Using ConvexAVS to Visualize Data

First Edition

CONVEX COMPUTER CORPORATION

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

Using ConvexAVS to Visualize Data

Order No. DSW-304

First Edition
February 1992

CONVEX Press
Richardson, Texas
United States of America

Using ConvexAVS to Visualize Data

Order No. DSW-304

Printed in the United States of America

Revision information for

Using ConvexAVS to Visualize Data

Edition	Document No.	Description
First	710-012830-004	Released February 1992. Initial release of this book. Contains user and scientific information related to visualizing data with the ConvexAVS V3.0 software.

Contents

1 How to use this book xxxv
- Purpose and audience xxxv
- Organization xxxvi
- Notational conventions xxxviii
  - Command syntax xxxviii
  - General conventions xxxviii
- Associated documents xl
- Ordering documentation xl
- Technical assistance xl
- Acknowledgments xli

2 What Is ConvexAVS? 1
- Introduction 1
- Scientific and engineering data 3
  - Other data formats 4
- Scientific visualization techniques 4
  - Pixel-based visualization 4
    - Colormap lookup 5
  - Further pixel processing 5
    - High-quality pixel-based visualization 6
  - Geometry-based visualization 6
- ConvexAVS subsystems 7
- Command Language Interpreter 8
- ConvexAVS modules 8
  - Modules: ports and parameters 9
    - Data inputs 10
    - Input parameters 11
    - Widget types 12
    - Data outputs 12
    - Subroutine and coroutine modules 12
    - Standard modules and module libraries 13
    - User-written modules 13
- ConvexAVS networks 14
  - Data flow in a ConvexAVS network 15
  - Network control panel 16
    - Image and volume viewer networks 17
- ConvexAVS display windows 17
- Visualization network example 17
  - Mapper modules: Geometries 19

Composite mapping techniques 20
Mapper modules that produce images 22
Combining imaging techniques 24
Producing graphs 26
Techniques combined 28
Curvilinear, vector data 29

3 Starting ConvexAVS 31
Requirements and setup information 32
PseudoColor X servers 32
Color 33
Environment variables 34
Command-line options 35
Command-line options 37
Geometry specific options 41
ConvexAVS .avsrc start-up file resource options 43
Format of the start-up file 43
Adding to the Applications menu 48
Setting Xdefaults for ConvexAVS 49
ConvexAVS Default values (avs.Xdefaults) 49
Xdefaults file (Xdefaults.X) 51
Main menu 53
Subsystem control panels 54
Switching among the subsystems 55
Canceling operations 55
Using online help 56
Finding a help topic 56
Closing the help window 56
Getting documentation for a ConvexAVS module 56
Getting help for a subsystem 57
Using help from the shell command line 57
Opening the Help Demos script browser 57
Scrolling through a topic 59
Resizing the help window 59
Changing fonts 59
Avoiding delays displaying the help window 59
Module editor 60
File browsers and dialog type-in panels 61
Exiting ConvexAVS 63
Running on remote servers 64

4 Network Editor subsystem 65
General information 65
Starting the Network Editor 66
Getting help 67
Help demos 67
Closing the Network Editor 67
Switching subsystems 68
Status widget 68
Using the Network Editor 69
Using the module palette and the workspace 70
Loading module libraries 70
Module types 72
Data input modules 72
Filter modules 72
Mapper modules 72
Data output modules 72
Module input and output ports 73
Module editor and parameter editor windows 75
Show Module Documentation 75
Disable Module 75
Parameter Editor 75
Port color-coding 76
Finding the module you want 77
Scrolling a module list 77
Incremental search through a module category 78
Making the module palette larger 78
Moving icons into the workspace—left button 79
Moving modules within the workspace 80
Deleting modules from the workspace 80
Connecting modules—middle button 81
Disconnecting modules—right button 82
Completing a network 82
Data, parameter, and upstream data ports 83
Data ports 83
Parameter ports 83
Single parameter modules 84
Sending serial parameter changes 85
Upstream data ports 86
Connecting field ports 88
Connecting parameter ports 90
Creating the connection 91
Connecting upstream data ports 92
Controlling the execution of a network 93
Canceling an operation 95
Module restart option 96
accept 96
restart 96

restart same 96
Using control widgets 97
Using type-in controls 97
Using dial controls 98
Dial Editor 98
Using slider controls 101
Using a set of choices (radio buttons) 101
Using toggle controls 102
Using tristate controls 102
Using one-shot controls 102
Using file browser controls 103
New Dir 103
New File 104
Close 104
Other browsers 104
Using the colormap control 105
composite 106
Edit panel 107
Minimum and maximum ranges 107
From/Value and To/Value 108
invert 108
flip 108
cycle 108
ramp 109
smooth 109
Lo Value/High Value 109
Read and Write 110
Organizing a network's display windows 110
Picture size and window size 110
Images 111
Pixmaps 111
Using the window manager 111
Using a display window's pulldown menu 112
Using the Network Editor menu system 114
Help 114
Close 114
Network tools 115
Read Network 115
Write Network 115
Clear Network 116
Print Network 116
Disable Flow Executive (toggle) 117
Save Parameters 117
Restore Parameters 117
Module tools 117
Read Module(s) 117
Read Remote Module(s) 118
Read Module Library 118

Write Module Library 118
Select Module Library 118
Edit Module Library 119
Flash Active Modules 119
Verbose Mode 119
Restart Modules 119
Redesigning the user interface 120
Elements of a layout 121
Working with the Layout Editor 121
Create Page 121
Create Stack 121
Undo 121
Snap to Grid 122
Resizing a text browser 124
Including display windows in a reorganized layout 124
Remote module execution 125
Remote system 126
Setting up a remote module directory 127
Finding remote modules: the hosts file 128
File format for the hosts file 128
Network Editor user interface 130
Constructing a module library 132
Sample procedure 133
Module library file format 135

5 Geometry Viewer subsystem 137
Entering the Geometry Viewer 140
Leaving the Geometry Viewer 141
Scenes: objects, lights, cameras 141
Objects 142
Where objects appear in world space 144
Geometry Viewer control panel 146
Top control bar 147
Transformations and the Transform Selection area 147
Direct transformations 148
Transforming objects 148
Transforming lights 150
Transforming cameras 151
Transforming labels 151
Precise transformations 152
Degree of rotation: arrow keys 152
Absolute and Relative 153
Override 155
Camera example module 155
Bounding Box 155
Current object area 156
Current object indicator 157

Current object browser 157
Additional transformations 158
Using function keys (view window is active) 160
Geometry Viewer submenus 160
Objects 161
Read Object 161
Save Object 163
Delete Object 164
Edit Property 164
Object Information 168
Show Object 169
Rendering methods 170
Lights 171
Cameras 173
Labels 178
Creating labels 179
Labeling the top-level object 179
Picking and moving a label 180
Attaching a label to a vertex 180
Making a label into a title 180
Editing a label 181
Label menu selection 181
Font selection submenu 181
Label Height 182
Label Attributes submenu 182
Action 182
Adding frames 183
Special considerations for adding frames 184
Playing back the frames 184
Deleting frames 185
Command Language Interpreter 186
Data storage formats 187
Geometries 187
Objects 187
Scenes 188
File type summary 188

6 Image Viewer subsystem 189
Introduction 189
Entering the Image Viewer 191
From the shell directly 191
From the main menu 191
From another subsystem 192
In a network 192
Leaving the Image Viewer 193
Basic layout 193
Image Viewer control panel 194

Help 195
Data Viewers 195
Close 195
Transform Selection controls 196
Transform Image 196
Transform View 197
Transform Subimage 197
Bounding box 197
Naming images 198
Retitling and picking images 199
Current Image window 200
Reset/Normalize 200
Menu selection controls 200
Submenu controls 201
Transforming viewports 202
Switching among viewports 203
Resizing viewports 204
Browsers 204
Images submenu 205
Read Image 206
Write Image 207
Duplicate Image 207
Delete Image 207
Show Image/Hide Image 207
Zoom In/Zoom Out 208
Image stacking order—Using Raise and Lower 209
Raise to Front and Lower to Back 209
Views submenu 210
Create Scene 210
Create View 211
Delete View and deleting scenes 211
Save Scene 211
Read Scene 212
Scale X, Y, X and Y 212
Viewport background color 213
Image Processing submenu 214
Selecting processing techniques 215
Applying techniques on whole images 217
Applying techniques on subimages 218
Image processing focus controls 219
Performing multiple techniques on one image 220
Restore Current Image 221
Window management 221
Using Select Processing Technique 222
Special considerations 223
Defining image processing techniques 224
Labels submenu 226
Creating labels 227

Editing labels 227
Selecting and moving labels 227
Creating title labels 228
Label menu selection 228
Font selection submenu 229
Label Attributes submenu 229
Action submenu: Flipbook animation 230
Store Frames 232
Append Frame 232
Total Frames 233
Current Frame 233
Step Forward/Step Backward 233
Continuous 233
Bounce 233
Replay Speed 233
Delete Current Frame 234
Save Cycle 234
Read Cycle 234
A network for geometries 235
Image Viewer and the CLI 235

7 Graph Viewer subsystem 237
Entering and leaving the Graph Viewer 238
Basic Interface 240
Using the Graph Viewer 241
Multiple plot windows—the current window 241
Read Data 243
File type submenu 244
Read ASCII file 244
Read AVS field file 244
Graphing fields 245
Read AVS plot file 246
Read X image file 246
Plot control submenu 247
Replace Current Plot 247
Add to Current Plot 247
Create New Plot 247
Delete Plot Window 248
Data formats submenu 248
Plotting as Y Data 248
Plotting as XY data 250
Plotting as contour data 252
Color Selection 252
Contour level selection 253
Using column data for color control 254
Plot styles submenu 255
Line Plots 255

Area Plots 255
Scatter Plots 255
Bar Plots 255
Write Data 256
File Type submenu 256
Axis Display 257
Border Display 258
Axis Selection 258
Axis Scale 258
Axis Range 258
Axis Tic Marks 259
Number of Tics 259
Decimal Precision 259
Titles and Labels 260
Label menu selection 261
Font Selection 261
Label Attributes 261
Color Controls 261
Select plot submenu 262
Display Crosshair 263
Plot attributes 264
Simple Line 264
Area 264
Scatter 264
Bar 265
Color Control 265
Delete Plot data set 265
Cursor Position 265
Selected Point 266

8 Data types and import strategies 267
Primitive and aggregate data 267
Byte 268
Integer 268
Real (or float) 268
String 269
Declarations and references 269
Importing data choices 273
Importing fields 274
Field components 274
read field 275
Hints on usage 275
Limitations 277
read PLOT3D 277
read image 279
read image data file format 279
read volume 280

read volume data file format 280
Examples 282
Importing geometries 283
render geometry 283
read geom 283
pdb to geom 284
Geometry filters 284
Automatic data filtering 286
Shell-level usage of geometry filters 286
Postprocessor filters 287
Writing a new filter utility 289
Converting a polyhedron 289
Converting a polygon 290
Converting a scalar mesh 290
Converting a mesh 290
Converting a sphere 290
Converting a disjoint line 290
Converting a polyline 290
Importing unstructured cell data 291
read ucd 291
Examples 291
read_ucd.c 291
gen_ucd.f 291
Importing colormaps 292

9 Fields 293
Definition of fields 293
Points and data 293
Field classes 294
Irregular fields 295
Rectilinear fields 295
Uniform fields 296
Field mapping examples 297
Uniform field 1 297
Uniform field 2 297
Rectilinear field 298
Rectilinear or irregular field 298
Irregular field 1 300
Irregular field 2 301
Field components 302
Declaring fields 305
Manipulating fields from C 307
Manipulating fields from FORTRAN 309
Creating fields 310
Scatter data 311
Image data 311
Volume data 312

C language field macros 312
Grouped by function 312
Defined alphabetically 313
COORD_X_3D 313
COORD_Y_3D 313
COORD_Z_3D 313
I1DV 314
I2D 314
I2DV 314
I3D 315
I3DV 315
I4D 315
I4DV 316
MAXX, MAXY, MAXZ 316
RECT_X, RECT_Y, RECT_Z 316

10 Colormaps and pixel maps 317
Colormaps 317
Pixel maps 318

11 Geometries 319
What is an edit list? 319
Why use edit lists? 319
How an edit list is used 320
Design factors 320
Manipulating edit lists 322
Supported objects 323
Label objects 324
Mesh objects 324
Polyhedron objects 324
Polytriangle objects 325
Sphere objects 326
Creating objects 326
Extents 327
Flags 327
User-supplied primitive data 328
User-supplied vertex data 328
FORTRAN binding 329
Maintaining transformation matrices 330
Object transformations 330
Light transformations 331
Camera transformations 331

12 Geometry routines 333
Grouped by topic 334
Object creation routines 334
Generic—GEOMobj 334
Lines and polytriangles—GEOM_POLYTRI 334
Quadrilateral meshes—GEOM_MESH 334
Polygons and polyhedrons—GEOM_POLYHEDRON 334
Spheres—GEOM_SPHERE 334
Text labels—GEOM_LABEL 335
Object modification routines 335
Color—all GEOMobj except GEOM_POLYTRI 335
Normals—GEOM_MESH and GEOM_POLYHEDRON 335
Vertices—GEOM_MESH and GEOM_POLYHEDRON 335
Type conversion—GEOM_POLYTRI 335
Object transformation routines 336
Automatic placement—all GEOMobj 336
Extent—all GEOMobj 336
Object property routines 336
Edit list manipulation routines 336
Creation—GEOMedit_list 336
Object insertion—all GEOMobj 336
Object transformation—all GEOMobj, light, and camera 337
Object hierarchy—all GEOMobj 337
Object properties—all GEOMobj 337
Light source—light 337
Geometry file utilities 337
Input and output—all GEOMobj 337
Object management—all GEOMobj 337
Listed alphabetically 338
GEOMadd_disjoint_line 338
GEOMadd_disjoint_polygon 339
GEOMadd_disjoint_prim_data 340
GEOMadd_disjoint_vertex_data 340
GEOMadd_float_colors 341
GEOMadd_int_colors 342
GEOMadd_int_value 342
GEOMadd_label 343
GEOMadd_normals 344
GEOMadd_polygon 345
GEOMadd_polygons 346
GEOMadd_polyline 347
GEOMadd_polyline_prim_data 347
GEOMadd_polyline_vertex_data 348

GEOMadd_polytriangle 349
GEOMadd_polytriangle_prim_data 350
GEOMadd_polytriangle_vertex_data 350
GEOMadd_prim_data 351
GEOMadd_radii 351
GEOMadd_vertex_data 352
GEOMadd_vertices 352
GEOMadd_vertices_with_data 353
GEOMauto_transform 354
GEOMauto_transform_list 354
GEOMauto_transform_non_uniform 354
GEOMauto_transform_non_uniform_list 355
GEOMcreate_label 355
GEOMcreate_label_flags 356
GEOMcreate_mesh 357
GEOMcreate_mesh_with_data 358
GEOMcreate_normal_object 359
GEOMcreate_obj 359
GEOMcreate_polyh 360
GEOMcreate_polyh_with_data 361
GEOMcreate_scalar_mesh 362
GEOMcreate_sphere 363
GEOMcvt_mesh_to_polytri 364
GEOMcvt_polyh_to_polytri 365
GEOMdestroy_edit_list 365
GEOMdestroy_obj 366
GEOMedit_center 366
GEOMedit_color 367
GEOMedit_concat_matrix 368
GEOMedit_geometry 369
GEOMedit_light 370
GEOMedit_parent 370
GEOMedit_picked 371
GEOMedit_position 371
GEOMedit_properties 372
GEOMedit_projection 373
GEOMedit_render_mode 373
GEOMedit_selection_mode 374
GEOMedit_set_matrix 375
GEOMedit_transform_mode 376
GEOMedit_visibility 378
GEOMedit_window 378
GEOMflip_normals 380
GEOMgen_normals 380
GEOMinit_edit_list 380
GEOMnormalize_normals 380
GEOMquery_int_value 381
GEOMread_obj 381

GEOMset_color 381
GEOMset_computed_extent 382
GEOMset_extent 382
GEOMset_object_group 382
GEOMset_pickable 383
GEOMunion_extents 383
GEOMwrite_obj 383
GEOMwrite_text 384

13 Unstructured cell data 385
What is UCD? 385
UCD hierarchy 386
Comparing UCD structures with fields 389
An example 389
Connectivity 390
UCD formats 391
Internal format 392
Top-level view 392
Creating a UCD structure 393
Instantiating UCD structures 395
Specifying vector length at run time 396
ASCII format 396
Comment section 396
Header section 397
Node position section 397
Cell connectivity section 398
Data sections 399
Examples 400
Binary format 401
Magic number 402
Header 403
Cell block 403
Cell nlists block 404
Coordinate blocks 404
Data blocks 404
Structure of the binary file at a glance 405

14 UCD structure routines 407
Grouped by function 408
Structure manipulation 408
Structure query 408
Cell manipulation 408
Cell query 409
Node manipulation 409
Node query 410
Listed alphabetically 411
UCDcell_get_information 411
UCDcell_set_information 412
UCDnode_get_information 413
UCDnode_set_information 414
UCDstructure_alloc 415
UCDstructure_free 416
UCDstructure_get_cell_active 417
UCDstructure_get_cell_components 418
UCDstructure_get_cell_data 419
UCDstructure_get_cell_label 420
UCDstructure_get_cell_labels 421
UCDstructure_get_cell_minmax 422
UCDstructure_get_cell_unit 423
UCDstructure_get_cell_units 424
UCDstructure_get_data 425
UCDstructure_get_data_label 426
UCDstructure_get_data_labels 427
UCDstructure_get_data_unit 428
UCDstructure_get_data_units 429
UCDstructure_get_extent 430
UCDstructure_get_header 431
UCDstructure_get_node_active 433
UCDstructure_get_node_components 434
UCDstructure_get_node_data 435
UCDstructure_get_node_label 436
UCDstructure_get_node_labels 437
UCDstructure_get_node_minmax 438
UCDstructure_get_node_positions 439
UCDstructure_get_node_unit 440
UCDstructure_get_node_units 441
UCDstructure_invalid_cell_minmax 442
UCDstructure_invalid_node_minmax 442
UCDstructure_set_cell_active 443
UCDstructure_set_cell_components 444
UCDstructure_set_cell_data 445
UCDstructure_set_cell_labels 446
UCDstructure_set_cell_minmax 447
UCDstructure_set_cell_units 448

UCDstructure_set_data 449
UCDstructure_set_data_labels 450
UCDstructure_set_data_units 451
UCDstructure_set_extent 452
UCDstructure_set_header_flag 453
UCDstructure_set_node_active 454
UCDstructure_set_node_components 455
UCDstructure_set_node_data 456
UCDstructure_set_node_labels 457
UCDstructure_set_node_minmax 458
UCDstructure_set_node_positions 459
UCDstructure_set_node_units 460

15 Modules 461
Components 461
Name 461
Type 461
Data 461
Filter 461
Mapper 462
Renderer 462
Ports 462
Parameters 463
Functions 464
Description function 464
Computation function 467
Subroutines and coroutines 469
Subroutine modules 469
Coroutine modules 470
Module examples 470
Handling errors 471
Creating online help 473
Using the AVS_HELP_PATH variable 473
Selective computation 474
Building and linking modules 475
Include files 475
C language include files 475
FORTRAN language include files 475
Compiling and linking modules 476
Converting applications into modules 477
Converting coroutine modules 477
Converting subroutine modules 478
Module internal workings 478
Subroutine modules 478
Coroutine modules 480

16 Module routines 483
Grouped by function 484
Initializing modules 484
Modifying and interpreting parameters 484
Monitoring status 484
Command Language Interpreter 484
Coroutine modules 484
Module description functions 485
Selective computation 485
Creating fields 486
Accessing fields 486
Accessing user data 487
Accessing colormaps 487
Accessing FORTRAN arrays 487
Accessing FORTRAN single bytes 487
Handling errors 487
Defined alphabetically 488
AVSadd_float_parameter 488
AVSadd_parameter 489
AVSadd_parameter_prop 492
AVSautofree_output 496
AVSbuild_2d_field 496
AVSbuild_3d_field 497
AVSbuild_field 498
AVSchoice_number 500
AVScolormap_get 501
AVScolormap_set 502
AVScommand 503
AVSconnect_widget 504
AVScorout_event_wait 506
AVScorout_exec 507
AVScorout_init 507
AVScorout_input 508
AVScorout_mark_changed 508
AVScorout_output 509
AVScorout_set_sync 510
AVScorout_wait 510
AVScorout_X_wait 511
AVScreate_input_port 512
AVScreate_output_port 513
AVSdata_alloc 514
AVSdata_free 514
AVSdebug 515
AVSerror 516
AVSfatal 517
AVSfield_alloc 518
AVSfield_copy_points 519

AVSfield_data_offset 519
AVSfield_data_ptr 520
AVSfield_free 520
AVSfield_get_dimensions 521
AVSfield_get_extent 521
AVSfield_get_int 522
AVSfield_get_label 523
AVSfield_get_labels 524
AVSfield_get_minmax 525
AVSfield_get_unit 525
AVSfield_get_units 526
AVSfield_invalid_minmax 527
AVSfield_make_template 527
AVSfield_points_offset 528
AVSfield_points_ptr 528
AVSfield_reset_minmax 529
AVSfield_set_extent 529
AVSfield_set_int 530
AVSfield_set_labels 531
AVSfield_set_minmax 531
AVSfield_set_units 532
AVSinformation 533
AVSinit_from_module_list 534
AVSinit_modules 535
AVSinitialize_output 536
AVSinput_changed 536
AVSload_byte 537
AVSload_user_data_types 537
AVSmark_output_unchanged 538
AVSmessage 539
AVSmessage_sub 543
AVSmodify_float_parameter 543
AVSmodify_parameter 544
AVSmodify_parameter_prop 546
AVSmodule_from_desc 546
AVSmodule_status 547
AVSparameter_changed 547
AVSparameter_visible 548
AVSport_field 548
AVSptr_alloc 549
AVSptr_offset 550
AVSset_compute_proc 550
AVSset_destroy_proc 551
AVSset_init_proc 551
AVSset_input_class 552
AVSset_module_flags 552
AVSset_module_name 553
AVSset_output_class 554

AVSset_output_flags 554
AVSset_parameter_class 555
AVSstore_byte 555
AVSudata_get_double 556
AVSudata_get_int 557
AVSudata_get_real 558
AVSudata_get_string 559
AVSudata_set_double 560
AVSudata_set_int 561
AVSudata_set_real 562
AVSudata_set_string 563
AVSwarning 564

17 Advanced module topics 565
Coroutine synchronization 565
Coroutine scheduling with X 566
Coroutine scheduling with other devices 566
Synchronous execution 567
Upstream data 568
Upstream data mechanism 568
Implementing upstream data 569
Transformation information 570
Notify mode 571
Redirect mode 571
Selection information 573
Rules for picking objects 575
User–defined upstream data 577
Automatic port connection 578
Port classes 578
Port visibility 579
User–defined data 580
Used with input ports 581
Used with output ports 582
Multiple modules in one process 583
Restrictions 583
Implementing multiple–module processes 585
Implementing reentrant modules 586
Modifying modules that share processes 586

18 Command Language Interpreter 587
Accessing the CLI 587
Command line options 587
Server options 588
Module access 588
The .avsrc file option 589
CLI concepts 590

Commands and tokens 590
Case sensitivity 590
Interrupting CLI execution 591
Multiple line commands 591
Variable references 591
Redirecting output 592
Identifiers 592
Module names and aliases 593
Parameter names 593
Port names 594
Combining networks 594
Module tags 594
Module maps 595
Pend operations 596
CLI scripts 596
Writing scripts 596
Playing back scripts 597
Script Controller browser 598
Script suites 599
Command notations 600
Basic commands 600
General commands 601
Script commands 602
Variables commands 603
Network Editor commands 604
Network commands 605
Module commands 607
Parameter commands 608
Port commands 608
Geometry Viewer commands 609
Matrix operations 609
Global object commands 609
Browser commands 610
Light commands 610
Action commands 610
Object commands 611
Camera commands 612
Image Viewer commands 613
Scene commands 613
View commands 613
Image commands 614
Image processing technique commands 615
Cycle commands 615
Label commands 616

19 Geometry Script Language commands 617
Scene and object files 617
Command summary 618
Object commands 619
read 619
group 619
cycle 619
set_color 620
set_material 621
set_matrix 621
set_position 621
set_render_style 621
rotate 622
translate 622
scale 622
Viewing commands 622
view 622
set_matrix 622
set_position 622
inactive 623
rotate 623
translate 623
scale 623
Lighting commands 624
light 624
set_matrix 624
set_position 624
set_color 624
Defaults file 625
Example scene file 625

Glossary 627

Index 637

Figures

Figure 1 ConvexAVS colormap lookup 5
Figure 2 Network of ConvexAVS modules 8
Figure 3 A module's interface: icon and control panel 10
Figure 4 Module control widgets 12
Figure 5 ConvexAVS Network Editor windows 14
Figure 6 Data flow in a network 15
Figure 7 Complex network structure 16
Figure 8 One-mapper geometry network 19
Figure 9 Arbitrary slice plane 19
Figure 10 Three geometry mapper modules in parallel 20
Figure 11 Three geometric objects in one view 21
Figure 12 Simple imaging technique network 22
Figure 13 Simple imaging technique network display 23
Figure 14 Tracing, gradient shading, and slicing combined 24
Figure 15 Tracer and orthogonal slicer images 25
Figure 16 Network to produce a graph and example display . 26
Figure 17 Plotting distribution with generate histogram 27
Figure 18 Bar plot in Graph Viewer 27
Figure 19 Techniques combined in a single network 28
Figure 20 Image, Geometry, and Graph viewer windows 28
Figure 21 A network to visualize vectors 30
Figure 22 Stream lines and scatter dots 30
Figure 23 ConvexAVS Main and AVS Applications menus 53
Figure 24 Data Viewers button 55
Figure 25 Help browser 57
Figure 26 Script demo controller 58
Figure 27 File browser widget 61
Figure 28 Directory and file name dialog 62
Figure 29 Network Control panel and construction window .. 66
Figure 30 Status widget 68
Figure 31 Module palette 70
Figure 32 Module icon 73
Figure 33 Module Editor 74
Figure 34 Scroll icon for a module category 77
Figure 35 Dragging a module into the workspace 79
Figure 36 Lassoed modules 81

Figure 37 Parameter ports connected to the integer module ... 84
Figure 38 Orthogonal slices of a hydrogen molecule ... 85
Figure 39 Simultaneous control of hedgehog and stream lines 85
Figure 40 Using animated float for upstream control ... 86
Figure 41 Upstream data ports made visible ... 87
Figure 42 Color-coding for field input/output ports ... 88
Figure 43 Gray color-coding as wildcard value ... 89
Figure 44 Making a module port visible ... 91
Figure 45 probe upstream data paths made visible ... 93
Figure 46 Organization of the Network Control panel ... 94
Figure 47 Type-in control widget ... 97
Figure 48 Dial control widgets ... 98
Figure 49 Dial Editor ... 98
Figure 50 Slider control widget ... 101
Figure 51 Set of radio buttons ... 101
Figure 52 Toggle control widget ... 102
Figure 53 Tristate control widgets ... 102
Figure 54 One-shot control widget ... 102
Figure 55 File browser control widget ... 103
Figure 56 Colormap control widget ... 105
Figure 57 Edit panel ... 107
Figure 58 Network Editor main menu ... 114
Figure 59 Window borders in the Layout Editor ... 122
Figure 60 Remote host browser ... 130
Figure 61 Edit Module Library panel ... 133
Figure 62 Geometry Viewer access ... 137
Figure 63 Network that produces three geometries ... 139
Figure 64 Three geometries combined in one scene ... 139
Figure 65 Geometry Viewer control panel ... 146
Figure 66 Top control bar ... 147
Figure 67 Rotating an object with the virtual trackball ... 149
Figure 68 Transformation Options panel ... 152
Figure 69 Absolute transformation values ... 154
Figure 70 Bounding Box button ... 155
Figure 71 Current object selection area ... 156
Figure 72 Current object browser ... 157
Figure 73 Objects menu selections ... 161
Figure 74 Geometry file browser ... 162
Figure 75 Entering a file name ... 163
Figure 76 The Edit Property window ... 165
Figure 77 Lights menu selections ... 171
Figure 78 Symbols for light types ... 173
Figure 79 Cameras menu selections ... 174
Figure 80 Labels menu selections ... 179
Figure 81 Font selection submenu ... 181
Figure 82 Action menu selections ... 183
Figure 83 Image Viewer button and module widget ... 189
Figure 84 Image Viewer control panel ... 194

Figure 85 Image Viewer top control bar 195
Figure 86 Transform Selection menu 196
Figure 87 Current Image controls 198
Figure 88 Current Image browser 199
Figure 89 Image Viewer menu buttons 200
Figure 90 Images button submenu controls 201
Figure 91 Viewport windows 203
Figure 92 Images submenu controls 205
Figure 93 Zoom In image showing pixel replication 208
Figure 94 Views submenu 210
Figure 95 Image scaled in the Y direction only 213
Figure 96 Background color controls 213
Figure 97 Image Processing submenu 214
Figure 98 Processing technique browser 215
Figure 99 Network and panel for gradient shade technique 217
Figure 100 Subimage area 218
Figure 101 Subimage in a new window 219
Figure 102 Labels submenu 226
Figure 103 Current Label type-in 227
Figure 104 View with labels and titles 228
Figure 105 Action menu 230
Figure 106 Action network 231
Figure 107 Example network 235
Figure 108 Graph Viewer control panel 239
Figure 109 Option Selection menu 240
Figure 110 Multiple plot windows. 242
Figure 111 Read Data submenu 243
Figure 112 Plot as Y Data menu 248
Figure 113 Example plot as Y data 249
Figure 114 Plot as XY Data menu 250
Figure 115 Example plot of XY data 251
Figure 116 Plot as contour data menu 252
Figure 117 Example of plotting contour data 254
Figure 118 Write Data File Type menu 256
Figure 119 Axis Display menu 257
Figure 120 Select plot options 262
Figure 121 Cross hair with a selected contour plot 263
Figure 122 Cursor position and selected point display 265
Figure 123 Sample network for importing volume data 276
Figure 124 Rectilinear field 279
Figure 125 Rectilinear field 298
Figure 126 Rectilinear or irregular field 299
Figure 127 Irregular field 300
Figure 128 Irregular field 301
Figure 129 Example of AVSfield structure 307
Figure 130 Diagram of an edit list data flow 321
Figure 131 UCD hierarchy 386
Figure 132 0-D and 1-D UCD cell types and node numbering 387

Figure 133 2-D UCD cell types and node numbering 388
Figure 134 3-D UCD cell types and node numbering 388
Figure 135 Single hexahedron with scalar data 400
Figure 136 Single hexahedron with mixed data 401
Figure 137 Several structures with scalar data 402
Figure 138 C language description function 465
Figure 139 FORTRAN language description function 465
Figure 140 Module selection information 570
Figure 141 Module selection information 574
Figure 142 Simple network CLI script session 605
Figure 143 An example grouping 620
Figure 144 An example cycle definition 620
Figure 145 Sample Geometry Viewer defaults file 625
Figure 146 Example scene file 625

Tables

Table 1 Data file in Brookhaven Protein Data Bank format 4
Table 2 Module input ports and ConvexAVS data types 10
Table 3 ConvexAVS start-up options, configuration keywords, and environment variables 36
Table 4 Color-coding for field input/output ports 89
Table 5 Modules that should not be removed 95
Table 6 Hue values 166
Table 7 Saturation values 166
Table 8 Brightness values 167
Table 9 Geometry Viewer file types 188
Table 10 Sample image processing techniques 222
Table 11 Field declarations for data types 269
Table 12 Application and data type cross reference 272
Table 13 Data file in Brookhaven PDB format 284
Table 14 General filters 285
Table 15 Specific filters 285
Table 16 Postprocessor filters 288
Table 17 Field mappings 296
Table 18 Field declarations andspecializing words 306
Table 19 Geometry objects and types 323
Table 20 Geometry types compatibility 328
Table 21 Changing data declarations 329
Table 22 Cell types and number of nodes 387
Table 23 Domain and range of fields and UCD structures 391
Table 24 Cell types and name abbreviations 398
Table 25 Module types 476
Table 26 Type values 490
Table 27 Property names and types 495
Table 28 Parameters and widgets 505
Table 29 Input ports and data types 512
Table 30 Module types and names 553
Table 31 GSL command summary 618

How to use this book

Purpose and audience

Using ConvexAVS to Visualize Data describes use of the Convex version of the Application Visualization System (AVS).

This is intended as both a user's guide and programmer's reference that includes the following information:

- Introductory material about the Application Visualization System
- How to get started
- Details about the subsystem tools
  - Using the Network Editor
  - Using the Image Viewer
  - Using the Geometry Viewer
  - Using the Graph Viewer
- Detailed information required to create custom modules and networks

This guide addresses scientists and engineers who want to visualize their data using ConvexAVS.

Organization

The fundamental information for using ConvexAVS is in Chapters 1 through 6. Online information is available through the online help system. Chapters 7 through 18 include detailed information needed to create custom modules to process data.

Specifically, this guide is organized into the following chapters and appendixes:

- Chapter 1, "What is ConvexAVS?," introduces general concepts about ConvexAVS.
- Chapter 2, "Starting ConvexAVS," describes the setup, environment variables, configuration files, menus and controls used to enter ConvexAVS.
- Chapter 3, "Network Editor subsystem," details the Network Editor subsystem.
- Chapter 4, Geometry Viewer subsystem," details the Geometry Viewer subsystem.
- Chapter 5, "Image Viewer subsystem," details the Image Viewer subsystem.
- Chapter 6, "Graph Viewer subsystem," details the Graph Viewer subsystem.
- Chapter 7, "Data types and import strategies," describes the recognized data types and methods for importing your data into ConvexAVS.
- Chapter 8, "Fields," describes the various field types and the C language field macros.
- Chapter 9, "Colormaps and pixel maps," describes the colormap structure.
- Chapter 10, "Geometries," describes geometry edit lists and design considerations.
- Chapter 11, "Geometry routines," contains a list of supported geometry routines, grouped by topic and alphabetically.
- Chapter 12, "Unstructured cell data," describes the UCD data structures.
- Chapter 13, "Unstructured cell data routines," contains a list of supported UCD routines, grouped by function and alphabetically.
- Chapter 14, "Modules," describes the construction of ConvexAVS modules and how to create new ones.

- Chapter 15, "Module routines," contains a list of supported module routines, grouped by function and alphabetically.
- Chapter 16, "Advanced module topics," describes coroutine synchronization, upstream data, ports, and user-defined data handling.
- Chapter 17, "Command Language Interpreter," describes the Command Language commands and usage.
- Chapter 18, "Geometry Script Language commands," describes the commands used to create scripts that control the Geometry Viewer functions, grouped by function.

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.
- **Bold** standard font is used to highlight module names
- *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
  - Data structures and types
  - Directives, program statements, display examples, printout examples, and error messages returned
- Horizontal ellipsis (...) shows repetition of the preceding item(s).

- 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. Buttons that you click on the screen with the mouse are also in this font.
- The word "enter" in a phrase such as "enter **`ls`**" means that you type the command and then press **RETURN**.
- References to the *ConvexOS man pages* appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

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.

- *ConvexAVS Module Reference* (DSW-305). This book is the standard reference for ConvexAVS.
- *Animating AVS Data Visualizations* (DSW-306). This book describes how to use the AVS Animation Application modules to create animations of your data visualizations.

Ordering documentation

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

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

Include order number or exact title, as listed on the front 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 continental U.S., contact local CONVEX office.

Acknowledgments

We would like to thank the following groups for their contributions to this document. This book would not have been possible without their help.

- Development—technical content
- Testing—technical accuracy
- Documentation and Training—technical support
- Editorial Services—technical quality

Henry C. Smith II
Technical Communication Specialist

Neal W. Johnston
Technical Communication Leader

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 display techniques.

A limiting factor in this growing field has been the existing software tools, which 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 basic data into a more usable form (more informative, smaller, and so on.)

Mapping filtered data into geometric primitives (triangles, lines, spheres, and so on.)

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

Flow networks are built from a menu of modules by using a direct-manipulation interface called the ConvexAVS *Network Editor*. 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.

The user views, organizes, and further processes the output of a network through one of the ConvexAVS subsystems. The *Geometry Viewer* displays 3-D geometric objects. The *Image Viewer* displays 2-D images. The *Graph Viewer* creates XY and contour graphs of data.

ConvexAVS includes a rich set of modules for construction of networks. ConvexAVS also supports the creation and dynamic loading of new modules. Detailed knowledge of the ConvexAVS implementation is not required. 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:

- 1-D, 2-D, and 3-D grids of numbers with scalar values or vectors of byte, integer, or floating-point values at each grid point. The grids can be regular (*uniform*) or irregular—where the distance between the grid points is variable (*rectilinear*). It is also possible for the grid to describe a curved or arbitrarily deformed space (*curvilinear*), or an arbitrary list of points in 3-D space (*scatter data*). This type of data is called a *field*.
- Unstructured cell data
- Geometric data
- Images

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, and so on).

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 2-D square grid (a 200x200 matrix). Similarly, a sequence of 500,000 numbers might represent a 100x200x25 lattice of data points.

A numerical grid is assumed to correspond 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.

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 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
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
...
```

Scientific visualization techniques

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 3-D geometric 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 table lookup, called a *colormap*. You can define, save, and retrieve your own colormaps. ConvexAVS includes an interactive Colormap Editor 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. 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.

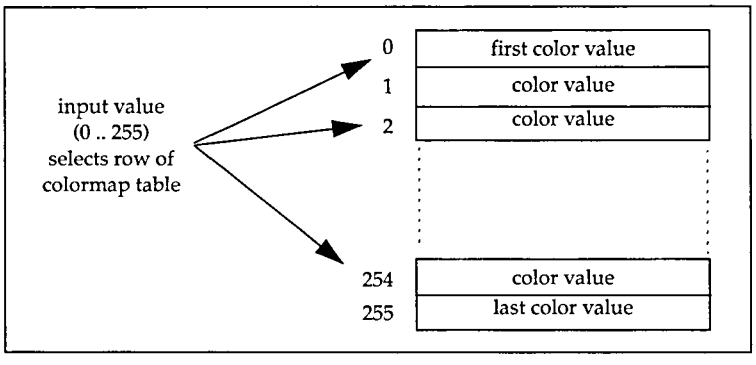


Figure 1
ConvexAVS
colormap lookup

By default, 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. However, you can also have ConvexAVS automatically scale the colormap to any integer or floating point data range. A gray-shade colormap can be used, if no colors are required.

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 display with only 8 or 12 color planes.

Further pixel processing

If multidimensional data is converted to pixels, the results must somehow be reduced to 2-D before they can be displayed as an image on screen. ConvexAVS provides several ways to perform such reductions:

- **Slicing**—A 2-D cross-section can be made through a 3-D block of pixels (**orthogonal slicer** module).
- **Blending**—If a 3-D block of pixels is passed though a colormap whose auxiliary field contains opacity and transparency data, pixels can be blended along the line of sight. This produces a 2-D picture of what appears to be a solid or semi-transparent object in space (**tracer** module).

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 6-by-6 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, 3-D 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 3-D 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 (**isosurface** module).
- 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 (**hedgehog** module).
- Given window velocity data as above, construct flow lines to represent the motion of an object through the field (**stream lines** module).

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.

Graph Viewer

The **Graph Viewer** subsystem is a tool for creating 2-D linear and contour graphs of data.

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, and so on.), and control the way in which the graphical images are rendered (lighting and shading, and so on). Multiple objects, such as an isosurface and a slice plane, can be combined into a single scene depicted in a display window.

3-D graphics rendering techniques (lighting models, 2-D and 3-D texture mapping, automatic removal of hidden surfaces, sphere rendering, and so on) rely heavily on the underlying capabilities of an individual platform's graphics subsystem, both hardware and software. Platforms differ in their support of these techniques. ConvexAVS attempts to use all of the graphics functionality present on a platform.

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.

Applications

The **Applications** button produces another menu of choices. Two sample applications are provided. Users can add their own applications to this menu.

Command Language Interpreter

The Image Viewer, Geometry Viewer, and Network Editor can also be driven through a Command Language Interpreter (CLI). You can type CLI commands in response to a prompt and interactively view the results, you can create a *command script* file that executes automatically, and you can write a module that sends CLI commands to the Image Viewer, Geometry Viewer, and the Network Editor via the ConvexAVS kernel.

The CLI also supports a *scripting* facility in the Network Editor. With scripting switched on, ConvexAVS records most of your actions in the Network Editor into an ASCII file than can be edited to produce a automated demonstration. This technique was used to produce the illustrative scripts stored in the /usr/avs/demo/man_scripts directory, accessible through the Help facility's **Help Demos** menu option.

See "Accessing the CLI," on page 587, in Chapter 17 for more information.

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 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 2 shows a network of modules.

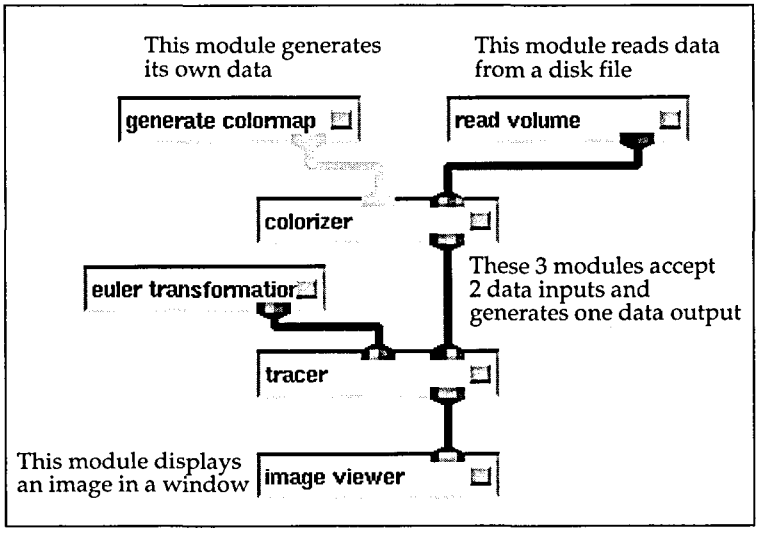


Figure 2
Network of ConvexAVS modules

ConvexAVS modules can execute locally on the same host system as the ConvexAVS program, or modules can execute *remotely*—on another host of the same (homogeneous) or different (heterogeneous) hardware type that runs ConvexAVS. A ConvexAVS module, compiled, linked, and stored on the remote host, is easily added to any ConvexAVS network. This remote module might be a data read and transform process, or a simulation that executes most efficiently on a network compute server.

The remainder of this section discusses the characteristics of individual modules. Networks of modules are discussed in the following section.

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, which includes:

- A set of *data inputs* (some optional, some required)
- A set of *input parameters* that control the way the module processes its input data or determines which data to use. One of ConvexAVS's most powerful features is that you can change parameter values interactively as a network executes. Input parameters can themselves be made into data input ports and receive values from other modules.
- A set of *data outputs*

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).

When you use ConvexAVS to create a network, each module's interface is represented visually by a *module icon* and a *control panel* as shown in Figure 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, and so on); the controls are assembled in a separate control panel window.

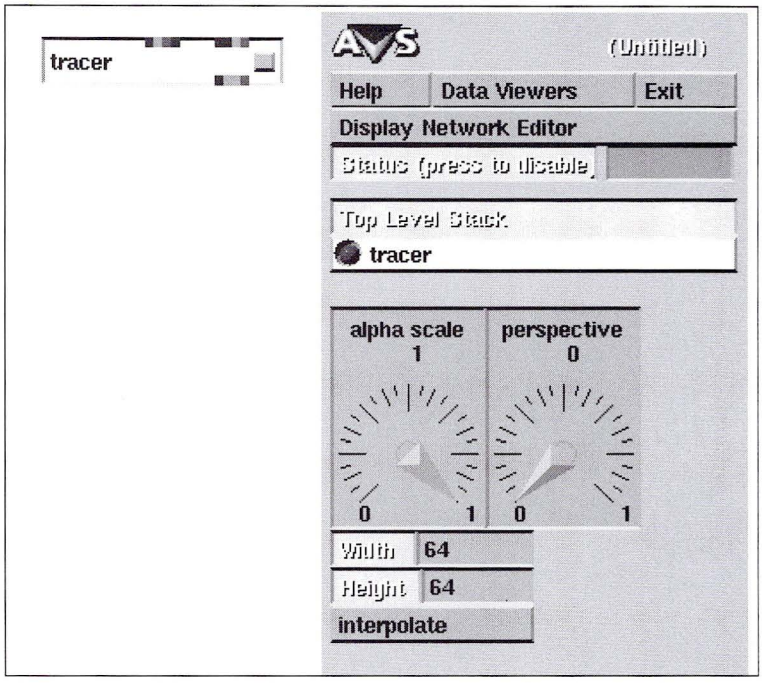


Figure 3
A module's interface: icon and control panel

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, and so on.The module doesn't care where its input data comes from, only that the data types are correct.

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, shown in Table 2.

Table 2
Module input ports and ConvexAVS data types

Port color	Data types
red	geometry
yellow	colormap
light blue	pixmap
multi-color	field
orange	unstructured cell data
light purple	integer
dark purple	floating point
green	string
white	user defined data

ConvexAVS checks 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 2 to illustrate several types of parameters:

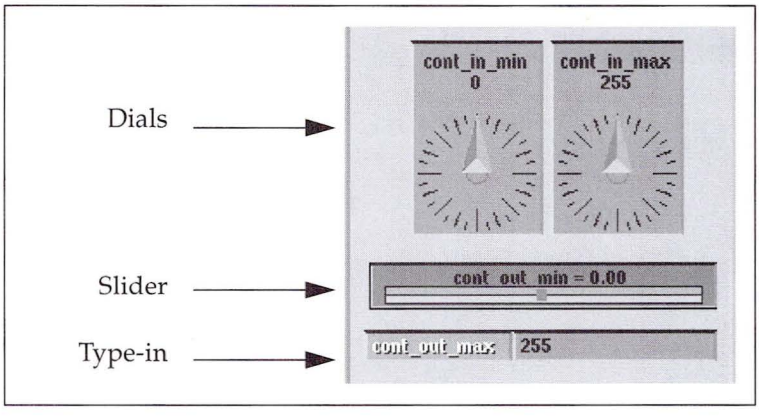
- **The read volume** module brings a 3-D block of byte values into a network. Its input parameter specifies the file from which the values are to be read.
- The **generate colormap** module creates and outputs a colormap that transforms byte values into color values. Its input parameter is implemented as an interactive colormap editor, with which you specify the 256-entry colormap.
- The **tracer** module reduces a 3-D block of partially-transparent color values to a 2-D image using a ray tracing algorithm. Its input parameters control the size of the output image, the opacity, the interpolation method used, and the global perspective.
- The **image viewer** module is the ConvexAVS Image Viewer. It displays the output image. It includes facilities for composing scenes of multiple images, of saving images to disk, for performing image processing techniques upon the image output of other modules, and for creating flipbook animations.

Parameters are the control knobs for a module. By adjusting the knobs, you can control the way in which a module processes its data. 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 4 presents examples of control widgets.

Figure 4
Module control widgets



Widget types

ConvexAVS implements the following types of control widgets:

- **Dials and sliders**—Indicate integers or floating point values.
- **Type-ins**—Specify a character string: title, label, file name, and so on. Type-ins can also be used to specify numeric values: integers or floating-point numbers.
- **Toggles**—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).

ConvexAVS also provides a set of Data Input modules that will produce each of the standard parameter data types (integer, floating point, string, and so on.) and send them to other modules' input parameter ports.

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 "Subroutines and coroutines," on page 469, and "Coroutine synchronization," on page 565.

- *Subroutine* modules are essentially passive, like subroutines in a standard program. When you execute a network, each subroutine module initializes itself (a system 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, it passes the output data back to the Flow Executive, then returns to its dormant state. Execution is synchronous; one module is active at a time.
- *Coroutine* modules are active, not passive. Rather than being like a subroutine, a coroutine is an autonomous, cooperative process that can continually execute, passing data to the Flow Executive on its own initiative, instead of doing so only when it is signalled. Coroutine modules typically implement computational simulations, such as repeatedly releasing particles to flow through a vector field.

Standard modules and module libraries

The ConvexAVS product includes a large number of general-purpose modules. This means that, often without any programming, you can begin to visualize your data sets.

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 or using an interactive module library editing facility

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.

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, shown in Figure 5. To make and break connections between modules, you click-and-drag the middle mouse button.

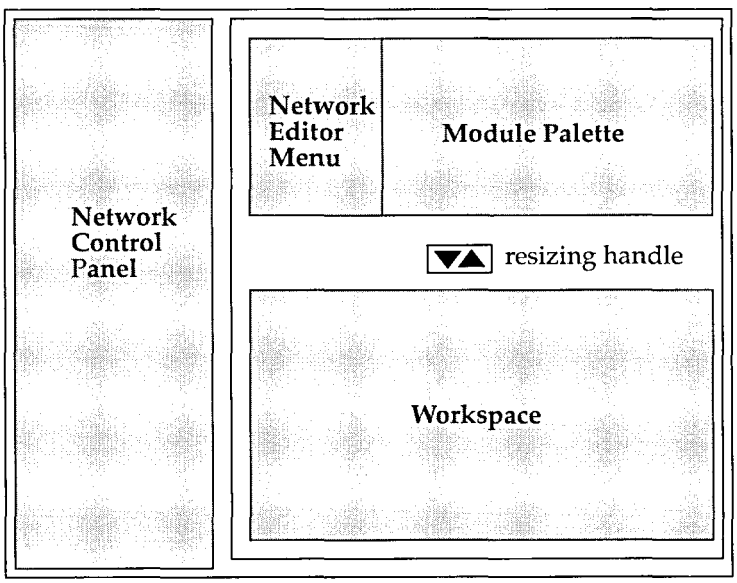


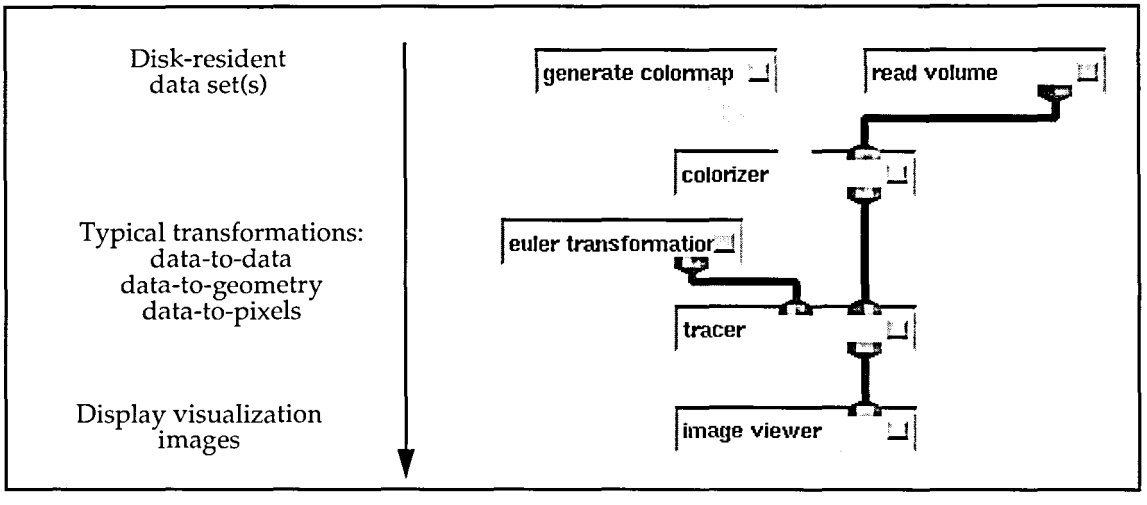
Figure 5
ConvexAVS Network Editor windows

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.

Data flow in a ConvexAVS network

Figure 6 repeats the network shown in Figure 2. The figure emphasizes the way data flows through a typical ConvexAVS network.

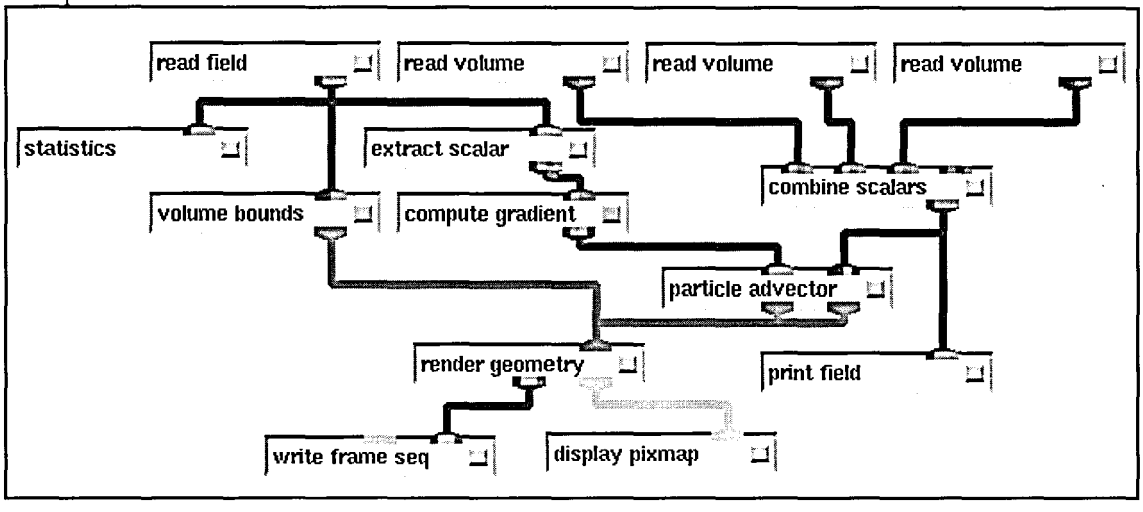
Figure 6
Data flow in a network



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 7 illustrates a more complex network.

Figure 7
Complex network structure



There are limits to network complexity, however. Networks are inherently flat—AVS provides no support for creating hierarchical structures. Networks may contain only one kind of cycle: a module's output data can subsequently be fed back into the immediately previous module as input, using the *upstream data* mechanism, described in "Upstream data," on page 568. The two modules must agree on the data structure they are communicating with. Upstream data ports and connections are usually invisible.

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 Editor 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.

Visualization network example

The following series of figures illustrate using the Network Editor's visual programming interface to construct a series of visualization networks.

All of the networks use the same sample data set, available online in the hydrogen.fld file in the file /usr/avs/data/field directory. The hydrogen.fld file is a 64 by 64 by 64 uniform field, where each data value is a byte quantity from 0 to 255 that represents the probability of an electron occurring around the nucleus of a hydrogen atom.

This same data set also exists as a volume-format file in /usr/avs/data/volume/hydrogen.dat. We use its field representation because it has more general applicability as an example.

With this single data set, we illustrate:

- Three geometry-based visualization techniques. The mapper modules involved (**isosurface**, **arbitrary slicer**, and **volume bounds**) produce 3-D geometry output representations of the input data. These geometries are assembled together and viewed through the ConvexAVS Geometry Viewer in its **render geometry** and **display pixmap** module form.
- Two image-based visualization techniques. The mapper modules (**orthogonal slicer** and **tracer**) produce 2-D colorized image output representations of the input data. These images are assembled together and viewed through the ConvexAVS Image Viewer in its **image viewer** module form.
- Two graph-based visualization techniques. In these techniques, the output of the **orthogonal slicer** and **generate histogram** modules are viewed as contour and linear plots with the ConvexAVS Graph Viewer in its **graph viewer** module form.

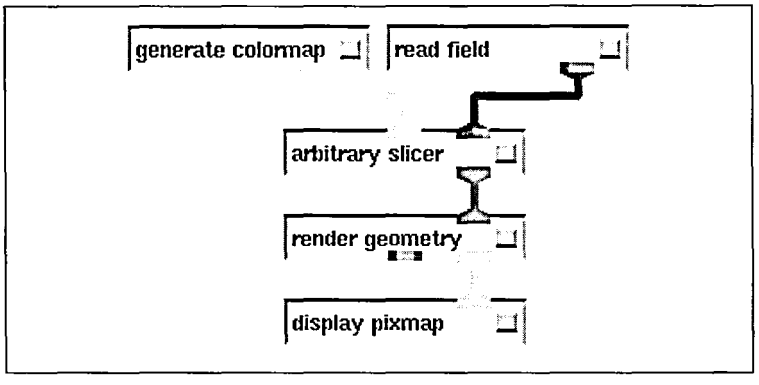
Over 140 visualization modules are supplied with ConvexAVS. The samples given below show only the most basic network structures applied to the simplest of input data sets. For more examples of visualization networks:

- See the *ConvexAVS Module Reference*. Each module has one or more example networks that show how it is used in conjunction with other modules.
- Within ConvexAVS, click on the **Help** button at the top of each subsystem's control panel. The avs_help **Window** menu includes a **Help Demos** option. Selecting this brings up a demonstration script browser. These scripts automatically construct illustrative networks, then run the networks with a variety of parameter settings. When the network completes, it stays in the Network Editor where you can manipulate the modules' parameter widgets and watch their effects.

Mapper modules: Geometries

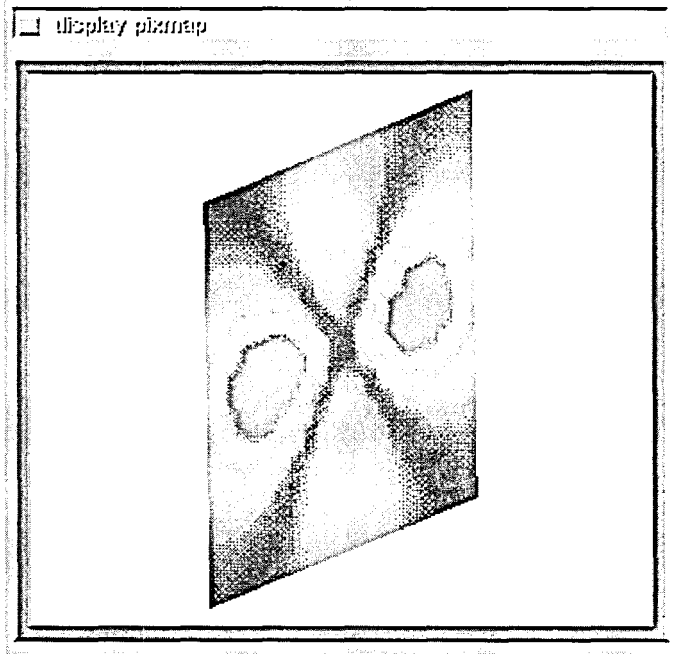
Figure 8 is a simple network. In this network, the **read field** module reads the /usr/avs/data/field/hydrogen.fld dataset into the network.

Figure 8
One-mapper geometry network



The **arbitrary slicer** module receives this data as input and creates a 2-D slice plane through the 64 x 64 x 64 volume of data, as shown in Figure 9.

Figure 9
Arbitrary slice plane



The slice plane would be a featureless gray plane, except that **arbitrary slicer** uses the colormap it receives from the **generate colormap** module to paint the numeric values intersected by the 2-D slice plane different colors.

In the default colormap, small values map to various shades of blue, mid-range values to greens, yellows, and oranges; while larger values map to reds. (The reds and blues appear nearly the same shade of gray in this black and white representation.) The color image would show the two circular medium-gray areas as red, indicating a high probability of an electron occurrence; the peripheral dark gray areas as blue indicating low probability, while the areas in between have fine distinctions of intermediate colors indicating intermediate probability of an electron.

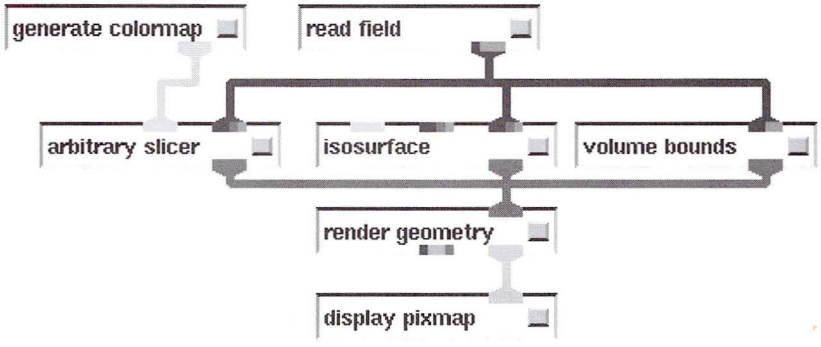
The mapping between numbers and colors is entirely arbitrary. You control the mapping with the **generate colormap** module's Colormap Editor widget, as discussed in "Using the colormap control," on page 105.

The **arbitrary slicer** module sends the geometric slice plane to the Geometry Viewer represented by the **render geometry** and **display pixmap** module pair for display and manipulation. This produces the output window. The Geometry Viewer's **Normalize** button is selected to center the slice plane in the view window, then the slice plane is rotated in space by using the mouse (middle button pressed) to drag it.

Composite mapping techniques

Figure 10 shows a slightly more complex network. The network is identical to the previous one, except that two additional geometry mapper modules, **isosurface** and **volume bounds**, have been added in parallel to the existing **arbitrary slicer** module. They receive the same hydrogen data set as input from the **read field** module.

Figure 10
Three geometry mapper modules in parallel



The **isosurface** module creates a 3-D contour through a volume of data. You supply its **level** parameter widget with a numeric value and **isosurface** produces a geometric surface that passes through all of the data values in the volume that equal the level value.

The **volume bounds** module is a utility mapper module that simply creates a geometry object that is a box around the data set's limits or *extents* in space. It shows where the data ends, how it is shaped, and permits you to orient yourself in space. the *hydrogen.fld* data set describes a uniform, cubical field, hence the cubical volume bounds. Curvilinear data produce much more interesting volume bounds.

All three of the geometries output by the mappers enter the same **render geometry** module. Hence, they are combined into one Geometry Viewer scene window, shown in Figure 11.

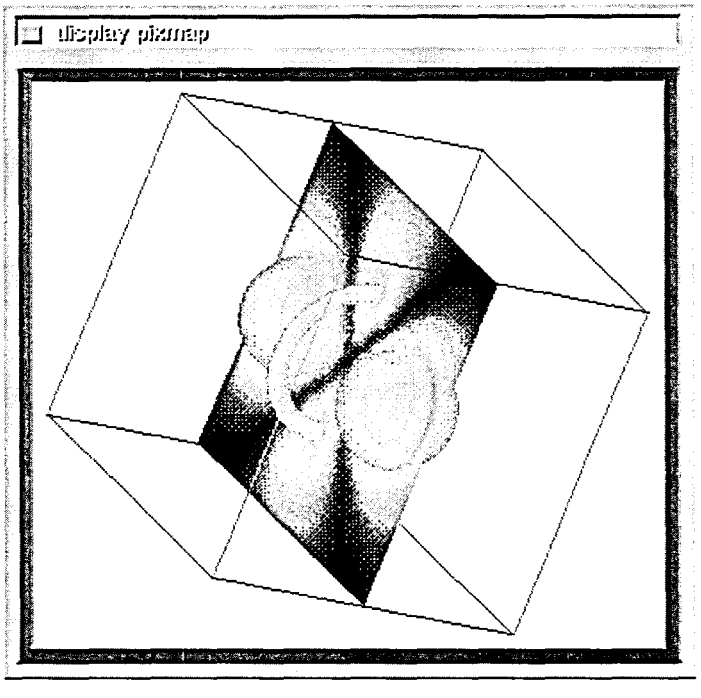


Figure 11
Three geometric objects in one view

The isosurface shows vividly the nature of the distribution of data values throughout the volume of the data set. One could have surmised this by moving the **arbitrary slicer**'s plane around the volume and piecing together an impression of the "doughnut and spheres" structure out of its 2-D portrayals, but **isosurface** captures the structure more graphically.

An isosurface is usually opaque. Here the user has used the Geometry Viewer's transparency controls (not present on all platforms) under its **Edit Property** button to make the surface semi-transparent, letting the colors of the **arbitrary slicer**'s slice plane show through.

Mapper modules that produce images

The next example networks use mapper modules that produce 2-D images to represent data rather than geometries. Figure 12 shows a simple imaging techniques network. As with the geometry network, the first module, **read field**, inputs the data to the network.

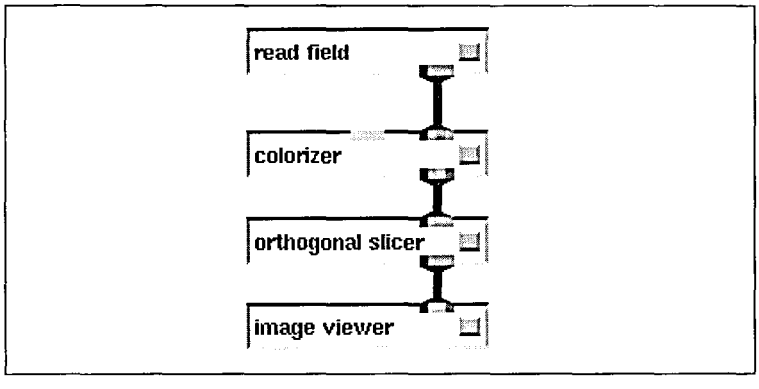


Figure 12
Simple imaging technique network

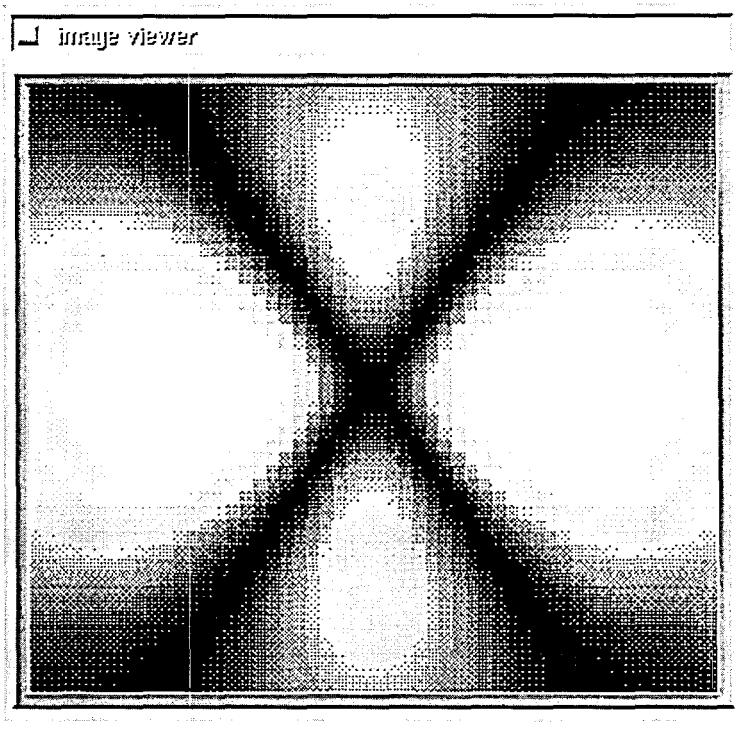
The **orthogonal slicer** module also produces a 2-D slice through a 3-D volume.

It differs from **arbitrary slicer** above in two regards. First, the slice plane must be orthogonal to the X, Y, or Z axis instead of arbitrarily placed within the volume. Second (and more significant) **orthogonal slicer** takes a 3-D field volume of data and outputs another *2-D field*. The **arbitrary slicer** module outputs a geometry which can have only one destination—the **render geometry** module. By outputting another field, **orthogonal slicer**'s output can be sent to other data filtering and mapper modules for further processing. It is a flexible tool for creating data subsets.

A close look reveals that **orthogonal slicer** has no colormap input port as **arbitrary slicer** has. To gain the same effect, the **colorizer** module is interposed to color the data values according to a colormap. **colorizer** effectively changes a field of numeric probability data into a field of color values, which **orthogonal slicer** separates into subsets.

Figure 13 shows the 2-D output displayed in the Image Viewer (**image viewer** module). The orthogonal slice plane is the middle Z plane of the volume data.

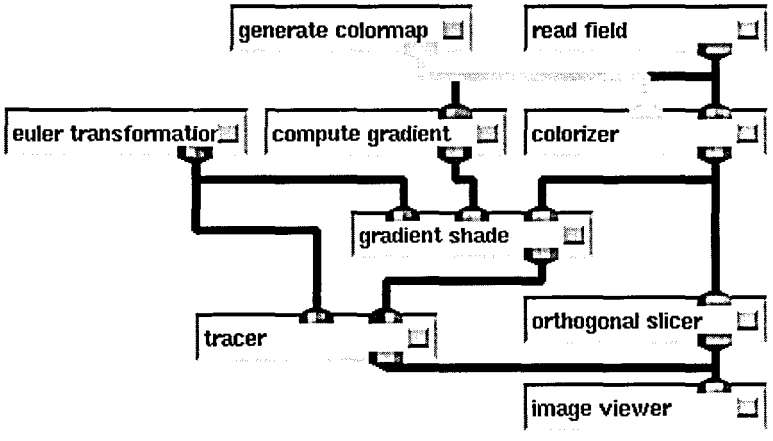
Figure 13
Simple imaging technique network display



Combining imaging techniques

Figure 14 shows a more complex image technique network.

Figure 14
Tracing, gradient shading, and slicing combined



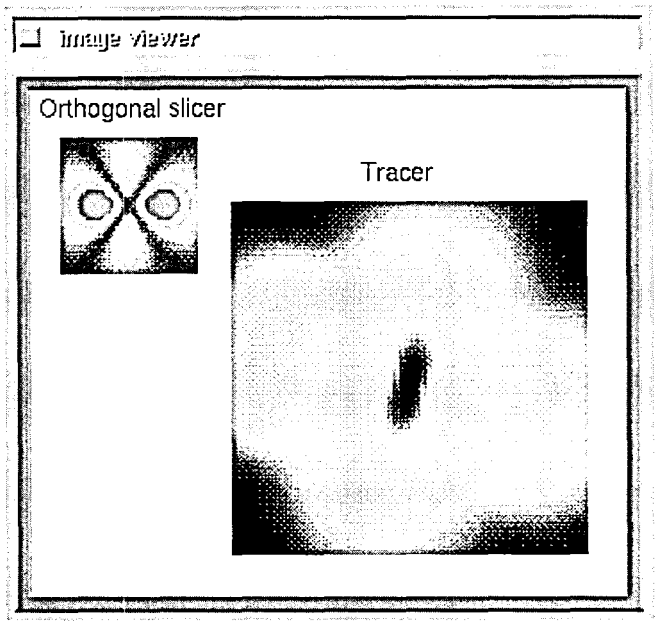
The new mapper module is **tracer**. It produces a ray-traced image rendering of volume data. By shooting hypothetical parallel rays of light into the volume, then coming up with a pixel value that represents how much that ray of light would have been diminished in brightness and changed in color as it passed through the volume, it creates an image that appears to be a solid, perhaps hazy and semi-transparent 3-D object in space.

To heighten the three-dimensional impression, the **compute gradient** and **gradient shade** modules are added to the network that pre-process the hydrogen volume data. These modules calculate how rapidly data is changing within the field (**compute gradient**), and create pseudo-shadows to indicate that change (**gradient shade**). Darker shadows indicate data changing more rapidly. When these pseudo-shadows are overlaid upon the **tracer** image, it creates an even more startling impression of a real, solid object.

The **euler transformation** module feeds parameter data to both **tracer** and **gradient shade**. This parameter-as-data module allows the user to orient the **tracer** module's volume in space; it acts as a kind of "this is where the camera is" module. **The gradient shade** module also receives the transformation data so that its pseudo-shadow overlays are at the same orientation.

Figure 15 shows the two images as they appear in the Image Viewer. The Image Viewer is used to scale, position, and label the two images.

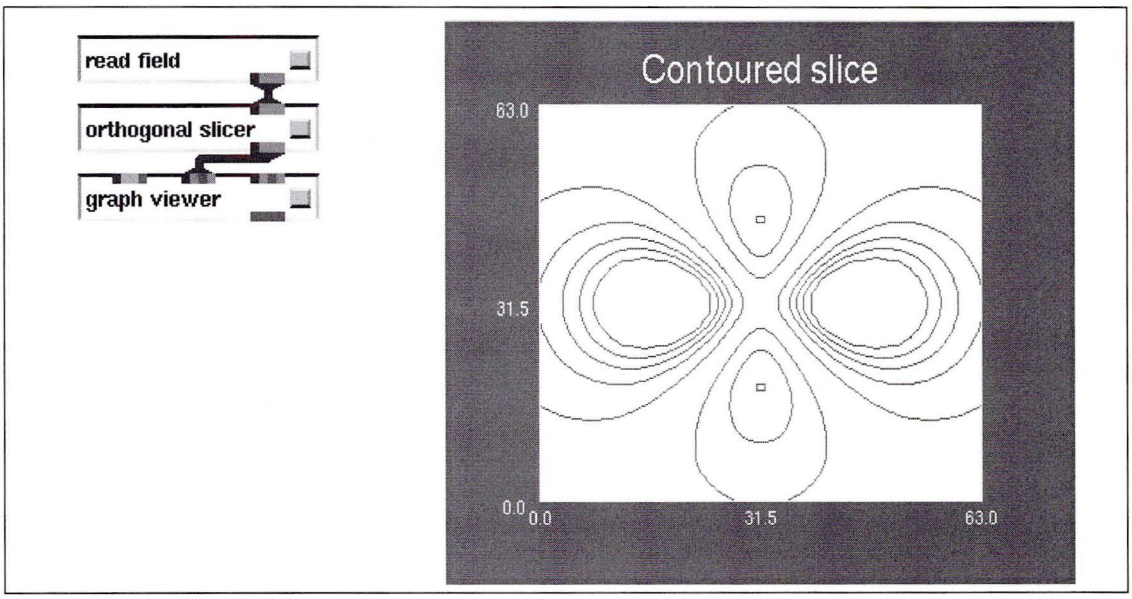
Figure 15
Tracer and orthogonal slicer images



Producing graphs

Figure 16 shows a simple network that produces a graph. The **read field** module reads the hydrogen atom data. The **orthogonal slicer** module sections out a 2-D slice of the 3-D field, which it feeds to the Graph Viewer's (**graph viewer** module) center input port. This port is used to make the Graph Viewer create a contour plot of the data in the 2-D slice. The contour plot that results from this network is shown on the right in Figure 16. The contour is, again, made from the middle Z plane of the hydrogen data.

Figure 16
Network to produce a graph and example display



In the next network, shown in Figure 17, the **generate histogram** module takes the hydrogen data and produces a 1-D field that represents the distribution of data within the data set. It sends this data to a different **graph viewer** module to plot as Y values against X values, in bar plot format, shown in Figure 18. You could use **generate histogram**'s minimum and maximum parameter widgets settings to plot the distribution of different subsets of the data values, effectively zooming in on different data ranges.

Figure 17
Plotting distribution with **generate histogram**

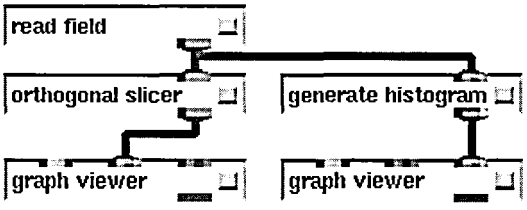
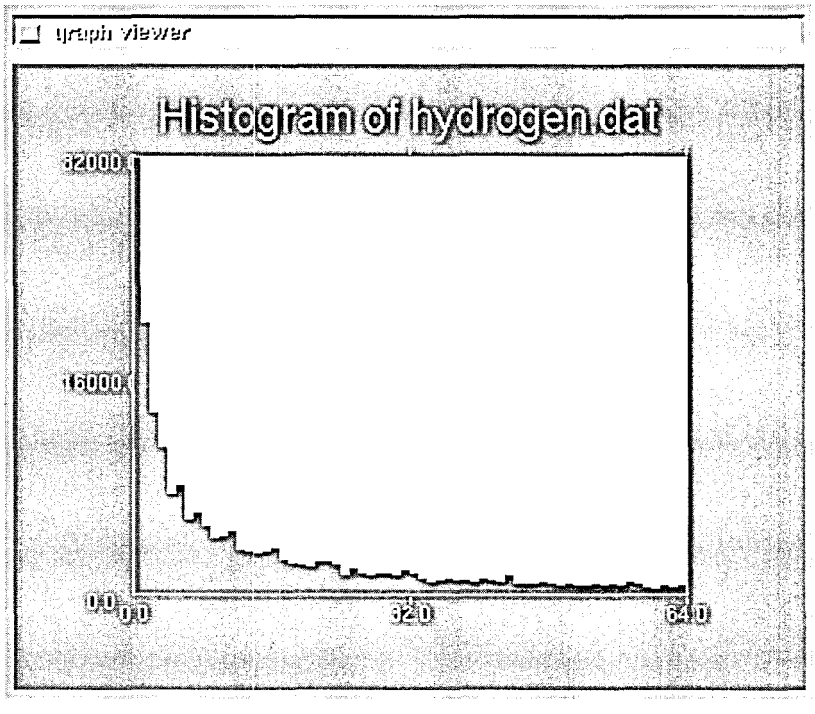


Figure 18
Bar plot in Graph Viewer



Techniques combined

The next two figures show a single network performing all of the visualization techniques shown so far. Figure 19 shows the single, multi-branched network; and Figure 20 shows the display windows it produces together on the screen. Again, all are representations of the single 3-D field data set *hydrogen.fld*.

Figure 19
Techniques combined in a single network

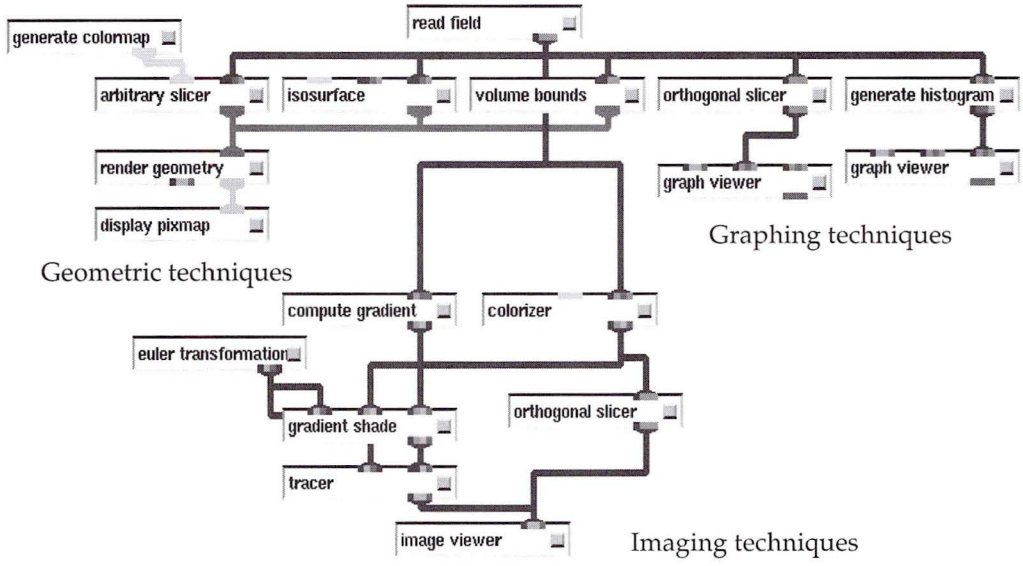
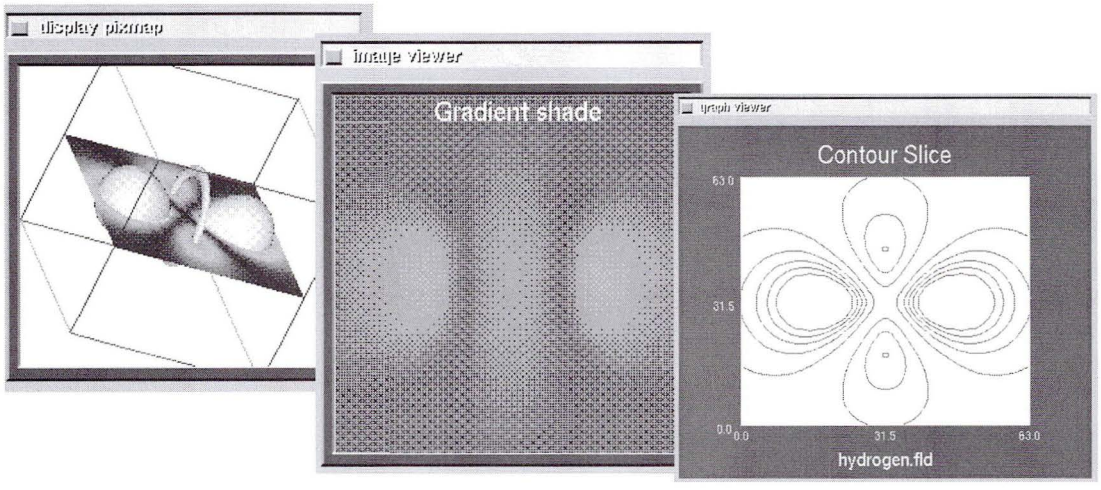


Figure 20
Image, Geometry, and Graph viewer display windows



Curvilinear, vector data

AVS also supports curvilinear, floating point vector data as well as the uniform scalar byte data represented by hydrogen.fld. The following network uses modules to produce geometric renderings that visualize the /usr/avs/data/field/bluntfin.fld data set. (This is actually two PLOT3D-format data sets in /usr/avs/data/plot3d read in using a field header.) The magnitude of the vectors (**extract vector** and **vector mag**) are plotted as spheres (**bubbleviz** and **scatter dots**) whose color reflects the size of the vector; **stream lines** are passed through the data; and the curvilinear shape of the data is reflected by **volume bounds**. The **color range** module is inserted after the **generate colormap** module in order to scale the colormap to the narrow data range (floating point values all near zero), rather than the default 0 to 255 byte value range. An example network is shown in Figure 21 and the resulting display is shown in Figure 22.

Figure 21
A network to visualize vectors

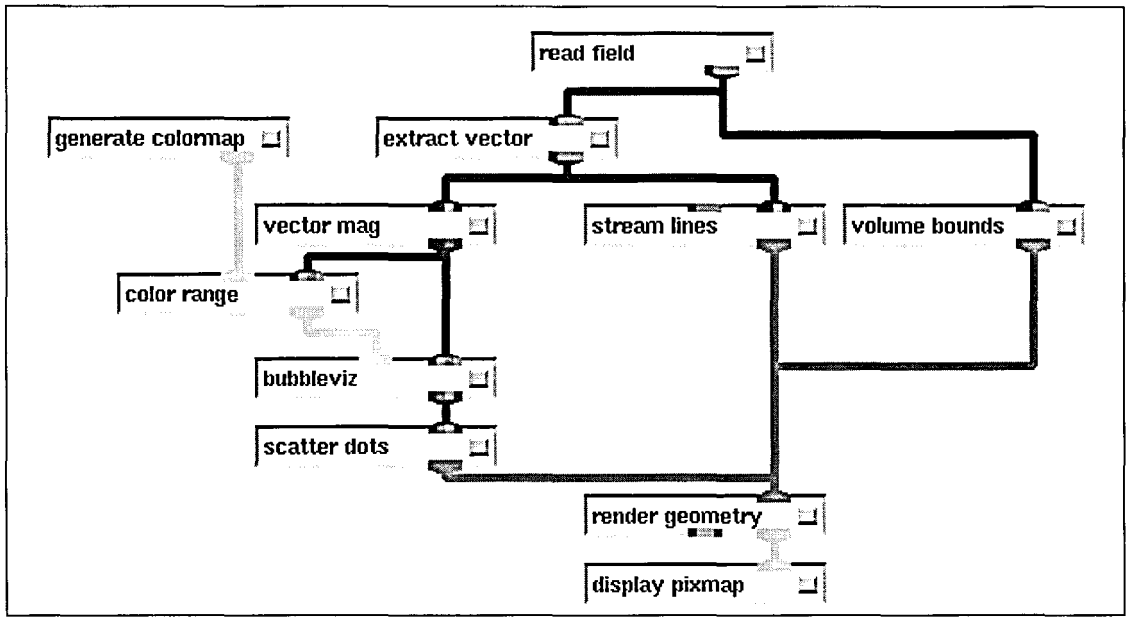
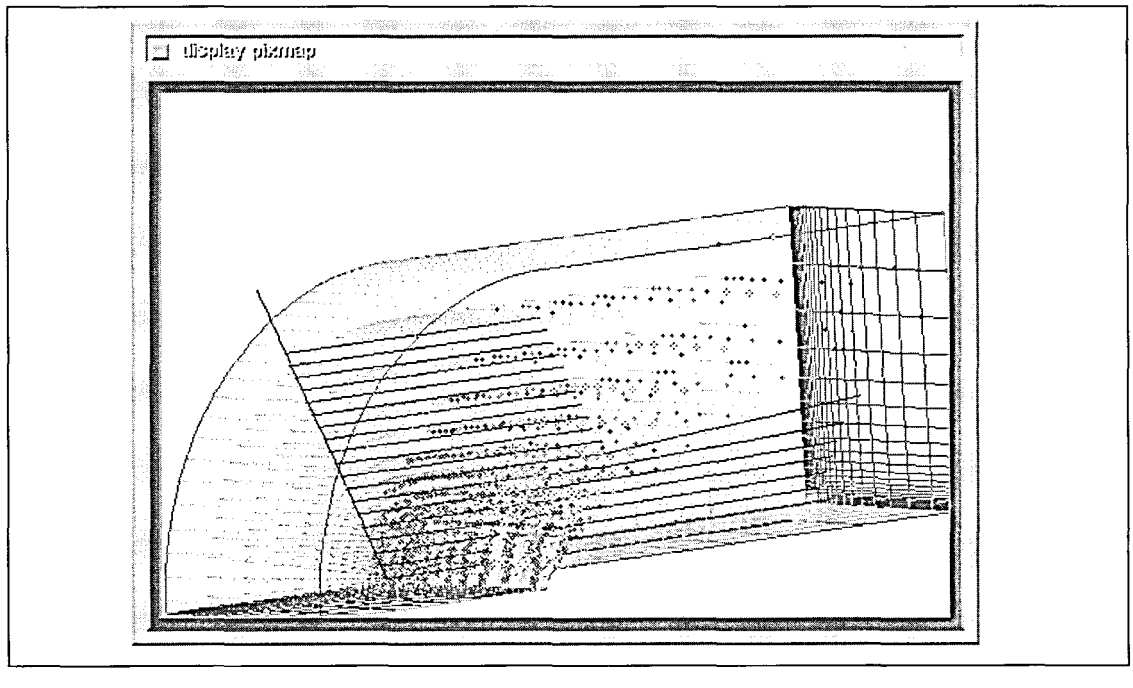


Figure 22
Stream lines and scatter dots in the Geometry Viewer



Starting ConvexAVS

2

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

This chapter describes:

- How to start a ConvexAVS session
- The layout of the main ConvexAVS menus; starting and switching among subsystems
- How to get online help on using ConvexAVS
- Command line, start-up file, and environment variable options that affect how ConvexAVS runs
- How to add applications to the ConvexAVS Applications submenu

ConvexAVS can be run from any workstation or X terminal with color display hardware and an X11 server that supports at least an 8-plane PseudoColor visual.

Note

The first time you execute a newly-installed copy of ConvexAVS you might see a notice explaining that ConvexAVS must be licensed before it will run. Either you or your system administrator should follow the instructions given in the product installation documentation to obtain a ConvexAVS license.

Requirements and setup information

The following requirements apply to this release of ConvexAVS:

- Your system must be running ConvexOS V9.1 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 X11R4 or later
- Has a screen resolution of at least 1024 by 768 pixels
- Its X server has a defined X Window System visual that supports one of the following:
  - PseudoColor
  - TrueColor
  - Limited support for DirectColor

  Use the AVS_VISUAL environment variable to set the default visual. The default visual can also be specified by Visual ID number.

  Monochrome is not supported at this time.

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 makes use of a variety of fonts in different sizes if they are available. This requires 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.

ConvexAVS can usually automatically determine your screen size. You can also 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 font sizes and may degrade the displayed menus and text windows.

Color

The number of color cells ConvexAVS will allocate to itself depends upon how many planes of pseudo color or true color the X visual calls for.

The lowest-usable number of planes is 8-plane pseudo color. In this case, ConvexAVS will try to allocate 236 cells: 6 red x 6 blue x 6 green, plus 26 grey tone cells (6 of the gray cells are shared). If this is unacceptable for some reason, perhaps because it takes too many cells away from other applications, you can modify the default color allocation with the **Colors** start-up file keyword in the .avsrc file.

Environment variables

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

AVS_FORCEX
: Force AVS to always use the X renderer. By default, AVS uses the supported available renderer. Set to one.

AVS_GAMMA *value*
: Used by your display to make the screen lighter at its lowest value (1.0) or darker at the highest value (3.0). The default is 1.5.

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.

AVS_VISUAL *visual_type*
: Specifies the default visual: PseudoColor, DirectColor, or TrueColor. This value can be specified in the .avsrc start-up file as VisualType. You can also specify the visual identification number if your Xserver supports multiple visuals. For example:

