AVS

AVS MODULE REFERENCE

Release 3.0
April, 1991

Stardent

Part Number: 340-0136-02

NOTICE

This document, and the software and other products described or referenced in it, are confidential and proprietary products of Stardent Computer Inc. (Stardent) or its licensors. They are provided under, and are subject to, the terms and conditions of a written license agreement between Stardent and its customer, and may not be transferred, disclosed or otherwise provided to third parties, unless otherwise permitted by that agreement.

NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT, INCLUDING WITHOUT LIMITATION STATEMENTS REGARDING CAPACITY, PERFORMANCE, OR SUITABILITY FOR USE OF PRODUCTS OR SOFTWARE DESCRIBED HEREIN, SHALL BE DEEMED TO BE A WARRANTY BY STARDENT FOR ANY PURPOSE OR GIVE RISE TO ANY LIABILITY OF STARDENT WHATSOEVER. STARDENT MAKES NO WARRANTY OF ANY KIND IN OR WITH REGARD TO THIS DOCUMENT, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

STARDENT SHALL NOT BE RESPONSIBLE FOR ANY ERRORS THAT MAY APPEAR IN THIS DOCUMENT AND SHALL NOT BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF STARDENT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The specifications and other information contained in this document for some purposes may not be complete, current or correct, and are subject to change without notice. The reader should consult Stardent for more detailed and current information.

RESTRICTED RIGHTS LEGEND (U.S. Department of Defense Users)

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights In Technical Data and Computer Software clause at DFARS 252.227–7013.

RESTRICTED RIGHTS NOTICE (U.S. Government Users excluding DoD)

Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in the Commercial Computer Software — Restricted Rights clause at FAR 52.227–19(c)(2).

Stardent Computer Inc.
Six New England Tech Center
521 Virginia Road
Concord, MA 01742

NAME

AVS modules – introduction to manual pages for AVS modules

DESCRIPTION

This section includes a manual page for each module in the AVS distribution. (Note, however, that some versions of AVS may not include all these modules.) The manual pages are also available on-line. You can view them either within AVS itself, or from a shell.

Within AVS

Click on the small square in any module icon to open its Module Editor window. Then click the **Show Module Documentation** button to view the complete manual page for the module. They may also be seen using the regular help browser, in the following directory:

/usr/avs/runtime/help/modules

From a shell

Issue a command in the form:

man 6 *module_name*

Note that in *module_name*, you should replace any SPACE characters that appear on the module icon with underscore characters. For instance, to display the manual page for the **read volume** module, issue this command:

```
man 6 read_volume
```

The data-types that AVS modules operate on are described in Chapter 1 of the *AVS User's Guide*, in the sections on file formats, and in the *AVS Developer's Guide*, in the chapter, "AVS Data Types". Throughout the manual pages for AVS modules, a number of terms are used to describe these data types. It is important to understand these terms, as they specify what inputs a given module can receive, and what outputs it will generate.

any-dimension:
: when a module accepts fields of *any-dimension*, this means that it can process fields that are 1D, 2D, 3D, and in some cases 4D; but never more than this.

n-vector:
: if a field has one value at each location, it is a *scalar* field. When a module accepts *n-vector* fields, it can receive fields with an indeterminate number of values at each location.

any-data:
: if a module accepts *any-data*, this means it can receive byte, integer, float, or double data. If it is more restrictive, this will be declared.

any-coordinates:
: if a module accepts data of *any-coordinates*, this means that it can operate on fields which have uniform, rectilinear, or irregular coordinates. If a module cannot operate on one of these types of field, this will be declared.

MODULE LISTING

The modules included in this release of AVS are:

AVS modules	introduction to manual pages for AVS modules
AVS module groups	Types of AVS modules
avs	Application Visualization System

avsx	run avs as a remote X client on a Stardent ST1000 and ST2000 systems
alpha blend	generate 2D image from 3D colored data (unsupported library on Stardent ST1000 and ST2000 systems)
animated float	send a sequence of floating point numbers to a module's parameter port
animated integer	send a sequence of integers to a module's parameter port
animate lines	animate stream lines for a vector field
antialias	antialias an image
arbitrary slicer	map 3D scalar field to 3D mesh
background	create a shaded backdrop image
boolean	send a user-entered boolean value to one or more module(s) boolean parameter port(s)
brick	show uniform volume as a solid (3D texture mapping systems only, Stardent ST1000 and ST2000 systems)
bubbleviz	generate spheres to represent values of 3D field
cfd values	calculate values for a field containing read plot3D data (unsupported library)
character string	send a user-entered string to one or more module(s) string parameter port(s)
clamp	restrict values in data field
color range	store minimum and maximum field values in an AVS colormap
colorizer	convert field of data values to color values
colormap manager	share colormaps among subnetworks (unsupported library)
combine scalars	combine scalar fields into a vector field
compare field	compare two AVS fields, display and write data difference
composite	blend two images using alpha transparency
compute gradient	compute gradient vectors for 2D or 3D data set
contour to geom	create geometry of 2D or 3D scalar field contour slices
contrast	perform linear transformation on range of field values
convolve	apply a signal processing filter to 2D field
crop	extract subset of elements from a field
display image	show image in a display window
display pixmap	show pixmap in a display window
display tracker	display and directly manipulate the tracer module's output
dot surface	generate points that define an isosurface
downsize	reduce size of data set by sampling
euler transformation	send object transformation matrix to other modules
extract scalar	extract a scalar field from a vector field

extract vector	subset of field vector elements as new field
field legend	select value from scalar field using color legend
field math	perform math operations between fields
field to byte	transform any field to an byte-valued field
field to double	transform any field to a field of double-precision floating point values
field to float	transform any field to a field of single-precision floating point values
field to int	transform any field to an integer-valued field
field to mesh	transform a 2D scalar field to a surface in 3D space
field to ucd	convert AVS field to unstructured cell data format
file browser	send a filename to one or more module(s) filename parameter port(s)
flip normal	change direction of each vertex normal for a geometry object
float	send a floating point number to one or more module(s) floating point parameter port(s)
generate colormap	output AVS colormap
generate filters	generate 2D filters for image processing
generate histogram	plot distribution of data values in a scalar field
gradient shade	apply lighting and shading to colored data set
graph viewer	create XY and contour plots of data (Graph Viewer subsystem)
hedgehog	show vectors in a 3D 3-vector field
histogram stretch	balance the histogram of a data set
image compare	display two images together
image manager	share images among subnetworks (unsupported library)
image to pixmap	convert image to pixmap
image viewer	display and manipulate collections of images (Image Viewer subsystem)
integer	send a user-entered integer to the integer parameter port of one or more module(s)
interpolate	compute intermediate values to change the size of a field
isosurface	generate an isosurface for a volume of data
local area ops	image processing based on pixel neighborhoods
luminence	compute the luminence of an image
mirror	reverse array indices in a 2D or 3D data set
offset	deform, or "blow up" a geometry object based on vector values at each node
oneshot	send a oneshot value to one or more module(s) "oneshot" parameter port(s)

orthogonal slicer	slice through 3D or 2D field with plane perpendicular to coordinate axis
output postscript	convert pixmap to PostScript™ and store in file
particle advector	release grid of particles into velocity field
pdb to geom	create molecule geometry from Protein Data Bank(PDB) file
pixmap to image	transform AVS pixmap to AVS image
print field	create an ASCII printable/readable version of an AVS field
probe	interactively show numeric data values in a geometry rendered field
read field	read AVS field from a disk file, or import data files into AVS field format
read geom	reads a data file containing an AVS 'geometry'
read image	read image file from disk into a field
read plot3d	read a PLOT3D format file into an AVS field (unsupported library)
read ucd	read UCD structure from disk file
read volume	read volume file from disk into a field
render geometry	convert geometric description to image (Geometry Viewer subsystem)
render manager	share geometries among subnetworks (unsupported library)
replace alpha	replace the alpha channele(transparency) in an image
samplers	extract a subset of locations from a 3-vector 3D field
scatter dots	generate spheres at points in 3D space
shrink	make polygons of a geometry object smaller
sobel	apply an edge detecting filter to 2D field
statistics	display statistics on AVS field contents
stream lines	generate stream lines for a vector field
threshold	restrict values in data field
thresholded slicer	slice through volume data with high/low values invisible
tracer	perform ray-traced volumetric rendering on volume data
transform pixmap	perform 3D transformation on pixmap (hardware texture mapping systems only)
transpose	exchange dimensions in a 2D or 3D data set
tristate	send a tristate value to one or more module(s) tristate parameter port(s)
tube	convert lines to cylindrical tubes
ucd anno	show data values of cells or nodes of a UCD structure
ucd contour	generate list of color values associated with unstructured cell data
ucd crop	subset UCD structure data using slice plane or box

ucd extract	extract single node component from a UCD structure
ucd hex to tet	convert a UCD structure from hexahedral cells to tetrahedral cells
ucd hog	show UCD node vector values as line segments in 3D space
ucd iso	generate an isosurface for a UCD structure with scalar node data
ucd legend	creates a color legend relating UCD data to a color scale
ucd offset	deform a UCD structure based on vector values at each node
ucd probe	interactively show numeric data values in a geometry rendered UCD structure
ucd rslice	slice away portions of a UCD structure
ucd slice2D	extract 2D slice from a UCD structure
ucd streamline	generate stream lines for a UCD structure with vector node data
ucd threshold	restrict values in a UCD structure
ucd to geom	convert a UCD structure into an AVS geometry
ucd tracer	perform ray-traced volumetric rendering on a UCD structure
vbuffer	perform volumetric rendering on volume data (unsupported library)
vector curl	compute the curl of a vector field
vector div	compute the divergence of a vector field
vector grad	compute the vector gradient of a scalar field
vector mag	compute the magnitude of a vector field
vector norm	normalize a vector field
volume bounds	generate bounding box of 3D 3-vector field
volume manager	share volumes among subnetworks (unsupported library)
wireframe	convert object from surface to wireframe representation
write field	write a field description to disk
write image	store image data in a file
write ucd	write unstructured cell data to disk
write volume	write volume data to a file

NAME

AVS module groups– Types of AVS modules

DESCRIPTION

The AVS modules can be grouped according to the type of data they operate on, and the operations they perform on that data. This can be helpful, for instance, when you need to find out which modules take fields and convert them to geometries, or which modules save data to disk. The following is a possible division of AVS modules by data type and function.

MODULE GROUPS

READING DATA

read field	read image	read_ucd
read geom	read volume	pdb to geom
read plot3D		

DISPLAYING DATA

display image	display pixmap	image viewer
graph viewer	display tracker	print field
compare field		

SAVING / PRINTING DATA

output postscript	write field	write image
write volume	write ucd	print field

COLORING DATA

colorizer	color range	generate colormap
field legend	ucd contour	ucd legend

GENERATING VALUES TO PARAMETER PORTS

animated float	animated integer	boolean
character string	generate colormap	integer
float	file browser	float
oneshot	tristate	generate filters
samplers	field legend	euler transformation
ucd legend		

FIELD CONVERSION

field to byte	field to double	field to float
field to int	extract scalars	combine scalars
extract vector	field to mesh	field to ucd

CONVERTING FIELDS TO GEOMETRIES

bubbleviz	field to mesh	isosurface
contour to geom	hedgehog	probe
stream lines	volume bounds	thresholded slicer
arbitrary slicer	scatter dots	brick
particle advector	scatter dots	

CONVERTING VOLUMES TO IMAGES

tracer	orthogonal slicer	display tracker

CONVERTING GEOMETRIES TO PIXMAPS

render geometry

CONVERTING PIXMAPS AND IMAGES

AVS Module Groups (6)

pixmap to image | image to pixmap | transform pixmap

CONVERTING FIELD TO UCD

field to ucd

CONVERTING UCD STRUCTURES TO GEOMETRIES

ucd to geom

FIELD PROCESSING AND FILTERING

clamp	crop	downsize
threshold	histogram stretch	interpolate
mirror	offset	transpose
extract scalar	extract vector	combine scalars
cfd values		

IMAGE PROCESSING

contrast	crop	mirror
generate filters	convolve	luminence
background	sobel	interpolate
threshold	clamp	antialias
composite	image compare	local area ops
transpose	replace alpha	

VECTOR PROCESSING

hedgehog	particle advector	stream lines
extract scalar	combine scalars	extract vector
compute gradient	vector div	vector grad
vector mag	vector norm	vector curl
samplers		

GEOMETRY UTILITIES

flip normal	offset	shrink
tube	wireframe	

UCD UTILITIES

ucd anno	ucd extract	ucd hex to tet
ucd contour	ucd legend	write ucd

CFD UTILITIES

read plot3D | cfd values

NAME

avs – Application Visualization System

SYNOPSIS

avs *option(s)*

DESCRIPTION

The Application Visualization System (AVS) is an interactive tool for scientific visualization. It includes the following subsystems:

- **Image Viewer.** A high-level tool for manipulating and viewing images.
- **Graph Viewer.** A high-level tool for graphing data.
- **Geometry Viewer.** Allows you to compose "scenes" that contain geometrically-defined objects. The objects must have been created by programs or AVS modules that use AVS's GEOM programming library. You can transform the objects themselves (move, rotate, scale); you can change the viewing parameters (e.g. move the eye point, perspective view, etc.); and you can control the way in which the graphical images are rendered (lighting and shading, Z-buffering, etc.).
- **Network Editor.** A visual programming interface for connecting computational modules together into networks that perform visualization functions.

AVS also includes two sample applications, the **AVS2 Image Viewer** and the **AVS2 Volume Viewer.** These applications show sample networks that manipulate image and volume data. They are primarily useful for trying out the AVS interface to networked modules on sample data sets.

Use the *avs* command to start AVS when your terminal or workstation is directly-connected to the system that will run AVS. You should also use the *avs* command when running AVS as a remote X client on a Stardent ST1500, ST3000 system. However, if you are running AVS as a remote X client on a Stardent ST1000, ST2000 system, you should use the *avsx* command instead.

CONTROLLING AVS STARTUP

Three entities can affect how AVS starts. They are listed in their order of precedence:

1. Command line options.
2. The *.avsrc* startup file. The startup file contains keyword-value pairs. AVS looks for a startup file in three places, in this order: *./.avsrc* (in the current directory); *$HOME/.avsrc* (in your HOME directory); */usr/avs/runtime/avsrc* (the system-supplied default). AVS uses the first *.avsrc* file that it finds.
3. Environment variables.

OPTIONS

All option keywords begin with a hyphen (e.g. **–data**). In many cases, the keyword is followed by an additional word (e.g. a directory name). You must separate the keyword and the additional word with whitespace (SPACE and/or TAB characters).

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

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

–class *string*

(startup file equivalent: none) This is the command line option equivalent of the DISPLAYCLASS environment variable. You can use it

to make AVS behave in different ways when it is started from different types of display hardware. **-class** has two effects:

1. An *Xdefaults* file specifies the "look" of the AVS interface; what shades of grey are used for command buttons, what fonts to use, whether the background is "stippled" or a flat color, etc. When **-class** *string* is given, AVS does not use the default */usr/avs/runtime/avs.Xdefaults* file. Instead, it looks for an Xdefaults.*string* file in the */usr/avs/runtime* directory and uses it. At present, the only alternate X defaults file supplied is *Xdefaults.X*.
2. AVS will look for a .avsrc.*string* file in your HOME directory and use it instead of your usual *.avsrc* file.

   -class is used when running AVS from an "X terminal." See the full discussion under "Running AVS on PseudoColor X Servers" in the "Starting AVS" chapter of the *AVS User's Guide*.

–cli (startup file equivalent: none) Run AVS with the Command Language Interpreter functioning in the terminal emulator window from which AVS was invoked. This takes an optional argument, which is a CLI command string, to be executed after AVS starts up. See the chapter on the "Command Language Interpreter" in the *AVS User's Guide* for details.

–data *directory*
(startup file equivalent: **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 (files used an input to computational modules). This is the major tool for redirecting AVS's default data input focus off the sample data files provided in */usr/avs/data* and onto your own data files.

The default data directory is */usr/avs/data*.

–dials *devicefilespec*
(startup file equivalent: **DialDevice**) Specifies the serial communications port to which a DIGIT dialbox device is attached (e.g. */dev/tty2*). If **-dials** is present, AVS automatically connects the DIGIT dialbox dials to the Geometry Viewer's rotation, translation, and scaling transformations, *without* requiring you to first enter the Network Editor, instance a copy of the **render geometry** module, then enter the Layout Editor, in order to connect the DialMatrix Manager. You must know which serial communications port your dialbox is connected to. This argument also corresponds to the environment variable DIALS.

