational space.

RECT_Z

```
#include <avs/field.h>
RECT_Z(field)
    AVSfield      *field;
```

For a rectilinear field, RECT_Z provides a pointer to the first element of the coordinate array that corresponds to the third dimension of computational space.

Geometry Routines

The GEOM library provides a set of routines for creating, modifying, and displaying 3D shaded graphics. Module developers can use these routines to pass geometric descriptions of their data to the ConvexAVS general renderers.

The basic entity in the geom package is the geom object. The supported types are:

- GEOM_LABEL
- GEOM_MESH
- GEOM_POLYHEDRON
- GEOM_POLYTRI
- GEOM_SPHERE

Label	A label contains one or more text strings normally used to label a ConvexAVS object or view. The label is presented as annotation text: its position can be transformed, but the text always appears upright in a plane parallel to the display surface.
Mesh	A mesh object contains one, two-dimensional array of vertices.
Polyhedron	A polyhedron contains a list of vertices and a list of polygons, which are defined by indirectly referencing the vertices that the polygon contains.
Polytriangle	A polytriangle object contains a list of polytriangle strips, a list of polylines, or a list of disjoint lines.
Sphere	A sphere object is an array of spheres containing a list of points and a corresponding list of radii.

Grouped by Topic

The following section groups geometry routines by topic.

Object Creation Routines

The following routines create new geometry objects and are grouped by the type of geometry object they operate upon. Object Modification Routines may be needed afterwards to complete the geometric description before inserting the object into an edit list. GEOMcreate routines return a pointer to GEOMobj of the appropriate type. GEOMadd routines take a GEOMobj pointer as an argument, and further modify the description.

Generic - GEOMobj

- GEOMcreate_obj
- GEOMdestroy_obj

Lines and Polytriangles - GEOM_POLYTRI

- GEOMadd_disjoint_line
- GEOMadd_polyline
- GEOMadd_polytriangle
- GEOMcreate_normal_object

Quadrilateral Meshes - GEOM_MESH

- GEOMcreate_mesh
- GEOMcreate_mesh_with_data
- GEOMcreate_scalar_mesh

Polygons and Polyhedrons - GEOM_POLYHEDRON

- GEOMcreate_polyh
- GEOMcreate_polyh_with_data
- GEOMadd_polygon
- GEOMadd_polygons
- GEOMadd_disjoint_polygon

Spheres - GEOM_SPHERE

- GEOMcreate_sphere
- GEOMadd_radii

Text Labels - GEOM_LABEL

- GEOMcreate_label
- GEOMcreate_label_flags
- GEOMadd_label

Object Modification Routines

The following routines modify geometry objects that have already been created and have not yet been placed in an edit list.The objects are referenced by the GEOMobj pointer, which is returned by the creation function.

Color - All GEOMobj except GEOM_POLYTRI

- GEOMadd_float_colors
- GEOMadd_int_colors

Normals - GEOM_MESH and GEOM_POLYHEDRON

- GEOMflip_normals
- GEOMgen_normals
- GEOMnormalize_normals
- GEOMadd_normals

Vertices - GEOM_MESH and GEOM_POLYHEDRON

- GEOMadd_vertices
- GEOMadd_vertices_with_data

Type Conversion - GEOM_POLYTRI

- GEOMcvt_mesh_to_polytri
- GEOMcvt_polyh_to_polytri

Object Transformation Routines

Automatic Placement - All GEOMobj

- GEOMauto_transform
- GEOMauto_transform_list
- GEOMauto_transform_non_uniform
- GEOMauto_transform_non_uniform_list

Extent - All GEOMobj

- GEOMset_extent
- GEOMset_computed_extent
- GEOMunion_extents

Edit List Manipulation

The following routines operate upon edit lists that are type GEOMedit_list. Objects are referenced by the name string they were given upon insertion to the edit list by the GEOMedit_geometry routine.

Creation - GEOMedit_list

- GEOMinit_edit_list
- GEOMdestroy_edit_list

Object Insertion - All GEOMobj

- GEOMedit_geometry

Object Transformation - All GEOMobj, Light, and Camera

- GEOMedit_set_matrix
- GEOMedit_concat_matrix

Object Hierarchy - All GEOMobj

- GEOMedit_parent
- GEOMedit_picked

Object Properties - All GEOMobj

- GEOMedit_properties
- GEOMedit_color
- GEOMedit_render_mode
- GEOMedit_visibility

Light Source - Light

- GEOMedit_light

Geometry File Utilities

Input and Output - All GEOMobj

- GEOMread_obj
- GEOMwrite_obj
- GEOMwrite_text

Object management- All GEOMobj

- GEOMset_object_group
- GEOMset_pickable

Listed Alphabetically

The following section lists geometry routines alphabetically. Their parameters are also listed along with all possible values.

GEOMadd_disjoint_line

```
GEOMadd_disjoint_line(obj, verts, colors, n, alloc)
GEOMobj         *obj;
float           verts[n][3];
float           colors[n][3];
int             n;
int             alloc;
```

Add an array of disjoint line segments to an existing object of type GEOM_POLYTRI. These lines will be added to any lines already present in the object.

*obj	An existing object of type GEOM_POLYTRI.	
verts[*n*][3]	An array of disjoint-line pairs of vertex coordinates (X, Y, Z).	
colors[*n*][3]	An array of pairs of RGB colors with each color corresponding to a vertex.	
	GEOM_NULL	The lines will be white. You cannot use either GEOMadd_float_colors or GEOMadd_int_colors with this data type.
n	The number of vertices (an even value because a disjoint line has two vertices).	
alloc	GEOM_COPY_DATA Make a copy of the data for internal use.	
	GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.	

GEOMadd_disjoint_polygon

```
GEOMadd_disjoint_polygon(obj, verts, normals, colors, n,
                                        flag, alloc)
GEOMobj         *obj;
float           verts[n][3];
float           normals[n][3];
float           colors[n][3];
int             n;
int             flag;
int             alloc;
```

Add an array of vertices defining a disjoint polygon to an existing object of type GEOM_POLYHEDRON. These polygons will be added to any polygons present in the object.

*obj	An existing object of type GEOM_POLYHEDRON.	
verts[*n*][3]	An array of vertex coordinates (X, Y, Z).	
normals[*n*][3]	An array of normals (X, Y, Z) with each normal corresponding to a vertex.	
	GEOM_NULL	Add normals later with GEOMadd_normals.
colors[*n*][3]	An array of RGB colors with each color corresponding to a vertex.	
	GEOM_NULL	Add colors later with GEOMadd_float_colors.
n	The number of vertices.	
alloc	GEOM_COPY_DATA Make a copy of the data for internal use.	
	GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.	
flag	GEOM_SHARED Compare each vertex to all existing vertices present in the object. If a shared vertex is found, a reference is used rather than duplicating the vertex. This is a slow process but yields smooth objects.	
	GEOM_NOT_SHARED Do not check for shared vertices. This is a fast process but yields faceted objects.	

GEOMadd_float_colors

```
GEOMadd_float_colors(obj, colors, n, alloc)
GEOMobj         *obj;
float           colors[n][3];
int             n;
int             alloc;
```

Add an array of RGB colors to an object.

*obj	An existing object of type GEOM_LABEL, GEOM_MESH, GEOM_POLYHEDRON, or GEOM_SPHERE.

colors[*n*][3]	An array of RGB colors with each color corresponding to a vertex.
n	The number of RGB triples present.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_int_colors

```
GEOMadd_int_colors(obj, colors, n, alloc)
GEOMobj        *obj;
int            colors[n];
int            n;
int            alloc;
```

Add a list of integer-packed RGB colors to an object

*obj	An existing object of type GEOM_LABEL, GEOM_MESH, GEOM_POLYHEDRON, or GEOM_SPHERE.
colors[*n*]	An array of integer-packed RGB colors (packed into 4 bytes). Each byte value ranges from 0 to 255. The byte order (low order byte on the right) is:

unused	red	green	blue

n	The number of colors to be added.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_label

```
GEOMadd_label(obj, text, ref_point, offset, height, color,
                                                label_flags)
GEOMobj        *obj;
char           *text;
float          ref_point[3];
float          offset[3];
float          height;
float          color[3];
int            label_flags;
```

Add a text string and related characteristics to an existing GEOM_LABEL object. This text string will be added to any strings already present in the GEOM_LABEL object. Labels always appear upright and are read from left to right. All labels except title labels (specified by GEOMcreate_label_flags) are transformed with the view. Title labels always appear in the same place.

*obj	An existing object of type GEOM_LABEL
*text	The text string to be displayed as graphic text.
ref_point[3]	A reference point coordinate (X, Y, Z) in screen space. If the label is a title (specified by GEOMcreate_label_flags), the lower left corner is (-1,-1), and the upper right front corner is (1,1). The Z-coordinate is not used.
offset[3]	An offset (X, Y, Z) that is applied after the reference point is transformed.
height	The height of the label given as a screen space Y-coordinate.
color[3]	An RGB triple with values ranging from 0.0 to 1.0. GEOM_NULL — Use the foreground color that is usually white.
label_flags	An integer returned by GEOMcreate_label_flags. 0 — Use the following default values: font: Plain Roman title: Not a title background: No background drop: No drop shadow align: GEOM_LABEL_LEFT

GEOMadd_normals

```
GEOMadd_normals(obj, normals, n, alloc)
GEOMobj         *obj;
float           normals[n][3];
int             n;
int             alloc;
```

Add an array of normals to an object.

*obj	An existing object of type GEOM_MESH or GEOM_POLYHEDRON.

normals[*n*][3]	An array of normals (X, Y, Z) with each normal corresponding to a vertex.
n	The number of normal coordinates.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_polygon

```
GEOMadd_polygon(obj, n, indices, flags, alloc)
GEOMobj         *obj;
int             n;
int             indices[n];
int             flags;
int             alloc;
```

Add a single polygon index list to a polyhedron object.

*obj	An existing object of type GEOM_POLYHEDRON.
n	The number of indices into the vertex array.
indices[*n*]	An array of indices into the object vertex array. The vertex array is created by GEOMadd_vertices and may be called before or after GEOMadd_polygon. GEOMadd_polygon will always reference the first vertex array added. The first vertex in the list is referenced by the value 1 rather than 0.
flags	0 This parameter is not currently implemented.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_polygons

```
GEOMadd_polygons(obj, plist, flags, alloc)
GEOMobj         *obj;
int             *plist;
int             flags;
int             alloc;
```

Add an index list with multiple polygon definitions to a polyhedron object.

*obj	An existing object of type GEOM_POLYHEDRON.
*plist	A list of vertex indices for multiple polygons. For each polygon, specify the number of indices, then the indices. Terminate the list with an index count of zero. The first vertex is indexed by the value 1, not 0.
flags	0 This parameter is not currently implemented.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_polyline

```
GEOMadd_polyline(obj, verts, colors, n, alloc)
GEOMobj         *obj;
float           verts[n][3];
float           colors[n][3];
int             n;
int             alloc;
```

Add an array of points defining a continuous line. This is the most efficient way to draw connected lines.

*obj	An existing object of type GEOM_POLYTRI.
verts[*n*][3]	An array of vertex coordinates (X,Y, Z).
colors[*n*][3]	An array of RGB colors with each color corresponding to a vertex.
n	The number of vertices.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_polytriangle

```
GEOMadd_polytriangle(obj, verts, normals, colors, n, alloc)
GEOMobj         *obj;
float           verts[n][3];
float           normals[n][3];
float           colors[n][3];
int             n;
int             alloc;
```

Add an array of vertices and normals defining a polytriangle strip.

*obj — An existing object of type GEOM_POLYTRI.

verts[*n*][3] — An array of vertex coordinates (X,Y, Z).

normals[*n*][3] — An array of normals (X,Y, Z) with each normal corresponding to a vertex.

GEOM_NULL — Add normals later with GEOMadd_normals.

colors[*n*][3] — An array of RGB colors with each color corresponding to a vertex.

GEOM_NULL — Add colors later with GEOMadd_float_colors.

n — The number of vertices.

alloc — GEOM_COPY_DATA
Make a copy of the data for internal use.

GEOM_DONT_COPY_DATA
Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_radii

```
GEOMadd_radii(obj, radii, n, alloc)
GEOMobj         *obj;
float           radii[n];
int             n;
int             alloc;
```

Add or modify the radii of an array of spheres.

*obj — An existing object of type GEOM_SPHERE.

radii[*n*] — An array of radii, one for each sphere in the object.

n — The number of spheres in the object.

alloc — GEOM_COPY_DATA
Make a copy of the data for internal use.