```
setenv AVS_VISUAL "VisualID 080064"
```

AVSXDEFAULTS
: Specify the path for the alternate Xdefaults file to be used by ConvexAVS. Refer to"Setting Xdefaults for ConvexAVS," on page 49, for more information about the contents of this configuration file.

DISPLAY *(host:server.screen)*
: The DISPLAY variable is used by the X Window System to indicate the display screen at which you are working.

DISPLAYCLASS X
: Specifies the display class. This value is used to specify a different defaults file in the /usr/avs/runtime directory (for example, `Xdefaults.DirectColor`).

Other environment variables may be required to configure the system for your window manager.

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 **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

A special process, *avsclean*, is forked as soon as avs starts up. It waits for the AVS process to terminate, and then cleans up any temporary files left in /tmp.

Table 3
ConvexAVS start-up options, configuration keywords, and environment variables

Command line option	Start-up file option (.avsrc)	Environment variable
`-appsfile`	ApplicationsFile	
`-class`		DISPLAYCLASS AVSXDEFAULTS
`-cli`		
`-data`	DataDirectory	
`-display`		DISPLAY
	ForceXRender	AVS_FORCEX
`-geometry`		
`-graph`		
`-image`		
`-library`	ModuleLibraries	
`-modules`		
`-netdir`	NetworkDirectory	
`-network`		
`-path`	Path	PATH
`-separate`		
`-server`		
`-shm/noshm`	SharedMemory	
`-size`	ScreenSize	
`-usage`		
`-version`		
`-viewer`		
`-volume`		
	Gamma	AVS_GAMMA
	VisualType	AVS_VISUAL
	BoundingBox	
	Colors	
`-geometry` *-geometry*	DisplayGeometryWindow	
	DisplayPixmapWindow	
	GridSize	
		AVS_HELP_PATH
	Hosts	
	ImageAutomagnify	
	ImageScrollbars	
	ModulePanelHeight	
	NetworkWindow	
	NetWriteAllParms	
	PrintNetwork	
	PdbDataDir	
	ReadOnlySharedMemory	
	SaveMessageLog	
	StackSelector	
	VisualType	
	WindowMgr	
	XWarpPtr	

Command-line options

The following section describes each of the possible command line options that can be used with ConvexAVS.

-appsfile *file_specification* ***(ApplicationsFile)***

Use this option to specify the location of the AVS.applns file. This file specifies the buttons that appear in the Applications menu. Refer to "Adding to the Applications menu," on page 48 for more information.

-class *string*

This is the command-line option equivalent of the DISPLAYCLASS environment variable. You can use it to make ConvexAVS behave in different ways when it is started from different types of display hardware. This option has two effects:

- An Xdefaults file specifies the look of the ConvexAVS interface; what shades of grey are used for command buttons, what fonts to use, the background, and so on. When `-class` *string* is given, ConvexAVS does not use the default avs.Xdefaults file, located in the /usr/avs/runtime directory. Instead, it looks for an Xdefaults.*string* file in the /usr/avs/runtime directory and uses it. You can specify an alternate location by setting the AVSXDEFAULTS environment variable.
- ConvexAVS looks for a configuration file in the following locations in the following order:

```
./.avsrc.<class>
./.avsrc$HOME/.avsrc.<class>
$HOME/.avsrc
/usr/avs/runtime/avsrc.<class>
/usr/avs/runtime/avsrc
```

Refer to "Setting Xdefaults for ConvexAVS," on page 49, for more information about the contents of this configuration file.

-cli *(any CLI command)*

Run ConvexAVS with the Command Language Interpreter functioning in the terminal emulator window from which ConvexAVS was invoked. The `cli` argument accepts an optional initial command string, which must be enclosed in quotes, for example:

```
-cli "script -play name.scr"
```

See "CLI scripts," on page 596 for details.

Note

Demo scripts cannot be run from the avs_help browser if the `-cli` option is used.

-data *directory* **(DataDirectory)**

Specifies the directory in which all subsystem data input file browsers, including the Image Viewer, the Graph Viewer, the Geometry Viewer, and the data input modules in the Network Editor, will initially look for data files. This redirects ConvexAVS's default data input directory to your own data files. The default data directory is /usr/avs/data.

-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))***

Automatically invokes the Geometry Viewer subsystem at start-up. Refer to the next section for sub-options to use with this option.

Note

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

-graph

Automatically invokes the ConvexAVS Graph Viewer at system start-up.

-image

Automatically invokes the ConvexAVS Image Viewer at system start-up.

-library *filespec* **(ModuleLibraries)**

Specifies which ConvexAVS module library files to load into the Network Editor at system start-up. Module library files are ASCII files describing sets of modules. The /usr/avs/avs_library/Supported file is an excellent example. The `-library` option (and its start-up file equivalent ModuleLibraries) allows you to load your own sets of modules—either modules you've written yourself or subsets of the supplied modules that you have customized to your needs—instead of relying on the system default module libraries.

It is equivalent to using the Network Editor's **Read Module Library** function.

This option causes ConvexAVS to load only the libraries specified on the command line. Be sure to give the name of a valid module library file; not a directory or a module binary.

-modules *directory or file_name*

Specifies a directory or file in which the ConvexAVS Network Editor will look for executable modules. All executable files in the directory are examined to determine whether they contain one or more modules. Those that do are added to the default module library modules in the Network Editor's module Palette. This option differs from the `-library` option in that it loads binary module files, not ASCII module library files. It is slower to load modules as binary files rather than libraries.

You can use more than one `-modules` option to have ConvexAVS search through multiple directories for modules. This is a tool for loading individual modules (perhaps modules that you are debugging) that you have not yet formalized into a module library. It is equivalent to the Network Editor's Read Module(s) function. It cannot be used to read remote modules.

The default modules directory is /usr/avs/avs_library.

-netdir *directory* **(NetworkDirectory)**

Specifies the directory in which the ConvexAVS Network Editor subsystem initially will look for network files (**Read Network** and **Write Network** functions). Use this tool to redirect ConvexAVS's default network focus from the samples provided in the /usr/avs/networks directory to your own network files.

The default network directory is /usr/avs/networks.

-network *network-file*

Automatically invokes the Network Editor subsystem and loads the specified network file.

-path *directory* **(Path)**

Specifies the directory tree in which ConvexAVS is installed.

The default path is /usr/avs. If you specify another path, then, the default data directory and network directory are modified accordingly. That is, all subdirectories of avs use this path prefix:

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

-separate

This option disables ConvexAVS's multiple modules in one process feature. It forces each module to execute as a separate process, whether or not it is combined in an executable with other modules. The option is primarily useful for debugging. Refer to "Multiple modules in one process," on page 583.

-server

This option opens a connection that an external process can use to connect to ConvexAVS and exchange with it a stream of Command Language Interpreter (CLI) commands and their output. Refer to "Server options," on page 588.

Note

Demo scripts cannot be run from the avs_help browser if the -server option is used.

-shm
-noshm **(SharedMemory)**

These options enable and disable the ConvexAVS shared memory option. When shared memory is on, ConvexAVS keeps only one copy of ConvexAVS field and UCD data that all modules in a network share. GEOM-format data and pixmaps do not use shared memory. This improves performance by saving memory and processor time. `-noshm` can disable shared memory if, for example, ConvexAVS's use of the finite shared memory area is interfering with other applications. Shared memory is on by default.

-size *Xdim x Ydim* **(ScreenSize)**

Specifies size, in pixels, to use for ConvexAVS's virtual display screen size. ConvexAVS automatically resizes its interface to fit into the virtual screen. You could use this to confine ConvexAVS to run within one section of your screen instead of across the whole screen. The upper-left portion of the screen is used. The aspect ratio should be 5 by 4 for proper results. A minimum pixel size of 400 by 400 is allowed.

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

Displays a usage message for ConvexAVS.

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

Displays the ConvexAVS version number:

```
AVS Version: 3 (25.58 CONVEX)
```

-viewer *viewer-file*

Automatically creates a viewer that provides turnkey access to a group of existing networks. The Image and Volume Viewer systems under the **Applications** menu button are implemented in this way. The volume_viewer file in the /usr/avs/applications directory is a sample viewer file. The *viewer-file* name must be enclosed within double quotation marks.

-volume

Automatically invokes the ConvexAVS Volume Viewer application at start-up.

Geometry specific options

You can include the following sub-options that are specific to the Geometry Viewer subsystem immediately after the `-geometry` option.

Note

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

-defaults *file_name*

Specifies a Geometry Viewer defaults file. The format of this file is described in "Defaults file," on page 625.

-dir *path_name*

Specifies *path_name* as the default directory used by the functions Read Object, Save Object, Read Scene, Save Scene, and the Read and Save functions in the Edit Property window.

The default data directory is /usr/avs/data (same as for the rest of ConvexAVS data input file browsers).

-filter *path_name*

Specifies *path_name* as the directory to search for geometry conversion utilities, called *name*`_to_geom`.

The default directory for these programs is /usr/avs/bin.

-geometry *geom_spec*

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

-noroll

Disables auto-roll. Use this option to disable the ability to spin an object in the pixmap display window.

-scene *scene-file*

Automatically loads a scene from disk storage. This option executes the Read Scene function within the Geometry Viewer, using the file scene-file.scene.

-usage

Displays a list of Geometry Viewer start-up options. ConvexAVS does not recognize normal command line options after it encounters the `-geometry` option.

ConvexAVS .avsrc start-up file resource options

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 file , avsrc in the /usr/avs/ runtime directory is included on the ConvexAVS distribution tape. This file contains example settings for each resource. You can specify a different defaults file by using the DISPLAYCLASS environment variable.

Format of the start-up file

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

```
ModuleLibraries     /avs_library/Supported
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 applicable, the corresponding command-line option is listed in parentheses after the keyword. 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:

ApplicationsFile (command-line equivalent: **-appsfile)**

Specifies the file specification for the ConvexAVS applications file (AVS.applns). This file specifies the applications available through the applications menu. Refer to "Adding to the Applications menu," on page 48 for more information.

BoundingBox switch

If `BoundingBox 1` is set, then the Image Viewer and Geometry Viewer will come up with their **Bounding Box** control already turned on. A bounding box is a less compute-intensive style of moving geometric objects and Image Viewer subimages. Instead of moving the object real time, it only moves a wirebox representation of the object. Only when you release the mouse button is the object/subimage rendered at its new location.

CLIinit cli_file_spec

Use this option to specify a file that you wish to source as part of the initialization so that personal variables can be set (using set_var). Use the `-cli` option on the command line to run active CLI scripts. Refer to "The .avsrc file option," on page 589 for more information.

Colors red green blue gray

This option controls how many cells of a system colormap ConvexAVS will attempt to allocate to itself when it starts on a pseudo-color system. **Colors** takes four numeric parameters separated by spaces that specify how many red, green, blue, and gray cells to try to allocate:

```
Colors 6 6 6 26
```

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

Specifies the directory in which the various ConvexAVS data input file browsers used in the subsystems (Image Viewer, Graph Viewer, and Geometry Viewer) and Network Editor modules read data modules (read field, read geometry, and so on) initially will look for data files. This is the main tool to refocus ConvexAVS's data input attention off the sample data files in /usr/avs/data and onto your own data files.

DisplayGeometryWindow *width x height ±x-offset ± y offset*

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

DisplayPixmapWindow *width x height ± x-offset ± y offset*

Controls the default X Window System geometry (size and position) of the **display pixmap** module's window.

ForceXRender (1 or 0)

Force ConvexAVS to always use the X renderer. By default, ConvexAVS uses the supported available renderer. Set this variable to 1 to enable.

Gamma *value*

Used by your display to make the screen lighter or darker. *value* ranges from 1.0 (darker) to 3.0 (lighter) intensities. The default is 1.5. You can also set the AVS_GAMMA environment variable.

GridSize *n*

Controls the size in pixels of the Layout Editor's alignment squares when **Snap to Grid** is switched on. The default is 10.

Hosts *fullfilespec*

Gives the name of a hosts file that lists machines, access methods, and directories of remote modules. It provides a personal override to the system default hosts file in the /usr/avs/runtime directory when you click on the Network Editor's **Read Remote Module(s)** button under Module Tools. See "Finding remote modules: the hosts file," on page 128, for details.

ImageAutomagnify *switch*

Causes the image magnification factor that is selected to be based on the window's size. The default is *off*. This option affects the **display image** module output.

ImageScrollbars

If set to the value off (0), suppresses the adding of scrollbars to display windows that are too small for the image they are currently displaying. (You can always see more of the image simply by dragging it with the mouse.)

ModuleLibraries *filespec filespec ...* **(-library)**

Specifies which libraries of modules will be loaded into the Network Editor's module palette.

The last module library listed is the default library showing in the module palette when you enter the Network Editor. The other module libraries listed can be called up with the Network Editor's **Select Module Library** function under Module Tools. There is no way to continue the list of module libraries to a new line; the list must be on one line. Line length is limited to 255 characters.

Note

The .avsrc or .avsrc.X file **must contain a ModuleLibraries line. If it does not, then the Network Editor constructs its module description from the *binary* files in** /usr/avs/avs_library. **This is extremely slow and can cause timing problems if you are also loading networks, since ConvexAVS may try to read them before it has successfully built the module descriptions.**

ModulePanelHeight *integer*

Controls the proportion of the Network Construction window devoted to the module palette as opposed to the Workspace. Enter an integer value, in pixels, to represent the palette size.

NetworkDirectory (-netdir)

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

NetworkWindow *width x height*

Specifies the X Window system geometry of the Network Construction Window, which includes the Network Editor menu, the Module Palette, and the Workspace in which you construct networks of modules. You may need this if your display is substantially smaller than the usual 1280x1024 pixels. This is not affected by the ScreenSize option. That is, it is assumed to be in pixels for the screen you are using and not the standard 1280x1024 screen size.

NetWriteAllParms *(boolean)*

Saves all parameter values when writing out a network with the Network Editor's **Write Network** button, not just those changed since the network was created. The default is to save only the changed parameters.

Path (-path)

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

PdbDataDir

Specifies the default directory used by the PDB Viewer.

PrintNetwork *command*

The Network Editor's **Print Network** button normally sends output to your default printer. This lets you specify an alternate print command to execute. The output file name is appended to the string you provide. For example:

```
PrintNetwork lpr -Pbeeps tmp.ps
```

ScreenSize *Xdim x Ydim*

Specifies the size of ConvexAVS's virtual display in pixels, confining ConvexAVS to run within this area at the upper-left corner of the screen. A minimum screen size of 400 by 400 pixels is allowed.

SharedMemory *switch* **(-shm/noshm)**

Specifying `SharedMemory` ***off*** turns off ConvexAVS's shared memory feature.

StackSelector *type*

Building very large networks can cause the Network Editor's control panel to overflow, making some of the module buttons difficult to access. Setting the *type* to `choice_browser` displays the module names as a fixed-size scrolling list similar to the file browsers instead of as the default `radio_buttons`.

VisualType *visualtype*

ConvexAVS attempts to choose the best visual for your workstation. Use this option to specify a visual type of either `PseudoColor, Direct,` or `TrueColor`. ConvexAVS searches the X server's visual list until it finds the first visual with the given visual type and uses it. For example:

```
VisualType Pseudocolor
```

You can also specify the visual identification number if your Xserver supports multiple visuals. For example:

```
VisualType VisualID 080064
```

WindowMgr *mgr*

This option ensures that the Network Editor's Layout Editor and the X Window System window manager that you are using work correctly together (window size). The default for this parameter is specified in the avs.Xdefaults file located in the /usr/avs/runtime directory. The currently recognized values are `awm, mwm, twm, uwm, dxwm, and olwm`.

XWarpPtr *(on or off)*

When enabled, this option causes the mouse cursor to be automatically moved (warped) into type-in dialog boxes when they appear. `XWarpPtr` is *off* by default.

Adding to the Applications menu

The items that will appear on the Applications menu are defined in the AVS.applns file in the /usr/avs/runtime directory. You can specify a unique applications file by using the `-appsfile` command-line option with ConvexAVS or specify a different file specification with the ApplicationsFile resource in the avsrc start-up file. The applications file is organized as follows:

```
# ConvexAVS Application File
#
builtin AVS2 Image Viewer
builtin AVS2 Volume Viewer
```

You or your ConvexAVS system administrator can add applications to this menu. An application is a single ConvexAVS network. To add a network to the Applications menu, append a line to the AVS.applns file similar to the following:

```
/usr/user_name/networkfile Application name
```

- The first string specifies an absolute path name to the network file. This file would have been created with the Network Editor, then saved with the **Write Network** button. (You might have subsequently edited this ASCII network file to, for example, remove references to specific input files, or to change the default location of display windows.)
- The remainder of the line is taken as the label for the Applications menu button. You do not need to enclose it in quotes, even though it may contain blank characters.

Each line must not exceed 128 characters. The title should not be more than 32 characters. The usable size of the title depends on the fonts used in the main menu buttons. You can specify the font size in the Xdefaults file.

As you continue to add networks to the AVS.applns file, the buttons on the menu panel are resized to accommodate the new buttons, up to a reasonable limit.

When you click on an application button, ConvexAVS reads in the network file defined for it and puts its network control panel upon the screen, along with any output display windows associated with the network modules.

Setting Xdefaults for ConvexAVS

ConvexAVS uses the values found in the Xdefaults.x and avs.Xdefaults file to define X resources. Only those resources identified in the file can be used.

ConvexAVS Default values (avs.Xdefaults)

The following resources can be changed. The values listed below reflect the defaults set for the resource.

User interface

The following resource controls the maximum intensity for labels:

```
avs.lui.labelbrightcolor: #ffffff
```

The next resources affect the contrasting intensities for labels. Values are between 0.0 and 1.0

```
avs.lui.labeldarkvalue:    0.25
avs.lui.labelbrightvalue:  0.75
avs.lui.labelhighvalue:    1.0
```

The following resources control the range of the grey shades used by AVS in LUI widgets. Values are between 0.0 and 1.0.

```
avs.lui.darkgreyvalue:     0.20
avs.lui.lightgreyvalue:    0.80
```

The following resources define the main and text fonts (LUI_TextBrowser):

```
avs.lui.mainfont: -adobe-helvetica-bold-r*iso8859-1
avs.lui.textbrowserfont: -adobe-courier-bold-r*iso8859-1
```

The asterisks (*) should be used to allow AVS to scale fonts over different screen sizes.

The following resources define the colors used for buttons or other widgets being on and off:

```
avs.ui.buttonoff:          #404040
avs.ui.buttonon:           #ffffff
```

The following resources define the background colors used:

```
avs.ui.back:               #646464
avs.ui.page:               #282828
avs.ui.backcolor:          #202020
```

The following resource defines the color of the edit border during layout editing. It should remain fixed at some shade of red.

```
avs.ui.editborder:                    #ff0000
```

The following resource enables or disables the use of the stipple pattern. The stippled background of all pixmaps are overriden with the standard background color.

```
avs.ui.usestipple: on
```

Network Editor colors

```
avs.neted.selection:                  #686868
avs.neted.backcolor:                  #484848
```

Main menu colors (PTUI)

The following resources define the colors used for the main menu buttons. The text color is the off color, and off/push/run are the colors used to replace the pixmaps.

```
avs.main.buttontextcolor:             #a5a5a5
avs.main.buttonoff:                   #282828
avs.main.buttonpush:                  #c80000
avs.main.buttonrun:                   #000080
```

The following resource controls the overall hue of the grey shades. A value of -1.0 means shades of black and white, or 0 - 360 as a hue scale. This value should be set in coordination with other main colors such as ui.back above.

```
avs.main.defaulthue: -1.0
```

The following resource defines the font used for the main menu buttons:

```
avs.main.buttonfont: -adobe-helvetica-bold-r*iso8859-1
```

The following resources define the ConvexAVS message warning borders and primary colors. These are used for the message borders, and some other places where primaries are used. They should remain as some shade of grey, yellow, red and black respectively to align with the documentation.

```
avs.message.info:                     #00c800
avs.message.warning:                  #c8c800
avs.message.error:                    #c80000
avs.message.fatal:                    #000000
```

Xdefaults file (Xdefaults.X)

This section defines the available resources that can be set in the Xdefaults.X file for ConvexAVS. The values used in the examples represent the built-in defaults.

Network Editor colors

The following resources define group selection and main workspace colors:

```
avs.neted.selection:        #787878
avs.neted.backcolor:        #606060
```

User interface

The following resources determine the button on/off colors, basic background shades, and so on:

```
avs.ui.buttonoff:           #585858
avs.ui.buttonon:            #ffffff
avs.ui.back:                #747474
avs.ui.page:                #505050
avs.ui.backcolor:           #484848
avs.ui.editborder:          #ff0000
```

The following resource controls the use of stipple patterns for backgrounds:

```
avs.ui.usestipple: off
```

The following resource controls the color of label text on buttons when highlighted:

```
avs.lui.labelbrightcolor: #ffffff
```

Label colors are based on a range of shades of gray. These values control the subrange within 0.0 (black) to 1.0 (white).

```
avs.lui.labeldarkvalue:    0.35
avs.lui.labelbrightvalue:  0.80
avs.lui.labelhighvalue:    1.0
avs.lui.darkgreyvalue:     0.35
avs.lui.lightgreyvalue:    0.85
```

The following resource controls the color of directory entries in a browser (shade of red):

```
avs.lui.browserdircolor:   #A00000
```

The following resources control the fonts used in most contexts. The text font is usually a fixed-width font.

```
avs.lui.mainfont: -adobe-helvetica-bold-r*iso8859-1
avs.lui.textbrowserfont: -adobe-courier-bold-r*iso8859-1
```

Main menu controls

The following resources control the text color, off/pushed/running values, and font used for the entry menu with the application buttons.

```
avs.main.buttontextcolor:          #c0c0c0
avs.main.buttonoff:                #585858
avs.main.buttonpush:               #c80000
avs.main.buttonrun:                #000080
avs.main.defaulthue:               -1.0
avs.main.buttonfont: -adobe-helvetica-bold-r*iso8859-1
```

The following resources control the `AVS message` warning borders—info (green), warning (yellow), error (red) and fatal (black).

```
avs.message.info:                  #00c800
avs.message.warning:               #c8c800
avs.message.error:                 #c80000
avs.message.fatal:                 #000000
```

Window manager

The window manager resource is used for recording module parameter widget layouts to correct inconsistencies in widget location between different window managers. This resource can also be modified by the WindowMgr option in the avsrc configuration file or `through the CLI window_mgr` command.

```
avs.ui.windowmgr: mwm
```

Main menu

When you start ConvexAVS, the main menu, shown on the left in Figure 23, appears along the left edge of the screen.

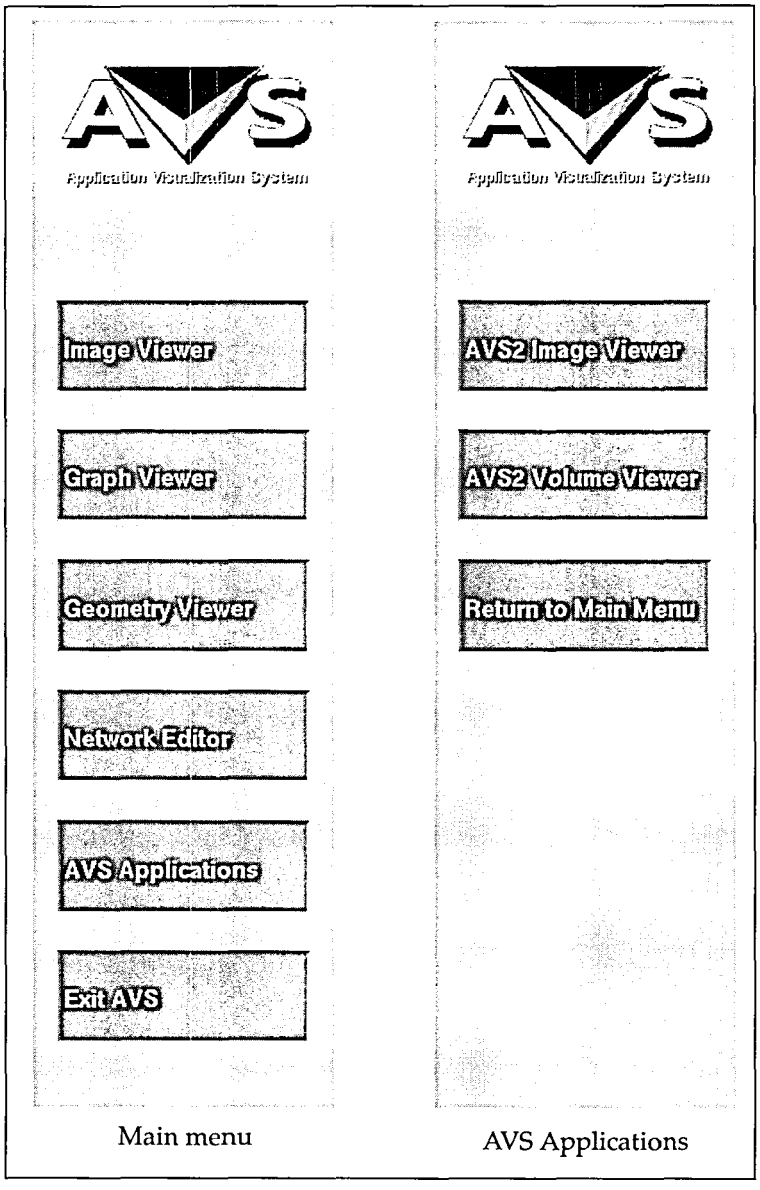


Figure 23
ConvexAVS Main and AVS Applications menus

Each subsystem 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.

The ConvexAVS **Applications** button produces an additional menu of ConvexAVS applications as shown on the right in Figure 23. The list will vary depending upon what applications are installed with your system. The standard release of ConvexAVS includes two applications, the *AVS 2 Image Viewer* and the *AVS 2 Volume Viewer.* These applications are primarily illustrative—each demonstrates a collection of pre-constructed ConvexAVS networks that show various image processing and volume visualization techniques. They are a good way to try out ConvexAVS before attempting to use the Network Editor to construct your own networks.

You can add applications to this Applications menu as required for your system. See "Adding to the Applications menu," on page 48 for more information.

Subsystem control panels

Each of the subsystems has its own control panel (usually, along the left edge of the screen). The control panel is made up of a series of buttons that invoke various subsystem functions. Click with any mouse button to select any control panel function.

When you click the **Close** button at the top of a subsystem's control panel, the control panel is unmapped from the screen, rather as though you had iconified the control panel (but without the icon). If you re-enter the subsystem at a later time, its control panel reappears in the same state that you left it. Once a subsystem is initialized, it remains available for use at any time. The associated control panels can be opened and closed as required.

The exception to this is the Network Editor. It has a true **Exit** button. You must save your work before leaving the Network Editor with functions such as the **Write Network** button.

Subsystem control panels are like any other window on the screen. You can move and resize them. Often, you want more than one control panel on the screen at a time (for example, both the Network Editor control panel and the Geometry Viewer control panel). Use your window manager to move the control panels to more convenient locations.

If all subsystem control panels are closed, or if they are moved away from their original location, the ConvexAVS Main menu reappears.

Switching among the subsystems

You can switch from any main menu subsystem to any other subsystem. At the top of each subsystem's control panel is a **Data Viewers** buttons, shown in Figure 24. Press and hold down any mouse button over this button. A pop-up menu appears listing the other subsystems. Drag the mouse cursor down to the subsystem you want and release the mouse button. Its control panel appear.

Figure 24
Data Viewers button

You are not switching among subsystems so much as you are causing control panels to be mapped and unmapped from the screen. When a subsystem's control panel reappears, it is always the same position on the screen where it was closed.

Each subsystem has only one control panel associated with it. Selecting the Graph Viewer three times does not produce three Graph Viewer control panels; it only maps the same panel three times in succession.

The exception to this is the Network Editor. To get to the Network Editor, you must press the main menu's **Network Editor** button. This may involve closing other control panels or moving them with the window manager so that the original ConvexAVS main menu is no longer obscured.

As noted above, the Network Editor is also the only subsystem with an **Exit** button that deletes the current state of your Network Editor work. The Network Editor asks you to confirm that you want your work deleted before it exits, giving you a chance to save your work.

Canceling operations

In general, ConvexAVS has no cancel function. If you press a button, or make a manipulation in the Geometry Viewer, Image Viewer, or Graph Viewer, then you must wait for the results. While you are waiting, *do not* click on other buttons or try to use the mouse buttons to move objects around just to see if the interface is alive. ConvexAVS will queue these operations.

The one exception is in the Network Editor. You can remove a running module by dragging its icon to the Hammer icon in the lower right corner with the left mouse. The module's process will exit.

Some modules cannot be removed. See "Canceling an operation," on page 95, for more information.

Be aware of the size of your data and the computational implications of operations you request. Many ConvexAVS modules and subsystems will warn you if an operation is going to take a long time and give you the chance to change your mind. You can use filter modules such as **crop** and **downsize** in ConvexAVS networks to reduce the size of field data sets.

Using online help

At all times during a ConvexAVS session, online help is available. Help takes several forms.

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

Finding a help topic

Colored text in the help window indicates a link to a related topic. Click on the colored text to display the topic. Note that some topics might not be available if your system administrator chose not to install them.

Choose an item from the Topic menu to display its topic:

Go Back	Display the most recent topic you visited.
AVS Help	Display the top of the ConvexAVS online help system.
Module List	Display a list of all standard ConvexAVS modules, with a one-line description and a link to the reference page.
Help	General online help browser usage information.

Closing the help window

Choose **Close** from the Window menu to close the help window.

Getting documentation for a ConvexAVS module

Using the right mouse button, click on the small square (dimple) in any module icon to open its Module Editor window. Then click the **Show Module Documentation** button to view the complete reference page for the module.

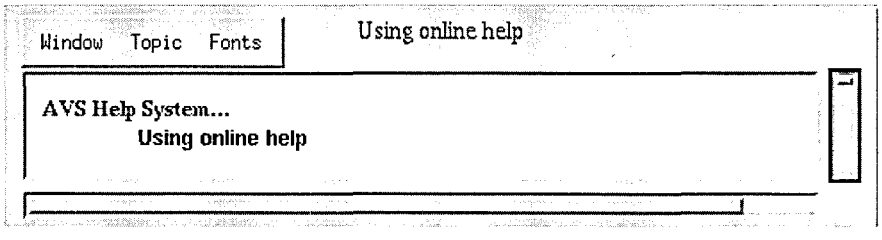
Getting help for a subsystem

In the control panels for the ConvexAVS subsystems (Image Viewer, Geometry Viewer, and so on.), click the **Help** button. The help window will open and display a short description and a list of topics with more information.

Help text browser

The ConvexAVS help browser provides online help and tutorials.

Figure 25
Help browser



Using help from the shell command line

The ConvexAVS help system is available from the shell command line as:

```
% avs_help
```

You can enter the name of a module reference or subsystem you are interested as the first argument:

```
% avs_help arbitrary slicer
```

or

```
% avs_help Image Viewer
```

Opening the Help Demos script browser

The **Window** option includes a **Help Demos** option. Select this option to pop up a scrolling browser of script files, shown in Figure 26.

If you invoke avs_help from the shell rather than from a ConvexAVS help button, this menu option is disabled.

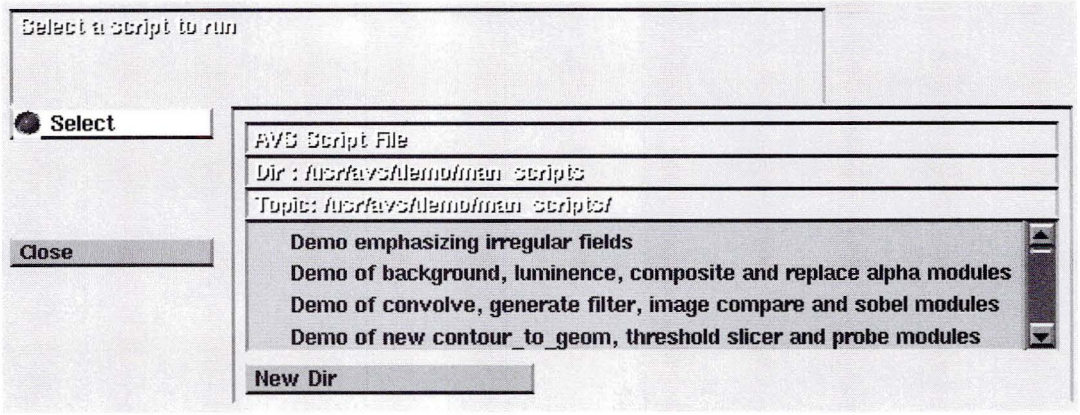
These files, written in the Command Language Interpreter (CLI) language, automatically execute parts of ConvexAVS. They can be used to dynamically illustrate the interface and visualization modules.

Clicking on a script causes it to bring up the Network Editor, load a sample network, and run some interesting data and parameter settings through it. When the script is finished, it leaves the network in the Network Editor so that you can experiment with it yourself. The next script read in will clear the previous network. You must remove the last network manually, either by hitting **Clear Network**, or by simply exiting the Network Editor.

While a script is running, you may cause it to **Pause**, **Continue**, or **Abort**.

Other demonstration scripts are installed in the /usr/avs/demo/image_viewer and /usr/avs/demo/examples directories. The example scripts require that the modules in /usr/avs/examples that they illustrate must have first been compiled. The README file in the /usr/avs/examples directory has instructions for doing this.

Figure 26
Script demo controller



By default, the Help Demo browser comes up showing the sample module and network scripts in the /usr/avs/demo/man_scripts directory. Most of these are constructed from the example networks shown in the module reference pages.

Scrolling through a topic

If the text of a topic is too large to fit in the help window, scrollbars will appear. You can drag the scrollbar's thumb to display more of the text, or you can click outside the thumb. Clicking outside the thumb with the left mouse button scrolls one screenfull at a time. Clicking with the middle button moves the thumb to where you click, and scrolls the text accordingly.

Resizing the help window

If you want to use more or less of your screen to display a help topic, use your window manager to resize the help window. The text will flow to fit the size of the window unless you make it too narrow. In that case, a horizontal scrollbar appears.

Changing fonts

If you would like the text to be larger or smaller, try the **Small** or **Large** items in the Fonts menu. If you have another font you prefer, choose **Other...** and enter the name of the font you like. Use the `xlsfonts` shell command to see what fonts your display has.

Note

You will lose the distinction between plain text, bold, italics, and sans serif fonts if you use your own fonts.

Avoiding delays displaying the help window

The first time you bring up the help window in your ConvexAVS session, there is a delay while your X display transfers font size information to the help system. You can avoid this delay by overriding the X resource that controls the selection of fonts in the help window. For example, add the following line to your .Xdefaults file to use only the fixed font:

```
Avs_help*text.fontList: fixed
```

To use the scroll bars present on all the Help browsers:

- The left mouse button scrolls upward.
- The effect of the middle button depends on exactly where the cursor is:
  - **In the arrow box at the top**. Click to scroll to the very top of the help text.
  - **In the elevator shaft**. Click and hold down the button to grab the elevator bar. Moving the bar up or down causes the help text to scroll accordingly.
  - **In the arrow box at the bottom**. Click to scroll to the very bottom of the help text.
- The right mouse button scrolls downward.

You can change the size of the viewing area by using the X window manager to make the entire Help window larger or smaller. You can also move the window using the window manager.

When you're done, click the **Close** button to close the Help panel window.

Red entries in a help browser indicate subdirectories that contain additional help screens. You'll often see the red entry `..(help)` at the top of the Help Topic Browser list. This indicates the parent directory, /usr/avs/runtime/help, which contains a group of help screens that provide overall ConvexAVS orientation.

Module editor

Each computational module in ConvexAVS is represented onscreen 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.

File browsers and dialog type-in panels

In addition to the subsystem control panels, ConvexAVS uses a number of different kinds of interaction *widgets*. By far the most common are various *browsers*. These are new windows that pop-up on the screen when you press a control panel button, showing you a selection of choices, like the Help Browsers described above. If the length of the list exceeds the size of the browser window, you can use the scroll bars at the right of the browser to see all the choices. Most browsers remain on the screen until you explicitly remove them by pressing their **Close** button.

The most common browser is the file browser, shown in Figure 27. File browsers are associated with each subsystem's Read function (for example, **Read Object, Read Image, Read AVS Plot File**). They also appear on all of the read modules.

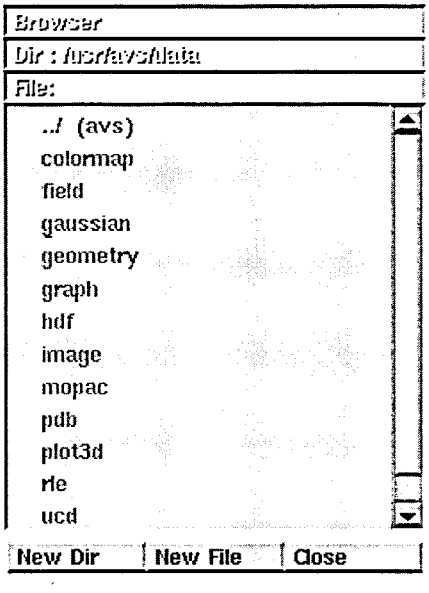


Figure 27
File browser widget

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:

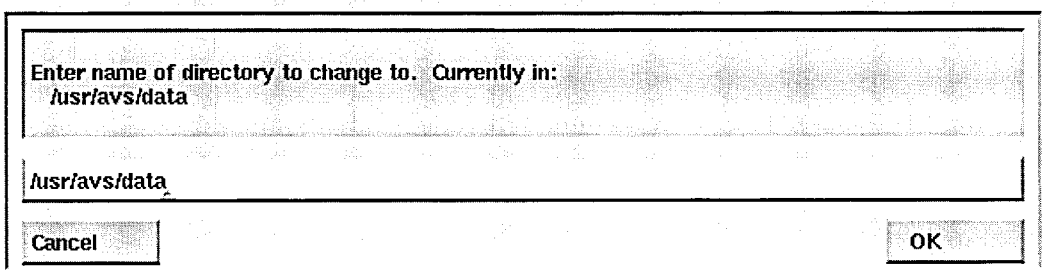
- The left mouse button scrolls upward.
- The effect of the middle button depends on exactly where the cursor is:
  - **In the arrow box at the top.** Click to scroll the list to the very top.
  - **In the elevator shaft.** Click and hold down the button to grab the elevator bar. Moving the bar up or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

A file browser has these buttons at the bottom:

New Dir

Pops up a dialog type-in panel, as shown in Figure 27, in which you can type the name of another directory (full path name or path relative to the current directory). File browsers default to showing you the contents of the /usr/avs/data directory. You can change this with the DataDirectory .avsrc file option.

Figure 28
Directory and file name dialog



Be sure the mouse cursor is within the dialog type-in 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 type-in box, or press the **RETURN** key, or move the cursor outside the panel, the directory whose name you've typed becomes current, and its file names are displayed in the browser window.

Should you inadvertently give a file name, it will select that file name as though you had used **New File** *and* select the directory it is in.

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. Should you give a directory rather than a file name, the directory is changed.

Close (not always present)

This removes the file browser widget from the screen.

You normally have to manually move your mouse cursor into a dialog type-in panel when it appears. This follows X Window System interface conventions which say that a program should never warp a mouse cursor for a user. You can override this and have ConvexAVS move the cursor automatically into dialog type-in panels and other dialog widgets that require a response by setting XWarpPtr on in your .avsrc start-up file.

The other ConvexAVS control widgets, most of which are associated with ConvexAVS modules, are discussed in "Using control widgets," on page 97.

Exiting ConvexAVS

To leave ConvexAVS, bring back the ConvexAVS Main menu. This may involve moving and/or closing subsystem control panels or exiting the Network Editor. Then press **Exit AVS**. ConvexAVS brings up a dialog box, asking you to confirm that you wish to exit.

ConvexAVS does not save the state from one session to another. However, the Image Viewer, Geometry Viewer, and the Network Editor each have commands that perform a similar function. In the Image Viewer, you can press **Save Scene** under the Views menu; in the Geometry Viewer, you can **Save Scene** under the Cameras menu; and in the Network Editor, you can **Write Network** under the Network Tools menu. Each of these uses the ConvexAVS Command Language Interpreter (CLI) to save a snapshot of the current state of an individual image or geometry scene, or ConvexAVS network that you can read in the next time you use ConvexAVS. Check with each subsystem's chapter to find out what each saves.

Running on remote servers

The following section includes special considerations for running on remote servers.

Image Viewer

The main difference between the Image Viewer running on a true color system versus running on a pseudo color X server is the appearance of the images.

On 24-plane true color systems, each pixel can have one of (2**8)**3 (16,777,216) color values. There are 8 bits to represent red tones, 8 bits for green tones, and 8 bits for blue tones. The red, green, and blue tones combine to create the actual pixel color. The sample image files in /usr/avs/data/image are all true color images.

On 8-plane, pseudocolor systems, in ConvexAVS each pixel can normally have one of 216 color values. There are 6 red tones, 6 green tones, and 6 blue tones. To display a true color image on an 8-plane pseudo color device, ConvexAVS takes the original red value for each pixel and finds the closest numeric value from among the 6 reds available. It does the same for green and blue. The final pixel color is the combination of these three best-matches. This might sound very limited, but the end result is surprisingly satisfactory.

Pseudo color and true color displays with more planes use the same method to produce a closer approximation to the original 24-plane true color representation.

The Image Viewer *dithers* its display images when they are displayed. This means that it uses an algorithm to trade apparent image resolution for objective color accuracy. This reduces the banding you see when a colors change shade slowly over a wide screen area and produces a closer approximation to a true color image.

The **hq display image** module displays an image in a window on an 8-bit-plane pseudo-color X terminal. It performs color quantization on the image and chooses the best set (one that has the minimum sum-of-squares error) of colors for the colormap.

In some circumstances, the **display image** module can produce a better version of an image than the Image Viewer display can.

Network Editor subsystem 3

The ConvexAVS Network Editor subsystem is a visual programming interface for creating, testing, and revising ConvexAVS networks. It supports these major features:

- You can save the visualization networks that you create as a visualization application.
- You can use the Network Editor's Layout Editor to redesign the user interface to a network, so that others can perform visualization tasks without having to be knowledgeable about network construction.
- The applications can be made accessible from the main menu's Applications submenu. The mechanism for doing this is described in "Adding to the Applications menu," on page 48.
- Networks can contain a mixture of modules that execute locally, and modules that execute on a remote host that runs ConvexAVS. The remote host can be heterogeneous, that is, of a different hardware type than your workstation.
- You can interactively create libraries of modules to support an individualized repertoire of visualization functions.

General information

ConvexAVS provides primary access to the Network Editor functions through the main menu.

The Network Editor can be started by the ConvexAVS Command Language Interpreter (CLI) in three ways:

- By typing Network Editor CLI commands to the CLI prompt
- By reading a CLI script file
- From a user-written module that sends CLI commands to the Network Editor via the ConvexAVS kernel

The Network Editor supports a journaling feature for capturing CLI commands. You can record your Network Editor interactions into an ASCII file, which you can then edit to produce a demonstration script that can be replayed. This is how the Help Demo scripts were created. Chapter 17, "Command Line Interpreter" provides more information on the CLI and how to create scripts.

Starting the Network Editor

To start the Network Editor, 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 **Close** 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, as illustrated in Figure 29.

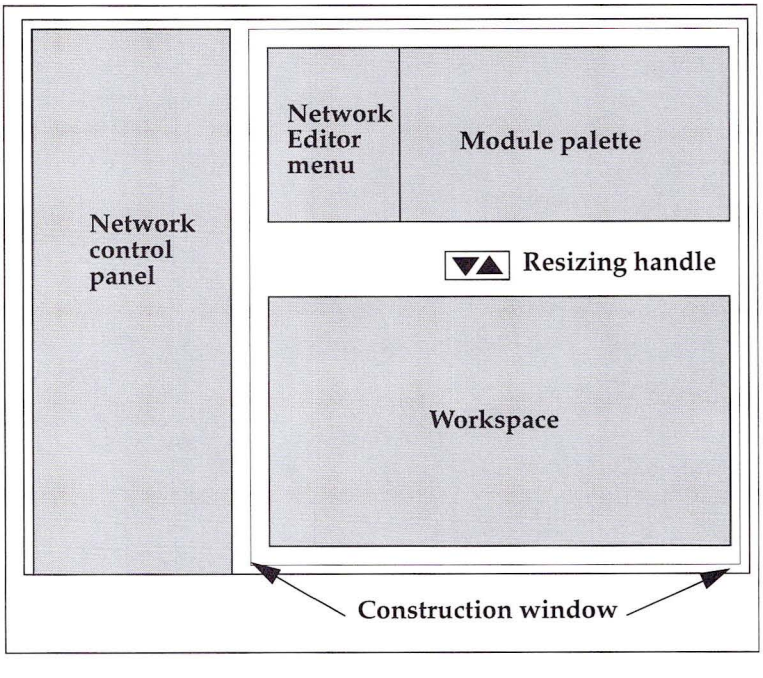


Figure 29
Network Control panel and construction window

Getting help

Online help for the Network Editor is available by clicking on the **Help** buttons or by using the **Show module documentation** button in the Module Editor. Module reference help can also be accessed through the command line by running the `avs_help` command.

The **Help** button in the Network construction window brings up a Help browser.

Help demos

Clicking on the **Help Demos** option in the Windows menu produces a browser of automatic scripts that illustrate various visualization modules performing in networks. The scripts load a sample network, and run sample data and parameter settings through it. When the script is finished, it leaves the network in the Network Editor so that you can experiment with it yourself. The next script read in will clear the previous network.

While the script runs, you can cause it to **Pause**, **Continue**, or **Abort**.

Remove the last network manually, either by clicking on **Clear Network** on the Network Editor's panel, or by simply exiting the Network Editor.

See "Using online help," on page 56 for details on using help.

Closing the Network Editor

The Network Editor has two top-level windows that can be closed separately:

- Click the **Exit** button at the top of the Network 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 subsystems

In many situations, you'll want to switch to the other ConvexAVS subsystems 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 with the Geometry Viewer. There are two ways to go directly from the Network Editor to the other subsystems:

- Position the mouse cursor over the **Data Viewers** button at the top of the Network Control panel. Press *and hold down* any mouse button. A pop-up menu appears listing the other subsystems. Roll the mouse cursor down to the subsystem you wish to start, then release the mouse button. The control panel for the selected subsystem will appear on top of the Network Control panel. You can return to the Network Editor by clicking the **Close** button at the top of each subsystem's control panel, or simply use your window manager to move the control panel to another part of the screen.
- If a network includes the **render geometry**, **image viewer**, or **graph viewer** modules, use the left mouse button to click the small square box in the icon for the module.

Status widget

At the top of the Network Control panel is a Status widget.

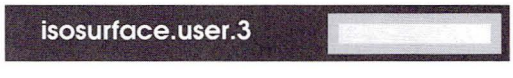


Figure 30
Status widget

This widget shows which module is executing, and gives an approximate idea of its progress. If the widget shows 10%, it means that the module is receiving data. Ninety percent (90%) means it is done processing the data and is outputting the result. Anything in between means that the module is executing. Individual modules can use this Status widget to give more precise statements of progress. The **read field** module does this. Not all modules take advantage of this.

Using the Network Editor

In general, creating a network includes these steps:

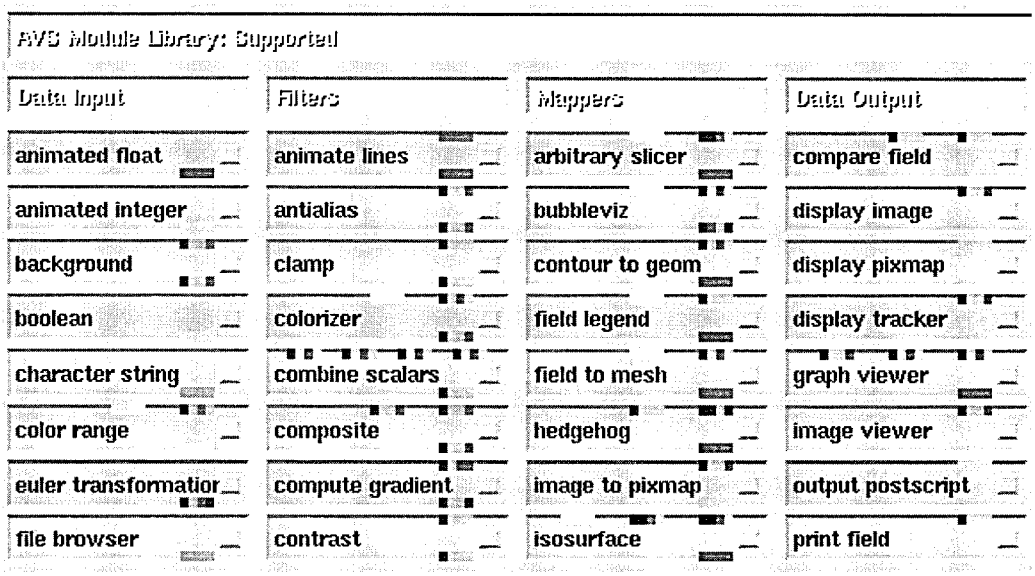
1. Using the mouse to pull modules from the ConvexAVS Module Library palette into the workspace, and reading already-existing networks from disk storage.
2. Using the mouse to connect the modules' input and output ports. The connections define the network by specifying the flow of data among the modules.
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

When you first start ConvexAVS, the Network Editor loads its module palette, shown in Figure 31, with one or more sets of ConvexAVS modules.

Figure 31
Module palette



Loading module libraries

Several things affect which modules are loaded into the module palette:

- If you *don't* have a .avsrc start-up file, then ConvexAVS uses its system-default start-up file in /usr/avs/runtime/avsrc. This start-up file might contain the following line:

```
ModuleLibraries /usr/avs//Unsupported /usr/avs/avs_library/Supported
```

  This instructs ConvexAVS to load two module library files: Unsupported and Supported. A module library is an ASCII file containing lines that describe the names of modules and where to find them. It is used to quickly construct the icons in the module palette and make them known to the ConvexAVS kernel without actually loading the binary module files.

The last module library specified is the set that is showing when you enter the Network Editor; thus the system default .avsrc file lists the Supported library file last.

Supported and unsupported modules are documented in the *ConvexAVS Module Reference*.

Note

If you have a *.avsrc* or *.avsrc.X* file, be sure to put a ModuleLibraries line in it.

- If you have a .avsrc or .avsrc.X file, but you don't have a ModuleLibraries line in it that specifies what libraries ConvexAVS should use, then ConvexAVS individually loads a description for each binary module file in the directory /usr/avs/avs_library. The unsupported modules are not loaded.
- If you place a ModuleLibraries line in your personal .avsrc or .avsrc.X file, then ConvexAVS loads modules from the library files specified. The last library listed shows as the default set in the module palette.

Note

The ModuleLibraries line needs full file specifications. The line length is limited to 255 characters (when specified in the .avsrc file).

- You can supplement any library that would normally be loaded by using the `-library` option on the ConvexAVS command line, or by loading the library interactively with the **Read Module Library** button under the Network Editor's Module Tools menu.
- Use the **Select Module Library** browser from the Module Tools menu to select the module set for the module palette.
- You can create your own module library files. These may contain your own modules, or be a subset of the ConvexAVS supported and unsupported modules containing only the modules you regularly use. This is discussed in "Constructing a module library," on page 132.

Module types

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 (for example, **read volume**) read a data file from disk storage. Other modules (for example, **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, and applying a linear transformation.

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 2-D surface in 3-D space. It does so by interpreting each scalar value of a 2-D array as the height of a point above a base plane. The collection of points defines (an approximation to) a 2-D surface above the plane.

Data output 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.

Module input and output ports

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, as shown in Figure 32. The ports are color-coded to indicate the type of data that can pass through the port.

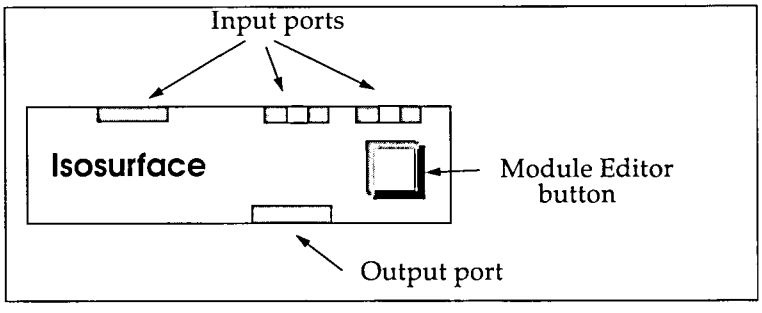


Figure 32
Module icon

You need not memorize the color-coding scheme— ConvexAVS allows you to connect ports only if their data types are compatible. 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 module reference page for the module in a help browser window, as shown in Figure 33.

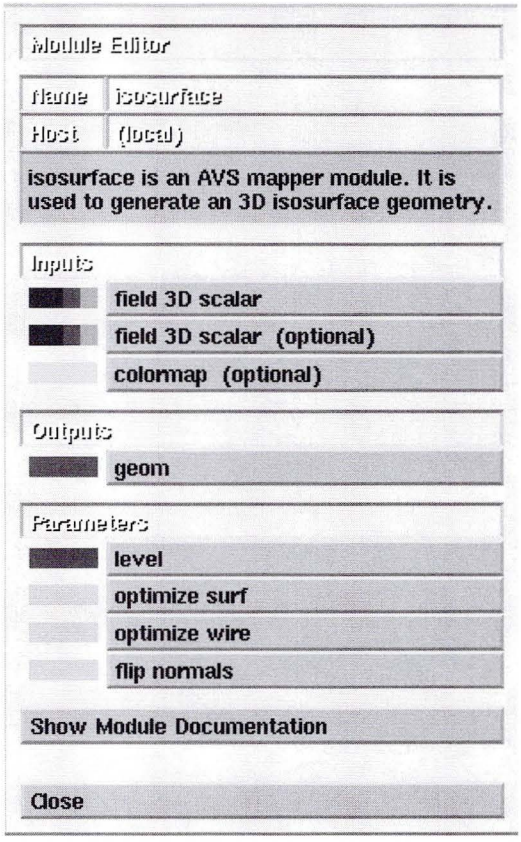


Figure 33
Module Editor

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.

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.

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 raises the module's control panel to the top of the Network Control panel stack.

The Module Editor window also includes these function buttons:

Show Module Documentation

Displays the entire reference page in a help browser window.

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.

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. This would allow you to enter an exact value, such as 48.2, rather than using the mouse to fine-tune a dial setting.

Port color-coding

The following list defines which color are used for each data type:

red = *geometry*

A *geometry* is a geometric description of one or more objects. 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, and spheres, a geometry can include specifications for vertex and surface colors.

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.

Several modules dynamically convert raw data into geometries, such as the **field to mesh** module.

yellow = *colormap*

A *colormap* is a table that converts an integer value to a pixel value (that is, 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 to 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.

orange = *unstructured cell data*

Unstructured cell data (UCD) is commonly used in finite element analysis. For more information, see "Unstructured cell data," on page 385.

Parameter data ports (again, normally invisible) are color-coded by this scheme:

- medium purple = float
- light purple = integer
- grey blue = string
- white = 0 or 1 bit

Boolean and tristate parameters are implemented internally as integers.

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 button: Move one or more icons.

Middle button: Establish a connection between two icons.

Right button: Break an existing connection between two icons.

Finding the module you want

The Network Editor's standard module library includes more than 140 modules. This number is large enough so that you may not immediately see 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 several facilities for handling such situations.

Scrolling a module list

The icons in each category are listed alphabetically. If ConvexAVS cannot simultaneously display them all in the allotted space, it adds a scroll widget to the category's title bar, shown in Figure 34:

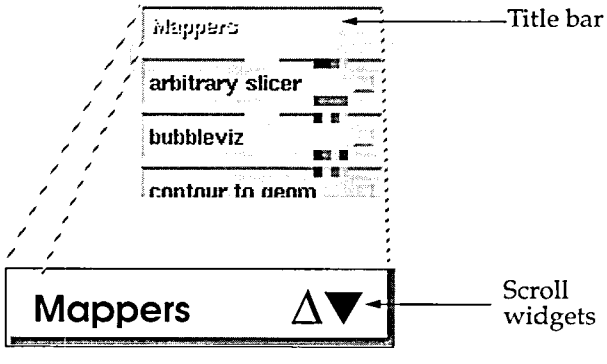


Figure 34
Scroll icon for a module category

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. There are two ways to continue searching:
   - Type more letters 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 "iz".
   - Press **RETURN** to continue the search on the current basis, that is, search for the *next* icon containing the letters you typed.
4. You can repeat the preceding step as many times as you like, either adding characters to the search string, or clicking on **RETURN** to continue the search for the same string.
5. Any time the cursor is in the title bar of a category, you can press **BACKSPACE** to scroll the category back to the top.

There is no need to explicitly end the search. Whenever you're finished searching, just stop typing. If a search string fails to match any module, the list does not scroll.

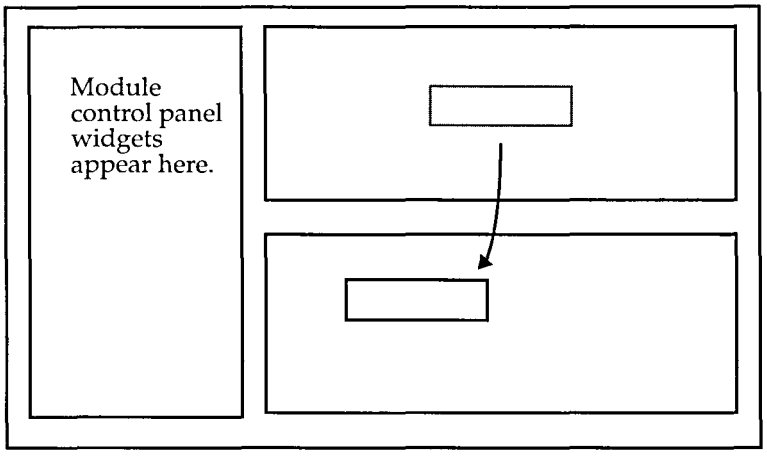
Making the module palette larger

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, shown in Figure 29. 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, then release the button. This changes the partitioning of the Network construction window between the upper area (palette/menu) and the lower area (workspace).

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, as shown in Figure 35.

Figure 35
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. One module control panel is visible at a time. 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, image viewer, and graph viewer modules are the exception. When you drag them into the workspace, their control panel does not appear, and the name is not added to the Network Control panel menu. The control panel for render geometry is the entire Geometry Viewer, Image Viewer, and Graph Viewer subsystems, described in their own chapters.

The appearance of the Network Editor's control panel can be changed. If your networks contain so many modules that the supplied control panel is not long enough to hold all of the radio buttons, you can change the radio buttons into a scrolling browser similar to a file browser (see the StackSelector keyword description in Chapter 2, "Starting ConvexAVS," on page 47. You can also use the Layout Editor to make more module control panels visible simultaneously.

When you click the small square dimple on one of these module icons with the left mouse button, the 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 viewer control panel and the Network Control panel at the same time.

Note

Do not drag a module from the palette directly to the Hammer icon in the workspace in one continuous motion. This deletes the module from the palette. If you do this by mistake, you can get a new copy of the module using the Module Tools submenu's Read Module button.

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.

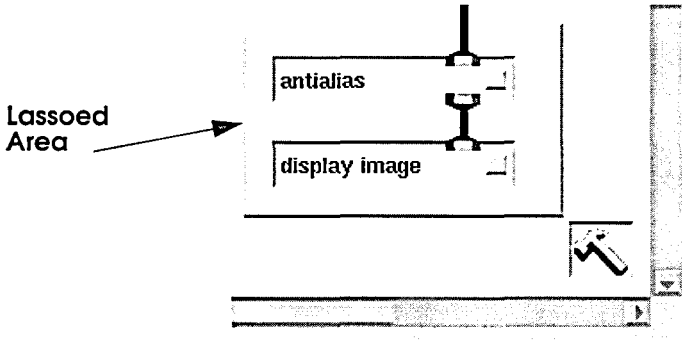
To remove a lasso, click the left mouse button in the background part of the lasso area.

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.

Figure 36 shows two modules within a lassoed area.

Figure 36
Lassoed modules



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 Editor control panel.

Dragging an executing module icon to the Hammer icon causes that module to terminate execution. (However, see the cautions in "Canceling an operation," on page 95).

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 ports, because they introduce new data into a network rather than process data that is already in memory.)

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

The ports are color coded to represent the type of data that can pass through. See Chapter 7, "Data types and import strategies," on page 267 for a full discussion of ConvexAVS data types. An output port can only be connected to an input port with a matching color. The color codes are defined in the section, "Port color-coding," on page 76.

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 (turns it *white*) the connection you want to make, 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.

The details of connecting data, parameter, and upstream data ports are covered in the next section.

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 (drag it to the Hammer icon), all its connections are automatically deleted.

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.

Data, parameter, and upstream data ports

Modules have three kinds of input/output ports. Each type is discussed in the following sections.

Data ports

Modules receive and transmit data through one or more *data ports*. Data includes the following:

- Field, image, volume, or unstructured cell representations of scientific data
- Colormaps that modules use to represent data as colors
- Pixmaps, images, and geometries that are the viewable constructs of data that modules produce

Module data ports are always visible on the module icon.

Parameter ports

Modules can receive and send parameters through one or more *parameter ports*. Parameters are the values generated by module widgets or by a another module that sends the parameter value to other modules through its output port(s). Parameter ports are normally invisible.

Parameter data (integers, floating point values, boolean on/off switches, strings of characters) control some aspect of the module's execution. An integer parameter might determine which slice plane to take through a volume, or which element of a vector to extract and map. Floating point values might control what values should be used to construct an isosurface (3-D contour). Switches can control whether or not to interpolate data. Strings might specify what file a module should read.

The parameters generated by widgets normally just affect the module to which they belong. However, it is possible to generate parameter values and send them to other modules' widgets through a ConvexAVS network connection.

Here are some examples of what you can do with parameter ports:

Single parameter modules

A single module that produces parameter-type output can send the same parameter value simultaneously to multiple modules.

Figure 37 shows an **integer** module connected to two orthogonal slicer modules. The network produces two slices through the field in different planes (I and J) but at the same offset value (set by the integer module's dial widget). The results are converted to geometries and combined together.

Figure 37
Parameter ports connected to the integer module

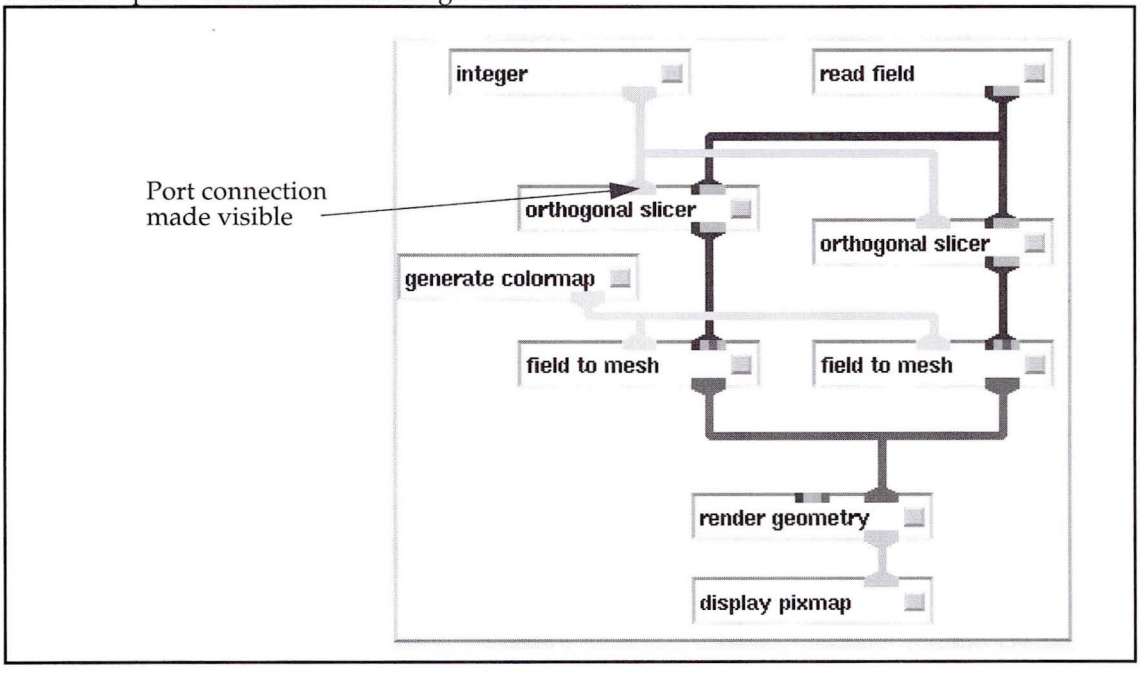


Figure 38 shows the results of using this net to display the hydrogen molecule field.

Figure 38
Orthogonal slices of a hydrogen molecule

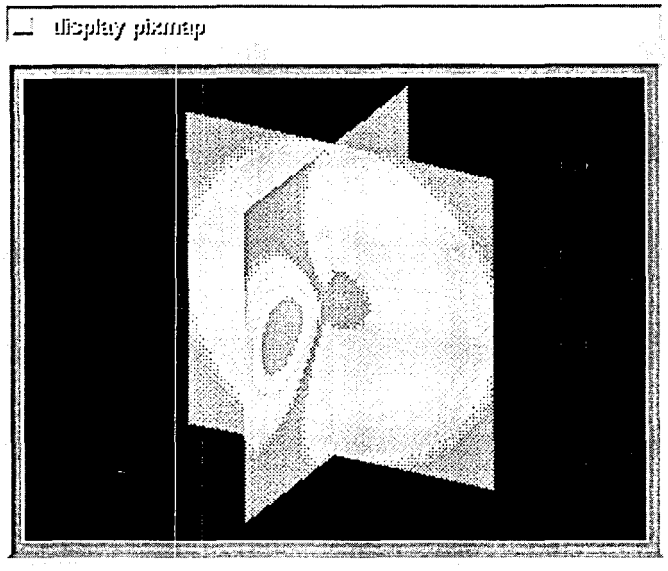
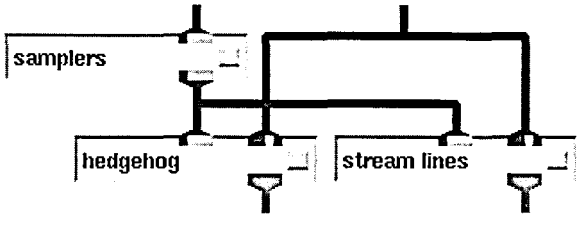


Figure 39 shows a single **samplers** module which causes hedgehog vector arrows and streamlines to appear simultaneously for the same set of sample points.

Figure 39
Simultaneous control of **hedgehog** and **stream lines**



Sending serial parameter changes

Coroutine modules that execute independently from the rest of a network, such as user-written simulations, can send an automated series of parameter values to other modules. For example, one could implement a master clock module for controlling multiple coroutine simulations.

The data coroutine module **animated float** is used in the example shown in Figure 40.

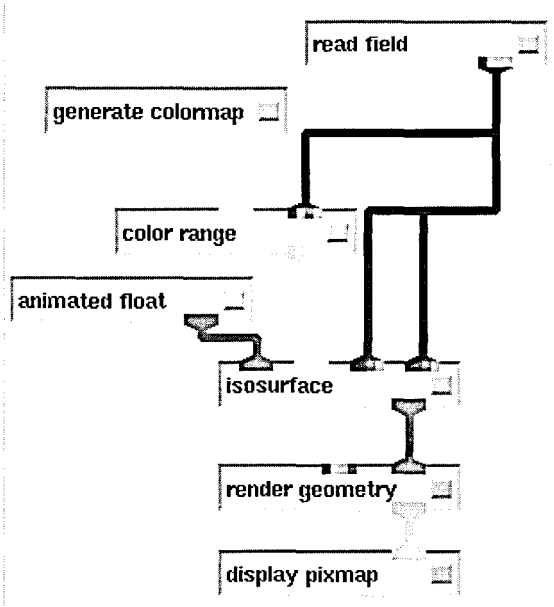


Figure 40
Using **animated float** for upstream control

On this module's control panel, you set minimum, maximum, and step values. When switched on,the **animated_float** module sends a stream of evenly-spaced floating-point values to another module's floating-point parameter port. Used with the **isosurface** module's **level** parameter, this animates a sequence of isosurface contours. One can animate any floating-point port on an ConvexAVS module in this way.

Upstream data ports

Data in an ConvexAVS network normally flows from top to bottom. There is one exception. If two modules can agree on a data structure, then if module B receives input from module A, then module B can also send the agreed-upon data structure back up the network to an input port on module A. This is called *upstream data*. Modules can control the visibility of upstream data ports and the connections between them. Generally, upstream data ports and the connections between them are invisible. The upstream connection between the two modules usually occurs automatically, once any other connection between them has been made. This is also under the control of the modules involved.

Two important uses for this upstream data feedback mechanism are:

- You can have direct mouse manipulation control of objects that highlight data such as slice planes and data probes.
- You can pick structures like chemical bonds by clicking on them.

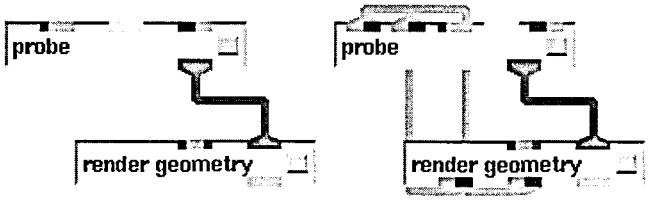
AVS supplies two kinds of feedback data, the *upstream transform* and the *upstream geometry*, to flow from data output modules such as render geometry back up the network to the data mapper modules such as **probe**, and **arbitrary slicer.**

For example, the **probe** module takes a data field as input. It outputs an object to the **render geometry** module (Geometry Viewer) that looks like an electronics probe. The idea is that you use the Geometry Viewer's virtual trackball mouse cursor to move this probe around the volume of data, and the **probe** module tells you what numeric values are present at any given point in 3-D space.

The Geometry Viewer knows where you have moved the mouse cursor. But the Geometry Viewer does not know what the numerical values in the field are at that point.

To produce this information, The Geometry Viewer outputs an upstream transform that specifies where in 3-D space the probe has moved back up the network to an input port on the **probe** module. The probe takes this information, maps it to the correct data cell in its data field, producing a numeric value for the probe's position, which it sends back to the Geometry Viewer for display. The network in Figure 41 shows the hidden port connections.

Figure 41
Upstream data ports made visible



Picking bonds in a molecule is much the same problem. The Geometry Viewer knows where a user has pointed the mouse and clicked, and it knows what geometry vertex is intersected by a ray pointed at the object from the mouse cursor, but it knows nothing about the molecular structure. It sends an upstream geometry reporting the selected vertex back up the network to a molecular mapper module.

This module can translate the vertex to a particular molecular bond, then highlight it, delete it, or whatever, producing a new output geometry to give to the Geometry Viewer. The **probe** module also supports picking in its Pick Geometry mode.

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: 1-D, 2-D, 3-D, and so on. Fields can also differ in the type of data that is specified for each point: scalar byte, 4-D vector of bytes, or scalar float.

The various field sub-types are incompatible. A module that outputs a 2-D field cannot be connected to one that expects to input a 3-D 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 2-D slice from any 3-D field. You can extract one scalar element from a vector field using **extract scalar**; you can assemble a vector field from scalar components 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, as shown in Figure 42 and Table 4.

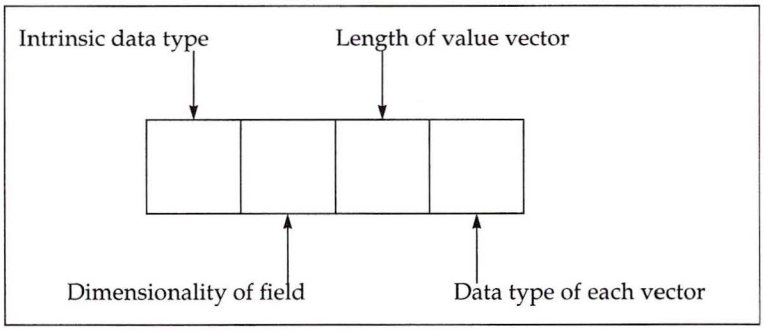


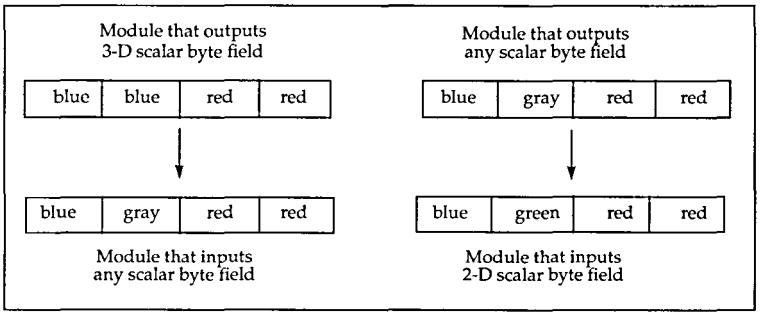
Figure 42
Color-coding for field input/output ports

Table 4
Color-coding for field input/output ports

	Color	Meaning
Intrinsic Data Type	Blue	field
Dimensionality of field	Red Green Blue Gray	1-dimensional 2-dimensional 3-dimensional Any dimensionality
Length of value vector at each point in field	Red Blue Green Gray	1 (value is a scalar quantity) 3 vector (space) 4 vector Any of the above
Data type of each value	Red Green Blue Yellow	Byte Integer Single-precision floating point (real) Double-precision floating point

The color gray is a wildcard; it indicates that the module can 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. As illustrated in Figure 43, the color bars for field ports don't have to match exactly to be candidates for connection.

Figure 43
Gray color-coding as wildcard value



ConvexAVS checks to see that a field-to-field connection is at least plausible. For example, if the sending module (A), outputs an unrestricted field, and the receiving module (B) accepts only uniform scalar fields, it is plausible that A will send B the right type, so the connection is allowed.

If A actually sends B an irregular vector field when the network executes, then ConvexAVS's runtime type checking will catch it and send you a notice of the runtime incompatibility.

Connecting parameter ports

ConvexAVS comes with a set of parameter modules that generate each of the standard parameter data types. They appear in the Supported Module palette in the Data Input column. The parameters modules are as follows:

- Integer (light purple)
- Float (dark purple)
- File browser (grey blue)
- Boolean (light purple)
- Oneshot (white)
- Tristate (light purple)
- Character string (grey blue)

In addition there are two new coroutine modules produce a stream of parameter values that can be used to make automated animations:

- **animated integer**
- **animated float**

One module, **field legend**, takes field and colormap input, but outputs a single floating point parameter. Another parameter module, **euler** outputs a small 2-D field representing a transformation matrix for input to the **tracer** module.

For more information, including networks which show these modules in use, see the *ConvexAVS Module Reference* manual, or call up the online module documentation by clicking on the **Show Module Documentation** button in each module's Module Editor panel. You can also click on the **Help** button at the top of the Network Editor menu. This brings up the Help panel. On the Help panel is a **Help Demos** button. This raises a browser of automatic scripts that you can run that illustrate the use of most key modules in sample networks.

Caution

The amount of type-checking performed by ConvexAVS, both when you construct a network and when you run data through a network, is limited. It is possible to construct networks that will pass ConvexAVS checks, but may still fail when you execute the network.

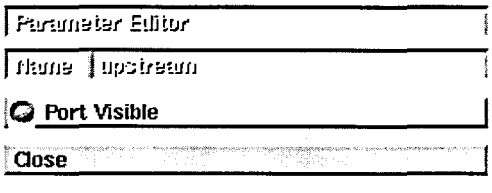
Creating the connection

Because some modules have many parameters, showing all the parameter ports on all the modules in the palette and network would be confusing. Therefore, input parameter ports are invisible by default.

Before you can connect output parameter ports to input parameter ports, you must make the input parameter ports visible.

1. Call up each module's Module Editor window by clicking on the module icon's dimple with the middle or right mouse button.
2. The module's parameters are listed in the Module Editor window. The associated color bar shows the parameter type. Click on the parameter's button to bring up its Parameter Editor panel, shown in Figure 44.

Figure 44
Making a module port visible



3. The **Port Visible** button on the Parameter Editor panel is gray, showing that it is off. Click on this button. A colored parameter port will appear on the module's upper (input) edge.

Note

If you close the main Module Editor panel, it will also close the Parameter Editor panel.

Connect and disconnect the parameter ports in the usual way: middle mouse button to make a connection; right mouse button to disconnect ports.

Connecting upstream data ports

You don't have to do anything to use upstream data. Connections happen automatically and transparently when you construct an ConvexAVS network. When a module with upstream data output ports, such as **render geometry**, has another module connected to one of its input ports, it automatically checks to see if that module has a compatible upstream input port. If it does, it connects its upstream output port to the previous modules upstream input port. This automatic connection only happens between adjacent connected modules.

The following module pairs use upstream data and can be used together:

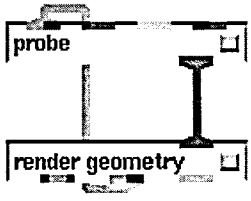
render geom and **arbitrary slicer**
render geom and **hedgehog**
render geom and **probe**
render geom and **stream lines**
render geom and **thresholded slicer**
render geom and **animated lines**
render geom and **samplers**
render geom and **particle advector**
display tracker and **tracer**
display tracker and **gradient shade** (no auto-connect)
hedgehog and **samplers**
stream lines and **samplers**
animate lines and **samplers**

The **display tracker** module does not use either an upstream geometry or an upstream transform. Rather, it passes a 4 by 4 field upstream to control the **tracer** module. Any two modules that can agree on an upstream data type can use the upstream feedback mechanism.

You can make upstream connections visible. Bring up the Module Editor panel for both modules by clicking the middle or right mouse button on their icon dimples. Find the color-coded **struct upstream-transform** or **struct upstream-geom** input or output button. Click on that button to bring up its Parameter Editor panel, then click on **Port Visible**.

When both output and input sides of the connection have been made visible, the upstream flow pipe shows in green, as shown in Figure 45.

Figure 45
probe upstream data paths made visible



You can disconnect upstream connections. However, if you disconnect an upstream transform to a module, the object for that module in the Geometry Viewer is disabled. That is, you will no longer be able to move it with the mouse buttons or Transformation Options panel. You will still be able to move it from the mapper module's widget controls.

A better way to get Geometry Viewer control over the upstream probe or slice plane is to use the **Override** button on the Geometry Viewer's Transformation Options panel.