–display *display-name*
(startup file equivalent: none) Specifies the X Window System display on which AVS is to execute. This overrides the current setting of the DISPLAY environment variable.

–geometry *[geom-option(s)]*
(startup file equivalent: none) Automatically invokes the Geometry Viewer subsystem at startup. You can include the following options that are specific to this subsystem:

–scene *scene-file*
(startup file equivalent: none) Automatically loads a "scene" from disk storage. This option executes the Geometry Viewer's **Read Scene** function, using the file

scene-file.scene.

-dir *pathname*
Specifies *pathname* 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 AVS data input file browsers).

-filter *pathname1*
Specifies *pathname* as the directory to search for geometry conversion utilities, named *..._to_geom*. See the *Geometry Conversion Programs* appendix.

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

-defaults *filename*
Specifies a Geometry Viewer defaults file. The format of this file is described in the *Geometry Viewer Script Language* appendix.

-geometry *geom_spec*
Specifies an X Window System geometry (e.g. **500x500-5-5**) for the initial window created by the Geometry Viewer.

-usage Displays a list of Geometry Viewer startup options.

-graph Automatically invokes the AVS Graph Viewer at system startup.

-image Automatically invokes the AVS Image Viewer at system startup.

-library *filespec filespec ...*
(startup file equivalent: **ModuleLibraries**) Specifies which AVS module library files to load into the Network Editor at system startup. Module library files are ASCII files describing sets of modules. */usr/avs/avs_library/Supported* is an excellent example. This is the major tool that 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 always relying on the system default Supported and Unsupported module libraries specified in the */usr/avs/runtime/avsrc* file.

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

-modules *directory*
(startup file equivalent: none) Specifies the directory in which the AVS Network Editor subsystem initially will look for executable modules. All executable files in the directory are examined to determine whether they contain one or more modules.

-modules differs from **-library** above 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 AVS search through multiple directories for modules. This is the main 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*

(startup file equivalent: **NetworkDirectory**) Specifies the directory in which the AVS Network Editor subsystem initially will look for network files (**Read Network** and **Write Network** functions). This is the tool to use to redirect AVS's default network focus away from the samples provided in */usr/avs/networks* and onto your own network files.

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

-network *network-file*

(startup file equivalent: none) Automatically invokes the Network Editor subsystem at startup, and loads the specified network file using the **Read Network** function.

-path *directory*

(startup file equivalent: **Path**) Specifies the directory tree in which AVS itself is installed.

The default path is */usr/avs*. If you specify another path, then the default data directory and network directory are modified accordingly. For example:

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

This option is also useful to switch between multiple versions of AVS (for example, a test release and a production release).

-reindex (startup file equivalent: none) This option creates AVS help system *.topics* files. It does not start an AVS session. It is useful if you are creating help files for applications that you want to be accessible through the AVS help system. See the appendix on creating help files in the *AVS Developer's Guide* for more information.

-separate (startup file equivalent: none) This option disables AVS'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. See the section on "Multiple Modules in a Single Process" in the *AVS Developer's Guide.*

-server (startup file equivalent: none) This option opens a connection that an external process can use to connect to AVS and exchange with it a stream of Command Language Interpreter (CLI) commands and their output. See the chapter on the CLI in the *AVS User's Guide* for details.

-shm/noshm

(startup file equivalent: **SharedMemory** *on/off*) This turns the AVS shared memory option on and off. When shared memory is on, AVS keeps only one copy of AVS 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, AVS's use of the finite shared memory area is interfering with other applications. Shared memory is on by default.

-size *XDIMxYDIM*
(startup file equivalent: **ScreenSize**) Specifies size, in pixels, to use for AVS's virtual display screen size. AVS will automatically resize its interface to fit into the virtual screen. You could use this to confine AVS to run within one section of your screen instead of across the whole screen.

-spaceball *devicefilespec*
(startup file equivalent: **SpaceballDevice**) Specifies the serial communications port to which a Spaceball device is attached (e.g. */dev/tty2*). If **-spaceball** is present, AVS automatically connects the Spaceball device to the Geometry Viewer's rotation, translation, and scaling transformations, *without* requiring you to first enter the Network Editor, instance a copy of the **render geometry** module, then enter the Layout Editor in order to connect the Spaceball Manager. You must know which serial communications port your spaceball is connected to. This entry also corresponds to the environment variable SPACEBALL.

-version Displays the AVS version number. (Does not start an AVS session.)

-viewer *viewer-file*
(startup file equivalent: none) Automatically creates a "viewer" that provides "turnkey" access to a group of existing networks. The AVS2 Image and Volume Viewer systems under the **Applications** menu button are implemented in this way. See */usr/avs/applications/volume_viewer/volume_viewer* for a sample viewer file.

-volume Automatically invokes the AVS2 Volume Viewer application at startup.

-usage Displays a usage message for AVS. No AVS session is started.

/AVS STARTUP FILE

When it begins execution, AVS searches for a *startup file*, which specifies such things as which module libraries to load, the locations of various directories, where to look for Help files, how big to make the **display pixmap** window, etc. AVS looks for the following files, in the order listed:

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

At most *one* of these startup files is read. If AVS finds one of them, it ignores the others. A default */usr/avs/runtime/avsrc* file is included on the AVS distribution tape. It is intended that you should copy this file to your HOME or other directory, modify it according to your needs and preferences, and rename it with the "." prefix.

If you give the **-class X** command option, or set the DISPLAYCLASS X environment variable, AVS will first look in your HOME directory for a *.avsrc.X* before it looks for any of the other *.avsrc* startup files. This file is used to customize AVS when you are running it from an "X terminal."

.avsrc Startup File Format

Each line of the AVS startup file consists of keyword-value pair, with whitespace separating the keyword and the value. For example:

```
ModuleLibraries      /usr/avs/avs_library/Supported  /usr/johnp/avs/modules/Modlib
NetworkWindow        867x567+407+2
NetworkDirectory     /usr/johnp/avs/nets
DataDirectory        /usr/johnp/avs/data
DialDevice           /dev/tty02
```

Often, the keyword corresponds to one of the command line options described in the preceding section. If you use a command line option, it overrides the specification, if any, in the startup file.

Startup File Keywords

The AVS startup file keywords are listed below. Keywords that have command line equivalents are listed first in alphabetical order, followed by keywords without command line equivalents, also in alphabetical order.

NOTE: Where startup file keywords have command line equivalents, see the command line description above for the most complete discussion of the feature.

DataDirectory
: (command line equivalent: **-data**) Specifies the directory in which the various AVS 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,** etc.) initially will look for data files. This is the main tool to refocus AVS's data input attention off the sample data files in */usr/avs/data* and onto your own data files.

DialDevice *devicefilespec*
: (command line equivalent: **-dials**) Indicates the serial communications port to which a DIGIT dialbox device is attached (e.g. */dev/tty1*). If **DialDevice** is specified, AVS automatically connects the dialbox dials to the Geometry Viewer's rotate, translate, and scale transformations. It is not necessary to connect the DialMatrix Manager in the Network Editor's Layout Editor.

  This entry also corresponds to the environment variable DIALS; if DIALS is set, the startup file entry (if any) is ignored, and the DIGIT dialbox must be connected through the DialMatrix Manager in the Network Editor's Layout Editor.

ModuleLibraries *filespec filespec ...*
: (command line equivalent: **-library**) Specifies which libraries of modules will be loaded into the Network Editor's module palette. The *last* module library listed will be 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 (perhaps very long) line.

NetworkDirectory
: (command line equivalent: **-netdir**) Specifies the directory in which the AVS Network Editor subsystem initially will look for network files (**Read Network** and **Write Network** functions).

SharedMemory *switch*
: (command line equivalent: **shm/noshm**) Specifying **SharedMemory off** turns off AVS's shared memory feature.

Path
: (command line equivalent: **-path**) Specifies the directory tree in which AVS itself is installed.

ScreenSize *XDIMxYDIM*
: (command line equivalent: **size**) Specifies the size of AVS's virtual display in pixels, confining AVS to run within this area. AVS scales its interface to fit the virtual screen.

SpaceballDevice *devicefilespec*

(command line equivalent: **-spaceball**) Indicates the serial communications port to which a Spaceball device is attached (e.g. */dev/tty1*). If **Spaceball** is specified, AVS automatically connects the Spaceball to the Geometry Viewer's rotate, translate, and scale transformations. It is not necessary to connect the Spaceball Manager in the Network Editor's Layout Editor.

This entry also corresponds to the environment variable SPACEBALL; if SPACEBALL is set, the startup file entry (if any) is ignored, and the Spaceball must be connected through the Spaceball Manager in the Network Editor's Layout Editor.

The following *.avsrc* startup file keywords have no command line equivalents. They are listed in alphabetic order.

BoundingBox *switch*

(command line equivalent: none) If **BoundingBox on** is set, then the AVS 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. **BoundingBox** is most useful when you are using AVS from an "X terminal," and hence may be more likely found in a *.avsrc.X* file than your regular *.avsrc*. See the discussion on "Using AVS on PseudoColor X Servers" below.

Colors *r g b g*

(command line equivalent: none) This option controls how many cells of a *system* colormap AVS will attempt to allocate to itself when it starts. *r g b g* represent numbers for red, green, blue, and gray. This is primarily intended for people who are using AVS from an "X terminal" or PseudoColor workstation that objects to the number of colormap cells that AVS tries to allocate for itself. See the discussion on "Using AVS on PseudoColor X Servers" below.

DisplayPixmapWindow *Xgeometry*

(command line equivalent: none) Controls the default X Window System geometry of the **display pixmap** module's window.

GridSize *n* (command line equivalent: none) Controls the size in pixels of the Layout Editor's alignment squares when **Snap to Grid** is switched on. The default is 10.

HelpPath *directory ...*

(command line equivalent: none) Expands the list of directories that AVS will search to find a module's documentation when you click **Show Module Documentation** in the module's Module Editor window. This is useful when you are using modules other than the set provided with AVS. For the format of the "Help" path, see Appendix D of the *AVS Developer's Guide*, concerning "On-Line Help".

Hosts *fullfilespec*

(command line equivalent: none) 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 */usr/avs/runtime/hosts* file when you click on the Network Editor's **Read Remote Module(s)** button under **Module Tools**. See the "Running Remote Modules" section

in the *AVS User's Guide* Network Editor chapter for details.

ImageAutomagnify
(command line equivalent: none) In AVS3, the display image window will try to select an appropriate magnification factor when the window changes size. Turning this option "on" will restore the AVS2 behavior of automatically magnifying the image.

ImageScrollbars
(command line equivalent: none) If set to the value **off**, 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.)

ModulePanelHeight *Xgeometry*
(command line equivalent: none) Controls the proportion of the Network Construction window devoted to the module palette as opposed to the Workspace.

NetworkWindow *Xgeometry*
(command line equivalent: none) 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.

PrintNetwork *command*
(command line equivalent: none) 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 command should be a regular shell command, such as:

```
lpr -Plw2
```

NetWriteAllParams
(command line equivalent: none) AVs now saves only parameters that have been modified out to a network file. Setting this option to "on", will enable saving all parameters, as was the default in AVS 2.

ReadOnlySharedMemory *switch*
(command line equivalent: none) Shared memory is normally "read only." Occasionally, the system developer might wish to keep shared memory turned on, but allow it to be written into. Setting **ReadOnlySharedMemory 0** accomplishes this.

SaveMessageLog
(command line equivalent: none) If set to the value **on**, causes the AVS message log to be preserved when the AVS session ends normally. By default, the message log (*/tmp/avs_message.log_XXX*, where ***XXX*** is the AVS process number) is deleted automatically. The log file is always preserved if AVS exits abnormally (e.g. **Ctrl-C** interrupt, system crash).

StackSelector *option*
(command line equivalent: none) People who build very large networks sometimes find that the Network Editor's control panel "overflows," making some of the module buttons difficult to access, because the radio buttons take up too much of the control panel. Setting **StackSelector choice_browser** displays the module names as a scrolling list similar to the file browsers instead of as the default **radio_buttons**.

VisualType *visualtype*

(command line equivalent: none) AVS normally uses the X server's default visual. Occasionally, this is the wrong visual to use. For example, the default may be set to DirectColor when there actually is a TrueColor visual available. **VisualType** lets you specify a *visualtype*, either **PseudoColor** or **TrueColor**. AVS will then search the X server's visual list until it finds the first visual with the given visual type and use it. This option may also be useful to people using AVS from "X terminals."

WindowMgr *mgr*

(command line equivalent: none) This option ensures that the Network Editor's Layout Editor and the X Window System window manager that you are using work correctly together. The default for this parameter is specified in the */usr/avs/runtime/avs.Xdefaults* file. On Stardent ST1500/3000/3000VS systems, it is **awm**. The currently recognized values are: **awm, mwm, twm, uwm, olwm**(open look), and **dxwm**(Dec XVI).

XWarpPtr *on*

(command line equivalent: none) Causes the mouse cursor to be automatically moved ("warped") into typein panels when they appear. **XWarpPtr** is off by default.

AVS ENVIRONMENT VARIABLES

AVS uses the following environment variables. Only DISPLAY must be set correctly before AVS will work.

DISPLAY — Used by the X Window System to indicate the display screen at which you're working.

SPACEBALL — (optional) Indicates the serial communications port to which a Spaceball device is attached. When AVS relies on the SPACEBALL environment variable instead of the **-spaceball** command line option or the **SpaceballDevice** startup file keyword, then the Spaceball must be attached to the Spaceball Manager through the Network Editor's Layout Editor.

DIALS — (optional) Indicates the serial communications port to which a DIGIT dialbox device is attached. When AVS relies on the DIALS environment variable instead of the **-dials** command line option or the **DialDevice** startup file keyword, then the dialbox must be attached to the DialMatrix Manager through the Network Editor's Layout Editor.

AVS_HELP_PATH

(optional) Specifies one or more locations in the file system for AVS to use when searching for on-line help files. See Appendix D of the *AVS Developer's Guide* for more on this variable.

DORE_ARGS — (optional) This environment variable only has meaning on Stardent "Titan" series systems. It controls some aspects of the Dore' graphics library execution. Specifically, it is the equivalent of the Dore' **DoDevice** function call. Its arguments are described in the "Device Driver" appendix of the Stardent *Dore' Programmer's Guide*. The argument string should be enclosed in quotes with individual arguments separated by semi-colons.

NAME

avsx – run avs as a remote X client on a Stardent ST1000, ST2000 systems

SYNOPSIS

avsx *option(s)*

DESCRIPTION

Use the "avsx" command to run avs as a remote X client on a Stardent GS system. If you are running avs as a remote X client on a standard Titan system, do not use avsx — use the usual avs command.

avsx takes all the same command line options, and .avsrc keywords as the avs command. For details, see the manual page for avs.

However, some of these options and keywords are particularly germane to running avs as a remote X client. These are repeated here.

OPTIONS

–class *string*

(startup file equivalent: none) This is the command line option equivalent of the DISPLAYCLASS environment variable. You can use it to make AVS behave in different ways when it is started from different types of display hardware. **-class** has two effects:

1. An *Xdefaults* file specifies the "look" of the AVS interface; what shades of grey are used for command buttons, what fonts to use, whether the background is "stippled" or a flat color, etc. When **-class** *string* is given, AVS does not use the default */usr/avs/runtime/avs.Xdefaults* file. Instead, it looks for an Xdefaults.*string* file in the */usr/avs/runtime* directory and uses it. At present, the only alternate X defaults file supplied is *Xdefaults.X*.

2. AVS will look for a .avsrc.*string* file in your HOME directory and use it instead of your usual *.avsrc* file.

-class is used when running AVS from an "X terminal." See the full discussion under "Running AVS on PseudoColor X Servers" in the "Starting AVS" chapter of the *AVS User's Guide.*

–size *XDIMxYDIM*

(startup file equivalent: **ScreenSize**) Specifies size, in pixels, to use for AVS's virtual display screen size. AVS will automatically resize its interface to fit into the virtual screen. You could use this to confine AVS to run within one section of your screen instead of across the whole screen.