GEOM_DONT_COPY_DATA
Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_vertices

```
GEOMadd_vertices(obj, verts, n, alloc)
GEOMobj          *obj;
float            verts[n][3];
int              n;
int              alloc;
```

Add an array of vertices to an object.

*obj — An existing object of type GEOM_MESH or GEOM_POLYHEDRON

verts[*n*][3] — An array of vertex coordinates (X, Y, Z).

n — The number of vertices.

alloc — GEOM_COPY_DATA
Make a copy of the data for internal use.

GEOM_DONT_COPY_DATA
Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMadd_vertices_with_data

```
GEOMadd_vertices_with_data(obj, verts, normals, colors, n,
                                                        alloc)
GEOMobj          *obj;
float            verts[n][3];
float            normals[n][3];
int              colors[n];
int              n;
int              alloc;
```

A combination of GEOMadd_vertices, GEOMadd_normals, and GEOMadd_int_colors.

*obj — An existing object of type GEOM_MESH or GEOM_POLYHEDRON.

verts[*n*][3] — An array of vertex coordinates (X, Y, Z).

normals[*n*][3] — An array of normals (X, Y, Z) with each normal corresponding to a vertex.

	GEOM_NULL	Add normals later with GEOMadd_normals.
colors[*n*]	An array of integer-packed RGB colors with each color corresponding to a vertex.	
	GEOM_NULL	Do not add colors. The object will be white.
n	The number of vertices.	
alloc	GEOM_COPY_DATA Make a copy of the data for internal use.	
	GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.	

GEOMauto_transform

```
GEOMauto_transform(obj)
GEOMobj          *obj;
```

Transform an object to appear in a cube with sides of length 2 centered at the origin. Scaling and translation factors are uniform.

*obj	An existing object of any GEOMobj type.

GEOMauto_transform_list

```
GEOMauto_transform_list(objs, n)
GEOMobj          *objs[n];
int              n;
```

Transform a list of pointers to objects to appear in a cube with sides of length 2 centered at the origin. Scaling and translation factors are uniform. The relative sizes of objects in the list are not affected.

*objs[*n*]	A list of pointers to objects of any GEOMobj type.
n	The number of objects in the list.

GEOMauto_transform_non_uniform

```
GEOMauto_transform_non_uniform(obj)
GEOMobj          *obj;
```

Transform an object to appear in a cube with sides of length 2 centered at the origin. Scale factors are non-uniform.

*obj	An existing object of any GEOMobj type.

GEOMauto_transform_non_uniform_list

```
GEOMauto_transform_non_uniform_list(objs, n)
GEOMobj         *objs[n];
int             n;
```

Transform a list of pointers to objects to appear in a cube with sides of length 2 centered at the origin. Scale factors are non-uniform. The relative sizes of objects in the list are not affected.

*objs[*n*]	A list of pointers to objects of any GEOMobj type.
n	The number of objects in the list.

GEOMcreate_label

```
GEOMobj         *
GEOMcreate_label(extent, label_flags)
float           extent[6];
int             label_flags;
```

Create a graphic text label and return a pointer to it of type GEOM_LABEL. To add text strings to the label object, use GEOMadd_label.

extent[6]	GEOM_NULL	The proper extent is automatically calculated.
	To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax.	
label_flags	The value returned by GEOMcreate_label_flags.	

GEOMcreate_label_flags

```
int
GEOMcreate_label_flags(font_number, title, background, drop,
                                                align, stroke)
int             font_number, title, background, drop, align,
                stroke;
```

Create and return an integer value that represents custom label characteristics. The flags are used by GEOMcreate_label, and GEOMadd_label.

font_number

The font used for the label's text string:

0	Plain Roman
1	Duplex Roman
2	Complex Roman

	3	Simplex Roman
	4	Helvetica
	5	Complex Script
	6	Simplex Script
	7	Mathematics
title	0	The label is not a title. Transform the label before it is drawn.
	1	Have the label be a title. It will be drawn in an absolute position where (-1,-1) is the lower left corner, and (1,1) is the upper right corner.
background	0	Only foreground text is drawn.
	1	Foreground text and the enclosing background rectangle are drawn.
drop	0	Do not add a drop shadow highlight.
	1	Add a one-pixel drop shadow to highlight the text.
align	GEOM_LABEL_LEFT	Place the label reference point at the lower left corner of the label.
	GEOM_LABEL_CENTER	Place the label reference point at the bottom center of the label.
	GEOM_LABEL_RIGHT	Place the label reference point at the lower right corner of the label.
stroke	0	This parameter is not currently implemented.

GEOMcreate_mesh

```
GEOMobj          *
GEOMcreate_mesh(extent, verts, m, n, alloc)
float            extent[6];
float            verts[m][n][3];
int              m;
int              n;
int              alloc;
```

Create and return a pointer to a quadrilateral mesh of type GEOM_MESH. The dimensions of the mesh are defined by a 2D array of vertices.

extent[6]	GEOM_NULL	The correct extent is automatically calculated.

To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax.

verts[*m*][*n*][3] An array of vertex coordinates (X, Y, Z).

m — The number of vertices that constitutes the first row of the mesh.

n — The number of rows of vertices.

alloc — GEOM_COPY_DATA
Make a copy of the data for internal use.

GEOM_DONT_COPY_DATA
Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMcreate_mesh_with_data

```
GEOMobj          *
GEOMcreate_mesh_with_data(extent, verts, normals, colors, m,
                                                   n, alloc)
float            extent[6];
float            verts[m][n][3];
float            normals[m][n][3];
int              colors[m][n];
int              m;
int              n;
int              alloc;
```

Create and return a pointer to a quadrilateral mesh of type GEOM_MESH. The dimensions of the mesh are defined by a 2D array of vertices.

extent[6]	GEOM_NULL	The correct extent is automatically calculated.

To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax.

verts[*m*][*n*][3] An array of vertex coordinates (X,Y, Z).

normals[*m*][*n*][3] An array of normals (X,Y, Z) with each normal corresponding to a vertex.

GEOM_NULL	Add normals later with GEOMadd_normals.

colors[*m*][*n*] An array of integer-packed RGB colors with each color corresponding to a vertex.

GEOM_NULL	Add colors later with GEOMadd_int_colors.

m	The number of vertices that constitutes the first row of the mesh.
n	The number of rows of vertices.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMcreate_normal_object

```
GEOMobj          *
GEOMcreate_normal_object(obj, scale)
GEOMobj          *obj;
float            scale;
```

Take an object that contains normals and represent them with disjoint lines. Return a pointer to an object of type GEOM_POLYTRI that contains the disjoint lines.

*obj	An object of type GEOM_MESH, GEOM_POLYHEDRON, or GEOM_POLYTRI.
scale	The length of each normal is scaled by this value before being converted into a disjoint line.

GEOMcreate_obj

```
GEOMobj          *
GEOMcreate_obj(type, extent)
int              type;
float            extent[6];
```

Return a pointer to an empty object. The only data type that must use this call is GEOM_POLYTRI. For the other types, you can use the respective GEOMcreate routine.

type	GEOM_LABEL	Graphic text label.
	GEOM_MESH	Quadrilateral mesh.
	GEOM_POLYHEDRON	Group of polygons.
	GEOM_POLYTRI	Line or polytriangle strip.
	GEOM_SPHERE	Sphere.
extent[6]	GEOM_NULL	The correct extent is automatically calculated.
	To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax.	

GEOMcreate_polyh

```
GEOMobj          *
GEOMcreate_polyh(extent, verts, n, plist, flags, alloc)
float            extent[6];
float            verts[n][3];
int              n;
int              *plist;
int              flags;
int              alloc;
```

Create and return a pointer to a polyhedron of type GEOM_POLYHEDRON. This function combines GEOMcreate_obj, GEOMadd_vertices, and GEOMadd_polygons.

extent[6]	GEOM_NULL	The correct extent is automatically calculated.
	To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax.	
verts[*n*][3]	An array of vertex coordinates (X,Y, Z).	
n	The number of vertices.	
*plist	A list of vertex indices for multiple polygons. For each polygon, specify the number of indices, then the indices themselves. Terminate the list with the value 0.	
flags	0 This parameter is not currently implemented.	
alloc	GEOM_COPY_DATA Make a copy of the data for internal use.	
	GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.	

GEOMcreate_polyh_with_data

```
GEOMobj          *
GEOMcreate_polyh_with_data(extent, verts, normals, colors, n,
                                         plist, flags, alloc)
float            extent[6];
float            verts[n][3];
float            normals[n][3];
int              colors[n];
int              n;
int              *plist;
int              flags;
int              alloc;
```

Create and return a pointer to a polyhedron of type GEOM_POLYHEDRON. This function combines GEOMcreate_polyh, GEOMadd_int_colors, and GEOMadd_normals.

extent[6]	To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax. GEOM_NULL — The correct extent is automatically calculated.
verts[*n*][3]	An array of vertex coordinates (X,Y, Z).
normals[*n*][3]	An array of normals (X, Y, Z) with each normal corresponding to a vertex.
colors[*n*]	An array of integer-packed RGB colors with each color corresponding to a vertex.
n	The number of vertices.
*plist	A list of vertex indices for multiple polygons. For each polygon, specify the number of indices, then the indices themselves. Terminate the list with the value 0.
flags	0 This parameter is not currently implemented.
alloc	GEOM_COPY_DATA Make a copy of the data for internal use. GEOM_DONT_COPY_DATA Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMcreate_scalar_mesh

```
GEOMobj          *
GEOMcreate_scalar_mesh(xmin, xmax, ymin, ymax,
                        scalar_mesh, colors, m, n, alloc)
float            xmin, xmax;
float            ymin, ymax;
float            scalar_mesh[m][n];
float            colors[m][n][3];
int              m;
int              n;
int              alloc;
```

Create a scalar quadrilateral mesh from an array of scalar values. Each scalar value is interpreted as the Z-component of the corresponding vertex coordinate.

xmin, xmax Interpolate *m* times between the minimum and maximum values producing the X-vertex components for the vertex array.

ymin, ymax Interpolate *n* times between the minimum and maximum values producing the Y-vertex components for the vertex array.

scalar_mesh[*m*][*n*]An array of Z-coordinate vertices that correspond to the interpolated X- and Y-vertex values.

colors[*m*][*n*][3]An array of RGB colors with each color corresponding to a vertex.

GEOM_NULL Add colors later with GEOMadd_float_colors.

m The number of vertices that constitutes the first row of the mesh.

n The number of rows of vertices.

alloc GEOM_COPY_DATA
Make a copy of the data for internal use.

GEOM_DONT_COPY_DATA
Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMcreate_sphere

```
GEOMobj          *
GEOMcreate_sphere(extent, verts, radii, normals, colors, n,
alloc)
float            extent[6];
float            verts[n][3];
float            radii[n][3];
float            normals[n][3];
int              colors[n];
int              n;
int              alloc;
```

extent[6] — GEOM_NULL: The correct extent is automatically calculated.

To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax.

verts[*n*][3] — An array of vertex coordinates (X,Y, Z).

radii[*n*][3] — An array of radii, one for each sphere.

GEOM_NULL: Add radii later with GEOMadd_radii.

normals[*n*][3] — An array of normals (X, Y, Z) with each normal corresponding to a vertex.

GEOM_NULL: Add normals later with GEOMadd_normals.

colors[*n*] — An array of integer-packed RGB colors with each color corresponding to a vertex.

GEOM_NULL: Add the colors later with GEOMadd_int_colors.

n — The number of spheres.

alloc — GEOM_COPY_DATA
Make a copy of the data for internal use.

GEOM_DONT_COPY_DATA
Keep a pointer reference to the data that is deallocated when the object is destroyed.

GEOMcvt_mesh_to_polytri

```
GEOMcvt_mesh_to_polytri(obj, flags)
GEOMobj         *obj;
int             flags;
```

Convert a quadrilateral mesh to a polytriangle strip for more efficient rendering. The object type is changed from GEOM_MESH to GEOM_POLYTRI.

*obj	Passed in as a pointer to type GEOM_MESH and is converted to a pointer to type GEOM_POLYTRI.	
flags	GEOM_SURFACE	The resultant object is a polytriangle strip.
	GEOM_WIREFRAME	The resultant object is is a representation of the mesh drawn with polylines.
	Logical OR of above	The resultant object contains both polytriangle and wireframe descriptions.

GEOMcvt_polyh_to_polytri

```
GEOMcvt_polyh_to_polytri(obj, flags)
GEOMobj         *obj;
int             flags;
```

Convert a polyhedron to a polytriangle strip for more efficient rendering. The object type is changed from GEOM_POLYHEDRON to GEOM_POLYTRI.