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

Controlling the execution of a network

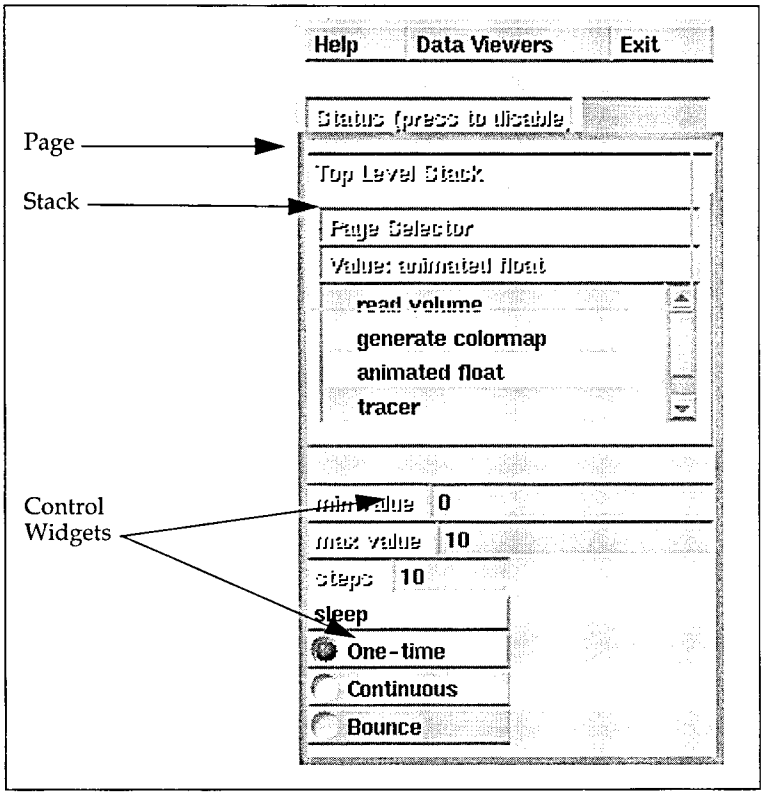
As you build a network, its modules start to execute. In most cases, nothing useful will occur until the network is complete and you specify the input data to be visualized. Thereafter, the network re-executes each time the following actions take place.

- 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 shown in Figure 46.

Figure 46
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.)
- You can change this default layout with the Layout Editor, described in "Working with the Layout Editor," on page 121. If you save a network with a modified layout, the new layout is saved with it.

Canceling an operation

In some situations, you can remove a running module. Use the left mouse button to drag the module to the Hammer icon in the lower right corner of the Workspace. This causes the module's process to exit.

There are some restrictions. You cannot remove built-in modules. These modules are part of the main ConvexAVS process. Also, if the module you want to cancel is part of a set of multiple modules executing as a single process, the ConvexAVS process may hang. More than one module in the set must be part of the network. About one half of the modules, including most of the UCD modules, fall into this category.

The modules in Table 5 should not be removed while they are running.

Table 5
Modules that should not be removed while being executed

Built-in modules	VEX filters	UCD modules	vector modules
generate colormap	histogram stretch	field to ucd	vector div
display image	contrast	read ucd	vector grad
display pixmap	threshold	ucd anno	vector curl
graph viewer	clamp	ucd contour	vector norm
image viewer	transpose	ucd crop	vector mag
render geometry	mirror	ucd extract	extract scalar
write image	downsize	ucd hex to tet	combine scalars
write volume	crop	ucd hog	
colormap manager	interpolate	ucd iso	
image manager	colorizer	ucd legend	
render manager	compute gradient	ucd offset	
	background	ucd probe	
	composite	ucd rslice	
	luminence	ucd slice2D	
	replace alpha	ucd streamline	
	antialias	ucd threshold	
	image compare	ucd to geom	
		ucd tracer	

Module restart option

If a module dies while executing, its module icon will turn black. Modules can die for a variety of reasons, either because of a bug, they are trying to process incompatible data (usually occurs with incompatible dimensions or extents), or the module runs out of memory.

When a module dies, you get a message panel that informs you of the problem and gives you three options:

accept

This means that you accept that the module has died, and that you intend to discard the dead module by dragging it down to the Hammer icon in the workspace. To get a new copy of the module, drag its icon down from the palette and reconnect it.

restart

The module is restarted with its default parameter settings.

restart same

The module is restarted with the same parameter setting. The network will not re-execute until you have changed a parameter setting. This gives you a chance to correct a parameter value that may be killing the module.

There is also a **Restart Module** button on the Module Tools submenu. This restarts a module with its default parameter settings. It is a way to get a fresh instance of a module without hammering the old copy, then instancing a new version from the palette and reconnecting it to the network.

Note

Multiple modules can now be in the same process. If one of these modules dies, all of the modules in the process also die, and turn black. Restart Module will restart all current dead modules in sequence to try to get the same modules running in the same process again.

Using control widgets

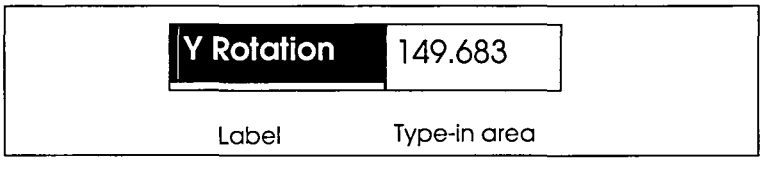
ConvexAVS has a variety of control widgets that allow you to specify module input parameters: integers, floating point numbers, text strings, file names, mutually-exclusive choices, non-mutually-exclusive choices, and colormaps. The following sections describe how to use the various types of control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

Module writers can give control widgets conditional visibility. This means that all of a module's widgets may not be visible at a given time, and only become visible if some condition is satisfied.

Using type-in controls

Figure 47 shows a typical type-in control widget.

Figure 47
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

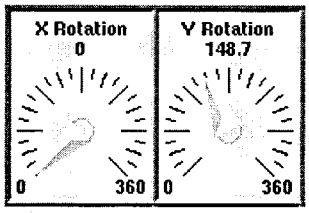
You can also use the X11 ICCCM selection processing techniques to paste input into the type-in.

There are two ways to finish the entry. You can press **RETURN** to finish entering the new value, or you can simply move the mouse cursor out of the type-in area. ConvexAVS checks the new value against the parameter's bounds and type. In some cases, the value you type is converted (for example. a decimal value converted to an integer, or an out-of-bounds value converted to the allowable maximum), and the result of the conversion is displayed.

Using dial controls

Figure 48 shows two typical dial control widgets.

Figure 48
Dial control widgets



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

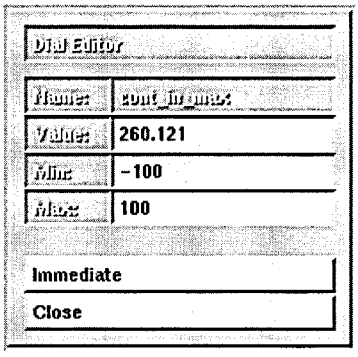
- If you click with any mouse button at a location along the edge of the dial, the needle jumps immediately to that location and the current value indicator changes accordingly.
- Alternatively, click and hold down any mouse button near the needle; then use a circular motion to drag the needle either clockwise or counterclockwise. 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 minimum and maximum bounds will fail—the needle stops moving when you reach the limit, or it snaps back to its limit value.

Dial Editor

Dial control widgets are special in that they have an associated control window called the Dial Editor, shown in Figure 49. 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 49
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.

Streaming data values

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 also causes the image that depends on the parameter to change continuously. (This may not be advisable for compute-intensive networks— changes to the image may lag behind the movement of the needle.) In most modules, **Immediate** is off by default

Some dial widgets are conditionally visible. They may not appear if other modules with overriding parameters are connected to the module.

Dial interaction considerations—Bounds

Using dial widgets is quite intuitive. However, there are some subtleties that become evident when certain data value ranges interact with some modules. These subtleties are explained below. You can probably skip over the remaining paragraphs in this section, and only refer to them if you find your dials behaving in a perplexing manner.

A module can declare a dial to be bounded or unbounded. It will declare it bounded when it is confident that parameter values will almost always fall within a certain range. It will declare it unbounded when it cannot predict a parameter range, or when it wishes the parameter range to be data-dependent (for example, based on the minimum and maximum data values in the input field.) You can check the module's reference page to see how the module defines each of its dial parameters.

Bounded dials—In the Dials figure, resolution is a bounded dial. The title is specified by the module. The module's declared minimum and maximum allowable values for the dial are shown at the lower left and lower right ends of the dial's scale. The current value for the dial appears at the top, beneath the title. The scale (or resolution) of the dial is max-min/270 degrees. As you move the dial needle, the current value is updated correspondingly. You cannot move the needle past the minimum and maximum bounds. It will just stop. You cannot change the dial's minimum or maximum bound values. Bounded dials appear, for example, as the radius multiplier factor for scatter dots spheres, as the number of sample points from which to generate streamlines, or as the rotation of the **arbitrary slicer** plane.

Unbounded dials—Unbounded dials are also common, such as are found in the **distance**, and **high** and **low** threshold in the Dials figure. Again, the title is specified by the module, and the dial's current value is displayed below the title. When an unbounded dial first appears, its indicator needle is always pointing straight up. The scale (or resolution) of the dial is always, initially, once around the dial is 200. A module has no control over the resolution of unbounded dials; there is no function that can normalize the dial resolution to data value range.

In theory, you should be able to increment or decrement an unbounded dial's values indefinitely—just keep turning. However, a module may enforce data-dependent minimum or maximum bounds on unbounded dials. If you move the needle past the software-defined minimum or maximum, it will snap back to the bound value.

Unbounded dials appear, for example, as the isosurface module's level parameter, and as the minimum and maximum threshold values for the **thresholded slicer** module.

Setting specific values— It sometimes happens that the module defines, for example, the minimum field data value as the bounds for one dial, and the maximum field data value as bounds for another dial. If the input field data happens to be floating point values all falling between 0 and 1, and given that the default dial resolution once around is 200, the dials become too sensitive to use. Anything you try to set just snaps back.

In this case, you can use the Dial Editor to either type in specific values, or you can manually change the resolution of the dial from its default minimum and maximum of -100/100 to 0 and 1.

It also helps to hook up **statistics**, **print field**, **generate histogram**, and **graph viewer** module(s) to your network. The **statistics** module displays the field minimum and maximum data values:

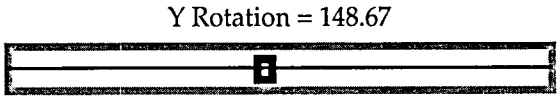
- **print field** reports field minimum and maximum extents.
- **generate histogram** and **graph viewer** plot the distribution of data in a field.

These modules can provide information about the characteristics of your data and can explain dial behavior.

Using slider controls

Figure 50 depicts a typical slider control widget.

Figure 50
Slider control widget



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

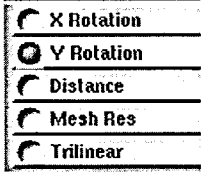
- If you click with any mouse button at a location along the slider, the cross hair jumps immediately to that location and the current value indicator changes accordingly.
- Alternatively, click and hold down any mouse button near the cross hair; then drag it to the left or right. As you do so, the current 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 51 shows a typical choice (radio buttons) control widget, which allows you to select from a mutually-exclusive set of choices.

Figure 51
Set of radio buttons

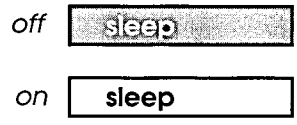


A red ball and highlighting indicates which one of the choices is currently selected. To select another choice, move the cursor anywhere within its box. (The label inside the box lights up to indicate the cursor's presence.) Then, click any mouse button to move the red ball to the new selection.

Using toggle controls

Figure 52 shows a toggle control widget, in both the off state and the on state.

Figure 52
Toggle control widget

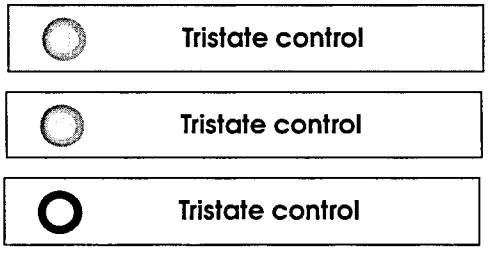


To change the state of a toggle switch, move the cursor anywhere within its box. The label inside the box lights up to indicate the cursor's presence. Then click any mouse button.

Using tristate controls

Figure 53 shows a typical tristate control widget, in its three states. This type of control is used for parameters that can assume three values, not just two.

Figure 53
Tristate 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 54 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.

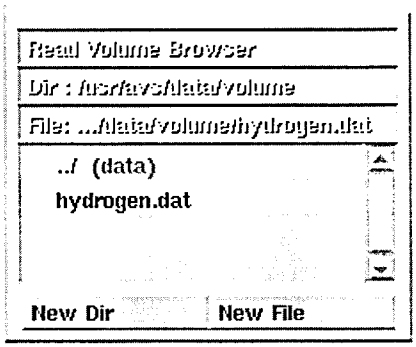
Figure 54
One-shot control widget

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 55 shows a typical file browser control widget.

Figure 55
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

The **New Dir** button pops up a dialog box in which you can type the name of another directory. You can enter either the 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.

Should you inadvertently give a file name, it will select that file name as though you had used **New File** *and* select the directory it is in.

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

The **New File** button 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. Should you give a directory rather than a file name, it will change to the directory.

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.

Other browsers

There are two browser types that serve slightly different purposes.

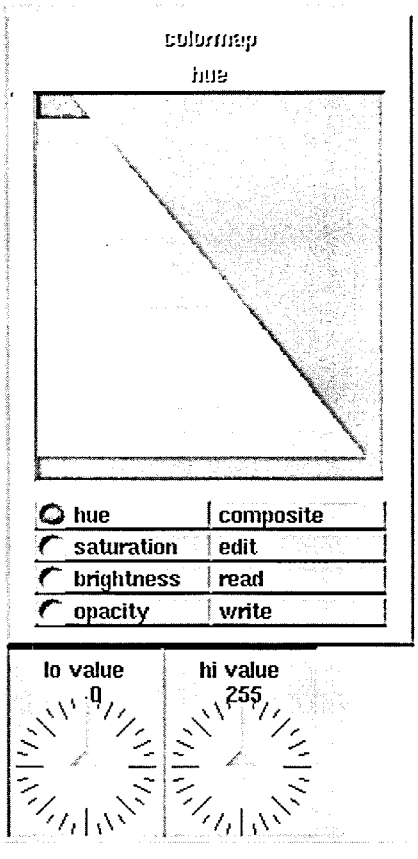
Choice browsers—Choice browsers look just like file browsers, except that the items displayed are not files. The Remote Host Browser described below, for example, allows you to select an alternate host to run modules on. The Help Demo browser is another choice browser.

Text browsers—Text browsers produce scrolling windows of text. They are read-only; you do not select items.

Using the colormap control

Figure 56 shows the control widget that generates a colormap. You can also use it to maintain a set of colormaps in disk files.

Figure 56
Colormap control widget



Translating numbers into colors is an important technique in scientific visualization. The generate colormap module's Colormap Editor widget is the palette you use to mix the colors. 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 is mapped. The next bar specifies both the hue number for line 1 and the hue to which data values of 1 is mapped, and so on.

Initially, the hue numbers form a linear ramp. Smaller numbers is mapped into the blue part of the spectrum; larger numbers are 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:

composite

The hue, saturation, and brightness pages of the Colormap Editor normally show a graphical representation of just the hue, or just the saturation, or just the brightness. Switching on **composite** displays the entire colormap, with colors that accurately reflect the shade that the combined hue, saturation, and brightness values will produce in the rendered image.

For example, with hue toggled, the default colormap page shows lower data values mapping to a medium clear blue. If you then switch to saturation, you could edit the saturation of this blue to be 50%, by drawing a line down the middle of the editor page, producing a pale clear blue. But when you switch back to the hue page, the color still shows as medium blue.

If you toggle **composite**, the true combined hue, saturation, and brightness shade will appear—the actual pale clear blue.

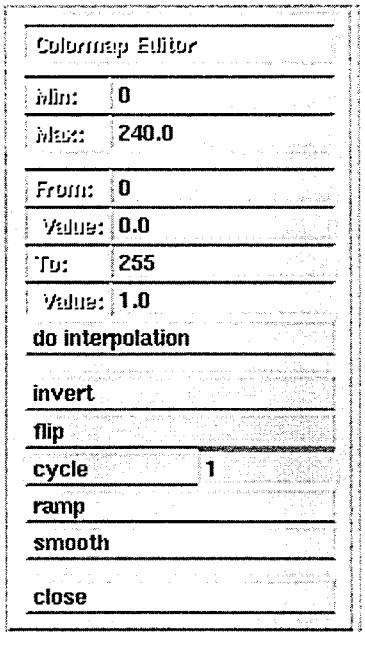
Drawn across the actual colormap is a black line. This black line shows the setting for the hue for that data value (if **hue** is toggled), or for saturation (if **saturation** is toggled), or for the brightness (if **brightness** is toggled).

The composite colormap does not display opacity values; there is nothing behind the colormap to obscure or reveal.

Edit panel

The **edit** button raises another control panel with additional colormap editing features, shown in Figure 57.

Figure 57
Edit panel



Minimum and maximum ranges

The colormap editor uses the HSV color model. Hue is represented as a circle. The default ConvexAVS colormap produces hues from 0 to 240 degrees around this 360 degree circle; that is, from red to blue. One third of the hue spectrum (the magentas and blue-reds from 240 degrees to 359 degrees) is normally excluded from the colormaps. The **Min** and **Max** type-in controls let you set the beginning and endpoints of the spectrum of hues in the colormap. A minimum and maximum range of 100 to 140 would produce 256 fine distinctions in greens. This range applies only to hues.

From/Value and To/Value

These type-in controls give you precise numeric control over which colormap slots map to which explicit hues. Rather than drawing curves in the editor window freehand with a mouse, you type beginning and ending slots in the colormap, and the range of hues, saturation, or brightness that is associated with them.

To make the type-ins take effect, click on **do interpolation**.

For example, you have a set of data values ranging from 0 to 200. The values from 160 to 165 are important and you want them to show up in a contrasting color—not the evenly blended shade they would have with the neighboring values. 160 would be the **From** value; 165 would be the **To** value. Pick any contrasting hue, saturation, or opacity—say bright red (0.0), and enter it as both the **From** value and the **To** value, then click on **do interpolate**.

When you have byte data that ranges from 0 to 255, the mapping between the data values and the colormap slots is easy to figure out. You can use the **field legend** module, which takes both a colormap and a field as input, to see how the numeric values are mapping onto the colormap.

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, and so on. Visually, this flips the colormap over a horizontal axis.

flip

Similar to invert; it inverts the colormap for hue, saturation, brightness, or opacity, whichever is selected. But, where **invert** inverts the colormap about the horizontal axis, **flip** flips it about the vertical axis.

```
New value = (Max Colormap Value - Old Value)
```

cycle

The **cycle** button performs a circular shift on the colormap. How much the hues, saturation, or brightness moves is set by the type-in. The default is 1. The colormap shifts each time you click on the **cycle** button.

For example, **cycle** can be used as a pseudo contour generator for volume data. Create a colormap with an opacity like a narrow square wave - most values wholly transparent, with a narrow band wholly opaque (you could use the **From/To** type-ins). Setting a cycle step of 10, then repeatedly clicking on **cycle** would show the data as a series of pictures 0-10 data, 10-20 data, 20-30 data, and so on.

ramp

The **ramp** button sets the hue, saturation, brightness, or opacity values in a colormap to be an even graded interpolation between its minimum and maximum values. This is the default for hue and opacity.

smooth

The **smooth** button smooths out the distribution of values in the hue, saturation, brightness, or opacity of a colormap. Where the black hairline shows spikes of sudden changes, smooth produces curves. It performs a Gaussian convolution on the colormap values.

Lo Value/High Value

These dials let you set the minimum and maximum values of your data. These values are included in the **generate colormap** module's output colormap. Modules will use these values to normalize the range of the colormap to the range of the data in the field. You can use these dials when your data's range of values is either not evenly distributed between 0 and 255 (for example, your data is floating-point values from -1 to 1), or much of the data's values lie outside the 0 to 255 range (for example, from 0 to 10,000). You can also use it if you want to fan out the colors representing a narrow range of data. To see what your data's minimum and maximum values are, use the **statistics** module.

Note

The color range module will calculate the minimum and maximum data values in a field and store them in the output colormap. These dials are the manual way of doing the same thing. They are also more flexible. For an explanation of why this is a necessary option, see the color range module reference page in the *ConvexAVS Module Reference*.

Set the **Lo** and **High** values either with the floating-point dials, or by clicking the mouse on the dial's center to bring up a floating-point type-in.

Read and 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 the output windows produced by the **display image** and **display pixmap** modules. The **image viewer** and **graph viewer** modules have their own behavior, described in their respective chapters.

Images

An image is originally defined in terms of pixels. The only scaling ConvexAVS performs on images is successive doubling: x2, x4, x8, and so on. 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, and so on.), 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. Only part of the image is visible; to see more of it, use any mouse button to click-and-drag the image or use the scroll bars that appear along the window edges. They work the same way as a file browser.
- The left mouse button scrolls upward.
- The effect of the middle button depends on exactly where the cursor is:
  - **In the arrow box at the top (or left).** Click to scroll to the very top or left of the image.
  - **In the elevator shaft.** Click and hold down the button to grab the elevator bar. Moving the bar causes the image to scroll accordingly.
  - **In the arrow box at the bottom or right.** Click to scroll to the very bottom or right of the image.
- The right mouse button scrolls downward.

Pixmaps

In most cases, a *pixmap* is generated by the **render geometry** module. ConvexAVS can scale pixmaps continuously. When you resize a pixmap window, the picture always resizes accordingly by re-rendering at the new resolution.

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, and so on.

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 "Including display windows in a reorganized layout," on page 124.

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 a display window's pulldown menu

Each display window has a pulldown menu that allows you to resize the window's picture without having to use the X Window System window manager. To use the menu, click and hold down any mouse button on the small square at the left side of the windows title bar. Drag the mouse to highlight the desired menu choice, then release the button.

The **display pixmap** has only the first two options; **display image** has all of the following menu choices:

Zoom Full Screen
: Enlarges the display window to be (approximately) the largest possible square size. An image is scaled up accordingly; a pixmap is re-rendered.

Unzoom
: Restores a zoomed window to its former size. If you use the window manager to move and resize a zoomed window, AVS remembers the previous configuration as the unzoomed position and size. If the window had been moved inside a *page* or *stack* using the Layout Editor, it returns to that location. This option only appears if an image is resized.

Auto-Fit (Turn On/Off)
: This toggle switch controls the automatic sizing of display windows to exactly fit the current image size. By default, this feature is enabled.

Scrollbars (Turn On/Off)
: When an image is larger than its display window, ConvexAVS automatically adds scrollbars unless you turn this toggle switch off. You can configure Convex AVS not to use scrollbars by default: use the ImageScrollbars parameter in the ConvexAVS start-up file, described in Chapter 2.

Resize Window to Fit Image

Use this choice when **Auto-Fit** is turned off and an image is too big for its window. (If you don't use this function, you can scroll the image in order to see different parts of it.) This choice is also useful when ConvexAVS does not scale up an image (as described above), and you want to trim off the unused portion of the display window.

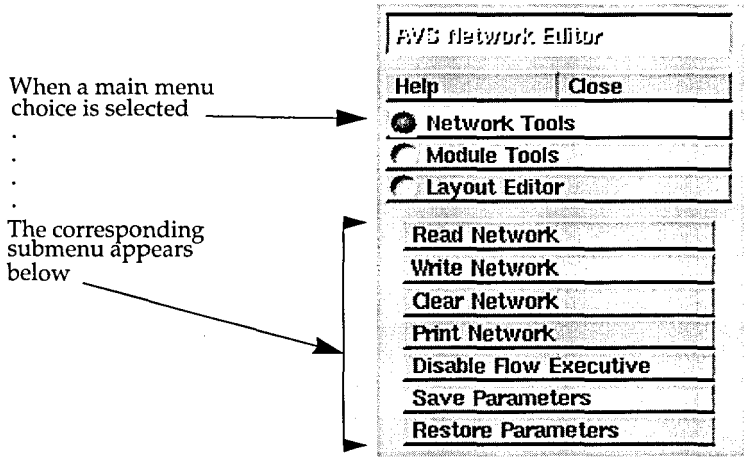
x1, x2, x4, etc.

Scales the image by a power of 2. The "(selected)" annotation indicates the scaling currently in use.

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, as shown in Figure 58. At any moment, one of the menu choices is selected, and the corresponding submenu appears below the main menu.

Figure 58
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 avs_help browser. This browser provides an overview of Network Editor usage and includes an online tutorial.

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 a network 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.

Performing a **Read Network** with the **Merge** option 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.

Write Network saves networks in Command Language Interpreter (CLI) instructions, as described in the section "Writing scripts," on page 596. It saves the state of the network to the extent that its CLI instructions allow. In general, **Write Network** saves:

- All the modules that have been placed in the workspace, and all of the connections between them. (CLI `net_show` command.)
- The current state of all module parameter settings that have been changed from their default values. The includes the names of files that have been read into the network. (CLI `parm_save` command.)

  If you wish to save all parameters, not just those which have changed, you must use the NetWriteAllParms keyword in your .avsrc file.

- The current state of the layout of the screen, including all changes made to widgets using the Layout Editor and the Parameter Editor (such as changing dials to sliders), and the current location of all display windows. (CLI layout command.)

Write Network does not save the current state of any of the viewers (Geometry Viewer, Image Viewer, Graph Viewer). Rather, it gives the viewers a chance to save their own state. Each does so in different measure. For example, the Geometry Viewer saves the current properties of its Cameras, Lights, and the Top-level object, but does not save any operations that have been done to child objects.

When saving references to ConvexAVS executables, **Write Network** always prefaces the binary file name with `$PATH`. `$PATH` usually resolves to /usr/avs. When saving references to data files or network files, **Write Network** by default saves absolute path names to the files. It is possible to substitute the `$DataDirectory` and `$NetDir` strings (if you have defined them) if you use `NetWriteAbsPath` **off** in your .avsrc file. Pathways to remote modules and remote data directories are always saved as `$RemMods` and `$RemData`, no matter how `NetWriteAbsPath` is set.

The network files are editable ASCII CLI files.

Clear Network

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

Print Network

This option creates a Postscript™ file:

/tmp/AVSnet*PID*.ps

The *PID* 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 is overwritten.

The default command is `lpr`. You can set this to another value using the PrintNetwork *.avsrc* keyword, as described on page 46 in Chapter 2, "Starting ConvexAVS."

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 do not wwant the network's modules to recompute after each change.

Save Parameters

Saves the current module parameter values for the current network in a parameters file, named

/tmp/avs_snap*N.PID*

(*PID* is the process number, and *N* an internal sequence 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 Remote Module(s)

This brings up a Remote Host Browser that lets you select another system in your network from which to read and execute modules. This is described in "Remote module execution," on page 125.

Read Module Library

The **Read Module Library** option 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. The previous module library is still accessible through the **Select Module Library** browser.

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/hsmith/avs_modules/smooth
file       /usr/hsmith/avs_modules/rough
directory  /usr/hsmith/avs_modules/tools_dir
```

A module library name must be unique. It cannot be the same as a library already loaded for the current session. Use the **Edit Module Library** option to list these libraries and to create new ones.

Write Module Library

Us the **Write Module Library** option to write a module library file, consisting of the modules currently in the palette. The module name must be unique.

Select Module Library

This option invokes the Module Library browser, 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 or through the `ModuleLibraries` resource in your .avsrc file or `-libraries` command line option. 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.

Edit Module Library

This option brings up a facility for interactively editing and creating module libraries. This is discussed in the section, "Constructing a module library," on page 132.

Flash Active Modules

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

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

Restart Modules

Brings up a panel that gives options for restarting modules that have died. Selecting **Restart** restarts all blacked out modules with their default parameter settings. **Restart** also will throw away an instance of a module and instantiate a fresh copy without requiring you to go through the usual user interface steps to accomplish this. **Restart Same** restarts all blacked-out modules with the same parameter settings. *The module(s)may still die.* Refer to "Module restart option," on page 96.

Redesigning the user interface

Selecting **Layout Editor** places the Network Editor in a mode that allows you to redesign the user interface of the current network. You can, for example, place widgets outside the Network Control panel to use screen space more effectively.

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.

For example, if you are developing a network useful for visualizing your type of data as a convenient, packaged application, you should use the Layout Editor to design your interface to the application. You can put the most commonly-used widgets on a single control panel for immediate access. Perhaps you create a bar along the bottom of the screen that shows all the controls for all of the modules in the network at once.

For another example, the **print field** and **compare field** modules display the contents of an ConvexAVS field file as ASCII data that you can read and interpret. They are often used when you are trying to import data with **read field**. However, their scrolling output browser is very narrow—often too narrow to actually see most of the contents of the field. You can use the Layout Editor to make the output browser as wide as you need.

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

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

This button creates a new, empty control panel page.

Create Stack

This button creates a new, empty stack.

Undo

This button everses 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 following form:

`/tmp/avs_undN.PID`

N is a small integer
PID is the process number

This file is part of the *undo* feature implementation.

Do not 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**.

Snap to Grid

Causes widgets to automatically align themselves according to a grid, making the resulting interface neater. The resolution of the grid squares defaults to 10x10 pixels. You can change this with the **GridSize** .avsrc startup file keyword.

Click the choices described above 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, as demonstrated in Figure 59; 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.

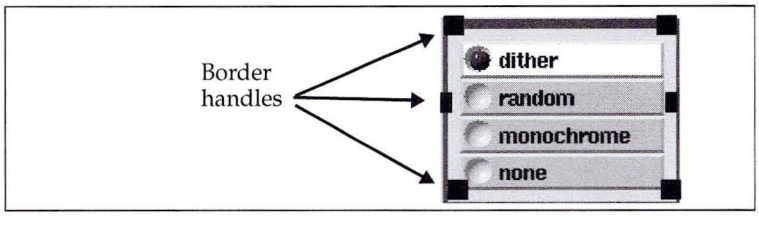


Figure 59
Window borders in the Layout Editor

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, described 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 (for example, 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**—Change the type of control widget (for example, change a slider to a dial). You can also change the type of a parameter's control widget using the Parameter Editor, as described above.

Resizing a text browser

To resize scrolling output browser of the **print field** module, you would do the following:

1. Drag a **print field** module into the workspace and make it the currently-selected module by clicking on its menu button on the Network Control panel.
2. Click on Layout Editor in the Network Editor main menu.
3. Place the mouse cursor inside the **print field** module scrolling output window. The border of the window should turn grey.
4. Press the left mouse button and drag the output window all or partially outside of the Network Control panel. The window is re-parented by your window manager.
5. With the cursor inside the output window, press the middle mouse button and drag it to the left or right to widen the window.
6. Position the output browser window anywhere that is convenient on the screen.
7. Leave the Layout Editor by clicking on either **Network Tools** or **Module Tools**.

Including display windows in a reorganized layout

You can include the display windows created by the **display image, display pixmap, image viewer,** and **graph viewer** 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.

In this way you can completely hide unneeded Network Editor functions, reduce the need for frequent window manager operations, and present a more traditional application interface.

Remote module execution

ConvexAVS supports the synchronous execution of modules on remote hosts. The network communication and data transfer mechanism between the ConvexAVS kernel/flow executive and the remote module are based on standard UNIX TCP/IP network protocols. Data representation is based on Sun's External Data Representation (XDR). Thus, the remote host can be heterogeneous—of a hardware type other than type of machine the ConvexAVS kernel is executing on. The user interface to remote modules is smoothly integrated in the ConvexAVS Network Editor.

There are many situations where you might want to include one or more remote modules in an ConvexAVS network; these are a few typical cases:

- You have a compute-intensive module that runs best on a particular kind of hardware
- You have a module, perhaps a real time data collection application, that *only* runs on a particular kind of hardware.
- The data files your ConvexAVS network needs reside on a remote host's file system that is not easily accessible from your workstation; it would be easier to read the files on the remote host, than to manually transfer the data to your workstation's file system domain.

All modules in an ConvexAVS network (except coroutines) execute synchronously—one finishes, the next begins. In any given situation you must ascertain whether the benefits of remote heterogeneous module execution outweigh any data transfer overhead that might be introduced.

There are three aspects to remote module execution:

- What must be present on the *remote system*
- The *hosts* file that ConvexAVS uses to access remote modules
- The ConvexAVS Network Editor user interface to remote modules

The exact interface to the remote system, particularly the command (`rsh` or `c_rsh` or some other variant) that is used to first establish contact with the remote host, may differ from system to system. This manual describes the general case that will usually work between UNIX hosts on an open network that does not place barriers to a user going from one system to another. Users should consult the AVS documentation for the remote system.

Remote system

The remote module(s) must have been compiled and linked on the remote machine against Version 3 of the standard AVS libraries.

Note

Modules compiled and linked under ConvexAVS 1.0 cannot be executed remotely.

No additional libraries or calls are needed to make a module into a remote module.

Certain modules cannot execute remotely:

- The following built-in ConvexAVS modules are part of the ConvexAVS kernel, and therefore cannot be remote modules:
  - **generate colormap**
  - **display image**
  - **display pixmap**
  - **graph viewer**
  - **image viewer**
  - **render geometry**
  - **write image**
  - **write volume**
  - **colormap manager**
  - **image manager**
  - **render manager**
- Modules that write results to a /tmp file if other parts of the system must also read the /tmp file. This is because the remote module and the ConvexAVS running on the workstation may not share the same file space domain. Modules that do this include:
  - **print field**
  - **compare field**
  - **vbuffer**

The remote host must accept TCP/IP network communications through the Berkeley UNIX socket mechanism.

The remote host must have some way of accepting remote requests to execute programs on its system, such as the Berkeley UNIX `rsh` (remote shell) command. (Your workstation must be able to produce these requests.)

Note

Data is transferred between modules in device-independent XDR format. However, if the remote hardware has a 64 bit word size and the local workstation has a 32 bit word size, integers and real numbers past a certain size will lose significance when transferred.

The remote host does not need to share a file system domain with the local host.

Setting up a remote module directory

Together with ConvexAVS, the self-contained directory of modules described below is all that is required on the remote host side. Multiple sets of modules can be accessible on the remote host, each set in its own directory.

- Make a directory on the remote host.
- Place in it a collection of modules compiled on the remote host (or an identical system) and linked against ConvexAVS libraries. The format is the same as in any directory containing ConvexAVS modules—there can be one binary per module, or multiple modules in a single binary. (The Remote Module execution mechanism does **not** support reading remote ASCII module **library** files. However, one can have a local library, some or all of whose elements are remote modules. This is described later.)
- The directory must contain an executable binary of the AVS module **list_dir**. The **list_dir** module is the point of first contact for the ConvexAVS kernel attempting to access remote modules. Specifically, it is a special module that lists the contents of the remote directory for the remote user and implements the Remote File Browser. The **list_dir** module is not in any module palette. A binary of this module can be usually be copied from a file named list_dir on the remote host in the /usr/avs/avs_library directory.

Finding remote modules: the hosts file

The ConvexAVS kernel uses a *hosts* file to find remote module directories. When you click on **Read Remote Module(s)**, ConvexAVS looks for a *hosts* file in two places, in this order of precedence:

- Your .avsrc startup file may contain a Hosts specification like the following:

  `Hosts` *full-file-specification*

  For example:

  ```
  Hosts /home/users/username/avsstuff/hosts
  ```

  or

  ```
  Hosts /hostname/ourproj/avs/util/remotehosts
  ```

  If such a specification exists, ConvexAVS will use the file named as a *hosts* file. The file must exist and be accessible on your local workstation.

  This mechanism lets you create and maintain your own list of remote module directories.

- ConvexAVS uses the system default hosts file in /usr/avs/runtime/hosts. It is expected that a ConvexAVS system administrator would maintain this file. The ConvexAVS product comes with a sample /usr/avs/runtime/hosts file to use as a template.

File format for the hosts file

The *hosts* file is an ASCII file. Each line in the file has four columns of information. If the string occupying a column contains blanks, it must be enclosed in double quote marks ("/usr/ucb/rsh host -n"). Here is a sample:

```
machine1_std "/usr/ucb/rsh machine1 -n" /usr/avs/avs_library  /usr/avs/data
machine1_my  "/usr/ucb/rsh machine1 -n" /usr/my/avs/modules   /usr/my/avs/data
machine3     "/usr/avs/bin/c_rsh mach3" /usr/avs/avs_library  /usr/avs/data
```

- **First column**—Contains a logical name. This is the name that will appear in the Remote Host browser described below. It identifies the line of instructions to execute. This makes it possible to have multiple sets of AVS modules and data on remote hosts that you can select among as the situation demands. You can also run a network on a different host by modifying the mapping of the logical name to the actual remote host.

- **Second column**—Full file specification of a program that will run a command on the remote host. This is where the specific remote host is actually identified.

  The obvious choice for this function in networks of Unix-type machines is *rsh* (remote shell). This program is usually found in the /usr/ucb directory in your workstation's file system domain.

Note

Add the -n flag to *rsh*. It will prevent the remote process from grabbing terminal input, placing your ConvexAVS in the background where it will seem to hang.

  Many workstations are on secure networks where the remote host will not permit a `rsh` from the network to execute. The remote host may require that you authenticate yourself first. Consult the remote host's documentation.

  Because there are blanks in this column, enclose it with double quote marks.

- **Third column**—Directory on the remote host where the modules are kept. This string is part of the `rsh` command that ConvexAVS sends the remote machine. Thus, this file specification is in the file system domain that the remote machine understands. This may or may not be the same as your workstation file system domain.
- **Fourth column**—Default data directory on the remote machine. Many ConvexAVS modules such as **read field** use a file browser widget. This tells them what to use as the default directory for the file browser widget. Again, this file specification is in the file system domain that the remote machine understands. This may or may not be the same as your workstation file system domain.

The example shows three cases:

- The first is a standard UNIX file specification, presumably on some powerful cycle server machine called `machine1_1`. The location of the modules and data specified are the usual standard ConvexAVS modules in the standard place.
- The second is also a UNIX file specification for the cycle server `machine1`; however, here the modules and data are kept in some private directory for a research project.
- The third example uses the `c_rsh` program to run modules on a remote system called `mach3`.

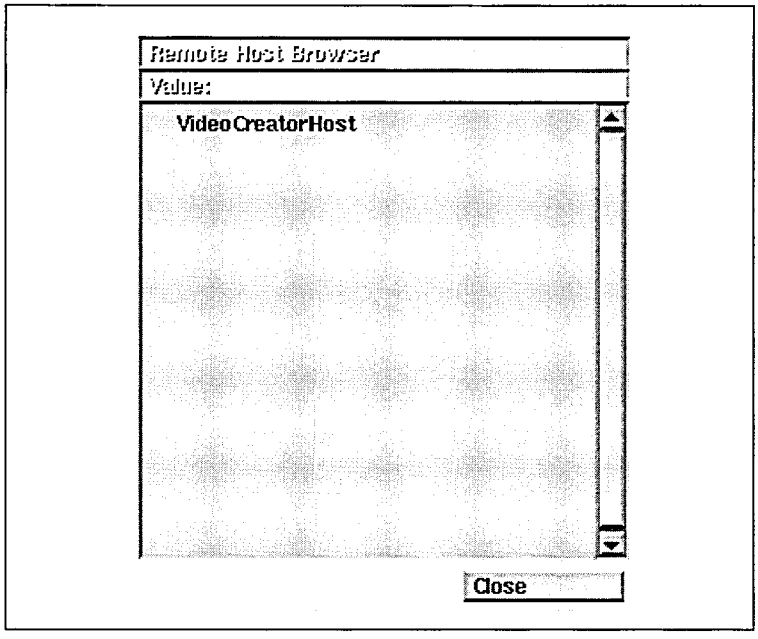
The first module that the ConvexAVS kernel directs the remote machine to execute is **list_dir**, which produces the directory listing of the Remote Module Browser. Once started, the **list_dir** module continues to run until the ConvexAVS session ends.

Network Editor user interface

The user interface to run remote modules is logically and smoothly integrated into the ConvexAVS Network Editor. The interaction is basically identical to reading and using modules from a local file system with **Read Module(s)**.

1. Start ConvexAVS and enter the Network Editor as usual.
2. Click on **Module Tools** on the main Network Editor menu.
3. The Module Tools submenu has a button: **Read Remote Module(s)**. Clicking on this button raises the Remote Host Browser, shown in Figure 60.

Figure 60
Remote host browser



The host names in the Browser window list are taken either from the hosts file specified by the `Hosts` line in your .avsrc startup file, or from the system /usr/avs/runtime/hosts file.

The Remote Host Browser functions like any other ConvexAVS browser; click on a host name to select it. Scrollbars scroll the list of host names. **Close** takes down the browser.

4. Once you click on a host name, the Remote Host Browser is replaced with the Remote Module Browser. The names in the Remote Module Browser are the file names of modules on the remote host.

   ConvexAVS reads the "run remote program" command for that host from the second column of the hosts file, usually /usr/ucb/rsh. It executes the remote program giving the host name, remote module and file directory specifications (third and fourth column of hosts), and **list_dir** as parameters. It also passes the X Window System DISPLAY environment variable to the remote host so that it can make windows on your workstation. If the connection to the remote host was successful, and the remote directories were found, and **list_dir** was initiated, then the Remote Module Browser widget appears with the remote modules listed.

5. To load a remote module into the module palette, click on its name in the Remote File Browser. If the module has the same name as an existing module in the palette, it replaces it. If it has a new name, it is added to the module palette. The visual clue that this is a remote module is the module icon's dimple—it is pink instead of the usual gray.

   Thereafter, use the remote module like any other module: drag it into the workspace with the left mouse button, display its Module Editor by clicking the right mouse button on its dimple; turn its parameter ports on and off, connect it up to other modules in the network, click on its button, manipulate its widgets, run data through it, disconnect and remove it just like any other module.

Only two differences may be noticeable:

- Remote modules with file browser widgets, such as a remote read field, have Remote File Browser widgets. These communicate with the remote **list_dir** module. Remote File Browser widgets navigate in the file space domain of the remote host.

  You can use the Remote File Browser's **New Dir** or **New File** button to select a remote directory.

- Remote modules that are not part of the standard ConvexAVS set may not have online module reference pages associated with them that are accessible through the Module Editor's **Show Module Documentation** button.

Otherwise the interaction is the same.

If you save a network that contains remote modules, remote modules in the directory specified in the hosts file are saved with the string **$RemMods** pre-pended to the module's name. When the network is read in again, **$RemMods** are replaced with the module directory for the destination host. This makes it possible to define networks containing both local and remote modules that can be initiated independent of any specific remote host. The network should be read in correctly provided that the **Read Network** function is issued in an environment where the logical host name can be interpreted by a hosts file.

If you save an ASCII module library containing some (or all) remote modules, it should also work correctly when it is read in again, providing that the Read Module Library function is issued in an environment where the logical name can be interpreted by a hosts file. You can have such libraries loaded automatically when you start ConvexAVS by adding the library's file name to your .avsrc startup file.

Note

This release of ConvexAVS does not support reading or writing *remote* module *libraries*—that is, libraries of modules whose ASCII description file exists on a remote machine.

Constructing a module library

You may find that the collection of modules in the ConvexAVS supported and unsupported module libraries 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, and so on.

There are two approaches to constructing your own module libraries:

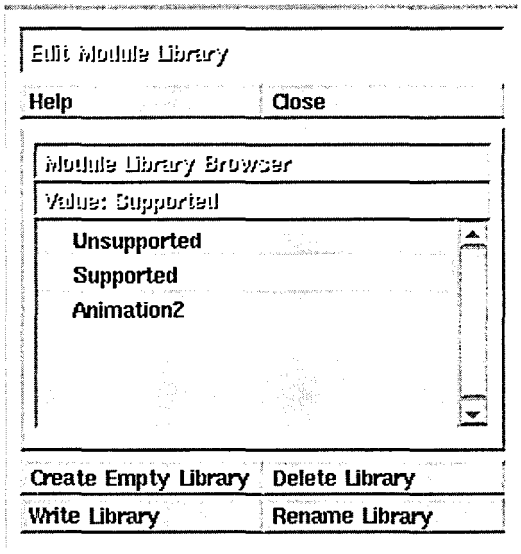
- Module libraries are defined by ASCII text files. You can delete and move module descriptions using any text editor. It is not practical to type in new module descriptions.
- You can use the Module Tools menu's **Edit Module Library** function to interactively add and delete modules from a module library. This interactive editing does not support direct manipulation moving of modules from one palette to another.

Sample procedure

Suppose you are an instructor teaching a class on the behavior of particles in vector fields. You are going to use ConvexAVS for your students' lab work. The default Supported module library in the ConvexAVS palette contains far more modules than are pertinent to the subject. You've also written a few data input and visualization modules that you want to add to the students' module palette.

1. Start ConvexAVS and enter the Network Editor with the default Supported module library in the palette.
2. Click on **Module Tools** on the Network Editor menu.
3. Click on **Edit Module Library** on the new submenu. This brings up the Edit Module Library panel, shown in Figure 61.

Figure 61
Edit Module Library panel



4. Delete the modules you don't want:
   - Position the mouse cursor over the module's icon in the module palette.
   - Press and hold down the left mouse button.
   - Drag the module icon into the workspace all the way down until it is over the Hammer icon and release the mouse button. The module is deleted from the palette.

   If you release the mouse button before the module icon is over the hammer, then the module is not deleted from the palette. You must remove the instanced version of the module and repeat the operation again from the start.

There is no module editing mode. It is possible to inadvertently delete modules from the palette if you accidently drag a module icon all the way to the hammer without releasing the mouse button. If this happens, you can read in a complete copy of the library with **Read Module Library.**

It may be faster to do this initial large-scale deletion using a text editor on a copy of the actual library file.

5. Now begin adding in your own modules. Click on **Read Modules** and individually type in the file specification for each module's binary file. They are added to the palette.
6. If your module binary files have multiple modules in each file, you can delete the individual elements that you don't want interactively as described above.
7. Next, click on **Rename Library** on the Edit Module Library panel and type-in a new library name. This changes the string that is displayed at the top of the palette from *Supported* to your own name.
8. The final step is to use the Edit Module Library panel's **Write Library** function to create the ASCII library file version of what is in the palette to disk. Module library files do not have a particular file suffix. The name must be unique (different from existing module library names).

   You will have to exit ConvexAVS and then re-enter it before your new library will show in the Read Module Library browser.

One case is not explicitly covered here: suppose there is one module in the *Unsupported* library that you want in your own module palette. There is no direct manipulation way to move modules from one palette to another as there is to delete modules from a palette. Instead, use **Read Module** to read the binary form of the unsuported module found in the directory /usr/avs/unsupp_mods.

The Edit Module Library panel's **Create Module Library** functions creates a new, empty library in the palette area. The new library is not written to disk until you enter **Write Library.** The Delete Library function deletes the library from the palette, not from disk.

Lastly, you should give your students a .avsrc or .avsrc.X file with a ModuleLibraries line in it that will load your module library automatically when they start ConvexAVS. To be the default library that shows in the Network Editor module palette, your library file specification must be the last listed on the ModuleLibraries line.

Module library file format

You can look at the ConvexAVS-supplied module library in /usr/avs/avs_library/Supported to see what a library file looks like. ConvexAVS itself uses a somewhat more complex file format for the module library file it creates when you select the **Write Library** function on the Edit Module Library panel. The more complex format allows the Network Editor to load the module library more quickly. The format encapsulates the information created by the module's `AVS module_from_desc` call.

ID Line

The module library file must begin with this line:

```
# AVS Module Library
```

Other lines that begin with # comment character may precede this line.

Command Lines

Each subsequent line must begins in one of these forms:

```
builtin "module_name"
external "module_name" n "filename"
remote host module_name n filename
file filename directory  dirname
```

`builtin` specifies a module that is built into ConvexAVS itself, rather than being implemented as a separate executable file. These are the current built-in modules:

- **generate colormap**
- **display image**
- **display pixmap**
- **graph viewer**
- **image viewer**
- **render geometry**
- **write image**
- **write volume**
- **colormap manager**
- **image manager**
- **render manager**

`external` The modules is not built-in. This entry expects the detailed module description information created by **Write Library** to be present in the library file. Here, module_name defines what string to display for the module on the module icon. *filename* is the file where the binary form of the module can be found.

`remote` The same as `external` except a logical host name is specified. It is used to locate modules that will execute on remote hosts in the network.

`file` Specifies an executable file that includes one or more module definitions. `file` is the same as `external`, except that it directs the Network Editor to read the binary file *filename* and, from that, construct the detailed description information. This is much slower for the Network Editor.

`directory` Specifies a directory that includes executable module definition files. Like `file`, it makes the Network Editor open each file to discover the description information, rather than finding it in the library file. This can be very slow for large directories.

In each case, if the file name or directory is a simple file name or directory name without a path, then ConvexAVS assumes that it resides in the same directory as the library file. Otherwise, it expects a complete path.

The description information is used when a person brings up the Module Editor panel for the module by clicking on its dimple to fill in the panel's information. It is only relevant before a module is dragged into the workspace. It is not practical to type in this information yourself. You should let the **Write Library** function do it for you.

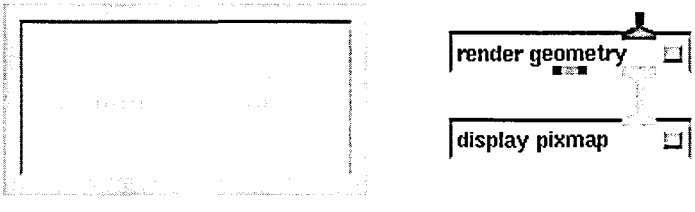
Geometry Viewer subsystem

4

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

The Geometry Viewer exists in two forms. It is a ConvexAVS subsystem accessible from the main menu and from any other ConvexAVS subsystem through the **Data Viewers** pop-up menu. It also exists as the **render geometry** module in ConvexAVS networks. (As **render geometry** it is almost always coupled with the **display pixmap** module that produces the output viewing window.) The Main menu button and the module pair are shown in Figure 62.

Figure 62
Geometry Viewer access



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

The *geometries* can enter the Geometry Viewer from two sources:

- You can read .geom format files directly into the Geometry Viewer through the Read Object button on the Objects submenu. These .geom format files were either created by a program making calls to the libgeom library, or were saved to disk during an earlier Geometry Viewer session. The geometries can be pure geometries (.geom files), or included as part of higher abstractions called *objects* (.obj files), or as components of an entire 3-D scene (.scene file).

  The Geometry Viewer can also read some external file formats directly (Wavefront, Movie.BYU, Brookhaven Protein Data Bank, and so on.) and transparently convert them to .geom files. See "Geometry filters," on page 284 for a list of supported external file formats.

- Geometries can flow into the **render geometry** module from a network. Most mapper modules (**scatter dots, isosurface, arbitrary slicer, hedgehog, ucd to geom,** and so on.) create geometries as their visualization representation of numeric data. This is signified by their red (geometry) output ports. For example, the **isosurface** module takes the numeric data in a 3-D field, then uses the level value you give it to create a 3-D contour surface through the volume. This 3-D contour is output as a geometry of vertices in 3-D space connected together with surfaces.

Note

The data values from which the geometry was derived are not present in the geometry itself.

The **render geometry** module's red geometry input port can accept connections from multiple modules, shownin Figure 63. The geometries from all the modules will be combined together into one scene, as in Figure 64.

Sample geometries, objects, and scenes are found in the directory /usr/avs/data/geometry. Sample scripts that produce networks with geometries are accessible through the Help Demos browser.

Figure 63
Network that produces three geometries

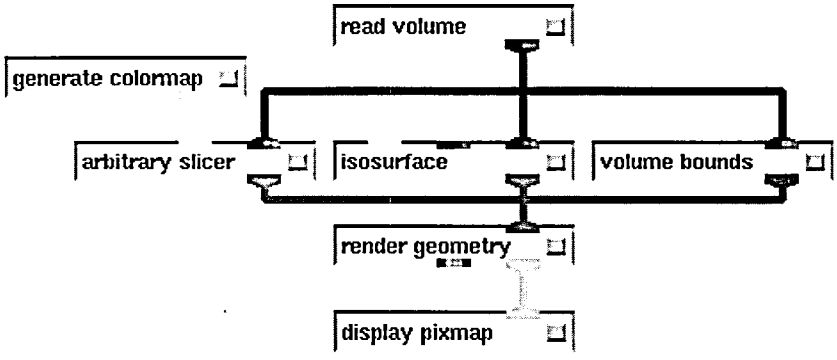
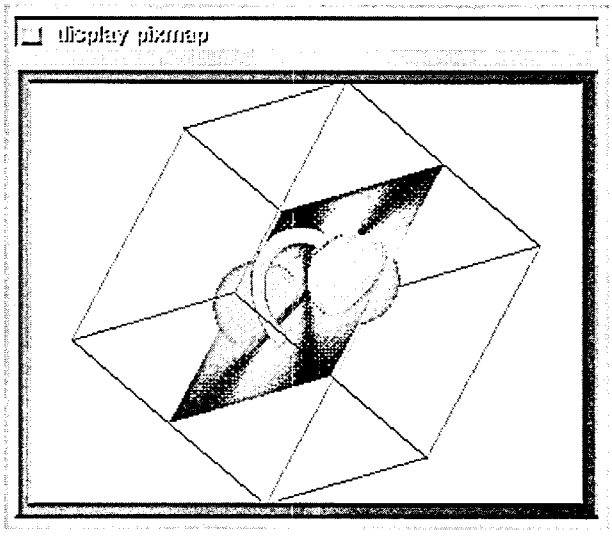


Figure 64
Three geometries combined in one scene



Entering the Geometry Viewer

The Geometry Viewer 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 "Geometry specific options," on page 41 in Chapter 2 for additional command-line options that affect the manner in which the Geometry Viewer is invoked.

From the ConvexAVS main menu

Click the Geometry Viewer choice on the ConvexAVS main menu.

From another subsystem

At the top of each of the four major AVS subsystem control panels (Image Viewer, Graph Viewer, Geometry Viewer, Network Editor) is a button titled **Data Viewers**. Position the mouse cursor over **Data Viewers**, then press *and hold down* any mouse button. A pop-up menu appears. Still holding the mouse button down, roll the cursor down the pop-up menu until **Geometry Viewer** is highlighted, then release the mouse button. This calls up the Geometry Viewer's control panel. If you transfer to the other subsystems, then return to the Geometry Viewer, the Geometry Viewer's control panel will remain in the state that you left it.

You can move the Geometry Viewer's control panel with your window manager. This is convenient if there is more than one control panel you need to use at one time.

From the Command Line Interpreter (CLI)

It is also possible to drive the Geometry Viewer from the ConvexAVS Command Language Interpreter (CLI). You can type Geometry Viewer CLI commands to the CLI and interactively view the results, you can compose a script of CLI Geometry Viewer commands that will run automatically, or you can write a module that sends Geometry Viewer CLI commands to the Geometry Viewer through the ConvexAVS kernel. The CLI interface is discussed in Chapter 17, "Command Language Interpreter."

Leaving the Geometry Viewer

If the Geometry Viewer was invoked from the shell command line as `avs -geometry` then at the top of its main control panel will be a button labeled **Exit**. Press **Exit** with any mouse button to end the ConvexAVS session and return to the system shell.

If the Geometry Viewer was entered from the main menu or through the **Data Viewers** pop-up menu from another subsystem, then there will be a **Close** button at the top of its main control panel. **Close** is not an exit button. **Close** simply takes down the Geometry Viewer's control panel; one could get a similar effect by using the system's window manager to iconify the control panel. When you later re-enter the Geometry Viewer, the control panel is in the same state that you left it. The only way to exit the Geometry Viewer is to exit ConvexAVS altogether from the main menu.

If the Geometry Viewer was invoked as the **render geometry/ display pixmap** modules, then there is still only one Geometry Viewer, even though there may be multiple **render geometry/ display pixmap** modules. The modules are associated with individual Geometry Viewer scenes, not with the Geometry Viewer itself. Throwing away an **render geometry/display pixmap** module pair deletes a scene.

Scenes: objects, lights, cameras

Using the Geometry Viewer, you can work with one or more scenes.

Each scene consists of:

- A *world space*. World space is an unbounded 3-dimensional volume. Location in world space is measured by *world coordinates*. World coordinates are a single set of right-handed X, Y, and Z axes. When the Geometry Viewer first comes up, the world coordinate origin (0,0,0) is positioned in the center of the view window. The XY plane is parallel to the screen face. Positive Y is straight up, positive X is to the right, and positive Z is coming straight out of the screen toward you. World coordinates are also referred to synonymously as *scene coordinates*. You do not see world/scene axes unless you turn on **Axes for Scene** under the **Cameras** submenu. Then they will appear, unlabeled, extending from +5 to -5 in world space in the X, Y, and Z directions. From a conceptual point of view, world space is immobile.

- A collection of 3-D *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.

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.

Within each view window, you can *translate* (move) the objects, lights, and cameras; *scale* them (make them larger or smaller), and *rotate* them. These three operations are called *transformations*. Objects, lights, and cameras are collectively called *transformables*.

Note

In order to save *exactly* what you see in a view window, including the positioning of multiple objects, all of the lighting effects and colors, and all camera manipulations including Perspective, you must use Save Scene. New users often spend some time composing an attractive scene, then make the mistake of using only Save Object, which saves none of these features.

Objects

Objects are read into world space with the Geometry Viewer's **Read Object** function under the Objects submenu, or flow into the **render geometry** module from another module.

Objects are organized into a *hierarchy*. At the top of the hierarchy is a root object named simply **Top**.

All objects that you read into the Geometry Viewer, either through **Read Object** or that flow into the **render geometry** module, are children of the top level object, and peers of each other.

The hierarchy is normally only these two levels—top and everything else. Though you cannot do it from within the Geometry Viewer itself, it is possible to create multi-level object hierarchies with the Geometry Viewer's Script Language, as described in Chapter 18, "Geometry Script Language commands," on page 617, the CLI `geom_set_parent` command, and the libgeom library.

Each object has:

- Its own *coordinate system*. The top level object's coordinates are initially coextensive with the world coordinate axes. The child objects' coordinates are initially coextensive with those of the top level object.
- *Extents*. Its extents are the minimum rectangular volume in its own coordinates that the object occupies (max x - min x, max y - min y, max z - min z). The **Bounding Box** button will show you the current object's extents as you scale, rotate, or translate the object.
- A *center*. This is the point, in its own coordinate system, about which the object will rotate.
- A *name*. The name is of the form *name.number*. The name is set by the program that created the geometry. Geometry objects that come from modules are usually named after the module that created them. The Geometry Viewer appends a sequence number to the object name to distinguish between objects coming from two modules with the same name. You can rename objects using the Current Object Browser described in "Current object browser," on page 157.
- A *visibility attribute* which can be set on or off to make the object visible or invisible.
- *Surface properties* such as color, transparency, reflectiveness, and so on. A simple geometry in a .geom file has no properties except possibly vertex color and normals—a vector perpendicular to each plane used to calculate the angle of reflection. Properties affect the way the Geometry Viewer portrays an object. Object properties can be saved permanently in .obj and/or .prop property files. The surface properties determine how the object reflects light. By selecting various property combinations (**Edit Property** in the Object submenu), you can make an object appear to be made of different materials like metal or clay, or even semi-transparent glass.

You can transform the top level object, which carries all its child objects along with it, within world coordinates. You can also transform individual objects *within* the top level object's coordinate system.

Where objects appear in world space

When you read an object into the Geometry Viewer with the **Read Object** button, it is placed into the top level object's coordinate system at its true location and size. The object *might* be all or partially outside the current view window. Clicking on the **Normalize** button in the Geometry Viewer's control panel brings it immediately into the view window.

When an object enters the Geometry Viewer (render geometry module) through a network from a mapper module such as **arbitrary slicer**, **volume bounds**, or **hedgehog**, one of two things may occur:

- As with the **Read Object** option, the object is placed at its true location and size within the top level object's XYZ coordinate system. Part or all of the object may be outside the current view volume. (This is how, for example, the **arbitrary slicer** module behaves.) You can get the entire object into the view volume by pressing the **Normalize** button.
- Some mapper modules elect to normalize an object within the view volume themselves. By making an additional call to the libgeom library (GEOMedit_window), they instruct the Geometry Viewer to rescale and translate the top level object's coordinate system so that the object will fit within the display viewport. If a mapper module makes this call, the object will *appear* centered in the display window and filling it automatically. If you bring up the Transformation Options panel (described below), and examine the absolute scale and translation of the top level object and the mapper's object, you will see that the mapper object is at its real size and location in the top level object's coordinate space, and that the top level object has been rescaled and its coordinate axes shifted within world coordinates to accommodate the object in the view.

  This approach has the advantage that you always immediately see all of an object. This can be a disadvantage if you are, for example, creating an animation. The scene in the display window may jump each time a new geometry enters as the top level coordinates are normalized.

If multiple modules are sending output to the **render geometry** module, and if any one of the modules makes this call, then all objects will appear within the view volume. The **volume bounds** module makes this call, for example, and will affect the output of **arbitrary slicer** even though the **arbitrary slicer** module does not make the call.

Geometry Viewer control panel

Figure 65 shows the main Geometry Viewer control panel. Throughout the control panel and its submenus, press *any* mouse button to select *any* of the control buttons. The control panel is just another window on the screen. You can use your window manager to move it around, or even partially off the screen.

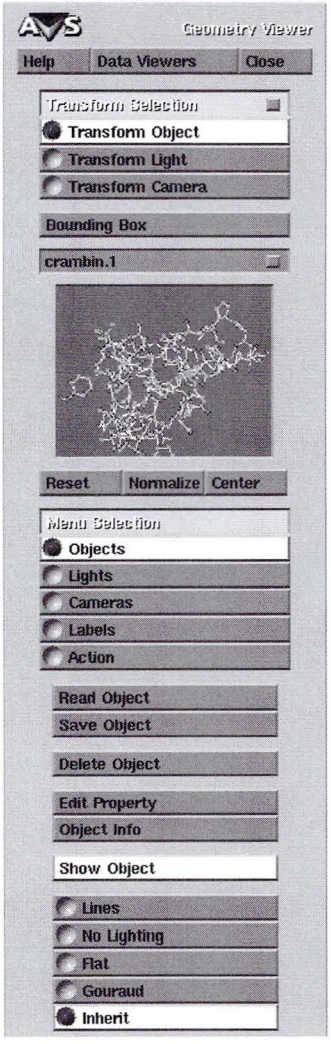


Figure 65
Geometry Viewer control panel

Top control bar

The three buttons at the top of the Geometry Viewer, shown in Figure 66, have the same meaning as in other ConvexAVS subsystems.

Figure 66
Top control bar

The **Help** button invokes the Help browser, with a selection of topics relevant to the Geometry Viewer.

The **Data Viewers** button brings up a pop-up menu that brings up the control panels of the other subsystems. Note that if you use **Data Viewers** to switch to another subsystem, the Geometry Viewer's control panel is not taken down from the screen, it is merely obscured by the new control panel. You could use your window manager to move it to another part of the screen and have multiple control panels showing at once.

The **Close** button unmaps the Geometry Viewer control panel from the screen. It does not exit the Geometry Viewer.

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

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 buttons select which class of transformable (objects, lights, or cameras) is going to be affected by a mouse button or Transformation Options panel selection. It is a transformation mode switch.

There are two main styles of transforming objects:

- Direct manipulation transformations performed with the workstation's mouse buttons in concert with the main keyboard's **SHIFT** key. In this type of transformation, you simply make the transformable current, then point at it with the mouse cursor, press the correct mouse buttons, and move the object around the view window.
- Precise object transformations performed with the Transformation Options type-in panel, or using the keyboard's arrow keys. In this mode, you also make the transformable current, then you type in to the Transformation Options panel the exact absolute or relative movement you want the object to make.

Direct transformations

Transformations using the mouse apply to actions performed within the view window. 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 within the Geometry Viewer windows:

Left mouse button

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.

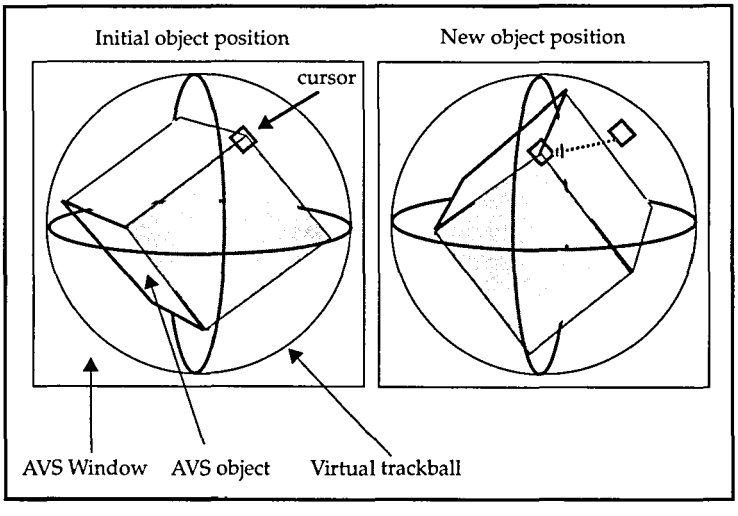
Three clicks: Expands the selection to the entire top-level object, which includes the jet and, perhaps, several other objects.

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

Middle mouse button

A dragging action (holding down the middle mouse button and then moving the mouse) rotates the current object in 3-D space. The object behaves as if it were attached to a trackball whose center is at the center of the view window, as illustrated in Figure 67. The mouse cursor is attached to the part of the trackball that protrudes above the surface of the window.

Figure 67
Rotating an object with the virtual trackball



Right mouse button

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

SHIFT -right mouse button

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.)

Note

You will not see the size of the object change when you transform it in the Z plane unless you turn on Perspective under the Cameras menu.

SHIFT-middle mouse button

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 inside the Geometry Viewer window:

Left mouse button

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 **Lights** menu is selected).

Middle mouse button

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

Right mouse button

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

SHIFT-right mouse button

Applies to point and spot lights only. A dragging action translates the selected light perpendicular to the plane of the window.

If the scene is not drawn in perspective (see "Cameras" section below), this will have no effect.

SHIFT-middle mouse button

Scales the **Show Lights** representation of the light source. For point and spot lights, this also translates the light source itself.

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 inside the Geometry Viewer window:

Left mouse button

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. The current camera window will be surrounded with a red border.

Middle mouse button

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 button

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

SHIFT-right mouse button

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

SHIFT-right mouse button

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

Note

When transforming cameras, light locations do not change relative to the objects in the scene.

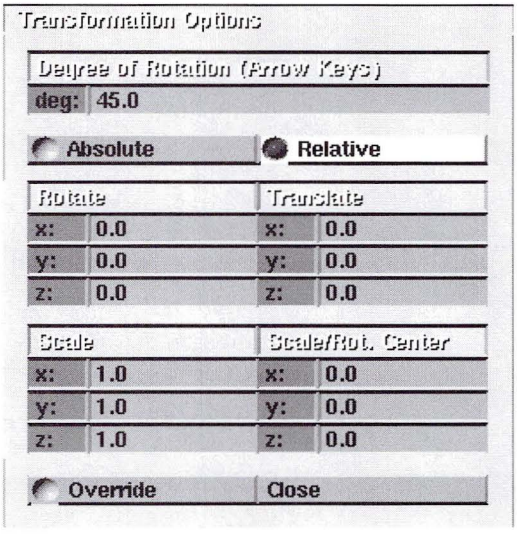
Transforming labels

Labels are *objects*. For this reason, there is no **Transform Label** button on the Transformation Selection menu. See "Transforming objects," on page 148.

Precise transformations

The Transform Selection menu bar has a dimple button. Clicking on this dimple produces the Transformation Options panel, shown in Figure 68.

Figure 68
Transformation Options panel



The Transformation Options panel allows you to transform objects, hierarchies of objects, lights, and cameras by precise values typed in. The movements can be relative to the current position, or to an absolute position.

Degree of rotation: arrow keys

You can rotate objects, hierarchies of objects, lights, and the camera with the keyboard's arrow keys. The degree of rotation is set with the type-in at the top of the Transformation Options panel. The default is 45.0 degrees. The minimum is 0.0 degrees and the maximum is 360.0 degrees. Change the default value by moving the mouse cursor into the type-in area and type a new value. As with all type-ins, **CTRL-U** deletes the entire line, and **BACKSPACE** deletes the previous character. You do not have to press **ENTER** for the new value to take effect. However, you *do* have to move the mouse cursor into the display window before the arrow keys work.

- **Left and Right Arrow Keys**—The left and right arrow keys rotate the current transformable about its Y axis.
- **Up and Down Arrow Keys**—The up and down arrow keys rotate the current transformable about its X axis.

- **Shift Left and Right Arrow Keys**—Pressing the **SHIFT** key together with the left and right arrow keys rotates the current transformable about its Z axis.

Note

If your window manager has defined the arrow keys for its own purposes, the functions just described may not work.

Transformation Options panel type-ins do not have to be finished with the **ENTER** key. Moving the mouse cursor out of the type-in window will cause the transformation to occur. Repeatedly pressing the **ENTER** key repeatedly applies the transformation.

Absolute and Relative

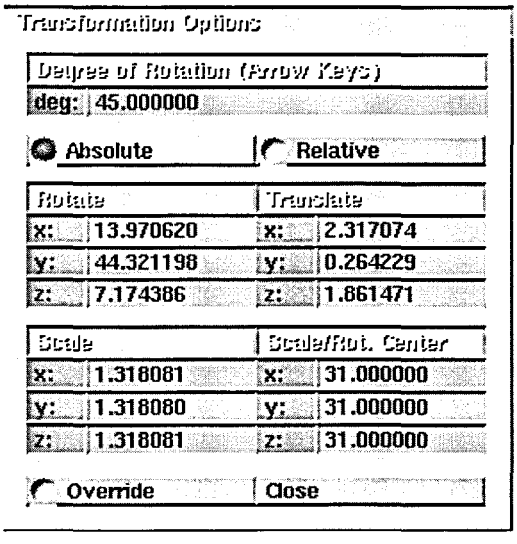
These two switches select between transforming the current transformable relative to its current position or state, or to an absolute position or state, as shown in Figure 69. The default is **Relative**.

Absolute sets the object's transformation matrix to be specified values. **Relative** appends the object's transformation matrix with the values specified.

When an object is selected:

- **Translate**—Objects are translated with respect to the object's *parent* coordinate system, not within its own coordinate system. With individual objects this usually means, translate the object in the top level object's coordinate system. For the top level object, it means translate the top level object within the world coordinate system.
- **Rotate**—Objects are rotated with respect to their center point, in their own coordinate system. Thus, if the top level object's Z axis is sticking straight out of the screen, and the object's Z axis is pointing up, the object will rotate around the Z axis that is pointing up.
- **Scale**—Objects are scaled in their own coordinate system with respect to their center points. Thus, if you scale an object to be twice its original size, then select the top level object and scale it to be twice its original size, both believe they are twice their original size. The object does not think it is four times its original size, although that is how it appears.
- **Scale/Rotate Center**—This defines the center point for the object. Objects are rotated and scaled with respect to their center point. The center point of an object is within its own coordinate system, not that of its parent.

Figure 69
Absolute transformation values



Absolute

When you first toggle **Absolute**, it reports the current state of the current transformable: where the object is, how it is rotated, scaled, and where its center is. You can then type in new values for all these cells, *except* Rotate. You cannot do absolute rotations.

Note

The Absolute display of the Transformation Options panel is not updated automatically if you perform a transformation with the mouse or arrow keys, or if you change the current transformable. To see the new absolute position, click on Relative then back to Absolute and the new information will be displayed.

On occasion you may see the string "**Not Avail**" instead of the absolute positions. It is possible for an object to get into a state where the equations that calculate its position, rotation, and scale are undefined (not available). This might occur, for example, if you rescaled an object in the X direction, but not in any other.

The meaning of the absolute values for cameras are not intuitive. They do not represent where the camera is in space and how big it is. Rather, **Scale** shows how much world coordinate space had to be scaled up to fill the camera's view volume, and **Translate** shows how much it had to be shifted over.

Relative

Relative repositions or rescales the current transformable as a delta value from its current location or scale. You can type in either positive or negative values.

Override

Override disables a module's control over the current object. Some objects in the Geometry Viewer such as the slice planes of the **arbitrary slicer** module, the **volume bounds** object and **isosurface** objects, and the probes of the **probe** and **hedgehog** modules, are being controlled by the module that produces them, not by the Geometry Viewer, through the invisible upstream data connection described in Chapter 3,"Upstream data ports," on page 86. Switching on **Override** for the current object overrides the module's control of the object.

Camera example module

The example module in /usr/avs/examples/camera.c provides yet another way to move Geometry Viewer cameras. This module lets you specify a *from* and a *to* position in world space, giving you direct access to the camera's transformation matrix. *From* defines where the camera is in world space; *to* defines a point in world space that the camera is looking at. You can specify a series of values that will give a flying camera effect. Follow the instructions in the /usr/avs/examples/README file to compile the **camera.c** module.

Bounding Box

Switching on **Bounding Box**, shown in Figure 70, makes transformations (rotate, translate, scale) performed with the mouse behave differently.

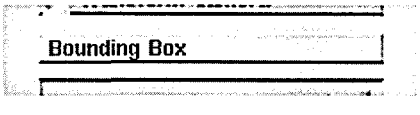


Figure 70
Bounding Box button

Normally, when you transform an object with the mouse, the system does its best to update the rendering of the object continuously, in real time as it tracks the mouse. So, as you rotate a teapot through an arc from A to E, the system tries to update the picture in real-time. In practice, what you get are several intermediate views B, C, D as the teapot rotates from A to E. The more processing power your system has, the smoother and more continuous the rendering. The more complex the scene is (objects with many surfaces, lights), the slower the rendering.

Bounding Box disables this resource-expensive effort at real time rendering. With Bounding Box turned on, when you place the mouse over the current transformable and press the middle or right button, a wireframe box enclosing the volume of the object appears. If the current transformable is a light, the lines representing the light change color. 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 let go of the mouse button. Only then is the object rendered at its new location/scale. It started at A, it ended at E, and positions A and E were all you saw— there were no B, C, D intermediate renderings of the object.

The Bounding Box also provides a more accurate image of the object's orientation in space. Toggling **Bounding Box** affects all view windows on the screen. There is a bounding box for the top object and all objects in the scene.

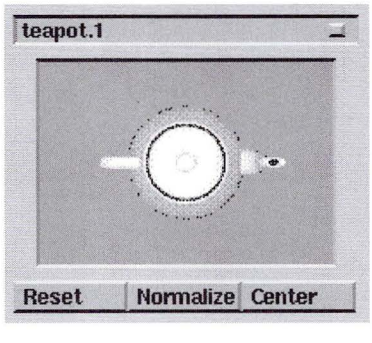
You can have **Bounding Box** turned on by default for all your ConvexAVS sessions. In your .avsrc or .avsrc.X file, add the following line:

```
BoundingBox 1
```

Current object area

The current object area, shown in Figure 71, includes a text widget that displays the name of the currently active object and contains a dimple for bringing up the current object browser, a window for viewing the current object or bounding box, and three control buttons.

Figure 71
Current object selection area



Current object indicator

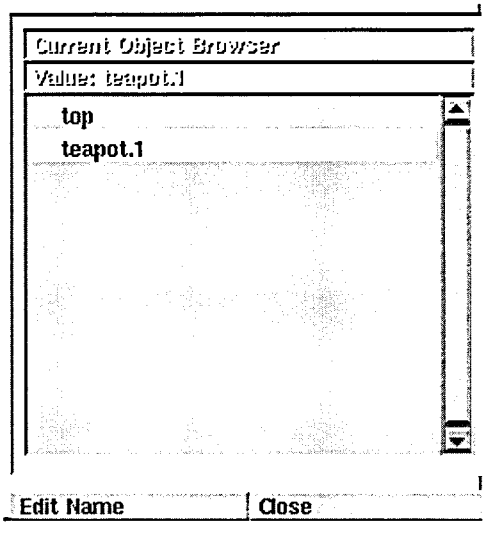
This small window shows a miniature graphical representation of the current object. Clicking any mouse button repeatedly in the Current Object Indicator window cycles through the list of objects for the current scene. This same action is performed by pressing the **F5** function key.

Current object browser

The Current Object name bar shows the name of the current object. Clicking on this dimple produces the Current Object Browser, shown in Figure 72. This is an alternate way to select the current object. It is intended primarily for cases where there are too many objects in a single level hierarchy for it to be reasonable to select them by clicking on them in the display window, or to cycle through them in the miniature Current Object Indicator window.

It also supports current object selection in multi-level object hierarchies.

Figure 72
Current object browser



The names of the objects appear in a browser window similar to the file browser widget. The current object is highlighted. Clicking any mouse button on any object name will make it the current object.

Renaming objects

The Current Object Browser has an **Edit Name** button at the bottom of its panel. Pressing this button brings up a type-in panel that lets you name the current object. Object names are case-sensitive.

When you first read in an object, it comes with a name that appears in the Current Object Name bar. This name was set by the program that created the geom object. By convention, modules name objects after themselves. The Geometry Viewer appends a sequence number to the object name, *name.n*. This is to distinguish between two objects from two modules with the same name, such as two **arbitrary slicer** modules.

There is a good reason to rename objects. The issue arises when you are using ConvexAVS modules that produce geometries and you are just using the Geometry Viewer to examine the results. As the module produces successive geometries, it repeatedly gives them the same name and the Geometry Viewer gives them the same sequence. Each new geometry replaces its predecessor. You will not be able to deal with these geometries as separate objects unless you rename them.

For example, you might be using the **arbitrary slicer** module in the Network Editor to cut several slices at different angles through a 3-D volume of data, and display them composited together.

To create the composite slices, you would position the slice plane at the first desired angle, then freeze it by using the renaming option, creating a new object. Move the slice plane to the second desired angle, freeze it by renaming it, and so on until the whole composite picture has been constructed.

Additional transformations

Just below the Current Object Indicator are three 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).

Center

The **Center** button does *not* center an object within the display viewport. Rather, along with the **Scale/Rot. Center** type-ins on the Transformation Options panel, it allows you to control the *center of rotation* and scaling of individual objects and the entire top level hierarchy of objects.

When an object or a collection of objects within a hierarchy (usually top) is rotated (middle mouse button) or scaled (**SHIFT**-middle mouse button), it is always rotated or scaled about its center.

An object's center defaults to `0, 0, 0` in an object's own coordinate system, but can be overwritten in two ways:

- The module creating the object can use the GEOM(3v) library to explicitly set an object's center (using the `GEOM_edit_window` function).
- The user can type in a new center point from the Transformation Options control panel.

Pressing the **Center** button recalculates the center point for an object or a hierarchy of objects, overriding the previous setting.

For individual objects, the center becomes:

```
center coord = (min extent + max extent)/2
```

For collections of objects in hierarchies, the center point is calculated the same way, but using the minimum and maximum X, Y, and Z values of the smallest rectangular volume enclosing all of the objects within the hierarchy.

The **Reset** button by itself does not reset the center point of objects or hierarchies of objects. You can keep track of centers and reset them manually using the Scale/Rot. Center type-in on the Transformation Options panel described above.

Using function keys (view window is active)

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 from 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 submenus

The Menu Selection part of the Geometry Viewer control panel provides access to most functions for creating 3-D scenes grouped under the following topics:

- Objects
- Lights
- Cameras
- Labels
- Action

Use any mouse button to click on one of these topics to bring up the control panel associated with the particular topic in the lower portion of the control panel.

Objects

Selecting **Objects** causes the Menu Selection area to appear as shown in Figure 73.

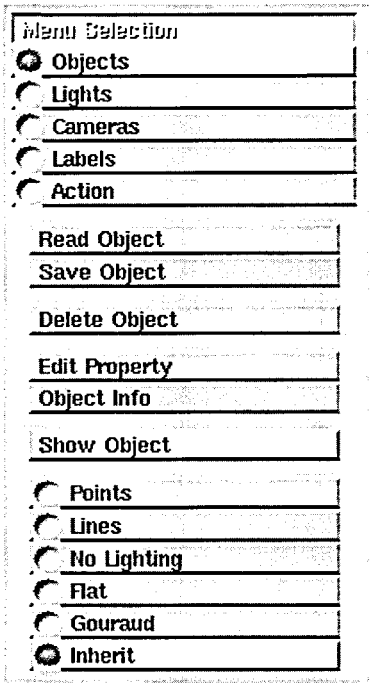


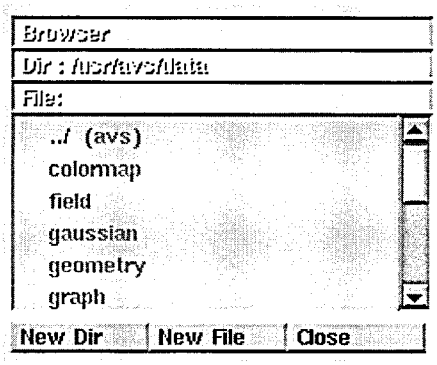
Figure 73
Objects menu selections

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 74
Geometry file browser



The File Browser 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.

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

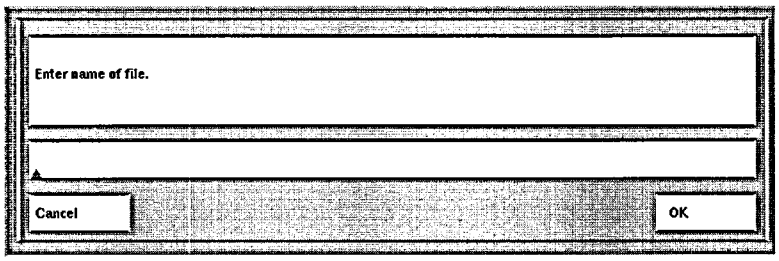
Selecting one of the red (directory) entries changes the working directory. The names of the Geometry Viewer object files in that directory are displayed, along with the names of any subdirectories.

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 of the box 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. See Figure 75. 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 75
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.

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).

Note

You must use Save Scene to save exactly what you see in a view window. Using Save Object saves just the object, its properties, and the position and orientation of the object in space. It does not save light or camera positions.

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 RETURN 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 window using **Read Object**.

ConvexAVS actually uses a special script language to create the object file, as described in Chapter 18, "Geometry Script Language commands," on page 617.

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

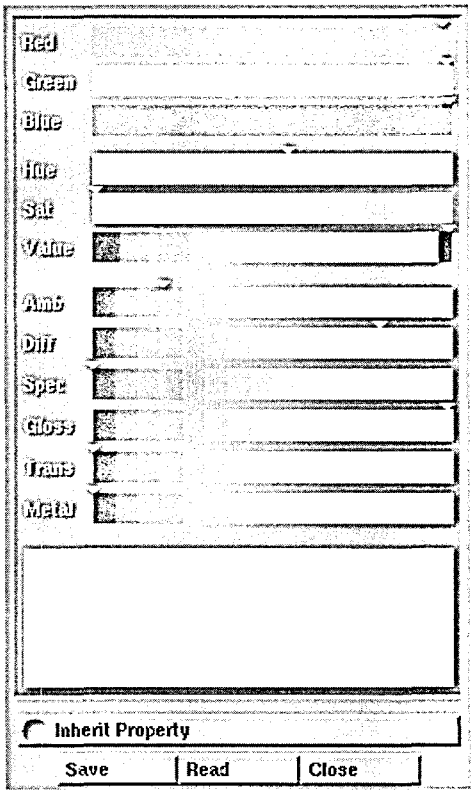
Since there is no way to undo deleting an object, you may want to perform a *Save Object* before deleting something that might be useful later on.

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 (for example, plastic, metal, fabric), smoothness, and so on. 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 surface properties. (see Figure 76).

Figure 76
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 click once at the spot where you want the slider to move.

Like the File Browser, the Edit Property window 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` (these are similar to the tint, color, and brightness controls on a television). White is specified by a zero saturation (slider all the way to the left, value=1). 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.

The hue, saturation and brightness (HSB) color space can be thought of as an inverted cone. The *hue* axis runs circularly around the cone. Example hue values and corresponding hues are given in Table 6.

Table 6
Hue values

Value	=	Color
0.00	=	Red
0.16	=	Yellow
0.33	=	Green
0.50	=	Cyan
0.66	=	Blue
0.83	=	Magenta

- The *saturation* axis runs from the center of the cone (white) to its perimeter (fully saturated color).

Example saturation values are shown in Table 7.

Table 7
Saturation values

Value	=	Saturation
0.00	=	White
0.50	=	Partially saturated hue
1.00	=	Fully saturated hue

- The *brightness* axis runs from the tip of the cone (black) to the base (white). Example brightness values are shown in Table 8.

Table 8
Brightness values

Value	=	Brightness
0.00	=	Black
0.50	=	Partially darkened hue
1.00	=	Full intensity hue

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 **Lights** top-level menu choice.

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 reflected 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).

Note

Plastic surfaces reflect the color of the light. Metal surfaces reflect the color nearest that of the surface.

Transparency (Trans)

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 is a toggle that when enabled, replaces the current slider settings with those of the parent of the current object. When this toggle is disabled, the original object properties are used and displayed. For longer-term storage of properties settings, use the **Save** and **Read** buttons.

Note

The Inherit Property toggle is automatically disabled when you change any property.

Object Information

Clicking the **Object Info** button displays a window of information pertaining to the current object:

- Name of the 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, polyhedrons, labels, and polygons in the object
- Additional object data: vertex normals, vertex colors

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

Show Object

The **Show Object** button is a toggle that changes the visibility of the current object in all windows that show the same scene. Objects are visible when this button is enabled and invisible when it is disabled.

A hidden object is still part of its scene. If you perform a **Save Scene** (described under "Cameras," on page 173), 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.

Note

Points rendering is not available with PEX and GL devices.

Lines

The object is drawn as a wireframe. Whether or not an object has lines depends on how the object was created. For isosurfaces, the **optimize wire** option may need to be selected.

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 polygon has a single shade.

Gouraud Shading

The object is drawn using Gouraud shading. Gouraud shading blends the colors of a polygon's vertices across the polygon face, simulating smooth shading.

Inherit

The **inherit** button is a toggle that, when enabled, causes the current object to inherit the rendering mode of its parent object.

The **inherit** mode is enabled by default. If no rendering method is selected and all levels have **inherit** enabled, then Gouraud shading is used.

Lights

Selecting **Lights** causes the Menu Selection area to appear, shown in Figure 77.

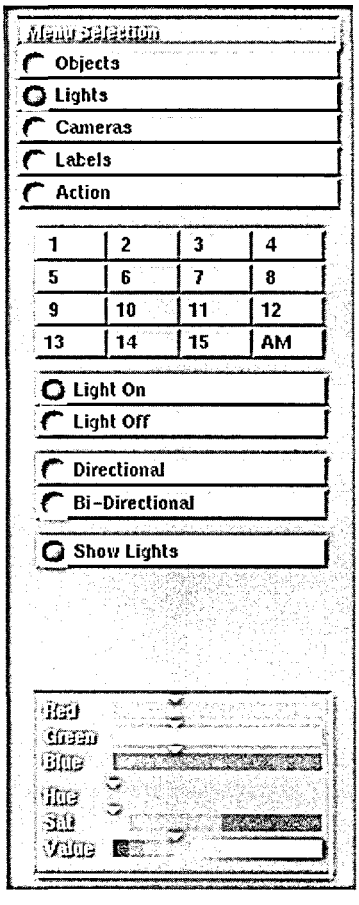


Figure 77
Lights menu selections

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.

Note

Only 7 directional light sources are available with PEX and GL devices.

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.

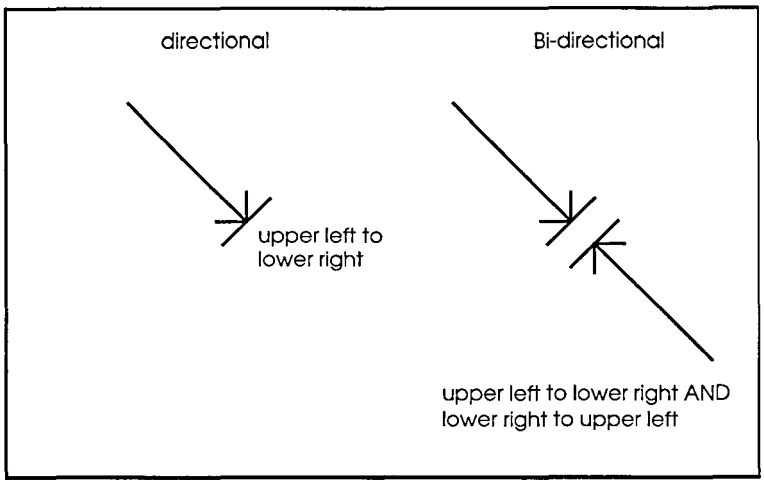
For GL renderer, the limit is n+4.

Show Lights

Displays a symbol for each light source, indicating its position and direction (see Figure 78). 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 78
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

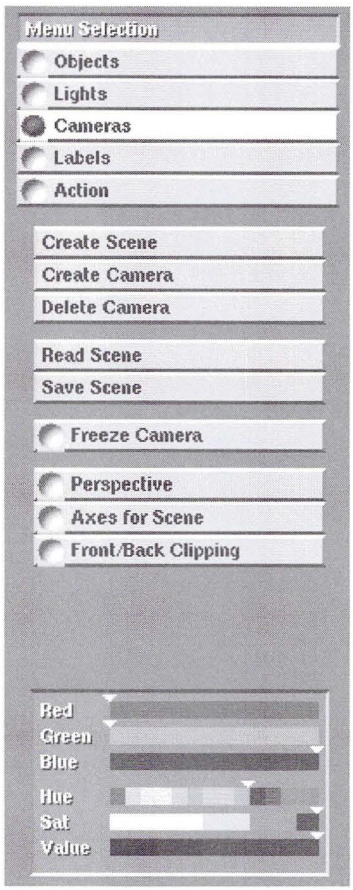
At the bottom of the Lights menu selection area, there are RGB and HSV sliders for setting the color of the light source. See the section, "Edit Property," on page 164, for an explanation of how to use the sliders.

The default color of all lights is white.

Cameras

Selecting **Cameras** causes the Menu Selection area to appear as displayed in Figure 79.

Figure 79
Cameras menu selections



Cameras defined

You look upon a world space through one or more *cameras*. The camera creates a *view* or *view window* on the screen. You can create multiple views of the same world space, by clicking on **Create Camera** under the this submenu. Cameras are initially located at about (0,0,100) in world space, looking toward the origin along the Z axis.

The camera sees a *view volume*. Initially, the camera's view volume sees a rectangular subset of world space extending from +5 to -5 X, +5 to -5 Y, and +100 to -100 Z in world coordinates.

The view volume travels with the camera as it is moved about world space. As such, it has its own coordinate system, sometimes called *camera coordinates*. The camera is always looking from the positive Z axis of camera coordinates toward the negative Z.

Although one can translate the camera, scale the camera (like zooming in and out), and rotate the camera, in some sense this is a misnomer. What is really happening is that world space is being rotated, scaled, and translated in order to fit within the camera's view volume.

You can change the shape of the camera's view volume. By default, the view volume is an oblong box extending from the camera's location to infinity along the Z axis.

- You truncate this box along the camera's Z axis by creating two *clipping* planes. Turn on **Front/Back Clipping** under the Cameras submenu. Objects in front of or behind the two clipping planes will not be rendered in the view window.
- You can change the box from a rectangle to a frustum (a squared-off cone) by turning on **Perspective**. With **Perspective** on, the portions of objects in front of the view volume's Z=0 plane are uniformly exaggerated in size as they approach the camera, and uniformly diminished behind the Z=0 plane. The angle of the side of the frustum is 45 degrees. Perspective is not adjustable from the control panel, but can be affected through the example **camera** module described in "Camera example module," on page 155.

  Perspective projection gives a more real world rendering of objects. You may need to turn on **Perspective** for your eye to be able to interpret a scene.

There is one last point: The camera view volume's Z=0 plane can intersect world space from any angle. This intersection is called *screen space* or *screen coordinates*. When you translate objects and lights using "direct manipulation" movements with the workstation's mouse buttons, the transformables move with respect to the camera's view volume, not world, top, or object-level coordinates. By and large, this is a distinction that you can ignore. You will usually only notice it when you use direct manipulation on objects such as the **arbitrary slicer** module's slice planes or the **probe** module's pointer. When you move the slice plane, for example, in the Z direction, the slice plane moves straight toward or away from the camera (camera's Z axis), not along the object's Z axis.

The Cameras submenu 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.

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.

Note

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 Chapter 18, "Geometry Script Language commands," on page 617. 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 scenes.

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.

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 "Transformations and the Transform Selection area," on page 147.

The degree of perspective exaggeration (45 degrees) is not adjustable from the Geometry Viewer control panel. However, it can be adjusted by using the libgeom library as used in the /usr/avs/examples/camera.c source code example. Perspective also affects the apparent location of the camera. The camera seems to be much closer to the object, and its clipping planes are similarly shifted inwards.

The way to have a camera seem to zoom inside an object is to turn **Perspective** on, then translate the camera or object with the **SHIFT**-middle mouse button.

Axes for Scene

This choice toggles display of X-Y-Z axes in the current scene. All views of the current scene will reflect the change. 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 and back clipping takes place.

The actual location of the clipping planes depends upon how the camera has been scaled, and whether **Perspective** is turned on or not. Clipping is defined in the camera's view volume, not world coordinates.

When a camera is first created, the clipping planes are at Z=100 and Z=-100 in world space when clipping is turned off; and at about Z=5 and Z=-5 in world space when clipping is turned on. If you also turn on **Perspective**, the clipping planes move inward to about Z=10 and Z=-10 with clipping off, and to about Z=3 and Z=3 with clipping turned on. Such statements rapidly lose meaning as you begin to transform objects, the camera, and the top level coordinate system.

Setting clipping *off* puts the near clip plane very near the camera itself, and setting clipping *on* puts it out in front of the camera.

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 "Edit Property," on page 164 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
```

The file format is defined in "Defaults file," on page 625 in Chapter 18.