STARTUP FILE

When it begins execution, AVS searches for a *startup file,* which contains a number of keyword-value pairs. If you give the **-class X** command option, or set the DISPLAYCLASS X environment variable, AVS will first look in your HOME directory for a *.avsrc.X* before it looks for any of the other *.avsrc* startup files. This file is used to customize AVS when you are running it from an "X terminal." The following *.avsrc* startup file keywords are of special relevance when running avs as a remote X client. They are listed in alphabetic order.

BoundingBox *switch*

(command line equivalent: none) If **BoundingBox 1** is set, then the AVS 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. **BoundingBox** is most useful when you are using AVS from an "X terminal," and hence may be more likely found in a *.avsrc.X* file than your regular *.avsrc*. See the discussion on "Using AVS on PseudoColor X Servers" in the manual page for avs.

Colors *r g b g*

(command line equivalent: none) This option controls how many cells of a *system* colormap AVS will attempt to allocate to itself when it starts. *r g b g* represent numbers for red, green, blue, and gray. This is primarily intended for people who are using AVS from an "X terminal" or PseudoColor workstation that objects to the number of colormap cells that AVS tries to allocate for itself. See the discussion on "Using AVS on PseudoColor X Servers" in the manual page for avs.

VisualType *visualtype*

(command line equivalent: none) AVS normally uses the X server's default visual. Occasionally, this is the wrong visual to use. For example, the default may be set to DirectColor when there actually is a TrueColor visual available. **VisualType** lets you specify a *visualtype*, either **PseudoColor** or **TrueColor**. AVS will then search the X server's visual list until it finds the first visual with the given visual type and use it. This option may also be useful to people using AVS from "X terminals."

SEE ALSO

The section on "Running AVS as a Remote X Client" in the "Starting AVS" chapter of the *AVS User's Guide.*

NAME

alpha blend – generate 2D image from 3D colored data

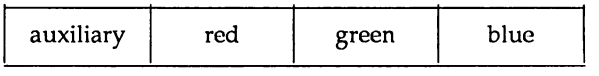
SUMMARY

Name	alpha blend				
Unsupported	this module is in the unsupported library				
Type	data output				
Inputs	field 3D 4-vector byte uniform				
Outputs	pixmap				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	X-Rot	float	0.0	*none*	*none*
	Y-Rot	float	0.0	*none*	*none*

DESCRIPTION

The **alpha blend** module generates an image (2D grid of pixels) from a 3D block of voxels. (*Voxels* are the 3D analogue of *pixels.*) The *alpha blending* technique treats the voxel block as a set of 2-dimensional images, stacked on top of one another. For each line of sight, you can see though layers that contain semi-transparent voxels, up to the nearest layer with an opaque voxel.

The voxel color values are blended from back to front, using each voxel's opacity value:



this field interpreted as voxel's opacity value

these three fields make up voxel's color value

This produces cloud-like images, with the densities of the clouds controlled by the **Opacity** ramp of the colormap that assigned the color values.

INPUTS

Data Field (required; field 3D 4-vector byte uniform)
The input data must be a 3D block of voxels. That is, the data at each point of the 3D field must be a 4-vector of bytes in the alpha-red-green-blue format used in images.

PARAMETERS

By default, the "front" from which the block is viewed is the direction of the positive Z-axis. You can change the direction by rotating the block about the X-axis and/or Y-axis, using these parameters:

X-Rot A floating point value that simulates rotating the data set around the X-axis (horizontal).

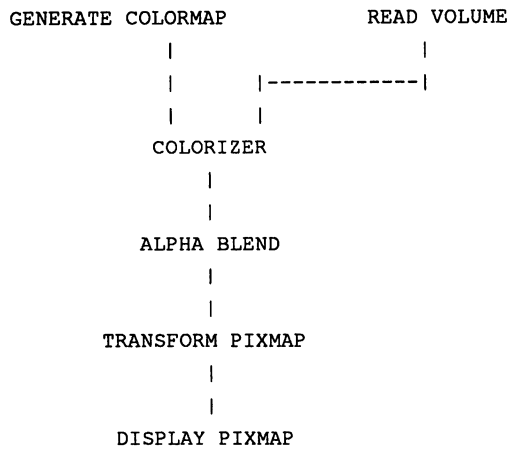
Y-Rot A floating point value that simulates rotating the data set around the Y-axis (vertical).

OUTPUTS

Pixmap The output data is in the form of an AVS pixmap.

EXAMPLE 1

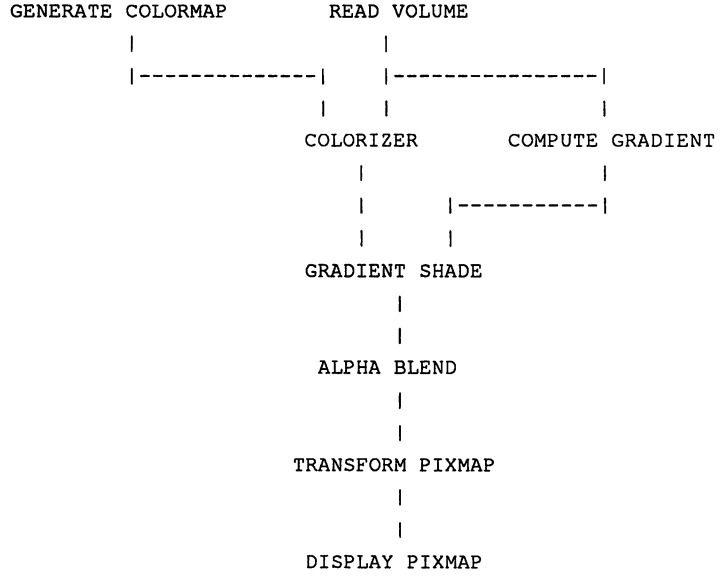
The following network shows how 3D data can be colored using the **colorizer** module, then blended into a 2D image using the **alpha blend** module:



Note that this network uses the **transform pixmap** module to allow the user to resize the image with the window manager. Otherwise, the generated image will be a fixed size, determined by the size of the original data set. For instance, a 64x64x64 data set would produce a fixed-size 128x128 pixel image. (The extra pixels accommodate rotation of the data, which produces a larger image.)

EXAMPLE 2

Another interesting technique is to apply a light source to the data. In order, to do this, the gradient of the data (which approximates the "surface normal") must be computed. A network for doing this "gradient shading" is:



LIMITATIONS

Because of the shearing technique used to simulate axis rotations, there are certain X- and Y-axis angles for which the image breaks up and eventually disappears completely. Complete rotations around one axis only (zero rotation around the other axis) always work correctly.

alpha blend is only available on Stardent ST1000 and ST2000 systems. **alpha blend** will only appear in the unsupported module palette on systems where it is available.

RELATED MODULES

Modules that could provide the **Data Field** input:

colorizer

gradient shade

Modules that could be used in place of **alpha blend**:

vbuffer
orthogonal slicer

Modules that can process **alpha blend** output:

transform pixmap
display pixmap

NAME

animated float – send a sequence of floating point numbers to a module's parameter port

SUMMARY

Name	animated float				
Type	data coroutine				
Inputs	*none*				
Outputs	float				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	min value	float typein	0.0	*unbounded*	*unbounded*
	max value	float typein	0.0	*unbounded*	*unbounded*
	steps	int typein	10	2	*unbounded*
	sleep	switch	on		
	mode	choice	one time		

DESCRIPTION

The **animated float** module automatically modifies floating point parameters. It is used to create simple animations or to drive user simulation code. You plug **animated float** into another module's floating point parameter port (color-coded dark purple), type in minimum and maximum floating point values, and a number of steps (default 10). When you turn off sleep, **animated float** calculates the delta value ((max-min)/step), starts at the minimum value, and begins to send a continuous sequence of evenly-spaced floating point numbers down the connection to the receiving module. Because **animated float** is a coroutine, the AVS flow executive passes one floating point parameter value down the network at a time until the network has fully executed, then signals **animated float** to send the next floating point parameter value. **animated float** can be set to either "one time" (e.g., 1 2 3 4 5), "continuous" (e.g., 1 2 3 4 5 1 2 3 4 5) or "bounce" (e.g., 1 2 3 4 5 4 3 2 1) when it reaches the maximum value. In the last two cases, **animated float** continues to execute until you again toggle "sleep."

For example, you could connect **animated float** to the **isosurface** module's "level" parameter port. By setting minimum, maximum, and step values, you could watch a series of output pixmaps that show the different isosurfaces for each value.

It is often useful to set the minimum and maximum values relative to the range of your data. The **statistics** module can be used to determine reasonable value for these parameters.

The "frame rate" (speed) of the animation depends upon how compute-intensive the downstream modules are. With a compute-bound module like **tracer**, the animation will be quite slow. With simple modules, it will more closely resemble continuous motion. There is no direct way to regulate the speed at which **animated float** executes.

Before you can connect **animated float** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor appears, click any mouse button on its "Port Visible" switch. A purple parameter port should appear on the module icon. Connect this parameter port to the **animated float** module icon in the usual way.

If you bring up the receiving module's control panel, you can watch the parameter values change.

animated float can be connected to multiple modules.

You can save an animation created with **animated float**. Use the **image viewer** module's Action submenu to save a "flipbook" cycle of images (See Example 1).

PARAMETERS

minimum value
A typein to specify the lowest value in the floating point number sequence. It is typed in as a real number (e.g., 1.25 or -.005). There are no upper or lower bound restrictions. The default is 0.0.

maximum value
A typein to specify the maximum value in the floating point number sequence. It is typed-in as a real number (e.g., 5.5 or .003). If the maximum value is less than the minimum value, the delta calculated will be negative and the animation will run backwards. There are no upper or lower bound restrictions. The default is 0.0.

steps An integer typein specifying how many steps the interval between minimum and maximum should be divided into. It cannot be less than two. The default is 10.

sleep A toggle switch that turns **animated float** on and off. It is off by default. When you turn off the stream of floating point numbers by pressing sleep, some number of additional values may continue to flow through the network before **animated float** actually goes to sleep.

mode A set of choices which determine what **animated float** does when it reaches its maximum value. The default is "one time".

one time
With "one time" on (the default), the values are sent only once (e.g., 1 2 3 4 5), and **animated float** sleeps once the values are sent.

continuous
When "continuous" is selected, the values being sent wrap around continuously from highest to lowest (e.g., 1 2 3 4 5 1 2 3 4 5 ...).

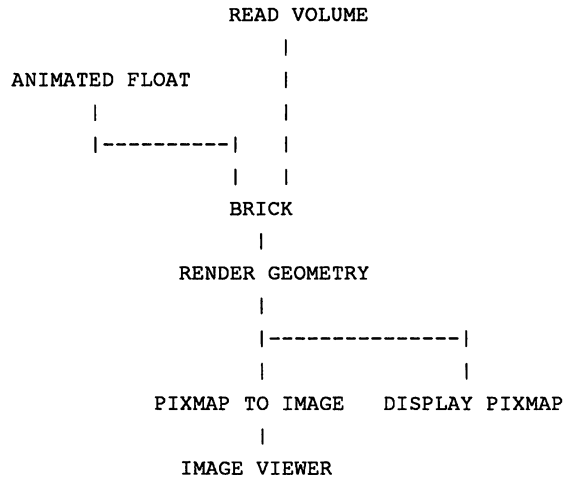
bounce
When "bounce" is selected, the values count up and then count down again repeatedly (e.g., 1 2 3 4 5 4 3 2 1 ...).

OUTPUTS

Floating Point Number (parameter)
A floating point number intended to be input into a floating point parameter port of another module.

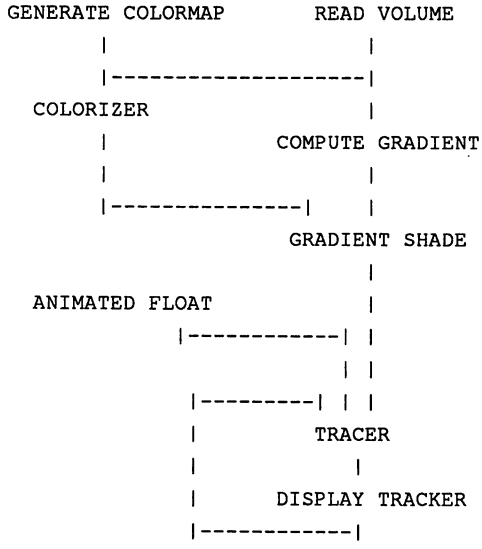
EXAMPLE 1

The following network animates the Offset parameter of the **brick** module. The output is sent to two places: to the usual **display pixmap** module, and to the **image viewer** module through **pixmap to image** conversion where the animation can be saved with the **image viewer**'s action.



EXAMPLE 2

The following network animates the alpha value (transparency) of a volume that has been gradient-shaded, then rendered with **tracer**. Note that **display tracker** sends an upstream transform to the **tracer** module.



RELATED MODULES

Modules that can process **animated float** output:

any module with a floating point parameter

SEE ALSO

animated integer, which behaves exactly like **animated float**, but for integer parameters.

The example script ANIMATED FLOAT demonstrates the **animate float** module.

NAME

animated integer – send a sequence of integers to a module's parameter port

SUMMARY

Name	animated integer				
Type	data coroutine				
Inputs	*none*				
Outputs	integer				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	min value	int typein	0	*unbounded*	*unbounded*
	max value	int typein	0	*unbounded*	*unbounded*
	steps	int typein	10	2	*unbounded*
	sleep	switch	on		
	mode	choice	one time		

DESCRIPTION

The **animated integer** module automatically modifies integer parameters. This can be used to create simple animations or to drive user simulation code. You plug **animated integer** into another module's integer parameter port (color-coded light purple), type in minimum and maximum integer values, and a number of steps (default 10). When you turn off sleep, **animated integer** calculates the delta value ((max-min)/step), starts at the minimum value, and begins to send a continuous sequence of evenly-spaced integer numbers down the connection to the receiving module. Because **animated integer** is a coroutine, the AVS flow executive passes one parameter value down the network at a time until the network has fully executed, then signals **animated integer** to send the next integer parameter value. **animated float** can be set to either "one time" (e.g., 1 2 3 4 5), "continuous" (e.g., 1 2 3 4 5 1 2 3 4 5) or "bounce" (e.g., 1 2 3 4 5 4 3 2 1) when it reaches the maximum value. In the last two cases, **animated float** continues to execute until you again toggle "sleep."

For example, you could connect **animate integer** to the **orthogonal slicer** module's "slice plane" parameter port. By setting minimum, maximum, and step values, you could watch a series of output pixmaps that show progressive slices through the volume data. Without interrupting **animated integer**, you could change the axis from among I, J, and K and see the animated slice sections from any axis.

It is often useful to set the minimum and maximum values relative to the range of your data. The **statistics** module can be used to determine reasonable value for these parameters.

The "frame rate" (speed of the animation) depends upon how compute-intensive the downstream modules are. With a compute-bound module like **tracer**, the animation will be quite slow. With simple modules it will more closely resemble continuous motion. There is no direct way to regulate the speed at which **animated integer** executes.

Before you can connect **animated integer** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A light purple parameter port should appear on the module icon. Connect this parameter port to the **animated integer** module icon in the usual way.

If you bring up the receiving module's control panel, you can watch the parameter values change.

animated integer can be connected to multiple modules.

You can save an animation created with **animated integer**. Use the **image viewer** module's Action submenu to save a "flipbook" cycle of images.

PARAMETERS

minimum value
A typein to specify the lowest value in the integer number sequence. It is typed-in as a whole number (e.g., 25 or -170). This parameter has no upper or lower bounds. The default is 0.

maximum value
A typein to specify the maximum value in the integer number sequence. It is typed-in as a whole number (e.g., -255 or 700). If the maximum value is less than the minimum value, the delta calculated will be negative and the animation will run backwards. This parameter is unbounded. The default is 0.

steps An integer typein specifying how many steps the interval between minimum and maximum should be divided into. If the (max-min)/step delta calculation produces real values, each value is rounded down to the nearest whole integer value. Step cannot be less than two. The default is 10.

sleep A toggle switch that turns **animated integer** on and off. It is off by default. When you turn off the stream of integer numbers by pressing sleep, some number of additional values may continue to flow through the network before **animated integer** actually goes to sleep.

mode A set of choices which determine what **animated float** does when it reaches its maximum value. The default is "one time".

one time
With "one time" on (the default), the values are sent only once (e.g., 1 2 3 4 5), and **animated float** sleeps onbce the values are sent.

continuous
When "continuous" is selected, the values being sent wrap around continuously from highest to lowest (e.g., 1 2 3 4 5 1 2 3 4 5 ...).