*obj	Passed in as a pointer to type GEOM_POLYHEDRON and is converted to a pointer to type GEOM_POLYTRI.	
flags	GEOM_SURFACE	The resultant object is a polytriangle strip.
	GEOM_WIREFRAME	The resultant object is a representation of the polyhedron drawn with polylines.
	Logical OR of above	The resultant object contains both polytriangle and wireframe descriptions.

GEOMdestroy_edit_list

```
GEOMdestroy_edit_list(list)
GEOMedit_list     list;
```

Deallocate the memory associated with an edit list. Decrement object reference counts for all objects in the edit list. If an object's reference count is 0, then its memory will also be deallocated.

If an existing edit list is referenced in GEOMinit_edit_list, then this routine is called implicitly.

list	An existing edit list.

GEOMdestroy_obj

```
GEOMdestroy_obj(obj)
GEOMobj          *obj;
```

Decrement the object reference count. When all of the edit lists this object was associated with are destroyed, the object's memory will also be deallocated. Any memory privately allocated and passed in with the GEOM_DONT_COPY_DATA flag will be deallocated.

*obj	A pointer to an existing object.

GEOMedit_color

```
GEOMedit_color(list, name, color)
GEOMedit_list     list;
char              *name;
float             color[3];
```

Set the edit list color associated with the object. This color is different than the color assigned to the object when it was created. This routine only affects the object color if the object was created with GEOM_NULL as the color parameter. Otherwise, the color associated with the object at its creation remains active.

For the color adjustments of the property editor in the Geometry Viewer to work properly, create your objects with GEOM_NULL as the color, then use GEOMedit_colors. This method also yields much better performance when interactively modifying object colors from a module.

list	An initialized edit list.
*name	The name associated with an object when it was added to an edit list with GEOMedit_geometry.

	"camera *n*"	Set background color for camera *n*. The value of *n* ranges from 1 to the number of views.
	"light *n*"	Set light color for light source *n*. The value of *n* ranges from 1 to 16.
color[3]	An RGB color.	

GEOMedit_concat_matrix

```
GEOMedit_concat_matrix(list, name, matrix)
GEOMedit_list      list;
char               *name;
float              matrix[4][4];
```

Post-concatenate a transform matrix to the specified object name.

list	An initialized edit list.	
*name	The name an object was given in the call to GEOMedit_geometry.	
	"camera *n*"	Post-concatenate the matrix for view *n*. The value of *n* ranges from 1 to the number of views.
	"light *n*"	Post-concatenate the matrix for light source *n*. The value of *n* ranges from 1 to 16.
matrix[4][4]	A 4 by 4 transformation matrix.	

GEOMedit_geometry

```
GEOMedit_geometry(list, name, obj)
GEOMedit_list      list;
char               *name;
GEOMobj            *obj;
```

Enter a reference to an object in an edit list, and assign the object a text string name. An object will not appear in the Geometry Viewer unless it is entered in an edit list and the edit list is sent.

list	An initialized edit list.
*name	An ASCII string used to reference the object in other GEOMedit routines.

If the name has been entered previously in the same edit list, then a ".*n* "will be added automatically to the end of the name and it will be a unique reference. The value *n* refers to the order in which the names were entered. This feature keeps different modules from modifying each other's geometry. To suppress this, place "%" (the percent character) at the beginning of the string.

*obj — A pointer to an existing object. It is possible for objects to be in intermediate states. Ensure that the object has all of its fields (vertices, colors, normals) filled in before sending the edit list.

GEOMedit_light

```
GEOMedit_light(list, name, type, status)
GEOMedit_list    list;
char             *name;
char             *type;
int              status;
```

Change light source representation and visibility.

list	An initialized edit list.	
*name	"light *n*"	A light source name with *n* ranging from 1 to 16.
*type	"directional"	Set light to directional.
	"bi-directional"	Set light to bi-directional.
status	0	Light source is OFF.
	1	Light source is ON.

GEOMedit_parent

```
GEOMedit_parent(list, name, parent)
GEOMedit_list    list;
char             *name;
char             *parent;
```

Build a hierarchy of objects. Properties, transformations, and visibilities are inherited from the parent and can be set afterwards. An object can have no more than one parent. Neither argument has to be an object.

list — An initialized edit list.

*name — The name an object was given in a call to GEOMedit_geometry or a parent from a previous GEOMedit_parent call.

*parent	The name an object was given in a call to GEOMedit_geometry or an arbitrary unique name.
	"NULL" — Reference the top-level object.

GEOMedit_picked

```
GEOMedit_picked(list, name)
GEOMedit_list    list;
char             *name;
```

Set the specified object name to be picked in the Geometry Viewer pick window. This routine is unique to ConvexAVS.

list	An initialized edit list.
*name	The name an object was given in a call to GEOMedit_geometry or GEOMedit_parent.

GEOMedit_properties

```
GEOMedit_properties(list, name, ambient, diffuse, specular,
specular_exponent, transparency, specular_color)
GEOMedit_list    list;
char             *name;
float            ambient;
float            diffuse;
float            specular;
float            specular_exponent;
float            transparency;
float            specular_color[3];
```

Change the lighting calculations for material properties of the specified object name. If any value is 1.0 then it is not changed from its current state. An object inherits the properties of its ancestors (parents, grandparents, etc.). If an object has no ancestors, then it will have the default property values.

list	An initialized edit list.
*name	The value an object was given in the call to GEOMedit_geometry.
ambient	Material property value ranging from 0.0 to 1.0 with the default at 0.3. Object becomes lighter as value approaches 1.0.
diffuse	Material property value ranging from 0.0 to 1.0 with the default at 0.7. Object becomes lighter as value approaches 1.0.

specular	Material property value ranging from 0.0 to 1.0 with the default at 0.0. Object becomes lighter as value approaches 1.0.
specular_exponent	The object's gloss. The larger the value the glossier the object. The default is 50.
transparency	Material property value ranging from 0.0 to 1.0 with the default at 0.0. Object becomes less transparent as value approaches 1.0 and opaque at 0.0 and 1.0.
specular_color[3]	RGB color with the default value set to 0.99, 0.99, 0.99.

GEOMedit_render_mode

```
GEOMedit_render_mode(list, name, mode)
GEOMedit_list     list;
char              *name;
char              *mode;
```

Set the render mode for a specified object name.

list	An initialized edit list.
*name	The name an object was given in the call to GEOMedit_geometry. The object must have been of type GEOM_MESH, GEOM_POLYHEDRON, GEOM_POLYTRI, or GEOM_SPHERE.
*mode	A text string with one of the following values:

"gouraud"	Use gouraud shading algorithm. This is the default value for the top-level object.
"lines"	Represent the connected vertices of the object with disjoint lines.
"no_light"	Use object vertex and material colors with no lighting.
"inherit"	Inherit properties of the parent object. This is the default value for an object.
"flat"	Use flat shading algorithm.

GEOMedit_set_matrix

```
GEOMedit_set_matrix(list, name, matrix)
GEOMedit_list    list;
char             *name;
float            matrix[4][4];
```

Set the transformation matrix for the specified object name to the one specified.

list — An initialized edit list.

*name — The name an object was given in the call to GEOMedit_geometry.

"camera *n*"	Set matrix for view *n*. The value of *n* ranges from 1 to the number of views.
"light *n*"	Set light matrix for light source *n*. The value of *n* ranges from 1 to 16.

matrix[4][4] — A 4 by 4 transformation matrix.

GEOMedit_visibility

```
GEOMedit_visibility(list, name, visibility)
GEOMedit_list    list;
char             *name;
int              visibility;
```

Delete or set the visibility of the specified object name. This is the only way for a program to delete an object once it is in the Geometry Viewer.

list — An initialized edit list.

*name — The name an object was given in the call to GEOMedit_geometry.

visibility

-1	Delete the object.
0	Turn object visibility OFF.
1	Turn object visibility ON.

GEOMflip_normals

```
GEOMflip_normals(obj)
GEOMobj          *obj;
```

Invert the direction of the normals in the specified object.

*obj	A pointer to an existing object of type GEOM_MESH, GEOM_POLYHEDRON, or GEOM_POLYTRI.

GEOMgen_normals

```
GEOMgen_normals(obj, flags)
GEOMobj          *obj;
int              flags;
```

Generate surface normals for objects of type GEOM_MESH or GEOM_POLYHEDRON. The normals generated are guaranteed to be of unit length.

*obj	A pointer to an existing object of type GEOM_MESH or GEOM_POLYHEDRON.	
flags	0	This parameter is not currently implemented.

GEOMinit_edit_list

```
GEOMedit_list
GEOMinit_edit_list(list)
GEOMedit_list    list;
```

Initialize a new edit list.

list	GEOM_NULL	Return a new empty edit list.
	pointer to edit list	Implicitly execute a GEOMdestroy_edit list and return a new empty one.

GEOMnormalize_normals

```
GEOMnormalize_normals(obj)
GEOMobj          *obj;
```

Convert the normals of the specified object to unit length. This is done automatically by GEOMgen_normals.

*obj	A pointer to an existing object.

GEOMread_obj

```
GEOMread_obj(obj, fd, flags)
GEOMobj        *obj;
int            fd;
int            flags;
```

Read a geometry object from a file descriptor. The data is interpreted as being in binary geometry format (.geom suffix).

*obj	A pointer to an existing object.	
fd	An open file descriptor.	
flags	GEOM_NORMAL	Strip off normals.
	GEOM_VCOLORS	Strip off colors.
	Logical OR of above	Strip off normals and colors.
	0	Leave data intact.

GEOMset_computed_extent

```
GEOMset_computed_extent(obj, extent)
GEOMobj        *obj;
float          extent[6];
```

Set the object extent to the one specified. Relative scaling of objects can be performed by setting an object extent larger or smaller than its actual extent.

*obj	A pointer to an existing object.
extent	To provide your own extent, use the format: xmin, xmax, ymin, ymax, zmin, zmax.

GEOMset_extent

```
GEOMset_extent(obj)
GEOMobj        *obj;
```

Calculate and set the extent of the object specified. This routine is necessary only for compatibility. ConvexAVS will correctly calculate and set the extents of all objects it receives.

*obj	A pointer to an existing object.

GEOMset_object_group

```
GEOMset_object_group(obj, name)
GEOMobj          *obj;
char             *name;
```

Maintain object group names for use by the read_subset script language command. This allows selected objects to be grouped so when they are saved in a binary geometry file (.geom suffix), they can be explicitly retrieved. By default, ConvexAVS retrieves all objects in a binary geometry file.

*obj	A pointer to an existing object.
*name	A text string that will be used later for explicit object retrieval.

GEOMset_pickable

```
GEOMset_pickable(obj, pickable)
GEOMobj          *obj;
unsigned long    pickable;
```

Allow an object to be picked when it is written to a binary geometry file (.geom suffix). By default, ConvexAVS combines all objects into one pickable unit when a binary geometry file is written. Object files (.obj suffix) retain all object names, hierarchies, and pickable properties.

*obj	A pointer to an existing object.	
pickable	0	Object is not pickable.
	1	Object is pickable.

GEOMunion_extents

```
GEOMunion_extents(obj1, obj2)
GEOMobj          *obj1, *obj2;
```

Set the extent of obj1 to include the extent of obj2.

*obj1	A pointer to an existing object.
*obj2	A pointer to an existing object.

GEOMwrite_obj

```
GEOMwrite_obj(obj, fd, flags)
GEOMobj        *obj;
int            fd;
int            flags;
```

Write a geometry object to a file descriptor. The data is written in binary geometry format (.geom suffix).

*obj	A pointer to an existing object.	
fd	Specify an open file descriptor.	
flags	GEOM_NORMAL	Strip off normals.
	GEOM_VCOLORS	Strip off colors.
	Logical OR of above	Strip off normals and colors.
	0	Leave data intact.

GEOMwrite_text

```
GEOMwrite_text(obj, fp, flags)
GEOMobj        *obj;
FILE           *fp;
int            flags;
```

Write an ASCII version of the geometry object to the stream file pointer. This routine is useful for debugging your code as well as for transporting geometric data between different architectures.

*obj	A pointer to an existing object.
*fp	Specify an open file pointer.
flags	0 This parameter is not currently implemented.

Programming Information

The following sections give general programming information.