Labels

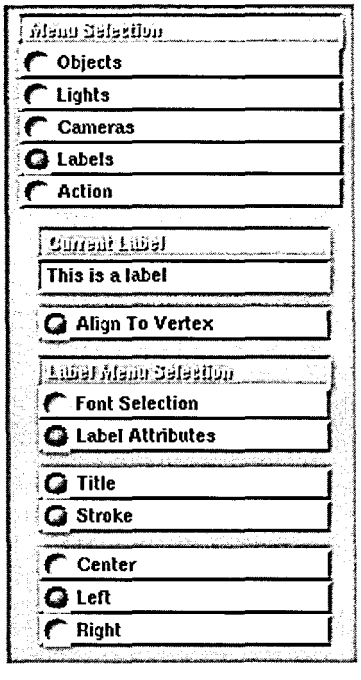
The Labels menu selection provides access to the Geometry Viewer's annotation text facility. You can attach one or more labels to any object. Each label consists of a single line of text. As you manipulate the object— move it, resize it, temporarily hide it, permanently delete it, and so on.— the object's 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 80.

Figure 80
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.

Press **RETURN** when you've finished the label. When you do so, the label appears centered on the current object in a box.

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 section "Transformations and the Transform Selection area," on page 147 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 box reappears 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, shown in Figure 81, lets you select the X Window System font to be used for the label.The submenu for Font Selection may include the following choices:

- Courier
- Helvetica
- New Century
- Times
- Bitstream
- Symbol

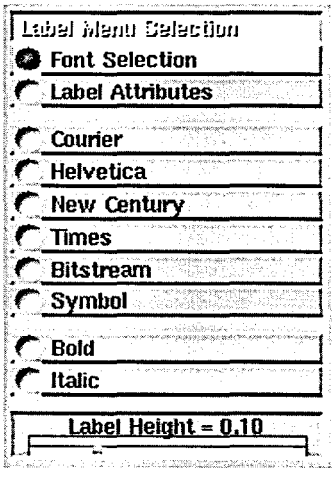


Figure 81
Font selection submenu

Label Height

The Label Height slider 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.

The box 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. You might also try the *xfontsel()* utility.

Label Attributes submenu

The submenu for Label Attributes has the following choices:

Drop Shadow:

If set, the label has a black shadow offset one pixel below and to the right.

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 3-D location. See "Making a label into a title," on page 180.

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.

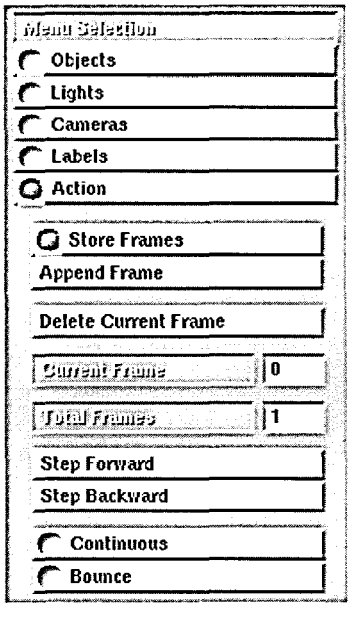
Action

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.

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 objects that are not leafs to create frames.

When you select **Action**, the submenu shown in FIgure 82 appears if the current object has associated cycle information.

Figure 82
Action menu selections



If no cycle information is present, then only the **Store Frames** and **Append Frame** buttons appear.

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.

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.

Caution

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

Special considerations for adding frames

The **Store Frames** option has some restrictions.

To store a new frame for an object, the object with the same name must contain a different geometry. 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 Geometry 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. 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 produces geometries and then invoke the Geometry Viewer and select the **Store Frames** function. Changes to any parameters within the network will cause the refresh to regenerate the geometries with the same name but different data. These will be recorded as a new frame.

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**.

Command Language Interpreter

It is possible to drive the Geometry Viewer with the Command Language Interpreter (CLI) rather than the X display interface. The commands can be either typed in interactively from a terminal emulator window while ConvexAVS is running, they can be read from a script file, or they can be sent from a user-written module.

Note

The Geometry Viewer Command Language Interpreter is not the same thing as the Geometry Script Language.

You can create scripts that animate the Geometry Viewer, not just the network-produced images within it, producing demonstration, illustration and test scripts.

To run ConvexAVS with the Command Language Interpreter (CLI) active, type this:

`avs -cli` *other-options*

This starts ConvexAVS as usual, but also starts the CLI command line interpreter in the invoking window. (You might have to press **RETURN** to get the `avs>` prompt.)

To get a list of the Geometry Viewer CLI commands, type the following:

```
avs> help Geometry
```

This produces a list of Geometry Viewer CLI commands. To get help on an individual commands, type **help** plus the command name:

```
avs> help geom_set_matrix

geom_set_matrix Sets a transformation for an object,
       camera, or light.
Usage: geom_set_matrix {-object<name>}
       {-camera{1-n}}...
                    .
avs>
```

There are no online examples of Geometry Viewer CLI files. However, many of the Help Demo scripts in /usr/avs/demo/man_scripts do contain Geometry Viewer CLI commands.

The Command Language Interpreter and the Geometry Viewer set of CLI commands are documented in detail in the Chapter 17, "Command Language Interpreter."

Data storage 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 3-D space, along with additional information (typically, indicating connectivity). The geometry defines a simple or complex 3-D 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 (for example, 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 3-D 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 Geometry 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 a two-level hierarchy— a top-level object with a number of objects as its children. The Script Language gives you more flexibility. A script can define multiple-level hierarchies, using the **group** command. Refer tothe group command on page 619 in Chapter 17.

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 Geometry 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 Geometry 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.

File type summary

Table 9 summarizes the types of files that the Geometry Viewer can read directly.

Table 9
Geometry Viewer file types

File type	Extension	Format	Contents
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).

Image Viewer subsystem

5

Introduction

The ConvexAVS Image Viewer subsystem is an interactive tool for displaying, manipulating, and processing images.

The Image Viewer exists in two forms: the Image Viewer subsystem accessible from the main ConvexAVS menu and the **image viewer** module in the Network Editor's module palette.

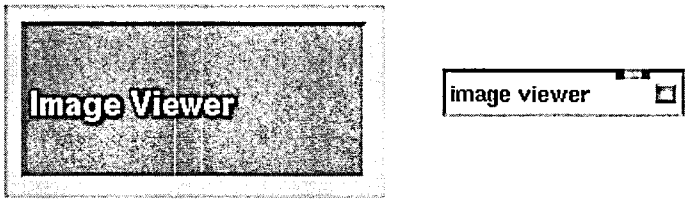


Figure 83
Image Viewer button and module widget

The Image Viewer performs these functions:

- **Image display manager**—The Image Viewer manages a window full of ConvexAVS images in the same way that a window system such as the X Window System manages windows. You can create, delete, move, resize, raise and lower images of different sizes which overlap or are stacked one above the other.

  The images can come from two sources. You can read them in directly from disk with the Image Viewer's **Read Image** button, or they can flow into the **image viewer** module from a ConvexAVS network.

- **Image processor**—The Image Viewer's **Image Processing** button calls up a sample choice of networks that implement image processing techniques such as edge detection, and contrast stretching. These techniques can be applied to whole images, or to interactively defined parts of images called *subimages*. Subimages are sometimes referred to as Regions of Interest (ROI). The results can be viewed then erased, or made a permanent part of the output image.

You can apply image processing techniques serially to the same image in place. You can interpolate, then manipulate contrast, without writing intermediate images to disk and reading them back in again between each step.

You are not restricted to the sample image processing networks supplied. You can create your own networks, save them, and then call them up through the Image Viewer's interface.

Supporting these two basic functions are these additional features:

- **Views**—Sets of images are collected together in a *scene*. There can be multiple scenes, that is, there can be multiple sets of images on the screen at once.

  Each scene can have multiple *viewports*. A viewport is a window displaying the same set of images in the same configuration, but from a different point of view.

  Configurations of images in scenes can be saved to disk, then read in again at a later session.

- **Subimages**—A rectangular view of an image may be selected for image processing.

- **Labels**—Whole scenes and individual images can have alphanumeric *labels* attached to them in various font styles, sizes, and colors.

- **Action animation**—The *Action* submenu implements a form of flipbook animation. A sequence of images flowing into the Image Viewer from the network can be collected into a cycle of images that can be replayed at a controlled speed. The Image Viewer's Action flipbook animation differs from that of the **display pixmap** module's pseudo-animation controls in one very important regard—you can save the cycle of images to disk and replay them at a later session. A more sophisticated method of animation is available through the AVS Animation Application.

- **Command Language Interpreter**—Most of the Image Viewer's functions can be driven from a command file through the ConvexAVS Command Language Interpreter (CLI).

All ConvexAVS data that is being displayed by a network in a window exists as either a ConvexAVS *geometry*, a ConvexAVS *pixmap*, or a ConvexAVS *image*.

This is true whether they are:

- Geometries being displayed with **render geometry** and **display pixmap**
- Colorized volume, vector field, or unstructured cell data being converted to geometries by modules such as the **arbitrary slicer** or **stream lines** modules
- Graph data going through the **graph viewer** module
- 3-D volumes being rendered as an image with **tracer**

If it exists as a geometry, the **render geometry** module can convert it to a pixmap or a ConvexAVS image. Once it is an image, it can be displayed, manipulated, processed, and animated with the Image Viewer and AVS Animator.

Entering the Image Viewer

The Image Viewer can be entered in four ways:

From the shell directly

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

```
avs -image
```

When you start ConvexAVS and the Image Viewer in this fashion, you cannot transfer to any other ConvexAVS subsystem. See "Command-line options," on page 37 for additional command line options that affect the way the Image Viewer is invoked.

From the main menu

You can start the Image Viewer from the ConvexAVS main menu. It is the first choice. (Note that under the "AVS Applications" selection on the ConvexAVS main menu, there is an Image Viewer application. This is a sample imaging application that is not described here.

From another subsystem

At the top of each of the four major ConvexAVS subsystem control panels (Image Viewer, Graph Viewer, Geometry Viewer, Network Editor) is a button titled **Data Viewers**. Position the mouse cursor over **Data Viewers**, then press and hold down any mouse button. A pop-up menu appears. Drag the cursor down the pop-up menu until Image Viewer is highlighted, then release the mouse button. This calls up the Image Viewer's control panel. If you transfer to the other subsystems, then return to the Image Viewer, the Image Viewer's control panel will remain in the state that you left it.

In a network

You can include the **image viewer** module in a ConvexAVS network. If you click on the **image viewer** module's dimple with the left mouse button, it calls up the Image Viewer control panel.

There are some important distinctions between the way the Image Viewer handles images it receives through its own **Read Image** menu button, and images that flow into it from a network with the **image viewer** module:

- Images received from a network are titled differently.
- The Image Viewer Action submenu can only create animations of images that it receives through a network.

Using the **image viewer** module as a general rendering utility to display data images and pixmaps produced by networks is as powerful and flexible as using the **render geometry** module to display geometries. It is a more powerful alternative to the **display image** module.

Objects that are always represented in ConvexAVS X image format. See "Read Image," on page 206 for a discussion of image format.

There are sample ConvexAVS image format files in the /usr/avs/data/image directory.

Leaving the Image Viewer

If the Image Viewer was invoked from the shell command line as `avs -image`, then at the top of its main control panel will be a button labeled **Exit**. Click on **Exit** with any mouse button to return to the UNIX shell.

If the Image Viewer was entered from the main ConvexAVS menu or through the Data Viewers pop-up menu from another subsystem, then there will be a **Close** button at the top of its main control panel. **Close** is not really an exit button. **Close** removes the the Image Viewer's control panel from view; similar to using the window manager to iconify the control panel. When you later re-enter the Image Viewer, the control panel is in the same state that you left it. To exit the Image Viewer, you must exit ConvexAVS from the main menu.

If the Image Viewer was invoked as the **image viewer** module, then there is still only one Image Viewer, even though there may be multiple **image viewer** modules. The modules are associated with individual Image Viewer scene windows, not with the Image Viewer itself. Throwing away an **image viewer** module deletes its associated scene window.

Basic layout

The Image Viewer has three types of windows:

- The main Image Viewer control panel
- The viewports. These are display windows that contain sets of images.
- Various pop-up browser windows and type-in panels that are used for file input, selecting image processing techniques, designating the current image, and specifying input and output files.

Image Viewer control panel

Figure 84 shows the main Image Viewer control panel. Use any mouse button to click on and select any of the control buttons.

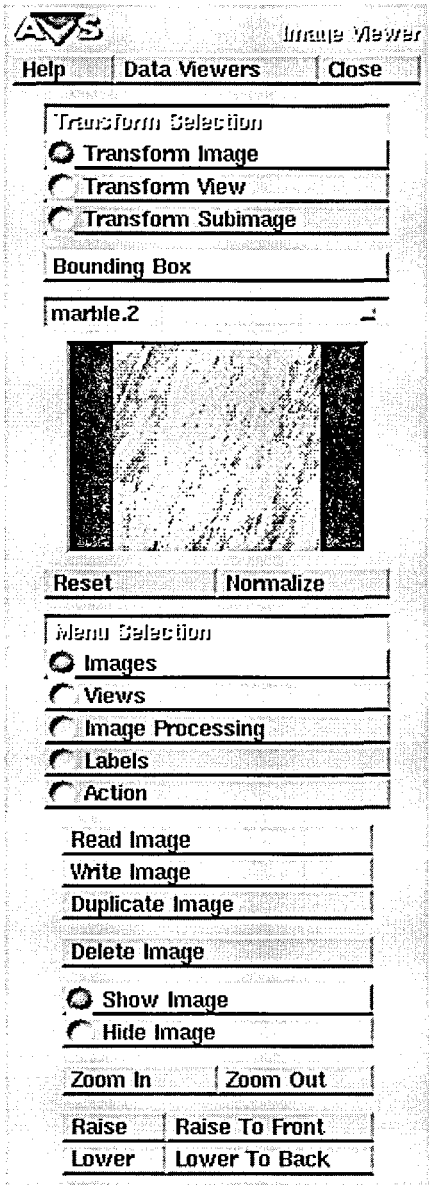


Figure 84
Image Viewer control panel

The following sections discuss each of the major areas of the main control panel.

Figure 85
Image Viewer top control bar

Figure 85 shows the top control bar.

Help

Click on **Help** using any mouse button to invoke the ConvexAVS Help Browser window. By default, the Help Browser displays topics specific to the Image Viewer. The Help Browser is described in detail in "Using online help," on page 56 in Chapter 2, "Starting ConvexAVS."

Data Viewers

Pressing any mouse button over **Data Viewers** brings up a pop-up menu that switches among the three ConvexAVS viewers (Image Viewer, Graph Viewer, and Geometry Viewer). Position the mouse cursor over **Data Viewers**, then press and hold down any mouse button. A pop-up menu appears. Drag the cursor down the pop-up menu until Image Viewer is highlighted, then release the mouse button.

Switching to another subsystem does not exit or halt the Image Viewer, it just brings up the new viewer's control panel on top of the Image Viewer control panel.

Close

Click on the **Close** button to remove the Image Viewer's control panel from the screen. It does not remove any Image Viewer viewport or browser windows, nor does it exit the Image Viewer. The effect is as though one had iconified the control panel.

You do not need to close the Image Viewer control panel to make room for other subsystems' control panels. Just use your window manager to move it to another part of the screen.

Transform Selection controls

Figure 86 shows the Transform Selection menu. There are three kinds of things that you can move around in the Image Viewer using mouse button controls:

- Images within viewports
- The viewport's position over the scene of images (like moving a frame over a stationary picture)
- The rubber-banded subimage region over an image

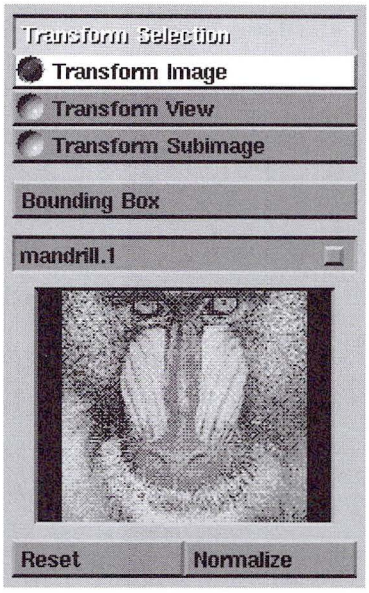


Figure 86
Transform Selection menu

The three Transform Selection buttons select which of the three objects (image, viewport, subimage) is going to be moved by a mouse button command.

Transform Image

With **Transform Image** selected, the right mouse button moves the current image around within the window. The **SHIFT**-middle mouse button combination scales the current image, making it larger or smaller.

Transform View

With **Transform View** selected, the right mouse button moves the current viewport over the fixed set of images—shifting the point of view rather than the objects. The **SHIFT**-middle mouse button moves the point of view in towards the images, or away from the images.

Transform Subimage

To define a subimage, press and hold down **SHIFT**-left mouse button. Drag the mouse cursor to define the subimage region, then release the mouse button. Subimages can be defined at any time; **Transform Subimage** does not have to be selected.

The right mouse button will move the rubber-banded subimage region around over the images. **SHIFT**-middle mouse button does not, however, resize the subimage region. To resize the subimage, select a new subimage region.

Bounding box

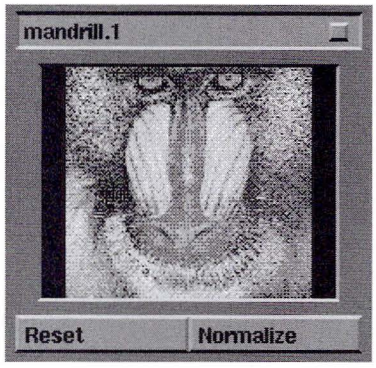
The **Bounding Box** toggle switch changes the way images, viewports, and subimages move with the mouse.

Normally, when you move or resize an image, viewport, or subimage, the system updates the rendering of the picture continuously as it tracks the mouse. As you move an image from point A to point E, the system tries to keep the picture live. In practice, what you get are several intermediate pictures, B, C, D, as the image moves from A to E. The actual redraw rate depends on the speed of your system, the type of workstation hardware you are using, and the number and size of other processes being run on the machine. In a related way, the more complex the image sets, the slower the image display.

The Bounding Box avoids this resource-intensive effort when displaying the changing image. With **Bounding Box** turned on, when you place the mouse over the window and press the right or **SHIFT**-middle mouse button, a white wireframe box enclosing the area of the image, viewport, or subimage appears. As you hold the button down and move the mouse, the bounding wireframe box moves—the image, viewport, or subimage does not. You move the bounding box to the destination position, then let go of the mouse button. Only then is the image, viewport, or subimage rendered at its new position.

Toggling **Bounding Box** affects all images, viewports, and subimages on the screen.

Figure 87
Current Image controls



The next set of buttons and the miniature image window control operations on the current image. See Figure 87 above. You can have many images existing in many scenes and many viewports. When you start performing image processing techniques, or moving images around with the mouse buttons, or raising and lowering images with respect to one another in a scene, one image must be the object of the command.

Naming images

The Image title bar shows the name of the current image.

Each image that enters the Image Viewer gets a name. The name has two parts, a name and a sequence number:

name.sequence_number

If the **Read Image** button is selected to read in an image directly from a disk file, the image *name* will be the same as its disk file name minus the extension. If the image came into the Image Viewer from a ConvexAVS network, its name will be the same as the name of the ConvexAVS module that sent it to the Image Viewer (for example, **crop**).

Sequence_number (.1, .2, .3, ...) is assigned sequentially, according to the order the images entered the Image Viewer.

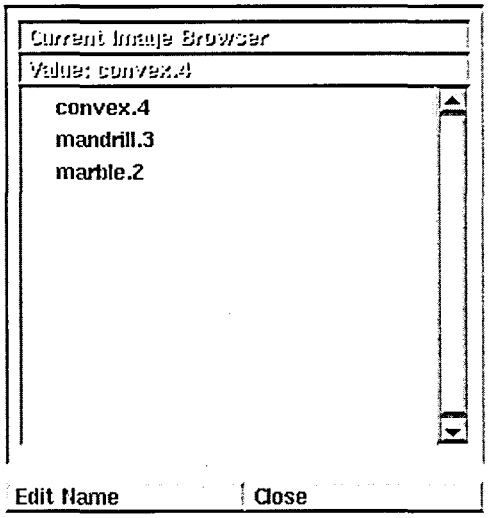
The Image Viewer has no concept of top image as the ConvexAVS Geometry Viewer has (see "Objects," on page 142). Images are not organized into hierarchies, nor do Image Viewer scenes and viewports get names.

Retitling and picking images

The Image Title Bar, shown at the top of Figure 87, also provides an alternate way to select the current image. It is useful when there are large numbers of images, some of which may be in obscured windows not easily designated with the mouse cursor.

Use any mouse button to click on the dimple at the right of the Current Image title bar. This produces a Current Image Browser window, shown in Figure 88.

Figure 88
Current Image browser



Like all ConvexAVS browsers, you pick an image by highlighting its name with the mouse cursor and clicking any mouse button. The Current Image Browser window is *sticky*, that is, it stays on the screen until removed by clicking on its **Close** button. Closing the browser window reduces it to the icon form. If you select it again, it returns in the same state as you left it.

The Current Image Browser window also renames images. Click on the browser's **Edit Name** button. A new window pops up with a type-in area for the new image name. The mouse cursor must be moved inside the type-in area or anything you type is ignored. For simple editing functions, the **BACKSPACE** key deletes the previous character, and **CTRL-U** erases the whole line.

Note

Do not retitle images if you intend to save the network containing the image viewer module. When the network is read in again, it will not be able to find the retitled image.

Current Image window

The Current Image window displays a miniature picture of the current image.

You can also use the Current Image window to select the current image. With the mouse cursor in the Current Image window, start clicking any mouse button. This action cycles through the images in the currently-selected scene.

Reset/Normalize

The **Reset** button causes the current image to revert back to the same size and same position it had when the image object first entered the scene, undoing any translations or scaling changes that might have been done to the image. It does not remove Image Processing techniques performed on the image, even those not made permanent with **Set Current Image**, nor does it reset an image's name. **Reset** only works on individual images and views. Transform subimages are not affected by this operation.

The **Normalize** button expands the size of an image until the larger of its two dimensions fills the current viewport. This works even if the viewport window has been resized. If an image is offset within the viewport window, it is enlarged to the same degree it would have been if it were centered in the window, but its relative position within the viewport is not altered. The **Normalize** button is disabled when **Transform View** or **Transform Subimage** is selected.

Menu selection controls

This area of the main control panel contains five buttons that select among the five Image Viewer submenus (Figure 89). This is the main Image Viewer menu. The default is **Images**.

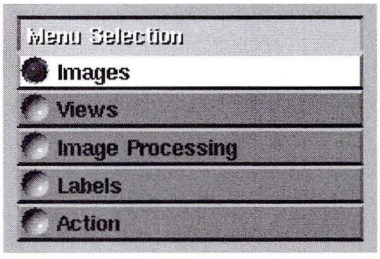


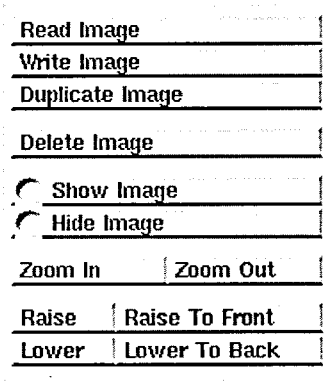
Figure 89
Image Viewer menu buttons

Submenu controls

This part of the main control panel changes when you select among **Images, Views, Image Processing, Labels,** and **Action.** Unique submenus containing individual controls appear in this area. Figure 90 shows the submenu for the Images selection. Buttons in this submenu area are indented, indicating their sub-level status. Some sub-level menus have their own submenus that also appear in this area.

When the Image Viewer first invoked, the default Images submenu is displayed. As you move among these submenus, their state is saved for the next time you enter them.

Figure 90
Images button submenu controls



The state of these submenus may also differ among scenes and images. For example, in one scene an image may be hidden while in another it is visible (depending on the state of the **Show Image** and **Hide Image** buttons). The correct state of the controls is saved for each image and scene; as you change the current image or scene, the controls change to reflect the correct settings.

Image Viewer *Scenes* are sets of images, of various sizes and positions, with a particular stacking order, that is, A is above B, which are both above C.

Scenes, in themselves, never appear on the screen. What you see on the screen is one or more viewport windows looking onto a scene.

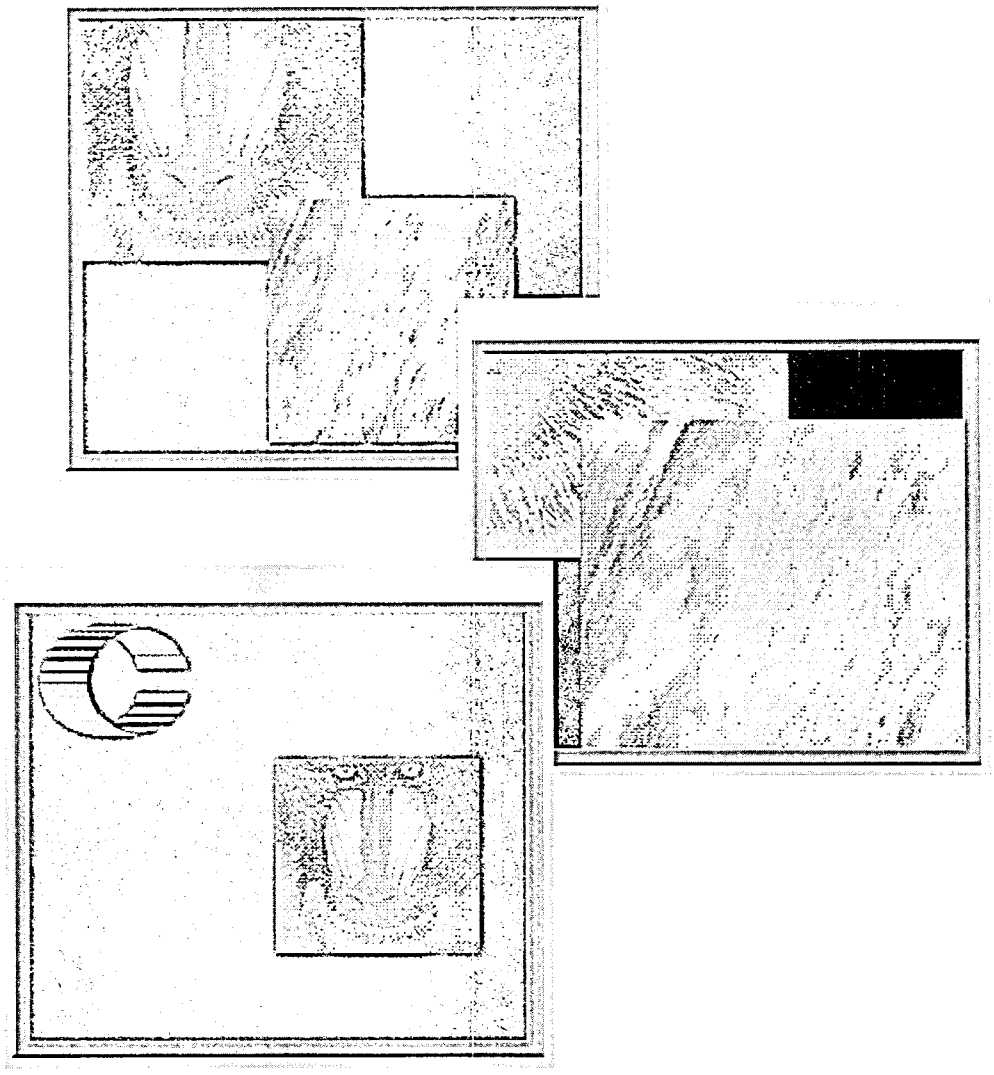
In Figure 91, the top two windows are two viewports onto the same scene, while the bottom window is a viewport onto a different scene. The bottom viewport is the current viewport, which is designated by a red border.

Transforming viewports

Viewport windows move in two dimensions. To move a viewport, Transform View must be toggled at the top of the Image Viewer main control panel. The transformations that work on viewports are as follows:

- **Right mouse button**—Moves the viewport over the collection of images. This is like moving a frame over a collage of pictures.
- **SHIFT-middle mouse button**—Moves the viewport in and out to and from the collection of images. The effect is similar to zooming in and out from the scene.

Figure 91 Viewport windows



Switching among viewports

Just as there is a current image that is the object image of any command so there is a current viewport that is the object of any viewport command. The current viewport is surrounded with a red border.

Click any mouse button anywhere in any viewport window to make it the current viewport.

Resizing viewports

You resize a viewport window using whatever X Window System window manager you have running, just like any other X window.

When you resize a viewport window, the sizes and relationships among the images inside the viewport do not change. As a window gets larger, more background is displayed.

Note

The Image Viewer tries to keep at least part of all the images in a scene visible in the viewport window by shifting the viewport slightly, but it will abandon this strategy and clip images if the size of the images becomes too great to be retained in the window.

Browsers

The Read Image, Read Scene, Select Processing Technique, and the Current Image Title bar all bring up ConvexAVS browsers. The browsers remain on the screen until they are removed with their **Close** button. They maintain their current settings. To pick an item from the browser, highlight it with the mouse cursor then click any mouse button. If a list is too long to fit on one page of the browser, use the scroll bar.

Each browser's use is discussed under the menu button that invokes it.

The remainder of this chapter discusses each of the Image Viewer submenus and the Image Viewer Command Line Interpreter.

Images submenu

The Images submenu shown in Figure 92 provides utilities for manipulating individual images.

Activate the functions by positioning the mouse cursor over the button until it is highlighted, then click any mouse button. The buttons always act upon the current image or scene. The current scene is the one that has one of its viewports surrounded with a red border.

To select the current scene, move the mouse cursor until it is in one of the viewports looking upon the scene, then click any mouse button.

Select the current image in one of three ways:

- Position the mouse cursor over the image in a viewport that you want to become the current image, then click the left mouse button.
- With the mouse cursor over the current image window on the Image Viewer control panel, click any mouse button. The Current Image window cycles through the images in the current scene.
- Click any mouse button over the button at the right of the Current Image Title bar. A browser appears listing all the image names in the current scene. Move the mouse cursor down the list until the image name that you want is highlighted, and click any mouse button.

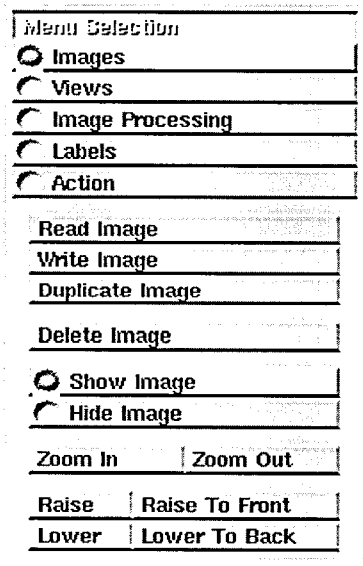


Figure 92
Images submenu controls

As a by-product of selecting a current image, you are also setting the current scene to be the scene of which the image is a member.

Read Image

Click on **Read Image** to read an image directly into the current scene. The image file must be in ConvexAVS image file format. Read Image produces a file browser. Which directory the file browser will be showing is controlled by three things, in this order of precedence:

- If you invoked ConvexAVS with the `-data` *directory_name* option, then the file browser will come up displaying the contents of *directory_name*.
- If your .avsrc file contains a DataDirectory specification, then the file browser will come up displaying the contents of the data directory.
- If no other specifications are made, the file browser displays the contents of the /usr/avs/data directory. Sample image-format files supplied with ConvexAVS can be found in the image subdirectory.

The file browser uses the following conventions:

- Directory names appear in red.
- File names with the .x image file suffix appear in black.
- File names with the .ims Image Viewer scene file suffix appear in black.

All other files are invisible. If you want to read in an image file that does not have the .x or .ims suffix, you must click on the file browser's **New File** button and type out the full file name explicitly.

The file browser window stays up on the screen until you click on its **Close** button. If you close the file browser window, then open it again by clicking **Read Image**, it will once again display the contents of the directory chosen according to the precedence list above.

Write Image

Write Image writes a copy of the current image into a file in ConvexAVS image file format. In the absence of any `-data` command line option or .avsrc file DataDirectory specification, the Image Viewer will try to write the file into the /usr/avs/data directory. This is a system-owned directory that may not permit users to write to it. Instead, move the mouse cursor into the type-in window and type a full directory/file name specification for a directory that you have write access to. When in the type-in window, **CTRL-U** deletes the entire line, and **BACKSPACE** deletes the previous character. The Image Viewer will automatically append the .x image file suffix. Finish the type-in by pressing **ENTER** or by clicking the **OK** button.

When you write the current image, you are saving it as you see it on the screen.

Duplicate Image

The **Duplicate Image** button makes a copy of the current image within the same scene. It does not copy images between scenes. (To do this, first select **Write Image** from one scene, change current scenes, then use **Read Image** to get it into a new scene). The new image will have the same name as the original image, but a higher sequence number. It will be centered within the scene, rather than the viewport.

Delete Image

This button deletes the current image from the current scene. *There is no undo function.*

Show Image
Hide Image

By switching between the show and hide image modes, the current image can be made visible or invisible within a scene. Though invisible, an image can still be manipulated just as though it were still visible.

Zoom In
Zoom Out

Zoom In and Zoom Out buttons provide a separate way of resizing images when you want to preserve the original image data.

Zoom In makes an image larger by integral values; 2x's, 3x's, 4x's, and so on, from its original size. **Zoom Out** makes a zoomed in image smaller again by the same fixed multiples. Figure 93 shows the eye section of the mandrill.x image enlarged to great size.

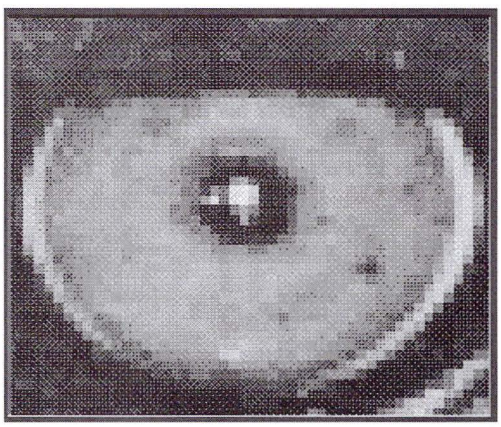


Figure 93
Zoom In image showing pixel replication

Zooming is similar to changing the size of the image with the **SHIFT**-middle mouse button, but with these important differences:

- Both Zoom In and Zoom Out and rescaling with the **SHIFT**-middle mouse button redraw the image using pixel replication. The difference is that Zoom In works by taking one pixel in the original image and making it into four pixels (replicating it) in the new image, enlarging the area of the image by four. Zoom Out takes the four equal pixels in an image that was made larger with Zoom In and turns them back into one pixel. Zoom In/Out are inverse operations that resize an image by multiples of four. On-screen image fidelity is always preserved.

  On the other hand, when you rescale an image with the **SHIFT**-middle mouse button, the result does not have to be an even multiple of the original image size. The picture might be 30% larger or smaller. In this style of rescaling, an algorithm is used to select 30% of the pixels in each dimension of the image, and replicate (or delete) only them.

The picture gets bigger or smaller, but absolute image fidelity on the screen is sacrificed. Because the Image Viewer keeps a copy of the original image, pixel data is not lost if you reduce an image, then enlarge it again. You will only see the effect if you use **Write Image** on a highly-reduced image, then read it back with **Read Image**.

- Because **Zoom In** and **Zoom Out** follow a precise approach to enlarging and shrinking images, you cannot use Zoom Out (make it smaller) unless you have first used Zoom In to make it larger.

  Nothing happens if you click **Zoom In** on an image that has been rescaled to be smaller than its original size with the **SHIFT**-middle mouse button. Image data would be lost.

  Similarly, nothing happens if you click **Zoom In** on an image that has been rescaled larger or smaller in just the X or Y direction. Again, image fidelity would be compromised.

  For all the same reasons, you cannot zoom in on an image, then use **SHIFT**-middle rescaling, then try to use **Zoom Out**. When you click on the latter, nothing will happen.

Image stacking order—Using Raise and Lower

Multiple images in scenes have a stacking order in the same way that windows on a display screen have a stacking order. Though the images might not even overlap, still one image is on top, one is on the bottom, and any additional images are in some order in the middle.

Raise raises the current image one level in the stacking order.

Lower drops the current image one level in the stacking order.

If you have three images in a stacking order A, B, C, and C is the current image, clicking **Raise** will change the display and stacking order to A, C, B.

Raise to Front and Lower to Back

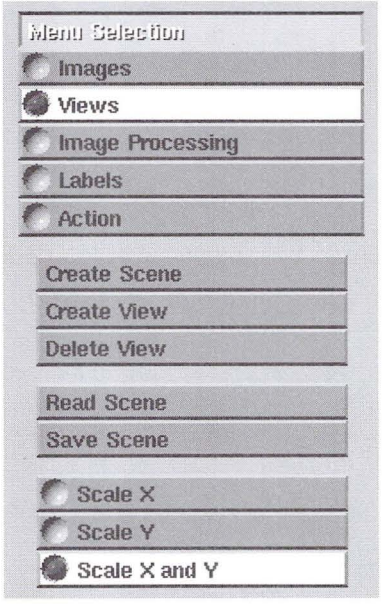
Raise to Front takes the current image, without regard to where it is in the stacking order, and makes it the top image.

Lower to Back makes the current image the bottom image.

Views submenu

The Views submenu (Figure 94) provides utilities for manipulating scenes and viewports.

Figure 94
Views submenu



Create Scene

Two or more independent sets of images can be manipulated on the screen at the same time by creating multiple scenes. Click on **Create Scene** to create a new, empty scene without any images. The Image Viewer also creates a new viewport onto the empty scene. The new viewport has the red border that shows it has become the current viewport, and its empty scene becomes the new current scene.

Note

When you are in the ConvexAVS Network Editor and you drag an image viewer module down from the Data Output column in the palette into the Network Editor workspace, you have done the equivalent of a Create Scene.

Create View

Click on **Create View** to create a new viewport window onto the current scene. The new viewport becomes the current viewport, with its red border. The new window contains the same set of images as the viewport you were in when you clicked on **Create View**. It uses the default Image Viewer viewport size.

This new viewport always shows exactly what the older viewport displays—images moved in one will move in the other, and so on.—unless you toggle **Transform View** at the top of the Image Viewer control panel. Then, movements you make (right mouse button or **SHIFT**-middle mouse button) control the viewport's position over the scene, not the contents of the scene itself.

Delete View and deleting scenes

Click on **Delete View** to delete the current red-bordered viewport. If that viewport is the last viewport onto a scene, it will also delete that scene.

If there are other viewports on the screen, one of them becomes the red-bordered current viewport and scene in the reverse order that they were originally created.

Save Scene

Click on **Save Scene** to save the current state of a scene into a file that can be read back at a later session. It is the way to save a snapshot of your work.

Clicking on **Save Scene** puts a file name type-in window up on the screen. By default, the scene file is saved in the directory specified by the `-data` *directory_name* option on the ConvexAVS command line, or by the directory defined as the DataDirectory in your .avsrc file.

Note

You must specify a directory that you have write permissions to. The default directory (/usr/avs/data) **is normally set to read-only.**

To change the directory, type in a complete file specification including the directory path.

Scene files should end with the file suffix .ims. This suffix is automatically appended to the file name.

The following information is saved from a scene:

- File references to all the images that have been read into a scene using Read Image. The image itself is not saved, just the full file name from where it was originally read.
- All viewports associated with the scene, their background color, and any transformation to the viewport's position over the set of images. The position of the viewports on the display screen is also saved.
- Any transformation that has occurred to the images themselves, including their position within the scene, and any rescaling.
- All scene and image labels, their sizes, fonts, colors, and positions.

Save Scene does not save:

- Any image processing technique that was performed on an image, even those made permanent with the Image Processing submenu's Set Current Image command. (To save a processed image, use Save Image.)
- Any image that entered an **image viewer** module from a ConvexAVS network.

The general reason in both cases is that the .ims file contains references to image files to read in, and image-processed images and images that have arrived through a ConvexAVS network have no file associated with them. .ims scene files are ASCII files, written in the Image Viewer's set of Command Language Interpreter (CLI) commands.

Read Scene

Click on **Read Scene** to read a .ims scene file into the Image Viewer. It works just like the other file browsers in the Image Viewer.

Note

The file browser will not show scene files unless they end with the .ims file suffix. To read in a scene file that has no .ims file suffix, click on New File and type in its name.

Scale X, Y, X and Y

A set of radio buttons modifies how the **SHIFT**-middle mouse button rescales images. It does not affect scenes or viewports. By default, **Scale X and Y** is selected. This just means that when you resize an image with the **SHIFT**-middle mouse button, it gets bigger or smaller in both the X and Y direction evenly.

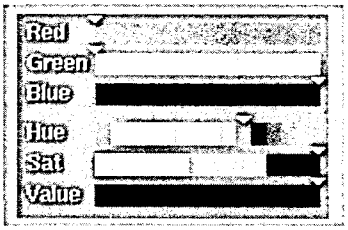
If **Scale X** is chosen, then the **SHIFT**-middle mouse button changes the size of an image only in the Y direction, making it wider or narrower. If **Scale Y** is toggled, then **SHIFT**-middle mouse button makes the image taller or squatter in the Y direction alone, shown in Figure 95.

Figure 95
Image scaled in the Y direction only

Viewport background color

The set of colormap controls at the bottom of the Views submenu (Figure 96) establishes the background color of the current viewport.

Figure 96
Background color controls



There are two ways to set the color; either using the Red Green Blue color model, or the Hue Saturation Value model. If you change one set of controls the other set moves accordingly. The default background viewport color is black.

Image Processing submenu

The Image Processing submenu shown in Figure 97 contains the facilities for performing image processing techniques upon images.

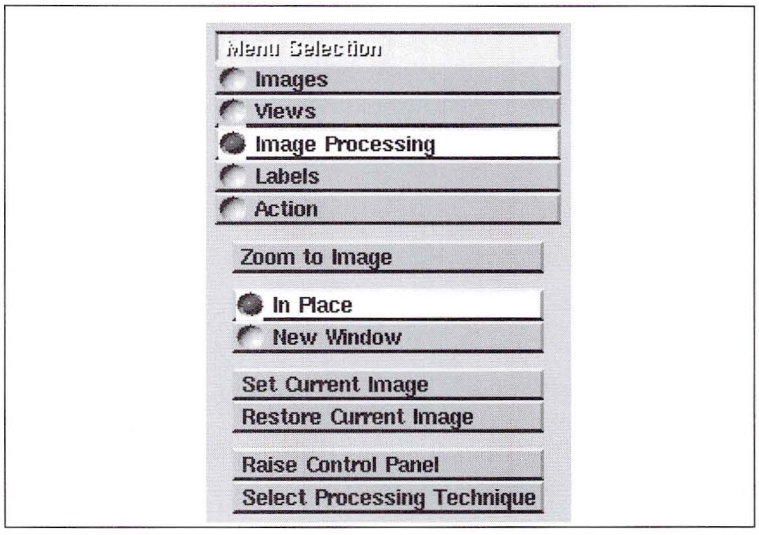


Figure 97
Image Processing submenu

You can apply the technique to an entire image (**Zoom to Image**), or to a movable, rubber-banded region of interest of the image called a subimage (**SHIFT**-left mouse button). The techniques can be applied experimentally to an image or subimage, then erased by using **Restore Current Image**, or the change can be made a permanent part of the Image Viewer's working copy of the image by using **Set Current Image**. Techniques can also be applied serially to the images and subimages. For example, you can **contrast stretch** an image, then use **edge detect** upon the modified version of the image. The results of the technique can be displayed on the original image (**In Place**), or made to appear in a separate scratch scene (**New Window**).

Any image, whether it entered the Image Viewer from a file through Read Image or from a ConvexAVS network, can have the techniques performed on it.

The image processing techniques (**Select Processing Technique**) are pre-defined ConvexAVS networks containing ConvexAVS modules. The Image Viewer comes with a set of sample networks. You can also define and use your own image processing networks. This is described in "Defining image processing techniques," on page 224.

Selecting processing techniques

This section describes how to select and apply image processing techniques. To see the networks that implement the techniques, perform the following actions:

1. Select the ConvexAVS Network Editor from the main ConvexAVS menu
2. Enter the Image Viewer by clicking **Data Viewers** at the top of the left Network Editor control panel and select Image Viewer. The large Network Editor window remains on the screen, while the Image Viewer control panel appears on the left.

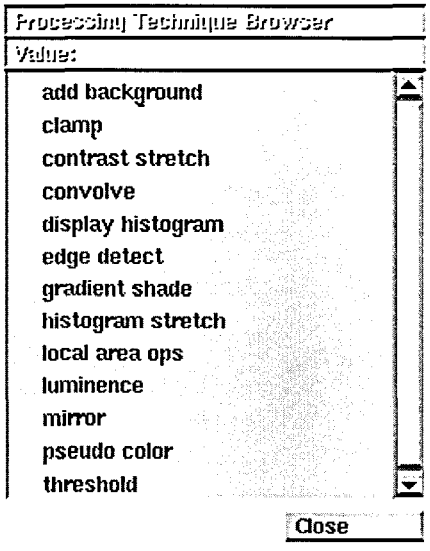


Figure 98
Processing technique browser

3. Using either **Read Image** or a network with the **image viewer** module at the bottom, get an image or images into the Image Viewer viewport.
4. Click on **Image Processing** to bring up the correct submenu.
5. Click on **Select Processing Techniques.** This brings up the Processing Technique Browser, shown in Figure 98. Techniques are listed alphabetically, usually by their core module name. A scroll bar at the right of the browser provides access to the rest of the techniques.

If you are viewing the process in the Network Editor, two modules appear in the workspace: **IV read image** and **IV write image**. These are special internal modules that retrieve images from an Image Viewer scene (**IV read image**), and return them to the Image Viewer (**IV write image**).

6. Drag the mouse cursor down the technique browser and click on any mouse button to select a technique.

   You will see:

- If the modules that make up the image processing network have control widgets associated with them, then the Network Editor control panel containing these widgets appears.
- If the modules that make up the image processing network do not have any control widgets associated with them, no control panel appears. The mouse cursor provides the visual clue that everything is now ready. When you select a technique, it turns into the busy clock. When it changes back to its regular form, the network has been loaded.
- The **generate histogram** technique produces a graph viewer window. The output from these techniques will appear in these new windows.

The image processing technique is now loaded and ready to apply to an image or subimage.

At this point you can adjust the technique's control widgets, if present, to control the parameters of the technique's modules.

Note

The Image Viewer control panel window can be obscured by the technique's control widget panel. Use your window manager to move one of the control panels to another part of the screen.

Figure 99 shows the network for the **gradient shade** technique as it would appear in the Network Editor. Immediately next to the network is part of the control panel for the technique showing **gradient shade**'s dial widgets.

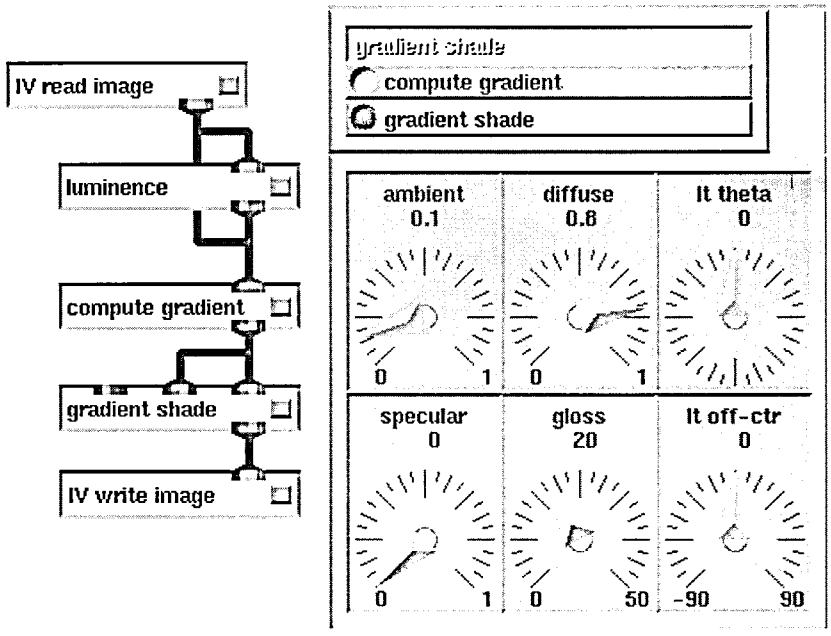


Figure 99
Network and control panel for gradient shade technique

Applying techniques on whole images

When you click on **Zoom to Image**, the network processes the entire current image. With the default **In Place** toggled, The resulting image is displayed in place of the original image in the current scene. The original image is still present and can be recalled with **Restore Current Image**, or by pointing at the image with the mouse cursor and clicking the left mouse button, just as though you were redesignating the image to be the current image.

You must click on **Zoom to Image** each time you want to apply a technique to an entire image. (Subimages do not require you to click on **Zoom to Image**.)

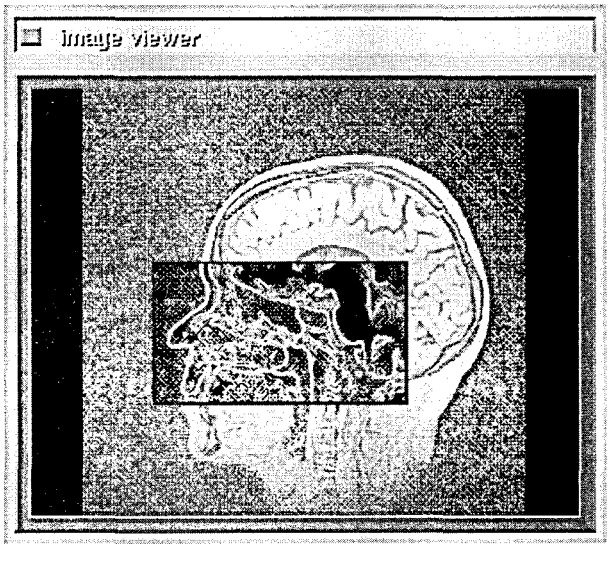
Applying techniques on subimages

To apply an image processing technique to a part of an image, you specify a subimage area.

1. Position the mouse cursor over the portion of the image you are interested in, press and hold down **SHIFT**-left mouse button.
2. With the mouse button still held down, move the mouse. A rubber-band rectangle appears on the screen in a contrasting color.
3. Move the mouse until the area of the screen you want to process is enclosed, then release the mouse button. At this point, the network executes, performing the technique on the subimage.

With the default **In Place** selected, the modified subimage appears as a section of the original current image. (See Figure 100.) The original whole image is still present and can be recalled with **Restore Current Image**, or by clicking on the image with the left mouse button.

Figure 100
Subimage area



To redefine a subimage area, press **SHIFT**-left mouse button again in a new area. The old subimage disappears as you draw the new subimage area.

Subimages are movable. Near the top of the Image Viewer control panel is the **Transform Subimage** button. When you click on this, you can move the subimage around with the right mouse button, just as though it were a whole image. The network continuously processes the pixels under the moving subimage.

If you enable **Bounding Box** below the **Transform Subimage** button, the subimage area can be moved, but the pixels are not processed until after you release the right mouse button, and the subimage is in its new location.

You can not use **SHIFT**-middle mouse button to resize subimages. Instead, just redraw the subimage area to be larger or smaller.

Image processing focus controls

Two buttons, **In Place** and **New window**, control where the processed image or subimage will be displayed.

The default is **In Place**, which causes the output image to be displayed in place of the current image in the current scene. Subimages are displayed on top of their original image. Neither is permanent unless you click on **Set Current Image**.

New Window creates a new, empty scene, and a new viewport looking onto that scene. This new window can be thought of as a scratch window.

If you process the whole image with **Zoom to Image**, the entire new image appears in the new window. The image is the same size as the original image. If you process a subimage, then the processed subimage rectangle appears in the new window, as shown in Figure 101.

Figure 101
Subimage in a new window

The new window scene can only hold one processed image. Every time you process an image or subimage, the result replaces the processed image in the new window scene.

There can be only one **New Window** scene and viewport; you can't make a new one every time you enact a technique.

Note

You cannot perform image processing techniques on the contents of the New Window scene. This kind of serial processing of an image is discussed in the next section.

Performing multiple techniques on one image

All of the image processing techniques applied to an image or to a subimage using In Place are temporary. The processed version of the image is displayed in place of the original image, but the Image Viewer keeps a copy of the original image.

Click on **Set Current Image** to make the changes to the image permanent. The Image Viewer throws away its original copy of the image, making the modified version of the image into the actual image.

Perform multiple image processing techniques on one image by using the following method.

1. Select a processing technique and experiment with it on the current image until you have achieved the desired result.
2. Make the modifications permanent with **Set Current Image**. You can also use **Duplicate Image** to save this intermediate version.
3. Select a new processing technique and continue the process on the new version of the image.

You cannot use **Set Current Image** to make the changes shown in the temporary New Window permanent on the original image.

There is no way to undo the modifications to an image once they have been made permanent with **Set Current Image**. You can, however, re-read the original image from disk, or re-execute the network that produced the original image to read a new copy into the Image Viewer.

Restore Current Image

Click on **Restore Current Image** to reset temporary changes that have been made to an image or subimage. The processed version of the original image that is being displayed in its place is discarded and the original image restored. Clicking the left mouse button anywhere in any Image Viewer viewport has the same effect.

Restore Current Image does not undo the effect of **Set Current Image**.

Window management

Many windows can be generated during the image processing session. This can cause confusion. There are usually just two control panels in active use: the Image Viewer's control panel, and the processing technique's control panel that holds the widget controls for the modules in the network. If you selected the technique that produces a graph, space for the Graph Viewer's control panel is also needed.

Note

Not all techniques produce a control panel.

When you invoke a technique with a control panel, that control panel is placed over the Image Viewer's control panel. The Image Viewer's control panel is still there (unless you explicitly remove it with **Close**).

There are two ways to deal with context switching between the technique's control panel and the Image Viewer's:

- Use your window manager to move the technique's control panel (and any other control panels) to another part of the screen, perhaps even partially off the screen. The Image Viewer control panel is revealed. Switch contexts by moving the mouse cursor back and forth between the control panels.
- Open and close the Processing Technique control panel. When it is closed, open it by clicking on **Raise Control Panel**. Close it by clicking on **Exit**.

Using Select Processing Technique

The **Select Processing Technique** button produces a browser that displays the sample image processing technique options. Table 10 summarizes what the image processing techniques actually do. The techniques provided are samples; you can design your own image processing techniques. This is described in the next section.

For more information on each technique, including what the techniques dials, slider bars, and other manipulators do:

- See the appropriate module reference page in the *ConvexAVS Module Reference*.
- Refer to the online module reference pages. These are accessible by clicking the right mouse button on the module's icon in the Network Editor. This produces a Module Editor window. Click on the **Show Module Documentation** button to see the reference page.

The technique networks are stored in the /usr/avs/networks/iview directory.

Table 10
Sample image processing techniques

Technique	Control Panel	Produces	Description
add background	yes	image	Blend image with a shaded backdrop
clamp	yes	image	Set pixel values <min = min, and >max = max
contrast stretch	yes	image	Reduce or or increase image contrast
convolve	yes	image	Apply various convolution filters to image
display histogram	yes	graph	Plot distribution of RGB pixel component values
edge detect	no	image	Find edges in an image (sobel module)
gradient shade	yes	image	Create pseudo-3D image using shading where values change rapidly
histogram stretch	yes	image	Enhance low-contrast or uneven pixel distribution images
local area ops	yes	image	Change pixels to be like neighboring pixels with various filters
luminence	no	image	Create black and white version of an image
mirror	yes	image	Reverse image about X or Y axis
pseudo color	yes	image	Repaint RGB pixel components any color with colormap editor
threshold	yes	image	Set pixel values <min and >max to 0

Special considerations

With **In Place** selected, the Image Viewer does not correctly handle modules that change the extents (size) of an image, such as **crop** and **downsize**. When you click on **Zoom to Image**, it places the new, smaller image on top of the existing image in a kind of collage, instead of replacing it. This is true even if you **Set Current Image**.

You can compensate for this effect by selecting **New Window** to receive the new, smaller image. With the new window the current window, write the cropped or downsized image to disk with **Save Image**. You can then read it back into the Image Viewer with **Read Image**.

You must be careful if you are simultaneously using networks created by the Image Viewer's Processing Technique browser with networks you yourself have created or read into the Network Editor's workspace area.

Both the Image Viewer and you are trying to share the same Network Editor workspace. The Network Editor can not differentiate. Any Network Editor function that you perform (for example, **Clear Network, Write Network**) is applied to all networks in the workspace.

The Image Viewer will not properly handle saved Image Viewer networks that also contain peer, independent networks.

The Image Viewer can get confused if a Clear Network function has deleted the **IV read image** and **IV write image** modules that it needs to import and export data to the Network Editor.

You can use multiple networks from different sources in the Network Editor workspace, but you must not perform global functions on them.

Defining image processing techniques

You can create your own directory of image processing technique networks.

When you click on the Image Processing submenu's **Select Processing Technique** button, the Image Viewer first looks in the current directory for a file called image_tech.lst. The current directory is the current directory for the terminal emulator window that invoked ConvexAVS. If this file is not present in the local directory, it uses the system default in /usr/avs/networks/iview/image_tech.lst.

To create your own image processing technique networks:

1. Create a directory to hold the technique networks and utility files. You will have to invoke ConvexAVS with this as your current directory. The Image Viewer does not automatically look for techniques in your HOME directory, nor according to any command line or .avsrc file option.
2. Copy the file /usr/avs/networks/iview/base.net into your techniques directory. Select Processing Techniques uses this file to bring up the **IV read image** and **IV write image** modules when first invoked.
3. Create a image_tech.lst file. Use the one in the system directory as a model:

   ```
   /usr/avs/networks/iview/image_tech.lst
   ```

   Here is what the first few lines of that file look like:

   ```
   "add background"      "back"
   "clamp"               "clamp"
   "contrast stretch"    "contrast"
                          .
                          .
                          .
   ```

   The left column is the alphanumeric text string that appears in the Technique Browser window.

   The right column is the name of the network file that contains the image processing network, without its .net suffix. These network files must also be in the technique directory; one cannot give system file directory specifications.

   The image_tech.lst file must also be in your technique directory.

4. Create the technique networks using the Network Editor. Use its **Write Network** option to write the networks to your technique directory.

   The **IV read image** module must be at the top of your network to retrieve an image from the Image Viewer. **IV write image** must be at the bottom of the network to return the image to the Image Viewer.

   These modules do not appear in the Network Editor's module palette. Instead, either use the Network Editor's **Read Network** option to load the file /usr/avs/networks/iview/base.net, or create an instance of the modules by invoking the Image Viewer's **Select Processing Technique** option and picking any technique. Then enter the Network Editor. (You are likely to be doing this anyway, as you construct and test a network.)

A system administrator can expand the list of image processing techniques available to everyone by creating networks for them in /usr/avs/networks/iview and editing its image_tech.lst file.

Labels submenu

The **Labels** menu selection (Figure 102) provides access to the Image Viewer's annotation text facility. You can attach one or more labels to any image. Each label consists of a single line of text. As you manipulate the image—move it, resize it, temporarily hide it, permanently delete it, etc.—the image's label(s) change accordingly.

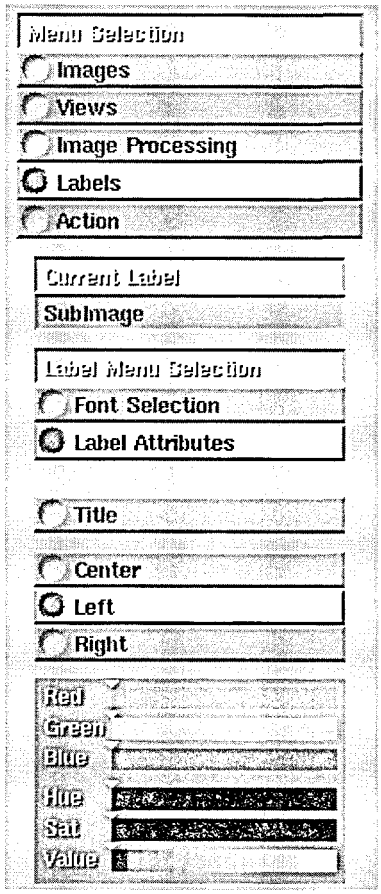


Figure 102
Labels submenu

You have considerable typographic control over these labels, 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 image; one alternative is to designate the label as a *title*. Titles always appears at the same location in the scene, no matter how the image is transformed.

Creating labels

To create a label, first make sure the image to be labeled is the current image. If necessary, click on the image with the left mouse button.

1. Click the **Labels** menu selection to bring up the Labels submenu.
2. 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.
3. When you've finished the label, press **RETURN** or move the mouse cursor out of the Current Label box.

When you do so, the label appears on the current image.

To create additional labels for the same image:

1. Select the image again by clicking on it with the left mouse button. This clears the Current Label box.
2. Check that the Image Title Bar shows the image and its name.
3. Type in a text string and click on **RETURN**.

Editing labels

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; see Figure 103. 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.

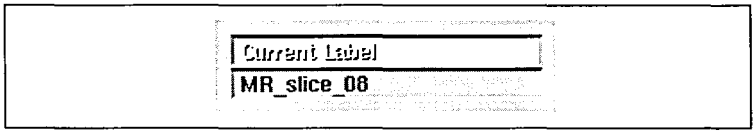


Figure 103
Current Label type-in

Selecting and moving labels

Each of an image's labels is attached to a particular point on the image. Initially, this base point is the center of the image. You can move an existing label so that its base point is at a different X-Y coordinate.

To change the location of the base point of the label, click and hold down the left mouse button on the label. Drag the cursor to any other location, then release the button.

Creating title labels

It is sometimes desirable to have one or more labels that are associated with a scene, but remain fixed on the screen as the images are transformed. Such labels are called *titles*. For instance, you might want a title string for a scene to appear in the upper left corner of the window wherever the images are displayed. You can change any regular label into a title label by clicking the **Title** selection.

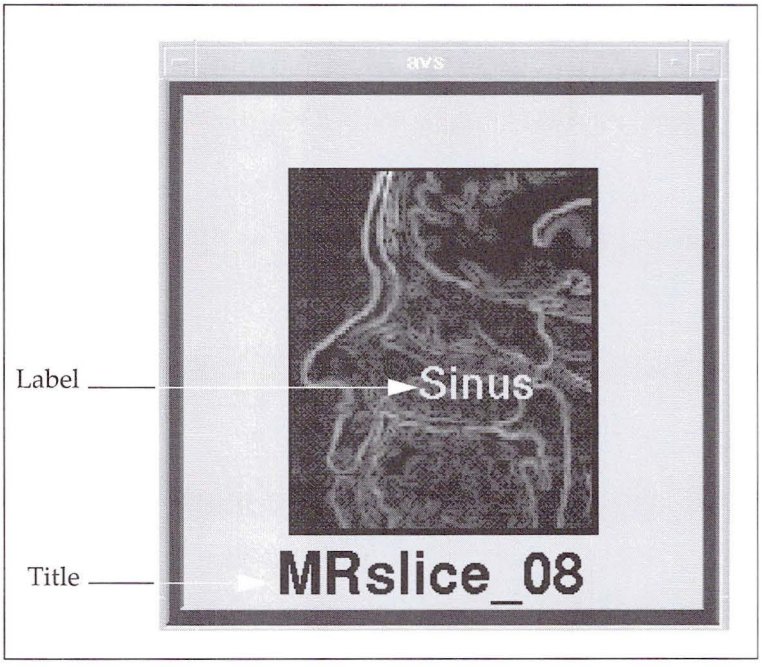


Figure 104
View with labels and titles

A title label is positioned using the scene's X-Y coordinate system. Therefore, you can change the position of a title label using the left mouse button to drag the title to any position within the view.

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.

Font selection submenu

The submenu for **Font Selection** includes a list of the available fonts, a **Bold/Italic** selection, and a **Font Height** slider bar.

- **Fonts**—The set of font radio buttons selects the X Window System font to be used for the label. The fonts that appear may vary depending on your X server.
- **Bold/Italic**—Selects the type style. You can click both of these choices to produce a bold-italic label. (Not all X servers support bold and/or italic fonts.)
- **Label Height**—Selects the 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.

Label Attributes submenu

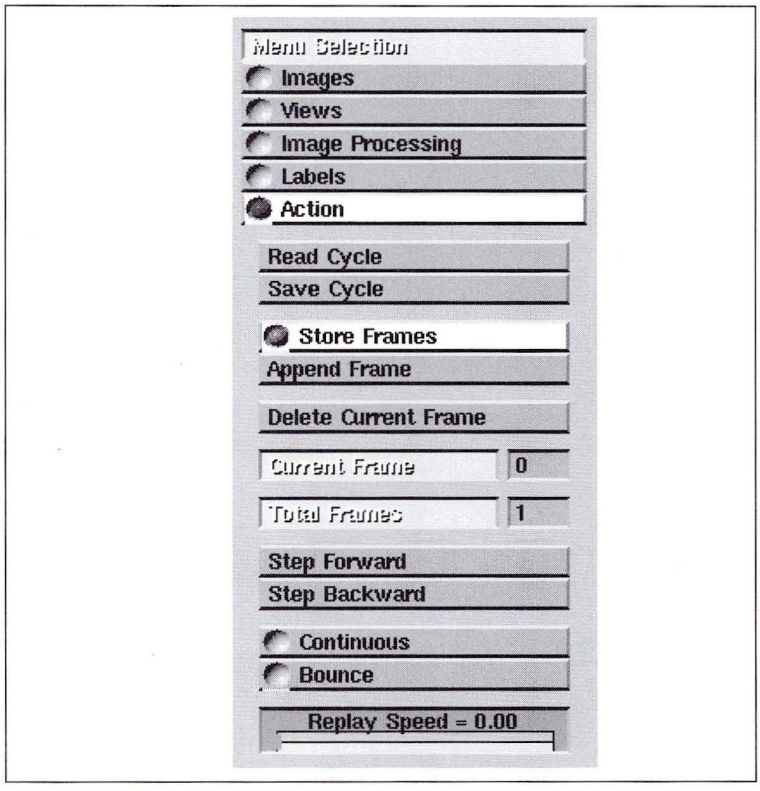
The submenu for Label Attributes includes the following choices:

- **Title**—Makes the current label into a title, whose position is fixed within the scene's coordinates. See "Creating title labels" above.
- **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**—The RGB-HSV color editor allows you to specify the color of the label.

Action submenu: Flipbook animation

The Action submenu, shown in Figure 105, is used to create and play back simple flipbook animations of images. You can save these animations to a disk file and read them in again at a later session for replay.

Figure 105
Action menu



Because any data displayed in a ConvexAVS output window is either a geometry, a pixmap, or an image; and geometries can be output as images through the **render geometry** module, you can animate and save any sequence of displays produced by ConvexAVS.

It is helpful to understand what **Action** will not do:

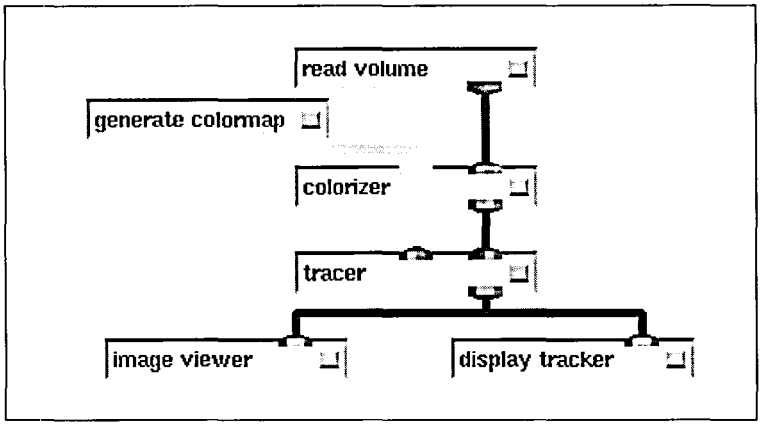
- You cannot animate scenes. For example, you cannot have several images in a scene and create an animation of them swirling about one another. (This is possible, but you must use the Image Viewer CLI interface, not **Action**.)
- You cannot create animations of changes to images that have been read into the Image Viewer with the **Read Image** button.

For example, you cannot click on **Read Image**, read in mandrill.x, then create an animation of the mandrill's image as its contrast fades, or as it zooms in to gigantic proportions. (Again, this is possible, but you have to use CLI.)

What Action can do is create an animation out of a series of images flowing into an **image viewer** module from another module through a ConvexAVS network.

Figure 106 shows one possible network:

Figure 106
Action network



The person using this network reads in the file /usr/avs/data/volume/hydrogen.dat. This is a 3D uniform volume data file that shows the probabilities of an electron occurring around a hydrogen nucleus. The **generate colormap** and **colorizer** modules together paint the probabilities as colors—low probabilities are blue, high probabilities are red. the **tracer** module draws this volume of painted probabilities as though it were a slightly hazy, semi-transparent, but solid 3D object existing in space. The **tracer** module outputs an image. One copy of this image goes to the **display tracker** module. The **display tracker** module lets you move the rendered atom as though it really were a 3D solid, not a 2D image. The second copy of the output image from the **tracer** module will go to the Image Viewer for display and animation.

By adjusting the controls on **display tracker**, you could animate the volume rotating in space.

Adjusting the Alpha (opacity) controls on the **tracer** module would animate the hydrogen atom going from mostly opaque to nearly transparent.

Adjusting the **generate colormap** module's colormap editor would animate the volume data changing color and/or opacity.

To begin an animation, turn on the **Store Frames** button from the Action submenu in the Image Viewer control panel. This is like pressing a record button.

Each time the **tracer** module outputs a new image of the hydrogen atom, this image flows into the **image viewer** module and is added as a new page in the flipbook.

A network that captures geometry animations is shown at the end of this section.

Note

Images get large quickly. Each pixel is one 32-bit word. A 512x512 image (roughly 6x6 on the screen) takes up about 1 megabyte of memory. A flipbook animation consisting of 20 images thus takes up 20 megabytes of memory. Be aware of the memory size and swapspace limitations of your machine.

Saving this 20-frame animation as a cycle for future replay will also require 20 megabytes of memory. When flipbook images are stored, they are stored in their original size. For example, if a 256x256 pixel image enters the image viewer, and it is then rescaled to 512x512 pixels, it is stored in its original 256x256 pixel size.

Store Frames

Store Frames is a toggle button that enables and disables image recording. It also causes the remainder of the Action control widgets, normally invisible, to appear. It is off by default. With **Store Frames** on, each image flowing into the **image viewer** module is added sequentially to the flipbook. The count shown under Total Frames will increment. When you have finished capturing frames, disable **Store Frames**.

Append Frame

Append Frame adds single images, already in the Image Viewer window, to the end of the image cycle. It adds one frame at a time to the sequence. Frames are always added at the end of the cycle. (To change the order of frames, you can edit the ASCII cycle file created by the **Read Cycle** button described below.)

Total Frames

The Total Frames counter counts the number of frames that are stored in the cycle. It increments as images are added, either through **Store Frames** or **Append Frames**.

Current Frame

Current Frame shows which of the numbered frames is presently being shown when you play cycles of images back. This counter starts at zero, where Total Frames starts at 1.

Step Forward/Step Backward

These two controls play back cycles of images one frame at a time. As you move forward or back, the Current Frame indicator updates the number of the frame being shown.

Continuous

Continuous is an toggle switch that plays the cycle of images continuously, instead of stepping through them one-by-one manually. It plays them in a continuous cycle, for example, 9 10 11 12 0 1 2. To turn replay off, click on **Continuous** again.

Bounce

Bounce also plays the cycle of images continuously, but in a different order than **Continuous**. With **Bounce** toggled, frames count up then count down, for example, 9 10 11 12 11 10 9. To turn **Bounce** replay off, click on it again.

Replay Speed

This slider bar widget speeds up or slows down the rate at which images are replayed. (This is also affected by the power and load of your machine.) Moving the slider to the right slows replay down.

Delete Current Frame

This removes the current frame from the cycle. The total number of frames, and the sequence shown in Current Frame are immediately renumbered. To delete an entire cycle, just click on **Delete Current Frame** repeatedly.

Save Cycle

Save Cycle saves Image Viewer flipbook animation cycles to disk where they can be retrieved at a later date. **Save Cycle** raises a type-in window that prompts for a file name in the defined DataDirectory. To save the cycle elsewhere, type in a full directory and file name specification. Use **CTRL-U** to delete the entire type-in line, **BACKSPACE** deletes the previous character, and **ENTER** or **OK** completes the entry. ConvexAVS automatically appends the .cyc file suffix that denotes a cycle.

The cycle of images comprises a series of files, one for each frame in the series. The file *name*.cyc defines the cycle. It is an ASCII file written in Image Viewer CLI commands. The images that make up the cycle are automatically stored in separate files in the same directory, one file for each frame. The file *name*.cyc.x is the original image. The rest of the image files in the series have the same name as the primary cycle file, a *midfix* (for example, .cyc0, .cyc1, .cyc2, etc.) that denotes the image's sequence in the cycle, and the standard image file suffix, .x.

Note

Be aware of the size of the images in the cycle and the number of images and be careful that there is enough available disk space to hold them.

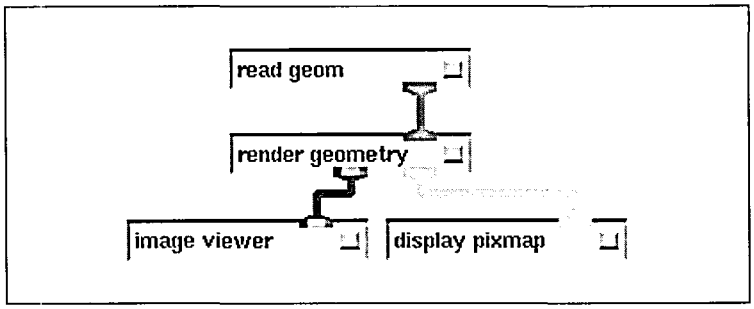
Read Cycle

Click on the **Read Cycle** button to raise a file browser displaying the defined DataDirectory. Only directory files and files with the .cyc suffix are displayed. (.x, and .ims files, if present, are also displayed.) By clicking on the primary cycle file, the entire cycle is read into the Image Viewer. Because this could consist of many large files, it may take some time. Read Cycle also puts up the Action animation controls. You can now replay and edit the cycle.