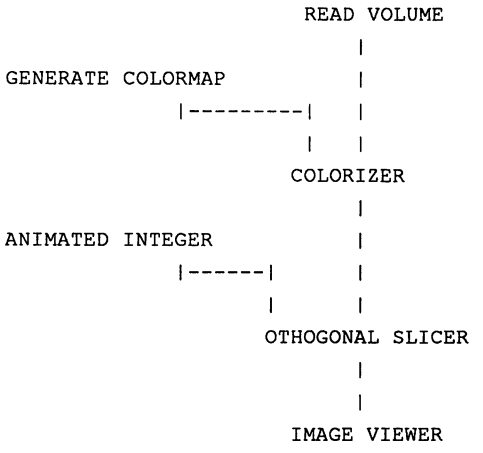
bounce
When "bounce" is selected, the values count up and then count down again repeatedly (e.g., 1 2 3 4 5 4 3 2 1 ...).

OUTPUTS

Integer Number (parameter)
An integer number intended to be input into an integer parameter port of another module.

EXAMPLE 1

The following network animates slices through a volume:



RELATED MODULES

Modules that can process **animated integer** output:

any module with an integer parameter

SEE ALSO

animated float, which behaves exactly like **animate integer**, but for floating point parameters.

The example script ANIMATED INTEGER demonstrates the **animate integer** module.

NAME

animate lines – animate lines for a vector field

SUMMARY

Name	animate lines				
Type	filter				
Inputs	geometry upstream transform				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*MaxValues*
	Objects	text			
	Max Length	text			
	Length	integer	2	2	16
	Animate	oneshot	off		

DESCRIPTION

animate lines takes a set of streamlines output by the **stream lines** module and animates them. **animate lines** outputs successive segments of the streamlines to produce a dynamic representation of them.

Because **animate lines** is a coroutine, the AVS flow executive passes one set of line segments down the network at a time, until the network has fully executed, then signals **animate lines** to send the next set of line segments.

The "frame rate" (speed of the animation) depends upon how many streamlines are passed as input to **animate lines**. With up to an intermediate number of streamlines the animation appears as continuous motion. There is no direct way to regulate the speed at which **animate lines** executes.

INPUTS

Stream Lines (geometry)
: A set of disjoint lines generated by the module **stream lines**.

Upstream Transform (optional, invisible, autoconnect)
: When the **animate lines** module coexists with **stream lines**, and **render geometry** in a network, **render geometry** feeds information on how **stream lines'** point, circle or other "sample probe" has been moved back to this input port on the **animate lines** module. **animate lines** then relays the information up the network to **stream lines**. The modules connect automatically, through data pathways that are normally invisible. This gives direct mouse manipulation control over **stream line's** sample probe.

PARAMETERS

Objects
: A text window which displays the number of line segments which make up the input streamlines.

Max Length
: A text window which displays the maximum length of the input streamlines.

Length
: An integer dial which controls the length of the line segments that are animated along the path of the streamlines.

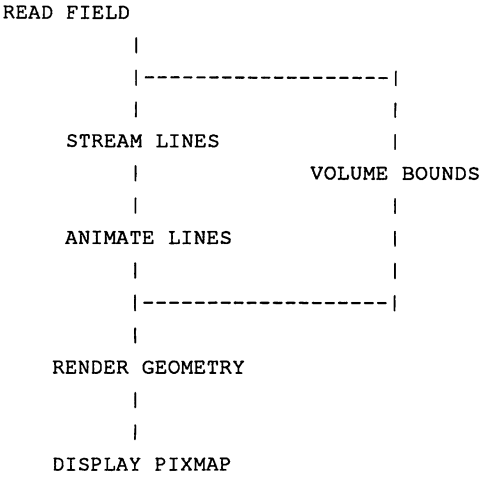
Animate (oneshot)
: A oneshot button that initiates the animation of the streamlines.

OUTPUTS

Animated Lines (geometry)
successive portions of the input streamlines are output sequentially.

EXAMPLE

The following network reads in a 3D vector field, and calculates streamlines for the field. **animate lines** is used to dynamically represent the output of **stream lines**.



RELATED MODULES

hedgehog, particle advector, stream lines

NAME

antialias – antialias an image

SUMMARY

Name	antialias
Type	filter
Inputs	field 2D uniform 4-vector byte *(image)*
Outputs	field 2D uniform 4-vector byte *(image)*
Parameters	none

DESCRIPTION

The **antialias** module downsamples an image using a Gaussian 3x3 convolution filter. This produces an antialiasing effect, reducing jagged edges. The output image is half the size of the input image in each dimension—a 512x512 image becomes a 256x256 image after antialiasing.

INPUTS

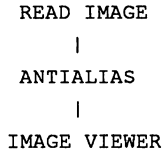
Image (required; field 2D uniform 4-vector byte)
The image to be antialiased.

OUTPUTS

Image (field 2D uniform 4-vector byte)
The output antialiased image. This image is half the size of the input image in each dimension.

EXAMPLE 1

The following network reads an image, antialiases it, and displays it through the **image viewer**.



RELATED MODULES

Modules that could provide the **Image** input:

colorizer
composite
convolve
field math
localops
read image
replace alpha

Modules that can process **antialias** output:

extract scaler
image viewer
display image

See also **downsize, interpolate**

The script ANTIALIAS demonstrates the **antialias** module.

NAME

arbitrary slicer – map 3D scalar field to 3D mesh

SUMMARY

Name arbitrary slicer

Type mapper

Inputs field 3D scalar *any-data any-coordinates*
colormap (optional)
upstream transform (optional, invisible, autoconnect)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Values*
X rot	float	0.0	0.0	360.0	
Y rot	float	0.0	0.0	360.0	
distance	float	0.0	-2.0	2.0	
mesh res	integer	36	8	144	
method	radio	point			point, trilinear

DESCRIPTION

The **arbitrary slicer** module extracts a 2D slice from a 3D volume of data. The slice plane can be oriented arbitrarily — it need not be parallel to any of the coordinate axes.

The volume of data is represented as a 3D scalar field (which defines a uniform lattice within the volume). The slice plane is represented as a 2D grid, with a parameter-controlled resolution. The intersection of the volume and the grid is a *mesh* of vertices in 3D space.

Each vertex in the mesh is assigned a color that corresponds to one or more values of the 3D scalar field. Since, in general, the mesh vertices do *not* coincide with the original lattice points, an interpolation method can be used — see the *method* input parameter below.

By default, the volume is placed at the origin and the slice plane is the X-Y plane. The orientation of the slice plane is controlled by two mechanisms. First, you can control the position of the slice plane using the floating-point dials, X rotation and Y rotation. Second, you can "pick" the slice plane object by clicking on it with the left mouse button. Once it has been "picked" you can orient the slice plane using the same "virtual trackball" paradigm that is used in the Geometry Viewer. Then **arbitrary slicer** receives an upstream transform from the **render geometry** module which tells it how the slice plane has beem moved. Using this information **arbitrary slicer** computes a new mesh output. These two mechanisms can be used together to manipulate the slice plane, in which case the dial transformations are applied first, followed by the upstream transform.

You can control the resolution of the mesh using the **mesh res** parameter. At lower resolutions, fewer original data points are used in the computations; at higher resolutions, more points are used.

Note that by default the mesh is displayed with **No Lighting** selected. To override this feature, select the slice plane object in the Geometry Viewer, and change its type from **No Lighting** to **Gouraud, lines,** or **flat**.

The optimal way to use this module is to start off with a low resolution mesh, position it as desired, then increase the resolution and turn on trilinear mapping.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)
The input data must be a 3D field, with any type of scalar data value at each location in the field. The field can be uniform, rectilinear, or curvilinear.

Colormap (optional; colormap)
By default, the value computed for each vertex of the mesh is used as the hue in HSV space. If you specify a colormap, the values are used to index into the colormap.

Upstream Transform (optional, invisible, autoconnect)
When the **arbitrary slicer** module coexists with the **render geometry** module in a network, and the slice plane object has been "picked", **render geometry** feeds information on how the slice plane has been moved back to this input port on the **arbitrary slicer** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **arbitrary slicer**'s slice plane.

PARAMETERS

X rotation A floating point dial widget that controls the rotation of the slice surface in the X direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction. This controls the orientation of the slice plane in object space.

Y rotation A floating point dial widget that controls the rotation of the slice surface in the Y direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction. This controls the orientation of the slice plane in object space.

distance A floating point value between -2.0 and 2.0 which moves the slice plane back and forth in the direction of the normal to the slice plane. This value is scaled by the largest dimension of the input field. Consequently, you can move the slice plane along the normal from -(2 * **max dimension**) to (2 * **max dimension**).

mesh res Controls the resolution of the slice plane mesh. Higher resolution meshes result in higher quality representations, but take longer to compute and render. The default mesh is 8x8.

method (radio buttons) Controls the way in which each vertex of the output mesh is assigned a color:

- If **point**, a nearest-neighbor algorithm is used. Each mesh vertex is assigned the byte value of the nearest point in the lattice.
- If **trilinear**, a trilinear interpolation is performed. The value at each vertex depends on the byte values at the eight lattice points that are the corners of the "enclosing cube".

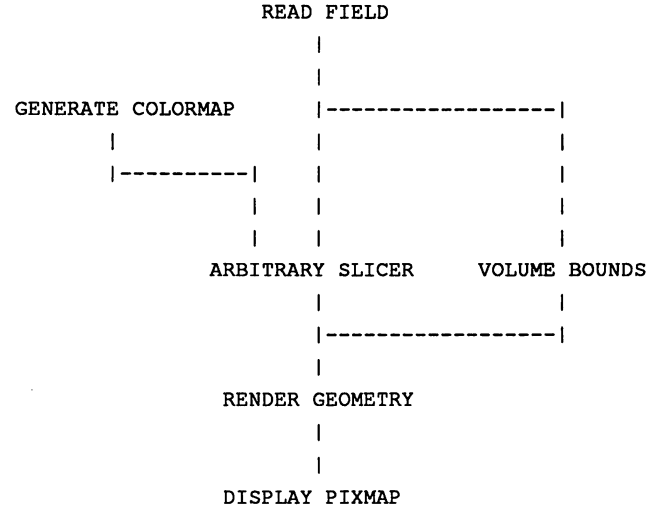
The trilinear interpolation method is more accurate but takes longer to compute, particularly with larger meshes.

OUTPUTS

Geometry (geometry)
The output is an AVS *geometry*.

EXAMPLE

This example shows a common usage of the **arbitrary slicer** module. The **volume bounds** modules gives a reference frame for orienting the slice plane.



RELATED MODULES

Modules that could provide the input field:

read field
read volume
Any module that outputs a 3D field.

Modules that can replace **arbitrary slicer**:

brick
orthogonal slicer
thresholded slicer

Modules that can process **arbitrary slicer**'s output:

render geometry
Any module that inputs a geometry

SEE ALSO

The example script PROBE demonstrates the **arbitrary slicer** module.

NAME

background – create a shaded backdrop image

SUMMARY

Name background

Type data

Inputs field 2D 4-vector byte uniform *(image)* (OPTIONAL)

Outputs field 2D 4-vector byte uniform*(image)*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Upper Left Hue	Dial float	0.67	0.0	1.0
Upper Right Hue	Dial float	0.67	0.0	1.0
Lower Left Hue	Dial float	0.0	0.0	1.0
Lower Right Hue	Dial float	0.0	0.0	1.0
Upper Left Sat	Slider float	1.0	0.0	1.0
Upper Left Value	Slider float	1.0	0.0	1.0
Upper Right Sat	Slider float	1.0	0.0	1.0
Upper Right Value	Slider float	1.0	0.0	1.0
Lower Left Sat	Slider float	1.0	0.0	1.0
Lower Left Value	Slider float	0.0	0.0	1.0
Lower Right Sat	Slider float	1.0	0.0	1.0
Lower Right Value	Slider float	0.0	0.0	1.0
X Resolution	Typein int	128	0	1024
Y Resolution	Typein int	128	0	1024
Dither		Switch	off	

DESCRIPTION

background generates a linearly-shaded image that is typically used as a background for other renderings. You specify the color of each corner with a separate Hue dial. You then use sliders to specify the saturation and value of the color, again individually for each corner. **background** takes the hue-saturation-value of each corner and evenly blends them toward the center of the image.

The results of **background** can be used with the **replace alpha** and **composite** modules to create the effect of a semi-transparent tinted film overlaid upon a regular image. For example, you could create a grey overcast on the image of a sunny sky. When doing this, connect the image to **background**'s input port—this will create a background image the same size as the input image.

The default output image is a 128x128 pixels, shaded blue-to-black image.

INPUTS

Image (optional; field 2D 4-vector byte uniform)

The input image automatically sets the **X Dimension** and **Y Dimension** of the output image. It has no other effect.

PARAMETERS

Upper Left Hue
Upper Right Hue
Lower Left Hue
Lower Right Hue

Floating point dials to select the hue (color) of each corner. The defaults for the upper left and right are .67 (blue); the defaults for the lower left and right are 0.0.

Note:

0.000 = black 0.320 = green 0.670 = blue 1.000 = red
0.167 = yellow 0.500 = cyan 0.833 = magenta

Upper Left Sat
Upper Left Value
Upper Right Sat
Upper Right Value
Lower Left Sat
Lower Left Value
Lower Right Sat
Lower Right Value

Floating point slider bars to select the saturation (how much "white" is mixed in with the hue (1.0=none) and value (how much "black" is mixed in with the hue (1.0=none). All parameters default to 1.0 (fully saturated with no black) except both lower values. These are set to 0.0, making the default lower part of the image all-black.

X Resolution
Y Resolution

An integer typein specifying the size, in pixels, of the output image. The default is 128x128. These parameters will not be visible if there is an optional input image.

Dither A close examination of the **background** image would reveal contour bands of color as the corners shade off if interpolating over a small range of colors over a large screen distance. **Dither** adds a bit of noise in the lower bits of the color value to smooth out this contouring effect. This is a boolean switch that is off by default.

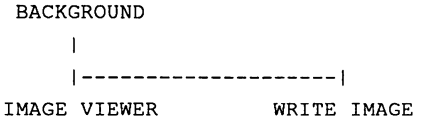
OUTPUTS

Image (field 2D 4-vector byte uniform)

The shaded output image.

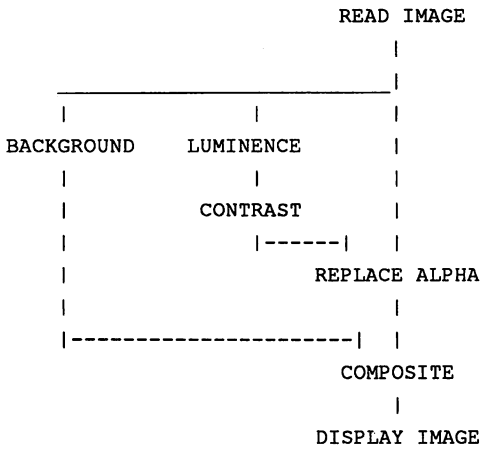
EXAMPLE 1

The following network creates a shaded image and writes the image to disk:



EXAMPLE 2

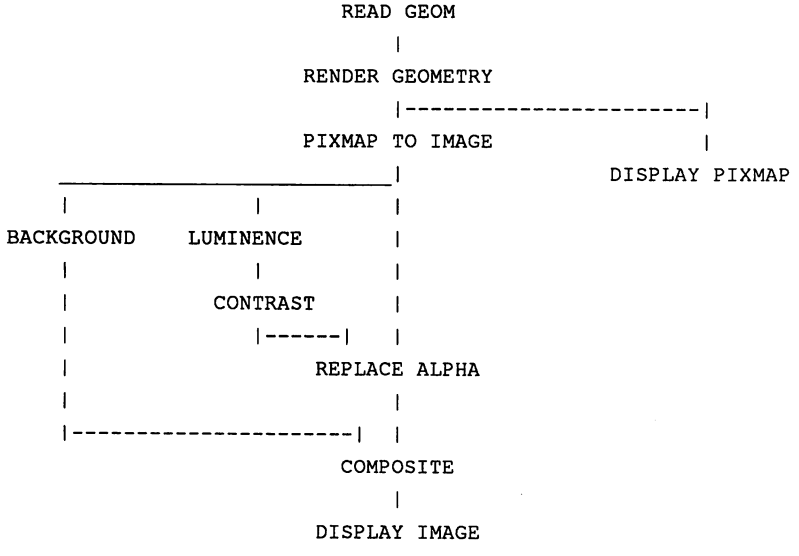
The following network takes an image, computes the luminence, uses that to create an alpha mask, renders a shaded background, and composites the rendered image over the shaded background:



EXAMPLE 3

This network takes a geometry, displays it on the screen, then converts the screen pixmap to an image, computes its luminence, uses that to create an alpha mask, renders a shaded background and composites the rendered image over the shaded background.