Object Creation Routines

The creation routines for the geom library define the following data formats:

- A list of vertices is a 2D array of floats of X, Y, and Z.
- A list of normals is a 2D array of floats of NX, NY, and NZ.
- A list of float colors is a 2D array of floats of R, G, B (in the range of 0.0 to 1.0).
- A list of integer colors is also defined where R, G, and B are bytes packed into a 32-bit word:
  - R occupies bits 23-16
  - G occupies bits 15-8
  - B occupies bits 7-0

  All colors are stored internally as arrays of floats.
- A list of extents is an array of 6 floats in the following order: xmin, xmax, ymin, ymax, zmin, zmax.

Creating An Object

A geom object is initially created without any data at all. Data is then added to the object incrementally. A sequence would be to create an object of type GEOM_POLYHEDRON, add a polygon list, add vertices, then add normals. Notice that an object can be in an intermediate state where it does not make sense. It can have a polygon list without vertices, for example.

To reduce the number of procedure calls required to create an object, GEOM also provides functions that create and add various pieces of data. When one of these calls has a parameter for an optional piece of data (normals and colors, for example), GEOM_NULL can be used to indicate that you do not want to add this data or that it has already been added. You will have to add this data later before the geometry description is complete.

Flags

Many routines have an alloc flag as a parameter. If this flag is set to GEOM_COPY_DATA, the geom routine allocates its own space and copies the data of the object. If this flag is set to GEOM_DONT_COPY_DATA, the geom routine uses a pointer to the data. In this case, ensure that the pointer passed in is the value returned by `malloc` (3C). This memory will be automatically deallocated by `free` (3C) when the object and its edit list are destroyed.

Extents

Every object has extent information associated with it. Routines that create objects take optional extent information.

If you do not supply an extent during the creation of your object (by passing GEOM_NULL), it will be correctly generated by ConvexAVS. It is calculated by finding the minimum and maximum of X, Y, and Z in your vertex list. For spheres, this includes the radii.

The bounding box in the Geometry Viewer subsystem shows the currently picked object's extent.

Edit List Manipulation

A geometry object describes changes to the geometry of a particular scene that is represented by a module input or output. ConvexAVS allows your module to create geometry objects as outputs. ConvexAVS V1.0 does not support using geometry input in user-defined modules. Geometry output is used as input to a ConvexAVS-supplied render module such as the render manager module.

A geometry data object is called an edit list. This is an arbitrarily long list of changes to be made in the current scene. Each change pertains to a particular object, camera, or light source. Changes are made in the order specified in the edit list. The ConvexAVS data type for an edit list is GEOMedit_list. A C language module declares an argument representing an output port as GEOMedit_list * (note the single asterisk). In FORTRAN, the argument is declared as INTEGER.

Each object, camera, or light is referred to by a name that is an ASCII string. By default, an object name is modified by the port through which it is communicated. This prevents two different modules from modifying each other's objects. For example, two "plate" modules would each try to modify the data for the object

named "plate." Because the name is modified by the port, the first plate module modifies "plate.0," and the second modifies "plate.1." When it is desirable for a module to use the absolute name of an object, it can precede the object name by a percent (%) character (for example, "%plate").

Camera names are ASCII strings of the form "camera *n*," where *n* ranges from 1 to the number of views on the particular scene.

Light names are ASCII strings of the form "light *n*," where *n* ranges from 1 to 16.

ConvexAVS has routines that allow a module to change several properties of an object in an edit list:

- The geometric data defining the object.
- Surface or line color.
- Render mode (Gouraud, flat, wireframe, etc.).
- Parent (for defining object hierarchies).
- Object material properties.
- Object, camera, and light transformation.
- Object visibility, deletion.
- Object color, light source color, and camera background color.
- Light source on/off.

FORTRAN Binding

All of the geom routines also have a FORTRAN calling sequence To call a routine from a FORTRAN program, you must use a slightly different routine name and different data declarations:

Routine Name — Replace the GEOM prefix with geom_.

Data Declarations — The following information shows how to convert C language data declarations into FORTRAN declarations:

Table G-1
Changing Data Declarations

C Declaration	FORTRAN Declaration
int var	INTEGER var
int *var	INTEGER var
unsigned int var	INTEGER var
unsigned int *var	INTEGER var
float var	REAL var
float *var	REAL var
double var	REAL var
double *var	REAL var
GEOMobj *var	INTEGER var
GEOMobj **var	INTEGER var
GEOMedit_list var	INTEGER var
GEOMedit_list *var	INTEGER var

Compiling and Linking

A C language application that uses geometry routines must use the /usr/avs/include/geom.h header file.

A FORTRAN language application that uses geometry routines must use the /usr/avs/include/geom.inc header file.

Any application that uses geometry routines must be linked to the following libraries in this order:

- /usr/avs/lib/libgeom.a
- /usr/avs/lib/libutil.a

Module Code Examples

H

C Language Subroutine

```
#include <avs/avs.h>
#include <avs/field.h>

/**********************************************************************/

/*
 * This is a C example to compute the threshold of a 3D scalar field of
 * floating point numbers.
 */

/*
 * The threshold function examines each element of a field to see
 * whether it falls within the range specified by a minimum and maximum
 * parameter (controlled by dials). Elements in the range are passed
 * unchanged to the output field, elements outside the range are
 * set to zero in the output field.
 */

/*
 * The function AVSinit_modules is called from the main() routine supplied
 * by AVS. In it, we call AVSmodule_from_desc with the name of our
 * description routine.
 */

AVSinit_modules()
{

     int threshold();

     AVSmodule_from_desc(threshold);

}

/* The routine "threshold" is the description routine. */

threshold()

{
```

```
    int thresh_compute();     /* declare the compute function (below) */
    int in_port, out_port;    /* temporaries to hold the port numbers */

    /* Set the module name and type */

    AVSset_module_name("ex1-threshold", MODULE_FILTER);

    /* Create an input port for the required field input */

    in_port =
      AVScreate_input_port("Input Field",
                       "field 3D uniform scalar float", REQUIRED);

    /* Create an output port for the result */

      out_port = AVScreate_output_port("Output Field",
                       "field 3D uniform scalar float");

    /* Tell AVS to allocate space for the output data based on the size */
    /* of the input data - note that this only works when the output */
    /* port has the same type as the input port */

    AVSinitialize_output(in_port, out_port);

    /* Add two floating point parameters, both unbounded. Min has */
    /* an initial value of zero, max of 255 */

    AVSadd_float_parameter("thresh_min", 0.0, FLOAT_UNBOUND, FLOAT_UNBOUND);
    AVSadd_float_parameter("thresh_max", 255.0, FLOAT_UNBOUND, FLOAT_UNBOUND);

    /* Tell avs what subroutine to call to do the compute */

    AVSset_compute_proc(thresh_compute);
}

/*
 * thresh_compute is the compute routine. It is called whenever AVS wants to
 * compute new threshold results. The arguments are: the value of the input
 * field, the new output field (doubly undirected), the minimum parameter
 * value and the maximum parameter value. Note the order is always inputs,
 * outputs, parameters. The min comes before the max because in the
 * description routine above, the min is declared before the max.
 */

thresh_compute(input, output, pmin, pmax)
AVSfield_float *input, **output;
float *pmin, *pmax;
{
 register int i, j, k;
 register float min = *pmin;
 register float max = *pmax;
```

```
/*
 * We use a triply nested loop to traverse the field. The macros MAXX,
 * MAXY, and MAXZ determine the maximum extent of the field in each of
 * the three dimensions. We know this will be a 3-dimensional
 * field because of the declaration in the description routine, so
 * we don't need to check. When we want to reference an element of the
 * field we use the I3D macro which picks an element of a 3D field.
 * Note that the first index (i) varies the fastest in memory, so we
 * make that the innermost loop.
 */

for (k = 0; k < MAXZ(input); k++)
  for (j = 0; j < MAXY(input); j++)
    for (i = 0; i < MAXX(input); i++)
      if (I3D(input, i, j, k) > max) {
          I3D(*output, i, j, k) = 0.0;
      } else if (I3D(input, i, j, k) < min) {
          I3D(*output, i, j, k) = 0.0;
      } else {
          I3D(*output, i, j, k) = I3D(input, i, j, k);
      }

/* When we're done, we return 1 to indicate success */

return(1);
}
```

C Language Coroutine

```
#include <stdio.h>
#include <avs/flow.h>
#include <avs/avs_data.h>
#include <avs/field.h>
#include <avs/geom.h>

/*********************************************************************/

/*
 * This is a C example to create a geometry object. In this example,
 * the "simulation" program flow of control is used. Instead of providing
 * a compute module function that is called whenver a parameter or input
 * has changed, this module can determine when it wants to provide new
 * data to the network. Many existing applications will fit into this
 * model much more easily than the "compute function" model.
 *
 * This program is a "string art" program. It draws disjoint lines in a
 * random pattern.
 */

/*
 * The routine "qix" is the description routine. It provides
 * AVS some necessary information such as: name, input and output ports,
 * parameters etc.
 */

qix()
{
    int polygon_compute();   /* declare the compute function (below) */
    int out_port;            /* temporary to hold the port number */
    int parm;                /* temporary to hold the parm number */

    /* Set the module name and type */

    AVSset_module_name("qix", MODULE_DATA);

    /* There are no input ports for this module */

    /* Create an output port for the result */

    out_port = AVScreate_output_port("Output Geometry", "geom");

    /* Add parameter: an enable/disable toggle for the scope */

    parm = AVSadd_parameter("sleep", "boolean", 1, 0, 1);

    /* Add parameter: number of cycles before putting self to sleep */

    parm = AVSadd_parameter("cycles", "integer", 0, 0, 10000);

    /* There is no compute function for this module */
}
```

```
#define MAXV 200
#define PERFRAME 6

typedef float FLOAT3[3];

main(argc,argv)
int   argc;
char  *argv[];
{
 int qix();
 int count = MAXV;
 FLOAT3 verts[2], move0, move1, colors[2], movec0, movec1;
 int sleep = 1;
 int cycles = 0,cycle_count=0;
 GEOMobj *obj = NULL;
 GEOMedit_list output = NULL;
 int i;

/*
 * We initialize this module by passing a pointer to our description
 * function. The description function is defined below. It defines our
 * inputs, outputs and parameters.
 * When AVS loads this module, this program is invoked, the description
 * function is called to identify the module (and get the colors on the
 * palette), then the program exits. When the module is dropped onto
 * the palette, the program is invoked, the description function
 * is called and this function will return.
 */

 AVScorout_init(argc,argv,qix);

 while(1) {

/*
 * If our "sleep" parameter is true, we will wait for any parameters or
 * inputs to change. This mode allows you to mimick the behavior of normal
 * modules. Otherwise, you simply get the current value of the inputs.
 * If the sleep parameter is true, we wait until this (our only parameter)
 * changes.
 */

 if (sleep) {
    AVScorout_wait();
    cycle_count = 0;
 }

/*
 * We get the value of the "sleep" parameter here. This might cause us to
 * do one extra iteration of the loop but since this is a string art demo
 * it won't matter. Note that the parameters to AVScorout_input vary
 * based on the number of inputs and parameters that you have.
 */

 AVScorout_input(&sleep,&cycles);
```

```
/*
 * For each frame, we compute "PERFRAME" number of lines to add to our
 * object. We re-use the same object adding lines until we get to the
 * threshold, then destroy the object and start over again.
 */

 for (i = 0; i < PERFRAME; i++) {
   if (count >= MAXV) {
     start(verts,colors,move0,move1,movec0,movec1);
     count = 0;
     if (obj) GEOMdestroy_obj(obj);
     obj = GEOMcreate_obj(GEOM_POLYTRI,NULL);
   }
   else next(verts,colors,move0,move1,movec0,movec1);
   GEOMadd_disjoint_line(obj,verts,colors,2,GEOM_COPY_DATA);
   count++;
 }

/*
 * Once we have an object that we want to display, we create a new
 * edit list. An edit list contains a list of the changes that we want to
 * make to the scene on any given frame. We clear out the list (freeing
 * the one from the previous frame), then tell AVS to replace the geometry
 * of an object named "qix" with the new object. Then we take the
 * resulting edit list and output it to AVS. Note that the parameters
 * to AVScorout_output vary based on the number of outputs that you have.
 */

  output = GEOMinit_edit_list(output);
  GEOMedit_geometry(output,"qix",obj);
  AVScorout_output(output);
  if (cycles && (cycle_count++ > cycles))
  AVSmodify_parameter("sleep",AVS_VALUE,1,0,0);
 }
}

#define RA 5.0
#define DD 0.2
#define DC 0.05

start(verts,colors,move0,move1,movec0,movec1)
FLOAT3 *verts;
FLOAT3 *colors;
FLOAT3 move0, move1;
FLOAT3 movec0, movec1;
{
 float ran();

 verts[0][0] = ran(RA); verts[0][1] = ran(RA); verts[0][2] = ran(RA);
 verts[1][0] = ran(RA); verts[1][1] = ran(RA); verts[1][2] = ran(RA);

 move0[0] = ran(DD); move0[1] = ran(DD); move0[2] = ran(DD);
 move1[0] = ran(DD); move1[1] = ran(DD); move1[2] = ran(DD);

 colors[0][0] = ran(1.0); colors[0][1] = ran(1.0); colors[0][2] = ran(1.0);
 colors[1][0] = ran(1.0); colors[1][1] = ran(1.0); colors[1][2] = ran(1.0);
```

```
 movec0[0] = ran(DC); movec0[1] = ran(DC); movec0[2] = ran(DC);
 movec1[0] = ran(DC); movec1[1] = ran(DC); movec1[2] = ran(DC);
}

next(verts,colors,move0,move1,movec0,movec1)
FLOAT3 *verts;
FLOAT3 *colors;
FLOAT3 move0, move1;
FLOAT3 movec0, movec1;
{
 int i;
 for (i = 0; i < 3; i++) {
  verts[0][i] = verts[0][i] + move0[i];
  verts[1][i] = verts[1][i] + move1[i];
  colors[0][i] = colors[0][i] + movec0[i];
  colors[1][i] = colors[1][i] + movec1[i];
  if (verts[0][i] > RA && move0[i] > 0.0) {
     verts[0][i] = RA; move0[i] = -move0[i];
  }
  if (verts[0][i] < -RA && move0[i] < 0.0) {
     verts[0][i] = -RA; move0[i] = -move0[i];
  }
  if (verts[1][i] > RA && move1[i] > 0.0) {
     verts[1][i] = RA; move1[i] = -move1[i];
  }
  if (verts[1][i] < -RA && move1[i] < 0.0) {
     verts[1][i] = -RA; move1[i] = -move1[i];
  }

  if (colors[0][i] < 0.0 && movec0[0] < 0.0) {
     colors[0][i] = 0.0; movec0[i] = -movec0[i];

  }
  if (colors[0][i] > 1.0 && movec0[0] > 0.0) {
     colors[0][i] = 1.0; movec0[i] = -movec0[i];
  }
  if (colors[1][i] < 0.0 && movec1[1] < 0.0) {
     colors[1][i] = 0.0; movec1[i] = -movec1[i];
  }
  if (colors[1][i] > 1.0 && movec1[0] > 0.0) {
     colors[1][i] = 1.0; movec1[i] = -movec1[i];
  }
 }
}

float
ran(n)
float n;
{
  double drand48();
  return(n * drand48());
}
```