A network for geometries

The network in Figure 107 uses the **render geometry** module to output an image that can be animated with the Image Viewer.

Figure 107
Example network



At the top of this network is the **read geom** module. However, this could be replaced with any set of modules (for example, , **read field, isosurface**) that eventually produces a geometry.

Image Viewer and the CLI

It is possible to drive the Image Viewer with commands rather than the X Window System display interface. The commands can be either typed in interactively from a terminal emulator window while ConvexAVS is running, or they can be read from a script file.

With this method, you can:

- Create scripts that animate the Image Viewer itself, not just the network-produced images within it
- Create demonstration, illustration, and test scripts
- Create scripts that batch-process images

Use the `-cli` option on the command line to run ConvexAVS with the Command Language Interpreter (CLI) active:

```
avs -cli other-options
```

This starts ConvexAVS as usual, but also starts the CLI command line interpreter in the invoking window. (You might have to press **ENTER** to get the `avs>` prompt.)

To get a list of the Image Viewer CLI commands, type the following:

```
avs> help Image
```

This produces a list of the many Image Viewer CLI commands. To get help on an individual commands, type `help` plus the command name:

```
avs> help image_create_scene

image_create_scene Create a new scene with scene
location and size

Usage: image_create_scene <xlocation ylocation
width height>

avs>
```

Sample versions of Image Viewer CLI files can be found using the following methods:

- Create a fairly complex scene with multiple viewports, images, transformations, etc., then save it out with **Save Scene**. This .ims scene file is written in Image Viewer CLI.
- Look in the directory /usr/avs/demo/image_viewer.

  Script files have a .scr suffix.

  Scripts can be run interactively through the Help Browser. Click on **Help**, then click **Help Demos**. This puts up a demo browser. You can also run scripts interactively through the CLI interpreter as follows:

  ```
  avs> script -play filename
  ```

The Command Language Interpreter and the Image Viewer set of CLI commands are documented in detail in Chapter 17, "Command Language Interpreter."

Graph Viewer subsystem 6

The ConvexAVS Graph Viewer subsystem is an interactive tool for creating XY linear or contour plots of data.

XY linear plots can be displayed as a line plot, area plot, scatter plot, or bar plot. Contour plots are displayed as 2-D graphs of lines connecting data points with the same level (numeric) value. The graphs can be annotated with a title and axes labels in various font styles and colors. You also have control over the placement and style of axis tic marks.

The Graph Viewer exists in two forms:

- Graph Viewer subsystem, accessible from the main menu
- The graph viewer module in the data input section of the Network Editor's module palette.

The Graph Viewer supports the following views:

- Multiple data sets within the same window
- The same data set in different windows, such as a line plot showing trends and a scatter plot showing groupings
- Different data sets in separate windows

The Graph Viewer accepts one or two-dimensional data as input. The input data can be either an ASCII file in a particular format, or a field. In addition, the Graph Viewer will read in a ConvexAVS X image file and use it as a backdrop for its plots; and it can re-generate plots saved from an earlier Graph Viewer session in a ConvexAVS Plot file.

Data can enter the Graph Viewer in two ways:

- **Read Data function**—Read in ASCII data files, field files, Plot files, and X image files.
- **Graph viewer module**—Fields and image files can flow into the module from a network.

The Graph Viewer interprets the input data it receives in he following ways:

- As a series of Y values to be plotted against a set of constantly-spaced X axis intervals that the Graph Viewer generates automatically (**Plot as Y Data**)
- As a series of XY values to be plotted against Y and X axes that are automatically generated to match the range of the input data (**Plot as XY Data**)
- As a 2-D array through which it draws contour lines connecting the same level values (**Plot as Contour Data**).

You can save plots in the following formats:

- ASCII data
- AVS Plot-format
- AVS Geometry
- PostScript files

The **graph viewer** module outputs a ConvexAVS geometry.

The Graph Viewer is used for viewing data; it cannot be used to alter data.

There are three sample ASCII data files in /usr/avs/data/graph.

Entering and leaving the Graph Viewer

You can enter the Graph Viewer subsystem by using one of the following methods:

From the shell

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

```
avs -graph
```

From the main menu

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

Data viewers

At the top of each subsystem control panel is a **Data Viewers** button. Clicking on this button creates a pop-up menu listing the subsystems. While still holding the mouse button down, roll the mouse cursor down to Graph Viewer and release. This raises the Graph Viewer control panel. You may wish to move the control panel to another part of the screen so that other subsystem control panels are visible.

In a network

A graph viewer module is available for inclusion in a network.

The Graph Viewer starts by displaying its control panel along the left edge of the screen, as shown in Figure 108.

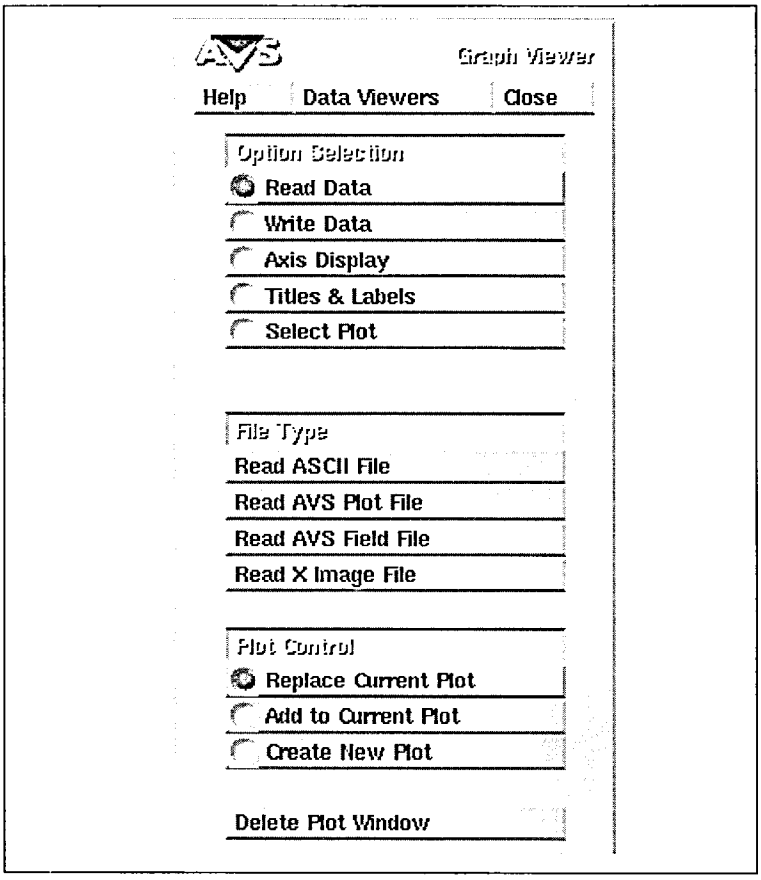


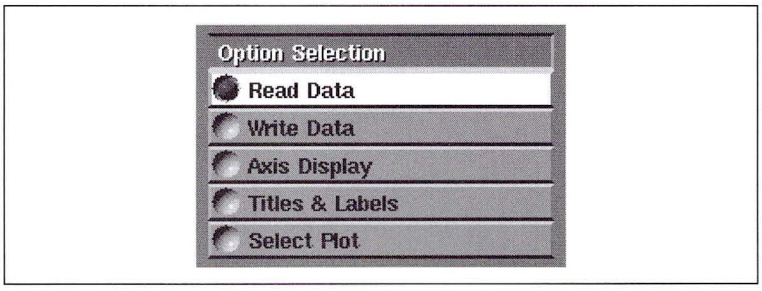
Figure 108
Graph Viewer control panel

To leave the Graph Viewer, click the **Close** 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.

Basic Interface

The Option Selection menu shown in Figure 109 provides access to the basic operations for creating a plot.

Figure 109
Option Selection menu



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

- **Read Data**—Select the input file type (File Type menu); select how to interpret the input data (Data Format menu); select whether the new plot will replace the existing plot, be added to the plot, or be plotted in an entirely new window (Plot Control menu); select the type of linear plot to make, or the number and level of contours; and Delete Plot Windows.
- **Write Data**—Write the current plot window as one of the four output file types.
- **Axis Display**—Control the border accents; the number, type, and placement of axis tick marks; the axis range, and the axis type (for example, logarithmic or linear).
- **Titles & Labels**—Create titles and labels for each plot window, and the individual X and Y axes.
- **Select Plot**—Control the type of plot (line, scatter, etc.), as well as the appearance of individual lines within the plot (dotted, dashed, thickness, color).

One of these choices is always selected. The area below the Option Selection menu changes depending on which choice is currently selected.

Using the Graph Viewer

Using the Graph Viewer consists of four steps:

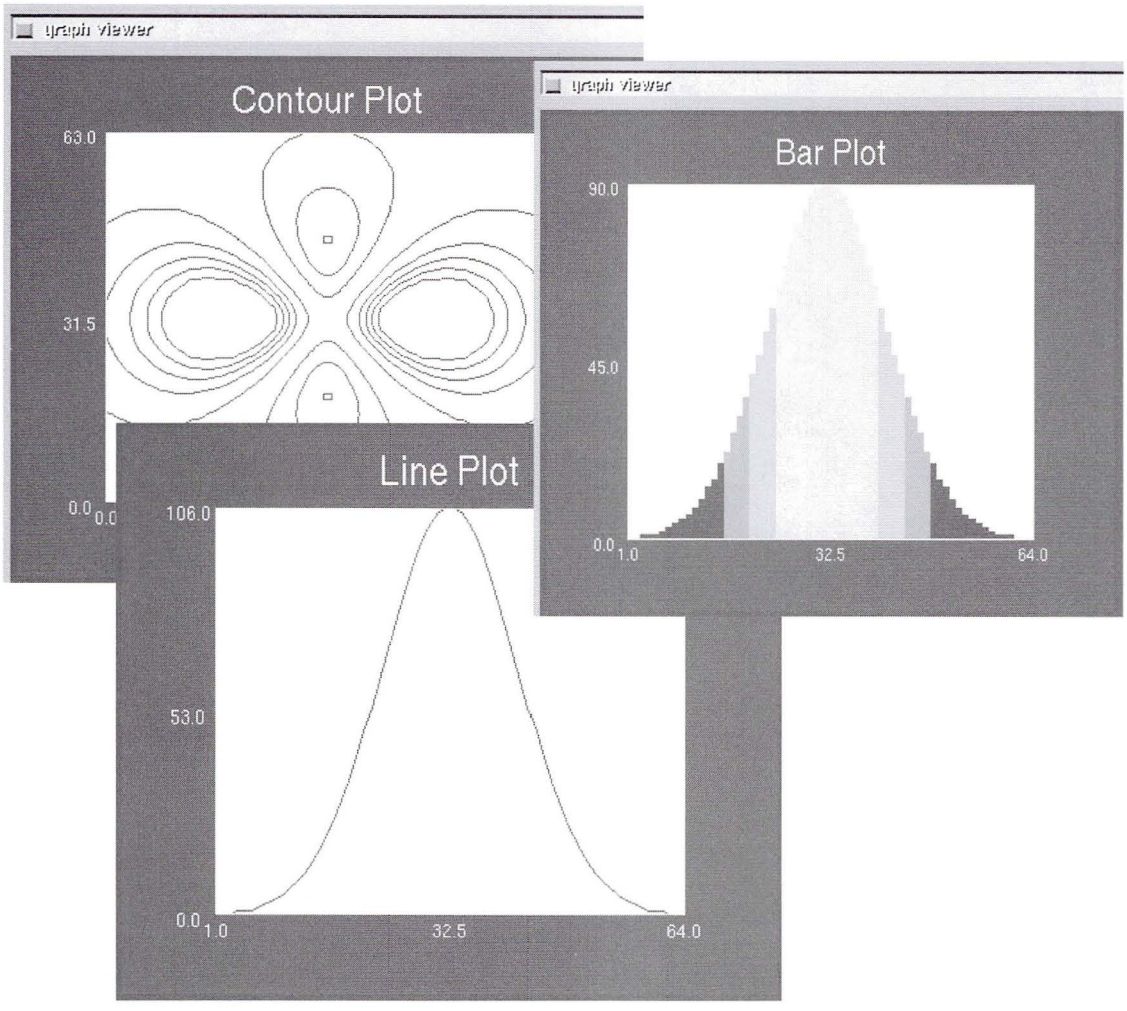
1. Input the data you want to view. You need to select the input file type (ASCII, field, etc.) and the data format (Plot as Y Data, Plot as Contour Data, etc.) you want to use.
2. If you are creating a linear plot, choose the type of plot (line, area, scatter, bar). If you are creating a contour plot, set the number, value, and range of the contour levels.
3. If you wish, edit the plot axes, titles, labels, and style of the plot lines.
4. If you want to save the output, you then select the file type you want to save or print.

Multiple plot windows—the current window

You can create and delete plot windows at any time. This provides the ability to display the same data set in different windows and work with each one independently, or to display completely separate data sets in different windows for comparison.

For example, Figure 110 shows three plot windows. The first window is a contour plot of one 2-D slice through the center of a 3-D field data set. The second window is a bar plot of a 1-D section through the middle of the same 2-D slice. The third is a line plot of another 1-D section through a different part of the same 2-D slice.

Figure 110
Multiple plot windows



Each window contains one or more plots and can be labeled with its own title, X and Y axis labels, and tick marks. Their identifiers can be altered in size, style, color, and position at any time.

One of these plot windows is the current plot window that will be affected by all menu operations.

The current plot window is indicated by a red border. You can change the current plot window by moving the mouse cursor into any plot window, then pressing any mouse button.

Read Data

The Graph Viewer accepts an ASCII file or a ConvexAVS field file as input and uses numerical data to generate a plot. It also accepts two other input file types that are not plotted as data: Plot files and image files. The menu of choices appears under the **Read Data** main menu button as shown in Figure 111.

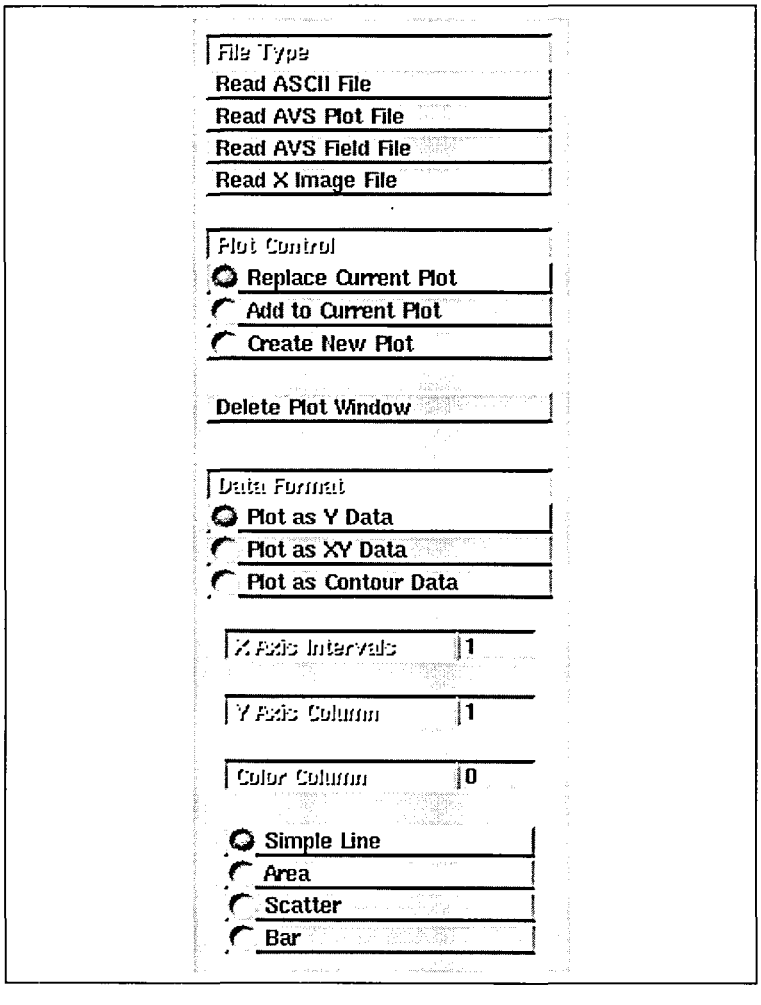


Figure 111
Read Data submenu

File type submenu

The Graph Viewer accepts the following data file types as input:

Read ASCII file

These are ASCII files with numeric data arranged in one or more columns. The data can be specified as integers, real numbers, or in floating point scientific notation style. You can read in any file by typing its name into the **Read Data New File** type-in panel. However, for a file to show up in the **Read Data** file browser window, it must have .dat as the file suffix.

The file can contain only numerical data as input. Non-numerical data in the file must be preceded by the pound sign (#) character in the first column.

ASCII files can only be read into the Graph Viewer through the Read Data function; they cannot enter the **graph viewer** module through an input port.

The Graph Viewer assumes that ASCII data files are tabular, that is, tables of columns of numbers. One column might be pressure, or density, or temperature; another column might be time, a Richter scale value, or daily rainfall. You can select any one column to plot (such as pressure), or you can select any column and plot it against any other (pressure against time). For example, here is the sample ASCII data file /usr/avs/data/graph/growth.dat:

```
0 0 1
10 20 1
30 30 2
80 50 3
90 20 4
95 80 5
```

Choosing **Read ASCII File** causes the Data Format menu to appear.

Read AVS field file

The Graph Viewer accepts fields of any dimension, of any vector length, and of any coordinate type (uniform, rectilinear, irregular).

However:

- If the field is 3-D or greater, the Graph Viewer will, by default, use just the first XY 2-D "slice" of the field. (The Graph Viewer does not create three-dimensional graphs.)

- If the field has a vector of data at each point, the Graph Viewer will graph just the first element of the vector.
- Though the field may have a rectilinear or irregular (curvilinear) coordinate system, this release of the Graph Viewer will plot all data on a uniform grid. The original spatial relation of the data points is not preserved.
- For a field file to show in the **Read Data** file browser, it must have .fld as the file suffix. You can read in any field file without this suffix using the file browser's **New File** type-in panel.

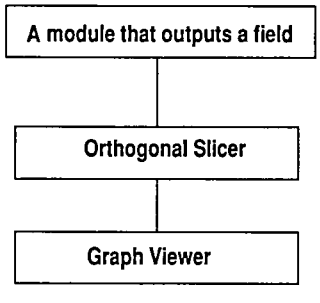
Choosing **Read AVS Field File** causes the Data Format menu to appear.

Graphing fields

It is more typical to graph field data entering the Graph Viewer through a network rather than reading the field data in through the **Read AVS Field file** function.

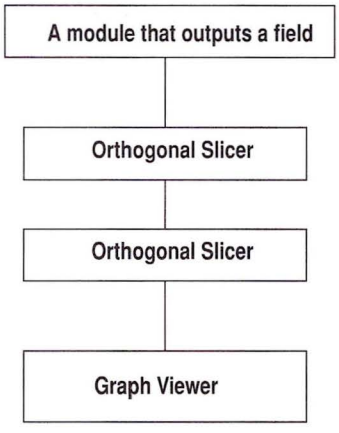
AVS fields enter the graph viewer module through either its right or center input ports. By default the rightmost port assumes you wish to make a linear plot, and the center port assumes you want to make a contour plot; but either can be changed from the Graph Viewer interface.

The Graph Viewer, by default, plots just the first XY plane of a 3-D field. Therefore, you might use one or more **orthogonal slicer** modules in the network to pare down the field to a dimensionality that is meaningful to the Graph Viewer, and that contains just the data you want to graph. (The **crop** module can also be used.) To turn a 2-D field into a 1-D field or to turn a 3-D field into a 2-D field use a network like the following:



Use the orthogonal slicer's controls to pick the column or plane of the field you want to graph. This is how you would create a 2-D slice of a 3-D scatter data field that the Graph Viewer could then use to make a contour plot.

If you have a 3-D field that you want to plot as a 1-D **Plot as Y Data** operation, use two **orthogonal slicer** modules in succession:



If the field contains vector data, you should use the **extract scalar** module to isolate the vector element that you want to graph. The Graph Viewer graphs the first vector element in a field. The **extract scalar** module can be placed either before or after any **orthogonal slicer** modules.

Read AVS plot file

AVS Plot files are files that you have already created with the Graph Viewer, and saved with the **Write Data** and **AVS Plot Data** buttons. The plot file does not contain numeric data. Rather, it contains the instructions for redrawing a Graph Viewer window, with plot lines, axes, titles and axis labels exactly as they were originally saved.

For a plot file to show in the **Read Data** file browser, it must have .plt as the file suffix. You can read in any plot file without this suffix using the file browser's **New File** type-in panel.

Note

These files create a new window when selected—they cannot be read into an existing window.

Read X image file

The Graph Viewer uses the term X Image File to mean an image data type file with the .x suffix. The Graph Viewer uses the image file as a backdrop to its plotting field. Image files can be read in with **Read Data**. They can also enter the graph viewer module through its leftmost input port.

When an image enters the Graph Viewer, the plot window is resized to fit the image.

For an image file to show in **Read Data**'s file browser, it must have .x as the file suffix. You can read in any image file without this suffix using the file browser's **New File** type-in panel.

Plot control submenu

The Plot Control submenu selects among three methods for reading plot data into the plot window. You should establish your choice before you read in the data file through the file browser, and before the upstream module sends its field or image to the **graph viewer** module.

Replace Current Plot

This selection replaces all of the plots in the current plot window with a new plot data set.

Add to Current Plot

Use this selection to read in a new plot data set (ASCII or field) and add it to the current plot window. This allows you to display multiple plots in the same plot window. You can quickly compare several data sets within a single window using the same axis and tick values.

You can also mix plot styles within the same window. For example, data set A as an area plot and data set B as a scatter plot. Because a data set plot style can be changed at any time, you can start with the same plot styles and change one window plot, or start with dissimilar plot styles.

Each data set read into a window as input is automatically assigned a different color. You can change its plot characteristics (line type, scatter character, color) at any time. The axis and tick values of a window plot are always updated to reflect the largest data set read into the window.

Note

If a previous data set has ranges greater than one, a new data set with ranges less than one is mapped as between zero and one without decimal value tick marks. Therefore, to get correct axis tick marks you need to create a new window and read the new data set into it.

Create New Plot

This selection creates a new plot window.

Delete Plot Window

Deletes the active display window. There is no request for confirmation-once a window is deleted, its display is gone.

Data formats submenu

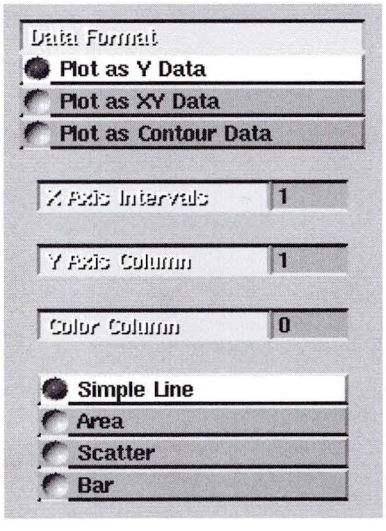
Once you have selected an input file type, the Data Format submenu and the file browser widget appear.

There are three different ways for the Graph Viewer to map its input data into a plot.

Plotting as Y Data

The **Plot as Y Data** button, sown in Figure 112, directs the Graph Viewer to use one column of the input data as a series of Y values. An example plot is shown in Figure 113. If the input data is a 2-D field, it is viewed as a two-dimensional array, where each X value (X=1, X=2, and so on.) is a column of Y values. By default, column one of either an ASCII file or field is used. The X Axis for the graph is generated automatically. The Graph Viewer counts the number of data items in the Y Axis column and places the same number of items on the X Axis in intervals of 10. You can change which column of the input data is used.

Figure 112
Plot as Y Data menu



X Axis Intervals

Use this type-in to set the interval range between item counts on the X Axis. You can change the interval range at any time.

Y Axis Column

Use this type-in to select which column of the input data to use as Y values. You must select the data column before inputting the file.

To change the input column after reading in the file, enter a number into the Y column type-in.

Note

If you type in an invalid column number, the Graph Viewer displays a message showing how many columns are available.

Color Column

You can also use column data to control line segment color (see "Using column data for color control," on page 254).

Plot Styles

You can select Simple Line, Area, Scatter and Bar plot styles for displaying the data. The default setting is Simple Line. You must select the plot style prior to reading the file. To change the plot style after you have read the file, see "Select plot submenu," on page 262.

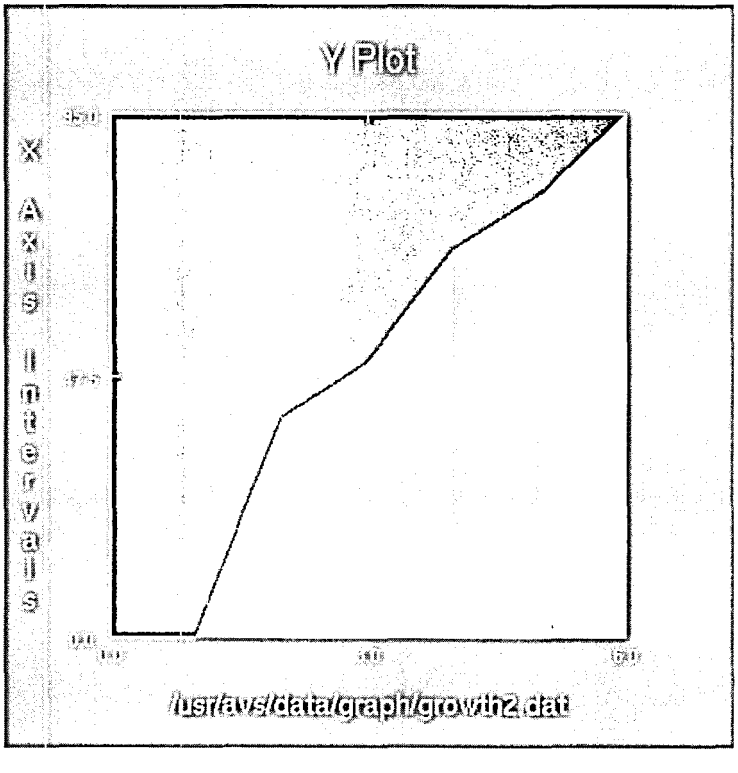


Figure 113
Example plot as Y data

Plotting as XY data

The **Plot as XY Data** option, shown in Figure 114, directs the Graph Viewer to use two columns of the input as a pair of X,Y data values to be plotted against correspondingly scaled X and Y axes. An example plot is shown in Figure 115. By default, the Graph Viewer uses column one as the X Axis values and column two as the Y Axis values. You can change the input columns as follows:

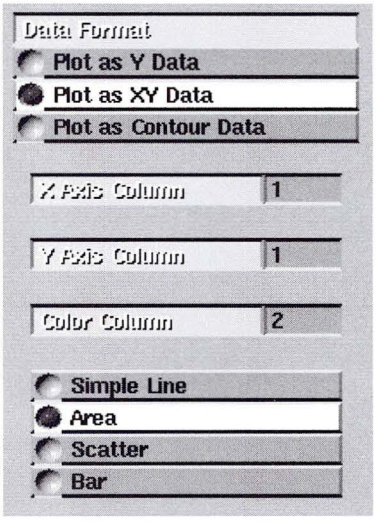


Figure 114
Plot as XY Data menu

X Axis Column

Selects which column of the input file or field to use as the X values. You must select the data column before inputting the file.

Y Axis Column

Selects which column of the input file or field to use as the Y values. You must select the data column before inputting the file. If you need to change the X Axis or Y Axis input column after reading in the file you must delete the data set (see "Delete Plot data set," on page 265) or delete the window (see "Delete Plot Window," on page 248), then reselect the file for input with the input columns you want. If you type in an incorrect column number the Graph Viewer displays a message showing how many columns are available. Repeat the steps to select a valid column number for input.

Color Column

You can also use column data to control line segment color (see "Using column data for color control," on page 254).

Plot Styles

You can select **Simple Line, Area, Scatter** and **Bar** plot styles for displaying the selected data. The default setting is **Simple Line**. You must select the plot style prior to reading the file. To change the plot style after you have read the file, see "Select plot submenu," on page 262.

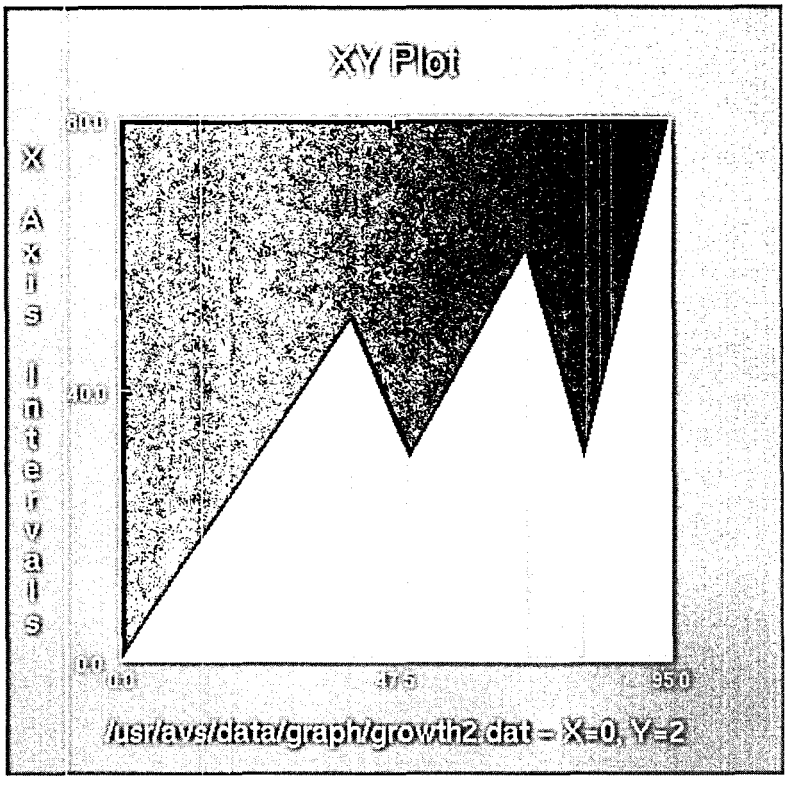


Figure 115
Example plot of XY data

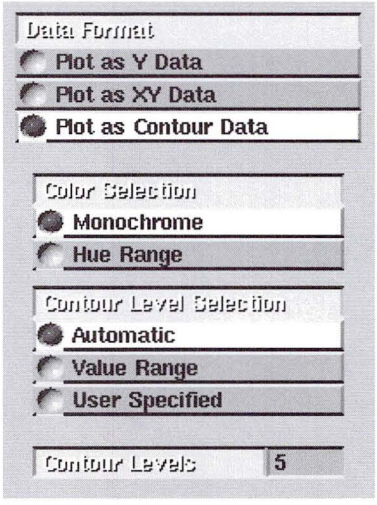
Note

It does not usually make sense to plot typical field data as XY data. You would essentially be plotting one column of X, Y, or Z values against another column of the same data—X1Y1 against X2Y1, X1Y2 against X2Y2, and so on. You could write a module that constructs an output field where this operation was meaningful, but no supplied modules do this.

Plotting as contour data

The **Plot as Contour Data** option directs the Graph Viewer to use all columns of the input data to construct a 2-D contour plot. By default, each contour level is displayed as a different color. You can also select monochrome contour mapping to assign all levels the same color. The number of contour levels and coloring can be changed at any time.

Figure 116
Plot as contour data menu



If you want to make a contour plot of a 3-D field, remember that the Graph Viewer plots only the first XY plane of field. Use the **orthogonal slicer** module in a network to select the 2-D planar slice of the data from which to create the contour plot.

Note

Because of the complexity of contour plot data, the computing and display time can be significant. Be aware of the size of your input data, and the number of levels you are trying to plot. If necessary, you can use the downsize **module to thin out field data before it enters the graph viewer module, and the Value Range or User Specified controls listed below to keep the number of levels reasonable.**

Color Selection

There are two options for color selection of contour plots:

Monochrome
Displays all contour levels in the same color (red).

Hue Range
Displays each contour level of the plot in a different color. This is the default. The range is not user-selectable.

Contour level selection

You specify the number of contour levels. The range of data (from low to high) is divided into evenly spaced contour levels. Data occurring on each contour level is assigned a color and plotted. Figure 117 shows a plot with 20 levels.

For example, data containing values from 0 to 90 can be assigned 10 contour levels. The contour levels are evenly divided between the low value and the high value (inclusive with the high value). Therefore, the contour levels are placed at the following values:

```
0, 10, 20, 30, 40, 50, 60, 70, 80, 90
```

Contour lines will not be generated at a data set's minimum (0) and maximum values (90).

If you are not sure what the range of your field data actually is, you can find out with modules such as **statistics** and **generate histogram**. The **statistics** module produces an ASCII output widget showing the data minimum and maximum values; and **generate histogram** produces a 1-D field profiling the distribution of data values in a field that is meant to be displayed with the **graph viewer** module.

There are three Contour Level Selection options:

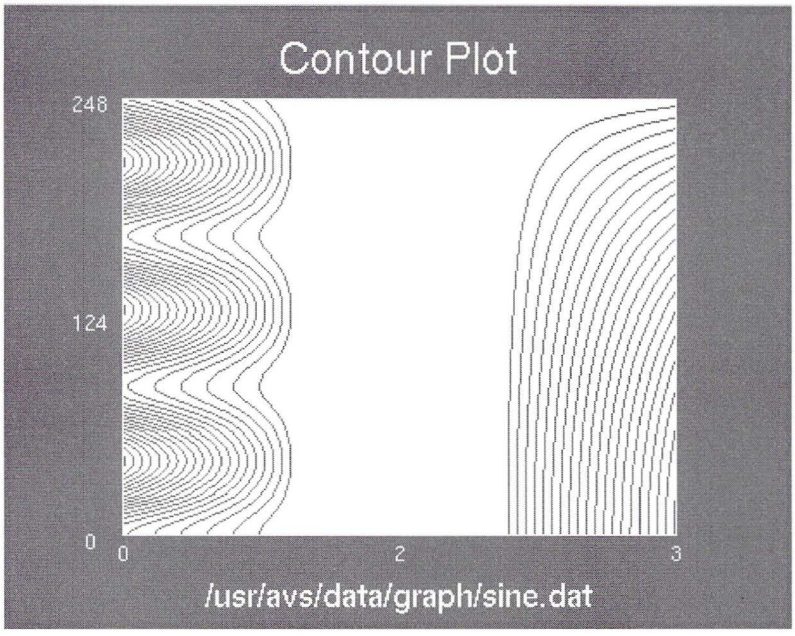
Automatic Divides the entire range of values into evenly spaced contour levels. In this case, you only specify the number of contour levels you want. It should be noted that the minimum contour value will not be the minimum value of the data set. It will be a value greater than the minimum value. This ensures that a contour line is drawn. Also, the maximum value will not be the maximum value of the data set. It will be a value less than the maximum value. The default number of contours is 5.

Value Range Selects a range between a low number and a high number and divide the range into contour levels. For example, you can read a file ranging from 1 to 50 but specify a starting value of 10 and ending value of 40. Only values occurring between these points (inclusive) are permitted. Then this range is evenly divided into the number of contour levels you specify. Only data points occurring on the contour levels are plotted.

User Specified

Explicitly states the contour values to be plotted. See the figure below. For example, you can read a file ranging from 0 to 500, then specify the contour levels of 26, 48, 89, 150, 223, 415, 487 and 499. Only data points occurring at these levels are plotted.

Figure 117
Example of plotting contour data



Using column data for color control

Color control enables you to select the colors used for displaying line segments. You need to choose an input column of data values to control the line segment colors.

A data value of one tells the Graph Viewer to use color one, a data value of two means use color two, and so on.

A value of 0 means no color is displayed.

For example, if Column Y has three data items (Y1, Y2, Y3) and Column X has three items (X1, X2,X3), then Column C items (C1, C2, C3) can control the colors used to connect line segments Y1-Y2 and Y2-Y3.

There are three data values in Column C, but only two (C2 and C3) are actually used for line colors. The first data value of the color control column is always ignored and reset to zero.

Color control values range from 0 to 499 to control Hue color. If your color control data value is greater than the number of available colors, the data value is wrapped around to the beginning color.

For example, a value of 500 wraps around to 0, 501 wraps to 1, 502 wraps to 2, and so on.

Non-integer values are rounded to the nearest integer. Values between one and zero are rounded to either one (color one) or zero (no color).

Plot styles submenu

The Graph Viewer provides four styles of plots for viewing linear data. You can change from one style of plot to another at any time. You can display different plot styles within the same window. The plot styles are:

Line Plots

Line plots connect data points with lines so that you can easily see changes or trends within your data. You can change the type of line (for example, solid, dashes), its thickness and color at any time.

Area Plots

Area plots fill in the data points with solid color so that similar and dissimilar data points are easily viewed. You can change the color of the filled area at any time.

Scatter Plots

Scatter plots show the data points as a character (for example, an asterisk) so that groupings of data can be easily seen. You can change the character type, style, size and color at any time.

Bar Plots

Bar plots display data in vertical bars so that you can easily compare data values. You can change the color of the bars at any time.

Write Data

Use the Write Data option to save the selected plot data for later retrieval or printing. Selecting this menu item causes the File Type menu to appear as shown in Figure 118.

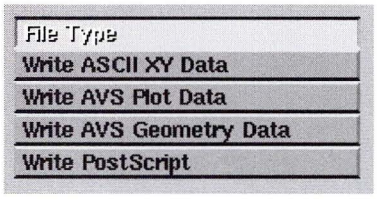


Figure 118
Write Data File Type menu

File Type submenu

You can select the following file types for output:

Write ASCII XY Data
Saves the selected plot data in a two column ASCII file. Title, axis and tick mark information (if any) is not saved. This format is typically used to save data without additional plot parameters in the file, possibly for use by another program. Note that a one column input file is saved with the generated X Axis as a two column output file. ConvexAVS automatically appends the .dat suffix to the output file name.

Write AVS Plot Data
Saves the selected plot in Plot format. Title, axis, tick mark and plot size information are saved. This format is typically used to save a completed plot for retrieval and display. ConvexAVS automatically appends the .plt suffix to the output file name.

Write AVS Geometry Data
Saves the selected plot in Geometry format. Title, axis and tick mark information is included. This format is typically used to save a completed plot for further use in the Geometry Viewer subsystem. ConvexAVS automatically appends the .geom suffix to the output file name.

Write PostScript File

Saves the selected plot in PostScript format. Title, axis and tick mark information is included. This format is typically used to save a completed plot for printing on a PostScript printer. Color information and background images are not saved in the PostScript file, nor sent to the printer. ConvexAVS automatically appends the .ps file suffix to the file name.

In order to actually write the file, you must click on the file browser's New File button, then enter the file name in the type-in panel that appears.

Axis Display

This menu option controls axis display information for the current plot window. You can change these settings at any time. See Figure 119.

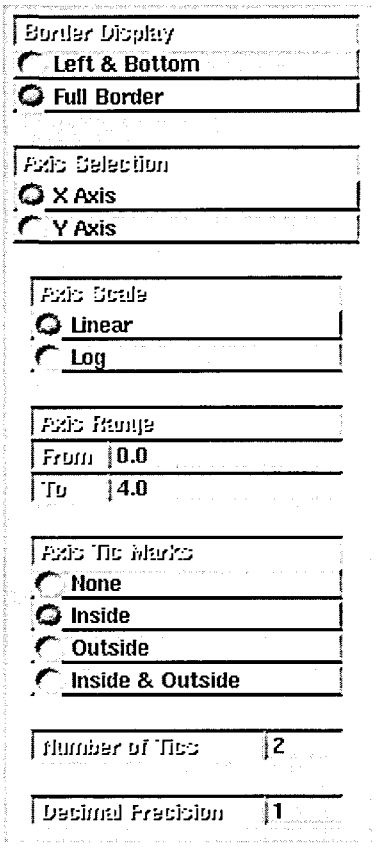


Figure 119
Axis Display menu

Border Display

Sets the border partially or completely around the selected plot.

Left & Bottom
Displays the border on the left side and bottom of the plot.

Full Border
Displays the border around all four sides of the plot.

Axis Selection

Selects the X or Y axis of the plot for control of all of the following menu items. Only one axis can be selected at a time.

X Axis Selects the X Axis of the plot.

Y Axis Selects the Y Axis of the plot.

Axis Scale

Sets the selected axis of the plot as linear or log based. Each axis can be only linear or log based.

Linear Sets the selected axis as a linear base display.

Log Sets the selected axis as a log base display.

Axis Range

Sets the low and high data input thresholds of the selected axis. The defaults are the lowest and highest data values of the axis data set.

This enables you to select a superset of the data as input for the selected axis. For example, you can narrow the range of 0-100 to 25-75 for a better examination of data between these points.

From Sets the lowest value of the data set allowed on the selected axis. The default is the lowest data set value.

To Sets the highest value of the data set allowed on the selected axis. The default is the highest data set value.

Note

Axis range is reset to 5.0 for each new plot.

Axis Tic Marks

Selects the type of tick marks (if any) that are displayed for the selected axis. The default setting is for tick marks to be displayed inside the plot axis. Only one of the following can be active at a time:

None Sets the tick marks to none for the selected axis-meaning no tick marks are shown.

Inside Sets the tick marks inside the selected axis.

Outside Sets the tick marks outside the selected axis.

Inside & Outside
Sets the tick marks both inside and outside the selected axis.

Number of Tics

Selects the number of tick marks shown on the selected axis. The default value is 2 tick marks. The axis is re-displayed with the new value of tick marks.

Decimal Precision

Sets the decimal precision values for the selected axis. The default value is one position right of the decimal. The affected menu item is **Axis Range** (From and To values).

The affected menu items are not shown with the new precision until the Axis Display menu is re-displayed.

If you need to see the new precision after changing the value, select another menu item (for example, **Read Data**) and then reselect **Axis Display**.

Titles and Labels

This menu option controls the titles and axis labels for the current plot window. You can change these settings at any time. Selects between the title, axis labels and tick labels for changing text and attributes.

Plot Title Selects the current plot window's title text and attributes for addition or change. If there is no current title the Current Label data area remains blank. If a title exists it is shown in the Current Label data area. Attributes (size, style, location, color) can be changed at any time (see the next section, "Label menu selection").

X Axis Label You can select the X Axis Label text and attributes for addition or change. If there is no current X Axis Label the Current Label data area remains blank. If an X Axis Label exists it is shown in the Current Label data area. Attributes (size, style, location, color) can be changed at any time (see the next section, "Label menu selection").

Y Axis Label You can select the Y Axis Label text and attributes for addition or change. If there is no current Y Axis Label the Current Label data area remains blank. If a Y Axis Label exists, it is shown in the Current Label data area. Attributes (size, style, location, color) can be changed at any time, as discussed in the "Label menu selection" section.

Axis Tic Labels
You can select the Axis Tic Labels for changing attributes. Tick mark labels are generated automatically using settings in the Axis Display menu. No value or entry is shown in the Current Label data area when **Axis Tic Labels** is selected.

Label menu selection

Selects between Font Selection and Label Attributes for the label chosen in the Label Display menu. Only one can be selected at a time.

Font Selection

Selects the type of font used for the label chosen in the Label Display menu. Only one font for the chosen label can be used at one time. You can use different fonts for different labels. Fonts can be changed at any time.

Bold Provides a darker character outline for the selected font. It can be used alone or with Italic and Drop Shadow attributes.

Italic Provides an angled script-like appearance for the selected font. It can be used alone or with Bold and Drop Shadow attributes.

Drop Shadow
Provides a 3-D shadow appearance for the selected font. It can be used alone or with Bold and Italic attributes. (Note: this control may not be present on all systems.)

Label Height
Controls the display size of the label chosen in the Label Display menu. Size can range from 0 to 40 points. The current size is shown in the Label Height menu area. Place the mouse pointer in the Label Height menu area (it will highlight) and drag the mouse pointer rightward to increase or leftward to decrease size. The changing point size is shown as you move the mouse. Release the mouse button to set the new label size.

Label Attributes

Changes the position and color of the label chosen in the Label Display menu. Position can be set to one of the following:

Center Places the label in a centered position.
Left Places the label in a left of center position.
Right Places the label in a right of center position.

Color Controls

Controls the color of the label. Label color is adjusted by dragging the mouse pointer on the color bars.

Select plot submenu

This submenu, shown in Figure 120, lets you change the plot attributes (plot type, line type, color) of already plotted data.

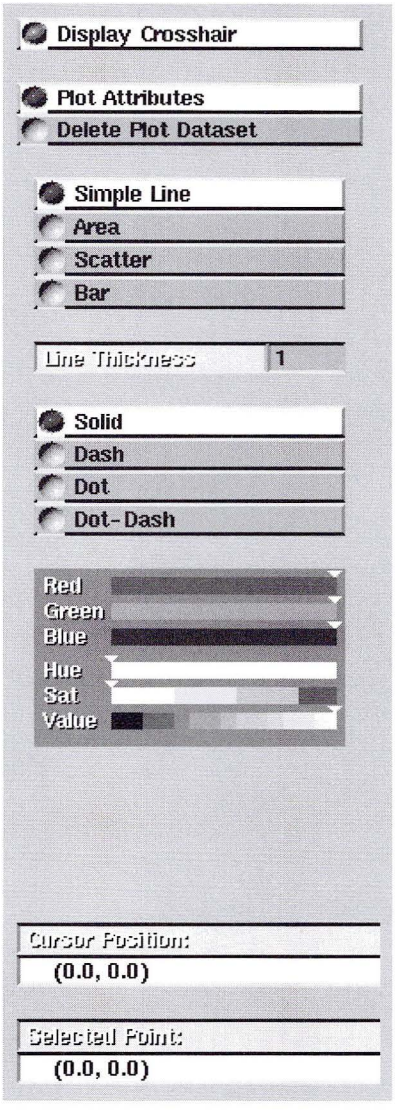


Figure 120
Select plot options

There may be multiple plots in a single plot window. Perform the following steps to select an individual plot within the current plot window:

1. Choose **Select Plot** on the Option Selection menu.
2. Position the mouse pointer over the plot to be selected.
3. Click any mouse button to select the plot.

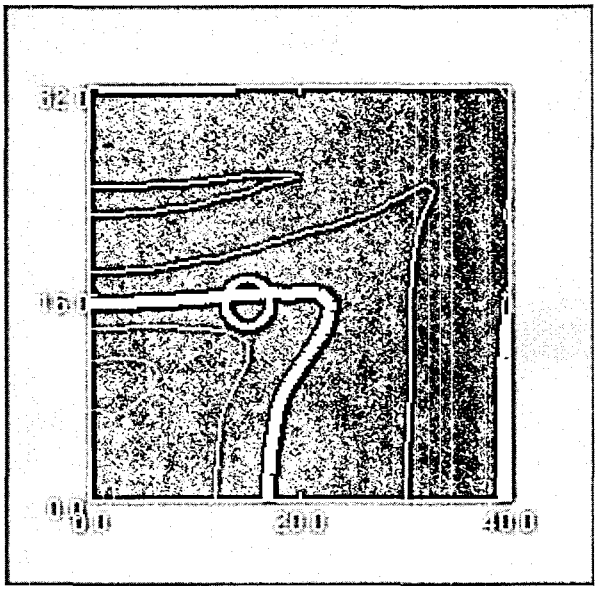
The plot changes color to white to show it has been selected. The selected point on the plot is highlighted with a circle (O). Note that clicking the mouse button finds the closest real data point, not the closest line.

The plot remains selected until another plot (if any) within the same window is selected. Perform the above steps to select another data set plot within the same window.

Display Crosshair

Displays a cross hair marker as the mouse pointer moves over the window display area. This makes the mouse pointer location within the window area easily seen (see also "Cursor Position," and "Selected Point," at the end of this section).

Figure 121
Cross hair with a selected contour plot



The default setting is for the cross hair to be displayed. When the **Display Crosshair** menu item is highlighted, the cross hair is displayed. Unlighted means the cross hair is not displayed.

Plot attributes

These controls change the plot type and plot attributes. Plot type can be changed at any time. Line type is available only when the plot type is Simple Line.

Simple Line

Displays the data set as a line plot. Line type can be a solid line, dashed line, dotted line, or dot-dash line. Line thickness can also be changed. The default setting is Simple Line as a Solid with a Line Thickness of zero. Simple Line types are as follows:

Solid Displays a solid line between data points.

Dash Displays a dashed line pattern between data points.

Dot Displays a dotted line pattern between data points.

Dot-Dash Displays an alternating dot-dash pattern between data points.

Line Thickness

A type-in that controls the line thickness of a data set plotted as a Simple Line. The default line thickness is zero. Line Thickness applies to all four Simple Line types.

Area

Displays the data set as an area plot. Coloring can be changed at any time.

Scatter

Displays the data set as a scatter plot. The default scatter plot symbol is the asterisk (*) character. Font style, height and coloring can be changed at any time.

Scatter Plot Symbol

The symbol can be changed to any displayable character by deleting the default asterisk (*) and typing in the character you want.

Symbol Height

A slider bar that controls the height of the scatter plot symbol. Height can range from 0 to 40 points. The current size is shown in the Height widget.

To change the height, place the mouse pointer in the Symbol Height menu area (it will highlight) and drag the mouse rightward to increase or leftward to decrease size. The changing point size is shown as you move the mouse. Release the mouse button to set the new size.

The actual size displayed on your screen depends upon the X Window font sizes installed. The size shown in the text widget is the size that is printed on PostScript printers.

Bar

Displays the data set as a bar plot. Coloring can be changed at any time.

Color Control

Controls the color of the plot lines, bars, area, or scatters. This can be changed at any time by selecting the data set and dragging the mouse pointer on the colorbar(s) you want. You cannot change the color of the black, background plotting field. However, you can replace it with an ConvexAVS image file.

Delete Plot data set

Deletes the currently-selected plot from the current plot window. Once deleted, the plot is lost forever. Confirmation of the delete is required. Once deleted, the plot is lost forever.

Cursor Position

Shows the relative position of the mouse pointer within the plot axis coordinates, shown in FIgure 122. Coordinates are based entirely upon the axis values within the plot.

For example, a plot with an X Axis range from 0 to 60 and a Y Axis range from 0 to 100 results in a minimum coordinate of 0.00; 0.00 and maximum coordinate 100.00; 60.00 displayed.

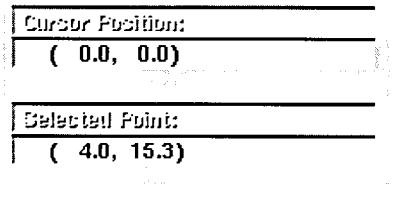


Figure 122
Cursor position and selected point display

Moving the mouse pointer within the plot area causes the coordinate display to vary. The left coordinate is the X Axis and the right coordinate is the Y Axis.

Selected Point

Shows the coordinate value of the selected point within the plot. Select a point by placing the mouse pointer within the plot area and clicking any mouse button. The X-Y Axis coordinate for that point is then displayed in the Selected Point data area. This control does not display the data value; only the coordinates.

Data types and import strategies

7

Primitive and aggregate data

ConvexAVS supports two types of data: primitive and aggregate. Primitive data includes simple scalar types and text strings. They are the basic building blocks of aggregate data types and are also used to represent parameters. 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 floating-point numbers.
  - *String* and *string block* implement simple text strings.
- Aggregate:
  - *Field* implements *n*-dimensional arrays with scalar or vector data at each point. Fields can also map rectilinear or irregular sets of points in arbitrary coordinate systems to scalar or vector data. Fields can contain floating-point (single and double precision), 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 geometry library.
  - *Pixel map* refers to the X server's representation of the rendered form of an image.
  - *Unstructured cell data* (UCD) associates data and discrete geometric objects within a single structure. The UCD structure is a generalization of the field concept to more complex topologies.
  - *User-defined data* helps you define a local data structure and pass it to other modules that understand that particular data structure.

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. Bytes represent unsigned integers in the range 0–255.

Bytes are used by aggregate data types to represent pixel or voxel intensities. If the intensities are discrete and within the specified range, byte data significantly reduces the amount of memory required to store a scientific data set when compared to integers or floating-point values.

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`.

Integers are used by aggregate data when discrete values are appropriate but greater than the range afforded by bytes. For example, integers can be used to represent voxel intensities. Integers can also be used as parameters. Examples include data replication (zoom=2), downsizing (downsize=4), or a 2-D orthogonal slice plane setting within a 3-D volume of data (k=21).

Real (or float)

ConvexAVS supports floating-point numbers in IEEE format. They are declared using the type real. This corresponds to the C type float and to the FORTRAN type `REAL` or `REAL*4`. Single-precision floating-point values are four bytes and double-precision floating-point values are two words, or eight bytes.

Single-precision floating-point data values are used in aggregate data to represent continuously varying values. Double-precision floating-point data is used in cases where the accuracy of single-precision is not great enough to contain the significant components of the input data values.

String

Text strings are 1-D character strings. A character string is declared using the type string. It is passed to a computation routine in C as a `char *` and to a subroutine in FORTRAN as a `CHARACTER *(*)`.

Text string parameters are used to represent file names. Aggregate data uses text strings for labels on elements of data. For example, in a multidimensional fluid flow data set, a text string might be used as a label for each of the vector components: density, x-momentum, and y-momentum.

There is also a multiple-line string parameter of type `string_block`. It is a character string that expects to handle embedded newlines.

Declarations and references

The type declarations in Table 11 are used for arguments to module computation functions that correspond to input ports, parameters, and output ports.

Table 11
Field declarations for data types

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 **`	`INTEGER` (or multiple args)	`INTEGER` (or multiple args)
Colormap	`AVScolormap *`	`AVScolormap **`	`INTEGER` (or multiple args)	`INTEGER`
Geometry	`GEOMedit_list`	`GEOMedit_list *`	`INTEGER`	`INTEGER`
Pixel map	`AVSpixdata *`	`AVSpixdata **`	--	--
UCD	`UCD_structure *`	`UCD_structure **`	`INTEGER`	`INTEGER`
User	structure *	structure	`INTEGER`	`INTEGER`

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 (indirect 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 3-D real", dim);
 ... compute ...
 *field_out = result;
 return(1);
}
```

The way to access a field input or output port in FORTRAN is to pass a field as a single integer argument that is used by many of the same field access functions that a C module calls, as well as by additional functions provided specifically for FORTRAN. The module must include an `AVSset_module_flags` call to use the single argument approach. This is illustrated in the /usr/avs/examples/test_fld2_f.f file.

A field is now passed to a FORTRAN routine as a single argument. Colormaps and user-defined data can also be input as single arguments. A FORTRAN routine cannot take a pixel map as an argument.

Table 12 presents some scientific applications and the ConvexAVS data types frequently used to represent them.

Table 12
Application and data type cross reference

Application	Type	Examples
Computational fluid dynamics (CFD)	Field	Simulation grids
		Finite difference data
	Geometry	Structures
	UCD	Finite element model and data
Mechanical computer-aided engineering (MCAE)	Field	Finite difference data
	Geometry	Structures
	UCD	Finite element model and data
Molecular chemistry	Field	Microscopic and x-ray imaging
		Scattered data
	Geometry	Ball and stick models
Quantum chemistry	Field	Molecular orbitals
		Molecular interactions
		Subatomic particles
Seismology	Field, UCD	3-D volumes—sampled data
		3-D volumes—simulated data
	Geometry	Structures
Strategic imaging	Field	2-D images
		Digital terrain maps
		Multispectral images
	Geometry	Structures
Medical imaging	Field	2-D images—CIT, MRI, PET scans
		X-rays
		3-D volumes—CIT, MRI, PET scans
	Geometry	Anatomical structures
Meteorology	Field	2-D images
		3-D volumes—sampled data
		3-D volumes—simulated data

Importing data choices

There are two choices you have when importing data:

- Use an existing module.
- Write a customized new module.

You must first determine if an existing ConvexAVS module can satisfy your data input requirements. To do so, you must understand the data types, the existing modules for reading data, and the format of external ConvexAVS-compatible data.

If an existing module is not available for importing your data, you will have to write a new module tailored to match your specific data format. At this point, several options are available:

- Write a shell-level *translator* that converts your data format into the appropriate ConvexAVS data format.
- Modify your application to produce data in ConvexAVS format.
- Write a module that converts your data format into the appropriate ConvexAVS data format.
- Modify your application as a module that can be incorporated into ConvexAVS networks.

If programming is required, the most effective use of ConvexAVS will involve writing a module. Once written, the module can become part of many different visualization networks and can easily be modified and enhanced to satisfy additional requirements. ConvexAVS has example module source programs in FORTRAN and C for each of its internal data formats located in the /usr/avs/examples directory. You can use these samples as a guide to writing your own modules.

There are many additional benefits to incorporating a data-producing application into a ConvexAVS module. Once an application has been transformed into a module, you can gain interactive control over various parameters (for example, starting conditions, boundary conditions, constraints). In effect, the process of visualization can be tightly integrated with the simulation or data acquisition process.

Importing fields

A field is a generalization of the familiar array structure. Whereas each element of an array has a single data value (for example, byte or integer), each element of a 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). The field can include coordinate data so that each field element is mapped to a real-world location. Refer to Chapter 8, "Fields," on page 293, for more detailed description about fields.

Field components

The field data type has the following components:

ndim	The number of computational dimensions in the field. For an image, ndim = 2, and for a volume, ndim = 3.
dim1 dim2 dim3	The dimension size of each axis (the array bound for each dimension of the computational array). The number of dim*x* entries is based on the value of ndim.
nspace	The dimensionality of the physical space that corresponds to the computational space (number of physical coordinates per field element).
veclen	The number of data values for each field element. All the data values must be of the same primitive type 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_type	The primitive data type of all the data values. Must be byte, integer, single, or double.
field_type	The field type. Must be uniform, rectilinear, or irregular.
min_ext max_ext	The minimum and maximum coordinate value that any member data point occupies in space for each axis in the data.
min_data max_data	The minimum and maximum data value in the field.

labels — A title for each of the individual elements in a vector of values.

units — A string that describes the unit of measurement for each vector element.

field_data — The data values for the field.

coordinate_data — The computational-to-physical space mapping coordinate data values.

read field

The **read field** module has two input modes, *native field input* and *data-parsing input*. In its first input mode, it reads a field data structure from a disk file into a network. In its second input mode, it converts data stored in ASCII, FORTRAN unformatted, or pure binary data files into field format.

Hints on usage

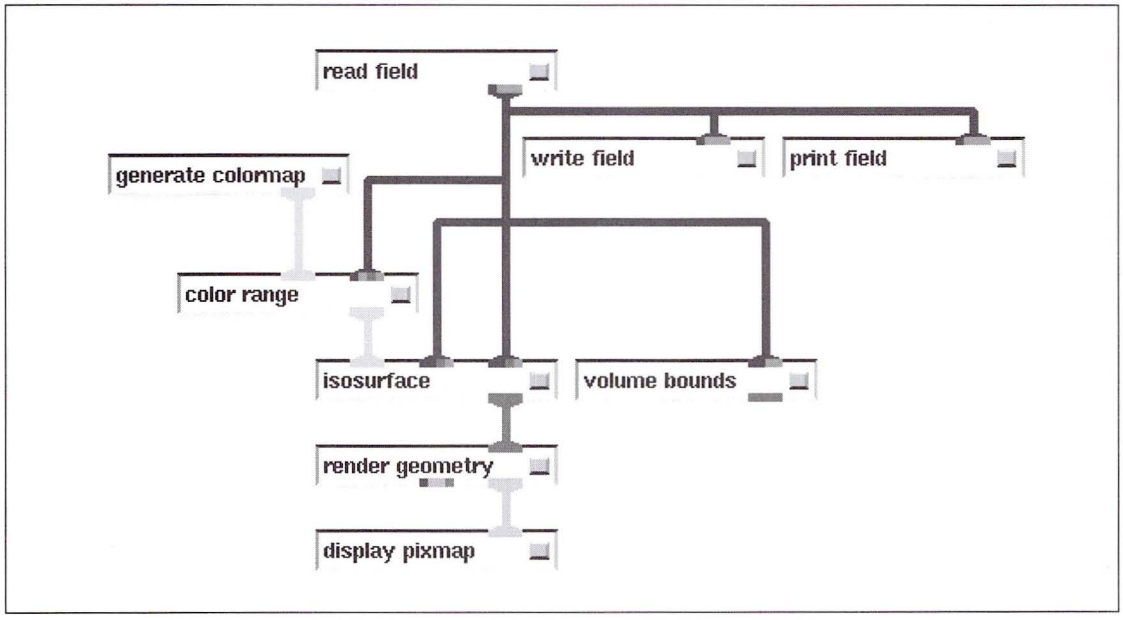
If you did not write the application that produces your data, and you use an application acquired in binary form from an outside source, then you may not know what the format of the application's binary or unformatted FORTRAN data files actually is. This can be difficult to determine just by looking at the raw data files. However, most established packages do document their internal file format, and you can usually contact the vendor's support organization for this information.

It can be difficult to tell if you read a data set correctly. To check on this, use the **print field** module. The **print field** module will display the contents of a field in ASCII format. You can look at the results in **print field**'s output display window. If this display window is too narrow, you can use the Layout Editor, described in "Redesigning the user interface," on page 120, to create a larger window. The **print field** module also writes its output to a temporary file that you can view with any text editor.

You can also use the **volume bounds** module to outline the resultant field's extents. If there has been a mistake reading the coordinate information for rectilinear and irregular data sets, it may show up with **volume bounds**.

Figure 123 shows one useful network for importing uniform or irregular volume data.

Figure 123
Sample network for importing volume data



You can replace **isosurface** with any of the modules that produce a colored geometry picture of the data (such as the **bubbleviz/scatter dots** module pair). If you are trying to produce a vector field, it might be easier to try reading just one vector element at first until all the other variables are correct. You would then include an **extract scalar** module between **read field** and **isosurface** in this network.

Once you have read the data correctly, you should write it permanently to disk in binary native field format with the **write field** module. **read field** reads native format fields much more quickly than it is able to parse foreign-format files.

You can also be creative with **read field**. For example, many imaging applications produce a series of separate images that actually represent slices through a volume (such as density slices through a cranium). you may wish to deal with the images as a 3-D volume of data. You could write a very simple C or FORTRAN program that generates the uniform X- and Y-coordinates of the images as ASCII data, as well as a series of rectilinear Z-coordinates that position the slices accurately in space. You can then use **read field** to read the original data file and the new coordinate file to produce a rectilinear 3-D volume representation of the data.

Limitations

read field does have limitations:

- Its parsing commands primarily deal with files that are structured with header information that can be skipped over, followed by data in a regular, repeating format. **read field**, for example, could not be used to read an image file that contained run-length encoded data.
- In many established packages (including PLOT3D), the data that is stored in the input data file is not the final data that is actually used by the packages when they perform their computations. For instance, values such as temperature or friction may be calculated from the raw data using well-known constant values and formulae. **read field** does not know what these automatic calculations should be. It may be necessary to write modules that further process the original field data into its final, usable form.

read PLOT3D

The **read PLOT3D** module reads computational fluid dynamics data files in the National Aeronautics and Space Administration's PLOT3D format and converts them into field format. There are two types of PLOT3D files:

- XYZ grid files that specify the irregular coordinate information
- Q solution files that contain a vector of values for each point in the grid

XYZ and Q file pairs can contain a single set of grid/data mappings or multiple grid/data mappings. The XYZ file can also contain an IBLANK value for each point, although these data values will not be stored in the output field. The data within the files can be in binary, FORTRAN formatted, or FORTRAN unformatted form. XYZ grid file and Q solution file formats must match in all respects.

read PLOT3D requires that you know the format (dimensionality, whole/plane, number of grids, binary/formatted/unformatted, and whether IBLANK values are present) of the PLOT3D files that you are trying to read. It does not check to verify that the values it is given map reasonably to the data.

Q solution files contain three to five floating point values for each point in the grid: X-momentum (1-D), Y-momentum (1-D and 2-D), Z-momentum (1-D, 2-D, and 3-D), density, and

stagnation. The four header values (FSMACH, ALPHA, RE, and TIME) are ignored.

read PLOT3D does impose some practical limits to the size of the data: No one dimension can be larger than 1,000,000; the output data can have no more than 1,000,000,000 points in any one grid; and the maximum number of data grids is 50.

read PLOT3D outputs a field (field irregular float 1-D, 2-D, or 3-D of 3-, 4-, or 5-vector). The field output will match the dimensionality of the original PLOT3D data set. At each point in the grid will be three to five floating point values: density, X-momentum (and Y-momentum, and Z-momentum, if appropriate), and stagnation, in that order. The output field represents only the first grid of multi-grid parameter files. There is no way to pack multiple grids into a field.

An additional module, **cfd values**, is often used in conjunction with **read PLOT3D**. **cfd values** inputs the field data from **read PLOT3D** (including density, X-momentum, Y-momentum, Z-momentum, and stagnation), and allows you to compute the following values:

- Energy
- Pressure
- Enthalpy
- Mach number
- Temperature
- Total pressure
- Total temperature

Note

A ConvexAVS-specific version of the module is available as read PLOT3D (note the capitalization in the name). The read PLOT3D module is in the unsupported module library, while the read PLOT3D module is in the supported module library. Refer to the *ConvexAVS Module Reference* for more information.

read image

Although there is a distinct image file format for storing images on disk, once ConvexAVS reads an image into a network, it treats image data as a 2-D, 4-vector byte field. Therefore, the best way to approach importing image data is to view the problem as converting your image data into a field.

read image data file format

The **read image** module and the Image Viewer subsystem (also implemented as the **image viewer** module) can read a file that contains an image—a 2-D 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, which 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 by row; 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 modules to represent an opacity/transparency value. Figure 124 shows the components of a pixel.

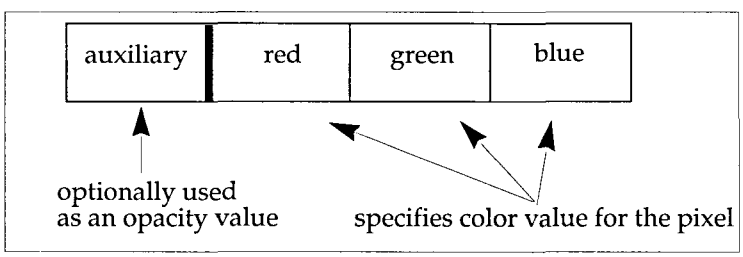


Figure 124
Rectilinear field

The image data file format is:

```
number_of_pixels_in_x_dimension
number_of_pixels_in_y_dimension
pixel_1
pixel_2
pixel_3
.
.
.
pixel_n
```

The total number of pixels is:

*number_of_pixels_in_x_dimension * number_of_pixels_in_y_dimension*

The image data type is primarily designed to hold data that was originally produced as RGB (red, green, blue) data, whether as full 24-bit true color or some more limited format. Many other kinds of scientific data are also referred to as image data. For example, medical imaging devices produce 2-D or 3-D arrays of numbers that represent some quantity, such as density. This is usually called image data. However, such data is best represented as a field, not as a .x image. A field can always be converted into an image that the Image Viewer can manipulate.

read volume

Although there is a distinct volume file format for storing 3-D uniform volumes on disk (where each element is a byte holding a value from 0-255), once ConvexAVS reads a volume into a network, it treats the volume data as a 3-D uniform scalar byte field. The best way to approach importing this narrow class of volume data is to view the problem as converting your volume data into a field.

read volume data file format

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

The 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.

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 255 by 255 by 255.

The remaining contents of the file are the values of a 3-D array of bytes, in column-major order. For example, the values in a 50 by 20 by 10 array, using FORTRAN notation, would be stored as:

```
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)
```

The volume data file format is:

```
number_of_voxels_in_x_dimension
number_of_voxels_in_y_dimension
number_of_voxels_in_z_dimension
voxel_1
voxel_2
voxel_3
...
voxel_n
```

For this example, the total number of voxels is:

number_of_voxels_in_x_dimension * *number_of_voxels_in_y_dimension* * *number_of_voxels_in_z_dimension*

Examples

If none of the above alternatives for importing data into the field format is satisfactory, you will need to write a module that will read the data into field format. ConvexAVS contains a library of routines for building fields and storing and accessing field elements.

There are several source code examples of modules that read data into field format. The following are found in the /usr/avs/examples directory:

read_image.c read_image_f.f	These modules read data in image file format into a 2-D, 4-vector byte field.
read_vol.c read_vol_f.f	These modules read data in volume file format into a 3-D scalar byte field.
read_scans.c	This module reads a set of hypothetically-formatted images stored in separate files, each representing a planar scan, into a uniform 3-D integer field (a 3-D volume of integer data). It assumes a certain location for the input files and a pattern to their file names; fixed-size images (64 by 64), a fixed-size header in each image file, an integer of data for each pixel, and that the image data is stored in row major order (that is, that the second index of the data varies most rapidly). This source file can be used as a template for a variety of image file formats that need to be converted to 2-D or 3-D fields.
read_PLOT3D.c	This file reads NASA PLOT3D whole format data sets into field format. This is probably the most sophisticated example source code module. Because its module description section includes the definition of several different kinds of interface widgets, it must: deal with allocating memory to store the field based upon the changing size of the input data sets, read column-major order data, read vector data instead of simple scalars, and use formulae to produce output values not found in the input data.

Importing geometries

Geometry data contains a description of three-dimensional geometries and scenes, including object type, coordinate data, surface attributes, rendering modes, material properties, and transformations.

Objects are one of the following types:

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 2-D 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.

render geometry

The **render geometry** module provides access within a 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 are 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**.

read geom

The **read geom** module reads a file containing a geometry and outputs the geometry to one or more modules connected to its output port. The resulting object is named after the file from which it was read. This module reads geometry files only.

pdb to geom

The **pdb to geom** module converts molecular data in *Protein Data Bank* (PDB) format into geometry format. It is in the Data Input column of the Network Editor's module palette. There are two sample input data sets,
/usr/avs/filter/example/bwdna.pdb and
/usr/avs/filter/example/crambin.pdb.

For instance, Table 13 shows part of a file written in the Brookhaven PDB format. This file defines the structure of a particular protein molecule called *crambin*. There is a data input module to read files in this format. You can supply your own data input modules for other data formats.

Table 13
Data file in Brookhaven PDB 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

Geometry filters

There are a number of geometry filters supplied with ConvexAVS that will convert data into the geometry format. These filters fall into two classes:

- Filters that read several file formats commonly found in the technical community. These geometry filters can be used to import existing data sets into geometry format.
- Filters that read ASCII files in a format unique to ConvexAVS. Each of these geometry filters inputs ASCII data to create one of the fundamental geometry structures manipulated by the geometry programming library, such as polygons, polyhedrons, meshes, and spheres. These ConvexAVS-specific geometry filters can be used in two ways:
  - You might be able to construct an ASCII file version of your geometry data that one of these geometry filters could read to create an object. The mesh.c filter has the

most general utility.

- The geometry filters serve as example programs that show how to construct the primitive geometry structures using geometry calls. Two of the geometry filters are written in FORTRAN in addition to C.

The sources to geometry filters are all located in the directory /usr/avs/filter. The executable versions of geometry filters are all located in the /usr/avs/bin directory. Sample data sets in the formats these geometry filters expect are located in the /usr/avs/filter/example directory. Table 14 shows general file formats and their geometry filters.

Table 14
General filters

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

There is a sample Movie BYU format data set in the /usr/avs/filter/example/cube.byu file and a sample Mathematica ThreeScript data set in the /usr/avs/filter/example/cone.ts file.

Table 15 shows specific geometry filters that are provided.

Table 15
Specific filters

File name	Executable 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

The filters are all located in the /usr/avs/filter directory.

The source to each specific geometry filter includes a description of the ASCII file format it expects to read. There are sample polygon, polyhedron, and mesh ASCII data files in the /usr/avs/filter/example directory.

Automatic data filtering

Geometry filters are normally executed at the shell level. The geometry filter facility is unique in this regard. If, during a Read Object function within the Geometry Viewer, ConvexAVS determines that the file you select is in a known data format, it invokes the appropriate geometry filter automatically to create a corresponding geometry file on disk. It then reads in the object from the .geom file.

When you execute the **Read Object** function, you select a file name 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 /usr/avs/bin directory for utility programs named *file*_to_geom. Each such geometry filter determines a particular file name extension. For instance, the geometry filter utility wfront_to_geom determines the extension .wfront.
- It compares the extension of the **Read Object** file name with its list of geometry filter extensions.
- If there is a match, ConvexAVS automatically invokes the appropriate geometry filter utility, creating a geometry-format file of the same name, with a .geom extension. This file is created in the directory where the original file is.
- The **Read Object** function is completed by reading in the newly created .geom file.

Shell-level usage of geometry filters

Each of the geometry filters can also be invoked from the shell, in the usual fashion. Each one reads a single file from stdin and writes a single geometry-format file to stdout. You should redirect stdout to a file whose extension is .geom so that it is directly readable by ConvexAVS.

Most of the geometry filters do not accept any command line options, but there are several exceptions.

The `ts_to_geom` filter 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.

The `pdb_to_geom` filter accepts the following option:

`-balls`
: Use the sphere representation instead of the default ball-and-stick representation.

Postprocessor filters

ConvexAVS supplies an additional set of geometry filters that you can use to process the output of the geometry-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 geometry-format file, flipping the normals of the object defined therein.

Table 16 lists the postprocessor geometry filters supplied. Except for `send_to` and `animate_to`, each filter reads a file from stdin and writes a file to stdout.

Table 16
Postprocessor filters

Filter name	Description
`geom_flip`	Flip normals of object.
`geom_pickable [-non]`	Make all objects pickable or not pickable, so that their attributes can be set (or not set) individually.
`geom_scale`	Scales the object uniformly, so that it lies within the unit cube, Also, converts the object's normals to unit length (normalizes them).
`geom_to_normls [-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.
`text_to_geom`	Produces a .geom version of the input ASCII file. The combination of `geom_to_text` and `text_to_geom` can be used to transfer geometry files between machines with different byte ordering or different word sizes.
`geom_split` [*name*]	Creates a series of .geom files, each of which contains one of the objects defined in the input file. This is used for input files that contain more than one geometry object. Each output file is named *name.n*.geom. If you do not specify the optional *name*, the files are named split.0.geom, split.1.geom, and so on.
`send_to` *filename*	Causes a currently-executing ConvexAVS program to read the specified file, which should be a .geom file.
`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 ConvexAVS program to read the script file. The cycle includes the specified *geom-file* sequence.

Writing a new filter utility

This section provides pointers for those who wish to create new filter utilities using the template programs.

The basic procedure for creating a geometry-format object is:

1. Decide which of the geometry-format objects conforms most closely to the application data:
   - Polyhedron
   - Polygon
   - Mesh
   - Sphere
   - Polytriangle

   No tools exist for direct conversion of non-linear geometries, such as spline surfaces and quadrics.
2. Create an instance of that geometry-format object.
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 following sections describe the steps for converting a variety of object types into geometry format.

Converting a polyhedron

Start with the polyh.c template, then:

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 polygon.c or polygon.f template, then:

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 mesh.c or mesh.f template, then:

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 mesh.c or mesh.f template, then:

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 sphere.c template, then 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 a polytriangle object.
2. Add zero or more polylines to this object.

Importing unstructured cell data

Unstructured cell data (UCD) is commonly used in structural analysis and computational fluid dynamics. A UCD structure consists of an irregular coordinate structure (or model) made up of cells. Cells may be:

- Points
- Lines
- Triangles
- Quadrilaterals
- Tetrahedrons
- Pyramids
- Prisms
- Hexahedrons

Each cell has a corresponding number of nodes. Data can be associated with the entire structure, with each cell, and with each node. The data is structured as a set of components. Each component can be either a scalar or a vector.

read ucd

The **read ucd** module reads a UCD structure from a file, which must have a .inp suffix. The file may be ASCII or binary.

Binary UCD files have a different format than ASCII UCD files.

Examples

read_ucd.c

The /usr/avs/examples/read_ucd.c file is the C program source to the **read ucd** module. It reads either a binary or ASCII format UCD data set and creates the UCD structure format used by the various UCD modules.

gen_ucd.f

The /usr/avs/examples/gen_ucd.f file is the FORTRAN source to a module that first generates its own scalar or vector data, then writes it out as a UCD structure.

Importing colormaps

A colormap is a data structure that implements a transfer function that assigns a color to each value between an upper and a lower bound. A colormap consists of:

- Four arrays of floating-point values, one each for hue, saturation, brightness, and opacity. Each value is normalized between 0.0 and 1.0 inclusive.
- An integer indicating the number of colors—this is the length of each of the four arrays.
- Floating-point lower and upper bounds that determine the resolution of the colormap. The lower bound is a data value that maps to the first element of each array. The upper bound is a data value that maps to the last element in each array. All intermediate data values are mapped to elements within the arrays.

The **generate colormap** module produces a colormap data structure, for use by modules that transform input data into color values. This module also allows you to both read and write ConvexAVS-compatible colormaps. These 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
```

All values in this file are in floating-point format, with the exception of `number_of_entries`, which is an integer. The hue, saturation, brightness, and opacity values are normalized to the range 0.0-1.0. For examples of files containing colormaps, look in the /usr/avs/data/colormap directory.

Fields

8

Definition of fields

A ConvexAVS field is a computer implementation of a mathematical relation between two sets of variables. Such relations are often called functions or *mappings*. One of the two sets of variables is usually considered to be *independent* and the other *dependent*.

The independent set of variables can be thought of as the mesh upon which the function is defined. At each point in the mesh, there is a representative of the dependent set that maps to that point. This is the value of the function at that point. In the following sections, we will call the independent data, *points*, and the dependent data, *data*. This reflects the naming convention used in the data structure that implements a field in the C programming language. All current mapper modules convert the representation (color, arrow, and so on) of each dependent data element to a position in the geometry determined by the coordinates of the point. Understanding the difference between the points and the data is vital to understanding fields and how they are converted by the mapper modules.

Points and data

An easy to understand example that illustrates a field is the temperature distribution on an object. At each point on the object, we can measure the temperature. The coordinates of each point at which we sample the temperature make up the independent variable or points array. These coordinate values are stored in the points portion of the field. The temperature value that we measure is viewed as dependent on the point at which it is measured and is considered the data portion of the field. ConvexAVS fields support a great variety of possibilities in the geometry of the independent variables (points) and the nature of the dependent variables (data).

To understand the possible variations in the geometry of the point mesh that fields support, look at our temperature example again. There are a variety of objects upon which we might want to measure temperature. For some objects, like a thin bar of metal, the points can be modeled as a subset of the Euclidean plane. Each point has an (x, y) coordinate pair that tells where it is on the bar. Other objects, like a room for example, can be modeled as a subset of 3D Euclidean space. In this case, each point has an (x, y, z) coordinate triple that tells where it is in the room. Still other objects are more complicated: points on the surface of the earth, for example, can be represented by either pairs of longitude/latitude coordinates or by coordinate triples (x, y, z). As this last example shows, there are two ways of thinking about the dimensionality of a field. More detailed examples are in "Field mapping examples," on page 297.

Field classes

Fields can be classified according to several criteria:

- The number of volumes represented in the dependent data for each point in the independent data. This number is called the *veclen* of the field. Scalar data would have a veclen = 1 and vector data would have a veclen = 2 (or more).
- The primitive data type used to represent the dependent data (byte, integer, float, or double).
- The geometric complexity of the mesh or points array (uniform, rectilinear, or irregular).

The first two criteria above are rather straightforward, but the third requires more explanation. The geometric complexity of the independent data affects not only the way that the independent data is stored, but also the way that the dependent data is stored. It is possible that the points on which the field is defined may lie on a surface (2D) but this surface may lie in 3D space. This is the case for the example above where we had temperature values defined on the surface of the earth. The surface of the earth is intrinsically 2D, that is, it only requires two numbers (longitude and latitude) to specify a point unambiguously. However, this surface is the boundary of a 3D volume. The way ConvexAVS deals with this is to have two numbers represent the dimensionality of the field: ndim (the intrinsic dimensionality) and nspace (the dimensionality of the surrounding space). In the case of the earth's surface, ndim = 2 and nspace = 3.

The ndim value determines the number of indices needed to access both the points and the data but in some cases this condition can be relaxed. There are three levels of geometric complexity supported by fields:

- Irregular fields
- Rectilinear fields
- Uniform fields

Irregular fields

Irregular fields are the most general type of field possible but are the simplest to describe. Both the points and the data can be thought of as ndim+1 dimensional arrays. The reason the dimensionality of the arrays is one larger than ndim is because we have to be able to access each component of the point (x, y, z) and each component of the vector of dependent data (vx, vy, vz) if the veclen is greater than one. Conceptually, we can think of the coordinates and the data being mapped to a computational space that has ndim dimensions. Points that map to adjacent positions in computational space (for example, (i, j, k) and (i+1, j, k)) are also adjacent in physical space. It is very important to remember that a line in computational space (for example, i varying but j and k constant) might actually be a curve in physical space. Likewise, a plane in computational space (for example, constant k but varying i and j) might be a curved surface in physical space. This is why the **orthogonal slicer** module often creates curvilinear slices when applied to irregular fields. For a more concrete description of the layout of the points and data arrays, see "Field components," on page 302.

Rectilinear fields

Rectilinear fields are very similar to irregular fields but the x–coordinate only depends on the index i in computational space, y only depends on j, and z only depends on k. This requires less space to represent the array for the point. The data array is indexed just like it is in irregular fields.

Uniform fields

The geometry of uniform fields is the simplest but their representation is the most complicated. Uniform fields are uniform rectangular grids of points with their associated dependent values. The spacing between points is fixed in each dimension of the physical space (nspace) and the intrinsic dimension of the computational space (ndim) are identical. Where the complication comes in is in the specification of the physical extent and location of the geometry of the independent data. There are two ways of specifying this geometry: implicitly (that is, by default) and explicitly. If, in the creation of the data, no values are supplied for the points array, then the independent data is assumed to lie in the rectangular region, (0 - maxi) by (0 - maxj) by (0 - maxk).

The coordinate functions are:

```
x(i) = i
y(j) =
z(k) = k
```

If, however, the points array is filled in, then the coordinate functions are:

```
x(i) = points[0] + i * (points[1] - points[0])/(dimensions[0]-1)
y(j) = points[2] + j * (points[3] - points[2])/(dimensions[1]-1)
z(k) = points[4] + k * (points[5] - points[4])/(dimensions[2]-1)
```

In the current implementation of uniform fields, it is important to set both points[] and max_extent[]/min_extent[] or set neither points[] or max_extent[]/min_extent[]. Mappers may not work correctly if either the extents and points do not match. The data array is indexed just like it is in irregular fields. Table 17 shows mapping information for the three types of fields.

Table 17
Field mappings

Mapping	Mapping information		Coordinates
Uniform	Implicit or explicit	Computational dimension to coordinate axis	X = points[0] + i * (points[1] - points[0]/dimensions[0]-1 ...
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, ...) ...

Field mapping examples

This section presents examples of fields and their mappings from computational to coordinate space.

Uniform field 1

A 2D image is represented by a mesh of data elements, each of which specifies the value of a pixel. Each data element is a vector of four bytes that specify the three color components and an alpha channel. The field consists of 65,536 elements, each with four values:

$\{ V_n(i,j), i=1,256, j=1,256, n=1, 4 \}$

The field is uniform.

The 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

Uniform field 2

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.

The 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

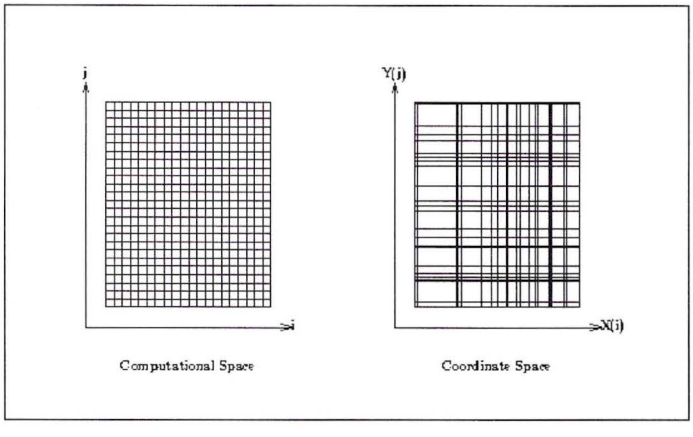
Rectilinear field

A scalar field is defined as a 2D mesh with nonconstant 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 125 shows the mapping between computational and coordinate space.