In the **contrast** module, you typically want contrast_in_minimum and contrast_in_maximum to both equal 1 to get any non-zero pixel to overlay the background.



RELATED MODULES

Modules that could provide the **Image** input:

read image, pixmap to image

Modules that can process **background** output:

any module that takes an Image as an input...
image viewer
composite

SEE ALSO

Two BACKGROUND example scripts demonstrate the **background** module.

NAME

boolean– send a user-entered boolean value to one or more module(s) boolean parameter port(s)

SUMMARY

Name	boolean		
Type	data		
Inputs	none		
Outputs	boolean		
Parameters	*Name*	*Type*	*Default*
	Boolean Value	*choice*	*off*

DESCRIPTION

The **boolean** module sends a single user-specified boolean value to one or more boolean-type parameter ports on one or more receiving modules. Its purpose is to make it possible for you to simultaneously control boolean parameter input to more than one module using only a single input widget.

Before you can connect **boolean** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **boolean** module icon in the usual way.

PARAMETERS

Boolean Value (boolean)

The single user-supplied boolean value, either on or off, to be sent to the receiving module(s) boolean parameter port(s). The default value is off.

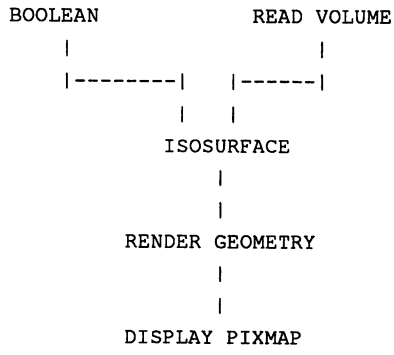
OUTPUTS

Boolean (boolean)

The boolean value is sent to all modules with boolean-type parameter ports that are connected to the **boolean** module.

EXAMPLE 1

In the following network, the **boolean** module has been connected to **isosurface**'s "Flip Normal" parameter:



RELATED MODULES

Modules that can process **boolean** output:

all modules with boolean-type parameter ports

NAME

brick – show uniform volume as a solid (hardware 3D texture mapping systems only)

SUMMARY

Name brick

Type mapper

Inputs field 3D uniform *n-vector any-data (volume)*
upstream transform (optional, invisible, auto-connect)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
X Rotation	float dial	0.0	*unbounded*	*unbounded*
Y Rotation	float dial	0.0	*unbounded*	*unbounded*
Offset	float dial	0.0	*unbounded*	*unbounded*
Sides	boolean	on		

DESCRIPTION

The **brick** module is another way of visualizing 3D uniform volume data. The **arbitrary slice** module displays a slice plane through a volume of data. Outside the slice plane, everything is clear "empty air." **brick** displays the volume as a *solid*— you see the six outside surfaces of an otherwise opaque volume (hence the name "brick"). You can use the **X Rotation, Y Rotation,** and **Offset** parameters to slice a chunk off the brick to reveal the data inside, as one might lop off part of a fruitcake. If you turn off the **Sides** switch, you will see just the slice plane. The effect is similar to the output of **arbitrary slicer**. Only one of the six surfaces of the volume is a moveable slice plane.

brick creates its picture of the volume data using 3D texture mapping (**arbitrary slicer** uses sampling). In this method, the boundary of the volume has three values, u, v, w, associated with each of its vertices. When **brick**'s slice plane intersects this volume, u, v, w values are computed for the vertices of the resulting solid. These values are attached to the vertices of the geometry object which **brick** produces, and are used by **render geometry** to perform 3D texture mapping.

Texture mapping is much faster than the sampling technique used by arbitrary slicer, particularly for large datasets. The point sampling is always done at the resolution of the data; thus differences in data values within a small area are not obscured as they can be with **arbitrary slicer**.

The 3D texture map is created with a combination of the **generate colormap, colorizer,** and possibly **color range** modules. Their output is connected to the **render geometry** module's left texture map port (see example below).

brick has the invisible "upstream transform" input port. This means that "brick" shows up as an object in the Geometry Viewer's object hierarchy. If you select the "brick" object and rotate, scale, or translate it with the mouse, the **render geometry** module informs the **brick** module of the new orientation of the slice plane, and **brick** remaps the volume data accordingly. The effect is that you have direct mouse manipulation control over the shape of the brick.

For **brick** to work, and for it to appear in the module palette, the system it is running on must support 3D texture mapping in both graphics software and hardware. The Stardent ST1000 and ST2000 machines have this feature. Other systems (e.g., the Stardent ST1500 and ST3000 series machines) may not.

INPUTS

Data Field (required; field 3D uniform n-vector any-data)
The input field is a 3D uniform volume. The data can be of any type; byte, integer, double or real.

Upstream Transform (optional, invisible, autoconnect)
When the **brick** module coexists with the **render geometry** module in a network, **render geometry** feeds information on how the "brick" object has been moved in the Geometry Viewer back to this input port on the **brick** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **brick**'s slice plane.

PARAMETERS

X Rotation A floating point dial widget that controls the rotation of the slice surface in the X direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction.

Y Rotation A floating point dial widget that controls the rotation of the slice surface in the Y direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction.

Offset A floating point dial widget that controls the movement of the slice surface in the Z direction. The 0.0 initial value is defined to be *midway* through the volume. Hence, a volume with a Z dimension of 64 has 0.0 in the middle, with +32.0 and -32.0 in either direction. The dial itself is unbounded. If you enter a value outside the actual volume, the slice surface stops at the actual bounds.

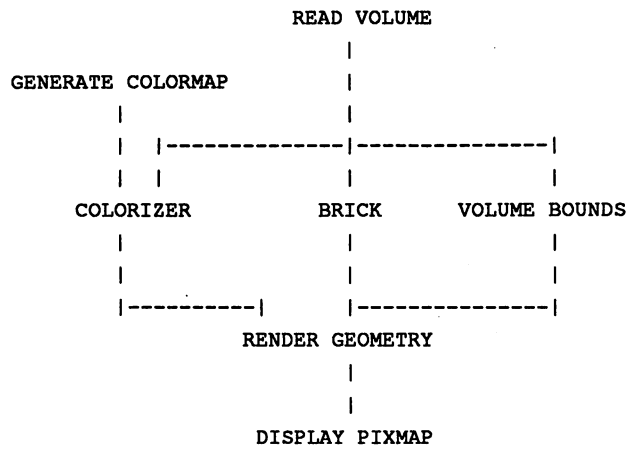
Sides A boolean switch that controls whether all six surfaces of the volume are displayed (on), or only the slice surface (off). **Sides** is on by default.

OUTPUTS

Geometry (geometry)
The output geometry is the solid version of the volume.

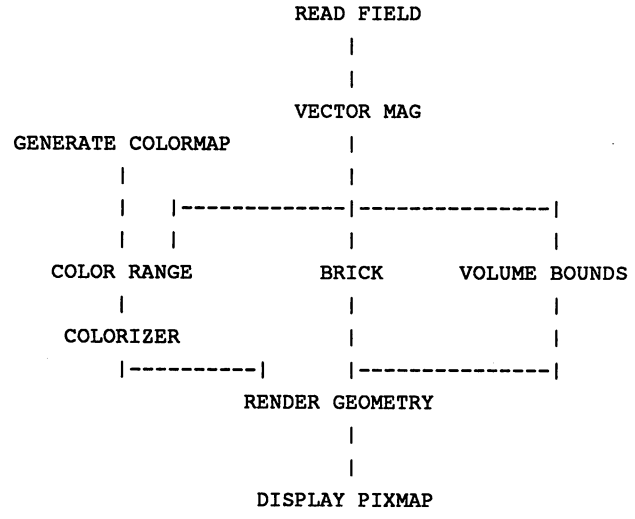
EXAMPLE 1

The following network reads a byte volume. The volume is fed to **colorizer** to paint the byte values as colors, to **brick** to map the surfaces, and to **volume bounds** to draw a box around the limits of the volume. The **generate colormap, colorizer,** and **render geometry** parts of the network are vital; they create the 3D texturemap. All in turn feed into **render geometry**.



EXAMPLE 2

The following network is the same as the previous example in basic structure. The difference is that the uniform volume data is a 3D field of real values, not bytes. The **vector mag** module is used to convert the vector field into a scalar float field. The addition of the **color range** module scales the color values in the colormap to match the range of the data. It should be included whenever the data is not of type byte.



RELATED MODULES

Modules that could provide the **Data Field** input:

- read volume
- read field
- Any module that outputs a 3D uniform field

Modules that could be used in place of **brick**:

- arbitrary slicer
- orthogonal slicer
- thresholded slicer

Modules that can process **brick** output:

- render geometry

SEE ALSO

Two BRICK example scripts demonstrate the **brick** module.

NAME

bubbleviz – generate spheres to represent values of 3D field

SUMMARY

Name	bubbleviz				
Type	mapper				
Inputs	field 3D scalar *any-data any-coordinates* colormap				
Outputs	field 1D 3-coord 4-vector real				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Radius	float	0.0	0.0	100.0

DESCRIPTION

The **bubbleviz** module generates spheres of various radii and colors at the element locations of a 2D or 3D field. This is a "cuberille" style of volume visualization, except that it uses spheres rather than cubes.

The colors and radii of the spheres are calculated by mapping the input field values to the color and opacity values in the colormap. This means that you can change the color of spheres by editing the hue, saturation and brightness panels of the colormap widget. The radii of the spheres is taken from the opacity data (last field) of the input colormap. To change the radii of an entire group of spheres, simply edit the **generate colormap**'s opacity panel.

This module can be used for non-uniform input fields (rectilinear or irregular).

Note that systems which do not have hardware support for sphere rendering have an additional Geometry Viewer control that lets you specify the number of polygons used to render spheres. The control's slider is located at the bottom of the Geometry Viewer control panel, and is titled "subdivision". The subdivision value ranges from 1 to 8; using a low value, e.g. 2, can improve the performance of **bubbleviz** considerably.

INPUTS

Data Field (required; field 2D/3D scalar *any-data any-coordinates*)

The principal input data for the **bubbleviz** module is a 2D or 3D field. The data at each point of the field can be byte, integer, float or double. The values will be interpreted as numbers in the range 0..255.

Color Map (colormap)

The optional colormap may be of any size. Since each input datum is a byte, the natural size for the colormap is 256. If you specify a larger colormap, its entries beyond the 256th are unused.

A zero value in the opacity field of the colormap suppresses the generation of a sphere for the input datum.

PARAMETERS

Radius — A multiplier factor for the sphere radii. This is particularly useful for irregular fields, for which the computational-to-physical mapping often makes the default spheres too small. The value of Radius is used to scale the opacity element in the input colormap.

The default **Radius** is zero; this causes spheres to be rendered as points (individual pixels).

OUTPUTS

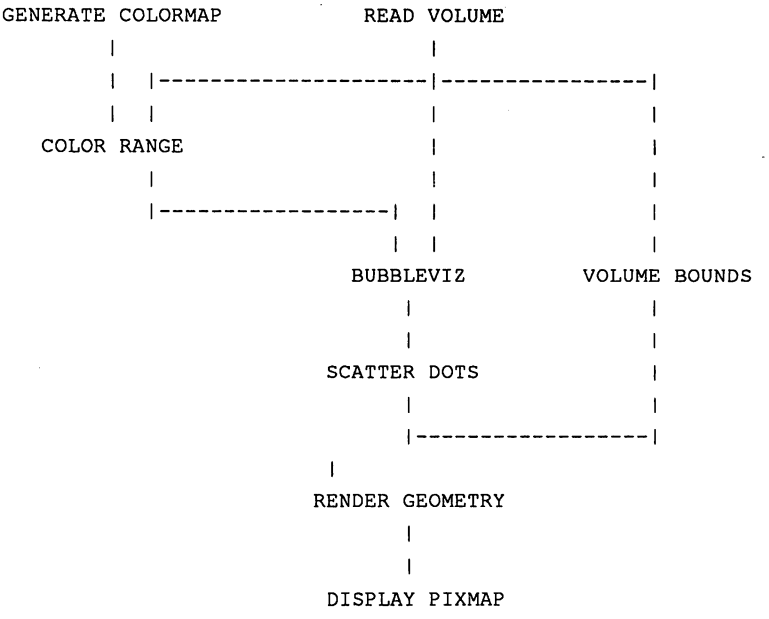
Data Field (field 1D 3-coord 4-vector real)

The output is a list of points in 3D space, with a 4-vector of reals at each point:

- The first element is interpreted as the sphere's radius. If the radius value is 0.0, no sphere is generated as output. If the radius value is 1.0, the sphere's radius will equal the current value of the **Radius** parameter.
- The 2nd–4th elements of the lookup value specify the red-green-blue components of the sphere's color (0.0 = no color; 1.0 = maximum color).

EXAMPLE

A typical network using this module looks like this:



Note that the list of points generated by the **bubbleviz** module is converted to a geometry by the **scatter dots** module.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume

Modules that could be used in place of **bubbleviz**:

colorizer
gradient shade

Modules that can process **bubbleviz** output:

scatter dots

LIMITATIONS

The **bubbleviz** module can generate extremely large databases (one sphere per voxel for volume data). Use 0.0 values in the last field of the input colormap ("opacity" field) to eliminate unnecessary data.

SEE ALSO

The example script BUBBLEVIZ demonstrates the **bubbleviz** module.

NAME

cfd values - calculate values for a field containing read plot3D data

SUMMARY

Name cfd values

Unsupported this module is in the unsupported library

Type filter

Inputs field 1D, 2D, or 3D irregular 5-vector float

Outputs field 1- to 12-vector irregular same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Gamma	float	1.4	1	5
Gas Const	float	1	0	5
Value	choice	*all*		
vector length	integer	12	1	12

DESCRIPTION

cfd values takes the 5 vector irregular field, which **read plot3D** outputs, and derives 7 additional values for each point in the field. Thus, **cfd values** outputs a field of the same type as its input field, but with a vector of up to 12 values at each field location. Note that the input field must have a 5-vector at each location.

The field that **cfd values** receives from **read plot3D** has the following 5 values: density, X momentum, Y momentum, Z momentum, and stagnation.

From the these 5 values **cfd values** computes 7 new values: energy, pressure, enthalpy, mach number, temperature, total pressure, total temp. The gamma constant(γ) and the gas constant (R) are user controllable parameters, and the following variables are defined:

$U_1 = density$

$U_2 = x\ momentum$

$U_3 = y\ momentum$

$U_4 = z\ momentum$

$U_5 = stagnation$

The equations used to derive the new values are as follows:

$$\text{energy (E)} = \frac{U_5}{U_1}$$

$$\text{pressure (p)} = (\gamma - 1)\left[U_5 - \frac{1}{2}\frac{(U_2^2 + U_3^2 + U_4^2)}{U_1}\right]$$

$$\text{enthalpy} = \frac{p}{U_1}$$

$$\text{mach number (M)} = \frac{(U_2^2 + U_3^2 + U_4^2)^{1/2}}{cU_1}$$

$$\text{temperature (T)} = \frac{p}{(U_1 R)}$$

$$\text{total pressure } (p_0) = p\left(1 + \frac{\gamma - 1}{2}M^2\right)^{\frac{\gamma}{\gamma - 1}}$$

$$\text{total temp } (T_0) = T\left(1 + \frac{\gamma - 1}{2}M^2\right)$$

Note that, in calculating the 7 derived quantities, **cfd values** uses the same assumptions about the non-dimensionality, or normalization, of data that the National Aeronautics and Space Administration's PLOT3D, and the **read plot3D** module themselves use.

cfd values displays a set of buttons for specifying which values to include in its output field. To specify the number of values in the output field, first select the desired number of values using the "vector length" parameter. Then, pick which values to include; **cfd values** will output when you have chosen vector length elements. Note that, **cfd values**, actually only computes the values required by your selections.

INPUTS

Data Field (required; field 1D, 2D, or 3D irregular 5-vector float)

cfd values receives its input field from the module **read plot3d**. This is a 1D, 2D, or 3D irregular field, with a vector of 3 to 5 values at each field location.