FORTRAN Subroutine

```
C
C This is a Fortran example to compute the threshold of a 3D
C scalar field of floating point numbers.
C
C The threshold function examines each element of a field to see
C whether it falls within the range specified by a minimum and maximum
C parameter (controlled by dials). Elements in the range are passed
C unchanged to the output field, elements outside the range are set to
C zero in the output field.
C

      subroutine AVSinit_modules
#include "avs.inc"
      external thresh_compute
      integer input, output, i

      call AVSset_module_name('threshold f','filter')

C Note!! AVSset_module_flags should be called by fortran modules using
C fields!

      call AVSset_module_flags(single_arg_data)
      input = AVScreate_input_port('field input',
 $         'field 3D uniform scalar float',REQUIRED)
      output = AVScreate_output_port('field output',
 $         'field 3D uniform scalar float')
      call AVSinitialize_output(input,output)
      call AVSadd_parameter('thresh_min','real',0.0,
 $         FLOAT_UNBOUND,FLOAT_UNBOUND)
      call AVSadd_parameter('thresh_max','real',255.0,
 $         FLOAT_UNBOUND,FLOAT_UNBOUND)
      call AVSset_compute_proc(thresh_compute)

      return
      end
```

```
C
C This version of thresh_compute accesses the field data by means of
C the AVSfield_data_ptr call. Pointers are returned as integers (ip
C and op in this case). They are then dereferenced with the %val
C operator, and passed to a subroutine which declares them as arrays
C of the proper dimension. A disadvantage of this approach is that
C it can be non-portable, since all fortran's do not support %val.
C
      integer function thresh_compute(in,out,min,max)
#include "avs.inc"
      integer in, out
      real min, max
      integer ip, op
      integer dims(3)

      call AVSfield_get_dimensions(in,dims)
      out = AVSfield_alloc(in,dims)
      ip = AVSfield_data_ptr(in)
      op = AVSfield_data_ptr(out)

C call another routine which can get to the data easier

      call doit(%val(ip),%val(op),min,max,dims(1),dims(2),dims(3))
      thresh_compute = 1
      return
      end

      subroutine doit(in,out,min,max,imax,jmax,kmax)
      real in(imax,jmax,kmax)
      real out(imax,jmax,kmax)
      real min, max

      do i=1, imax
       do j = 1, jmax
        do k = 1, kmax
         if (in(i,j,k).gt.max) then
          out(i,j,k) = 0.0
         elseif (in(i,j,k).lt.min) then
          out(i,j,k) = 0.0
         else
          out(i,j,k) = in(i,j,k)
         endif
        enddo
       enddo
      enddo
      return
      end
```

```
C
C This version of thresh_compute accesses the field data by means of
C the AVSfield_data_offset call. AVSfield_data_offset returns
C an offset value from a dummy array. This dummy array can be
C passed to another subroutine which declares the data arrays with
C the proper dimensions. Although somewhat awkward, this method
C is more portable than using AVSfield_data_ptr.
C

      integer function thresh_compute(in,out,min,max)
#include "avs.inc"
      integer in, out
      real min, max
      integer dims(3)
      integer inoffset, outoffset
      real indata(1), outdata(1)

      call AVSfield_get_dimensions(in,dims)
      out = AVSfield_alloc(in,dims)
      call AVSfield_data_offset(in,indata(1),inoffset)
      call AVSfield_data_offset(out,outdata(1),outoffset)
      ip = AVSfield_data_ptr(in)
      op = AVSfield_data_ptr(out)

C call another routine which can get to the data easier

      call doit(indata(inoffset+1),outdata(outoffset+1),
 $           min,max,dims(1),dims(2),dims(3))
      thresh_compute = 1
      return
```

Creating Online Help for Your Modules

Format and Naming Conventions

Each help screen in ConvexAVS is implemented as an ASCII text file, with a .txt filename suffix. The file is displayed in a Help Browser window using a monospace font (all characters have the same width). However you create the help file using a text editor is exactly how it appears in the browser.

If you use **tab** characters in help text files, be sure to set the tab stops in your text editor to every eight columns. It is safer to use **space** characters to align columnar material.

The name of the help file must match the name of the associated module or network. Replace **space** characters in the module or network name with underscores. For example:

Module/Network Name	Help Filename
render manager	render_manager.txt
field to int	field_to_int.txt

You can include comment lines in your help files. Any line that begins with a period (.) is suppressed when the file is displayed.

Integrating Your Help Files

There are two aspects to having your help files become part of the online help system: integrating the files into the ConvexAVS help facility, then integrating them into the standard man command facility.

ConvexAVS Help

By default, ConvexAVS searches for help files in the directories under /usr/avs/runtime/help. It is not advisable to store your help files in this location. Store your help files in /usr/local/avs/helpfiles, for example. When you invoke ConvexAVS, use the environment variable AVS_HELP_PATH to point to your help data. For example, in the C shell you would enter:

```
% setenv AVS_HELP_PATH /usr/local/avs/helpfiles
```

You can include more than one directory in the search path by separating the entries with colon (:) characters. For example:

```
% setenv AVS_HELP_PATH /usr/bill/avs:/usr/tom/avs
```

You might want to set the environment variable in your shell start-up file (.cshrc for the C shell, .profile for the Bourne shell).

The Network Editor uses the AVS_HELP_PATH variable in the following ways:

- When you click the **Help** button in the Network Control Panel window (along the left edge of the screen), the name of the current network is converted to a filename by replacing **space** characters with underscores and appending a .txt suffix. The help facility searches for that filename in the entire directory hierarchy under the first entry in AVS_HELP_PATH. If such a file exists, it is displayed in a Help Browser window. If not, the next entry in AVS_HELP_PATH is used.

  If no help file is found among all the AVS_HELP_PATH entries, a final search is made in /usr/avs/runtime/help, the default help location. If this fails, an error message window pops up.

- The module icon for your modules includes the same small square as the ConvexAVS-supplied icons. You can click this square with the middle or right mouse button to bring up the Module Editor window. When you click the **Show Module Documentation** button, the help facility converts the module name to a filename and searches for the file as described in the preceding paragraph.

The **Help** button in the Network Construction window does not use AVS_HELP_PATH. It always searches in the directories under /usr/avs/runtime/help for help files; likewise for the Geometry Viewer subsystem.

The Image and Volume Viewer subsystems do use the first entry in AVS_HELP_PATH as the initial help directory.

Man Command

The `man(1)` command can be used to view the help files for the ConvexAVS modules. These files appear to the `man` command to be in directory /usr/man/manavs.

Here is a suggested procedure for making man pages for your own modules visible to the `man` command:

1. Create the man pages as nroff/troff source in a nonsystem location and use the .avs filename suffix. For example:

   /usr/local/avs/man/manavs/scatter_dots.avs

2. Add the local man page area to your MANPATH environment variable. To your ~/.cshrc add:
   ```
   setenv MANPATH /usr/local/avs/man:/usr/man
   ```
   Or to your ~/.profile add:
   ```
   MANPATH=/usr/local/avs/man:/usr/man
   export MANPATH
   ```
3. Rebuild the whatis database by executing makewhatis:
   ```
   % /usr/lib/makewhatis -M /usr/local/avs/man
   ```
4. Enter the following command to view your help file:
   ```
   % man -S avs scatter_dots
   ```

Index

Symbols

.byu 19
.fld 22, 361
.geom 19, 73, 75, 85, 342
.obj 75, 85, 342, 349
.pdb 19
.ppoly 19
.prop 73, 75
.scene 74, 75, 95, 349
.topic 34
.ts 19
.wfront 19

Numerics

3D mesh 49

A

About AVS 9
accessing an array 161
Action menu 101
action, restrictions 102
adding new frames 101
aggregate data types
  colormap 145
  field 145
  geometry 145
  pixel map 145
alloc flag 455
allocating fields 158
AM light 87, 91
Ambient light 87
animate_to 344
Append Frame 102
application building 10
arbitrary slice 61
ASCII Header for AVS Field File 26
ASCII text format 26, 365
ASCII-format file 73
Attaching a Label 99
avs command 35
AVS_HELP_PATH 34
AVSadd_float_parameter 380
AVSadd_parameter 380
AVSadd_parameter_prop 383
AVSautofree_output 386
AVSbuild_2d_field 387
AVSbuild_3d_field 387
AVSbuild_field 388
AVSchoice_number 389
AVSconnect_widget 390
AVScorout_exec 392
AVScorout_init 392
AVScorout_input 392
AVScorout_output 393
AVScorout_wait 393
AVScreate_input_port 394
AVScreate_output_port 395
AVSdata_alloc 395
AVSdebug 396
AVSerror 397
AVSfatal 397
AVSfield_alloc 398
AVSfield_copy_points 399
AVSfield_data_offset 399
AVSfield_data_ptr 400
AVSfield_free 400
AVSfield_get_dimensions 401
AVSfield_get_int 401

AVSfield_make_template 402
AVSfield_points_offset 402
AVSfield_points_ptr 403
AVSinformation 403
AVSinit_from_module_list 404
AVSinit_modules 404
AVSinitialize_output 405
AVSinput_changed 405
AVSmark_output_unchanged 406
AVSmessage 406
AVSmodify_float_parameter 409
AVSmodify_parameter 410
AVSmodule_from_desc 411
AVSparameter_changed 412
AVSset_compute_proc 412
AVSset_destroy_proc 412
AVSset_init_proc 413
AVSset_module_flags 413
AVSset_module_name 413
AVS-view_number 76
AVSwarning 414
axes for scene 96

B

bi-directional, lighting 92
bounce 103
Bounding Box 71
Brookhaven Protein Data Bank 6, 71, 342
bubbleviz module 12
byte 5
byu_to_geom 342