Figure 125
Rectilinear field



The 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

Rectilinear or irregular field

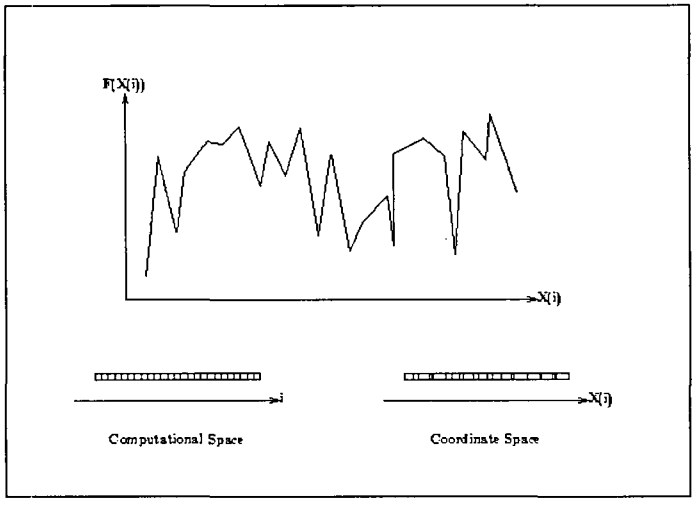
A data set consists of 25 data elements, each representing $F(X)$ for a given value of X. The field consists of 25 elements:

$\{ F(X(i)), i=1,25 \}$

Computational space is 1D with 25 values for $F(X)$. Coordinate space is also 1D 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 126 shows the mapping between computational and coordinate 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 126
Rectilinear or irregular field



The 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 consists 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

Irregular field 1

A 2D 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 127 shows the mapping between computational and coordinate space.

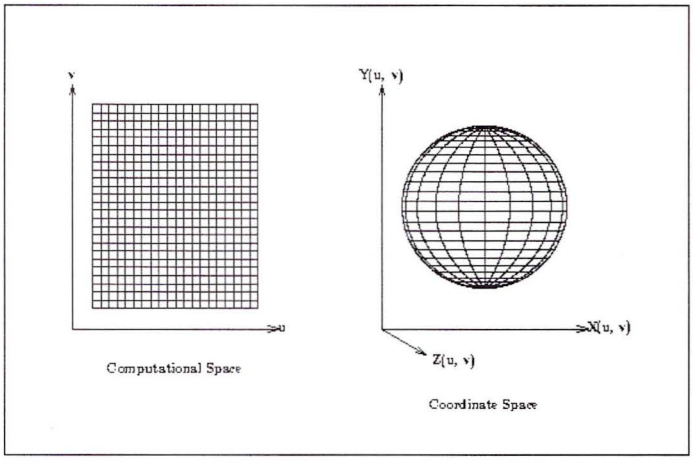


Figure 127
Irregular field

The 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

Irregular field 2

A fluid-dynamics application is a 3D 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 128 shows the mapping between computational and coordinate space.

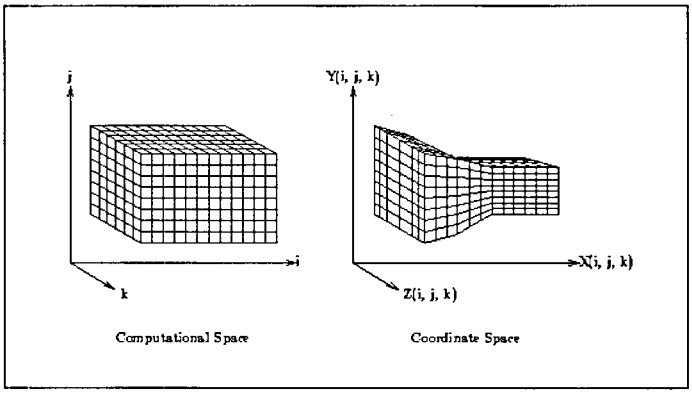


Figure 128
Irregular field

The 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

Fields have 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.
- MIN/MAX information for computational data elements. The minimum values for each variable in the array of data elements are stored in an array whose data type is the same as that of the data elements. The size of this array is equal to the vector length of the field. The maximum values for each variable in the array of data elements are also stored in an array whose data type is the same as that of the data elements. The size of this array is equal to the vector length of the field.
- MIN/MAX extents for coordinates in each dimension of nspace. The minimum extent is an array of floating-point numbers with a size equal to the number of dimensions in coordinate space. The maximum extent is also an array of floating-point numbers with a size equal to the number of dimensions in coordinate space.
- Labeling information for each vector element in the array of computational data. The labels are stored in a character array with a delimiter character as the first character in the array. The delimiter is followed by string or delimiter pairs. The number of pairs is equal to the vector length of the field.

The labels are useful for defining what each variable in the array of data elements is. For instance, one variable might be temperature, a second pressure, and a third density.

- The unit label associated with each vector element in the array of computational data. This is a character array with a delimiter character as the first character in the array. The delimiter is followed by string or delimiter pairs. The number of pairs is equal to the vector length of the field. The unit labels are useful for defining measurement units for each variable in the array of data. For instance, one variable unit might be degrees centigrade and another pounds per square inch.
- 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, and so on. 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)
```

- 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 uniform, rectilinear, or irregular field, an array of floating-point values representing the coordinates of the field:
  - For a uniform field, coordinate information is limited to minimum and maximum extent full-word values for each physical dimension (nspace) of the data. The minimum and maximum extent values in the coordinate binary area are the same as the min_ext and max_ext values in the field data structure, unless the field has been cropped, downsized, or interpolated.

    If the field has been cropped, downsized, or interpolated, the field data structure contains the original field's min_ext and max_ext values, while the coordinate section of the binary area contains the minimum and maximum extent of the subset data.

    Mapper modules can use this additional extent information to properly locate their geometric representation of the subset data in world coordinate space.

    The extents in the coordinate binary area are stored in the following order:

    ```
    minimum X, maximum X
    minimum Y, maximum Y
    minimum Z, maximum Z
    ```

  - 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, and so on. 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, and so on. 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, and so on) 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, and so on. The subscript for the first dimension of computational space varies fastest, followed by the subscript for the second dimension of computational space, and so on. The subscript for the dimension of coordinate space (X, Y, and so on) 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)
```

Declaring fields

When declaring or allocating fields, use 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, you 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.

The module's computation routine should then 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 18 lists possible specializing words.

Table 18
Field declarations and specializing words

Field component	Value	Specializing words
Number of dimensions	*n*	*n*D
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

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 computational 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 the field.h include file. Each field structure supports a different data type:

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 129 (using `AVSfield_char` as an example).

Figure 129
Example of `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;
```

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:

Member	Value
ndim	3
nspace	3
veclen	5
type	AVS_TYPE_REAL
size	sizeof(float)
single_block	True if field is single malloc
uniform	IRREGULAR
dimensions	dims[3] = {12, 8, 6}
points	coords[3][6][8][12]
data	data[6][8][12][5]
min_extent	Minimum extent of coordinates in each dimension
max_extent	Maximum extent of coordinates in each dimension
labels	Labels for each component
minimum	Minimum data value for each component
maximum	Maximum data value for each component
shm_key	Shared memory key
shm_id	Shared memory identifier
shm_key	Shared memory base address
units	Units of each component in data

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.

Manipulating fields from FORTRAN

To access a field input or output port in FORTRAN, pass a single integer argument for each field of the module's computational function. You must specifically request the single integer argument mode by adding the following call to the description function for the module:

```
CALL AVSSET_MODULE_FLAGS(SINGLE_ARG_DATA)
```

The module's computational function receives a single integer argument that is a pointer to the field rather than having the components of the field passed as multiple arguments. This field pointer value can then be passed directly to field accessor routines (for example, `AVSfield_get_minmax`) in order to access any desired field element. When using the old multiple argument passing technique, it is necessary to call the `AVSport_field` routine in order to retrieve the field pointer required by the field accessor functions.

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`:

```
LOCAL_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. Examples of using accessor functions, such as `AVSfield_get_int`, are provided in the /usr/avs/examples/test_fld2_f.f file.

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 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 passes 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 points and data arrays are automatically allocated) and takes advantage of shared memory storage. For example, to create a field 3D 3-vector 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)
```

Use the `AVSfield_alloc`, `AVSfield_get_int`, and `AVSfield_get_dimensions` routines when building fields. Examples of creating fields are in the /usr/avs/examples directory. Refer to the read_image.c, read_vol.c, threshold.c, read_image_f.f, read_vol_f.f, threshold_f.f, test_fld2_f.f, and test_field_f.f files.

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. Examples of how to create a 2D uniform vector field for use as an image are in the read_image.c and read_image_f.f files in the /usr/avs/examples directory.

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. Examples of how to create a 3D scalar uniform byte field for use as a volume are in the read_vol.c and read_vol_f.f files in the /usr/avs/examples directory.

C language field macros

ConvexAVS has a number of macros that make access to the array of data elements representing the computational space of a field more convenient when you program with the C language.

Grouped by function

This section groups macros by function.

- 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

This 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 3D uniform field, COORD_X_3D returns *i*. For a 3D 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 3D uniform field, COORD_Y_3D returns *j*. For a 3D 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 3D uniform field, COORD_Z_3D returns *k*. For a 3D 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 1D field, `I1DV` provides a pointer to the first element of the vector in the data array that corresponds to the index *i*.

I2D

```
#include <avs/field.h>
I2D(field, i, j)
   AVSfield  *field;
   int       i, j;
```

For a 2D 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. 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 2D 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. 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 3D 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. 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 3D 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. 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 4D 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. 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 4D 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. 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, MAXY, MAXZ

```
#include <avs/field.h>
MAXa(field)
   AVSfield  *field;
```

where *a* is `X`, `Y`, or `Z`.

`MAXX` provides the size of the first dimension of a field.
`MAXY` provides the size of the second dimension of a field.
`MAXZ` provides the size of the third dimension of a field.

RECT_X, RECT_Y, RECT_Z

```
#include <avs/field.h>
RECT_a(field)
   AVSfield  *field;
```

where *a* is `X`, `Y`, or `Z` 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` provides a pointer to the first element of the coordinate array that corresponds to the second dimension of computational space.

`RECT_Z` provides a pointer to the first element of the coordinate array that corresponds to the third dimension of computational space.

Colormaps and pixel maps 9

Colormaps

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^-1)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 declare an input 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 declare an output colormap as:

```
INTEGER FUNCTION my_module(size, lower, upper,
                           phue, psat, pval, palpha)
 INTEGER size
 INTEGER phue, psat, pval, palpha
 REAL lower, upper
 phue = MALLOC(4*size)
 psat = MALLOC(4*size)
 pval = MALLOC(4*size)
 palpha = MALLOC(4*size)
```

Pixel maps

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; /*set to 1 if using XdbUpdateWindows */
} AVSpixdata;
```

A FORTRAN computation routine cannot take a pixel map as an argument.

Geometries 10

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 database 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

The geometry rendering module is called **render geometry**. This module provides a conduit for geometry to get from a module to the internal geometry database. The render module unpacks the edit lists and rebuilds 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 module that produces the geometry. The second is in the Geometry Viewer panel where you manipulate display parameters.

The Geometry Viewer is the interface to the internal geometry database. It provides a tool for you to view and modify representations of geometry contained in the database.

Figure 130 shows an edit list data flow.

Design factors

All geometry commands must be stored in an edit list to be passed through the 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.

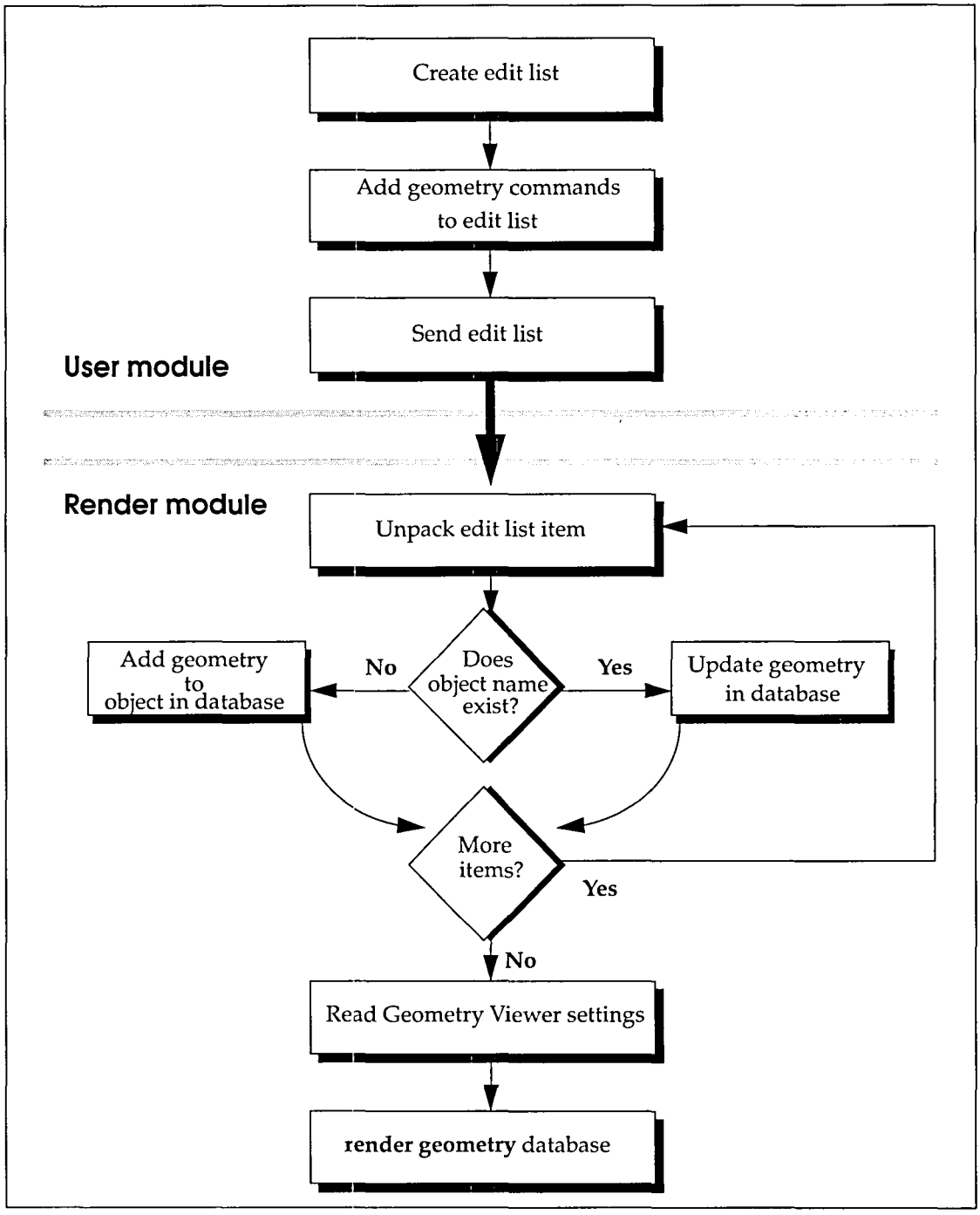
Geometry items are referenced in the database by unique strings. This means that a module must remember the name under which its geometry was created to reference it later.

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. Use as few geometry calls as possible when building a geometric description.

Performance can be improved by saving created edit lists and sending them later. This is useful if you want to send the same geometry commands more than once.

The Command Language Interface (CLI) can be used to set and query Geometry Viewer values. This can be done from modules or through networks. Refer to "Geometry Viewer commands," on page 609.

Figure 130
Diagram of an edit list data flow



Manipulating edit lists

The compute routine of most geometry producing modules starts by creating 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, a module uses the routines in the geometry library. Refer to Chapter 11, "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 geometry 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)
```

Examples of manipulating edit lists are in the /usr/avs/examples directory. The polygon.c and polygon_f.f programs are subroutines, while qix.c and qix_f.f are coroutines.

Supported objects

You can use the geometry library to create geometric objects and have ConvexAVS display them by writing a module that outputs a geometry data type containing geometric and attribute information.

Feed the output of the module to the **render geometry** module and use the Geometry Viewer to manipulate the display of the geometric data.

A geometry object contains a set of graphics primitives. A geometry object has no attributes associated with it and has a single object type. The objects and their supported types are shown in Table 19.

Table 19
Geometry objects and types

Object	Type
Label	GEOM_LABEL
Mesh	GEOM_MESH
Polyhedron	GEOM_POLYHEDRON
Polytriangle	GEOM_POLYTRI
Sphere	GEOM_SPHERE

Label objects

Label objects are used to represent text that generally annotates or titles other geometric data. There are two classes of labels:

- Titles
- Annotation labels

Titles have a location in screen coordinates and are not transformed by either the camera or the object's transformation. Annotation labels are transformed with geometry but are always parallel to the screen plane.

Mesh objects

The mesh object type contains a 2D array of vertices ordered such that adjacent vertices in the array are connected. If the dimensions of the 2D vertex array are *M* by *N*, a mesh object forms (*M*-1)*(*N*-1) quadrilaterals. A single quadrilateral is a simple mesh object with *M*=*N*=2.

Polyhedron objects

A polyhedron object contains a 2D array specifying the X-, Y-, and Z-coordinates of each vertex and a separate list of connectivity information. The connectivity list is a 1D array of integers. The first integer in the list (call it *n*) contains the number of vertices in the first polygon of the polyhedron. Following this integer are *n* consecutive integers (beginning with 1) representing the element in the vertex array that corresponds to the respective vertex data. The last index of the first polygon is followed by an integer representing the number of vertices in the second polygon. This pattern continues until the value of *n* is zero, which terminates the list. The following example shows the contents of the connectivity list used to describe a polyhedron containing two polygons, one with four vertices and the second with five vertices.

```
Connectivity list: {4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 0}
```

The index values in the connectivity list are 1-based.

The object types mesh and polyhedron contain an implied surface and wireframe description of the geometry that they represent. For the mesh object, the wireframe description is a 2D array of lines across the rows and columns of the mesh. For the polyhedron object, the wireframe description of the object is formed by the edges of each polygon in the object.

Polytriangle objects

Certain rendering platforms can make use of the shared vertices in adjacent polygons to improve the efficiency of rendering. For this reason, the polytriangle strip is a useful primitive for representing surface information. In a polytriangle strip, each vertex makes a triangle with the previous two vertices. For a list of N vertices, there are N-2 triangles. If your object is such that each vertex is shared by a number of different triangles, the polytriangle strip can be the most efficient description in which to represent your object. The geometry library provides routines that convert objects from the mesh and polyhedron primitive types into the polytriangle primitive.

Because the polytriangle primitive can only represent triangles, it does not normally contain information necessary for providing an appropriate wireframe description of an object. If an object is made up entirely of quadrilaterals and uses the polytriangle representation as its wireframe description, ConvexAVS would naturally draw edges in the object that should not be drawn. For this reason, the polytriangle object type contains both a wireframe and a surface description of the object. If the rendering mode is lines, the Geometry Viewer uses the wireframe description. If it is surface, it uses the surface description.

The surface description is an array of polytriangle strips (where each strip is a single array of connected triangles). The wireframe description contains an array of connected lines and an array of disjoint lines. Each connected line is a single array of vertices such that each vertex draws a line to the previous vertex. If a connected line has N vertices, it contains N-1 lines. The array of disjoint lines contains an even number of vertices such that each successive pair of vertices forms a line. If there are N vertices in a disjoint line, there are N/2 lines.

Sphere objects

The sphere object represents objects that contain a list of spheres or dots. Spheres have a radius as well as a location. Spheres are represented as dots if no radius information is provided.

Creating objects

Many routines can be used to create an object. The goal is to provide entry points that allow many different data formats to be simply converted to the internal geometry format. Different routines are used by different filters.

The creation routines for the geometry library define some simple data formats:

- A list of vertices is an array of floats of X, Y, and Z.
- A list of normals is an array of floats of NX, NY, and NZ.
- A list of float colors is an array of floats of R, G, B (in the range 0.0-1.0).
- A list of integer colors is also defined where R, G, and B are bytes packed into an integer using the shifts `AVS_RED_SHIFT`, `AVS_GREEN_SHIFT`, and `AVS_BLUE_SHIFT`, defined in the /usr/avs/include/port.h include file. All colors are converted internally to arrays of floats with values between 0 and 1.
- A list of extents is a 1D array of six floats in the order: xmin, xmax, ymin, ymax, zmin, zmax.
- User–supplied primitive data. You may associate a single integer with each primitive where a primitive is defined as a line or polygon.
- User–supplied vertex data. This is similar to user–supplied primitive data but is associated with each vertex instead of an entire primitive.

A geometry object is initially created without any data at all. Data is then added to the object incrementally. An example sequence would be to create an object of type polyhedron, add a polygon list, add vertices, then add normals. 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, the geometry library also provides macro 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 this object does not have this type of data.

Extents

Each object can have extent information associated with it. The extent of an object is determined by the minimum and maximum values of the coordinates of the object's vertices. Routines that create objects take optional extent information. Passing `GEOM_NULL` for this parameter indicates no extent is being specified. This is usually the best way to specify the extents unless you have explicit knowledge of what the extents should be for the object.

If you do not supply extent information during the creation of your object, it is generated for you when and if it is needed by some other part of the system. It is generated by finding the minimum and maximum values of X, Y, and Z in your vertex list. For spheres, the radii determine the extents.

In some situations, you might want to provide extent information that is not the same as the object's actual extents. For example, if you have a time series of data where the object's extents are expanding (an explosion, for example) you may want to set the extent for the whole series to be large enough to avoid clipping the scene as the extents required increase in dimension. This way you can normalize the object (center it in the view) and not lose portions of it as the time series progresses.

You should not set the extent so that it is smaller than the geometry of your object because the system relies on the extent to display all of the geometry.

Flags

Many routines have an *alloc* flag as a parameter. If this flag is set to `GEOM_COPY_DATA`, the geometry routine allocates its own space and copies the data of the object. If this flag is set to `GEOM_DONT_COPY_DATA`, the geometry routine does not copy the data. In this case, you must allocate space for the data using `malloc()`. It is easier to have the routine allocate the space.

User-supplied primitive data

You may associate a single integer with each line or polygon primitive within an object. This integer should contain no more than four bytes of significant information. Unlike other data values that are associated with the object, this data is not interpreted by the Geometry Viewer. It is returned to you for pick correlation purposes using upstream data.

For polyhedrons, there can be a single value for each polygon in the object. For meshes, there can be (M-1)*(N-1) values (this is the number of quadrilaterals in the mesh). For triangle strips, there can be (N-2) values. For disjoint lines, there can be (N/2). For polylines, there can be (N-1).

User-supplied vertex data

This is similar to user–supplied primitive data but is associated with a vertex instead of a primitive. During any particular pick, you pick a primitive, but ConvexAVS returns the information corresponding to the closest selected vertex as well. For a particular object, there can be a single piece of user-supplied data for each vertex in the object.

Vertex and primitive data are not applicable to all object types. Table 20 shows what object types can use each type of data.

Table 20
Geometry types compatibility

Type	Primitive	Vertex
GEOM_LABEL	NO	NO
GEOM_MESH	YES	YES
GEOM_POLYHEDRON	YES	YES
GEOM_POLYTRI	YES	YES
GEOM_SPHERE	NO	YES

FORTRAN binding

All of the geometry 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 information in Table 21 shows how to convert C language data declarations into FORTRAN declarations.

Table 21
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`

Maintaining transformation matrices

This section describes how the transformation matrices of objects are maintained.

Object transformations

Each object has a 4 by 4 homogenous transformation matrix and a 3 by 1 position vector that describes the current position and orientation of the object. The 4 by 4 matrix is treated in C as a 4 by 4 array of floats. For example, `float matrix[4][4]`, and in FORTRAN as `REAL*4 matrix(4,4)`.

In C, the translation component of this matrix is:

```
X = matrix[3][0], Y = matrix[3][1], Z = matrix[3][2]
```

In FORTRAN it is:

```
X = matrix(1,4), Y = matrix(2,4), Z = matrix(3,4)
```

At the point at which the matrix is applied to the object, the 3 by 1 position vector is added onto the matrix. It is kept separate so that a fixed object center is defined. The first transformation that is applied to the 4 by 4 matrix is a translate of the center of the object to the origin. After all of the rotations and scales, a translation from the object center back to the origin is applied. Then the translation to the position of the object is tacked on to the end.

The vertex transformation can be depicted as:

```
Verts*[Trans(-center)]*[Rotates+Scales]*[Trans(center)]+Position
```

In general, you do not have to be aware of this level of detail. If you use `GEOMedit_set_matrix` for an object, though, you are replacing the transformations that define the object center and therefore negate any center that you might have set.

Each child object is transformed by the complete matrix of its parent object. This occurs for each object all the way up to the top-level object. After its transformation has been applied, you are now in the *world coordinate system*. The world coordinate system is where light sources are defined.

Light transformations

Light sources have a simpler transformation scheme than objects. They currently do not have a center of rotation, only a 4 by 4 transformation matrix and a position. The resulting position of light sources is determined by applying the resulting complete transformation of the light by the default location of the light source.

This is handled differently for each type of light source:

- Ambient lights are not transformed at all.
- Directional lights are transformed as a direction vector with a homogenous coordinate of zero. This means that translations are ignored. The default direction vector is pointing into the scene (0, 0, -1).

Camera transformations

The camera position is defined by a single 4-by-4 matrix and a 3-by-1 position vector that defines the camera's orientation and position and a separate 4-by-4 matrix that defines the projection for the camera matrix.

The resulting viewing pipeline is depicted as:

(Verts * [Obj Matrices] * [View Orientation] + Position) * [Projection]

The projection is kept in a separate matrix because it prevents any transformations after the perspective transformation is applied.

In the Geometry Viewer, when you use the **Perspective** and **Front/Back Clipping** buttons, you are modifying the default projection matrix. When you scale or rotate the camera, you are post-concatenating onto the View Orientation matrix. Using `GEOMedit_set_matrix` with a camera sets the View Orientation matrix. Using `GEOMedit_projection` sets the projection matrix.

Geometry routines 11

This chapter lists the geometry routines:

- Grouped by topic
- Listed and defined alphabetically

A C language application that uses geometry routines must use the /usr/avs/include/geom.h include file.

A FORTRAN language application that uses geometry routines must use the /usr/avs/include/geom.inc include file.

Any application that uses geometry routines must be linked to the following libraries in this order:

1. /usr/avs/lib/libgeom.a
2. /usr/avs/lib/libutil.a

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

- `GEOMadd_disjoint_polygon`
- `GEOMadd_polygon`
- `GEOMadd_polygons`
- `GEOMcreate_polyh`
- `GEOMcreate_polyh_with_data`

Spheres—GEOM_SPHERE

- `GEOMadd_radii`
- `GEOMcreate_sphere`

Text labels—GEOM_LABEL

- GEOMadd_label
- GEOMcreate_label
- GEOMcreate_label_flags

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 object creation function.

Color—all GEOMobj except GEOM_POLYTRI

- GEOMadd_float_colors
- GEOMadd_int_colors

Normals—GEOM_MESH and GEOM_POLYHEDRON

- GEOMadd_normals
- GEOMflip_normals
- GEOMgen_normals
- GEOMnormalize_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_computed_extent
- GEOMset_extent
- GEOMunion_extents

Object property routines

A feature of the geometry library allows arbitrary value lists to be associated with each object. These value lists can then be interpreted by packages reading in the objects. The object format supports values that are arbitrarily long. Currently, only integer values are supported by the subroutine interface.

- GEOMadd_int_value
- GEOMquery_int_value
- GEOMset_color

Edit list manipulation routines

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

- GEOMdestroy_edit_list
- GEOMinit_edit_list

Object insertion—all GEOMobj

- GEOMedit_geometry

Object transformation—all GEOMobj, light, and camera

- GEOMedit_concat_matrix
- GEOMedit_set_matrix

Object hierarchy—all GEOMobj

- GEOMedit_parent
- GEOMedit_picked

Object properties—all GEOMobj

- GEOMedit_color
- GEOMedit_properties
- 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_pickable
- GEOMset_object_group

Listed alphabetically

The following section lists geometry routines alphabetically.

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_disjoint_prim_data

```
GEOMadd_disjoint_prim_data(obj, pdata, n, alloc)
GEOMobj         *obj;
float           *pdata;
int             n;
int             alloc;
```

This routine should only be used for objects of type `GEOM_POLYTRI` that have disjoint line primitives in them. It allows you to associate primitive data with the disjoint lines in a polytriangle type object. The number *n* should be the number of disjoint lines in the object. This is one-half the number of vertices that the object contains.

GEOMadd_disjoint_vertex_data

```
GEOMadd_disjoint_vertex_data(obj, vdata, n, alloc)
GEOMobj         *obj;
int             *vdata;
int             n;
int             alloc;
```

This routine should only be used for objects of type `GEOM_POLYTRI` that have disjoint line primitives in them. It allows you to associate vertex data with the disjoint lines in a polytriangle type object. The number *n* should be the number of vertices in the disjoint line object. This is twice the number of disjoint lines.

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 0-255. 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_int_value

```
GEOMadd_int_value(obj, type, value)
GEOMobj         *obj;
int             type;
int             value;
```

Adds an integer property to an object. Currently, the only supported property of an object is the color (GEOM_COLOR).

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 used.

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 used.

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

```
GECMadd_polyline(obj, verts, colors, n, alloc)
GECMobj          *obj;
flcat            verts[n][3];
flcat            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_polyline_prim_data

```
GEOMadd_polyline_prim_data(obj, pdata, i, n, alloc)
GEOMobj          *obj;
int              *pdata;
int              i;
int              n;
int              alloc;
```

This routine should be used only for objects of type `GEOM_POLYTRI` that contain polyline primitives. It allows you to associate vertex data with the polylines in a polytriangle type object. The number *n* is the number of vertices in the polyline object. This is the number of disjoint lines minus one. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, and so on.

GEOMadd_polyline_vertex_data

```
GEOMadd_polyline_vertex_data(obj, vdata, i, n, alloc)
GEOMobj         *obj;
int             *vdata;
int             i;
int             n;
int             alloc;
```

This routine should be used only for objects of type `GEOM_POLYTRI` that contain polyline primitives. It allows you to associate vertex data with the polylines in a polytriangle type object. The number *n* is the number of vertices in the polyline object. There are *n* vertices, but the number of lines is *n*-1. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, and so on.

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_polytriangle_prim_data

```
GEOMadd_polytriangle_prim_data(obj, pdata, i, n, alloc)
GEOMobj         *obj;
int             *pdata;
int             i;
int             n;
int             alloc;
```

This routine should be used only for objects of type `GEOM_POLYTRI` that contain polytriangle strip primitives. It allows you to associate vertex data with the polytriangle strips in a polytriangle type object. The number *n* is the number of triangles in the polytriangle object. This is the number of vertices minus two. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, and so on.

GEOMadd_polytriangle_vertex_data

```
GEOMadd_polytriangle_vertex_data(obj, vdata, i, n, alloc)
GEOMobj         *obj;
int             *vdata;
int             i;
int             n;
int             alloc;
```

This routine should be used only for objects of type `GEOM_POLYTRI` that contain polytriangle strip primitives. It allows you to associate vertex data with the polytriangle strips in a polytriangle type object. The number *n* is the number of triangles in the polytriangle object. This is the number of vertices minus two. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, and so on.

GEOMadd_prim_data

```
GEOMadd_prim_data(obj, pdata, n, alloc)
GEOMobj         *obj;
int             *pdata;
int             n;
int             alloc;
```

Associate the array of primitive data with the object specified. This routine can be used only for objects of type GEOM_POLYHEDRON and GEOM_MESH. For objects of type GEOM_MESH, the value *n* should be equal to: (*M*-1)*(*N*-1), where *M* and *N* are the dimensions of the mesh.

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_vertex_data

```
GEOMadd_vertex_data(obj, vdata, n, alloc)
GEOMobj         *obj;
int             *vdata;
int             n;
int             alloc;
```

Associates the array of vertex data with the object specified. This routine can be used only with objects of type GEOM_MESH, GEOM_POLYHEDRON, and GEOM_SPHERE. The number of data elements *n* should be equal to the number of vertices in the object.

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 used.

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

```
GECMobj          *
GECMcreate_normal_object(obj, scale)
GECMobj          *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 used.

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 used.

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

```
GECMcvt_polyh_to_polytri(obj, flags)
GECMobj          *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_center

```
GEOMedit_center(list, name, center)
GEOMedit_list   list;
char            *name;
float           center[3];
```

Sets the center of rotation of the object specified. This does not currently work for cameras or lights. The center of rotation is defined before the object's transformation matrix is applied. It should, therefore, be defined in the same coordinate system as the vertices of the object.

GEOMedit_color

```
GECMedit_color(list, name, color)
GECMedit_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_color`. 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.

See "Maintaining transformation matrices," on page 330 for more information.

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 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_position

```
GEOMedit_position(list, name, position)
GEOMedit_list   list;
char            *name;
float           position[3];
```

Sets the position vector for the object specified. Positions are always applied after the matrix you set with `GEOMedit_set_matrix`.

See "Maintaining transformation matrices," on page 330 for more information.

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, and so on). 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_projection

```
GEOMedit_projection(list, name, projection)
GEOMedit_list  list;
char           *name;
float          projection[4][4];
```

Sets the projection matrix for a particular camera. The argument *name* should be of the form "camera*n*" where *n* is 1, 2, and so on.

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 be 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_selection_mode

```
GEOMedit_selection_mode(list, name, mode, flags)
GEOMedit_list  list;
char           *name;
char           *mode;
int            flags;
```

This routine sets the selection mode of the object given. It can be used for two purposes:

- To make an object unpickable.
- To allow an upstream module to receive pick information when an object is selected.

Values for the *mode* argument are:

`notify` Notify the calling module when the object *name* is selected. If the object is subsequently selected and the module has an input port connected to the **render geometry** module's upstream geometry output port, the module will be executed with some information pertaining to the specifics of the selection.

`normal` Restore the selection mode of the object to normal. No module will receive selection information from the object.

`ignore` Do not allow the object specified to be picked. Any attempt to pick the object will result in a pick of the parent object instead.

The flags argument is only relevant when the `notify` mode is set. It should contain one or more of the following flags: `BUTTON_DOWN`, `BUTTON_UP`, `BUTTON_MOVING`. The definition for these flags is contained in the udata.h include file in the /usr/avs/include directory.

The *flags* field indicates for what button states information should be redirected to the module.

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_transform_mode

```
GEOMedit_transform_mode(list,name,redirect,flags)
GEOMedit_list  list;
char           *name;
char           *redirect;
int            flags;
```

This routine sets the transformation mode of the object with the given name. It can be used for two purposes:

- To prevent accidentally transforming an object that should be defined in the coordinate system of its parent.
- To allow an upstream module to receive notification when the object is transformed.

Values for the *name* argument are:

`normal` Restore the transform mode to the default or normal mode. In this case, the *flags* argument is ignored.

`parent` Any transformations that are applied to this object are redirected to the parent object. This mode can be used to prevent transforming this object relative to its parent object. In this case, the *flags* argument is ignored. If an object other than the parent is selected, then the bounding box will not move until you release the mouse button.

`notify` Notify the calling module when the object *name* is transformed. If the object is subsequently transformed and the module has an input port connected to the **render geometry** module's upstream transformation output port, the module will be executed with a variety of information including the transformation matrix of the object.

`redirect` This mode is similar to the `notify` mode above. There are only two differences:

1. The transformation matrix that is accumulated for the object and passed to the module is not used in transforming the geometry of the object.
2. Because the Geometry Viewer is not going to directly transform the object when the transformation matrix changes, it does not refresh the display. This mode is useful when the module always regenerates the geometry of the object each time that the transformation matrix changes. Because the identity matrix will be used when rendering the object, the module will have to transform any vertices generated.

The *flags* argument is only relevant when the `notify` or `redirect` transform mode is set. It should contain one or more of the following flags: `BUTTON_DOWN`, `BUTTON_UP`, `BUTTON_MOVING`. The definition for these flags is contained in the udata.h include file.

The *flags* field indicates what button states information should be redirected to the module. Transformations that are not caused by mouse movement use the state `BUTTON_UP`.

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 module to delete an object once it is in the Geometry Viewer.

list	An initialized edit list.	
**name*	The name an object is given in the call to `GEOMedit_geometry`.	
visibility	-1	Delete the object.
	0	Turn object visibility OFF.
	1	Turn object visibility ON.

GEOMedit_window

```
GEOMedit_window(list, name, window)
GEOMedit_list  list;
char           *name;
float          window[6];
```

This routine allows you to specify the *window* of interest for an object. It allows a module to cause the Geometry Viewer to automatically Normalize and Center the top-level object when the window changes. By default, the Geometry Viewer will only display geometry that is in the range -5 to 5 in X and Y. You must either scale and translate your data to be in this range or you must change the transformation matrix of either the object, a parent of the object, or the camera so that your geometry will become viewable.

The `GEOMedit_window` routine implements a mechanism whereby the Geometry Viewer will handle this scale and translate automatically. It does so in a way that allows multiple geometry producing modules to cooperatively decide on a global scale/translate that displays all geometries that are produced. It also keeps that data in the natural coordinate system in which the data is defined. This allows the Geometry Viewer to display data sets that are defined in the same physical coordinate system simultaneously without distorting their interrelationships.

The window that is specified by the module contains an array of six floating-point numbers in the order: minimum X, maximum X, minimum Y, maximum Y, minimum Z, maximum Z. These values define a bounding box relative to the top level object (that is, not transformed by the object's own transformation). This box should contain the range of the coordinate system of interest. For example, if your vertices lie within 0 to 100 in X, Y and Z, your window should be: 0, 100, 0, 100, 0, 100. The window should include any transformations that are going to be applied to the object (not including the top-level object). For example, if your vertices are defined as above, but you are going to scale your object down by a factor of 2, the window should be set to: 0, 50, 0, 50, 0, 50.

The window is associated with a particular object. While that object still exists in the scene and has a window defined for it, it will continue to be used to determine the scale and position of the top-level object.

The Geometry Viewer maintains a global window that includes the extent of all windows currently defined in the scene. Whenever a `GEOMedit_window` request is received, the Geometry Viewer recomputes the new window by computing a box that surrounds all of the windows currently defined. If the new global window is different from the old global window, the scene is scaled and translated so that the new global window will lie inside of the viewable region of the screen. The rotation/scale center of the top-level object is also set to be the center point of the global window.

If the window does not change between subsequent edit window requests, the top-level object's transformation is unchanged.

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 used.

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.

GEOMquery_int_value

```
int
GEOMquery_int_value(obj, type, value)
GEOMobj          *obj;
int              type;
unsigned int     *value;
```

**obj* A pointer to an existing object.

An integer value can be queried with this routine. The type is an integer value. The only supported property type is `GEOM_COLOR`. Its value can be queried with:

```
GEOMquery_int_value(obj, GEOM_COLOR, &color);
```

This routine returns 0 if no color was associated with the 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_color

```
GEOMset_color(obj, color)
GEOMobj          *obj;
unsigned long    color;
```

Sets the *color* property of an object by calling the `GEOMadd_int_value` routine.

**obj*	A pointer to an existing object.
color	An RGB color.

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*		An open file pointer.
flags	0	This parameter is not currently used.

Unstructured cell data

12

What is UCD?

In order to allow the visualization of results generated in programs that use finite element analysis, ConvexAVS supports a data type called unstructured cell data (UCD). Although UCD is primarily intended for use with finite element modeling (FEM) data, it is also perfectly capable of dealing with data that was created in other ways. The decision of which data type to use (field versus UCD structure) will depend primarily upon how the data is structured. As a rule of thumb, if the data has hexagonal connectivity in 3-space or quadrilateral connectivity in 2-space, an irregular field will suffice. If not, the data will have to be put into UCD format. If fields can be used, they should be, because the greater flexibility of UCD structures is a trade-off with performance and module functionality.

In a number of scientific disciplines, for example, computational fluid dynamics (CFD) or mechanical computer–aided design (MCAE), the most convenient way to find solutions is by the use of finite element analysis. In finite element analysis, the problem domain (usually a subset of 2-D or 3-D space) is partitioned into numerous cells or elements. The cells are geometric shapes defined by sets of edges and vertices. For example, in a triangle (2-D space) there are three vertices and three edges, and in a tetrahedron (3-D space) there are four vertices and six edges. Each of the cells contains a set of points called the *node points* or *nodes*. These nodes are usually the vertices of the cell in question, but they may also be midpoints of the edges of the cell. The *connectivity* of the resulting set of cells and nodes is often more complex than the field data type can represent. Here, connectivity means the relationship between a given cell (or node) and its neighbors.

In most physical problems where the mathematical model is a scalar or vector field, the actual data is continuous and defined on a continuous spatial domain but must be sampled to obtain a discretization that the computer can work with. The actual spatial domain can be thought of as the geometric points lying within and on the boundary of the cells. If the data varies over the cells, it can be sampled at the nodes and be represented as node data. If the data is constant over a given cell, it could be represented as cell data.

Each type of data can be represented in a UCD structure by several fields defined on the cells (or nodes). These fields can be vector, scalar, or a mix. In this way, a UCD structure can represent several related fields defined on the same domain. For example, in a CFD data set, it is common to define two scalar fields and a vector field over the fluid in question. All three of these fields (five floating-point numbers per position in the space the fluid occupies) could be represented in a single UCD structure. Table 23, "Domain and range of fields and UCD structures," on page 391 shows the domain and range of fields and UCD structures.

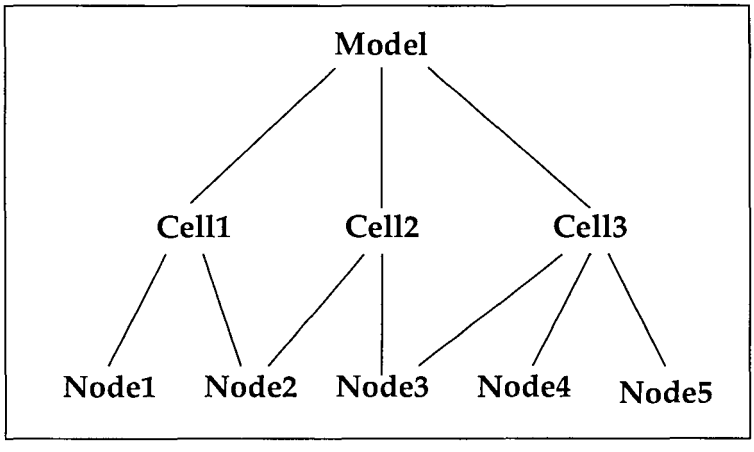
UCD hierarchy

UCD sets employ a hierarchical data structure. There are three levels in this hierarchy:

1. Model level
2. Cell level
3. Node level

Figure 131 shows this hierarchy.

Figure 131
UCD hierarchy



Data that is global to the entire problem can be stored at the model level. Data that is characteristic of the cells would go in the cell level. Data associated with the nodes would go in the node level. Each cell is defined by its type and the nodes that comprise it. You may have almost any mix of cell types in the same UCD structure.

Currently, eight cell types (or shapes) are supported in ConvexAVS. The cell types can be categorized by their dimension as shown in Table 22.

The number of nodes associated with each cell type depends upon the presence or absence of mid-edge nodes as shown in Table 22.

Table 22
Cell types and number of nodes

Cell type	Number of Dimensions	Number of nodes	
		No mid-edge nodes	With mid-edge nodes
Point	0	1	1
Line	1	2	3
Triangle	2	3	6
Quadrilateral		4	8
Tetrahedron	3	4	10
Pyramid		5	13
Prism		6	15
Hexahedron		8	20

Figure 132 shows the 0-D and 1-D UCD cell types. Figure 133 shows the 2-D UCD cell types. Figure 134 shows the 3-D UCD cell types. Cell connectivity in the UCD files and UCD structure should follow the numbering conventions shown here.

Figure 132
0-D and 1-D UCD cell types and node numbering

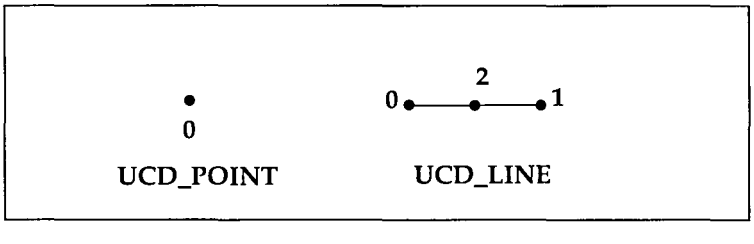


Figure 133
2-D UCD cell types and node numbering

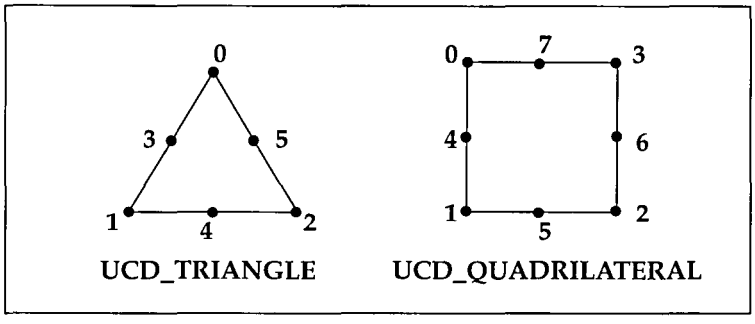
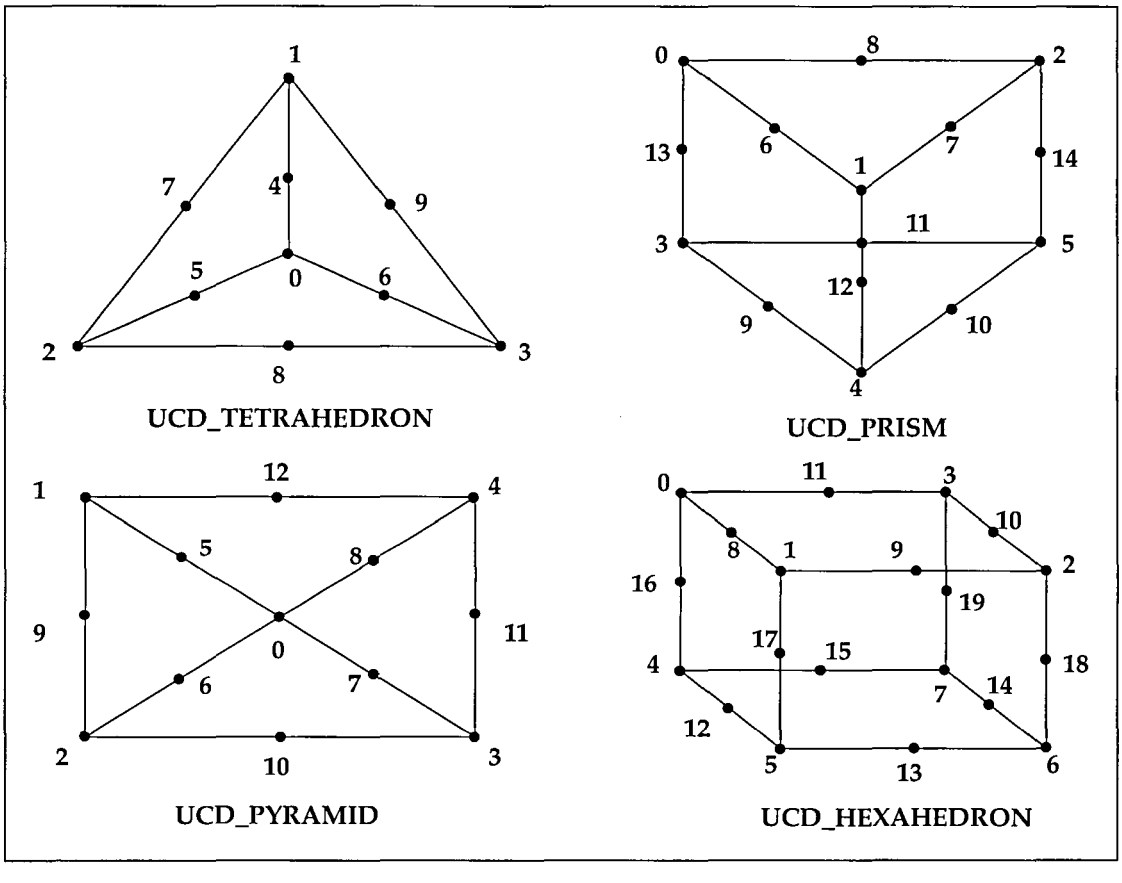


Figure 134
3-D UCD cell types and node numbering



In principle, data can be stored at each of the three levels in the UCD structure. The UCD library supports data access at all three levels, and thus you can write your own modules that access data on any level. Currently, none of the modules supplied with ConvexAVS operate on any data that is not present at the node level. In subsequent releases of ConvexAVS, this limitation will be lifted, and the operating level of the supported modules will be broadened to include cell data and model data.

Comparing UCD structures with fields

In order to understand how data is organized on each of the levels of the UCD structure, it is helpful to compare and contrast UCD structures with fields. Both fields and UCD structures are discrete computer representations of the mathematical concept of a mapping. That is, both represent one-to-one relationships between points in space and data sampled at those points.

In the discussion that follows, the set of points at which the data is sampled is called the *grid* and the sampled data at those points is called the *data*. The grid can have a variety of geometries. The dimensionality of the intrinsic geometry of the grid can be represented by the dimension of the computational space (in fields) or by the dimensionality of the cells used to partition the grid (in UCD structures).

An example

As an example of the relationship between a UCD structure and a field, consider a grid of points on the surface of the earth. The grid is intrinsically 2-D, but the physical space is 3-D. The fact that two numbers (longitude and latitude) uniquely determine a point on the grid tells us that the intrinsic geometry of the grid is 2-D. To represent these points uniquely in the surrounding Cartesian space, however, would require three numbers (you must specify the z-coordinate to distinguish the point at 30 West longitude 20 North latitude from the point at 30 West longitude 20 South latitude).

In a field, the data set could be represented by a 2-D, 3-space, irregular field. Associated with each 2-D (i,j) index pair in computational space would be the x-,y-, and z-coordinates of the point and the data for that point (for example, temperature).

In a UCD structure, the grid would be represented by nodes in which all three spatial coordinates of the node positions are specified, but the cells would all be 2-D (for example, triangles or quadrilaterals). The data would be then stored on a node-by-node basis.

It is important to remember that the concept of the dimensionality of the computational space of a field is analogous to the dimensionality of the basic cell type in a UCD structure. In fields, the dimensionality of the intrinsic geometry of the grid upon which the data is sampled is the dimensionality of the computational space. In UCD structures, the dimensionality of the intrinsic geometry of the grid is the dimensionality of the basic cell elements.

Connectivity

In UCD structures, the node positions are limited to at most three dimensions in physical space. This is not usually a problem because none of the mappers supplied with ConvexAVS support geometries of greater than three dimensions. In fields, the grid can be of any dimensionality in physical or computational space.

In order to use a field to represent your data, you must be able to access the data via an N-dimensional array. That is, the connectivity of the grid points (or nodes) must be simple enough so that each grid point in N-dimensional computational space is 2N-fold connected (that is, has 2N nearest neighbors). For example, in 3-D computational space, each grid point must have six nearest neighbors:

- point[i+1,j,k]
- point[i-1,j,k]
- point[i,j+1,k]
- point[i,j-1,k]
- point[i,j,k+1]
- point[i,j,k-1]

In general, fields are supported by more ConvexAVS visualization modules and these modules work faster than their equivalent UCD modules. If your data is purely 3-D hexahedral (6-fold connected) or 2-D quadrilateral (4-fold connected), then you should consider converting it to field format. If, however, the grid on which your data is defined is more complex, then you must use a UCD structure. An example of such a grid would be a grid defined by a tetrahedral partition of 3-D space. These sorts of partitions occur frequently in FEM problems where an irregularly shaped region must be modeled.

Table 23 shows the domain and range of fields and UCD structures.

Table 23
Domain and range of fields and UCD structures

	Field	UCD structure
Domain	Field –> Points	UCD –> x UCD –> y UCD –> z
Range	Field –> Data	UCD –> Cell data UCD –> Node data

UCD formats

UCD information can exist in two types of representation: internal and external. Internal representation is the form in which UCD structures exists when they are *in core* at module run time. The only internal representation of UCD information is `UCD_structure`. External representation of a UCD structure exists when it is written to a file. ConvexAVS supports two external UCD formats: binary and ASCII.

There are several ways to get your UCD information into ConvexAVS format:

- The most robust and flexible way is to write a module that outputs a UCD structure. This allows you complete and explicit control over the data in the UCD structure. It is also usually the most complicated and programming-intensive way because it involves using the UCD access routines as well as the programmer's interface to ConvexAVS.

- The least complicated way is to create an ASCII UCD file. The format of the ASCII UCD representation is simple and easy to create. Often, minor modifications to the I/O portions of code that already exists in your program might be all that is necessary. If you currently write your data out as an ASCII file, it is possible that a short `perl` or `awk` script could be written to filter the existing file into ConvexAVS format.
- A third approach is to create a binary UCD file. This can be complicated and harder to debug than an ASCII file, but the results will read faster than an ASCII file. If your data sets are large enough, it might be worth the extra effort. On the other hand, the binary format might change in future versions of ConvexAVS and not be portable or compatible.

Internal format

Top-level view

When a UCD structure exists in a module at run time, it exists as the C language `struct UCD_structure` as defined in the ucd_defs.h include file. You can access any of the fields of this structure from module code. It is strongly recommended, however, that the data be accessed only through the UCD routines documented in Chapter 13, "UCD structure routines." This is the only way to stay compatible with future versions of the UCD software. FORTRAN language users cannot directly access the structures because the FORTRAN language does not understand C structures.

The UCD structure contains a large number of items, most of which will not be described in this document. Some of the data in the structure is used internally by the software and is subject to change. This section contains an overview of items in the structure. More detailed information is provided in Chapter 13 where the UCD routines are documented.

Data in UCD structures can be divided into four parts:

- Model information
- Cell information
- Node information
- System/bookkeeping information

At each of the three hierarchical levels (model, cell, or node), it is possible to store a collection of data. In this respect, the model level is a special case because there is only one model. Typically, there are hundreds and perhaps thousands of cells and nodes, each of which might have associated data. In order to avoid confusing language, let's call this collection of data an *aggregate*. The aggregate can contain multiple data components. Furthermore, the aggregate can be mixed data in the sense that it can contain both *n*-vectors and scalars. To keep all this organized, the UCD structure contains not just a veclen field (length of the aggregate) but also a node (or cell) component mix array. This array contains the veclen of each of the components in the aggregate. As an example of how this works, consider some UCD information that represents a PLOT3D data set. Each point (that is, node) in the mesh has three different quantities associated with it:

- Density and stagnation energy per unit volume (scalars)
- Momentum per unit volume (vector)

The node mix vector might be {1,3,1}, and the labels would be {"density","momentum density","stagnation energy"}. The units might be {"g/cm^3","g/cm^2*s","erg/cm^3"}. The label and unit arrays have three entries even though the aggregate veclen is five. That is because there are only three separate components.

Creating a UCD structure

There are two ways to create a UCD structure in internal format:

- From scratch
- Derived from an input UCD structure

To illustrate these processes, we have included two modules in the /usr/avs/examples directory. They are ucd_grid.c and ucd_function.c.

ucd_grid.c

This module creates a UCD structure from scratch that represents a rectangular slice of space under the spherical coordinate transformation. There are dial widgets that help you:

- Select the boundaries of the space in (rho,theta,phi) space.
- Select the number of node samples to take along each axis in (rho,theta,phi) space.
- Move the origin of the spherical coordinate system in (x,y,z) space.

You can use this module to generate UCD information that can be written with the **write ucd** module. This will generate sample UCD ASCII and binary files.

This module has a system for allowing you to turn on debugging information by using an environment variable. This may be helpful to use while working with the module to try to understand how UCD structures are built. To activate this feature, the environment variable `UCD_GRID_DEBUG` must be set to the name of the file where you want debugging information to go. The strings "`stdout`" and "`stderr`" will cause the debugging information to be sent to standard out or standard error, respectively.

ucd_function.c

This module takes the UCD structure produced by ucd_grid.c and evaluates a function on the node positions. Currently, two different functions are implemented:

- The linear scalar function of the coordinates:

$$s = ax + by + cz$$

- The quadratic function of the coordinates:

$$s = ax^2 + by^2 + cz^2$$

This module is meant to provide simple examples for experimenting with the visualization of scalar fields and to illustrate how to create a UCD structure that is derived from another UCD structure.

As you look over the modules provided in the example directory, you should notice the following things:

- `UCDstructure_alloc` must be given a lot of information if it is to create a valid UCD structure. This information includes the lengths of internal tables. Refer to the section "Instantiating UCD structures," on page 395.
- `UCDstructure_alloc` can be passed the values of two flags. You should set the *util* flag to zero. The other flag, *ucd_flag* (also called *name_flag*), has several settings. Refer to the section on instantiating UCD structures.
- The *components* array must be specified along with the aggregate vector length if a correct UCD structure is to be created. This is done by a call to `UCDstructure_set_node_components` in the example. Refer to the section on specifying vector length at run time.
- The *min* and *max* values must be provided. Refer to the use of the `UCDstructure_set_node_minmax` routine.
- The *labels* and the *units* of the UCD structure must be provided for the level being used (in this case, nodes). This is done by calls to `UCDstructure_set_node_labels` and `UCDstructure_set_node_units`.
- The order in which the nodes are specified in a cell is extremely important. Incorrect node list order can result in downstream modules failing or wrong normals being generated in geometries. Wrong normals usually result in *black geometries* that become *colorful* when bi-directional lights are used. Refer to the earlier figures that give node order before you attempt to set a node list in a cell.

Instantiating UCD structures

When you call `UCDstructure_alloc` to instantiate a UCD structure, you have to supply a variety of information that is used to calculate the amount of memory to be allocated. Three of the arguments to `UCDstructure_alloc` should be examined closely.

The first two are *cell_tsize* and *node_csize*. These are the sizes of two internal tables that contain connectivity information. In a model with homogenous cell types, they should both be set to:

*number of cells * number of nodes per cell*

In mixed cell geometries, you will need one entry per node reference in each cell. Thus, a geometry with no mid-edge nodes consisting of a hexahedron and three tetrahedrons would require 8 + 3 * 4 = 20 in these two slots. For geometries with mid-edge nodes, refer to Table 22 for the number of nodes in each cell.

The third argument is the *name_flags* argument. This tells the UCD software whether to create tables of storage for cell IDs, node IDs, mid-edge nodes, material IDs, and so on.

Specifying vector length at run time

Because UCD structures can contain multiple and heterogenous data, you must specify both aggregate vector length and component vector length. The aggregate vector length is specified by the `UCDstructure_alloc` routine in the *data_veclen* (for model data), *cell_veclen* (for cell data), and *node_veclen* (for node data). If any particular level contains no data, the corresponding veclen must be set to 0. You must also set the component vector lengths (the sum of these must equal the aggregate vector length). Setting the individual vector lengths is done with a call to one of the following routines:

- `UCDstructure_set_cell_components` for cell data
- `UCDstructure_set_node_components` for node data

There is no `UCDstructure_set_data_components` routine.

ASCII format

An ASCII UCD file consists of the following sections:

- Comment section (optional)
- Header section
- Node position section
- Cell connectivity section
- Data sections

Comment section

The comment section may be used to document the file. It is optional. If present, the comment section must precede all other sections, and each line must begin with a pound sign (#). Comments that come after the header section will cause the **read ucd** module to malfunction, often resulting in unwanted behavior from the system.

Header section

The header section is a line containing five integer ASCII fields in the following order:

1. The number of nodes in the UCD
2. The number of cells in the UCD
3. The aggregate vector length of node data
4. The aggregate vector length of cell data
5. The aggregate vector length of model data

The aggregate vector lengths reflect the sum of the vector lengths of the individual components of the collection of data. If there were two scalars and a single 3-vector at each node, the aggregate vector length would be

```
2 * 1 + 3 = 5
```

Alternatively, if there was a single scalar value at each node, the third field would be a 1.

All fields of the header section must be present. If there is no data at one of the levels, then the aggregate vector length should be zero for that level. For example, a UCD structure containing 3,375 nodes and 2,744 cells with a single scalar data value at each node would have the following header:

```
3375 2744 1 0 0
```

Here, the vector length for the cell and model data is 0.

Node position section

The node position section consists of a line for each node of the UCD structure. In the example given above, there would be 3,375 lines in this section. Each line contains four ASCII fields. The fields are in the following order:

1. The node ID (integer)
2. The X-component of the node position (float)
3. The Y-component of the node position (float)
4. The Z-component of the node position (float)

The node ID is an integer *name* for the node. The node IDs do not need to be contiguous or even in order. The important thing to remember is that they must be unique.

Cell connectivity section

The cell connectivity section consists of a line for each cell of the UCD structure. In the example given above, there would be 2,744 lines in this section. Each line contains four fields with the last field being a variable length list. The length of the list depends on the type of cell being described and whether there are mid-edge nodes. The fields are in the following order:

1. Cell ID
2. Material ID
3. Name of the cell type
4. Node ID list

The cell ID is an integer that identifies the cell so that it can be matched with its data. Like node IDs, cell IDs do not need to be contiguous nor in order. The material ID is an integer that allows specification of the type of material that the cell consists of. Currently, this is not used by ConvexAVS. The cell type name is a string giving the type of cell being described. When you specify the type name of a cell in this file, you must use the abbreviations shown in Table 24. The node ID list is a field of variable length. Refer to Table 22 to determine the number of nodes that must be specified for each type of cell.

Table 24
Cell types and name abbreviations

Type	Abbreviation	Code
Point	pt	0
Line	line	1
Triangle	tri	2
Quadrilateral	quad	3
Tetrahedron	tet	4
Pyramid	pyr	5
Prism	prism	6
Hexahedron	hex	7

The `mid_edge_flag` variable in the UCD structure indicates which mid-edge nodes are present for a particular cell. Each bit, starting from the right, indicates the presence of one mid-edge node. For example, if the cell is UCD_TRIANGLE and has two mid-edge nodes, 3 and 5, the variable would be:

```
00...00101
```

Data sections

There are three types of data sections:

- Node data
- Cell data
- Model data

If more than one kind of data is present, the sections must be in the order shown. Currently, ConvexAVS modules do not use model data at all, and treatment of cell data is very limited. Your modules, however, might use all three levels of data.

Each of the data sections (node, cell, and model) is structured in the same way:

- Header
- Label list
- Data list

The header is a single line with the first field being number of components (`num_comp`) in the aggregate vector of data. Subsequent fields are the `num_comp` vector lengths of the individual components. The sum of the of the `num_comp` individual component vector lengths must equal the aggregate vector length given in the overall file header.

To clarify this, let's return to the example of a file that contains two scalars and a 3-vector for each node. Let's further assume that the data is organized so that the first number in the aggregate data vector is the first scalar, the next three numbers are the 3-vector, and the last number is the second scalar. The data block header would be

```
3 1 3 1
```

and the aggregate vector length would be

```
1 + 3 + 1 = 5.
```

The label list is a set of `num_comp` lines with the data label and the units label for each component. Each line consists of a pair of strings separated by a comma.

The data list is a set of lines, one for each node (or cell). In the case of model data, there is only one model so there can be only one line. Each line contains a node (cell) ID field and a collection of data. The number of fields in the vector of data will be equal to the aggregate data vector length from the overall file header. In the example above, a representative data list line would be

```
301 1.2 1.8 0.0 2.0 5.0
```

This line says that node 301 has a scalar with value 1.2, a 3-vector with values (1.8, 0.0, 2.0), and another scalar with value 5.0. Due to the way that the **read ucd** module works, if there is model-level data specified, then a single ID integer must be part of the data line. This ID has no meaning, but it must be present.

Examples

The first example shown in Figure 135 is a single hexahedron with scalar node data.

Figure 135
Single hexahedron with scalar data

```
8 1 1 0 0
   1 0.000 0.000 1.000
   2 1.000 0.000 1.000
   3 1.000 1.000 1.000
   4 0.000 1.000 1.000
   5 0.000 0.000 0.000
   6 1.000 0.000 0.000
   7 1.000 1.000 0.000
   8 0.000 1.000 0.000
   1 1 hex 1 2 3 4 5 6 7 8
   1 1
   stress, lb/in**2
   1 4999.9999
   2 18749.9999
   3 37500.0000
   4 56250.0000
   5 74999.9999
   6 93750.0001
   7 107500.0003
   8 5000.0001
```

The second example, shown in Figure 136, is also a single hexahedron but with mixed data. The aggregate vector length is five, but there are three components (two scalar and one 3-vector).

Figure 136
Single hexahedron with mixed data

```
8 1 5 0 0
   1 0.000 0.000 1.000
   2 1.000 0.000 1.000
   3 1.000 1.000 1.000
   4 0.000 1.000 1.000
   5 0.000 0.000 0.000
   6 1.000 0.000 0.000
   7 1.000 1.000 0.000
   8 0.000 1.000 0.000
   1 1 hex 1 2 3 4 5 6 7 8
   3 1 3 1
   scalar1, lb/in**2
   vector, lb/in**2
   scalar2, lb/in**2
   1 4999.9999 0.000 0.000 1.000 1.0
   2 18749.9999 1.000 0.000 1.000 2.0
   3 37500.0000 1.000 1.000 1.000 3.0
   4 56250.0000 0.000 1.000 1.000 4
   5 74999.9999 0.000 0.000 0.000 5
   6 93750.0001 1.000 0.000 0.000 6
   7 107500.0003 1.000 1.000 0.000 7
   8 5000.0001 0.000 1.000 0.000 8
```

The third example shown in Figure 137 contains a hexahedron and two pyramids. The data consists of three scalars.

Binary format

The binary format can be seen as a collection of data blocks:

- Magic number
- Header
- Cell block
- Cell nlist block
- Block of node x coords
- Block of node y coords
- Block of node z coords
- Model, cell, node data blocks

Figure 137
Several structures with scalar data

```
10 3 3 0 0
   1 0.000 0.000 1.000
   2 1.000 0.000 1.000
   3 1.000 1.000 1.000
   4 0.000 1.000 1.000
   5 0.000 0.000 0.000
   6 1.000 0.000 0.000
   7 1.000 1.000 0.000
   8 0.000 1.000 0.000
   9 0.500 0.500 1.500
   10 0.500 0.500 -0.500
   1 1 hex 1 2 3 4 5 6 7 8
   2 1 pyr 9 1 2 3 4
   3 1 pyr 10 8 7 6 5
   3 1 1 1
   scalar1, lb/in**2
   scalar2, lb/in**2
   scalar3, lb/in**2
   1 0.000 0.000 1.000
   2 1.000 0.000 1.000
   3 1.000 1.000 1.000
   4 0.000 1.000 1.000
   5 0.000 0.000 0.000
   6 1.000 0.000 0.000
   7 1.000 1.000 0.000
   8 0.000 1.000 0.000
   9 0.500 0.500 1.500
   10 0.500 0.500 -0.500
```

Magic number

The magic number is the first byte of the file and tells the **read ucd** module what kind of file it is dealing with. There are three meaningful states for this magic number:

- A value of 0x7, denoting a revision level 3.5 UCD binary file.
- A value of 0x5, denoting a revision level 3.0 UCD binary file.
- Any other value denotes that the file is an ASCII UCD file.

Note

The magic number scheme may not always work for every data set. Exercise care when creating UCD files, or the read ucd module will almost surely behave unpredictably.

Header

The header block contains information similar to the header of an ASCII UCD file. It is stored in the following structure:

```
typedef
   struct Ucd3_5Hdr {
      int num_nodes;
      int num_cells;
      int num_node_data;
      int num_cell_data;
      int num_model_data;
      int num_nlist_nodes;
   }Ucd3_5Hdr;
```

The fields in the header block are:

`num_nodes`	Number of nodes in the UCD structure.
`num_cells`	Number of cells in the UCD structure.
`num_node_data`	Aggregate vector length of the node data.
`num_cell_data`	Aggregate vector length of the cell data.
`num_model_data`	Aggregate vector length of the model data.
`num_nlist_nodes`	Number of entries in the cell node list array.

Cell block

The cell block contains the cell descriptions for each cell in the model. The description consists of the cell ID, the material ID, the number of nodes in the cell, and the cell type. The cell ID and material ID have the same meaning as in the ASCII format. The cell type is an integer from 0 to 7 as shown in Table 24. The cell block is implemented as an array of the following structure:

```
typedef struct UcdCell {
   int id;
   int mat;
   int n;
   int cell_type;
}UcdCell;
```

Another way to look at it would be

```
UcdCell cell[num_cells];
```

Cell nlists block

Each cell has to have a node list (nlist) that defines it. The nlists block consists of an array of node indices allocated by order of cell. For example, if there are five cells in a file: a triangle, two quadrilaterals, and two more triangles (in that order). The nlists block will be set up so that the first three entries belong to cell #1 (the triangle), the next eight entries belong to cells #2 and #3 (the two quadrilaterals), and the last six entries belong to cells #4 and #5 (the last two triangles). The indices are one-based, not zero-based, so when a binary file is read, the software must decrement each raw index to get a C-style index.

Coordinate blocks

The coordinate blocks are three floating-point arrays. Their order in the file is described by the following C language declarations:

```
float x[num_nodes];
float y[num_nodes];
float z[num_nodes];
```

Each node has its own x-, y-, and z-coordinate regardless of the dimensionality of the model in question.

Data blocks

As previously mentioned, data can be stored at each of the levels of the UCD structure. The header word corresponding to each level tells if a data block is present for that level:

Level	Header word
Node	`hdr.num_node_data`
Cell	`hdr.num_cell_data`
Model	`hdr.num_model_data`

A zero in these fields means that the corresponding data is not present. In other words, the data has a zero vector length.

There are a number of fields present in a data block besides the data itself. They are:

- Labels (in a 1024-byte block)
- Units (in a 1024-byte block)
- Number of components
- Component list
- Minimum and maximum values for each data *column*
- Data
- Active list

The labels are concatenated together into a fixed length block (padded to 1024 bytes). A period (.) is inserted between labels to allow them to be separated later. The same method is used to store the units. There is a label for each component (not for each aggregate vector component). In the example used above where we had a UCD structure with three components (a scalar, a 3-vector, and another scalar for a total of five pieces of data for each node), there would be three labels and three units strings, not five. The component list reflects the fact that a UCD structure can have scalar and vector data in it. In our example, the number of components would be three but the data vector length would be five. Therefore, there would be five minimum and five maximum values, and the active list would have five entries.

The active list is not currently used by ConvexAVS, but might be used by your software.

Structure of the binary file at a glance

The easiest way to refer to the structure of a binary UCD file is as a C language structure. The construct below would never compile, but does allow you to see the layout at a glance:

```
#define UCD_LABEL_LEN 1024

typedef struct UcdCell {
   int id;
   int mat;
   int n;
   int cell_type;
}UcdCell;
```

```
typedef
   struct Ucd3_5Hdr {
      int num_nodes;
      int num_cells;
      int num_node_data;
      int num_cell_data;
      int num_model_data;
      int num_nlist_nodes;
   }Ucd3_5Hdr;

typedef
   struct UCD_Binary_File {
      char magic_number;
      Ucd3_5Hdr hdr;
      UcdCell cells[hdr.num_cells];
      int cell_nlists[hdr.num_nlist_nodes];
      float x[hdr.num_nodes];
      float y[hdr.num_nodes];
      float z[hdr.num_nodes];

      /* node data block (if present) */
      char node_data_labels[UCD_LABEL_LEN];
      char node_data_units[UCD_LABEL_LEN];
      int num_node_comp;
      int node_comp_list[num_node_comp];
      float min_node_data[hdr.num_node_data];
      float max_node_data[hdr.num_node_data];
      float node_data[hdr.num_node_data *
                                    hdr.num_nodes];
      int node_active_list[hdr.num_node_data];

      /* cell data block (if present) */
      char cell_data_labels[UCD_LABEL_LEN];
      char cell_data_units[UCD_LABEL_LEN];
      int num_cell_comp;
      int cell_comp_list[num_cell_comp];
      float min_cell_data[hdr.num_cell_data];
      float max_cell_data[hdr.num_cell_data];
      float cell_data[hdr.num_cell_data *
                                    hdr.num_cells];
      int cell_active_list[hdr.num_cell_data];

      /* model data block (if present) */
      char model_data_labels[UCD_LABEL_LEN];
      char model_data_units[UCD_LABEL_LEN];
      int num_model_comp;
      int model_comp_list[num_model_comp];
      float min_model_data[hdr.num_model_data];
      float max_model_data[hdr.num_model_data];
      float model_data[hdr.num_model_data];
      int model_active_list[hdr.num_model_data];
   }UCD_Binary_File;
```

UCD structure routines 13

This chapter provides UCD library routines:

- Grouped by function
- Listed and defined alphabetically

A C language module that employs UCD routines must use the /usr/avs/include/ucd_defs.h include file.

A FORTRAN language module that employs UCD routines must use the /usr/avs/include/avs.inc include file.

A module that uses UCD routines must be linked with the following libraries:

C module	/usr/avs/lib/libflow_c.a
C coroutine module	/usr/avs/lib/libsim_c.a
FORTRAN module	/usr/avs/lib/libflow_f.a
FORTRAN coroutine module	/usr/avs/lib/libsim_f.a

Grouped by function

The following sections group UCD routines by functional category.

Structure manipulation

- UCDstructure_alloc
- UCDstructure_free
- UCDstructure_set_data
- UCDstructure_set_data_labels
- UCDstructure_set_data_units
- UCDstructure_set_extent
- UCDstructure_set_header_flag

Structure query

- UCDstructure_get_data
- UCDstructure_get_data_label
- UCDstructure_get_data_labels
- UCDstructure_get_data_unit
- UCDstructure_get_data_units
- UCDstructure_get_extent
- UCDstructure_get_header

Cell manipulation

- UCDcell_set_information
- UCDstructure_invalid_cell_minmax
- UCDstructure_set_cell_active
- UCDstructure_set_cell_components
- UCDstructure_set_cell_data
- UCDstructure_set_cell_labels
- UCDstructure_set_cell_minmax
- UCDstructure_set_cell_units

Cell query

- UCDcell_get_information
- UCDstructure_get_cell_active
- UCDstructure_get_cell_components
- UCDstructure_get_cell_data
- UCDstructure_get_cell_label
- UCDstructure_get_cell_labels
- UCDstructure_get_cell_minmax
- UCDstructure_get_cell_unit
- UCDstructure_get_cell_units

Node manipulation

- UCDnode_set_information
- UCDstructure_invalid_node_minmax
- UCDstructure_set_node_active
- UCDstructure_set_node_components
- UCDstructure_set_node_data
- UCDstructure_set_node_labels
- UCDstructure_set_node_minmax
- UCDstructure_set_node_positions
- UCDstructure_set_node_units

Node query

- UCDnode_get_information
- UCDstructure_get_node_active
- UCDstructure_get_node_components
- UCDstructure_get_node_data
- UCDstructure_get_node_label
- UCDstructure_get_node_labels
- UCDstructure_get_node_minmax
- UCDstructure_get_node_positions
- UCDstructure_get_node_unit
- UCDstructure_get_node_units

Listed alphabetically

The following sections group UCD routines alphabetically.

UCDcell_get_information

C

```
#include <ucd_defs.h>
int UCDcell_get_information (structure, cell, name,
                             element_type, material_type, cell_type,
                             mid_edge_flags, node_list)
 UCD_structure  *structure;
 int            cell;
 int            *name;
 char           *element_type;
 int            *material_type;
 int            *cell_type;
 int            *mid_edge_flags;
 int            **node_list;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDcell_get_information (structure, cell, name,
                                 element_type, material_type, cell_type,
                                 mid_edge_flags, node_list)
 INTEGER          structure
 INTEGER          cell
 INTEGER          name
 CHARACTER*(*)    element_type
 INTEGER          material_type
 INTEGER          cell_type
 INTEGER          mid_edge_flags
 INTEGER          node_list
```

This routine finds all the information about a particular cell and returns those values. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
cell	cell to find information

The output arguments are:

name	cell name
element_type	name of element type
material_type	user defined material type
cell_type	cell type (for example, `UCD_TRIANGLE`)
mid_edge_flags	does the cell have mid-edge nodes
node_list	array of node numbers

UCDcell_set_information

C

```
#include <ucd_defs.h>
int UCDcell_set_information (structure, cell, name,
                             element_type, material_type, cell_type,
                             mid_edge_flags, node_list)
 UCD_structure  *structure;
 int            cell;
 int            name;
 char           *element_type;
 int            material_type;
 int            cell_type;
 int            mid_edge_flags;
 int            *node_list;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDcell_set_information (structure, cell, name,
                                 element_type, material_type, cell_type,
                                 mid_edge_flags, node_list)
 INTEGER          structure
 INTEGER          cell
 INTEGER          name
 CHARACTER*(*)    element_type
 INTEGER          material_type
 INTEGER          cell_type
 INTEGER          mid_edge_flags
 INTEGER          node_list
```

This routine sets all the information about a particular cell. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
cell	cell to find information
name	cell name
element_type	name of element type
material_type	user defined material type
cell_type	cell type (for example, `UCD_TRIANGLE`)
mid_edge_flags	does the cell have mid-edge nodes
node_list	array of node numbers

UCDnode_get_information

C

```
#include <ucd_defs.h>
int UCDnode_get_information (structure, node, name, ncells,
                             cell_list)
  UCD_structure  *structure;
  int            node;
  int            *name;
  int            *ncells;
  int            **cell_list;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDnode_get_information (structure, node, name,
                                 ncells, cell_list)
  INTEGER        structure
  INTEGER        node
  INTEGER        name
  INTEGER        ncells
  INTEGER        cell_list
```

This routine finds all the information about a particular node and returns those values. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
node	node to find information

The output arguments are:

name	node name
ncells	number of cells in *cell_list*
cell_list	array of cell numbers

UCDnode_set_information

C

```
#include <ucd_defs.h>
int UCDnode_set_information (structure, node, name, ncells,
                             cell_list)
  UCD_structure  *structure;
  int            node;
  int            name;
  int            ncells;
  int            *cell_list;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDnode_set_information (structure, node, name,
                                 ncells, cell_list)
  INTEGER        structure
  INTEGER        node
  INTEGER        name
  INTEGER        ncells
  INTEGER        cell_list
```

This routine sets all the information about a particular node. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
node	node to find information
name	node name
ncells	number of cells in *cell_list*
cell_list	array of cell numbers

UCDstructure_alloc

C

```
#include <ucd_defs.h>
char *UCDstructure_alloc (name, data_veclen, name_flag,
                          ncells, cell_tsize, cell_veclen, nnodes,
                          node_csize, node_veclen, util_flag)
  char            *name;
  int             data_veclen;
  int             name_flag;
  int             ncells;
  int             cell_tsize;
  int             cell_veclen;
  int             nnodes;
  int             node_csize;
  int             node_veclen;
  int             util_flag;
```

FORTRAN

```
#include <avs.inc>
CHARACTER*(*) UCDstruct_alloc (name, data_veclen,
                               name_flag, ncells, cell_tsize,
                               cell_veclen, nnodes, node_csize,
                               node_veclen, util_flag)
  CHARACTER*(*)   name
  INTEGER         data_veclen
  INTEGER         name_flag
  INTEGER         ncells
  INTEGER         cell_tsize
  INTEGER         cell_veclen
  INTEGER         nnodes
  INTEGER         node_csize
  INTEGER         node_veclen
  INTEGER         util_flag
```

This routine creates a new top-level structure and returns a pointer to that structure. If a new structure could not be allocated, then NULL is returned.

The input arguments are:

name	structure name
data_veclen	length of structure data vector

name_flag	are node/cell names characters or integers:
	`char = UCD_CHAR`
	`int = UCD_INT`

ncells	number of cells in the structure
cell_tsize	expected size of the cell's connectivity list
cell_veclen	length of cell data vector
nnodes	number of nodes in the structure
node_csize	expected size of the node's connectivity list
node_veclen	length of node data vector
util_flag	utility flag for general usage (do not use two rightmost bits)

UCDstructure_free

C

```
#include <ucd_defs.h>
int UCDstructure_free (structure)
 UCD_structure *structure;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_free (structure)
 INTEGER        structure
```

This routine frees the storage used by *structure*. It returns 1 if successful and 0 if failure.

The input argument is:

structure	structure to free

UCDstructure_get_cell_active

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_active (structure, active)
 UCD_structure  *structure;
 int            **active;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_active (structure,
                active)
 INTEGER        structure
 INTEGER        active
```

This routine returns a pointer to the array containing the cell active component list. For instance, if there are four different components in the cell data vector and the module is using the second component, the list would be (0 1 0 0). The active list is useful when trying to communicate from one module to another which component in the cell data is being worked on. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output argument is:

active pointer to the cell active component list

UCDstructure_get_cell_components

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_components (structure,
                components)
 UCD_structure  *structure;
 int            **components;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_components (structure,
                components)
 INTEGER        structure
 INTEGER        components
```

This routine returns pointers to the array containing the cell component list. For instance, if there are four different components in the cell data vector (for example, scalar, 3-vector, 2-vector, scalar), the component list would be (1 3 2 1). It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output argument is:

components pointer to the cell component list

UCDstructure_get_cell_data

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_data (structure, data)
 UCD_structure  *structure;
 float          **data;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_data (structure, data)
 INTEGER        structure
 REAL           data
```

This routine returns a pointer to the array containing the data vectors for all of the cells in the structure. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output argument is:

data pointer to the cell data vectors

UCDstructure_get_cell_label

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_label (structure, number,
                                 label)
 UCD_structure  *structure;
 int            number;
 char           *label;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_label (structure,
                                  number, label)
 INTEGER        structure
 INTEGER        number
 CHARACTER*(*)  label
```

This routine allows you to query the label for an individual component in the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to get labels in
number	individual component number

The output argument is

labels	string with labels included

UCDstructure_get_cell_labels

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_labels (structure, labels,
                    delimiter)
  UCD_structure  *structure;
  char           *labels;
  char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_labels (structure,
                    labels, delimiter)
  INTEGER        structure
  CHARACTER*(*)  labels
  CHARACTER*(*)  delimiter
```

This routine allows you to get the labels for each component in the structure. These labels are for cases when there is cell-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, mach number, and so on. In turn, these labels would appear on the dials.

Example: *labels* = `"temp;density;mach number"`
delimiter = `";"`

The input argument is:

structure structure to find information

The output arguments are:

labels string with labels included

delimiter delimiter between each label

UCDstructure_get_cell_minmax

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_minmax (structure, min, max)
 UCD_structure *structure;
 float         *min;
 float         *max;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_minmax (structure, min,
                max)
 INTEGER       structure
 REAL          min
 REAL          max
```

This routine allows you to obtain the range of the structure cell data. *min* and *max* are arrays of dimension *structure->cell_veclen*. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to get min/max in

The output arguments are:

min value of minimum data point

max value of maximum data point

UCDstructure_get_cell_unit

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_unit (structure, number, label)
 UCD_structure  *structure;
 int            number;
 char           *label;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_unit (structure, number,
                label)
 INTEGER        structure
 INTEGER        number
 CHARACTER*(*)  label
```

This routine allows you to query the label for an individual unit in the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to get labels in
number	individual component number

The output argument is:

labels	string with labels included

UCDstructure_get_cell_units

C

```
#include <ucd_defs.h>
int UCDstructure_get_cell_units (structure, labels,
                                 delimiter)
  UCD_structure  *structure;
  char           *labels;
  char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_cell_units (structure, labels,
                                  delimiter)
  INTEGER          structure
  CHARACTER*(*)    labels
  CHARACTER*(*)    delimiter
```

This routine allows you to get the unit labels for each component in the structure. These labels are for cases when there is cell-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as degrees, meters, and so on.