PARAMETERS

Gamma A floating point value between 1 and 5, which determines the value of the (γ) constant. The formulas assume an ideal gas with a constant ratio of specific heats, (γ). The default value is 1.4.

Gas Constant

A floating point value between 0 and 5, which determines the value of the gas constant. The default value is 1.

Value A list of 12 buttons, displaying the names of the values that **cfd values** computes. To specify that a specific value should be included in **cfd values**'s output field, click on the value's button. The field output by **cfd values** can have between 1 and 12 values at each field location.

vector length

An integer dial, which specifies the number of data values at each location in the field **cfd values** outputs.

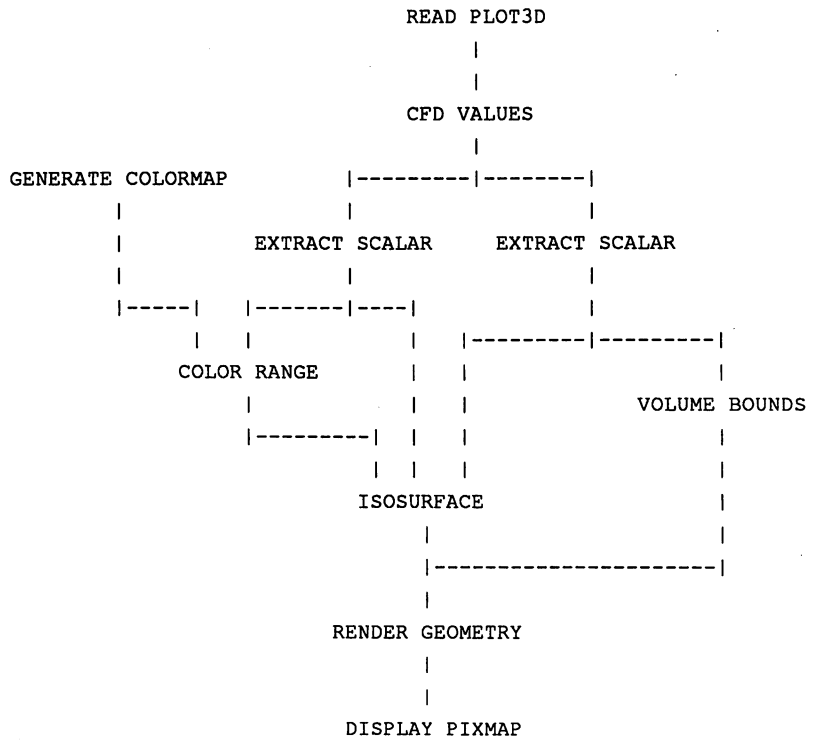
OUTPUTS

Output Field (field 1- to 12-vector irregular same type as input)

The output field is the same type as the input data field. However, the **cfd values** module computes up to 7 new values for each field location. Thus, the output may have a vector of between 1 and 12 values at every point in the field.

EXAMPLE

The following example shows how **cfd values** and **read plot3d** can be used. The **extract scalar** on the right extracts one value from the 12-vector that **cfd values** outputs. **isosurface** computes the isosurface for this scalar output, and **volume bounds** is used to draw a bounding box for the data. The left hand **extract scalar** module extracts another value from **cfd values** output. This second scalar field is used to color the isosurface. The **color range** module is used to scale the colormap to the range of the extracted cfd value. This network will allow you, for example, to generate an isosurface of the density in a field, and then color this isosurface based on the temperature values at each point on the isosurface.



RELATED MODULES

Modules that could provide the **Data Field** input:

read plot3D

Modules that can process **cfd values**'s output:

isosurface
orthogonal slicer
hedgehog
bubbleviz
tracer

REFERENCES

Pieter Buening, **PLOT3D Reference Manual.**

SEE ALSO

The example scripts READ PLOT3D and CFD VALUES demonstrate the **read plot3D** module.

NAME

character string - send a user-entered string to one or more module(s) string parameter port(s)

SUMMARY

Name	character string		
Type	data		
Inputs	none		
Outputs	string		
Parameters	*Name*	*Type*	*Default*
	character	typein	off

DESCRIPTION

The **character string** module sends a single user-specified string to one or more string parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control string parameter input to more than one module using only a single string input widget.

Before you can connect **character string** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A blue-green (teal) parameter port should appear on the module icon. Connect this parameter port to the **character string** module icon in the usual way one connects modules.

Note that the module **file browser** is functionally equivalent to **character string**. They both allow you to send strings to one or more other modules. Conceptually, however, the strings sent by **file browser** will tend to be filenames. While those sent by **character string** can be filenames, they are not limited to these.

PARAMETERS

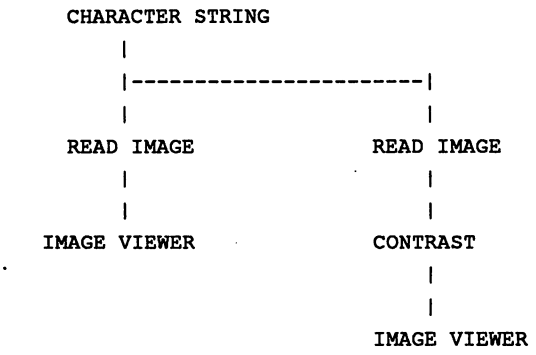
character string (string)
: The single string, specified through a string typein widget, to be sent to the receiving module(s) filename string parameter port(s). The default value is NULL.

OUTPUTS

string (string)
: The string value is sent to all modules with string-type parameter ports that are connected to the **character string** module.

EXAMPLE 1

The following network shows (a somewhat contrived) example of how the **character string** module can be used to send a string constant to two different modules:



RELATED MODULES

Modules that can process **character string**'s output:
all modules with string-type parameter ports

SEE ALSO

The DEMO script cli.scr demonstrates the **character string** module.

NAME

clamp – restrict values in data field to user-specified range

SUMMARY

Name clamp

Type filter

Inputs field *any-dimension any-data any-coordinates*

Outputs field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
clamp_min	float	0.0	none	none
clamp_max	float	255.0	none	none

DESCRIPTION

The **clamp** module transforms the values of a field as follows:

- Any value less than the value of the **clamp_min** parameter is set to **clamp_min**.
- Any value greater than the value of the **clamp_max** parameter is set to **clamp_max**.
- All values within the **clamp_min**-to-**clamp_max** range are not changed.

After being **clamp**'ed, a data set's values are all in this range:

clamp_min ≤ *value* ≤ **clamp_max**

If appropriate, **clamp** also changes the values of the **min_val** and **max_val** attributes of the output field in accordance with the **clamp_min** and **clamp_max** values. **clamp** works with uniform, rectilinear and irregular fields, whether they are vector or scalar.

The **statistics** module can be used to determine the **min_val** and **max_val** of the input field, so you can know what range is reasonable to clamp to.

Note the difference between the **clamp** and **threshold** modules:

- **threshold** sets values outside the specified range to be zero.
- **clamp** sets values outside the specified range to be the range's minimum and maximum values.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field. It may be uniform, rectilinear or irregular; and either vector or scalar.

PARAMETERS

clamp_min
A floating-point number that specifies the minimum output value.

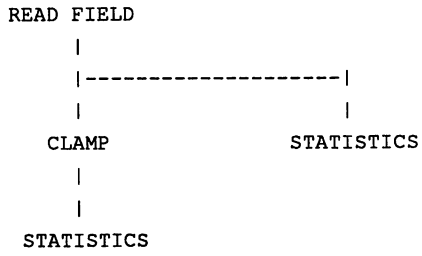
clamp_max
A floating-point number that specifies the maximum output value.

OUTPUTS

Data Field (field *same-dimension same-vectorsame-data same-coordinates*)
The output field has the same dimensionality and type as the input field.

EXAMPLE

The following network reads in an AVS field. The **statistics** module is used to display the field contents with and without clamping:



RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
any other filter module

Modules that could be used in place of **clamp**:

threshold

Modules that can process **clamp** output:

colorizer
any other filter module

Modules that tell you the range of data in the field:

statistics
print field
generate histogram

SEE ALSO

The example script CLAMP demonstrates the **clamp** module.

NAME

color range – scale AVS colormap to the range of data in a field

SUMMARY

Name	color range
Type	data
Inputs	field *any-dimension* scalar *any-data any-coordinates* colormap MODIFIES INPUT
Outputs	colormap
Parameters	*none*

DESCRIPTION

color range adjusts the minimum and maximum values of a colormap to those of an AVS field, thus normalizing the colormap to the range of the data in the field. To do this, **color range** examines a scalar AVS field to see if the minimum and maximum data values are specified in the field's data structure. If they are not, it calculates the minimum and maximum values and stores them in the field's data structure. In both cases, **color range** also stores the minimum and maximum data values into its output AVS colormap data structure.

Use **color range** whenever you have data that you want represented as colors, but that data's range of values is either not evenly distributed between 0 and 255, or much of the data values lie outside the 0 to 255 range.

For example, your input field contains floating point values between the range 0 and 1. If you were to give this range of data values to one of the modules that produces colors from numbers (e.g., **arbitrary slicer** or **field to mesh**) all of the numbers would map to the same color. Because data coloring is done by using a byte value 0-255 to index into the AVS colormap, all of these floating point values would map to the number 1, and hence to the same color. In the default colormap this is the same blue.

Similarly, if you have data that lies in the range -55 to +500, all values outside the range 0-255 will be "clamped" to the two boundary values and visual information about the data's true character will be lost.

Applying **color range** between the output of the **generate colormap** module and a scalar version of your data field stores the range of your data values into the colormap data structure. Modules downstream can use these minimum and maximum values to scale their index into the colormap intelligently. A narrow range of data values will be made to "fan out" across the whole colormap. A wide range of data values will be scaled to fit within the 0-255 range without clipping outlying values. Note, however, that this desirable effect does *not* occur just because **color range** is in the network; it occurs because the downstream modules that receive the modified colormap data structure have been written to make intelligent use of the new minimum/maximum values **color range** generates.

INPUTS

Data Field (required; field *any-dimension* scalar *any-data any-coordinates*)

This is the AVS field whose field data structure will be scanned to see if it already contains minimum and maximum data values. If it does, these data values will be stored into the output colormap data structure. If it does not, **color range** calculates the minimum and maximum values and stores them into both the original AVS field's data structure and the output colormap. Because **color range** can modify the original AVS field, data passing through this module is not shared.

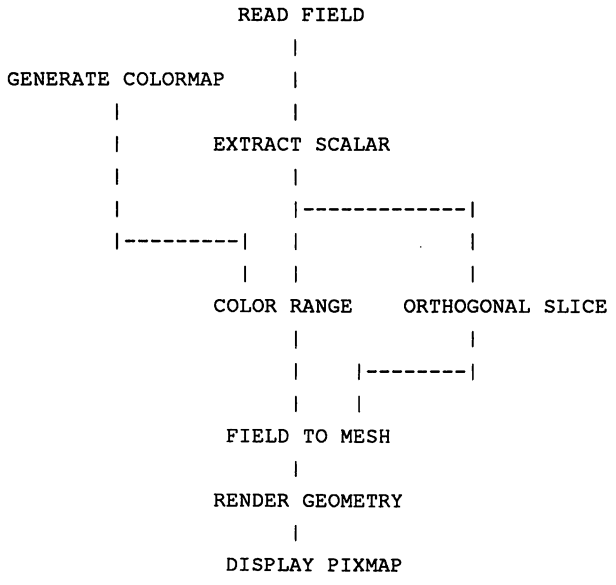
Color Map (required; colormap)
This is the original AVS colormap. Any minimum or maximum values that may have been set in the input colormap are ignored.

OUTPUTS

Color Map (colormap)
The output from **color range** is a new colormap containing the calculated (or transferred from the input field data structure) minimum/maximum data values.

EXAMPLE

The following network reads in a 3-vector field, i.e. every field location has 3 values associated with it. The **extract scalar** module selects one of the fields values. **color range** stores the field's min and max values so that the colormap can be scaled to the range of data in the field:



RELATED MODULES

Modules that could provide the **Data Field** input:

read field
extract scalar (for fields with vectors)

Modules that could provide the **Color Map** input:

generate colormap

Modules that can process **color range** output:

arbitrary slicer
bubbleviz
colorize
field legend
field to mesh
isosurface
probe

SEE ALSO

The example script COLOR RANGE demonstrates the **color range** module.

NAME

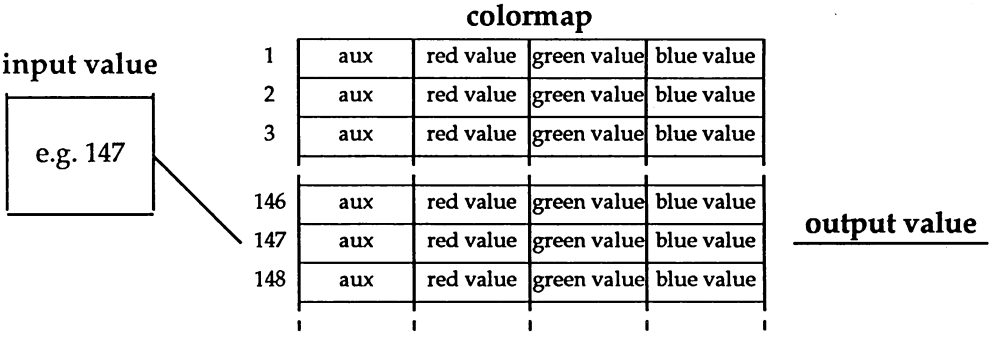
colorizer – convert field of data values to color values

SUMMARY

Name	colorizer
Type	filter
Inputs	field *any-dimension* scalar *any-data any-coordinates* colormap
Outputs	field *any-dimension* 4-vector byte *any-coordinates*
Parameters	*none*

DESCRIPTION

The **colorizer** module converts the data at each point of a scalar field from the input value (which can be any data type) to a *color* (4-vector of bytes). The conversion is accomplished by using the input value as an index into a *colormap*:



colorizer accepts field of any type (byte, integer, real, double). However, the field of colors output by **colorizer** contains only byte data.

INPUTS

Data Field (required; field *any-dimension* scalar *any-coordinates*)

The principal input data for the **colorizer** module is a field, which can be of any dimensionality. The data at each point of the field may be of any data type.

Color Map (optional; colormap)

The optional colormap may be of any size, but any entries beyond the 256th are unused. **Default:** If this input is omitted, a gray-scale colormap is used (lo-value = black; hi-value = white).

OUTPUTS

Field of Colors (field *any-dimension* 4-vector byte *any-coordinates*)

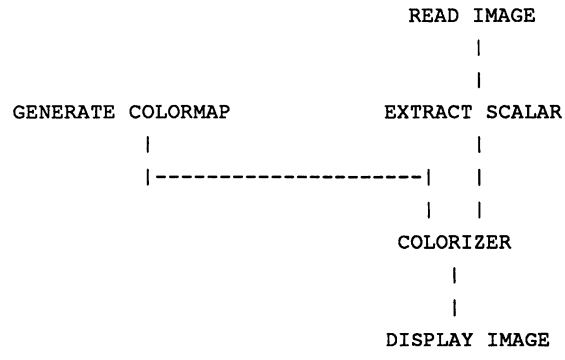
Each input value is transformed into a color value, which is structured as four bytes, as illustrated above. The red, green, and blue bytes specify a true-color pixel value. The *auxiliary* byte is typically used to specify an opacity value (lo-value = completely transparent; hi-value = completely opaque).

The dimensionality of the output field is the same as that of the input field. For byte input, the output field is four times as large as the input field, since each byte (8 bits) is converted to a color value (32 bits).

The **min_val** and **max_val** attributes of the output field are invalidated. The dimensions of the 4-vector output data are assigned the labels "Alpha", "Red", "Green", and "Blue".

EXAMPLE

The following network reads in an AVS image, which is a 2D field of 4-vector bytes. **extract scalar** takes one of the bytes, generating a 2D field with a single byte at each location. These bytes are then translated back into colors by **colorizer**:



RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
field to byte

Modules that could provide the **Color Map** input:

generate colormap

Modules that could be used in place of **colorizer**:

arbitrary slicer

Modules that can process **colorizer** output:

alpha blend
gradient shade
display image
tracer

SEE ALSO

Many of the AVS example scripts demonstrate the **colorizer** module.