C

cameras 72, 94
choice control 123
clamp 49
Clear Network 131
Close button 70, 106, 130
Close, file browser 126
Closing the Network Editor 106
color coding 11, 13
color coding error severity 180
color-codiing for Field I/O Ports 114
colorizer 49
colormap 7
  hue 163
  opacity 163
  saturation 163
  value 163
colormap arrays 163
colormap control 126, 127
colormap, module port color 112
Colormaps 373
compiling modules 184
computational space 148
computational space, 2D 22, 361
connecting modules 111, 142
connecting ports 172
connections, making 113
Continuous 103
contrast 49
control panel 11
control widgets 13, 17, 52, 121
converting applications into modules 184
Convex internal format 71
ConvexAVS, what is it? 3
COORD_X_3D 416
COORD_Y_3D 416
COORD_Z_3D 416
coordinate space 148
coroutine module 14, 180
coroutines 13
create a label 97
Create Camera 72, 95
Create Scene 94
creating an object 454
creating fields 162
creating online help 469
creation routines 454
crop 46
Ctrl-U 84
current object cycle, function key 80
current object indicator 80
cycle command 352

D

-data 39
data conversion 30
-data directory 35
data directory 31
data formats 72

data input 11
data inputs 10
data module 171
data outputs 10, 13
data preprocessing 46, 62
data preprocessing techniques 62
data types
  aggregate 145, 373
  primitive 145
DataDirectory 39, 357
data-flow diagram 15
dataflow network 371
dataflow, kernel perspective 372
dataflow, user perspective 371
Deactivate button 53
deactivate a technique 65
declaring fields for data types 146, 158
default visual 33
defaults file 355
-defaults, geometry 36
Delete Camera 95
Delete Object 85
deleting modules 111
determining if input changed 182
dial control 121
Dial Editor 122
dials 13
diffuse light reflectance 87
-dir, geometry 36
DirectColor 33
directional light_number 76
directional, lighting 92
Disable Flow Executive (toggle) 132
Disable Module 116
disconnecting modules 115
disjoint line, converting 347
disjoint lines 89
display 49
-display display-name 36
Display Network Editor button 106
display windows 52, 128
dot surface 61
double-precision 5
downsize 46, 375
Duplicate 53, 54, 66

E

Edit layout function 129
edit list 165
Edit Network 55, 67
Edit Property 85
environment variables 34
error severity 180
errors
  debug 180
  error 181
  fatal 181
  information 180
  warning 181
executable flow networks 3
Exit button 106
extents 455

F

F1, function key 80
F1, key 77
F2, function key 80
F2, key 78
F3, function key 79, 80
F5, function key 80
F6, function key 81
F7, function key 81
Field 20
field components 155–157
field file format 360
field file, ASCII header 365
field macros
  COORD_X_3D 416
  COORD_Y_3D 416
  COORD_Z_3D 416
  I1DV 417
  I2D 417
  I2DV 417
  I3D 418
  I3DV 418
  I4D 418
  I4DV 419
  MAXX 419
  MAXY 419
  MAXZ 419
  RECT_X 420

RECT_Y 420
RECT_Z 420
field mapping
irregular 149
rectilinear 149
uniform 149
field mapping examples 151–155
field structures 159
field to float 375
field, module port color 112
file browser 13, 47, 83
file browser control 125
file formats 19
filter facility 341
filter module 171
filter templates 345
filter utilities 342
-filter, geometry 36
Filtering 3
Flash Active Modules (toggle) 133
flow executive 14, 142
flow networks 3
Font Selection menu 99
freeing field structures 162
Freeze Camera 95
Front/Back Clipping 96
functions of a module 174
fundamental data type 145

G

generate colormap 12
GEOM file format 20
GEOM library 341
geom_flip 344
geom_pickable 344
geom_scale 344
geom_split 345
geom_to_normals 344
geom_to_text 344
GEOMadd_disjoint_line 426
GEOMadd_disjoint_polygon 426
GEOMadd_float_colors 427
GEOMadd_int_colors 428
GEOMadd_label 428
GEOMadd_normals 429
GEOMadd_polygon 430
GEOMadd_polygons 430
GEOMadd_polyline 431
GEOMadd_polytriangle 432
GEOMadd_radii 432
GEOMadd_vertices 433
GEOMadd_vertices_with_data 433
GEOMauto_transform 434
GEOMauto_transform_list 434
GEOMauto_transform_non_uniform 434
GEOMauto_transform_non_uniform_list 435
GEOMcreate_label 435
GEOMcreate_label_flags 435
GEOMcreate_mesh 436
GEOMcreate_mesh_with_data 437
GEOMcreate_normal_object 438
GEOMcreate_obj 438
GEOMcreate_polyh 439
GEOMcreate_polyh_with_data 440
GEOMcreate_scalar_mesh 441
GEOMcreate_sphere 442
GEOMcvt_mesh_to_polytri 443
GEOMcvt_polyh_to_polytri 443
GEOMdestroy_edit_list 444
GEOMdestroy_obj 444
GEOMedit_color 444
GEOMedit_concat_matrix 445
GEOMedit_geometry 445
GEOMedit_light 446
GEOMedit_parent 446, 447
GEOMedit_properties 447
GEOMedit_render_mode 448
GEOMedit_set_matrix 449
GEOMedit_visibility 449
geometric descriptions 8
geometric primitives 3
geometry 19
command 165
creation routines 454
database 165
format 19
object 165
Geometry Viewer 9, 55, 67, 69
Geometry Viewer button 107
Geometry Viewer File Types 75
Geometry Viewer, switching 107
geometry, defined 73
geometry, module port color 112

-geometry, X Window geometry 36
GEOMflip_normals 450
GEOMgen_normals 450
GEOMinit_edit_list 450
GEOMnormalize_normals 450
-geometry 36
GEOMread_obj 451
GEOMset_computed_extent 451
GEOMset_extent 451
GEOMset_object_group 452
GEOMset_pickable 452
GEOMunion_extents 452
GEOMwrite_obj 453
GEOMwrite_text 453
Gloss 88
gradient shade 49
group Command 351

H

Help 55, 67, 130
Help button 70
help files, module 469
histogram stretch 49, 51
HSV Color 87

I

I1DV 417
I2D 417
I2DV 417
I3D 418
I3DV 418
I4D 418
I4DV 419
IEEE 754
  double format 5
  single format 5
  hardware support 33
image 20, 128
-image 37
image file format 359
Image Processing 48
Image Processing Techniques 49
image transform 49
Image Viewer 9, 17, 43
Image Viewer, using it 44
Immediate button 122
improving edit list performance 166
Inactive command 355
include files 183
incremental search, modules 117
Inherit Property button 88
input parameters 10, 12
input ports 12, 172
integer 5
internal workings
  coroutines 187
  subroutines 186
interpolate 46
Invert, colormap 127
Irregular 2D Field 24, 364
irregular field 27, 366
isosurface 375
isosurface tiler 61

K

keywords, abbreviating 35

L

Label Attributes 100
Label Height 99
Labeling the Top-Level Object 98
Labels menu 97
lasso 111
Layout Editor 134
left mouse button 83
levels of error severity 180
libgeom programming library 71
libgeom.a 20
libraries 184
light blue ball 53, 54, 66
light Command 355
light on or off 92
lighting commands 355
lights 72, 87
lights, colors 93
linear ramp 127
linking modules 184
Lookup Table Techniques 48

M

manager modules 20
manipulating an edit list 168
mapper module 172
Mapping 3
mapping fields 149
mapping information 150
marching-cubes algorithm 61
Mathematica ThreeScript 19, 71, 342
MAXX 419
MAXY 419
MAXZ 419
memory utilization 371
menu choices 75
Menu Selection, Geometry Viewer 81
Mesh 346
mesh of dots. 61
Mesh, converting 347
mesh_to_geom 345
Metal 88
middle button 83
mirror 46
mixing geometries 165
module differences 178
Module Editor button 116
module errors 180
module execution 143
module functions
  computation 177
  description 174
  destruction 174
  initialization 174
module icon 11
module input ports 172
module libraries 14, 118, 184
module library file formats 358
module output ports 172
Module Palette, changing the partition 118
module ports
  input 172
  output 172
module routines
  AVSadd_float_parameter 380
  AVSadd_parameter 380
  AVSadd_parameter_prop 383
  AVSautofree_output 386
  AVSbuild_2d_field 387
  AVSbuild_3d_field 387
  AVSbuild_field 388
  AVSchoice_number 389
  AVSconnect_widget 390
  AVScorout_exec 392
  AVScorout_init 392
  AVScorout_input 392
  AVScorout_output 393
  AVScorout_wait 393
  AVScreate_input_port 394
  AVScreate_output_port 395
  AVSdata_alloc 395
  AVSdebug 396
  AVSerror 397
  AVSfatal 397
  AVSfield_alloc 398
  AVSfield_copy_points 399
  AVSfield_data_offset 399
  AVSfield_data_ptr 400
  AVSfield_free 400
  AVSfield_get_dimensions 401
  AVSfield_get_int 401
  AVSfield_make_template 402
  AVSfield_points_offset 402
  AVSfield_points_ptr 403
  AVSinformation 403
  AVSinit_from_module_list 404
  AVSinit_modules 404
  AVSinitialize_output 405
  AVSinput_changed 405
  AVSmark_output_unchanged 406
  AVSmessage 406
  AVSmodify_float_parameter 409
  AVSmodify_parameter 410
  AVSmodule_from_desc 411
  AVSparameter_changed 412
  AVSset_compute_proc 412
  AVSset_destroy_proc 412
  AVSset_init_proc 413
  AVSset_module_flags 413
  AVSset_module_name 413
  AVSwarning 414
Module Tools 132
ModuleLibraries 39
Modules 4
modules
  coroutine 180

data 171
filter 171
mapper 172
renderer 172
subroutine 179
-modules directory 37
modules, data input 108
Modules, filter 108
modules, finding them 117
modules, renderer 108
mouse button, left drag 111
mouse buttons, functions 77
mouse buttons, transforming cameras 79
mouse buttons, transforming lights 78
mouse buttons, transforming objects 77
mouse, left button 110
Movie BYU 19, 71, 342
Moving Icons 110
Moving Modules 111

N

naming online help files 469
-netdir 39
-netdir directory 35
-network 37
network 142
Network Construction Window 106
Network Control Panel 17
Network Editor 9, 105
Network Tools 131
network, limits 16
network, typical 15
NetworkDirectory 39, 357
NetworkWindow 39, 358
New Dir, button 84, 125
New File, button 84, 126
Normalize, function key 80-81

O

object commands 350
Object Info 89
object, defined 73
Objects 72, 82
Objects Menu Selections 82
objs_to_geom 342
oneshot control 124
online help files 469
orthogonal slice 60, 63
output ports 172

P

parameter 173
Parameter Editor 116
passing bytes 147
passing integers 147
passing reals 148
passing strings 148
Path 39, 358
-path 35, 39
pdb to geom 19
pdb_to_geom 342, 343
perspective 79, 96
Pick Window button 70
Picking and Moving a Label 98
Picture Size 128
pixel map 169
Pixel Processing 7
pixmap 169
pixmap, module port color 112
Pixmaps 129, 373
pobj_to_geom 342
Points 90
Polygen protein data bank 342
polygon 346
polygon, converting 347
polygon_to_geom 345
polyh_to_geom 345
Polyhedron 346
Polyhedron, converting 346
Polyline, converting 347
polylines 89
Polytriangle 346
ports ,connecting 112
Ports and Parameters 10
Postprocessor Filters 344
PostScript 18
ppoly_to_geom 342
preprocess your data 46
primitive data types
byte 145
integer 145

real 145
string 145
Print Network 131
PseudoColor 33
pulldown menu, window 52

R

radio buttons 13, 123
Ramp, colormap 127
Read button, Edit Property 88
read Command 351
Read Image 47
Read Module Library 132
Read Module(s) 132
Read Network 131, 357
Read Object 82, 342, 349
Read Scene 95, 349
read volume 12
Read, colormap 128
RECT_X 420
RECT_Y 420
RECT_Z 420
rectilinear 2D field 23, 362
rectilinear field 27, 363, 366
-reindex 34, 37
render geometry 69, 71, 107, 375
renderer module 166, 172
rendering 3
Reset, function key 80-81
Restart, 53
Restore Parameters 132
RGB color 87
rotate command 353, 354

S

Save button, Edit Property 88
Save Object 85, 349
Save Parameters 132
Save Scene 95, 349
Scalar Mesh, converting 347
scale Command 353, 354
scatter 162
Scene File, example 356
-scene, geometry 36
scenes 74
Script Language 73, 74, 85, 349
script language commands 350
scroll bar, File Browser 83
Select Module Libary, function 118, 133
send_to 345
separator characters 27, 366
set_color command 352, 356
set_material command 353
set_matrix command 352, 354, 356
set_position command 352, 354, 356
set_render_style command 353
shared memory 373
sharing geometries 165
Show Lights 92
Show Module Documentation 109, 116
Show Network 51, 55, 67
Show Object 89
single-precision 5
slice with gradient 60
slider control 123
sliders 13, 75
Spec control 87
specializing words for fields 158
specular highlights 73, 87
Sphere 346
Sphere, converting 347
sphere_to_geom 345
start-up file 35, 38, 357
startup file format 38
Step Backward 103
Step Forward 103
Store Frames 101
subroutine modules 13, 179
SupportedModules 118