Example: *labels* = `"degrees;meters"`
delimiter = `";"`

The input argument is:

structure structure to find information

The output arguments are:

labels string with labels included

delimiter delimiter between each label

UCDstructure_get_data

C

```
#include <ucd_defs.h>
int UCDstructure_get_data (structure, data)
 UCD_structure  *structure;
 float          **data;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_data (structure, data)
 INTEGER        structure
 REAL           data
```

This routine returns a pointer to the array containing the data vector for the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
data	pointer to the structure data vector

UCDstructure_get_data_label

C

```
#include <ucd_defs.h>
int UCDstructure_get_data_label (structure, number,
                                 label)
 UCD_structure  *structure;
 int            number;
 char           *label;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_data_label (structure,
                                  number, label)
 INTEGER        structure
 INTEGER        number
 CHARACTER*(*)  label
```

This routine allows you to query the label for an individual component in the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to get labels in
number	individual component number

The output argument is:

labels	string with labels included

UCDstructure_get_data_labels

C

```
#include <ucd_defs.h>
int UCDstructure_get_data_labels (structure, labels,
                    delimiter)
 UCD_structure      *structure;
 char               *labels;
 char               *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_data_labels (structure,
                    labels, delimiter)
 INTEGER            structure
 CHARACTER*(*)      labels
 CHARACTER*(*)      delimiter
```

This routine allows you to get the labels for each component in the structure. These labels are for cases when there is structure-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, mach number, and so on. In turn, these labels would appear on the dials.

Example: *labels* = `"temp;density;mach number"`
delimiter = `";"`

The input argument is:

structure structure to find information

The output arguments are:

labels string with labels included

delimiter delimiter between each label

UCDstructure_get_data_unit

C

```
#include <ucd_defs.h>
int UCDstructure_get_data_unit (structure, number, label)
 UCD_structure  *structure;
 int            number;
 char           *label;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_data_unit (structure, number,
                label)
 INTEGER        structure
 INTEGER        number
 CHARACTER*(*)  label
```

This routine allows you to query the label for an individual unit in the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure structure to get labels in

number individual component number

The output argument is:

label string with labels included

UCDstructure_get_data_units

C

```
#include <ucd_defs.h>
int UCDstructure_get_data_units (structure, labels,
                                 delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_data_units (structure, labels,
                                  delimiter)
 INTEGER         structure
 CHARACTER*(*)   labels
 CHARACTER*(*)   delimiter
```

This routine allows you to get the unit labels for each component in the structure. These labels are for cases when there is structure-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as degrees, meters, and so on.

Example: *labels* = `"degrees;meters"`
delimiter = `";"`

The input argument is:

structure structure to find information

The output arguments are:

labels string with labels included

delimiter delimiter between each label

UCDstructure_get_extent

C

```
#include <ucd_defs.h>
int UCDstructure_get_extent (structure, min_extent,
                             max_extent)
 UCD_structure  *structure;
 float          *min_extent;
 float          *max_extent;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_extent (structure, min_extent,
                              max_extent)
 INTEGER        structure
 REAL           min_extent
 REAL           max_extent
```

This routine allows you to obtain the extent of the structure. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output arguments are:

min_extent minimum coordinate extent of structure

max_extent maximum coordinate extent of structure

UCDstructure_get_header

C

```
#include <ucd_defs.h>
int UCDstructure_get_header(structure, name, data_veclen,
                            name_flag, ncells, cell_veclen, nnodes.
                            node_veclen, util_flag)
 UCD_structure  *structure;
 char           *name;
 int            *data_veclen;
 int            *name_flag;
 int            *ncells;
 int            *cell_veclen;
 int            *nnodes;
 int            *node_veclen;
 int            *util_flag;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_header(structure, name,
                             data_veclen, name_flag, ncells, cell_veclen,
                             nnodes, node_veclen, util_flag)
 INTEGER          structure
 CHARACTER*(*)    name
 INTEGER          data_veclen
 INTEGER          name_flag
 INTEGER          ncells
 INTEGER          cell_veclen
 INTEGER          nnodes
 INTEGER          node_veclen
 INTEGER          util_flag
```

This routine finds all the header information about a UCD structure and returns those values. It returns 1 if successful and 0 if failure.

The input argument is:

structure	structure to find information

The output arguments are:

name	structure name
data_veclen	length of structure data vector

name_flag	are node/cell names characters or integers:
	`char` = `UCD_CHAR`
	`int` = `UCD_INT`

ncells	number of cells in the structure
cell_veclen	length of cell data vector
nnodes	number of nodes in the structure
node_veclen	length of node data vector
util_flag	utility flag

UCDstructure_get_node_active

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_active (structure, active)
 UCD_structure  *structure;
 int            **active;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_active (structure,
                active)
 INTEGER        structure
 INTEGER        active
```

This routine returns a pointer to the array containing the node active component list. For instance, if there are four different components in the node data vector and the module is using the second component, the list would be (0 1 0 0). It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output argument is:

active pointer to the node active component list

UCDstructure_get_node_components

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_components (structure,
                components)
 UCD_structure  *structure;
 int            **components;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_components (structure,
                components)
 INTEGER        structure
 INTEGER        components
```

This routine returns pointers to the array containing the node component list. For instance, if there are four different components in the node data vector (for example, scalar, 3-vector, 2-vector, scalar), the component list would be (1 3 2 1). It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output argument is:

components pointer to the node component list

UCDstructure_get_node_data

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_data (structure, data)
 UCD_structure  *structure;
 float          **data;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_data (structure, data)
 INTEGER        structure
 REAL           data
```

This routine returns pointers to the array containing the data vectors for the nodes. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output argument is:

data pointer to the node data vectors

UCDstructure_get_node_label

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_label (structure, number,
                                 label)
 UCD_structure  *structure;
 int            number;
 char           *label;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_label (structure,
                                  number, label)
 INTEGER        structure
 INTEGER        number
 CHARACTER*(*)  label
```

This routine allows you to query the label for an individual component in the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to get label in
number	individual component number

The output argument is:

label	string with label included

UCDstructure_get_node_labels

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_labels (structure, labels,
                                  delimiter)
  UCD_structure  *structure;
  char           *labels;
  char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_labels (structure,
                                   labels, delimiter)
  INTEGER        structure
  CHARACTER*(*)  labels
  CHARACTER*(*)  delimiter
```

This routine allows you to get the labels for each component in the structure. These labels are for cases when there is node-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, mach number, and so on. In turn, these labels would appear on the dials.

Example: *labels* = `"temp;density;mach number"`
delimiter = `";"`

The input argument is:

structure structure to find information

The output arguments are:

labels string with labels included

delimiter delimiter between each label

UCDstructure_get_node_minmax

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_minmax (structure, min, max)
 UCD_structure  *structure;
 float          *min;
 float          *max;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_minmax (structure, min,
                max)
 INTEGER        structure
 REAL           min
 REAL           max
```

This routine allows you to obtain the range of the structure node data. *min* and *max* are arrays of dimension *structure->node_veclen*. It returns 1 if successful and 0 if failure.

The input argument is:

structure	structure to get min/max in

The output arguments are:

min	value of minimum data point
max	value of maximum data point

UCDstructure_get_node_positions

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_positions (structure, x, y, z)
 UCD_structure  *structure;
 float          **x, **y, **z;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_positions (structure,
                x, y, z)
 INTEGER        structure
 REAL           x, y, z
```

This routine returns pointers to the arrays containing the x-, y- and z-coordinates of node positions. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to find information

The output argument is:

x, y, z pointer to the x-, y-, z-arrays

UCDstructure_get_node_unit

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_unit (structure, number, label)
 UCD_structure  *structure;
 int            number;
 char           *label;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_unit (structure, number,
                label)
 INTEGER        structure
 INTEGER        number
 CHARACTER*(*)  label
```

This routine allows you to query the label for an individual unit in the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure structure to get labels in

number individual component number

The output argument is:

label string with labels included

UCDstructure_get_node_units

C

```
#include <ucd_defs.h>
int UCDstructure_get_node_units (structure, labels,
                                 delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_get_node_units (structure, labels,
                                  delimiter)
 INTEGER         structure
 CHARACTER*(*)   labels
 CHARACTER*(*)   delimiter
```

This routine allows you to get the unit labels for each component in the structure. These labels are for cases when there is node-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as degrees, meters, and so on.

Example: *labels* = `"degrees;meters"`
delimiter = `";"`

The input argument is:

structure structure to find information

The output arguments are:

labels string with labels included

delimiter delimiter between each label

UCDstructure_invalid_cell_minmax

C

```
#include <ucd_defs.h>
int UCDstructure_invalid_cell_minmax (structure)
 UCD_structure *structure;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_invalid_cell_minmax (structure)
 INTEGER        structure
```

This routine allows you to set the min/max range of the UCD structure cell data to be invalid. This function should be used after the structure data has been changed by the module and the module does not want to spend the time recomputing the cell min/max. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to set cell min/max invalid

UCDstructure_invalid_node_minmax

C

```
#include <ucd_defs.h>
int UCDstructure_invalid_node_minmax (structure)
 UCD_structure *structure;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_invalid_node_minmax (structure)
 INTEGER        structure
```

This routine allows you to set the min/max range of the UCD structure node data to be invalid. This function should be used after the structure data has been changed by the module and the module does not want to spend the time recomputing the node min/max. It returns 1 if successful and 0 if failure.

The input argument is:

structure structure to set node min/max invalid

UCDstructure_set_cell_active

C

```
#include <ucd_defs.h>
int UCDstructure_set_cell_active (structure, active)
 UCD_structure  *structure;
 int            *active;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_cell_active (structure,
                active)
 INTEGER        structure
 INTEGER        active
```

This routine sets the array containing the cell active component list. For instance, if there are four different components in the cell data vector and the module is using the second component, the list would be (0 1 0 0). It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
active	pointer to the cell active component list

UCDstructure_set_cell_components

C

```
#include <ucd_defs.h>
int UCDstructure_set_cell_components (structure,
                components, number)
 UCD_structure  *structure;
 int            *components;
 int            number;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_cell_components (structure,
                components, number)
 INTEGER        structure
 INTEGER        components
 INTEGER        number
```

This routine copies the array containing the cell component list. For instance, if there are four different components in the cell data vector (for example, scalar, 3-vector, 2-vector, scalar), the component list would be (1 3 2 1). It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
components	pointer to the cell component list
number	number of components in the list

UCDstructure_set_cell_data

C

```
#include <ucd_defs.h>
int UCDstruct_set_cell_data (structure, data)
 UCD_structure  *structure;
 float          *data;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_cell_data (structure, data)
 INTEGER        structure
 REAL           data
```

This routine copies the cell data from the array pointed to by *data* into the structure's cell data array. There should be *cell_veclen*ncells* data elements in this array. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
data	pointer to the cell data vectors

UCDstructure_set_cell_labels

C

```
#include <ucd_defs.h>
int UCDstructure_set_cell_labels (structure, labels,
                delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_cell_labels (structure,
                labels, delimiter)
 INTEGER        structure
 CHARACTER*(*)  labels
 CHARACTER*(*)  delimiter
```

This routine allows you to set the labels for each component in the structure. These labels are for cases when there is cell-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, mach number, and so on. In turn, these labels would appear on the dials.

Example: *labels* = `"temp;density;mach number"`
delimiter = `";"`

The input arguments are:

structure	structure to find information
labels	string with labels included
delimiter	delimiter between each label

UCDstructure_set_cell_minmax

C

```
#include <ucd_defs.h>
int UCDstructure_set_cell_minmax (structure, min, max)
 UCD_structure  *structure;
 float          *min;
 float          *max;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_cell_minmax (structure, min,
                                   max)
 INTEGER        structure
 REAL           min
 REAL           max
```

This routine allows you to set the range of the structure cell data. It should be noted that *min* and *max* are arrays of dimension *structure* –> *cell_veclen*. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to set min/max in
min	value of minimum data point
max	value of maximum data point

UCDstructure_set_cell_units

C

```
#include <ucd_defs.h>
int UCDstructure_set_cell_units (structure, labels,
                                 delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_cell_units (structure, labels,
                                  delimiter)
 INTEGER         structure
 CHARACTER*(*)   labels
 CHARACTER*(*)   delimiter
```

This routine allows you to set the unit labels for each component in the structure. These labels are for cases when there is cell-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as degrees, meters, and so on.

Example: *labels* = `"degrees;meters"`
delimiter = `";"`

The input arguments are:

structure	structure to find information
labels	string with labels included
delimiter	delimiter between each label

UCDstructure_set_data

C

```
#include <ucd_defs.h>
int UCDstructure_set_data (structure, data)
 UCD_structure  *structure;
 float          *data;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_data (structure, data)
 INTEGER        structure
 REAL           data
```

This routine copies the data from the array pointed to by *data* into the structure's data array. There should be *data_veclen* data elements in this array. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
data	pointer to the data vector

UCDstructure_set_data_labels

C

```
#include <ucd_defs.h>
int UCDstructure_set_data_labels (structure, labels,
                delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_data_labels (structure,
                labels, delimiter)
 INTEGER        structure
 CHARACTER*(*)  labels
 CHARACTER*(*)  delimiter
```

This routine allows you to set the labels for each component in the structure. These labels are for cases when there is structure-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, mach number, and so on. In turn, these labels would appear on the dials.

Example: *labels* = `"temp;density;mach number"`
delimiter = `";"`

The input arguments are:

structure	structure to find information
labels	string with labels included
delimiter	delimiter between each label

UCDstructure_set_data_units

C

```
#include <ucd_defs.h>
int UCDstructure_set_data_units (structure, labels,
                                 delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_data_units (structure, labels,
                                  delimiter)
 INTEGER        structure
 CHARACTER*(*)  labels
 CHARACTER*(*)  delimiter
```

This routine allows you to set the unit labels for each component in the structure. These labels are for cases when there is structure-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as degrees, meters, and so on.

Example: *labels* = `"degrees;meters"`
delimiter = `";"`

The input arguments are:

structure	structure to find information
labels	string with labels included
delimiter	delimiter between each label

UCDstructure_set_extent

C

```
#include <ucd_defs.h>
int UCDstructure_set_extent (structure, min_extent,
                             max_extent)
 UCD_structure  *structure;
 float          *min_extent;
 float          *max_extent;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_extent (structure, min_extent,
                              max_extent)
 INTEGER        structure
 REAL           min_extent
 REAL           max_extent
```

This routine allows you to set the extent of the structure. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
min_extent	minimum coordinate extent of structure
max_extent	maximum coordinate extent of structure

UCDstructure_set_header_flag

C

```
#include <ucd_defs.h>
int UCDstructure_set_header_flag (structure, util_flag)
 UCD_structure *structure;
 int           util_flag;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_header_flag (structure,
                util_flag)
 INTEGER       structure
 INTEGER       util_flag
```

This routine sets the header flag bits. It returns 1 if successful and 0 if failure.

In *util_flag*, the eight rightmost bits are reserved for internal use.

The input arguments are:

structure	structure to find information
util_flag	utility flag

UCDstructure_set_node_active

C

```
#include <ucd_defs.h>
int UCDstructure_set_node_active (structure, active)
 UCD_structure *structure;
 int           *active;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_node_active (structure,
                active)
 INTEGER       structure
 INTEGER       active
```

This routine sets the array containing the node active component list. For instance, if there are four different components in the node data vector and the module is using the second component, the list would be (0 1 0 0). The active list is useful when trying to communicate from one module to another which component in the node data is being worked on. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
active	pointer to the node active component list

UCDstructure_set_node_components

C

```
#include <ucd_defs.h>
int UCDstructure_set_node_components (structure,
                components, number)
 UCD_structure  *structure;
 int            *components;
 int            number;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_node_components (structure,
                components, number)
 INTEGER        structure
 INTEGER        components
 INTEGER        number
```

This routine copies the array containing the node component list. For instance, if there are four different components in the node data vector (for example, scalar, 3-vector, 2-vector, scalar), the component list would be (1 3 2 1). It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
components	pointer to the node component list
number	number of components in the list

UCDstructure_set_node_data

C

```
#include <ucd_defs.h>
int UCDstructure_set_node_data (structure, data)
 UCD_structure  *structure;
 float          *data;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_node_data (structure, data)
 INTEGER        structure
 REAL           data
```

This routine copies the node data from the array pointed to by *data* into the structure's node data array. There should be *node_veclen*nnodes* data elements in this array. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
data	pointer to the node data vectors

UCDstructure_set_node_labels

C

```
#include <ucd_defs.h>
int UCDstructure_set_node_labels (structure, labels,
                                  delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_node_labels (structure,
                                   labels, delimiter)
 INTEGER        structure
 CHARACTER*(*)  labels
 CHARACTER*(*)  delimiter
```

This routine allows you to set the labels for each component in the structure. These labels are for cases when there is node-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, mach number, and so on. In turn, these labels would appear on the dials.

Example: *labels* = `"temp;density;mach number"`
delimiter = `";"`

The input arguments are:

structure	structure to find information
labels	string with labels included
delimiter	delimiter between each label

UCDstructure_set_node_minmax

C

```
#include <ucd_defs.h>
int UCDstructure_set_node_minmax (structure, min, max)
 UCD_structure  *structure;
 float          *min;
 float          *max;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_node_minmax (structure, min,
                max)
 INTEGER        structure
 REAL           min
 REAL           max
```

This routine allows you to set the range of the structure node data. *min* and *max* are arrays of dimension *structure->node_veclen*. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to set min/max in
min	value of minimum data point
max	value of maximum data point

UCDstructure_set_node_positions

C

```
#include <ucd_defs.h>
int UCDstructure_set_node_positions (structure, x, y, z)
 UCD_structure  *structure;
 float          *x, *y, *z;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_node_positions (structure,
                x, y, z)
 INTEGER        structure
 REAL           x, y, z
```

This routine copies the x-, y-, and z-coordinate arrays from the arrays pointed to by *x, y,* and *z* into the structure's node position arrays. There should be *nnodes* coordinates in each array. It returns 1 if successful and 0 if failure.

The input arguments are:

structure	structure to find information
x, y, z	pointer to the x-, y-, z-arrays

UCDstructure_set_node_units

C

```
#include <ucd_defs.h>
int UCDstructure_set_node_units (structure, labels,
                                 delimiter)
 UCD_structure  *structure;
 char           *labels;
 char           *delimiter;
```

FORTRAN

```
#include <avs.inc>
INTEGER UCDstruct_set_node_units (structure, labels,
                                  delimiter)
 INTEGER         structure
 CHARACTER*(*)   labels
 CHARACTER*(*)   delimiter
```

This routine allows you to set the unit labels for each component in the structure. These labels are for cases when there is node-based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD data set, you might want to label components of the field as degrees, meters, and so on.

Example: *labels* = `"degrees;meters"`
delimiter = `";"`

The input arguments are:

structure	structure to find information
labels	string with labels included
delimiter	delimiter between each label

Modules 14

Components

A module is a fundamental building block in a ConvexAVS network and is used for 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, or renderer. These module type distinctions affect only the presentation of the module in the ConvexAVS user interface. Module type determines which menu 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
Orange	UCD structure
Multi-color	Field
Light purple	Integer
Light purple	Byte
Dark purple	Real
Green	String

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. Filters and mappers process data and therefore have both input and output ports.

Each input port of an initialized module 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 module may have zero or more parameters. A *parameter* is a variable that can be changed by the user or by any other module that controls the function of that module. Parameter types include most primitive ConvexAVS data types along with constrained variants such as boolean and choice. Parameters have names and initial values. Some parameters also have bounding information such as a range of allowed values.

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.

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.

When appropriate, a module can alter the properties of a parameter dynamically. ConvexAVS then updates any widget associated with the parameter.

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* (subroutines only). 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 allocate memory or create a window, but it 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 can free memory or destroy a window, but it 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 138 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 138
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 139 is the FORTRAN version of the same routine.

Figure 139
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.

The compute function returns an integer value, indicating the validity of its output data. A non-zero value indicates the module has successfully computed the output data. A zero value indicates the output data is invalid (for example, a module encounters 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 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 module_1_desc();
int module_2_desc();
int module_3_desc();
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.

An example coroutine module program, qix.c is provided in the /usr/avs/examples directory.

Module examples

Source code for example modules 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`(3). 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`.

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.

By default, ConvexAVS searches for help files in the directories under /usr/avs/runtime/help.

Using the AVS_HELP_PATH environment variable

When you invoke ConvexAVS, you can 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/biff/avs:/usr/buff/avs
```

You could also set this environment variable in your shell start-up file (.cshrc for the C shell, .profile for the Bourne shell).

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 a single coroutine module or FORTRAN subroutine module, or one or more C language subroutine modules. To build your executable, you will need to use different ConvexAVS include files and libraries, depending on the source language of your module and whether it is a subroutine or 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.

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 language 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 geom.inc file.

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 25.

Table 25
Module types

Module type	Language	Library
Subroutine	C	libflow_c.a
Subroutine	FORTRAN	libflow_f.a
Coroutine	C	libsim_c.a
Coroutine	FORTRAN	libsim_f.a

In addition, modules must be linked with the geometry and utility 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
```

Note

Modules that use floating point must use IEEE floats either specifying the `-fi` flag to the compiler or having the IEEE floating point type as the machine default.

Converting applications into modules

Many existing simulations, batch data converters, and other scientific applications can be converted into modules.

Converting 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.

Note

Coroutine modules are not tracked by the AVS Animator module.

- 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.

Converting 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 internal workings

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:

1. Sets up a connection to ConvexAVS.
2. Invokes the `AVSinit_modules` routine. This routine in turn invokes the description functions of all modules in the executable file.
3. Conveys to ConvexAVS the module declarations for all modules in the executable file.
4. 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:

1. Sets up a connection to ConvexAVS.
2. Invokes the `AVSinit_modules` routine. This routine, in turn, invokes description functions of all modules in the executable file.
3. Conveys to ConvexAVS module declarations for all modules in the executable file.
4. Sets up an instance of the module that can receive data from and send data to ConvexAVS.
5. Invokes the module initialization function, if any.
6. 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:

1. Sets up a connection to ConvexAVS.
2. Invokes the module's description function.
3. Conveys to ConvexAVS the module declarations for the module.
4. 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:

1. Sets up a connection to ConvexAVS.
2. Invokes the module's description function.
3. Conveys to ConvexAVS module declarations for the module.
4. Sets up an instance of the module that can receive data from and send data to ConvexAVS.
5. Invokes the module initialization function, if any.
6. 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:

1. Call the `AVScorout_wait` routine. This routine waits until one of the module's inputs or parameters changes, then returns.
2. Call the `AVScorout_input` routine. This routine obtains the module's inputs and parameters from ConvexAVS.
3. Perform the module's computation.
4. 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 routines 15

This chapter provides module routines:

- Grouped by function
- Listed and defined alphabetically

ConvexAVS provides a number of include files you should use in conjunction with these routines:

- A C language module that employs these routines must use the /usr/avs/include/avs.h include file.
- A FORTRAN language module that employs these routines must use the /usr/avs/include/avs.inc include file.
- In addition, other include files are required by some routines. These files are specified with the individual descriptions.

Grouped by function

The following sections group module routines by function.

Initializing modules

- AVSinit_from_module_list
- AVSinit_modules
- AVSmodule_from_desc

Modifying and interpreting parameters

- AVSchoice_number
- AVSmodify_float_parameter
- AVSmodify_parameter
- AVSmodify_parameter_prop
- AVSparameter_visible

Monitoring status

- AVSmodule_status

Command Language Interpreter

- AVScommand

Coroutine modules

- AVScorout_event_wait
- AVScorout_exec
- AVScorout_init
- AVScorout_input
- AVScorout_mark_changed
- AVScorout_output
- AVScorout_set_sync
- AVScorout_wait
- AVScorout_X_wait

Module description functions

- AVSadd_float_parameter
- AVSadd_parameter
- AVSadd_parameter_prop
- AVSautofree_output
- AVSconnect_widget
- AVScreate_input_port
- AVScreate_output_port
- AVSinitialize_output
- AVSload_user_data_types
- AVSset_compute_proc
- AVSset_destroy_proc
- AVSset_init_proc
- AVSset_input_class
- AVSset_module_flags
- AVSset_module_name
- AVSset_output_class
- AVSset_output_flags
- AVSset_parameter_class

Selective computation

- AVSinput_changed
- AVSmark_output_unchanged
- AVSparameter_changed

Creating fields

- AVSbuild_2d_field
- AVSbuild_3d_field
- AVSbuild_field
- AVSdata_alloc
- AVSdata_free
- AVSfield_alloc
- AVSfield_copy_points
- AVSfield_free
- AVSfield_make_template
- AVSport_field

Accessing fields

- AVSfield_data_offset
- AVSfield_data_ptr
- AVSfield_get_dimensions
- AVSfield_get_extent
- AVSfield_get_int
- AVSfield_get_label
- AVSfield_get_labels
- AVSfield_get_minmax
- AVSfield_get_unit
- AVSfield_get_units
- AVSfield_invalid_minmax
- AVSfield_points_offset
- AVSfield_points_ptr
- AVSfield_reset_minmax
- AVSfield_set_extent
- AVSfield_set_int
- AVSfield_set_labels
- AVSfield_set_minmax
- AVSfield_set_units

Accessing user data

- AVSudata_get_double
- AVSudata_get_int
- AVSudata_get_real
- AVSudata_get_string
- AVSudata_set_double
- AVSudata_set_int
- AVSudata_set_real
- AVSudata_set_string

Accessing colormaps

- AVScolormap_get
- AVScolormap_set

Accessing FORTRAN arrays

- AVSptr_alloc
- AVSptr_offset

Accessing FORTRAN single bytes

- AVSload_byte
- AVSstore_byte

Handling errors

- AVSdebug
- AVSerror
- AVSfatal
- AVSinformation
- AVSmessage
- AVSmessage_sub
- AVSwarning

Defined alphabetically

The following sections define module routines alphabetically.

AVSadd_float_parameter

C

```
int AVSadd_float_parameter(param_name, init,
                           minval, maxval)
  char          *param_name;
  double        init, minval, maxval;
```

FORTRAN

There is no FORTRAN equivalent for this routine. Use `AVSADD_PARAMETER`.

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.

This routine returns an integer parameter identifier that is used as an argument to other routines such as `AVSconnect_widget`.

AVSadd_parameter

C

```
int AVSadd_parameter(param_name, type, init,
                     minval, maxval)
  char          *param_name, *type;
  int           init, minval, maxval;
```

FORTRAN

```
AVSADD_PARAMETER(NAME, TYPE, INIT, MINVAL,
                 MAXVAL)
  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 `int`s in C and `integer`s 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 `AVSadd_float_parameter` routine 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 26 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 26
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*(*)`
"string_block"	`char *`	`CHARACTER*(*)`
"choice"	`char *`	`CHARACTER*(*)`

Some additional information on 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.

Modules that have oneshot parameters should not be used with the animation facility in ConvexAVS. This parameter is currently incompatible with animation software.

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. In C, the value must be NULL), an empty string, or an allocated string. FORTRAN must pass a valid string of at least one character in length for the value to be recognized properly. Because trailing spaces are stripped off, a single space works as an empty string and is also handled properly when being used as a delimiter value. 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.
string_block	This value might contain embedded newline characters to delimit separate lines in a block of text. The entire value is displayed through several types of widgets for more extensive text output. In all other respects, it is equivalent to the string data type.
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.

This routine returns an integer parameter identifier that is used as an argument to 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 type-in 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`.

These properties are defined in the following paragraphs.

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.
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.

Table 27 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.

Table 27
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

AVSautofree_output

C

```
AVSautofree_output(out_port)
  int        out_port;
```

FORTRAN

```
AVSAUTOFREE_OUTPUT(OUT_PORT)
  INTEGER          OUT_PORT
```

This routine sets a flag 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.

Note

This routine cannot use shared memory and is provided for backward compatibility with previous releases of ConvexAVS. Use the `AVSdata_alloc` and `AVSfield_alloc` routines when creating new modules.

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.

Note

This routine cannot use shared memory and is provided for backward compatibility with previous releases of ConvexAVS. Use the `AVSdata_alloc` and `AVSfield_alloc` routines when creating new modules.

AVSbuild_field

C

```
#include <avs/field.h>
AVSfield * AVSbuild_field(ndim, veclen, uniform, ncoord,
                          type, dim1, dim2, ..., data, coords)
  int           ndim, veclen, uniform, ncoord, type;
  int           dim1, dim2, ...;
  unsigned char *data;
  float         *coords;
```

FORTRAN

```
AVSBUILD_FIELD(NDIM, VECLEN, UNIFORM, NCOORD,
               TYPE, DIM1, DIM2, ..., DATA, COORDS)
  INTEGER       NDIM, 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 a field structure. The arguments are:

ndim — A positive integer specifying the number of dimensions in the computational space of the field.

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.

Note

This routine cannot use shared memory and is provided for backward compatibility with previous releases of ConvexAVS. Use the `AVSdata_alloc` and `AVSfield_alloc` routines when creating new modules.

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.

AVScolormap_get

C

```
#include <avs/colormap.h>
int AVScolormap_get(cmap, max_size, size, lower, upper, hue,
                    saturation, value, alpha)
   AVScolormap*cmap;
   int          max_size;
   int          *size;
   float        *lower, *upper, *hue, *saturation, *value, *alpha;
```

FORTRAN

```
INTEGER AVSCOLORMAP_GET(CMAP, MAX_SIZE, SIZE,
                        LOWER, UPPER, HUE, SATURATION,
                        VALUE, ALPHA)
   INTEGER          CMAP, MAX_SIZE, SIZE
   REAL             LOWER, UPPER
   REAL             HUE(n), SATURATION(n), VALUE(n),
                    ALPHA(n)
```

This routine must be used by FORTRAN modules to access the contents of a colormap input or output when the SINGLE_ARG_DATA flag has been set using AVSSET_MODULE_FLAGS. For each colormap input or output, the module passes a single integer argument that is a pointer to a colormap. You can pass the pointer to this routine to get the contents of the colormap. The *HUE*, *SATURATION*, *VALUE*, and *ALPHA* data are copied into the arrays provided by the caller. It is up to the caller to ensure that these arrays are large enough to hold the returned information. ConvexAVS issues an error if the colormap size exceeds the *max_size* argument. A return of 1 indicates success, while a 0 indicates failure.

AVScolormap_set

C

```
#include <avs/colormap.h>
int AVScolormap_set(cmap, size, lower, upper, hue, saturation,
                    value, alpha)
   AVScolormap*cmap;
   int          *size;
   float        *lower, *upper, *hue, *saturation, *value, *alpha;
```

FORTRAN

```
INTEGER AVSCOLORMAP_SET(CMAP, SIZE, LOWER, UPPER,
                        HUE, SATURATION, VALUE, ALPHA)
   INTEGER          CMAP, SIZE
   REAL             LOWER, UPPER
   REAL             HUE(n), SATURATION(n), VALUE(n),
                    ALPHA(n)
```

This routine must be used by FORTRAN modules to access the contents of a colormap input or output when the SINGLE_ARG_DATA flag has been set using AVSSET_MODULE_FLAGS. For each colormap input or output, the module is passed a single integer argument that is a pointer to a colormap. You can pass the pointer to this routine to set the contents of the colormap. The four arrays for *HUE*, *SATURATION*, *VALUE*, and *ALPHA* are copied from the provided arrays. A return of 1 indicates success, while a 0 indicates failure.

AVScommand

C

```
AVScommand(destination, command_buffer, output_buffer,
           error_buffer)
  char     *destination, *command_buffer, **output_buffer,
           **error_buffer;
```

FORTRAN

```
AVSCOMMAND(DESTINATION, COMMAND_BUFFER,
  OUTPUT_BUFFER, ERROR_BUFFER)
  CHARACTER*(*)          DESTINATION,
                         COMMAND_BUFFER
  CHARACTER*<maxsize>    OUTPUT_BUFFER,
                         ERROR_BUFFER
```

Use this routine to send Command Language Interpreter commands to the ConvexAVS kernel.

The *destination* argument can have only the value kernel.

The *command_buffer* argument specifies a buffer that contains one or more CLI commands. You can include multiple commands in the same command buffer by separating them with newline characters.

The *output_buffer* and *error_buffer* arguments are used to capture output from commands and errors, respectively. The output buffers contain the accumulated output and error messages resulting from issuing all the commands in the command buffer.

C: The pointers pointed to by *output_buffer* and *error_buffer* are redirected to point to buffers containing the output and errors. The caller need not free the output and error buffers.

FORTRAN: Select a *maxsize* dimension for these buffers that is adequate to hold the expected output. Output that exceeds the specified size is lost.

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 invalid 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 28 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 28
Parameters and widgets

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 three highlight states.
oneshot	oneshot	Button to request single actions.
real	dial	Round dial with pointer; may be unbounded.
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.

AVScorout_event_wait

C

```
#include <time.h>
AVScorout_event_wait(nfds,readfds,writefds,exceptfds,timeout,
                     mask)
   int              nfds;
   fd_set           *readfds,*writefds,*exceptfds;
   struct timeval   *timeout;
   unsigned int     *mask;
```

FORTRAN

There is no FORTRAN equivalent for this routine.

This routine is used by a coroutine module that needs to simultaneously wait for data on one or more file descriptors or for its inputs and/or parameters to change. It can also be used by a module that does not have any file descriptors but waits for inputs and parameters to change using a timeout value.

In ConvexAVS, this routine uses the `select` system call. It allows the module to wait for all of the same events that `select` waits for and returns the same values that `select` returns. The only difference between this routine and `select` is that it takes an additional parameter that specifies the coroutine events to wait for. That parameter, *mask*, is the bitwise OR of two bits: `COROUT_WAIT` and `COROUT_EXEC`.

If `COROUT_WAIT` is set, `AVScorout_event_wait` returns when one of the module's inputs or parameters changes. If `COROUT_EXEC` is set, `AVScorout_event_wait` returns when downstream modules have finished executing.

The *timeout* parameter behaves just like `select`. If the value is 0, the routine blocks until either input or an error occurs. If the value points to a structure, the structure specifies the time in seconds and microseconds in which to wait. The timeval structure is defined in the time.h include file.

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, 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_mark_changed

C

```
AVScorout_mark_changed()
```

FORTRAN

```
AVSCOROUT_MARK_CHANGED()
```

This routine marks the module as having changed since the last call to `AVScorout_input`. The module will continue to be considered changed until the next call to `AVScorout_input` (or `AVScorout_output` for modules that have no inputs and parameters).

This routine can be used by coroutine modules that run continuously. It will cause the routine `AVScorout_wait` to return rather than wait for the next input or parameter to change.

AVScorout_output

C

```
AVScorout_output(output1, output2, ...)
  char          *output1, *output2, ...;
```

FORTRAN

```
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_set_sync

C

```
AVScorout_set_sync(value)
   int         value;
```

FORTRAN

```
AVSCOROUT_SET_SYNC(VALUE)
   INTEGER          VALUE
```

This routine allows you to force a coroutine module to execute synchronously. Set the *value* parameter as follows:

0 Execute asynchronously

1 Execute synchronously

A coroutine module can call this routine any time after it calls `AVScorout_init`. The module calls this routine only once.

By default, coroutine modules run asynchronously. This means that coroutine modules can run in parallel with other coroutine modules or other subroutine modules. However, modules that execute in parallel might cause the network to execute downstream modules more than once or have nondeterministic behavior. This is due to a lack of semaphores in the flow executive.

Coroutine modules can be run synchronously. To run synchronously means that except when the coroutine module is waiting in `AVScorout_wait`, `AVScorout_X_wait`, or `AVScorout_event_wait`, the flow executive executes no other module. This results in the network having a predictable order of execution.

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.

AVScorout_X_wait

C

```
#include <time.h>
AVScorout_X_wait(dpy, timeout, mask)
   Display        *dpy;
   struct timeval *timeout;
   int            *mask;
```

FORTRAN

There is no FORTRAN equivalent for this routine.

This routine is used by a coroutine module that needs to simultaneously wait for inputs and parameters to change and for X events.

The *dpy* argument is the X display on which X events are expected.

The *timeout* argument is a pointer to a timeval structure. This structure is defined in the time.h include file and has two fields: `tv_sec` and `tv_usec` that describe the number of seconds and microseconds to wait, respectively. If the pointer to the `timeval` structure is NULL, the routine will wait indefinitely. Otherwise, the `timeval` structure contains a number of seconds and a number of microseconds to wait before timing out. A structure containing 0 seconds and 0 microseconds can be used to poll the X socket and the state of ConvexAVS inputs and parameters.

The *mask* parameter is a pointer to an integer containing the coroutine events to wait for and is the bitwise OR of two bits: `COROUT_WAIT` and `COROUT_EXEC`.

If `COROUT_WAIT` is set, `AVScorout_X_wait` returns when one of the module's inputs or parameters changes. If `COROUT_EXEC` is set, `AVScorout_X_wait` returns when downstream modules have finished executing.

This routine will return a 1 if there are X events waiting to be processed. The flags set in *mask* will indicate the state of the coroutine events that were waited for.

If `AVScorout_X_wait` returns 0 and the value of *mask* returned is 0, then the routine timed out with the timeout value specified.

AVScreate_input_port

C

```
int AVScreate_input_port(port_name, type, flags)
  char      *port_name, *type;
  int       flags;
```

FORTRAN

```
AVSCREATE_INPUT_PORT(NAME, TYPE, FLAGS)
  CHARACTER*(*)     NAME, TYPE
  INTEGER           FLAGS
```

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 29.

Table 29
Input ports and data types

Data type	*type* string	Port color
Byte	"byte"	Light purple
Integer	"integer"	Light purple
Real	"real"	Dark purple
String	"string"	Green
Field	"field"	Multi color
Colormap	"colormap"	Yellow
Geometry	"geom"	Red
Pixel map	"pixmap"	Light blue
UCD structure	"ucd"	Orange

The *flags* argument can have the following values:

- REQUIRED
- OPTIONAL
- INVISIBLE

This routine returns an integer identifier that is used as an argument to other 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.

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

```
char * AVSdata_alloc(string, dims)
  char      *string;
  int       *dims;
```

FORTRAN

```
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);
```

AVSdata_free

C

```
#include <avs/field.h>
AVSdata_free(type, data_ptr)
  char      *type, *data_ptr;
```

FORTRAN

```
AVSDATA_FREE(TYPE, DATA_PTR)
  CHARACTER*n     TYPE
  INTEGER         DATA_PTR
```

This routine frees all memory associated with a data *type*. This *type* is the same string that was used in the `AVSdata_alloc` call to create the data. When `AVSdata_alloc` is used to create a field, the string includes the word "field" plus various field descriptors such as "2D uniform." When you are freeing the data, you should include only the string "field" without the other descriptors. The *data_ptr* is the pointer that the `AVSdata_alloc` call returned when the data structure was created.

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

```
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

```
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

```
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/field.h>
AVSfield_data_ptr(field)
  AVSfield  *field;
```

FORTRAN

```
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.

Note

This approach to obtaining the data pointer information is not portable across all platforms. Another approach is to use the `AVSfield_data_offset` routine.

AVSfield_free

C

```
#include <avs/field.h>
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/field.h>
AVSfield_get_dimensions(field, dimensions)
  AVSfield  *field;
  int       *dimensions;
```

FORTRAN

```
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_extent

C

```
#include <avs/field.h>
int AVSfield_get_extent(field, min_extent, max_extent)
   AVSfield  *field;
   float     *min_extent;
   float     *max_extent;
```

FORTRAN

```
INTEGER AVSFIELD_GET_EXTENT(FIELD, MIN_EXTENT,
                            MAX_EXTENT)
   INTEGER          FIELD
   REAL             MIN_EXTENT
   REAL             MAX_EXTENT
```

This routine allows you to obtain the extent of the field in `nspace`. The *min_extent* and *max_extent* parameters are arrays of dimension *field->nspace* and must be allocated by the caller. A return of 1 indicates valid data, while a 0 indicates invalid data.

AVSfield_get_int

C

```
AVSfield_get_int (field, selector)
  AVSfield  *field;
  int       selector;
```

FORTRAN

```
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
- AVS_FIELD_FLAGS

A return of a nonzero number indicates the value of the integer field specified by *selector*, while a 0 indicates an invalid *selector*.

AVSfield_get_label

C

```
#include <avs/field.h>
int AVSfield_get_label(field, number, label)
   AVSfield  *field;
   int       number;
   char      *label;
```

FORTRAN

```
INTEGER AVSFIELD_GET_LABEL(FIELD, NUMBER, LABEL)
   INTEGER           FIELD
   INTEGER           NUMBER
   CHARACTER*length  LABEL
```

This routine allows you to query the label for an individual component in the field. It is up to the caller to make sure that the allocated array is large enough to hold the label string. The label string can have a maximum size of AVS_FIELD_LABEL_LEN, which is 1024. A return of 1 indicates valid data while a 0 indicates invalid data.

AVSfield_get_labels

C

```
#include <avs/field.h>
int AVSfield_get_labels(field, labels, delimiter)
   AVSfield  *field;
   char      *labels;
   char      *delimiter;
```

FORTRAN

```
INTEGER AVSFIELD_GET_LABELS(FIELD, LABELS,
                            DELIMITER)
   INTEGER           FIELD
   CHARACTER*length  LABELS
   CHARACTER*length  DELIMITER
```

This routine allows you to query the labels for each component in the field. For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, and mach number. An example might be:

labels = `"temp;density;mach number"`
delimiter = `";"`

In turn, these labels would appear on the dials. It is up to the caller to make sure that the *labels* and *delimiter* arrays are long enough to contain the returned strings. These strings can be a maximum length of `AVS_FIELD_LABEL_LEN`, which is 1024. A return of 1 indicates valid data, while a 0 indicates invalid data.

AVSfield_get_minmax

C

```
#include <avs/field.h>
int AVSfield_get_minmax(field, min, max)
   AVSfield  *field;
   char      *min;
   char      *max;
```

FORTRAN

```
INTEGER AVSFIELD_GET_MINMAX(FIELD, MIN, MAX)
   INTEGER          FIELD
   <type>           MIN
   <type>           MAX
```

This routine allows you to obtain the range of the field data. The *min* and *max* are arrays of dimension *field->veclen* of the same type (`BYTE, REAL, INTEGER, DOUBLE`) as the computational data in the field. It is up to the caller to allocate enough space in these arrays to contain the returned information. A return of 1 indicates valid *min/max*, while a 0 indicates invalid *min/max*.

AVSfield_get_unit

C

```
#include <avs/field.h>
int AVSfield_get_unit(field, number, unit)
   AVSfield  *field;
   int       number;
   char      *unit;
```

FORTRAN

```
INTEGER AVSFIELD_GET_UNIT(FIELD, NUMBER, UNIT)
   INTEGER           FIELD
   INTEGER           NUMBER
   CHARACTER*length  UNIT
```

This routine allows you to query the unit string for an individual component in the field. It is up to the caller to allocate enough space in the *unit* array to contain the returned string. This string can be a maximum length of `AVS_FIELD_UNIT_LEN`, which is 1024. A return of 1 indicates valid data, while a 0 indicates invalid data.

AVSfield_get_units

C

```
#include <avs/field.h>
int AVSfield_get_units(field, units, delimiter)
   AVSfield  *field;
   char      *units;
   char      *delimiter;
```

FORTRAN

```
INTEGER AVSFIELD_GET_UNITS(FIELD, UNITS, DELIMITER)
   INTEGER           FIELD
   CHARACTER*length  UNITS
   CHARACTER*length  DELIMITER
```

This routine allows you to query the units for each component in the field. The unit label is associated with each vector element in the array of computational data. It is a character array with a delimiter character as the first character in the array. The delimiter is followed by string/delimiter pairs, the number of which is equal to the vector length of the field. The unit labels are useful for defining measurement units for each variable in the array of data. For instance, in the case of a CFD data set, you might want to specify components of the field as temperature, density, mach number. An example might be:

units = `"degrees C;g/cc;mach"`
delimiter = `";"`

It is up to the caller to allocate enough space in the *units* and *delimiter* arrays to contain the returned string. This string can be a maximum length of `AVS_FIELD_UNIT_LEN`, which is 1024. A return of 1 indicates valid data, while a 0 indicates invalid data.

AVSfield_invalid_minmax

C

```
#include <avs/field.h>
void AVSfield_invalid_minmax(field)
  AVSfield  *field;
```

FORTRAN

```
AVSFIELD_INVALID_MINMAX(FIELD)
  INTEGER          FIELD
```

This routine allows you to set the minimum/maximum range of the field data to be invalid. This function should be used after the field data has been changed by the module and you do not want the module to call the `AVSfield_reset_minmax` routine.

AVSfield_make_template

C

```
#include <avs/field.h>
AVSfield_make_template(field_in, template)
  AVSfield  *field_in, *template;
```

FORTRAN

There is no FORTRAN equivalent for this routine.

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.

Use this routine with 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));
```

AVSfield_points_offset

C

There is no C language equivalent for this routine.

FORTRAN

```
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/field.h>
AVSfield_points_ptr(field)
  AVSfield  *field;
```

FORTRAN

```
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.

AVSfield_reset_minmax

C

```
#include <avs/field.h>
void AVSfield_reset_minmax(field)
   AVSfield  *field;
```

FORTRAN

```
AVSFIELD_RESET_MINMAX(FIELD)
   INTEGER          FIELD
```

This routine computes the minimum and maximum values for the field's computational data and stores them in the field's data structure.

AVSfield_set_extent

C

```
#include <avs/field.h>
void AVSfield_set_extent(field, min_extent, max_extent)
   AVSfield  *field;
   float     *min_extent;
   float     *max_extent;
```

FORTRAN

```
AVSFIELD_SET_EXTENT(FIELD, MIN_EXTENT,
                    MAX_EXTENT)
   INTEGER          FIELD
   REAL             MIN_EXTENT
   REAL             MAX_EXTENT
```

This routine allows you to specify the extent of the field in `nspace`. *min_extent* and *max_extent* are arrays of dimension *field->nspace.*

AVSfield_set_int

C

```
#include <avs/field.h>
int AVSfield_set_int(field, selector, value)
   AVSfield  *field;
   int       selector;
   int       value;
```

FORTRAN

```
INTEGER AVSFIELD_GET_INT(FIELD, SELECTOR, VALUE)
   INTEGER          FIELD
   INTEGER          SELECTOR
   INTEGER          VALUE
```

This routine allows you to set one of the integer fields in the field structure. You can only perform this operation on a template. Otherwise, ConvexAVS issues an error message. Refer to the example in the file, /usr/avs/examples/colorizer_f.f for a module that uses this call.

The following *selectors* determine which structure element to change:

- AVS_FIELD_NDIM
- AVS_FIELD_NSPACE
- AVS_FIELD_VECLEN
- AVS_FIELD_TYPE
- AVS_FIELD_SIZE
- AVS_FIELD_UNIFORM
- AVS_FIELD_FLAGS

A return of a nonzero number indicates the value of the integer field specified by *selector*, while a 0 indicates an invalid *selector*.

AVSfield_set_labels

C

```
#include <avs/field.h>
void AVSfield_set_labels(field, labels, delimiter)
   AVSfield  *field;
   char      *labels;
   char      *delimiter;
```

FORTRAN

```
AVSFIELD_SET_LABELS(FIELD, LABELS, DELIMITER)
   INTEGER           FIELD
   CHARACTER*length  LABELS
   CHARACTER*length  DELIMITER
```

This routine allows you to set the labels for each component in the field. For instance, in the case of a CFD data set, you might want to label components of the field as temperature, density, mach number. In turn, these labels appear on the dials. An example might be:

```
labels = "temp;density;mach number"
delimiter = ";"
```

AVSfield_set_minmax

C

```
#include <avs/field.h>
void AVSfield_set_minmax(field, min, max)
   AVSfield  *field;
   <type>    *min;
   <type>    *max;
```

FORTRAN

```
AVSFIELD_SET_MINMAX(FIELD, MIN, MAX)
   INTEGER           FIELD
   <type>            MIN
   <type>            MAX
```

This routine allows you to set the range of the field data. The *min* and *max* parameters are arrays of dimension *field->veclen* of the same type (BYTE, REAL, INTEGER, DOUBLE) as the computational data in the field. They are initially declared as byte (that is, char) arrays in `AVSfield_set_minmax` for generality.

AVSfield_set_units

C

```
#include <avs/field.h>
void AVSfield_set_units(field, units, delimiter)
   AVSfield  *field;
   char      *units;
   char      *delimiter;
```

FORTRAN

```
AVSFIELD_SET_UNITS(FIELD, UNITS, DELIMITER)
   INTEGER           FIELD
   CHARACTER*length  UNITS
   CHARACTER*length  DELIMITER
```

This routine allows you to set the unit for each component in the field. The unit label is associated with each vector element in the array of computational data. It is a character array with a delimiter character as the first character in the array. The delimiter is followed by string/delimiter pairs, the number of which is equal to the vector length of the field. The unit labels are useful for defining measurement units for each variable in the array of data. For instance, in the case of a CFD data set, you might want to specify components of the field as temperature, density, mach number. An example might be:

```
units = "degrees C;mg/cc;mach"
delimiter = ";"
```

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;
```

FORTRAN

There is no FORTRAN equivalent for this routine. In FORTRAN, the module description function itself is called `AVSINIT_MODULES`.

`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.

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 flag 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, `AVSinput_changed` 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.

AVSload_byte

C

There is no C language equivalent for this routine.

FORTRAN

```
INTEGER AVSLOAD_BYTE(BASE, OFFSET)
   INTEGER          BASE, OFFSET
```

This routine loads a byte from memory. It is useful for compatibility with versions of FORTRAN that do not have a `BYTE` data type and do not allow `LOGICAL*1` to be used as a numeric value. This does not apply to CONVEX FORTRAN.

Refer to the example in /usr/avs/examples/colorizer_f.f for a module that uses this call.

AVSload_user_data_types

C

```
AVSload_user_data_types(filename)
   char        *filename;
```

FORTRAN

```
AVSLOAD_USER_DATA_TYPES(FILENAME)
   CHARACTER*n      FILENAME
```

This routine specifies a file name containing a description of one or more user-defined data types. It is called during the description function of the module. The file name is either an absolute path name of the file or, if a relative path name, the path is interpreted as relative to the directory /usr/avs/include. Refer to the program /usr/avs/examples/pick_cube.c for an example of using a user-defined data type for passing upstream data. Refer to the /usr/avs/examples/user_data.c and /usr/avs/examples/user_data_f.f programs for more general examples of using user-defined data.

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

```
char * AVSmessage(version, severity, module,
        function_name, 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

```
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.

AVSmessage_sub

C

There is no C language equivalent for this routine.

FORTRAN

```
AVSMESSAGE_SUB(ANSWER, VERSION, SEVERITY, MODULE,
                    FUNCTION_NAME, CHOICES,
                    MESSAGE)
  CHARACTER*length ANSWER
  CHARACTER*length VERSION
  INTEGER          SEVERITY
  CHARACTER*length MODULE, FUNCTION_NAME
  CHARACTER*length CHOICES, MESSAGE
```

This routine is a preferred alternative to AVSMESSAGE for FORTRAN modules that modify the *ANSWER* argument rather than returning it as a function result.

AVSmodify_float_parameter

C

```
AVSmodify_float_parameter (param_name, flags, init,
                           minval, maxval)
  char    *param_name;
  int     flags;
  double  init, minval, maxval;
```

FORTRAN

There is no FORTRAN equivalent for this routine. Use AVSMODIFY_PARAMETER instead.

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.

AVSmodify_parameter

C

```
AVSmodify_parameter(param_name, flags, init, minval,
                                                maxval)
  char      *param_name;
  int       flags, init, minval, maxval;
```

FORTRAN

```
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.

AVSmodify_parameter_prop

C

```
AVSmodify_parameter_prop(name, prop_name, prop_type,
                         prop_value)
  char          *name, *prop_name, *prop_type;
  int           prop_value;
```

FORTRAN

```
AVSMODIFY_PARAMETER_PROP(NAME, PROP_NAME,
                         PROP_TYPE, PROP_VALUE)
  CHARACTER*n   NAME, PROP_NAME, PROP_TYPE
  INTEGER       PROP_VALUE
```

This routine modifies a parameter property during computation. The *prop_value* argument is treated like the corresponding argument in `AVSadd_parameter_prop`. Unlike that function, this one takes the parameter name because it can be used only by the compute function. The widget attached to the parameter may change immediately as in the case of changing the title property of a parameter. The property does not have to have been created by `AVSadd_parameter_prop` first.

AVSmodule_from_desc

C

```
AVSmodule_from_desc(desc)
  int           (*desc)();
```

FORTRAN

There is no FORTRAN equivalent for this routine. In FORTRAN, the module description function is called `AVSINIT_MODULES`.

`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.

AVSmodule_status

C

```
AVSmodule_status(comment, percent)
   char      *comment;
   int       percent;
```

FORTRAN

```
AVSMODULE_STATUS(COMMENT, PERCENT)
   CHARACTER*n      COMMENT
   INTEGER          PERCENT
```

This routine sends the kernel status updates when a long operation is in progress and is of predictable length. The information may be displayed in the status bar on the main control panel to inform you of incremental progress. A module's status is considered to be broken into input transmission (0 - 10 percent), module operation (10-90 percent) and output processing (90-100 percent). The status *percent* argument is given in terms of 0-100 percent of the module operation and thus shows up as changes between 10 and 90 percent of the overall operation. The *comment* in the status bar for particularly long operations shows the intermediate operation in progress. For shorter operations, only the module name might actually appear. If no status calls are made, the status bar does not show any intermediate progress between the 10 and 90 percent mark.

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.

AVSparameter_visible

C

```
AVSparameter_visible(name, stat)
   char        *name;
   int         stat;
```

FORTRAN

```
AVSPARAMETER_VISIBLE(NAME, STAT)
   CHARACTER*n     NAME
   INTEGER         STAT
```

This routine controls the visibility of the widget attached to a parameter. The *name* argument is the string name of the parameter, and the *stat* value is 0 for invisible and 1 for visible.

AVSport_field

C

There is no C language equivalent for this routine.

FORTRAN

```
INTEGER AVSPORT_FIELD(PORT_NAME)
   CHARACTER*n     PORT_NAME
```

This routine returns the field pointer required by the new field accessor functions. The integer value returned by the function is the associated field pointer or 0 if there is no valid field data associated with that port. When fields are passed as single arguments, the field pointer is passed directly as an argument to the computation function.

AVSptr_alloc

C

There is no C language equivalent for this routine.

FORTRAN

```
INTEGER AVSPTR_ALLOC(NAME, NELEM, ELSIZE, CLEAN,
                     BASEVEC, ADDR, OFFSET)
  INTEGER            NELEM, ELSIZE, CLEAN, OFFSET,
                     ADDR
  DIMENSION          BASEVEC(1)
  CHARACTER*(*)      NAME
```

`AVSptr_alloc` allocates a new data block for the given pointer. The parameters are as follows:

NAME The name of the output port that the pointer belongs to. Use a name consisting of a single SPACE character if the data block is not associated with an output port.

NELEM The number of array elements to allocate.

ELSIZE The element size in bytes (`INTEGER*4` is 4, `REAL*8` is 8, and so on.).

CLEAN If 1, the new elements are initialized to 0. Otherwise they are not initialized.

BASEVEC The start of the local array, used as a reference location. This array can be dimensioned with 1 element.

ADDR The data block memory pointer. Initialize this to 0 if the data block is being used for a local array instead of for an output port.

OFFSET The offset index relative to the *BASEVEC* array that corresponds to the first element of the data block pointed to by *ADDR*.

If *ADDR* points to an existing data block, that block is first freed and then a new block is allocated. If the requested memory cannot be allocated, a value of 0 is returned.

The sample program in /usr/avs/examples/colorizer_f.f provides an example of using this routine.

AVSptr_offset

C

There is no C language equivalent for this routine.

FORTRAN

```
INTEGER AVSPTR_OFFSET(NAME, ELSIZE, BASEVEC, ADDR,
                      OFFSET)
  INTEGER             ELSIZE, OFFSET, ADDR
  DIMENSION           BASEVEC(1)
  CHARACTER*(*)       NAME
```

`AVSptr_offset` gets the offset index for an existing data block without reallocating it. The parameters are the same as for `AVSptr_alloc`. If *ADDR* is 0 (no space allocation), a value of 0 is returned. Otherwise a value of 1 is returned.

Once the *OFFSET* index is returned, there are several ways that it can be used, depending on the circumstances:

- For 1D arrays, add the *OFFSET* value to all of the local reference array, as in *BASEVEC(OFFSET+i)*. The example module in /usr/avs/examples/read_image_f.f uses this approach.
- For multi-dimensional arrays, a statement function can be used to perform the index arithmetic. The example module in /usr/avs/examples/threshold_f.f uses this approach.
- A more convenient approach to handling arrays of any dimension is to pass the *OFFSET* element of the local reference array to a second function that is expecting an array. This effectively dereferences the pointer and allows you to directly reference array elements. The example module in /usr/avs/examples/test_field_f.f uses this approach.

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_input_class

C

```
AVSset_input_class(port, class)
   int        port;
   char       *class;
```

FORTRAN

```
AVSSET_INPUT_CLASS(PORT, CLASS)
   INTEGER          PORT
   CHARACTER*n      CLASS
```

This routine sets the port class for an input port. The class for a port or parameter is used to make automatic upstream connections when particular downstream connections are made.

The *port* argument should be the integer value returned from `AVScreate_input_port` for `AVSset_input_class`.

The *class* argument is a character string that contains a class name and an optional port name to associate it with. The class name is determined by convention between the upstream and downstream module. This name is often the name of the data type of the downstream connection.

An optional port name can be specified as part of the class character string. If so, a ":" character separates the port name from the class name. If the port name is specified, it indicates that the upstream connection should only be made if the downstream port is being connected.

AVSset_module_flags

C

There is no C language equivalent for this routine.

FORTRAN

```
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

```
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 located in Table 30.

Table 30
Module types and names

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.

AVSset_output_class

C

```
AVSset_output_class(port, class)
   int         port;
   char        *class;
```

FORTRAN

```
AVSSET_OUTPUT_CLASS(PORT, CLASS)
   INTEGER          PORT
   CHARACTER*n      CLASS
```

This routine sets the port class for an output port. The class for a port or parameter is used to make automatic upstream connections when particular downstream connections are made.

The *port* argument should be the integer value returned from `AVScreate_output_port` for `AVSset_output_class`.

The *class* argument is a character string that contains a class name and an optional port name to associate it with. The class name is determined by convention between the upstream and downstream module. This name is often the name of the data type of the downstream connection.

An optional port name can be specified as part of the class character string. If so, a ":" character separates the port name from the class name. If the port name is specified, it indicates that the upstream connection should only be made if the downstream port is being connected.

AVSset_output_flags

C

```
int AVSset_output_flags(port, flags)
   int         port, flags;
```

FORTRAN

```
INTEGER AVSSET_OUTPUT_FLAGS(PORT, FLAGS)
   INTEGER          PORT, FLAGS
```

This routine is used by a module in the description function to set some optional properties of an output port. The INVISIBLE flag is currently supported. It causes the output to be invisible by default.

AVSset_parameter_class

C

```
AVSset_parameter_class(port, class)
   int         port;
   char        *class;
```

FORTRAN

```
AVSSET_PARAMETER_CLASS(PORT, CLASS)
   INTEGER          PORT
   CHARACTER*n      CLASS
```

This routine sets the port class for a parameter port. The class for a port or parameter is used to make automatic upstream connections when particular downstream connections are made.

The *port* argument should be the integer value returned from `AVSadd_parameter` for `AVSset_parameter_class`.

The *class* argument is a character string that contains a class name and an optional port name to associate it with. The class name is determined by convention between the upstream and downstream module. This name is often the name of the data type of the downstream connection.

An optional port name can be specified as part of the class character string. If so, a ":" character separates the port name from the class name. If the port name is specified, it indicates that the upstream connection should only be made if the downstream port is being connected.

AVSstore_byte

C

There is no C language equivalent for this routine.

FORTRAN

```
AVSSTORE_BYTE(BASE, OFFSET, VALUE)
   INTEGER          BASE, OFFSET, VALUE
```

This routine stores a byte into memory. It is useful for compatibility with versions of FORTRAN that do not have a `BYTE` data type and do not allow `LOGICAL*1` to be used as a numeric value. This does not apply to CONVEX FORTRAN.

Refer to the example in /usr/avs/examples/colorizer_f.f for a module that uses this call.

AVSudata_get_double

C

```
#include <avs/udata.h>
int AVSudata_get_double(ptr, name, value, value_elements)
   char      *ptr;
   char      *name;
   double    *value;
   int       value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_GET_DOUBLE(PTR, NAME, VALUE,
                            VALUE_ELEMENTS)
   INTEGER          PTR, VALUE_ELEMENTS
   CHARACTER*(*)    NAME
   REAL*8           VALUE
   DIMENSION        VALUE(VALUE_ELEMENTS)
```

This routine allows you to retrieve a double precision value named *name* from a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a double precision field, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure elements directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. Refer to the /usr/avs/examples/user_data_f.f file for an example FORTRAN module. A return of 1 indicates success, while a 0 indicates failure.

AVSudata_get_int

C

```
#include <avs/udata.h>
int AVSudata_get_int(ptr, name, value, value_elements)
   char        *ptr;
   char        *name;
   int         *value;
   int         value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_GET_INT(PTR, NAME, VALUE,
                         VALUE_ELEMENTS)
   INTEGER           PTR, VALUE_ELEMENTS
   CHARACTER*(*)     NAME
   INTEGER           VALUE
   DIMENSION         VALUE(VALUE_ELEMENTS)
```

This routine allows you to retrieve an integer value named *name* from a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not an integer, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure fields directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. A return of 1 indicates success, while a 0 indicates failure.

AVSudata_get_real

C

```
#include <avs/udata.h>
int AVSudata_get_real(ptr, name, value, value_elements)
   char      *ptr;
   char      *name;
   float     *value;
   int       value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_GET_REAL(PTR, NAME, VALUE,
                          VALUE_ELEMENTS)
   INTEGER          PTR, VALUE_ELEMENTS
   CHARACTER*(*)    NAME
   REAL             VALUE
   DIMENSION        VALUE(VALUE_ELEMENTS)
```

This routine allows you to retrieve a floating point value named *name* from a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a floating point, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure fields directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. A return of 1 indicates success, while a 0 indicates failure.

AVSudata_get_string

C

```
#include <avs/udata.h>
int AVSudata_get_string(ptr, name, value, value_elements)
   char      *ptr;
   char      *name;
   char      *value;
   int       value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_GET_STRING(PTR, NAME, VALUE,
                            VALUE_ELEMENTS)
   INTEGER           PTR, VALUE_ELEMENTS
   CHARACTER*(*)     NAME
   CHARACTER         VALUE
   DIMENSION         VALUE(VALUE_ELEMENTS)
```

This routine allows you to retrieve a string value named *name* from a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a string, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure fields directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. A return of 1 indicates success, while a 0 indicates failure.

AVSudata_set_double

C

```
#include <avs/udata.h>
int AVSudata_set_double(ptr, name, value, value_elements)
   char      *ptr;
   char      *name;
   double    *value;
   int       value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_SET_DOUBLE(PTR, NAME, VALUE,
                            VALUE_ELEMENTS)
   INTEGER          PTR, VALUE_ELEMENTS
   CHARACTER        NAME(n)
   REAL*8           VALUE
   DIMENSION        VALUE(VALUE_ELEMENTS)
```

This routine allows you to store a double precision value named *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a double precision field, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure fields directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. A return of 1 indicates success, while a 0 indicates failure.

AVSudata_set_int

C

```
#include <avs/udata.h>
int AVSudata_set_int(ptr, name, value, value_elements)
   char      *ptr;
   char      *name;
   int       *value;
   int       value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_SET_INT(PTR, NAME, VALUE,
                         VALUE_ELEMENTS)
   INTEGER           PTR, VALUE_ELEMENTS
   CHARACTER*(*)     NAME
   INTEGER           VALUE
   DIMENSION         VALUE(VALUE_ELEMENTS)
```

This routine allows you to store an integer value named *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not an integer, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure fields directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. A return of 1 indicates success, while a 0 indicates failure.

AVSudata_set_real

C

```
#include <avs/udata.h>
int AVSudata_set_real(ptr, name, value, value_elements)
   char        *ptr;
   char        *name;
   float       *value;
   int         value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_SET_REAL(PTR, NAME, VALUE,
                          VALUE_ELEMENTS)
   INTEGER              PTR, VALUE_ELEMENTS
   CHARACTER*(*)        NAME
   REAL                 VALUE
   DIMENSION            VALUE(VALUE_ELEMENTS)
```

This routine allows you to store a floating point value named *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not floating point, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure fields directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. A return of 1 indicates success, while a 0 indicates failure.

AVSudata_set_string

C

```
#include <avs/udata.h>
int AVSudata_set_string(ptr, name, value, value_elements)
   char        *ptr;
   char        *name;
   char        *value;
   int         value_elements;
```

FORTRAN

```
INTEGER AVSUDATA_SET_STRING(PTR, NAME, VALUE,
                            VALUE_ELEMENTS)
   INTEGER             PTR, VALUE_ELEMENTS
   CHARACTER*(*)       NAME
   CHARACTER           VALUE
   DIMENSION           VALUE(VALUE_ELEMENTS)
```

This routine allows you to store a string value named *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a string, then the function returns an error. This function is intended primarily for FORTRAN modules because the C programming language can access the structure fields directly. You must call the function `AVSload_user_data_types` in the module description function to describe the user data type. A return of 1 indicates success, while a 0 indicates failure.

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.

Advanced module topics 16

Coroutine synchronization

Coroutine modules have the ability to execute asynchronously from the ConvexAVS flow executive. This means that at any time, coroutine modules can call `AVScorout_input` to acquire their input values and `AVScorout_output` to send output. This provides coroutine modules with more flexibility than subroutine modules. For example, they can manage their own input sources (such as X events or keyboard input), run in parallel with other ConvexAVS modules, and schedule execution themselves rather than waiting for user interaction. Coroutine modules can execute continuously, wait for upstream input, or wait for you to change a parameter.

In order to perform these tasks effectively, coroutine modules must be able to communicate with ConvexAVS to determine when they should run. For example, if a coroutine module is in a tight loop calling `AVScorout_output`, the ConvexAVS kernel reads the data as fast as possible. If the coroutine module executes quickly enough (or the network takes long enough), some of the data produced by the coroutine may not be processed by the network.

To avoid this problem, also known as *overrun*, more synchronization with the flow executive is necessary. You can use the `AVScorout_wait` routine to facilitate flow executive scheduling of the module. This routine allows the flow executive to execute the module when it is the next changed module in the run queue. A changed module is one whose inputs or parameters have been modified or one that has been marked as changed by the `AVScorout_mark_changed` routine.

The flow executive executes the module when the following conditions are true:

- The flow executive is enabled.
- The module is the next changed module in the run queue.

The AVScorout_mark_changed routine is useful if you want to implement a continuously running module. When a module calls this routine, the flow executive marks the module as being in a changed state. ConvexAVS continues to consider this module as changed until the next call to AVScorout_input (or AVScorout_output if the module has neither inputs nor parameters). This causes the AVScorout_wait to return immediately rather than wait for input or parameter changes.

When using AVScorout_mark_changed, you should call AVScorout_input to determine when inputs and parameters change as AVScorout_wait is not responding to these events.

Another way to schedule the execution of a coroutine module with the flow executive is with the AVScorout_exec routine. Calling this routine causes module execution to wait until the flow executive stops running before returning. This is useful when you want to delay module execution until the network has completed its processing.

Coroutine scheduling with X

The AVScorout_wait routine does not allow you to schedule module execution with X events. To do this, use the AVScorout_X_wait routine. When called, this routine waits for both input or parameter changes and X events or errors. AVScorout_X_wait also allows you to set a time interval that determine how long the routine waits before returning. The ability to set a timeout interval is useful when implementing features like *double-click* mouse action, for example. Setting the timeout interval to zero makes this routine useful for polling the X server and to determine the status of module input or parameter changes. Remember to add the time.h include file when using this routine.

Coroutine scheduling with other devices

ConvexAVS provides a more general mechanism by which a coroutine module can schedule with other file descriptor type devices. You can use the AVScorout_event_wait routine to wait for data from one of the specified file descriptor devices or to determine if module inputs or parameters have changed. If no descriptors are of interest, you can still use this routine to wait for input or parameter changes within the confines of the specified time interval by specifying the descriptor arguments as zero pointers.

In ConvexAVS, this routine uses the `select` system call. It allows the module to wait for all of the same events that `select` waits for and returns the same values that `select` returns. The only difference between this routine and `select` is that it takes an additional parameter that specifies the coroutine events to wait for. That parameter, *mask*, is the bitwise OR of two bits: `COROUT_WAIT` and `COROUT_EXEC`.

If `COROUT_WAIT` is set, `AVScorout_event_wait` returns when one of the module's inputs or parameters changes. If `COROUT_EXEC` is set, `AVScorout_event_wait` returns when downstream modules have finished executing.

Remember to add the time.h include file when using this routine.

Synchronous execution

When running modules asynchronously, the flow executive does not wait before scheduling any other modules that are ready to run. If there are no other modules that are ready to execute, this behavior does not cause problems. However, if there are other modules waiting to execute, problems may ensue because the two processes may share data. If this is the case, modules might execute twice or in an unpredictable way.

To prevent this, you can run your coroutine module synchronously with the flow executive. When a coroutine module runs synchronously, ConvexAVS assumes that as long as it is not waiting in `AVScorout_wait`, `AVScorout_event_wait`, or `AVScorout_X_wait`, then it is running. Therefore, when the coroutine module is executing, no other modules are allowed to run.

To make your coroutine run synchronously, use the `AVScorout_set_sync` routine:

```
AVScorout_set_sync(value)
int value;
```

where *value* is 0 or 1. A value of 1 makes the coroutine run synchronously, a value of 0 makes it asynchronously. The coroutine can toggle its synchronous state during execution. The effects take place during the next call to `AVScorout_wait`.

Upstream data

Upstream data refers to the process of sending information from one module to another module that precedes it in the flow network. ConvexAVS uses the upstream data mechanism to communicate information about direct manipulation of geometry (such as rotating an object with the mouse) to modules in the network that need this information to recalculate their contribution to the displayed image.

While ConvexAVS defines special data types to facilitate the use of upstream data, you can define your own data types for this purpose.

Upstream data mechanism

In certain situations, modules need to receive information about events that occur after their own execution has completed. Examples of this feedback are the following:

- The **arbitrary slicer** module that produces a geometry object and needs to get information when you transform the slice plane with the Geometry Viewer so that it can regenerate the slice at the new location.
- A module defining a molecule that needs information about which bond you have selected so the bond can be highlighted in the image.
- The **probe** module that has a mode where each time you pick an object, it snaps the probe to a vertex on the object selected.

Feedback is implemented through the data flow mechanism. A downstream module must have an additional output port from which to send the data and the upstream module must have an additional input port on which to receive the data. In addition, both input and output ports must be defined as using the same data type.

The following hypothetical example of a **molecule** module uses upstream data:

1. The upstream module **molecule** executes and outputs geometry data to the **render geometry** module.
2. The **render geometry** module executes, marking its upstream output port as unchanged. This system is now idle.
3. You select a chemical bond. This causes the **render geometry** module to pass data on its upstream output port to the **molecule** module's input port.

4. The **molecule** module executes in response to the upstream data change and highlights the selected bond. It outputs a new geometry.
5. The **render geometry** module executes again, this time marking its upstream port as unchanged.

You have created a loop in the network. You must be careful when constructing loops. For example, a downstream module outputs data on its upstream connection each time it executes. Then the upstream module executes and outputs data to the downstream module. You have just constructed an infinite loop in the network.

Note

ConvexAVS assumes all output data changes after each invocation of a module.

In the example just discussed, the geometry module marks its upstream output port as unchanged (using the `AVSmark_output_unchanged` routine) when it executes in response to input from *upstream* in the network. However, when your interaction requires a change to the display (that is, changes occur *downstream* of the geometry module), the geometry module activates its upstream output port.

Implementing upstream data

In ConvexAVS, the only modules that can send upstream data are **render geometry** and **display tracker**. They support the following operations:

- Sending transformation information upstream (rotation, translation, and scaling information)
- Sending information on the selection of a particular object (that is, picking the object)

These two modules handle each of these cases using separate data types.

Transformation information

The **render geometry** and **display tracker** modules can send upstream transformation data any time they transform an object. The modules transmit the data structure shown in Figure 140.

Figure 140
Module selection information

```
typedef struct _upstream_transform {
   int flags;                    /* Button state */
   float msxform[4][4];          /* Transformation matrix */
   char object_name[256];        /* Current object */
   int camera_index;             /* View transformation */
   int x, y;                     /* Button x, y */
   int width, height;            /* Window width, height */
   } upstream_transform;
```

The parameters are described as follows:

flags — The current button state that existed when the transformation occurred. In this case, the transformation is generated by you rotating the object with the mouse. Possible values are: BUTTON_DOWN, BUTTON_UP or BUTTON_MOVING.

msxform — The object's current transformation matrix. This matrix transforms the vertices of the object from the modeling coordinate system (in which they are defined) to the coordinate system of the object's parent.

object_name — The name of the object whose transformation information is being passed upstream. The object name may have a suffix appended to it (for example, arbitrary slice may appear as arbitrary slice.1).

camera_index — The view number of the window in which the transformation was generated.

x, y — The x-, y-position of the cursor in pixels (or -1 if the transformation was not generated by the mouse).

width, height — The width and height of the window in which the transformation was generated, if it was generated by the mouse.

Keep in mind that this data structure is transmitted from the downstream module to an upstream module. Once received, this data structure is available to the upstream module.

A geometry object can have two modes associated with it that determine how this mechanism operates: *notify* and *redirect*. In both cases, ConvexAVS passes the same information upstream. The two modes differ with respect to how the Geometry Viewer treats the object.

Notify mode

In notify mode, the object is treated as any other object in the scene. When the transformation matrix is changed, the geometry for that object, and all child objects, are transformed and rendered. The downstream module then transmits the `upstream_transform` structure to the upstream module.

Notify mode is useful for cases where you want the upstream module to be informed of changes in the object's position or orientation but do not want it to regenerate the geometry (it is being regenerated anyway by the downstream module). If the upstream module modifies the geometry, its output causes the downstream module to refresh the scene again.

ConvexAVS uses notify mode to implement the **probe** module. As the probe is transformed, the module obtains the data probe's transformation matrix. The module then has the opportunity to update the values displayed by the probe according to the new position of the probe. Because the module uses notify mode, it does not have to update the position of the probe itself because this is handled by the Geometry Viewer.

Redirect mode

In redirect mode, the transformation matrix is maintained like it normally is for the object, but this transformation matrix is not used to render the object or any children of the object. However, the downstream module still transmits `upstream_transform` to the upstream module (which may send new output to the downstream module, causing it to render the object).

Redirect mode is useful when you know that the upstream module needs to regenerate the geometry in order for the display to be correct. The output of the upstream module then causes the downstream module to execute. If the upstream module is going to regenerate the geometry, using redirect mode provides better performance because the scene is regenerated only once.

When a user–defined upstream module receives upstream data from the **render geometry** or **display tracker** module, the designer of the upstream module has the flexibility to use or ignore any of the data received. Refer to the example in /usr/avs/example/pick_cube.c for sample code that uses upstream data.

The following is an example of how to use upstream data to send transformation information from the **render geometry** module to your upstream module:

- Add an input port to your module having the data type `struct upstream_transform`. This is a user–defined data type that is defined in the udata.h include file.
- Your module should have a geometry type output that is connected to the **render geometry** module.
- When constructing your edit list, you should use the routine `GEOMedit_transform_mode`:

```
GEOMedit_transform_mode(edit_list,object_name,mode,flags)
  GEOMedit_list     edit_list;
  char              *object_name;
  char              *mode;
  int               flags;
```

mode should have one of the following values:

- notify
- redirect
- normal
- parent

The *flags* argument should contain at least one of the following flags: BUTTON_DOWN, BUTTON_UP and BUTTON_MOVING. The flags indicate for a given mouse button state when transforms should be sent to the upstream module. For normal usage, you should specify (BUTTON_MOVING | BUTTON_UP). Because BUTTON_DOWN does not cause a change in the transformation matrix (but simply starts off a transform), specifying it causes a needless execution of the module.

The *object_name* can be either the name of an object that the upstream module produced, the name of an object that another module produced, or one that you read in directly from the Geometry Viewer.

Setting the transform mode of an object to either *notify* or *redirect* causes the **render geometry** and **display tracker** modules to output data from the Transform Info output port. If there is a connection from this port to the input port of type `struct upstream_transform` on an upstream module, that module executes when the object is changed. If you do not add automatic connection support in the module description function, you must make the connection by hand. Refer to the example in /usr/avs/examples/pick_cube.c for information on invisible ports.

Any subsequent call to `GEOMedit_transform_mode` for a particular object overrides the previous call. This means there can be only one requestor for the transformations of a particular object.

Selection information

You can cause a module to execute by sending selection information to an input port anytime you pick an object. Selection information includes:

- Name of the selected object
- Nearest vertex selected
- User–defined data associated with the nearest vertex
- User–defined data associated with the particular primitive selected (line, polygon, or sphere)
- Coordinates of the 3D point that were selected

The 3D point selected by ConvexAVS is the intersection between a ray projected from the pick point directly into the screen and the first encountered piece of geometry. The coordinates of the selected point is provided in several different coordinate systems, as well as the transformation matrices for the selected object and the view that contains the selected object.

ConvexAVS defines the structure in Figure 141 to contain this information.

Figure 141
Module selection information

```
typedef struct _upstream_geom {
  int flags;                  /* Button state, and selection mode */
  char current_obj[256];      /* Object whose selection mode set (coords are in */
                              /* the coordinate system of this object but */
                              /* vertices are defined for "picked obj" */
  float mscoord[3];           /* Modeling space coordinates */
  float wscoord[3];           /* World space coordinates */
  float sscoord[3];           /* Screen space coordinates */
  float objxform[4][4];       /* Object's coordinate matrix */
  float worldxform[4][4];     /* From modelling space to world space */
  float viewxform[4][4];      /* From world space to screen space */

  int x, y;                   /* Button x, y */
  int width, height;          /* window width, height */
  int camera_index;           /* Index of the view selection was made */

  char picked_obj[256];       /* Name of object whose vertex was picked */
  float vertex[3];            /* nearest vertex to selection */
  int vdata;                  /* per-vertex data -- user specified */
  int odata;                  /* per-object (line, poly, sphere) data */
} upstream_geom;
```

The parameters are described as follows:

flags	One of BUTTON_DOWN, BUTTON_UP or BUTTON_MOVING
current_obj	The name of the object selected.
mscoord	The coordinates of the 3D selection point in modeling coordinates (the coordinate system in which the object's vertices are defined).
wscoord	The coordinates of the 3D selected point in world coordinates (lighting is performed in the world coordinate system).
sscoord	The coordinates of the 3D selected point in screen coordinates (the screen coordinate system ranges from -1 to 1 with (-1,-1,-1) being the lower left hand furthest corner of the window and (1,1,1) being the upper right hand closest corner).
objxform	The current transformation matrix of the selected object.

worldxform	The matrix that transforms the current object from modeling coordinates to world coordinates.
viewxform	The matrix that transforms the current object from modeling coordinates to screen coordinates.
x, y	The x–, y–coordinates, in pixels, of the selection point.
width, height	The width and height, in pixels, of the view in which the selection was made.
camera_index	The view number of the window in which the transformation was generated.
picked_obj	The name of the object whose direct geometry was selected. This can either be *current_obj* or a descendant of *current_obj*.
vertex	The X–, Y–, and Z–values of the selected vertex. This is a vertex contained in the geometry of *picked_obj*.
vdata	If *picked_obj* had any vertex data associated with the selected vertex, this 32–bit integer is stored in this member. If there is no vertex data, this member is -1.
odata	If *picked_obj* has any primitive or object data associated with the primitive that was selected, this member contains that data. Otherwise this field has the value -1.

Rules for picking objects

When you select (or pick) an object, the **render geometry** module receives a list of selected objects. The first item in the list is the object that contained the picked geometry. The next item in the list is the parent of the picked object, then the parent of that object, and so on. The last item in the list is the top object.

Note that the top object is always picked regardless of where you make the selection (it is even put in the list when there is no picked object).

There might be multiple objects in the list that are requesting selection (that is, that need to be highlighted by the module owning the object). However, only a single object is reported as picked.

The algorithm for choosing which object is picked is:

1. If any object in the list is the current object, it has priority over all other objects, and the selection information is sent to the module that requested a pick on this object.
2. If the current object is not in the list, the first object in the list that has been selected is picked, and this object becomes the current object.

If you press the **SHIFT** key when the you make a selection, the **render geometry** module changes only the current object and does not process any selections.

If you do not pick any geometry, the top level object becomes the current selection. If you pick the top level object, ConvexAVS returns valid data only in the following members of the `upstream_geom` data structure: *current_obj*, *x*, *y*, *width*, *height*, *objxform*, *wsxform*, *viewxform*, and *picked_obj*.

In this case, *picked_obj* is a zero-length string.

The following example shows how to use upstream data to receive selection information at your upstream module:

- Add an input port to your module using the data type `struct upstream_geom`. This is a user–defined data type that is defined in the udata.h include file.
- Your module should have a geometry type output that is connected to the **render geometry** module.
- When constructing your edit list, you should use the routine `GEOMedit_selection_mode`:

```
GEOMedit_selection_mode(edit_list,object_name,mode,flags)
  GEOMedit_list     edit_list;
  char              *object_name;
  char              *mode;
  int               flags;
```

mode should have one of the following values:

- notify
- normal
- ignore

The *flags* argument should contain one or more of the following flags: BUTTON_DOWN, BUTTON_UP and BUTTON_MOVING. The flags indicate for a given mouse button state when picks should be sent to your module. For example, if you specify only BUTTON_DOWN, you only get picks when the button is pressed. If you specify BUTTON_DOWN and BUTTON_MOVING, you get picks each time the button is pressed and subsequently each time that the cursor moves until the button is released.

The Geometry Viewer does not process BUTTON_MOVING and BUTTON_RELEASE selections if the object picked on the BUTTON_DOWN did not have any module requesting it.

The object name can be either the name of an object that you produced, the name of an object that another module produced, or one that was read in by you directly from the Geometry Viewer.

Setting the selection mode of an object to *notify* causes the **render geometry** module to output data from the Geometric Info output port. If there is a connection from this port to the input port of type `struct upstream_geom` on your module, it executes when the object is selected. If you do not add automatic connection support in your module description function, you must make this connection by hand. Refer to the example in /usr/avs/examples/pick_cube.c for information on invisible ports.

User–defined upstream data

While ConvexAVS supports only two modules capable of outputting upstream data, you can develop you own modules with this capability. You are not limited to passing information on geometric transformations and on picking. As long as you specify the input and output ports correctly and ensure that both upstream and downstream modules recognize the data structures that you pass between them, you can build upstream data capabilities into any module you design.

You can define input and output ports to use any ConvexAVS data type. You can also configure ports to use any data type that you create with the user–defined capabilities of ConvexAVS. Once you define the desired data type, you can set up the port connections in the upstream and downstream module description functions.

Automatic port connection

To simplify the network building process, ConvexAVS can hide certain types of connections. This section describes a mechanism by which you can instruct the flow executive to automatically make an upstream connection when you make a downstream connection. You can make ports optionally visible when the port is not required for module execution. You can make ports with required connections invisible also, but this is not recommended.

Port classes

Data in a network can flow both downstream and upstream. It is often the case that every upstream connection is associated with a particular downstream connection in order to pass back data about an action to the module that produced the particular object.

A *class* attribute is associated with both input and output ports. A class attribute is a character string name that contains two fields:

- Port name (optional)
- Port type specification (required)

The port type specification is an arbitrary string that is meaningful to both upstream and downstream modules.

When the flow executive makes a connection between two modules, it looks for a match between the input ports of the upstream module and the output ports of the downstream module. If it finds a match, it makes this upstream connection.

A successful match occurs when the type of the input port class matches the type of the output port class. If a module has optionally specified a port name for the class, the match is made only if the port name specified is the name of the port being connected.