NAME

colormap manager – share colormaps among subnetworks

SUMMARY

Name	colormap manager	
Unsupported	this module is in the unsupported library	
Type	data	
Inputs	none	
Outputs	colormap	
Parameters	*Name*	*Type*
	Colormap Manager	colormap
	Colormap Choices	choice

DESCRIPTION

The **colormap manager** module produces an AVS *colormap* data structure, for use by modules that transform input data into color values. These modules include:

colorizer
arbitrary slicer
bubbleviz
field to mesh
isosurface

colormap manager works exactly like **generate colormap**, with one exception: separate active subnetworks, each with its own **colormap manager** module, share a single "pool" of colormaps.

A menu of all the active colormaps appears in a choice menu below each *colormap manager*'s editing widget. All the menus have the same entries — different maps can be selected in different managers.

PARAMETERS

Colormap Manager
: A colormap generator widget. See the **generate colormap** manual page for details on using this widget.

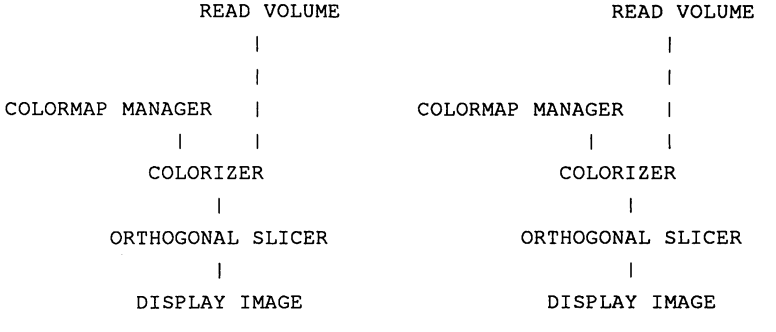
Colormap Choices
: A set of choices, listing each of the currently active colormaps.

OUTPUTS

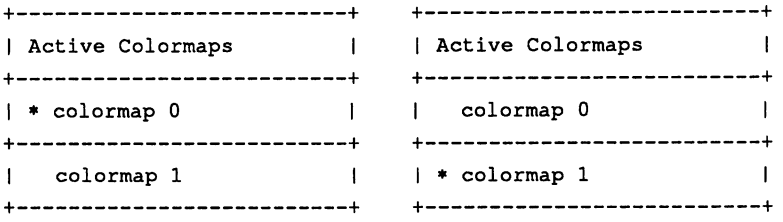
colormap The output is an AVS colormap.

EXAMPLE

Suppose the following two subnetworks are active, created to slice through two different databases:

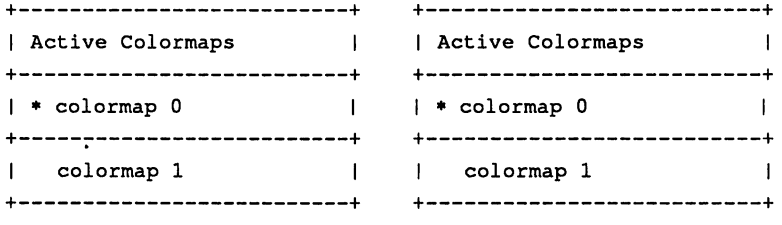


Each **colormap manager** module has its own *colormap editor* control widget. Below the two colormap editors are two choice menus:



The same "pool" of colormaps is shown in each menu, but a different colormap is currently selected for each subnetwork.

By default, each new **colormap manager** that is instantiated from the module Palette has it's own unique colormap editor. You can click on the **colormap 0** button for the second subnetwork in order to have both subnetworks use the same colormap:



Now, editing the colormap in *either* **colormap manager** module is reflected in both subnetworks.

You can extend the sharing of colormaps to any number of currently active subnetworks. Each must have its own **colormap manager** module.

NOTE

colormap manager modules are used in both the *Image Viewer* and *Volume Viewer* subsystems.

NAME

combine scalars – combine scalar fields into a vector field

SUMMARY

Name combine scalars

Type filter

Inputs field *any-dimension* scalar *any-data any-coordinates* (channel 0 — optional)
field *any-dimension* scalar *any-data any-coordinates* (channel 1 — optional)
field *any-dimension* scalar *any-data any-coordinates* (channel 2 — optional)
field *any-dimension* scalar *any-data any-coordinates* (channel 3 — optional)

Outputs field *same-dimension* 1D–4D *same-data*

Parameters

Name	Type	Default	Min	Max
Vector Len	Dial	4	1	4

DESCRIPTION

The **combine scalars** module combines up to four fields with scalar data values into a field whose data values are vectors. The input field must be of like dimension and the scalar values must be of the same type.

This module is generally most useful for constructing images or gradient fields from separately computed components.

The inputs ports on this module's Network Editor icon are processed right-to-left: the rightmost port contributes a value to the first element (lowest memory location) of each output vector; the leftmost port contributes a value to the last element (highest memory location) of each output vector.

If the selected scalars have labels and/or units associated with them, those labels will be carried over to the newly constructed vector.

INPUTS

None of the input fields is absolutely required, but at least one of them must be present. If an input field is omitted, zero values are output in the corresponding element of each output vector.

Channel 0 (optional; field *any-dimension* scalar *any-data any-coordinates*) A set of values to be output in the *fourth* dimension of the output vectors.

Channel 1 (optional; field *any-dimension* scalar *any-data any-coordinates*) A set of values to be output in the *third* dimension of the output vectors.

Channel 2 (optional; field *any-dimension* scalar *any-data any-coordinates*) A set of values to be output in the *second* dimension of the output vectors.

Channel 3 (optional; field *any-dimension* scalar *any-data any-coordinates*) A set of values to be output in the *first* dimension of the output vectors.

PARAMETERS

Vector Length

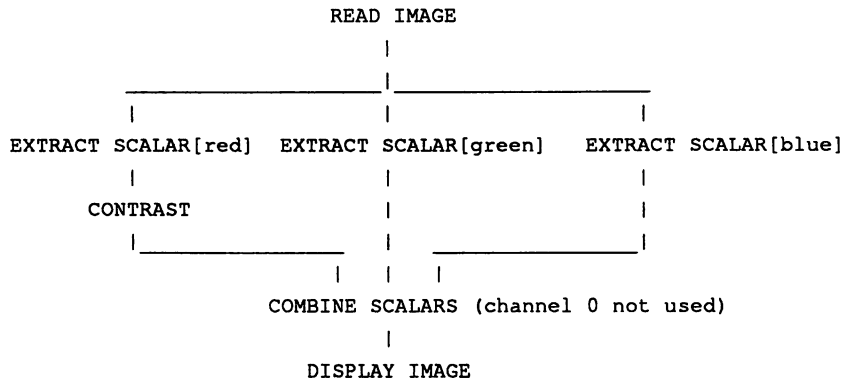
Specifies the dimension of the output vectors — 1 – 4. The number of input ports that appears on the Network Editor icon varies according to the value of this parameter.

OUTPUTS

Field (field *same-dimension* 1D–4D *same-data*) The scalar input streams are assembled into a single output stream consisting of vectors, whose dimension is specified by **Vector Length**. The coordinate type (e.g. uniform, rectilinear, or irregular) of the output field is the same as the rightmost, nonempty input field.

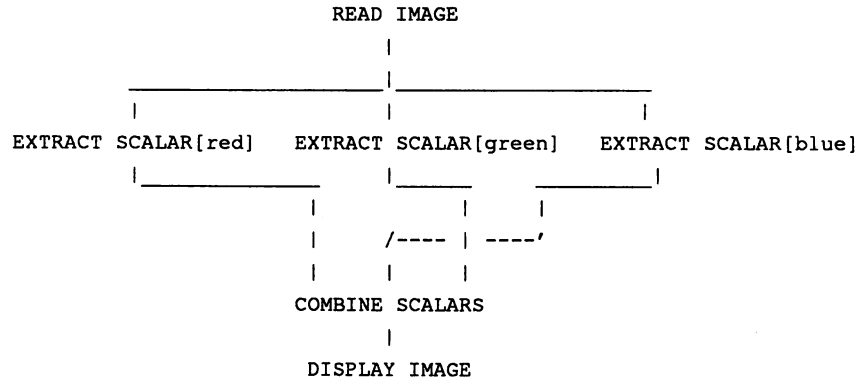
EXAMPLE 1

The following network performs contrast stretching on only the *red* band of an image.



EXAMPLE 2

The following network swaps the *green* and *blue* bands of an image:



RELATED MODULES

extract scalar

SEE ALSO

The example script CONTRAST demonstrates the **combine scalars** module.

NAME

compare field – compare two AVS fields, display and write data difference

SUMMARY

Name	compare field				
Type	data output				
Inputs	field *any-dimension n-vector any-data any-coordinates* field *same-dimension same-vector same-data same-coordinates*				
Outputs	none				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Do Compare	oneshot	off		
	Max Elements	integer	100	-1	1000
	Output File	typein	*/tmp/cfield_...*		

DESCRIPTION

The **compare field** module compares **any** two identically-structured AVS fields. It will print out differences between the headers if they are different. If the headers are the same, it will proceed to do a comparison of the data contents of the two fields. If the fields are not identical in their data components, **compare field** will print the message, "fields are DIFFERENT", to standard output.

The output of the compare is a list of up to **Max Elements** data *differences*. The results of the compare are both displayed in an **Output Browser** widget in the control panel and written to a file.

The Output Browser in which **compare field** displays its output can be resized, like any other widget, using the AVS Layout Editor. For a detailed description of how to do this, see the section titled "Layout Editor," in the chapter "The Network Editor Subsystem" of the *AVS User's Guide.*

compare field was originally written to make sure that two identical modules, one written in C and one written in Fortran, produced the same results. It could also be useful to compare the contents of a field before and after an operation has been performed on it.

INPUT

Input Field 1 (required; field *any-dimension n-vector any-data any-coordinates*)
The input AVS field can be 1, 2, 3, or 4 dimensional; it can be vector or scalar, can contain byte, int, float or double data, and can have uniform, rectilinear, or irregular coordinates.

Input Field 2 (required; field *any-dimension n-vector any-data any-coordinates*)
The second AVS input field must match the first in the number of dimensions (Ndim), the size of each dimension (Dims), the number of coordinate dimensions (Nspace), the vector length (Veclen), the data type (byte, float, double, etc.), and the type of coordinate system (uniform, rectilinear, curvilinear), if a comparison of the two fields' data is to be done.

PARAMETERS

Do Compare
A oneshot "do it now" switch that triggers the actual comparison after both input fields exist.

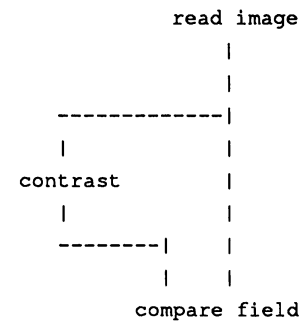
Max Elements
An integer dial that controls how many of the data *differences* to display in the **Output Browser** and write to the output file. The allowable range is -1 (none) to 1000. The default is 100. **compare field** compares the entire fields, until this limit is reached.

Output File

An ASCII typein for specifying the output file. By default, **compare field** writes to a file in the */tmp* directory called *cfield_nnnn* (where nnn is the process id of the **compare field** module. The **Output File** is rewritten whenever any of the other parameters or input files change. Since the Output Browser is limited in size, this output file can be useful to examine directly, using a conventional text editor.

EXAMPLE 1

The following network reads an image into an AVS field. One version of the image goes directly to **compare field**, the other is passed through a **contrast** filter. The "before" and "after" images are compared and the different alpha, red, green, blue values at each pixel are listed.



RELATED MODULES

print field

LIMITATIONS

compare field writes to /tmp by default. This can cause problems if: (1) there is no /tmp mounted on your system, (2) the /tmp directory does not have very much room in it or has inaccessible protections, or (3) the module is being run remotely.

SEE ALSO

The example script COMPARE FIELD demonstrates the **compare field** module.

NAME

composite – blend two images using alpha transparency

SUMMARY

Name	composite
Type	filter
Inputs	field 2D uniform 4-vector byte *(foreground image)* field 2D uniform 4-vector byte *(background image)*
Outputs	field 2D uniform 4-vector byte *(blended image)*
Parameters	none

DESCRIPTION

The **composite** module takes the contents of the foreground image's alpha channel (the image's opacity) and uses it to blend the foreground image over the background image. The equation for this blending is:

red = (Foreground(red) * ALPHA) + (Background(red) * (1.0 - ALPHA))
green = (Foreground(green) * ALPHA) + (Background(green) * (1.0 - ALPHA)
blue = (Foreground(blue) * ALPHA) + (Background(blue) * (1.0 - ALPHA)

where ALPHA is the foreground image's alpha channel byte value. If the two inputs are reversed, the alpha of the new foreground image will be used.

INPUTS

Image (required; field 2D uniform 4-vector byte)
The right input port on the module receives the foreground image.

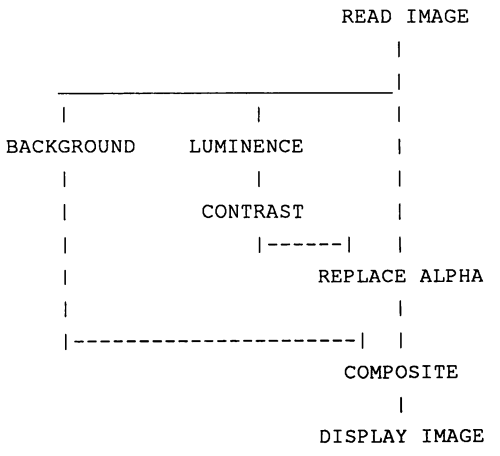
Image (required; field 2D uniform 4-vector byte)
The left input port on the module receives the background image. The size of the background image must be identical to the size of the input image.

OUTPUTS

Image (field 2D uniform 4-vector byte)
The blended image of the two input images.

EXAMPLE 1

The following network reads an image, computes its luminence,(gray scale intensities) uses that to create an alpha mask, generates a shaded background, and composites the rendered image over the shaded background image.



RELATED MODULES

Modules that could provide the foreground **Image** input:

read image
replace alpha

Modules that could provide the background **Image** input:

background

Modules that can process **composite** output:

image viewer
display image

See also **background, luminence, replace alpha, contrast,** and **extract scalar.**

SEE ALSO

The two BACKGROUND example scripts demonstrate the **composite** module.

NAME

compute gradient – compute gradient vectors for 2D or 3D data set

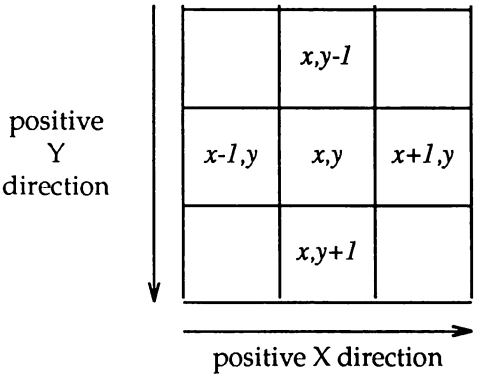
SUMMARY

Name	compute gradient				
Type	filter				
Inputs	field 2D/3D scalar byte *any-coordinates*				
Outputs	field *same-dimension* 3-vector real *same-coordinates*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	2D Height	float	0.5	0.0	1.0

DESCRIPTION

The **compute gradient** module computes the gradient vector at each point in a 2D or 3D field of data. The gradient is can be used (e.g. by **gradient shade**) as a "pseudo surface normal" at each point.

A "nearest neighbor" approach is used to compute the gradient: in each direction, the component of the gradient vector is the difference of the *next* data and the *previous* data. In two dimensions, this can be pictured as follows:



$$\Delta x_{x,y} = data_{x+1,y} - data_{x-1,y}$$
$$\Delta y_{x,y} = data_{x,y+1} - data_{x,y-1}$$
$$\Delta z_{x,y,z} = data_{x,y,z+1} - data_{x,y,z-1} \quad (\text{<— for 3D data})$$
$$\Delta z_{x,y} = 2D\ height \quad (\text{<— for 2D data})$$

INPUTS

Data Field (required; field 2D/3D scalar byte *any-coordinates*) The input field may be either 2D or 3D. The data at each point of the field must be a single byte. The byte values will be interpreted as integers in the range 0..255.

PARAMETERS

2D Height (appears for 2D data only) Supplies the Z-coordinate of the gradient. It can be used the change the apparent height of the surface. A value of 1.0 is generally a very "rough" or "noisy" surface, whereas values approaching 0.0 will show little effect for shading.