T

templates 345
text_to_geom 345
title bar, window 52
Toggles 13
top, level object 76
Transform Camera, function key 80
transform cameras 79
Transform Light, function key 80
Transform Object, function key 77, 80
transform objects 77

Transform Selection menu 76
transformations 76
translate command 353, 354
transparency 88
transpose 46
triangle strips 89
tristate control 124
ts_to_geom 342, 343
type-in control 121
Typeins 13
types of modules 171, 178

U

UNC 19, 342
Uniform 2D Field 22, 361
uniform field 27, 366
uniform, field type 362
unpacking edit lists 166
-usage 37
-usage, geometry 36
user-written module 14
using an edit list 168

V

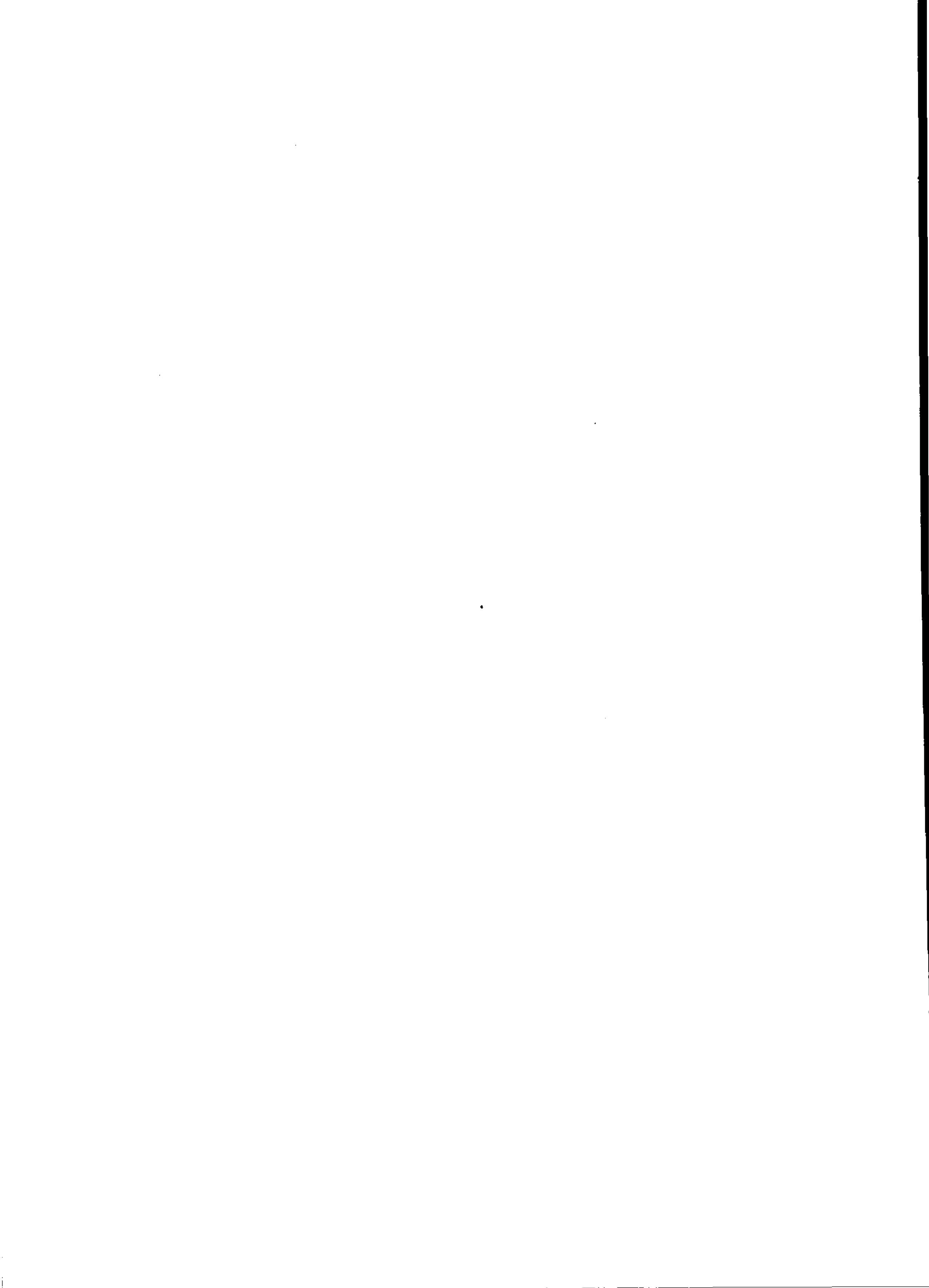
vbuffer 61
Verbose Mode (toggle) 133
-version 38
version number 38
vertex colors 89
vertex normals 89
view command 354
visualization techniques 60
-volume 37
volume bounds 61
volume data 20
volume data file format 29, 368
volume data format 29, 368
Volume Viewer 9, 17
Volume Viewer subsystem 57
Volume Viewer, starting it 59
Volume Viewer, using it 58
volumetric rendering 61

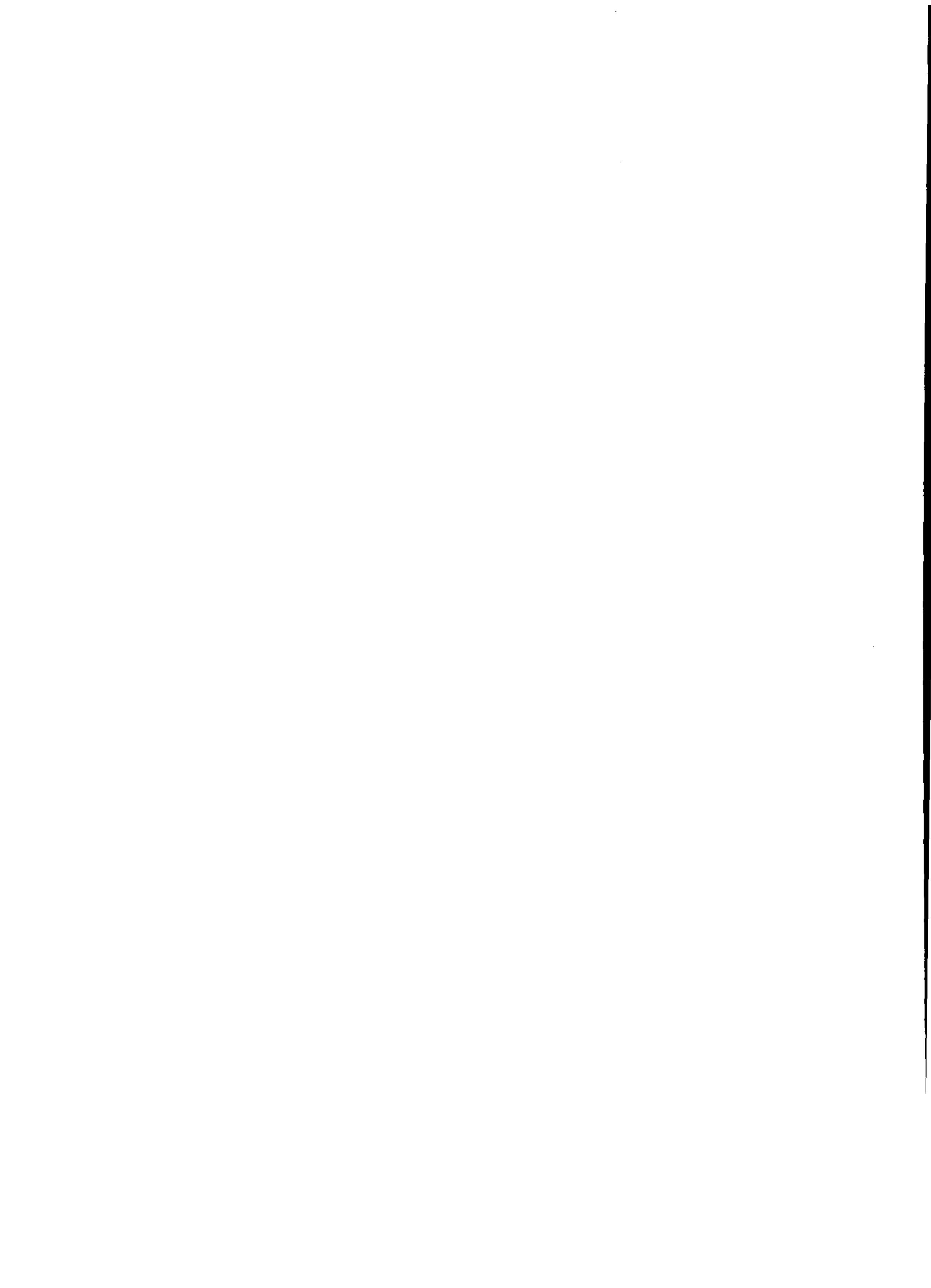
W

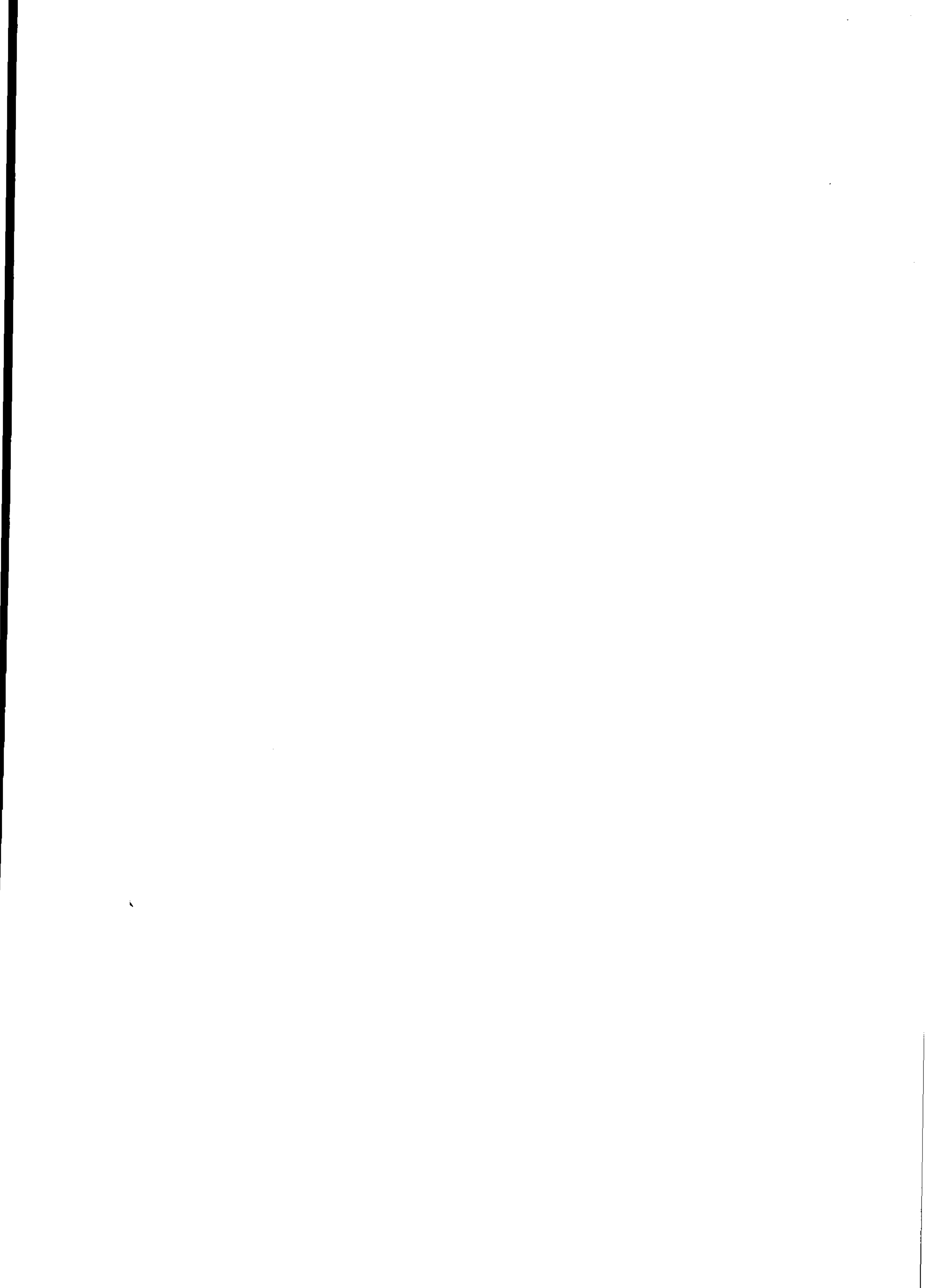
Wavefront 19, 71, 342
wfront_to_geom 342
wildcard 115
window background color 96
window size 128
window. 67
Workspace 111
Write Module Library 133
Write Network 131, 357
write, colormap 128