For example, the **probe** module defines a port class of type `upstream_transform` for its input port. The **render geometry** module defines a port class of type `upstream_transform` for its output port. When the **probe** module is connected to the **render geometry** module, the input port of the **probe** module is automatically connected to the output port of the **render geometry** module because the class matches.

Here is the code fragment that implements the appropriate part of the **probe** module's description function.

The data type of the input port is a user-defined data structure:

```
int port;
   .
   .
   .
port=AVScreate_input_port("Transform Info",
         "struct upstream_transform",
         OPTIONAL|INVISIBLE);
AVSset_input_class(port,"upstream_transform");
   .
   .
   .
```

Here is the code fragment that creates a compatible port for the output port on the **render geometry** module:

```
port=AVScreate_output_port("Transform Info",
   "struct upstream_transform");
AVSset_output_flags(port,INVISIBLE);
AVSset_output_class(port,"upstream_transform");
```

The correspondence between the class type (`upstream_transform`) and the data type of the port (`struct upstream_transform`) is a convention, not a requirement.

Automatic connections are automatically disconnected when the connection that caused their creation is broken. There can only be a single class for a port. Automatic connections are not saved in a network but are recreated when the network is read in, if the classes defined in the modules have not changed.

Port visibility

You can make a module's input and output ports invisible so that colored boxes do not appear on the module icon. The default visibility of a port is assigned through the module description function by setting the port flag INVISIBLE. For input ports, this flag is specified with the routine `AVScreate_input_port` in the module description function. An example of this call is:

```
port = AVScreate_input_port("obscure port" ,
   "integer" , INVISIBLE | OPTIONAL);
```

Unlike `AVScreate_input_port`, the routine `AVScreate_output_port` does not have a flags field. To make an output port invisible, you must use the routine `AVSset_output_flags`:

```
int port;
port = AVScreate_output_port("obscure out
port","integer");
AVSset_output_flags(port,INVISIBLE);
```

There are two situations where you can effectively use the port visibility feature. The simplest is when your module contains an optional input or output port that is tangential to the module's execution. It may be the case that the input confuses the intended use of the module to other users.

The second case is where you have two modules or a class of modules that are intended to be connected to each other. These modules may have a standard downstream connection and a standard upstream connection. Using the mechanism of port classes, you can arrange for an upstream connection to be made automatically when the downstream connection is made. This feature combined with the port visibility feature allows upstream data to be hidden from other users and makes upstream networks much simpler to understand visually.

ConvexAVS saves the port visibility attribute when a network is written out and restores it when the network is read in.

User–defined data

ConvexAVS allows you to define your own data types and use them for inter-module communication. The two ConvexAVS data types used for upstream data (`upstream_transform` and `upstream_geom`) are defined using this mechanism.

User–defined data is implemented as a *class* of data that can have an extensible number of subclasses. The class name for the user–defined data type is `struct`. You can define your own subclasses.

The user–defined data mechanism resembles the C structure definition, although FORTRAN users can also access these data types. ConvexAVS supports a subset of the mechanism for defining a `typedef` of a structure in C. For example, the following declaration defines a data type called `struct foo` that contains two integers, one named hop, the other named hog:

```
typedef struct _foobar {
   int   hop;
   int   hog;
} foo;
```

Elements in this structure are restricted to int, char, float, double and arbitrary dimensional arrays of int, char, float, double. Elements cannot be pointers, unions, structures, enums, bitfields, and so on.

ConvexAVS parses the header file containing these definitions with a parser that has limited understanding of valid C constructs. It ignores all C preprocessor directives and comments.

An example of a valid declaration is:

```
/* These are foo flags--we don't complain but */
/* don't look at them either */
#define FOO_FLAG 1
#define BAR_FLAG 2

typedef struct _decl {
   int flag;
   float matrix1[4][4], matrix2[4][4];
   char matrix_name[256];
} foobar;
```

The ConvexAVS user data definition capabilities are not designed to parse an arbitrary include file, but rather to provide you with the capability to design a header file that can be parsed by ConvexAVS and also included in a C program.

Used with input ports

Inputting a user–defined data type is straightforward:

- For a C module, include the header file defining your data type
- Declare an input port of type `struct` <*classname*>, where *classname* is the name of the type definition in your header file. For the previous example, it is:

  ```
  port=AVScreate_input_port("my port
     name","struct foobar",<flags>);
  ```

- For a C module, the argument to your compute function is a pointer to a structure of the type you have specified. A declaration for the compute function defined in the previous example is:

  ```
  #include "foo.h"

  foo_compute(fooptr)
  foo *fooptr;
  {
     if (fooptr->flag == ...)
  }
  ```

- For a FORTRAN module, there are two ways to access the data depending on whether or not the SINGLE_ARG_DATA flag is set using `AVSset_module_flags`. By default, the arguments to your compute function are expanded so that you have a separate parameter for each field in the structure of the data type. The previous example has four arguments for its input port.
- If you select SINGLE_ARG_DATA, then ConvexAVS passes a single integer value for each user–defined data input or output defined. This integer value is then passed to a number of accessor functions that copy data to or from the user–defined data structure into a local array or scalar variable. For example, you can use `AVSudata_get_int` to retrieve integer arrays or scalars from a user–defined data structure. The module must call `AVSload_user_data_types` before using the accessor functions so ConvexAVS has access to a description of the user–defined data structure. Refer to the example in the /usr/avs/example/user_data_f.f file.

Used with output ports

Using user–defined data for output requires slightly more effort. In the description function, you declare the output port in the same way that you declare the input port. In addition, you must specify the file name that contains the data types to load using the `AVSload_user_data_types` routine as follows:

```
AVSload_user_data_types(filename)
char *filename;
```

For example:

```
AVScreate_output_port("my out port
   name","struct foobar");
AVSload_user_data_types("/mydir/foo.h");
```

If you provide a relative path name to the routine `AVSload_user_data_types`, the file should be located relative to the /usr/avs/include directory.

For a C module, you must declare the data type in your module as a pointer to a pointer to the structure you declared, and then use the `AVSdata_alloc` routine to allocate the data as follows:

```
foo_output(foopp)
foobar **foopp;
{
   if (*foopp == NULL) *foopp =
     AVSdata_alloc("struct foobar",0);
   (*foopp)->flags = ...
   ...
}
```

ConvexAVS does not use the second argument to the `AVSdata_alloc` routine. To accommodate future enhancements to user-defined data, you should pass a 0 as the second argument.

Multiple modules in one process

It is now possible to run multiple ConvexAVS modules from the same process. The advantages are:

- There are fewer processes running, which in turn uses less of the system resources.
- Module start-up for the second and subsequent modules does not require the creation of a new process and is therefore faster.
- When two modules that are in the same process are connected, ConvexAVS can avoid some of the communication overhead of passing data between the two modules.
- It can reduce memory requirements.

There is, however, a disadvantage to running many modules in one process. One module could kill the process and thereby kill all the modules running in that process.

Restrictions

There are some restrictions on the conditions under which you can run multiple modules in the same process:

- The two modules must not interfere with each other's data allocation. For example, two modules in the same executable cannot use the same static memory locations to store read or write information. Here is an example of two modules that you cannot run in the same process:

```
int globalvar;

module1_compute(foo,output)
int foo, *output;
{
   globalvar++;
   *output = foo + globalvar;
}

module2_compute(bar,output)
int foo, *output;
{
   globalvar--;
   *output = bar - globalvar;
}
```

- Two instances of the same module cannot run in the same executable if they rely on any read or write *static* data. Modules that do not use any read or write static data are usually called *reentrant* modules. The term *static* as used here refers to the state information that is carried over between calls to the compute function. It is not used in the context of the keyword `static` used in the C language. Modules that are not reentrant cannot be executed in the same process.
- You cannot run coroutine modules in the same process with any other module.
- You cannot mark an input port with the MODIFY_IN flag and run multiple modules in a single process. This flag is used in situations when you want the module to be able to modify the data on its input port. Because all modules in a single executable can share the same data, modules that rely on the MODIFY_IN flag are not suitable to run with other modules in a process.
- In general, modules that do not free allocated static data in their destruction function should not be run with other modules in a single process. If memory is not freed, it is not available to other modules in the process that are still active.

Implementing multiple-module processes

To run a module cooperatively with other modules in the same executable, you must set the COOPERATIVE flag. Use the `AVSset_module_flags` routine in the description function of each module that you want to run cooperatively as follows:

```
AVSset_module_flags(COOPERATIVE);
```

In order to run a module cooperatively, the module must run cooperatively with all the modules in the executable file.

If the module is reentrant, (that is, multiple instances of a particular module can be run from the same process), you must explicitly mark it as such by setting the REENTRANT flag as follows:

```
AVSset_module_flags(COOPERATIVE | REENTRANT);
```

When ConvexAVS is about to instance a module, it searches the list of currently active modules for an existing process that matches *all* of the following conditions:

- The process is an executable that contains the same version of the module that ConvexAVS is going to execute.
- The module to instance and all active modules in the process are marked as COOPERATIVE modules.
- Either the module is marked as REENTRANT, or there is not an existing instance of the particular module in the executable.
- No modules in the process are currently executing. If ConvexAVS determines there is a module executing, it starts another process for the module. This means that you cannot necessarily rely on a module being run in an existing process.
- ConvexAVS was not started with the `-separate` option.

If you change the module flags or any other option in the module description function, you must regenerate any module library that contains that module.

If a module dies, the `AVSmessage` routine offers you a choice to restart the module. If you choose to restart the module, all modules running in that process are restarted sequentially so that ConvexAVS can run these modules from a single process.

You can also restart a module by pressing the **Module Tools** button while in the Network Editor. This brings up a choice box that contains a **Restart Modules** button.

Implementing reentrant modules

Modules that require the use of *static* data can use the `AVSstatic` feature of the module programmers interface. The term *static* as used here refers to the state information that is carried over between calls to the compute function. It is not used in the context of the keyword `static` used in the C language. This is an external variable of type `char *` that ConvexAVS retains on a per-module basis. It is defined in the include file /usr/avs/include/flow.h. ConvexAVS saves the values assigned to `AVSstatic` after executing the module and restores it before executing the next module. The value is also saved and restored for the initialize and destroy functions that your module might define.

You can use this variable to store a pointer to information that you want to keep available from one module invocation to another. `AVSstatic` is available only in C modules.

Modifying modules that share processes

When a module is instanced, it is possible that it will attach to an existing process rather than starting a new process. It is also possible that the module's executable could have been modified since the previous process was started. You could, therefore, end up running a stale copy of the module.

There are two ways to avoid this:

- Each time you change the module, select the **Read Module** button in the Network Editor. This causes ConvexAVS to mark the current process as running a different version of the module. Subsequent attempts to instance a module in this executable starts a new process.
- Start ConvexAVS with the `-separate` command line option. This causes ConvexAVS to run each module in a separate process regardless of how you set their module flags.

Command Language Interpreter

17

The ConvexAVS Command Language Interpreter (CLI) is an ASCII language that can drive most of the ConvexAVS system. It saves networks and widget layouts, parameter settings, and records basic user interaction in the form of script files for later playback. It is one approach to providing an animation capability within ConvexAVS that allows precise control of network operation for long operations (batch) or fixed sequences of parameter changes.

ConvexAVS modules can also send CLI commands to the kernel to build and modify networks, modify parameter values, change rendering transformations and properties, and other operations ordinarily performed by direct user interaction with ConvexAVS. This provides the basis for building application modules that employ ConvexAVS module networks in applications without requiring you to use the Network Editor directly.

Accessing the CLI

The CLI is accessed every time a network file is read or a script is run. It can also be explicitly accessed for direct user interaction or module access in several different ways.

Command line options

The easiest way to get into the CLI is to start ConvexAVS with the `-cli` command line option:

```
% avs -cli
```

All standard input is directed to the CLI for interpretation and all results from those commands are displayed on standard output. Pressing **RETURN** displays the CLI prompt (`avs>`) which indicates that the CLI is ready for the next command. Entering `help` provides basic summaries of the command sets and individual commands and their usage.

The `-cli` command line option may optionally take a CLI command string so that ConvexAVS can automatically be started running a script or other operation. For example, the following line would run ConvexAVS, play back script1, and automatically quit when it was complete:

```
% avs -cli "script -play /usr/avs/test/scripts/script1 -q"
```

Server options

ConvexAVS can also establish a connection to an external process that is providing the CLI commands and displaying their output. If /usr/avs/examples/`avs_client` is not compiled, run `make` in that directory to compile it. Then start ConvexAVS with the server option:

```
% avs -server &
```

As it starts, it displays the message:

`AVS server port is` *<port_number>*

In another xterm window, run `avs_client` giving the *port_number* value as a command line option:

```
% avs_client <port_number>
```

The standard input to `avs_client` is now sent to the ConvexAVS process as CLI input, and the resulting output appears in the `avs_client` window.

The `avs_client` program provides a simple example of an external CLI driver and demonstrates how additional local commands can be added to the `avs_client` side to extend the command set.

Module access

Individual modules can also send CLI commands to the ConvexAVS process that is running them. This allows ConvexAVS application modules to manage ConvexAVS networks in response to changes in their own parameters.

C:

```
AVScommand (destination, command_buffer, output_buffer,
            error_buffer)
char        *destination, *command_buffer, **output_buffer,
            **error_buffer;
```

FORTRAN:

```
AVSCOMMAND (DESTINATION, COMMAND_BUFFER,
            OUTPUT_BUFFER, ERROR_BUFFER)
CHARACTER *(*)          DESTINATION,
                        COMMAND_BUFFER
CHARACTER*<maxsize>     OUTPUT_BUFFER,
                        ERROR_BUFFER
```

This routine can be used to send ConvexAVS CLI commands to the kernel or other CLI receivers. The *destination* argument is currently only used by the kernel. The *command_buffer* is a buffer containing one or more CLI commands. The *output_buffer* and *error_buffer* are used to return the output and error output from executing the commands, respectively.

Choose a <maxsize> for the buffer arrays that is appropriate (the size you choose is communicated automatically from FORTRAN to the ConvexAVS C routines). Extra output beyond that amount is lost. Multiple commands can be included in the same command buffer and should be separated by newline characters. The accumulated output and errors are in the buffers returned with a single result for the overall operation.

When a module references itself in a CLI command, it can use the variable reference `$Module` instead of an explicit name. This is only valid in an `AVScommand` call.

The .avsrc file option

ConvexAVS recognizes an .avsrc file option that reads a CLI file in as part of system initialization. The following line will source /home/me/my_avs_cli_file as part of initialization so that personal variables can be set as part of start up:

```
CLIinit /home/me/my_avs_cli_file
```

This is useful for defining personal variables (using var_set) to customize the CLI environment. For performing active operations, such as starting up CLI scripts, use the `-cli` option.

CLI concepts

There are some overall concepts that are needed to understand how to use the CLI. Each command is in the following form:

<command_name> <subject> <options>

If an inadequate number of arguments is provided to a command, it will automatically print out the usage message for the command.

Commands and tokens

A command is made up of *tokens*, which may be one or more words. For example, in the following command there are three tokens:

```
parm_set mymodule.user.2:comment "This is a comment"
```

The first token is `parm_set`, which is the command to set a module parameter value. The second token is `mymodule.user.2:comment`, which specifies a module (mymodule.user.2) and one of its parameters (`comment`). The third token is `"This is a comment"`. The use of quotation marks indicates that the four words are to be treated as a single string so it is handled as a single token (the new value for the parameter).

Tokens are separated by one or more spaces. Quotation marks must be used when the spaces are part of a token and are optional otherwise. In the case where a module name contains spaces, quotes must be used to enclose the module name. For example, in the following parameter name, the quotes are used to override the embedded spaces in both the module name and parameter name:

```
"read image.user.1":"File browser"
```

A single set of quotes enclosing the entire token would also be valid (the colon is part of the module–parameter name format).

Case sensitivity

CLI commands are case sensitive. All command nmaes and option flags are lowercase. Identifiers such as module names and aliases and parameter names must use the same combination of upper and lower case letters that were originally defined. Path names must match the case used in the target file system.

Interrupting CLI execution

When the CLI is executing a script, pressing **RETURN** in the standard input window gets the attention of the CLI. It then offers the option of continuing or quitting from the script. Pressing **CTRL-C** to abort excess CLI output will exit from ConvexAVS itself and should not be used. Individual commands can not be interrupted. If the script is being run through the Script Controller Browser, use the **Pause** or **Abort** buttons to interrupt the CLI.

Multiple line commands

Most commands readily fit on a single line. In cases where they do not, a backslash (\) can be used to indicate that the command continues on the next line. The command ends when a line ends without the backslash. For example:

```
avs> parm set mymodule.user.2:comment \
"This is a comment"
```

Some tokens may require embedded newlines, as is the case with a comment that is several lines of text. A newline is recognized as the two characters "\n." In combination with a backslash, it permits lengthy strings to be used. For example

```
avs> script check \
"This is a longer comment \n\
than I thought would be able \n\
to fit on a single line"
```

would result in a three line comment to the `script_check` command.

Variable references

The CLI recognizes commands made up of tokens. Because some tokens can be long (path names for instance) or can change according to context (what platform you are running on or where to find the main ConvexAVS directory), it can be useful to replace literal tokens with variable references. The CLI recognizes local variables whose values are substituted in place of the original token (or part of a token) when the CLI command is interpreted. The $ character indicates a variable reference and will cause the remainder of the alphanumeric string to be replaced with the variable value. For example:

```
avs> module "read image.user.0" -ex $Path/avs_library/read_image
```

The reference `Path` is replaced with the internally maintained variable referencing the `-path` command line argument or the `Path .avsrc` option. Variable references can be contained in any part of the token. However, they are substituted with their values even if they are contained in quoted tokens.

Variables are defined using the `var_set` command and their values are retrieved using the `var_get` command. The following predefined variables are set by ConvexAVS and cannot be changed:

`$Path`	ConvexAVS path
`$DataDirectory`	Data directory
`$NetworkDirectory`	Network directory
`$Pid`	ConvexAVS kernel process ID

Their values represent the value provided by you from command line options or the .avsrc file (with the exception of the process ID).

Redirecting output

CLI commands recognize the > character as redirecting output when used at the end of commands. For example

```
avs> help * >/home/james/tmp/help.txt
```

will cause the help description to be written out to the given file. They also recognize a double character (>>) to append to existing files.

Identifiers

There is a limited variety of objects to be referenced in ConvexAVS, but it is important to understand how the naming conventions work. It is easier to see what names ConvexAVS has created after an interactive session than it is to directly predict names in advance.

Module names and aliases

Module names take the form *module*.user.*N* where *module* is the module's name and *N* is a unique module identifier number assigned when the module is created. The number may be different from what appears in the original network file because that number may already be used. This is what lets ConvexAVS merge network files readily. Unfortunately, it makes it hard to write scripts from scratch because the numbers change. When reading a network, all references are interpreted to use the newly assigned numbers. Subsequent files read in can find it hard to predict what numbers will be used. A Clear Network operation (or `net_clear` command) resets numbers and makes this easier.

The `module` command provides an option to explicitly provide a unique and permanent alias for a module. After building a network file and writing it out, edit the file using your text editor and add an alias clause to the `module` commands to specify an alias. For example

```
module "read image.user.1" -xy 100,100 -alias ReadImage
```

will allow references to the module as ReadImage as well as read image.user.1. If the alias ReadImage is already in use, an error will be reported and the alias will be ignored.

You can also directly tell the CLI to add an alias to the `module` command:

```
avs> module "read image.user.1" -alias ReadImage
```

Once the module has an alias applied to it, it will persist in any networks written out. Setting the alias to an empty string will eliminate any previously defined values.

Parameter names

Parameter names are qualified by what module they are associated with. For example, in the command

`parm_set mymodule.user.1:`*param_name value*

the reference to mymodule.user.1 indicates what module the parameter belongs to and *param_name* indicates which parameter it is. The colon (:) must appear to distinguish where the module name ends and the parameter name starts.

Port names

Port names are similar to parameter names in that they are qualified by the name of the module and a colon between the module name and port name. The port name is usually an integer indicating the number of the input or output port. Port numbers use zero-based indexing (that is, the first input port would be 0, the second input port is 1, and so on). Output ports use the same numbers. Parameter ports can also use integers, starting after the input port count (if there are three input ports (0, 1, and 2), the first active parameter port would be 3). Parameter ports may also use the parameter name itself just like references to the parameter value. Ports can be connected between modules even if they are not visible.

Combining networks

This section describes advanced features that give the CLI greater control over how networks get merged together and managed as separate subnets.

Module tags

Module tags are a means of grouping together related modules for easier handling. Modules that have the same tag can be moved together, hidden from view, disabled, or destroyed with a single command. This provides support for reading in a network as a single unit and deleting it as a single unit without disturbing other modules. It also allows for a generic reference to pieces of a network that fulfill a given function (data input, data processing, and so on) when grafting in a network.

The module command has a `-tag` option that permits a tag identifier to be added to the module using the CLI or being read back from a network file. A tag identifier can be any valid single string token (if spaces are enclosed, quotes are necessary). The `net_read` command also has a `-tag` option that will automatically add tags to any modules instantiated while reading the network file.

A number of commands optionally take tags to reference all modules with the given tag. The `net_clear` command, for example, will delete only those modules with the given tag (the `-not_tag` option deletes all but those with the given tag). If t six different tags are used and we want to delete three of those tagged groups, three calls to `net_clear` using the `-tag` option would selectively remove them.

Module maps

ConvexAVS networks are merged using the **Read Network** function, but this in itself does not allow the networks to be directly combined. Using module maps allows for more direct control of how the networks are grafted together to share select resources and operations.

A module map is a list of existing module instances that should be used when reading a network file rather than creating new instances. The map is created using the `net_map` command to add or remove specific modules or tagged groups to the map or to list the current map. When the network file is read in using a specific module map, each module creation request first checks to see if there is a module with the same base name in the map.

For example, if read image.user.1 is in the module map when ConvexAVS reads in a new network that has an instance of the **read image** module in it, then the existing read image.user.1 is connected to rather than instantiating a new module and adding widgets to it. The existing **read image** module only gets the connections that the network file specifies; any new widget layout and parameter settings are ignored. The existing module will retain its original tag value and will not pick up a new tag from the file being read in.

A module map can also use tags to reference all modules in a group. The first module in a tagged group that is found to match a map request while reading a network file is used.

In a few cases, it is useful to map modules to modules that have different names but that have the same types of inputs and outputs. For example, you might want to set up a generic base network with **read ucd** in it and add a generic network with **read field** in. Type tags allow similar modules to be identified. These are different from module tags in that they are only used in mapping operations. The module `-type_tag` option allows an additional qualifier to be added that provides another map match key. If a module is not otherwise matched and there is a module in the map with the same `-type_tag` value, then a match is made even though they have different names. In the example described, if both the **read ucd** and **read field** modules had the same type tag, then they would match.

Pend operations

A module can be prepended or postpended to another module, inserting itself in between the specified module and the modules that are connected to it on a particular port (the `-prepend` and `-postpend` options on the module command). This can be useful to provide additional filtering before or after a module without specifically referring to the connections that are being disconnected and reconnected to include the new module. When the new module is deleted or the `-unpend` option is used, the connections that were broken in the initial operation are remade between the original modules.

CLI scripts

CLI scripts are most easily generated by recording network operations and parameter settings using the `script` command. Scripts can also be written or edited by hand or generated by a program. ConvexAVS can then be driven under CLI control to reproduce a series of operations, in effect providing a basic mode of animation. They are useful for simple animations, test suites, and demonstrations.

A script is basically very similar to a network file except that each individual step is seen by you rather than being done all at once. It includes parameter settings that are to be done sequentially (changes are seen after each setting) rather than all at once during initialization. It also allows you to change network connectivity and control what is running.

Writing scripts

The easiest way to write a CLI script is to let ConvexAVS write the basic script for you and then use that ASCII file as a template. This minimizes the confusion in determining which commands to use and what names to use.

1. Start ConvexAVS with the `-cli` option

   % **avs -cli**

   and enter the Network Editor.

2. Enter the following:

   avs> **script -open** <*your_file*> **-echo yes**

   This will start a new file that records the CLI equivalent of the actions you perform using the mouse. It always begins with a `net_clear` operation so it can start with a clean slate.

It will prompt you for verification if there is an existing network. If you want to create a script using the current network, first do a Write Network to save the network and its current state. Once you have begun the script, use Read Network to bring it back in again.

3. Now you can do basic network editing operations and parameter settings. They are recorded out to your file.
4. Many common operations are not recorded to scripts. These include all operations in the Geometry Viewer, Image Viewer, and Graph Viewer. You have to edit these commands into your script manually to achieve full animation capabilities. The `-echo` option on the `script` command allows you to see what is and is not recorded to the script.
5. You can add comments to your script by typing:

   `avs>` **`script_check`** *"comment text"*

   These comments are echoed when the script is played back and appear in the script controller widget (if it is being used). They tell you what should be happening and provide breakpoints that can be used when playing the script back. When you are done, enter the following:

   `avs>` **`script -close`**

   If you quit without closing the script, you will lose the last output buffer.

Playing back scripts

You can now play the script back by entering the following command:

`avs>` **`script -playback`** *<your_file>*

All scripts generated directly by ConvexAVS will begin with a `net_clear` operation to clear away any existing network and start a session with a blank slate. It will not prompt you for verification at this point, so save your existing work if it is important.

Manually edited scripts need not have a `net_clear` operation in them. You need to be careful, however, that when they are played back, the network context they expect is available.

The `script` command has some options detailing how quickly it will play back the script and what it will do as it runs. It can pause, or break, at each command or just at the `script_check` comment points. When it breaks, it can pause for a fixed number of seconds or prompt for user input to continue.

You can also take advantage of the `sh` command to run an external system command such as the `sleep` command for a given number of seconds. There is also a CLI command, `script_sleep`, which will sleep for a given number of seconds while, at the same time, permitting other ConvexAVS operations to proceed.

If you wish to interrupt a script while it is playing back, you can press the **RETURN** key repeatedly until you see a prompt appear:

```
Script - User Interrupt (continue - c, quit - q):
```

You must then enter either **`c`** to continue with the script or **`q`** to quit and return immediately to the `avs>` prompt. Errors or warnings that normally cause a dialog box to appear will instead cause a message to be output on standard terminal output. If a choice is required, the default choice is automatically selected and the script proceeds.

Script Controller browser

An alternative to using the `script` command for playback is to use the Script Controller. Click on any **Help** button. In the avs_help browser that appears, under the Window pulldown menu, there is a button labeled **Help Demos**. Selecting this presents the Script Controller browser showing the scripts available in the /usr/avs/demo/man_scripts directory.

Once a script starts, new buttons appear on the control panel (**Pause**, **Continue**, and **Abort**) that let you pause momentarily during the script, continue after pausing, or stop completely. When the script is finished, the network is left in place (unless you added a `net_clear` command to your script). If it runs slowly, enter **`script -sleep 0`** to avoid pausing after each command that is executed.

The Script Controller is a topic browser that looks in the /usr/avs/demo/man_scripts directory for an optional .topics file that has the file name and a descriptive line about it. This allows you to provide a description of what the scripts do that is more informative than *script1*. The Script Controller browser can also be accessed using the `script` command with the `-interface` (`-i`) option (on the CLI command line):

```
avs> script -i /usr/avs/demo/man_scripts
```

Script suites

There are several additional features of script playback that allow for a series of scripts to be played in succession to allow for demonstration loops or test suites to be run.

- When playing back a script, you can give the `script` command the `-quit` option that tells ConvexAVS to quit when the script is finished.
- You can start ConvexAVS with a CLI command to run immediately. Entering

```
% avs -cli "script -play <myfile> -sleep 0 -quit"
```

  starts ConvexAVS, runs the script, and then exits. Without the `-quit` option, ConvexAVS continues to run, expecting subsequent commands from standard input or other direct interaction from you.

- You can specify multiple scripts to play in a single script command:

  `avs>` **`script -play`** *<file1> <file2> <file3>*

  When *file1* is done, *file2* runs and then *file3* runs.

- You can link from one script to another. If a script file finds a `script` command in it, it will close itself and start that other script file instead. An initial script file can contain a `script` command that starts a series of other scripts.

Command notations

All commands and options are in lower case. CLI commands are case sensitive.

{ }	Indicate optional clauses or tokens.
<name>	Indicate tokens to be replaced with your values as opposed to literal strings.
"..."	Indicates repeatable phrases.
|	Indicates "either OR."
/	Indicates similar tokens with same phrase structure. `-a/b foo` means `-a foo` and `-b foo` are allowed.
#	Indicates a comment describing an option.
( )	Indicates where an abbreviation is allowed; (c)lose means that "c" alone works.
*	Indicates a wildcard match for a value. Only a few commands recognize this value.

Basic commands

These commands control the flow of CLI commands from files, manage variables for easier reference and provide help and other general services.

General commands

`help` Lists available commands or command sets.

With no arguments, `help` lists a summary of the main command sets. A capital letter matches a command set name and lists that command set. A specific command name shows a full description for that command. An asterisk (*) shows a brief summary of all commands. A name ending in an asterisk (*) shows commands starting with that name. Except for specific command requests, a shortened description is given.

`quit` Quits from ConvexAVS with optional confirmation dialog.

`sh` Executes command in an external shell.

All output goes to standard output, so this may not work properly in some CLI modes (module access for instance). All arguments are sent to the shell together and do not need to be quoted into a single string. However, some shell conventions such as macros and output redirection are overridden by CLI command processing.

`version` Identifies version of ConvexAVS.

This command is used to check the current ConvexAVS version number and is also used in network files to record the version number of the ConvexAVS kernel that wrote the network file. In interactive situations, providing a version string has no effect.

Script commands

`script_check` — Checkpoint comment for script playback.

In record mode, write check command to script output in playback or normal mode, write out <*comment*> to output with optional pause or break depending on script command options.

`script` — Manage script output or playback.

This command is used to create scripts (`-open` and `-close`) and to play them back (`-play`). It can also be used to present the Script Controller Browser in a specific directory (`-interface`). During playback, the default mode is to read each command, wait until the network has finished executing, and then proceed with the next command immediately. The `-break` and `-action` options allow for ConvexAVS to pause after each command and wait for a given number of seconds or wait until you press **RETURN** to continue.

`script_sleep` — Pause for number of seconds during script.

This command permits modules to run while the CLI is waiting for the given number of seconds to elapse and allows X and other input events to be processed. The default value is zero.

`source` — Read commands from file without pausing after each.

Only use this command when all commands need to be executed immediately without the flow executive running until the file has been read in completely.

Variables commands

The CLI recognizes local variables whose values are substituted in place of the original token (or part of a token) when the CLI command is interpreted.

`var_get` Get current value of specific CLI variables.

If only one variable is given, only its value is returned. If multiple values are requested, then the name and value of the requested variables are displayed. If no variables are requested, all known variables are displayed.

`var_set` Set the current value of a CLI variable.

This command cannot be used to modify system defined variables, such as DataDirectory, NetworkDirectory, Module, Path, and Pid.

Network Editor commands

These commands are used to create and delete modules, connect ports, save and set parameter values, and list out current network state information. They are presented in subgroups of related commands that handle network–wide operations, manipulate modules, make connections, and modify parameters.

When typing directly in the CLI, it can be difficult to know what commands are of the greatest use. The following list discusses commands that provide information about the current state of the network as well as control it:

Available modules
: The `mod_show` command is useful to find out the available modules (module palette) that can be instanced *or* to find out general information about existing module instances. It will optionally list port or parameter descriptions as well but does not describe connections or parameter values.

Network state
: The `net_show` command describes the current network in terms of the modules it is made up of and the connections between those modules. It displays its output in terms of `module` and `port_connection` commands.

Flow control
: The `net_flow` command is very useful to disable module computation when setting a series of parameter values when they should be treated as a batch of changes instead of incremental changes.

Parameter settings
: The `parm_save` command will list the current values of select parameters or all parameters in the form of `parm_set` commands. It provides the best template for changing parameter values. The `mod_show` command provides some additional range information on parameters.

One of the best ways to see what the CLI commands do is to use the `script` command that will echo out the equivalent CLI commands for most interactive network editing operations. The CLI output may include options that are not required in direct input. For example, the session in Figure 142 produces a simple network.

Figure 142
Simple network CLI script session

```
avs> module "read image" -alias ReadImage -xy 10,10
"read image.user.0"
avs> module "display image" -alias DisplayImage -xy 10,60
"display image.user.1"
avs> mod_show -all
MODULE "read image.user.0" (ReadImage) TYPE: data FLAGS: C subroutine
OUTPUT [0] "Field Output" TYPE: "field 2D 4-vector byte"
PARM [0] "Read Image Browser" TYPE: string RANGE: $NULL ""
MODULE "display image.user.1" (DisplayImage) TYPE: render FLAGS: C subroutine
INPUT [0] "Image Input" TYPE: "field 2D 4-vector byte" FLAGS: required
PARM [1] Magnification TYPE: choice RANGE: "x1 x2 x4 x8 x16" " "
PARM [2] Automag_Size TYPE: integer RANGE: 50 1024
PARM [3] "Maximum Image Dimension" TYPE: integer RANGE: 100 4096
avs> port_connect ReadImage:0 DisplayImage:0
avs> sh ls /usr/avs/data/image/*.x
/usr/avs/data/image/mandrill.x
/usr/avs/data/image/convex.x
/usr/avs/data/image/marble.x
avs> parm_set ReadImage:"Read Image Browser" /usr/avs/data/image/mandrill.x
```

Network commands

`net_clear` — Clear the entire network or a tagged group of modules.

Without any argument, it will delete all modules and their user interface and reset the Network Editor; with an optional tag, it will only delete the modules that match that tag (the `-not_tag` option deletes those that do not match).

`net_flow` — Enable or disable the flow executive or wait for it to complete.

This command controls module computation. It is used to disable the flow executive and to batch–related parameter changes together so that they occur in one step rather than a number of incremental changes.

The `wait` option is only supported through the server communications port and not interactively. The `restart` and `restart_default` options are operations that restart dead modules with the current parameters and default parameters respectively.

`net_group` — Manipulate a group of modules together based on tag references.

`net_map` — List or modify maps of shareable resources.

A map is a list of existing modules that is used in place of new modules during a `net_read` operation and is used to build aggregate networks with shared data source modules and renderers or other shared modules. With no arguments, the existing map names are listed. With only a map name, that map's contents are listed. Other options add or delete individual modules or tagged groups to a map or clear it to empty.

`net_read` — Read network description from file.

A network file consists of optional comments, a version command, and a series of commands that define modules, make connections, set parameters, and define the user interface layout of the network. The network is read in its entirety before any modules begin execution. It may apply a tag identifier to all resulting modules and/or selectively substitute existing modules for new ones based on a map.

`net_show` — Print out existing network connectivity.

This is displayed in the form of `module` and `port_connect` commands. An optional tag shows those modules in a network with that tag and connections that are output from that set of modules.

`net_write` — Write network description to file.

Module commands

`module` — Create or modify a module instance.

The module is created if it does not already exist, and its name is printed out. If the module exists, it is changed to match the given arguments. If the name is given without a ".user.*N*" prefix, it is assumed to be a request to create a new module. The new name is printed out.

The pend operations connect the module in between other modules or remove it from a previous pend operation. A tag is an added identifier that allows a module to be referenced as part of a group of related modules.

`mod_delete` — Delete individual modules.

`mod_exec` — Execute module regardless of changed inputs or parameters.

If the flow executive is disabled, it adds the module to the queue for later execution when the flow executive is reenabled.

`mod_read` — Read the module(s) that are in the given executable file.

`mod_show` — Display information about one or more user or system modules.

May be used to see the ports and parameters of a module or some of its internal flags that are not displayed by the `module` command. If there is no module specified, it lists all modules; otherwise it lists one or more modules. The `-system` option lists the modules in the module libraries that can be instanced.

`present` — Select and display a module control or viewer panel.

This pops up the module panels just like clicking on the module dimple. The viewer names are those that appear on the main ConvexAVS menu buttons and must be given in their entirety (that is, Network Editor). Names are case sensitive. The

`-closed` option is only valid for the Network Editor and controls the initial appearance of the Network Editor work space.

Parameter commands

`parm_save` Save parameter values—all (default) or individually.

If no module name or parameter name is given, all current parameters are displayed. If a tag is given, only modules in the tagged group are checked. The `-since` flag allows selective display of parameters that have changed since particular reference points: since the module began execution (`exec`), since the module was created (`init`), or since an arbitrary checkpoint (`check`). Each `check` request clears the flags on the parameters requested and updates the checkpoint automatically. The `-range` option is only used in a few internal situations to record parameter range changes actively made by the module to a network file. The `-oneshots` option requests that oneshot parameters be shown.

`parm_set` Set parameter value.

If the value is not accepted by the parameter, it is ignored and an error message is displayed. The `-range` option is only used for internal state information when reading network files.

Port commands

`port_connect` Connect two module ports together.

`port_disconnect` Disconnect two modules from each other.

`port_vis` Control visibility of a port.

Geometry Viewer commands

These commands control the state of the Geometry Viewer and manage changes to transformation matrices and properties of objects, cameras, and lights. Most commands operate by default on the current object in the current scene. You can optionally specify a particular object for the duration of a single command or change the current object for all subsequent commands by specifying an object name.

Matrix operations

`geom_get_matrix`	Returns the transformation for an object, camera, or light. The transformation is a 4-by-4 matrix.
`geom_set_matrix`	Sets a transformation for an object, camera, or light.
`geom_concat_matrix`	Appends a transformation to an object or camera.
`geom_set_transformable`	Sets the transform to object, light, camera, or map.

Global object commands

`geom_set_bounding_box`	Turns the object bounding box feature on or off.
`geom_normalize`	Normalizes the current object.
`geom_reset`	Resets the current object.
`geom_refresh`	Refreshes the current scene.
`geom_get_center`	Returns the center of the object.
`geom_set_center`	Sets the center for rotation and scaling of an object.
`geom_set_position`	Sets the position of a camera, light, or object.
`geom_get_extents`	Returns extent information for an object. The extent consists of the *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, and *zmax* of the object.

Browser commands

`geom_show_prop_editor`	Raises the property editor.
`geom_show_texture_editor`	Raises the texture editor.
`geom_show_object_info`	Raises the object information window.
`geom_show_object_list`	Displays the current object browser.

Light commands

`geom_set_light`	Sets the color, type, and state of a light.
`geom_get_light`	Returns the properties of the light: color, state, and type.
`geom_show_lights`	Shows the position of the current light.

Action commands

`geom_cycle_store`	Sets the state of the Store Frames flag for cycles.
`geom_append_frame`	Appends the current object to the current cycle.
`geom_delete_frame`	Deletes the current object from the current cycle.
`geom_cycle_direction`	Sets the direction of cycles.
`geom_cycle_motion`	Sets the motion of cycles.

Object commands

`geom_get_cur_obj_name`	Returns the name of the current object.
`geom_set_cur_obj`	Changes the current object to the object named.
`geom_get_all_obj_names`	Returns the names of all the objects in a scene.
`geom_lookup_obj_names`	Returns object names associated with a module. Only returns the names of objects in the current scene.
`geom_read_obj`	Reads an object from a geometry file.
`geom_save_obj`	Saves all the current object's geometries.
`geom_delete_obj`	Deletes the named object's geometries.
`geom_set_trans_mode`	Sets the object's transform mode.
`geom_get_visibility`	Returns the visibility of the object.
`geom_set_visibility`	Sets the visibility or state of an object.
`geom_get_color`	Returns the color of the object or the background color of the camera or the light color.
`geom_set_color`	Sets the color of an object, light, or camera.
`geom_get_properties`	Returns the properties of the object.
`geom_set_properties`	Sets the material properties of an object.
`geom_set_parent`	Sets the parent for a given object.
`geom_set_texture`	Sets the texture for a given object.
`geom_delete_texture`	Deletes the current object's texture map.

`geom_set_UV_map`	Sets the current object's UV texture map.
`geom_set_texture_map`	Sets the texture map for a given object.
`geom_show_map`	Shows the current object's UV texture map.
`geom_get_render_mode`	Returns the render mode of the object given.
`geom_set_render_mode`	Sets the render mode for the object.
`geom_set_backface_cull`	Turns the backface cull feature on.

Camera commands

`geom_set_scene`	Sets the scene to operate on.
`geom_set_obj_window`	Sets the window range for an object for normalization.
`geom_create_scene`	Creates a new scene.
`geom_create_camera`	Creates a new camera.
`geom_delete_camera`	Deletes the current camera.
`geom_read_scene`	Reads a scene from a scene file.
`geom_save_scene`	Saves the current scene.
`geom_set_freeze_camera`	Turns the freeze camera feature on.
`geom_get_view_modes`	Returns the viewing modes of the current view.
`geom_set_view_modes`	Sets the viewing modes for the current scene.
`geom_set_background`	Sets the background color for the current scene.

Image Viewer commands

These commands control the state of the Image Viewer and manage changes to transformation matrices and properties of images, subimages, views, and other information. The easiest way to get started writing Image Viewer commands is to use ConvexAVS interactively to generate a complex scene in the Image Viewer and to then save it out with Save Scene. The resulting scene file is written in Image Viewer CLI commands. Refer to the /usr/avs/demo/image_viewer directory for Image Viewer scripts provided with ConvexAVS.

Scene commands

`image_read_scene`	Reads a scene from a scene file.
`image_save_scene`	Saves a scene to a scene file.
`image_create_scene`	Creates a new scene with scene location and size.
`image_show_image_list`	Displays the scrolling list of image names.

View commands

`image_create_view`	Creates a new view with view location and size.
`image_delete_view`	Deletes a view.
`image_set_view_size`	Sets the position and size of a view.
`image_get_view_size`	Returns the size of a view.
`image_set_view_transformation`	Sets a transformation for a view.
`image_get_view_transformation`	Returns the transformation of the image or view.
`image_set_color`	Sets the background color of a view.
`image_get_color`	Returns the background color of the view.

Image commands

`image_create_image`	Creates a new image without any data associated with the image.
`image_read_image`	Reads an image from an image file.
`image_write_image`	Writes an image to an image file.
`image_duplicate_image`	Duplicates an image.
`image_delete_image`	Deletes an image.
`image_reset`	Resets an image position to the initial setting.
`image_normalize`	Normalizes an image position to the current view.
`image_set_image_transformation`	Sets a transformation for an image.
`image_get_image_transformation`	Returns the transformation of the image or view.
`image_set_visibility`	Sets the visibility or state of an image.
`image_get_visibility`	Returns the visibility of the image.
`image_raise_image`	Raises the image.
`image_lower_image`	Lowers the image.
`image_zoom_in`	Zooms in the image.
`image_zoom_out`	Zooms out the image.
`image_set_scale_control`	Sets the scale control buttons on or off.
`image_set_bounding_box`	Turns the bounding box on or off.

Image processing technique commands

`image_read_technique`	Reads in an image processing technique.
`image_set_technique_position`	Sets the position and size of the image processing window.
`image_zoom_to_image`	Applies the current image processing technique to the entire image.
`image_set_technique_window`	Sets whether the current image processing technique will be either In Place or New Window.
`image_set_current_image`	Stores the current image into the image data area.
`image_restore_current_image`	Restores the current viewed image to the original image.

Cycle commands

`image_read_cycle`	Reads a cycle from a cycle file.
`image_save_cycle`	Save a cycle to a cycle file.
`image_cycle_read_data`	Reads data from an image file and puts it in the cycle.
`image_append_frame`	Appends the current image to the current cycle.
`image_delete_frame`	Deletes the current image from the current cycle.
`image_cycle_store`	Sets the state of the Store Frames flag for cycles.
`image_cycle_direction`	Sets the direction of cycles and moves one image in the new direction.
`image_cycle_motion`	Sets the motion of cycles.
`image_cycle_speed`	Sets the replay speed of cycles.

Label commands

`image_label_name`	Creates a new label for the image.
`image_label_transformation`	Sets a transformation for a label.
`image_get_label_transformation`	Returns the transformation for a label.
`image_label_color`	Sets the color of a label.
`image_get_label_color`	Returns the color of the label.
`image_label_height`	Sets the height of a label.
`image_get_label_height`	Returns the height of the label.
`image_label_attributes`	Sets the display properties of a label.
`image_get_label_attributes`	Returns the attributes of the label.

Geometry Script Language commands 18

The ConvexAVS Geometry Script Language (GSL) 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 that 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 file name with extension .obj or .scene. Such scripts can then be read using the Read Object and Read Scene functions.

The 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 and object files

The GSL 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, file name 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 differences between files with .scene and .obj extensions are:

- 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.

Command summary

Table 31 summarizes the Geometry Script Language commands.

Table 31
GSL command summary

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.

read

`read` *name geom-file {object-properties}*

This command reads in a new object, making it the child of the current object. The object is called *name* and is read from *geom-file*. The file must be a ConvexAVS 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.

group

`group` *name {object-properties}*

This command creates an object that groups together a list of sub-objects. The object is called *name* and has a set of initial properties (including a list of sub-objects) in *object-properties*. Figure 143 shows an example group.

cycle

`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 (for example, spheres versus lines for a molecule). The object is made a child of the current object, and initial properties can be specified for the object. Figure 144 shows an example cycle definition.

Figure 143
An example grouping

```
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
                        }
          }
}
```

Figure 144
An example cycle definition

```
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 {}
}
```

set_color

`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.

set_material

`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 *specular* vary from 0 to 1. The *specular* exponent, which specifies the roughness of the surface, should lie between 1 (roughest) and 200 (smoothest).

set_matrix

`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.

The specified matrix replaces the existing transform. Contrast this with `rotate`, `translate`, and `scale`, which concatenate transformations with the existing one.

set_position

`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.

set_render_style

`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`

rotate

`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.

translate

`translate` *x y z*

Translates the object by the vector (*x, y, z*). This transformation is concatenated with the object's current transform.

scale

`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.

view

`view` *name widthxheight+x+y bkg-red bkg-green bkg-blue*
{view-property commands}

The following example creates a 500-by-500 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 { }
```

set_matrix

`set_matrix` *4x4-array-of-floats*

Sets the viewing matrix.

set_position

`set_position` *x y z*

Sets the position of origin of the world coordinate system.

inactive

`inactive`

Sets the initial state of the view to be inactive.

rotate

`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.

translate

`translate` *x y z*

Translates the object by the vector (*x*, *y*, *z*). This transformation is concatenated with the object's current transform.

scale

`scale` *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.

Lighting commands

Like viewing commands, lighting commands should only be specified in a file that has a .scene extension.

light

`light` *type index {lighting-property commands}*

This command turns on a light of *type*, where *type* is one of the following:

- `ambient`
- `directional`
- `point`

The light is given an index of *index* and can contain properties specified in *lighting-properties*. Light indexes 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 {}
```

set_matrix

`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.

set_position

`set_position` *x y z*

This command sets the position of point light sources. This attribute does not affect directional light sources.

set_color

`set_color` *red green blue*

This command sets the light source color.

Defaults file

You can specify a defaults file to be read by the Geometry Viewer when you start ConvexAVS with the `-geometry` command–line option. For example:

```
% avs -geometry -defaults /usr/jones/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 145 shows a sample Geometry Viewer defaults file.

Figure 145
Sample Geometry Viewer defaults file

```
view HANK01 400x400+300+100 1.0 1.0 0.0
view HANK02 400x400+750+100 0.0 1.0 1.0
view HANK03 300x300+400+500 0.0 1.0 1.0
```

Example scene file

Figure 146 shows an example scene file.

Figure 146
Example scene file

```
view AVS 503x529+375+208 0.015686 0.078431 0.196078{
set_matrix 0.968946 -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
}
```

Glossary

A

aliasing
A reduction in image quality caused by representing an image as an array of discrete pixel values. Details that are too small to be resolved can cause large, inappropriate fluctuations in pixel values unless anti-aliasing steps are taken. An example is the *jaggies* that can be seen in line renderings.

alpha
A common name for opacity information calculated and stored for each pixel in a frame.

ambient light
Lighting that illuminates a scene from all directions, such as the lighting from a cloudy day. All objects are illuminated equally regardless of their location and orientation.

ambient reflection
The surface reflection of ambient light sources.

animation
A sequence of images that, when shown in succession, produces the illusion of a moving image.

anti-aliasing
A mathematical process that makes jagged or stair-stepped edges appear smoother by averaging the surrounding colors.

artifacts
Unwanted results from a process, such as imperfect shadowing.

B

bounding box
The rectangular box that barely contains a set of geometric objects. It is specified in the current coordinate system by giving its minima and maxima in x, y, and z.

C

children
For a given element in a hierarchical model, those elements in the next level down that are associated with it. For example, a "hand" object might include five "fingers" as children.

clipping plane
A plane in world space parallel to the image plane, used to delimit the parts of a scene to be rendered. Only the objects in the region between near and far clipping planes will be rendered.

computational space
A rectangular grid or mesh of two or more dimensions that is used to map field data for use in ConvexAVS.

concave polygon
A polygon with the property that a line can be found that passes through more than two edges. Equivalently, the sum of its interior angles is greater than 360 degrees.

convex polygon
A polygon whose interior angles (on the inside of the polygon) sum to 360 degrees. Equivalently, there is no line that passes through more than two edges of a convex polygon.

convolution
Calculating a single pixel value from multiple samples taken in the neighborhood of a pixel.

coroutine module
A module consisting 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.

crambin molecule
A protein molecule used as an example, stored in the Brookhaven Protein Data Bank format.

D

data module
A module that generates data or imports data from outside ConvexAVS and converts it into a ConvexAVS data type.

depth cue
A visual cue providing information about the relative distance to the camera of different surfaces, (for example, depth of field effects).

diffuse light
Light that is scattered in all directions

diffuse reflection
The reflection caused by a surface scattering light equally in all directions. The intensity of the reflection is proportional to the area the surface presents to the light source.

dithering
A technique used to reduce the effects of color quantization by adding random amounts of noise to pixel values before quantization.

E

edit list
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.

extrude
Taking a two-dimensional object (x and y) and turning it into a 3-D object by giving it a dimension of depth (z).

F

faceted
The appearance of an object that has many flat surfaces instead of smooth or rounded sides.

false contour
Regions of abrupt changes in pixel values in an image caused by problems with quantization.

filters
Modules that perform operations on data objects of one type and output data of the same or different types.

flat shading
A form of shading in which each surface has the same color at all points, causing abrupt color changes at the edge between one surface another.

flipbook
A method of viewing animation by first saving the images and then viewing them sequentially in real time.

frame
A single image that is part of a sequence.

G

genlock
The ability of a graphics card to accept sync signals from an external video source, such as a VCR, edit them, and then output them to video.

geometric transformation
A change in the geometric configuration of a scene such as rotation or translation (change in location). A transformation can be applied to individual objects or globally to the entire scene.

Gouraud Shading
A shading technique in which the surface shade is calculated at surface vertices, and interpolated for points in the interior.

H

hierarchical model
An organizational model useful for object description that divides a system into levels of abstraction.

I

interpolation
The process of determining a value between discrete points.

irregular field
A data array in which the coordinate space may not have the same number of dimensions as computational space. Each data element in computational space is mapped is explicitly mapped to a point in coordinate space.

J

jaggies
A staircase effect produced by an edge of an object that is inclined slightly with respect to the rows and columns of pixels.

K

keyframe
A particular frame in an animation that is used as a guide for subsequent action.

M

mapper module
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).

mapping
A method for applying an image to the surface of an object to create a realistic appearance.

module parameter
a variable for a module that can be changed by a user or another module that controls the module.

N

NTSC
The standard signal for television broadcasting. Also known as composite video and RS-170A, the acronym stands for the National Television Standards Commission in the United States.

non-planar polygon
A polygon in which the vertices do not lie in a plane.

normal (vector)
The vector perpendicular to a line, plane or surface at a given point.

O

one-sided surface
A surface that is only visible when it "faces toward" the viewer.

orthogonal
Hold the array index in one dimensions constant while letting the other index(es) vary, as in the action of the **orthogonal slicer** module.

P

parent
For a given element in a hierarchical model, the element in the next level up with which it is associated. For example, the fingers of a hand would share the same parent.

perspective projection
The projection of a three-dimensional scene onto a two-dimensional plane in which objects near the viewer appear larger than objects of similar size that are farther away.

Phong interpolation
The method of generating a surface normal vector at each point on a polygon by interpolating between normals that are provided at the polygon vertices.

Phong shading
A shading technique in which each point on a surface is shaded by using surface normals computed with Phong interpolation.

pixel
A picture element of a viewing screen. A rectangular screen is composed of thousands of pixels, each representing the color of an

image at a given point on the screen. The pixel value calculated and stored for each pixel typically consists of several channels such as red, green, and blue components of its color, and opacity or coverage information.

planar (polygon)
A polygon in which all vertices lie in the same plane in three-dimensional space.

polygon
A simple type of primitive surface composed of three or more vertices connected by straight edges.

polyhedron
A three-dimensional object with polygonal surfaces.

port
A channel through which data passes to or from other modules.

POV
Point Of View. The eye-point from which a 3-D scene is viewed.

primitive
The basic geometric building blocks that make up complex objects.

project
In computer graphics, to cast a three-dimensional scene onto a two-dimensional plane. Each point in the scene is mapped to a corresponding point on the plane. Those surfaces not obscured by other surfaces form the resulting projection. The image is formed from this projection by taking (color) samples at regular intervals and generating pixel values accordingly. This models vision, which captures light rays from a three-dimensional scene on the retina of the eye or the film in a camera.

Q

quantization
The process of reducing a sample pixel value to one of a number of discrete values.

R

raster display
The arrangement of pixels in a display monitor as a two-dimension array or grid of pixels.

ray tracing
A rendering method in which at least one ray is cast for each pixel from the viewer's perspective into the model. This technique can be used to produce photo-realistic images. The key feature is that

the ray through each pixel is allowed to reflect or refract off, or through, any surface encountered to simulate a true optical ray.

real time
The viewing of an event as it happens.

rectilinear fields
A field data type that has coordinate data. Each dimension of the data array has a separate and explicit coordinate mapping. The coordinate space has the same number of dimensions as computational space. The spacing of data elements along each axis need not be uniform.

reflection map
A method of mapping an image onto an object in which the image is a rendering of the object's environment. The resulting mirror effect adds realism.

render
The process of taking a 3-D model information and its associated values (camera position, lights, etc.) and creating an image on the computer screen.

renderer module
A module that renders or stores ConvexAVS data on an external device, such as the display screen or a file.

resolution (screen)
The degree of granularity of a display monitor specified by the number of rows and columns of pixels in the display, as well as the size of the monitor.

rotation
A transformation turning an object around an axis.

S

scaling
A transformation changing the size of an object in the x, y, or z directions.

scene description
The process of specifying a scene to be rendered in terms of objects, light sources, and viewing devices.

sobel operator
An edge detecting algorithm (used in the sobel module) originated by Erwin Sobel. Refer to the reference page on the **sobel** module for more information.

specular (reflection)
Having the qualities of a mirror.

Reflected light concentrated near, but not confined to the mirror direction. A specular reflection is brightest when seen from viewpoints along or near the mirror direction, and becomes dimmer away from that direction.

specular highlights
Lighting highlights in which the reflection is concentrated at the source of light, making the object appear shiny.

subroutine module
A module consisting of a description function and a computation function with optional initialization and destruction functions, The main program is supplied by the ConvexAVS library.

T

temporal aliasing
The undesirable strobing effect in an animated sequence caused by abrupt changes in a scene between frames. Temporal aliasing can be eased by motion blur to help the eye make a smooth transition between frames of a moving object.

texture map
A two-dimensional image placed onto a 3-D object to give it texture, such as steel, marble, and wood grain.

transformation
A function applied to the points in a coordinate system to redefine their coordinates. This is usually used to convert between coordinate systems or for rotation, translation, scaling, and so on.

transformation matrix
A 4x4 matrix used to specify a transformation between coordinate systems.

translation
A transformation changing the location of an object.

transparent
The property of a surface allowing light to pass through it.

U

uniform field
A field data type that has no coordinate data. Coordinate mapping is direct and explicit. The coordinate space has the same number of dimensions as computational space,

W

widget
A virtual input device, such as a dial or file browser.

world space
A coordinate system used for scene description, such as placement and orientation of the defined objects in space.

Index

Symbols

.cyc, image cycle file extension 234
.geom 163, 187, 188
.ims, image scene file extension 206
.obj 163, 188, 617
.prop 187, 188
.scene 176, 188, 617
2N-fold connected 390

A

Absolute, position 154
accessing an array 309
Action menu 182
Action submenu 190
adding new frames 183
aggregate data types
  colormap 267
  field 267
  geometry 267
  pixel map 267
  unstructured cell data 267
  user-defined data 267
aggregate vector length 396
alloc flag, geometries 327
allocating fields 305
alpha byte 311
AM 171
AM light 167
ambient light 167
Append Frame 183
Applications button 54
ApplicationsFile 43
ApplicationsFIle, command-line equiv. 37
-appsfile 37, 48
ASCII-format file 187
assistance xl
associated documents xl
Attaching a Label 180
audience xxxv
avs command 35
AVS.applns file 48
avs.inc include file 309
avs.Xdefaults 47
avs.Xdefaults file 49
AVS_FORCEX 34
AVS_GAMMA 34
AVS_HELP 57
AVS_HELP_PATH 34
AVS_VISUAL 34
AVSadd_float_parameter 488
AVSadd_parameter 489
AVSadd_parameter_prop 492
AVSautofree_output 496
AVSbuild_2d_field 496
AVSbuild_3d_field 497
AVSbuild_field 498
AVSchoice_number 500
AVSconnect_widget 504
AVScorout_event_wait 566
AVScorout_exec 507, 566
AVScorout_init 507
AVScorout_input 508, 566
AVScorout_mark_changed 565
AVScorout_output 509, 566
AVScorout_set_sync 567
AVScorout_wait 510, 511, 565, 566
AVScorout_X_wait 566
AVScreate_input_port 512, 579
AVScreate_output_port 513, 579
AVSdata_alloc 514, 582
AVSdebug 515
AVSerror 516
AVSfatal 517
AVSfield structure, member values 308
AVSfield_alloc 518
AVSfield_copy_points 519
AVSfield_data_offset 519
AVSfield_data_ptr 520
AVSfield_free 520
AVSfield_get_dimensions 521
AVSfield_get_int 522
AVSfield_make_template 527

AVSfield_points_offset 528
AVSfield_points_ptr 528
AVSinformation 533
AVSinit_from_module_list 534
AVSinit_modules 535
AVSinitialize_output 536
AVSinput_changed 536
AVSload 537
AVSload_user_data_types 582
AVSmark_output_unchanged 538
AVSmessage 539, 585
AVSmodify_float_parameter 543
AVSmodify_parameter 544
AVSmodule_from_desc 546
AVSparameter_changed 547
AVSset 555
AVSset_compute_proc 550
AVSset_destroy_proc 551
AVSset_init_proc 551
AVSset_module_flags 552, 582, 585
AVSset_module_name 553
AVSset_output_flags 579
AVSstatic 586
AVSudata_get_int 582
AVSwarning 564
AVSXDEFAULTS 34
Axes for Scene 177

B

Bi-Directional, lighting 172
binary format file example 405
Bounce 185
Bounding Box 44, 155, 197
brightness 167
Brookhaven Protein Data Bank format 4, 285
browsers 61
builtin 48
byte 3
byu_to_geom 285

C

camera coordinates 174
camera.c 177
cameras 142, 173
cameras, defined 174
Cancelling operations 55
cell types 387
Center button 159
center, object 143
choice control 101
choices when importing data 273
-class 37
class attribute 578
Clear Network 116
-cli 38
clipping 175
Close 114
Close button 67
Close, file browser 104
Closing the Network Editor 67
color cells 33
color coding 10
color coding error severity 471
Color-Codiing for Field Input/Output Ports 89
color-coding 12
colormap 5, 76
  hue 317
  opacity 317
  saturation 317
  value 317
colormap arrays 317
colormap control 105
colormap generator control 105
colormap, data structure 292
colormap, structure 317
Colors 33, 44
Command Language Interpreter (CLI), demo scripts 58
compiling modules 476
component vector length 396
computational fluid dynamics 385
configuration keywords 36
Connecting Modules 81
connecting ports 462
connectivity 385
connectivity in UCD structures 390
Continuous, 184
control panel 9, 54
control widgets 12, 16, 94, 97, 121
converting applications into modules 477
COORD_X_3D 313
COORD_Y_3D 313
COORD_Z_3D 313
coordinate system 143
coroutine module 470
coroutines 12
create a label 179
Create Camera 142, 176
Create Scene 176
creating a field 310
creating a UCD structure 393
CTRL-U 162
current object area 156
Current Object Browser 157
current object cycle, function key 160
Current Object Indicator 157, 158
cycle Command 619

D

-data 38, 44
data input 10
data module 461
data output 12
Data Output modules 81
data ports 83
data types
  aggregate 267
  primitive 267
  scientific applications used 271
data, UCD structure 389
DataDirectory 44, 62
DataDirectory, command-line equiv. 38
data-flow diagram 15
declaring fields 305
declaring fields for data types 269
defaults file 625
-defaults, geometry 41
Degree of rotation 152
Delete Camera 176
Delete Object 164
Deleting Modules 80
dependent data 293
determining if input changed 474
determining if parameter changed 474
dial control 98
Dial Editor 98
Dials 12
Diffuse Light Reflectance 167
dimple button 152
-dir, geometry 42
Direct 47
Directional, lighting 172
Disable Flow Executive (toggle) 117
Disable Module 75
disjoint lines 168
DISPLAY 34
-display 38
Display Network Editor 67
Display Network Editor button 67
display windows 110
DISPLAYCLASS 34, 43
DisplayGeometryWindow 44
DisplayPixmapWindow 44
double-precision 3

E

Edit layout function 111
edit list 319
edit list data flow 320
edit list protocol 319
Edit Name button, object browser 157
Edit Property 164
environment variables 34, 36
error severity 471
errors
  debug 471
  error 471
  fatal 472
  information 471
  warning 471
executable flow networks 1
Exit button 67
Exiting ConvexAVS 63
extent information, geometries 327
Extents 143

F

F1, function key 148, 160
F2, function key 150, 160
F3, function key 151, 160
F5, function key 160
F6, function key 160
F7, function key 160
field 76
field and UCD structure
  compare and contrast 389
  data dependence 385
  how data is organized 389
  performance 385
  when to use them 385
field components 274, 302 to 304
field macros
  COORD_X_3D 313
  COORD_Y_3D 313
  COORD_Z_3D 313
  I1DV 314
  I2D 314
  I2DV 314
  I3D 315
  I3DV 315
  I4D 315
  I4DV 316
  MAXX 316
  MAXY 316
  MAXZ 316
  RECT_X 316
  RECT_Y 316
  RECT_Z 316
field mapping examples 297 to 301
field structures 307
field, defined 293
fields, classification of 294
fields, irregular 295
fields, rectilinear 295
fields, uniform 296
fields, using FORTRAN 309
File Browser 162

file browser 12, 61
file browser control 103
filter module 461
-filter, geometry 42
Filtering 1
filters, geometries 284
finite element modeling 385
Flash Active Modules (toggle) 119
Flat Shading 170
flow networks 1
Font Selection menu 181
ForceXRender 45
FORTRAN calling sequence, geometry routines 329
FORTRAN module 309
freeing field structures 310
Freeze Camera 176, 177
Front/Back Clipping 177
functions of a module 464
fundamental data type 270

G

Gamma 45
generate colormap 11
generating colormaps 292
GEOM(3v) library 159
GEOMadd_disjoint_line 338
GEOMadd_disjoint_polygon 339
GEOMadd_disjoint_prim_data 340
GEOMadd_disjoint_vertex_data 340
GEOMadd_float_colors 341
GEOMadd_int_colors 342
GEOMadd_int_value 342
GEOMadd_label 343
GEOMadd_normals 344
GEOMadd_polygon 345
GEOMadd_polygons 346
GEOMadd_polyline 347
GEOMadd_polyline_prim_data 347
GEOMadd_polyline_vertex_data 348
GEOMadd_polytriangle 349
GEOMadd_polytriangle_prim_data 350
GEOMadd_polytriangle_vertex_data 350
GEOMadd_prim_data 351
GEOMadd_radii 351
GEOMadd_vertex_data 352
GEOMadd_vertices 352
GEOMadd_vertices_with_data 353
GEOMauto_transform 354
GEOMauto_transform_list 354
GEOMauto_transform_non_uniform 354
GEOMauto_transform_non_uniform_list 355
GEOMcreate_label 355
GEOMcreate_label_flags 356
GEOMcreate_mesh 357
GEOMcreate_mesh_with_data 358
GEOMcreate_normal_object 359
GEOMcreate_obj 359
GEOMcreate_polyh 360
GEOMcreate_polyh_with_data 361
GEOMcreate_scalar_mesh 362
GEOMcreate_sphere 363
GEOMcvt_mesh_to_polytri 364
GEOMcvt_polyh_to_polytri 365
GEOMdestroy_edit_list 365
GEOMdestroy_obj 366
GEOMedit_center 366
GEOMedit_color 367
GEOMedit_concat_matrix 368
GEOMedit_geometry 369
GEOMedit_light 370
GEOMedit_list 572
GEOMedit_parent 370, 371
GEOMedit_position 371
GEOMedit_projection 373
GEOMedit_properties 372
GEOMedit_render_mode 373
GEOMedit_selection_mode 374, 576
GEOMedit_set_matrix 375
GEOMedit_transform_mode 376, 572, 573
GEOMedit_visibility 378
GEOMedit_window 378
geometric descriptions 6
geometric primitives 1
geometries 138
-geometry 38
geometry 76
geometry command 319
geometry database 319
geometry filters
  automatic filtering 286
  creating your own 289
  general 285
  postprocessor 287
  shell usage 286
  specific 285
  supplied 284
geometry objects and types 323
geometry routines
  edit list manipulation 336
  geometry file utilities 337
  linking with libraries 333
  object creation 334
  object modification 335
  object property 336
  object transformation 336
  using include files 333
Geometry Script Language 617
Geometry Viewer 7, 137, 140
geometry, defined 187
-geometry, geometry 42
GEOMflip_normals 380
GEOMgen_normals 380

GEOMinit_edit_list 380
GEOMnormalize_normals 380
GEOMquery_int_value 381
GEOMread_obj 381
GEOMset_color 381
GEOMset_computed_extent 382
GEOMset_extent 382
GEOMset_object_group 382
GEOMset_pickable 383
GEOMunion_extents 383
GEOMwrite_obj 383
GEOMwrite_text 384
Gloss 167
Gouraud Shading 170
-graph 38
grid, UCD structure 389
GridSize 45
group Command 619

H

Help 114
help xl
Help Demos 57
help, using online 56
hierarchy, object 142
Hosts 45
hq display image 64
HSV Color 166
HSV slider 166
hue 166

I

I1DV 314
I2D 314
I2DV 314
I3D 315
I3DV 315
I4D 315
I4DV 316
IEEE 754 double format 3
IEEE 754 single format 3
IEEE hardware support 32
-image 38
image 111
image data 311
Image Processing button 189
Image Viewer 7, 17
ImageAutomagnify 45
ImageScrollbars 45
Immediate button 99
importing colormaps 292
importing data
  choices 273
  using existing module 273
  writing new module 273
importing fields 274
importing geometries 283
importing image data 279
importing UCD structures 291
importing volume data 280
improving edit list performance 320
inactive Command 623
include files 475
independent data 293
inherit button 170
Inherit Property 168
input parameters 11
input ports 11, 462
instantiating a UCD structure 395
integer 3
internal format
  creating UCD structures 393
  UCD include file 392
internal workings
  coroutines 480
  subroutines 478
Invert, colormap 108
irregular field, example 1 300
irregular field, example 2 301
iview, image technique network 222

K

keywords, abbreviating 35

L

Label Attributes 182
Label Height 182
label objects 324
Labeling the Top-Level Object 179
Labels menu 178
lasso 80
Layout Editor 120
levels of error severity 471
libgeom, library 144
libraries 476
-library 39
light command 624
light on or off 172
light, moving the direction of 173
lighting commands 624
Lights 167
lights 142
Lights, colors 173
linear ramp 106
Lines, wireframe 170
linking modules 476

M

magic number 402
maintaining transformation matrices 330
making ports invisible 579
man_scripts 186
mapper module 462
Mapping 1
mapping information 296
Mathematica ThreeScript 285
MAXX 316
MAXY 316
MAXZ 316
mechanical computer-aided design 385
Menu Selection, Geometry Viewer 160
mesh object type 324
mesh_to_geom 285
Metal 168
mid-edge nodes 387
midfix 234
mixing geometries 319
module differences 469
Module Editor 60, 75
Module Editor button 75
module errors 471
module functions
  computation 467
  description 464
  destruction 464
  initialization 464
module icon 9
module input ports 462
module libraries 13, 476
module output ports 462
module ports
  input 462
  output 462
module routine
  AVSmark_output_unchanged 538
module routines
  accessing colormaps 487
  accessing fields 486
  accessing FORTRAN arrays 487
  accessing user data 487
  AVSadd_float_parameter 488
  AVSadd_parameter 489
  AVSadd_parameter_prop 492
  AVSautofree_output 496
  AVSbuild_2d_field 496
  AVSbuild_3d_field 497
  AVSbuild_field 498
  AVSchoice_number 500
  AVScolormap_get 501
  AVScolormap_set 502
  AVScommand 503
  AVSconnect_widget 504
  AVScorout_event_wait 506
  AVScorout_exec 507
  AVScorout_init 507
  AVScorout_input 508
  AVScorout_mark_changed 508
  AVScorout_output 509
  AVScorout_set_sync 510
  AVScorout_wait 510
  AVScorout_X_wait 511
  AVScreate_input_port 512
  AVScreate_output_port 513
  AVSdata_alloc 514
  AVSdata_free 514
  AVSdebug 515
  AVSerror 516
  AVSfatal 517
  AVSfield_alloc 518
  AVSfield_copy_points 519
  AVSfield_data_offset 519
  AVSfield_data_ptr 520
  AVSfield_free 520
  AVSfield_get_dimensions 521
  AVSfield_get_extent 521
  AVSfield_get_int 522
  AVSfield_get_label 523
  AVSfield_get_labels 524
  AVSfield_get_minmax 525
  AVSfield_get_unit 525
  AVSfield_get_units 526
  AVSfield_invalid_minmax 527
  AVSfield_make_template 527
  AVSfield_points_offset 528
  AVSfield_points_ptr 528
  AVSfield_reset_minmax 529
  AVSfield_set_extent 529
  AVSfield_set_int 530
  AVSfield_set_labels 531
  AVSfield_set_minmax 531
  AVSfield_set_units 532
  AVSinformation 533
  AVSinit_from_module_list 534
  AVSinit_modules 535
  AVSinitialize_output 536
  AVSinput_changed 536
  AVSmessage 539
  AVSmodify_float_parameter 543
  AVSmodify_parameter 544
  AVSmodify_parameter_prop 546
  AVSmodule_from_desc 546
  AVSmodule_status 547
  AVSparameter_changed 547
  AVSparameter_visible 548
  AVSport_field 548
  AVSset_compute_proc 550
  AVSset_destroy_proc 551
  AVSset_init_proc 551
  AVSset_input_class 552

AVSset_module_flags 552
AVSset_module_name 553
AVSset_output_class 554
AVSset_output_flags 554
AVSwarning 564
Command Language Interpreter 484
coroutines 484
creating fields 486
description functions 485
handling errors 487
include files 483
initializing 484
interpreting parameters 484
modifying parameters 484
monitoring status 484
selective computation 485
Module Tools 117
ModuleLibraries 45
ModuleLibraries, command-line equiv. 39
ModulePanelHeight 46
Modules 2
-modules 39
modules
coroutine 470
data 461
filter 461
mapper 462
renderer 462
subroutine 469
modules, data input 72
Modules, filter 72
modules, renderer 72
Monochrome, monitors 32
mouse button, left drag 80
Mouse buttons, functions 148
mouse buttons, transforming cameras 151
mouse buttons, transforming lights 150
mouse buttons, transforming objects 148
mouse, left button 79
Movie BYU 285
Moving Icons 79
Moving Modules 80
multiple modules 583
restrictions 583

N

name, of object 143
-netdir 39, 46
-network 40
Network Construction Window 67
Network Control Panel 16
Network Editor 7, 65
Network Tools 115
network, typical 15
NetworkDirectory 46
NetworkDirectory, command-line equiv. 39
NetworkWindow 46
NetWriteAllParms 46
New Dir, button 162
New Dir, file browser 103
New File, button 162
New File, file browser 104
node numbering 387
node points 385
nodes 385
Normalize 158
Normalize button 200
Normalize, function key 160
-noroll, geometry 42
notational conventions xxxviii
notify mode, geometric objects 571

O

object commands 619
Object Info 168
object, defined 187
Objects 142, 161
Objects Menu Selections 161
objects, creating geometries 326
objects, label 151
oneshot control 102
ordering documents xl
output ports 81, 462
Override, position 155

P

page 121
parameter 463
Parameter data ports 77
parameter ports 83
passing bytes 268
passing integers 268
passing strings 269
Path 46
-path 40
Path, command-line equiv. 40
pdb_to_geom 285
PdbDataDir 46
Perspective 177
perspective 151, 177
Picking and Moving a Label 180
Picture Size 110
pixel map, structure 318
Pixel Processing 5
pixmap 76, 318
Pixmaps 111
PLOT3D
file types 277
reading files 277

Points 170
points, independent data 293
Polygen protein data bank 285
polygon_to_geom 285
polyh_to_geom 285
polyhedron object type 324
polylines 168
polytriangle objects 325
port selection information 573
port type specification 578
port visibility feature 580
Ports and Parameters 9
PostScript 17
ppoly_to_geom 285
primitive data types
  byte 267
  integer 267
  real 267
  string 267
  string block 267
primitive data, geometries 328
Print Network 116
Protein Data Bank format 284
PseudoColor 47
PseudoColor X servers 32
purpose of document xxxv

R

Radio button 12
radio buttons 101
Read button, Edit Property 168
read Command 619
Read Module Library 118
Read Module(s) 117
Read Object 161, 617
Read Scene 176, 617
read volume 11
Read, colormap 110
reading fields
  data-parsing input 275
  limitations 277
  native field input 275
  using combination of modules 275
reading geometries 283
reading UCD structures 291
RECT_X 316
RECT_Y 316
RECT_Z 316
rectilinear field, example 298
redirect mode, geometric objects 571
reentrant modules 584
reflectance properties, changing 164
Regions of Interest (ROI) 189
Relative, position 155
remote servers, running on 64
renderer module 462
Rendering 1
rendering geometries 283
Reset 158
Reset, function key 160
Restore Parameters 117
RGB Color 166
rotate 142
rotate Command 622, 623
running multiple modules 583

S

saturation 166
Save button, Edit Property 168
Save Object 163, 617
Save Parameters 117
Save Scene 176, 617
scale 142
scale Command 622, 623
scatter 311
scene coordinates 141
-scene, geometry 42
scenes 188
screen coordinates 175
screen space 175
ScreenSize 47
ScreenSize, command-line equiv. 41
Script Language 164, 187, 188, 617
script language commands 618
Select Module Library 118
selection information 573
-separate 40
-server 40
set_color Command 620, 624
set_material Command 621
set_matrix Command 621, 622, 624
set_position Command 621, 622, 624
set_render_style Command 621
SharedMemory 47
SharedMemory, command-line equiv. 40
sharing geometries 319
-shm/noshm 40
Show Lights 172
Show Module Documentation 60, 74, 75
Show Object 169
single-precision 3
-size 41
slider control 101
sliders 12
Spec 167
specializing words for fields 306
Specular highlights 167
specular highlights 187
sphere object 326
sphere_to_geom 285

stack 121
startup file 35, 43
startup file format 43
start-up file, format 43
start-up options 36
static data 584, 586
Step Backward 184
Step Forward 184
Store Frames 183
Subimages 190
subimages 189
subroutine module 469
subroutines 12
subsystem, switching 55
Surface properties 143
synchronous execution, modules 567

T

technical assistance xl
Technical Assistance Center xl
The Parameter Editor 75
Toggles 12
Transform Camera, function key 160
transform cameras 151
Transform Image 196
Transform Light, function key 160
Transform Object 148
Transform Object, function key 160
transform objects 148
Transform Selection controls 196
Transform Subimage 197
Transform View 197
transformables 142
transformation matrices 330
transformations 142, 147
  camera 331
  light 331
  object 330
translate 142
translate Command 622, 623
Transparency 168
triangle strips 168
tristate control 102
tristate parameters 77
TrueColor 47
ts_to_geom 285
type-in control 97
Typeins 12
types of modules 461, 469
typographic conventions xxxviii

U

UCD examples
  ucd_function.c 394
  ucd_grid.c 394
UCD format
  external 391
  internal 391
UCD hierarchy 386
UCD magic number 402
UCD routines
  cell manipulation 408
  cell query 409
  linking with libraries 407
  node manipulation 409
  node query 410
  structure manipulation 408
  structure query 408
  using include files 407
UCD structure
  aggregate 393
  ASCII format 396
  binary format 401
  cell abbreviations 398
  cell types 387
  compared to fields 389
  data organization 389
  division of data 393
  example 389
  instantiating 395
  magic number 402
  node numbering 387
  relation to fields 389
  specifying vector length 396
  structure of binary file 405
UCDcell_get_information 411
UCDcell_set_information 412
UCDnode_get_information 413
UCDnode_set_information 414
UCDstructure_alloc 415
UCDstructure_free 416
UCDstructure_get_cell_active 417
UCDstructure_get_cell_components 418
UCDstructure_get_cell_data 419
UCDstructure_get_cell_label 420
UCDstructure_get_cell_labels 421
UCDstructure_get_cell_minmax 422
UCDstructure_get_cell_unit 423
UCDstructure_get_cell_units 424
UCDstructure_get_data 425
UCDstructure_get_data_label 426
UCDstructure_get_data_labels 427
UCDstructure_get_data_unit 428
UCDstructure_get_data_units 429
UCDstructure_get_extent 430
UCDstructure_get_header 431
UCDstructure_get_node_active 433
UCDstructure_get_node_components 434
UCDstructure_get_node_data 435
UCDstructure_get_node_label 436
UCDstructure_get_node_labels 437

UCDstructure_get_node_minmax 438
UCDstructure_get_node_positions 439
UCDstructure_get_node_unit 440
UCDstructure_get_node_units 441
UCDstructure_invalid_cell_minmax 442
UCDstructure_invalid_node_minmax 442
UCDstructure_set_cell_active 443
UCDstructure_set_cell_components 444
UCDstructure_set_cell_data 445
UCDstructure_set_cell_labels 446
UCDstructure_set_cell_minmax 447
UCDstructure_set_cell_units 448
UCDstructure_set_data 449
UCDstructure_set_data_labels 450
UCDstructure_set_data_units 451
UCDstructure_set_extent 452
UCDstructure_set_header_flag 453
UCDstructure_set_node_active 454
UCDstructure_set_node_components 455
UCDstructure_set_node_data 456
UCDstructure_set_node_labels 457
UCDstructure_set_node_minmax 458
UCDstructure_set_node_positions 459
UCDstructure_set_node_units 460
UNC 285
uniform field, example 1 297
uniform field, example 2 297
unpacking an edit list 320
Unstructured cell data 76
unstructured cell data 385
upstream data 86
upstream_geom 576
-usage, avs 41
-usage, geometry 42
user-defined data, data class 580
user-defined data, input ports 581
user-defined data, output ports 582
user-written module 13
using an edit list 322
using this book xxxv

V

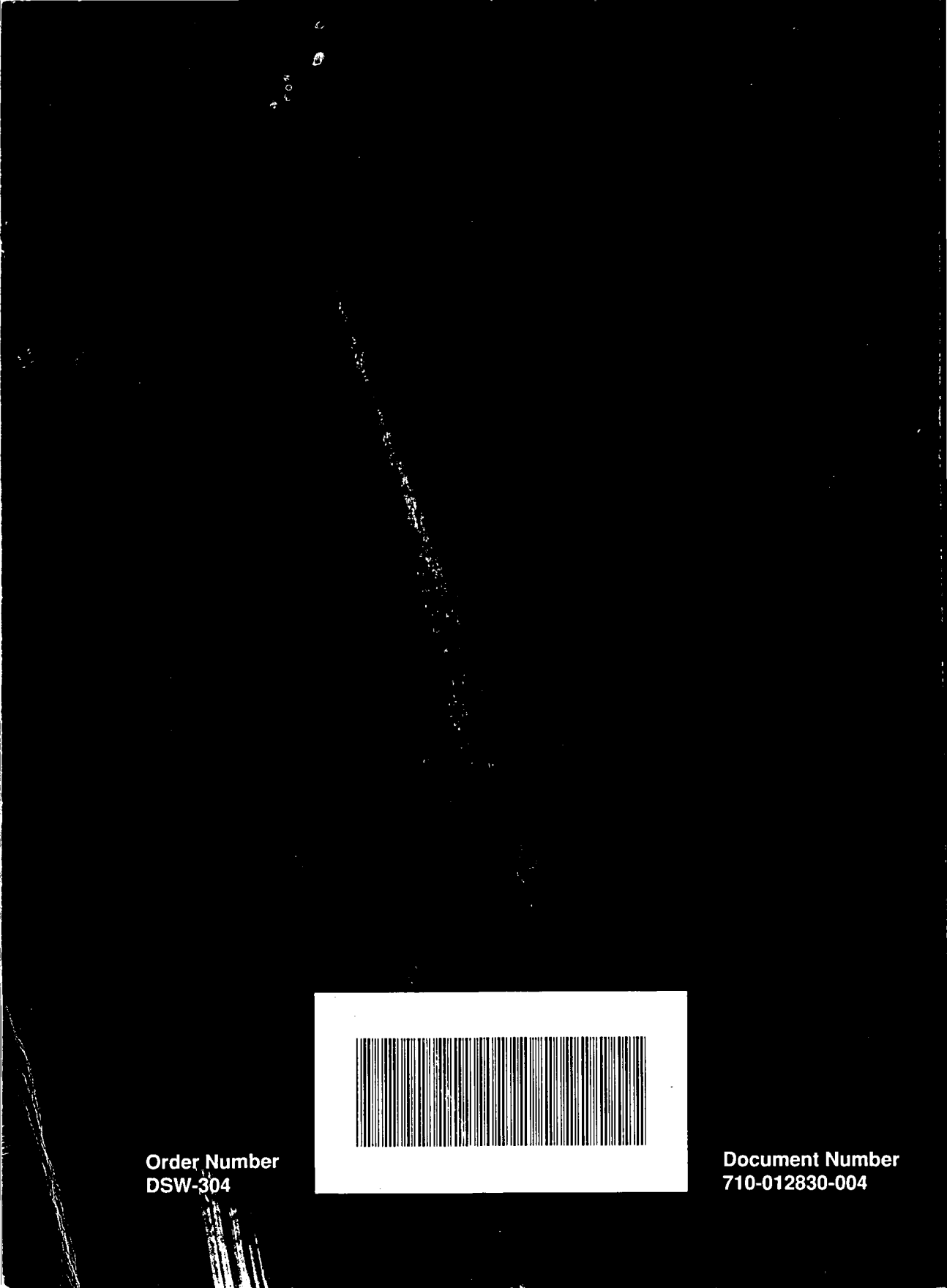
veclen, defined 294
vector length, specifying for UCD structure 396
Verbose Mode (toggle) 119
-version 41
version number 41
vertex colors 168
vertex data, geometries 328
vertex normals 168
view 174
view Command 622
view volume 174
view window 174
-viewer 41
viewports 201
Views. image 190
visibility attribute 143
VisualType 47
-volume 41
volume data 312
volume data format 280
Volume Viewer 17

W

Wavefront 285
wfront_to_geom 285
widgets 61
wildcard 89
Window Background color 178
Window Size 110
WindowMgr 47
wireframe, lines 170
Workspace 80
world coordinate system 330
world coordinates 141
world space 141
Write Module Library 118
Write Network 115
write, colormap 110
writing new geometry filters 289

X

X resource controls 49
X server version 32
X Window 32
Xdefaults.x 49
xlsfonts(1) utility 182
XWarpPtr 47



Order Number
DSW-304

Document Number
710-012830-004