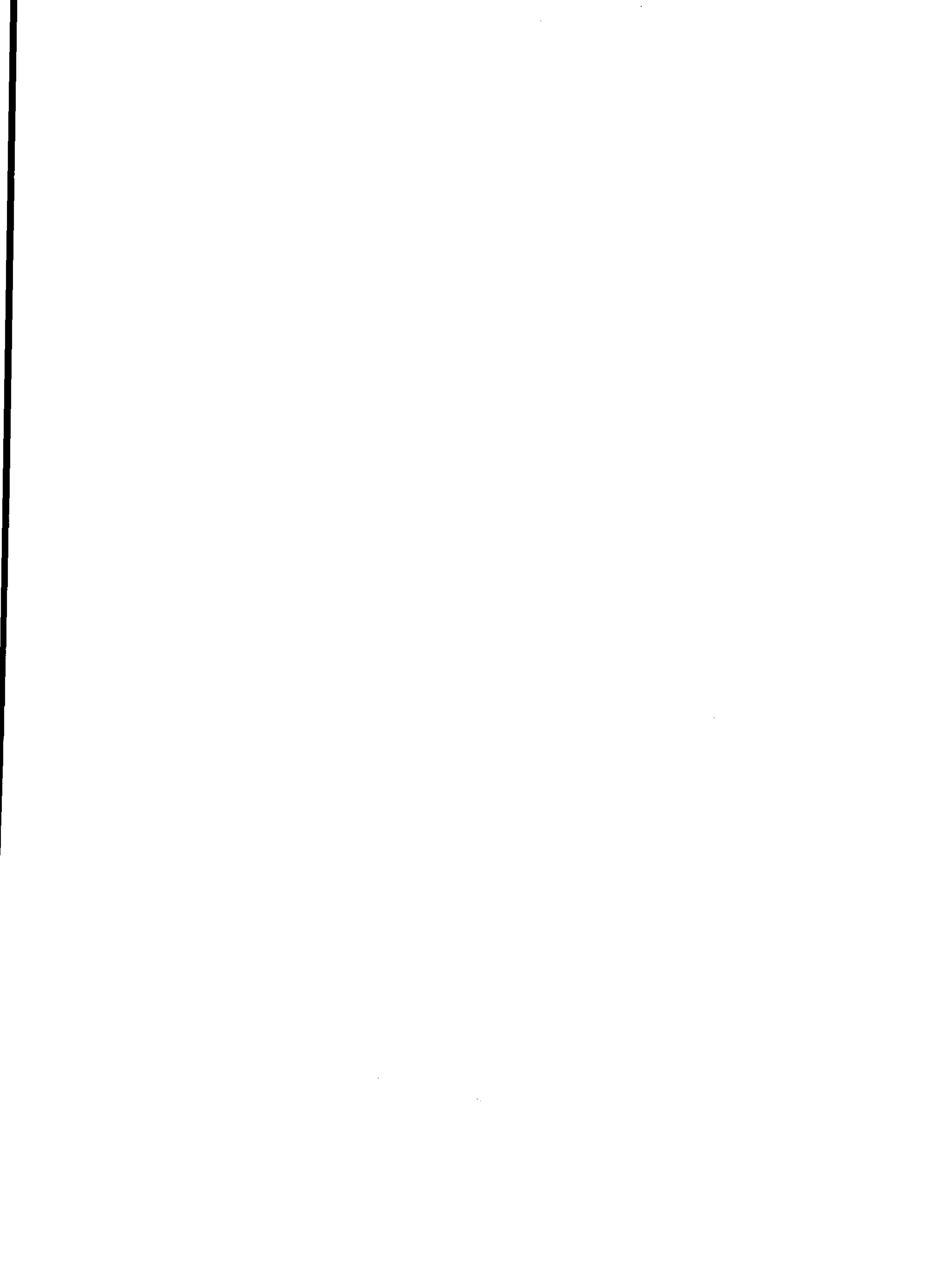
X

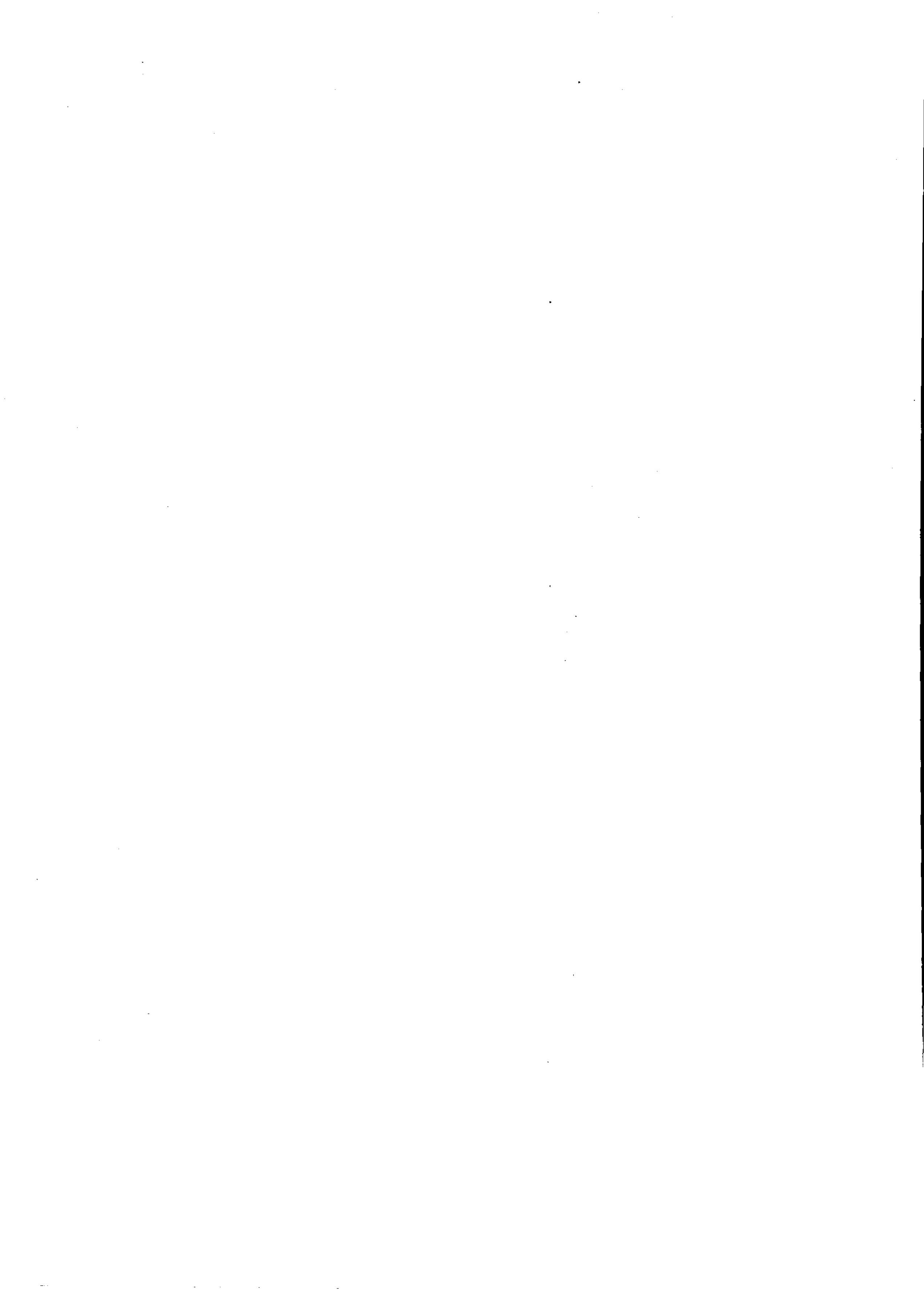
X server version 33
X Window 33
X Window System 65
xlsfonts(1) utility 100

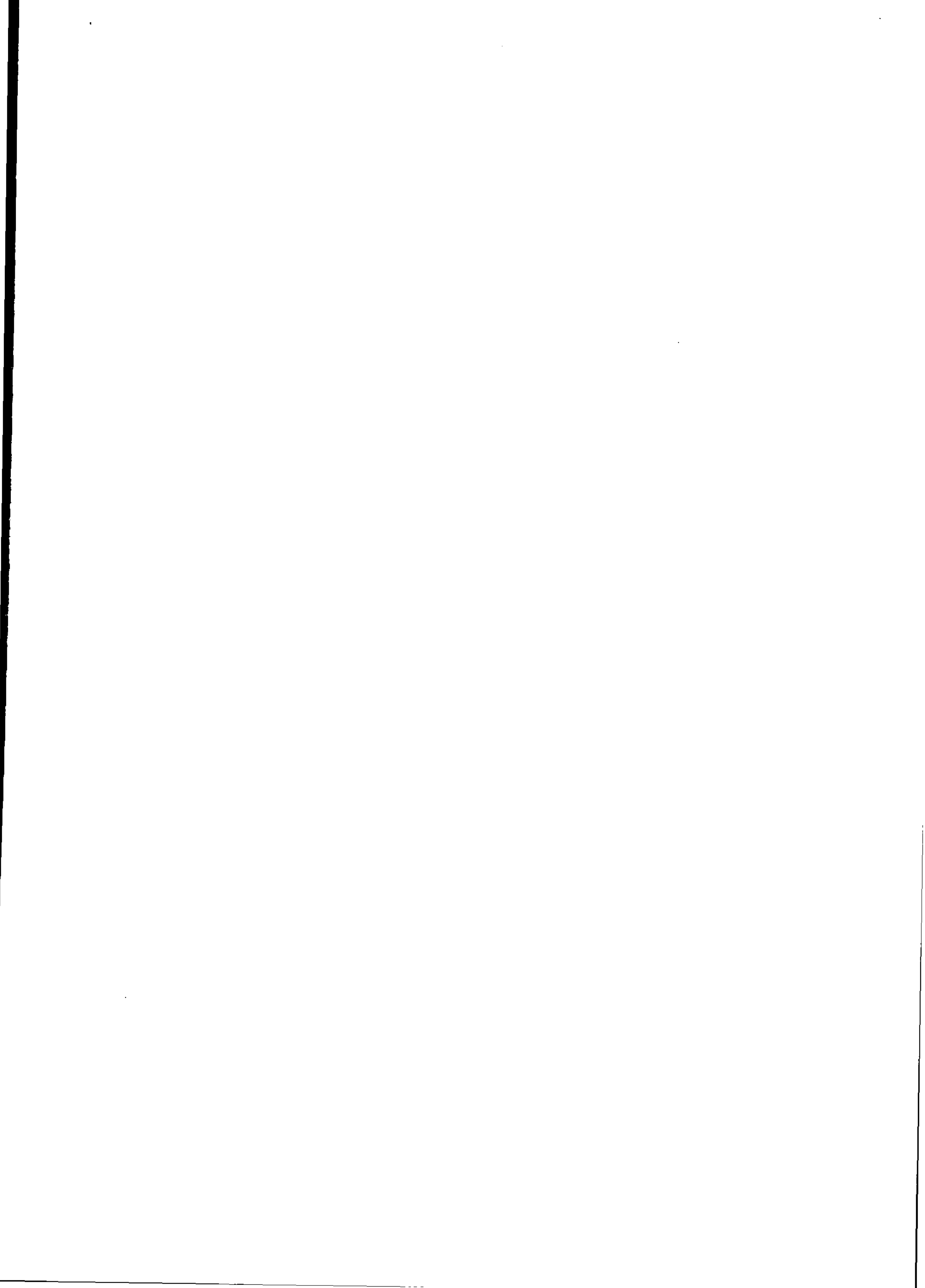




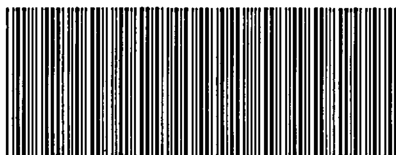








Order Number
DSW-300



Document Number
710-012830-001