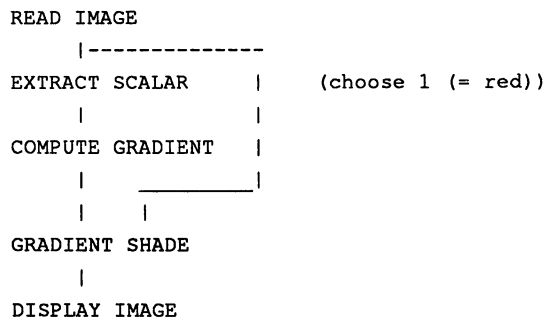
OUTPUTS

Data Field (field *same-dimension* 3-vector real *same-coordinates*)
The output field has the same dimensionality as the input field. For each element, the output data is a 3D vector of reals, representing the 3D gradient.

The **min_val** and **max_val** attributes of the output field are invalidated.

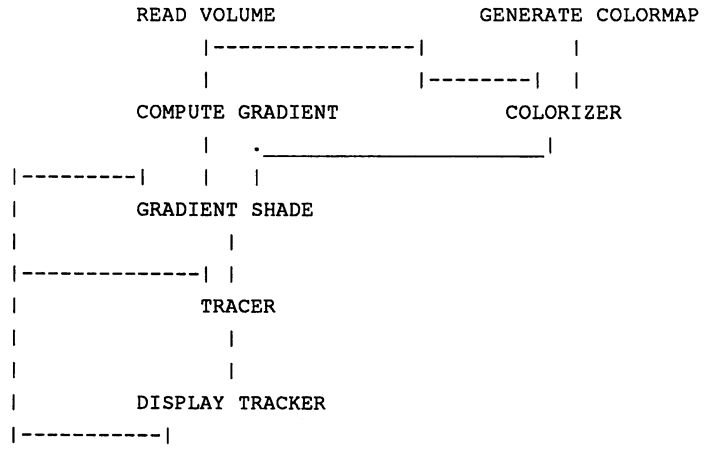
EXAMPLE 1

The following network shades a 2D image:



EXAMPLE 2

The following network shades a 3D image:



RELATED MODULES

gradient shade	
display image	(for two-dimensional data)
alpha blend	(for three-dimensional data)
extract scalar	(to get a single scalar height field from an image)

LIMITATIONS

There may be algorithms better than "nearest-neighbor" for computing the gradient.

This module produces 12 bytes per pixel (voxel). For example, a 128 x 128 x 128 byte volume is about 2.1 MB before the gradient is computed. The **compute gradient** module produces a 25.2 MB internal data set from this data. This will have an adverse performance effect on systems whose physical memory is 32 MB or less.

SEE ALSO

The example scripts ANIMATED FLOAT and HEDGEHOG demonstrate the **compute gradient** module.

NAME

contour to geom – create geometry of 2D or 3D scalar field contour slices

SUMMARY

Name	contour to geom				
Type	mapper				
Inputs	field 2D/3D scalar *any-data any-coordinates*				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	threshold	float dial	128.0	*unbounded*	*unbounded*

DESCRIPTION

The **contour to geom** module finds and creates contour lines of similar value in a scalar field, then outputs the result as an AVS geometry. The contour lines can be disjoint. The **threshold** parameter controls the contour level. **contour to geom** handles 2D and 3D datasets, and uniform, rectilinear, and irregular grids.

INPUTS

Data Field (required; field 2D/3D scalar *any-data any-coordinates*)
The input field is 2D or 3D scalar field, containing any data, using any coordinate system.

PARAMETERS

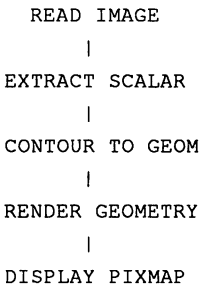
threshold A floating point dial that controls what value the contour lines are created for. The default is 128.0. This parameter is unbounded, with no minimum or maximum.

OUTPUTS

Geometry The contour lines are represented as an AVS geometry.

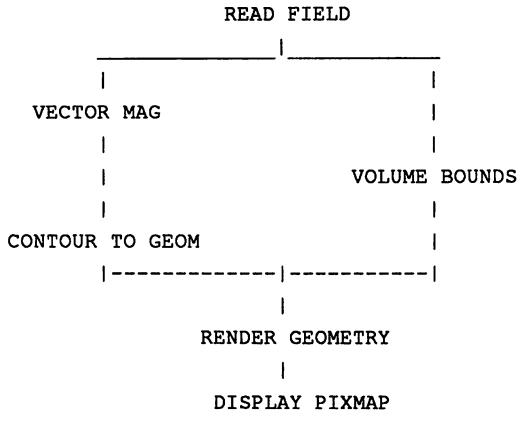
EXAMPLE 1

The following network finds a contour on the red channel of the mandrill.x image.



EXAMPLE 2

The following network finds the magnitude of a vector field and contours it.



RELATED MODULES

Modules that can process **contour to geom** output:

render geometry
render manager

SEE ALSO

Two CONTOUR GEOMETRY example scripts demonstrate the **contour to geom** module.

NAME

contrast – perform linear transformation on range of field values

SUMMARY

Name	contrast				
Type	filter				
Inputs	field *any-dimension n-vector any-data any-coordinates*				
Outputs	field of same type as input				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	cont_in_min	float	0.0	*none*	*none*
	cont_in_max	float	255.0	*none*	*none*
	cont_out_min	float	0.0	*none*	*none*
	cont_out_max	float	255.0	*none*	*none*

DESCRIPTION

The **contrast** module transforms all the values in a field. Two different types of transformation take place:

- **Linear transform:** All values that fall within the "input range" specified by the **cont_in_min** and **cont_in_max** parameters are transformed linearly to the "output range" **cont_out_min**..**cont_out_max**.

$$new\_value = \frac{(cont\_out\_max - cont\_out\_min) * (value - cont\_in\_min)}{cont\_in\_max - cont\_in\_min}$$

  (More precisely, this is an *affine transformation*.) In essence, this transformation "stretches" or "compresses" one specified range of data to fit another specified range.

- All values that fall outside the specified input range are "clamped" to the limit values of the output range.

The **contrast** module typically is used to remove low-level noise from images and volumes, or to increase the contrast in faded images and volumes.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input data may be an AVS field of any dimensionality.

PARAMETERS

cont_in_min
Specifies the bottom of the range of input values that will be transformed linearly.

cont_in_max
Specifies the top of the range of input values that will be transformed linearly.

cont_out_min
Specifies the bottom of the range of output values. All values ≤ **cont_in_min** will be transformed to this value.

cont_out_max
Specifies the top of the range of output values. All values ≥ **cont_in_max** will be transformed to this value.

OUTPUTS

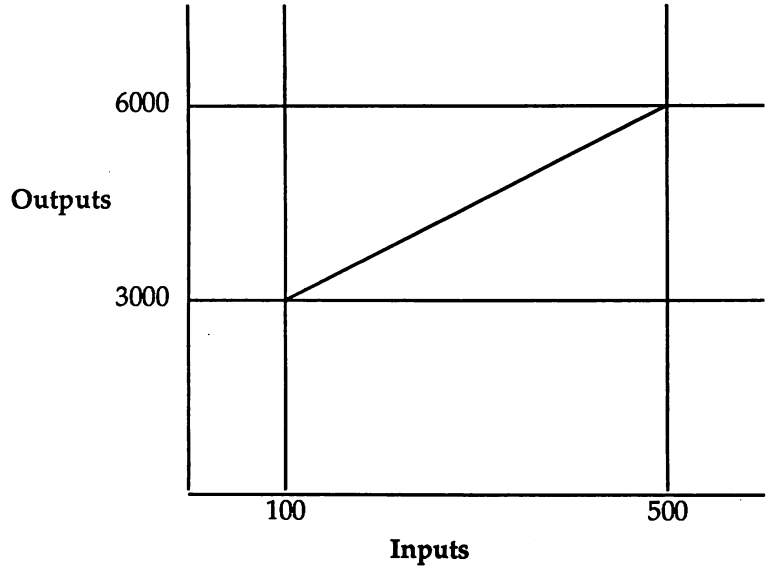
Data Field The output field has the same dimensionality and type as the input field.

If the input field has byte values, appropriate new **min_val** and **max_val** values are written to the output field.

EXAMPLE 1

The following diagram shows how field values are transformed given these parameters:

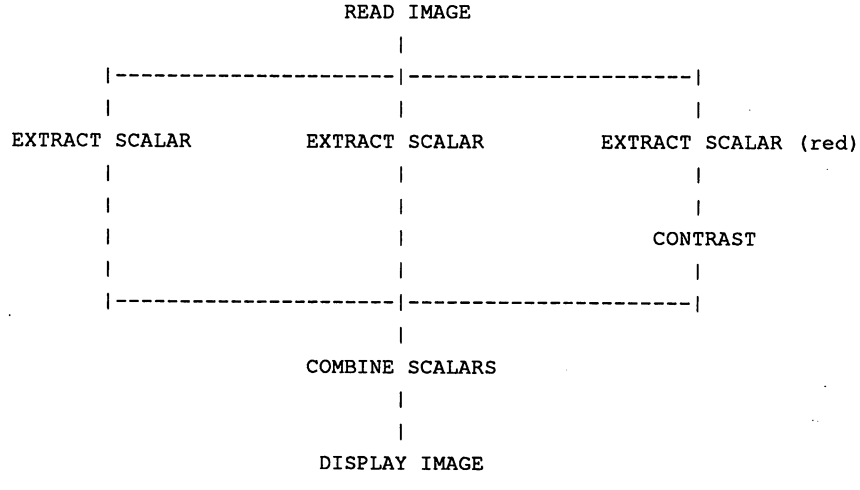
cont_in_min = 100
cont_in_max = 500
cont_out_min = 3000
cont_out_max = 6000



You can use **contrast** to make a negative out of an image by "flipping" the output values (e.g. **cont_out_min** = 255; **cont_out_max** = 0).

EXAMPLE 2

The following network reads in an image, extracts the red, green and blue channels, contrast stretches only the red channel, and then uses **combine scalars** to pack the seperate channels back into an image.



RELATED MODULES

Modules that could provide the **Data Field** input:

read volume

SEE ALSO

The example script CONTRAST demonstrates the **contrast** module.

NAME

convolve - apply a signal processing filter to 2D field

SUMMARY

Name	convolve		
Type	filter		
Inputs	field 2D n-vector *any-data any-coordinates* (image) field 2D scalar float uniform(convolution filter)		
Outputs	field of same type as input		
Parameters	*Name*	*Type*	*Default*
	normalize	boolean	on

DESCRIPTION

convolve takes a signal processing filter and applies it to a source field to produce a destination image. Typically, the source and destination fields will be AVS *images*, but they might also be 2D slices of 3D fields. Filters can be produced by the module **generate filters** or by user-written modules.

Convolution is a frequently used technique in signal and image processing. Applying a "high pass" filter, such as a Laplacian, to an image will emphasize edges in the image. On the other hand, a "low pass" filter, such as a Gaussian, will smooth images. These techniques can be helpful in removing artifacts from images, and in compensating for the inherently discrete nature of digital data.

The filter must be a 2D array of floating-point values. The source field must also be 2D, but it can hold any size vector of any data-type. The field output by **convolve** will be the same type as the source field. The filter must be smaller than the field it is being applied to. **convolve** typically normalizes filters to the range 0 to 1 before applying them to an image.

Filters are applied as follows, taking a typical case in which a small, 10x10 filter is applied to a larger, 256x256 image: One can imagine the filter sitting on top of the source image centered on one pixel in the image, say (45,45). Each of the 100 values in the filter array is multiplied by the value of the pixel *beneath* it. These 100 products are then added together, and their sum becomes the value of the pixel at (45,45) in the destination image. Then the filter is shifted so that it is centered over the next pixel. This process is repeated to produce each element in the output image.

This approach is known as the "sliding window" method. It is an N x M algorithm, where N is the number of elements in the convolution filter and M is the number of elements in the image. As a result, it is recommended that filters be small; larger filters (i.e. above 12x12) require a great deal of computation.

convolve accepts data of any type. In the case of an image, which is a 2D field of vectors each containing 4 bytes, **convolve** disregards the alpha bytes and separates the red, green and blue bytes. Then it applies the filter separately to each color field, before reassembling the bytes into image format. In the case of non-image data, for example a 2D field of 5-vector floats, **convolve** handles one component of the vector at a time. All data-types are converted to floats during computation and then converted back in **convolve**'s output.

To avoid edge effects, a border around the perimeter of the source field is not convolved. The border's width is half the width of the filter.

INPUTS

Data Field (required; field 2D n-vector *any-data any-coordinates*)

A 2D AVS field, typically an image, to be convolved. The field is input through the right input port.

Filter Field (required; field 2D scalar float uniform)

A 2D AVS field of floating-point scalar values. Filters can be created by using the module **generate filters**, which produces Gaussian, Laplacian, and other filters. Alternately, you can write your own modules to generate filters. Filters are input through the left input port.

PARAMETERS

normalize (toggle)

If **normalize** is selected, filters are normalized to the range 0 to 1 before they are convolved with the input field.

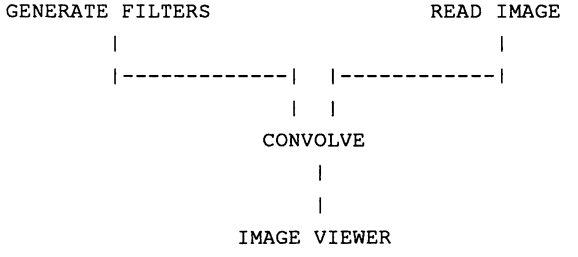
OUTPUTS

Output Field

The output field is the same type as the input data field.

EXAMPLE

The following network reads in an image and a filter, convolves the two, and displays the resulting image:



RELATED MODULES

Modules that could provide the **Data Field** input:

read image
pixmap to image
orthogonal slicer
any other module which outputs a 2D field

Modules that could provide the **Filter** input:

generate filters
any (user written) module which outputs a 2D scalar float field

Modules that can process **convolve** output:

display image
image viewer
any other module which takes a 2D field as input

SEE ALSO

The example scripts CONTRAST, GENERATE FILTERS, and SOBEL demonstrate the **convolve** module.

NAME

crop – extract subset of elements from a field

SUMMARY

Name crop

Type filter

Inputs field *any-dimension any-data any-coordinates*

Outputs field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
min x	int	1st indx	1st indx	last indx
max x	int	last indx	1st indx	last indx
min y	int	1st indx	1st indx	last indx
max y	int	last indx	1st indx	last indx
min z	int	1st indx	1st indx	last indx
max z	int	last indx	1st indx	last indx

DESCRIPTION

The **crop** module changes the size of a field by extracting the data within a specified range of elements. This process is analogous to "cropping" a photographic image.

This module is useful for subsampling the data without changing it (e.g. by interpolation). It preserves the resolution of the data, but may change its aspect ratio. Typical uses are to eliminate uninteresting portions of the data and to increase processing speed by reducing the amount of data.

Once a field is input to **crop**, the module's min and max dials are set to the min and max indices of the field's data array. From then on the dials cannot be turned lower than the min index, or higher than the max index. If you use the Dial Editor to change these values they would "snap back" to their original values. This makes sense, because you can only take a subset from the field within the field's array indices.

If the value of the min dial is set greater than the value of the max dial, the two dials will swap values. In addition, the min and max dial cannot be set to the same value.

INPUTS

Data Field (required; field *any-dimension any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

Note that the parameters indicate *positions* of elements in the field — they have nothing to do with the *values* of field elements.

min x Specifies the lower bound array index in the field's first dimension.

max x Specifies the upper bound array index in the field's first dimension.

min y Specifies the lower bound array index in the field's second dimension.

max y Specifies the upper bound array index in the field's second dimension.

min z (Does not appear for 2D input data sets) Specifies the lower bound array index in the field's third dimension.

max z (Does not appear for 2D input data sets) Specifies the upper bound array index in the field's third dimension.

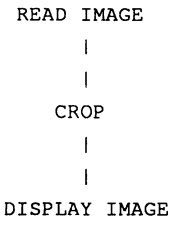
OUTPUTS

Data Field The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced.

The **min_val** and **max_val** attributes of the output field are invalidated. Appropriate new **min_ext** and **max_ext** values are written to the output field.

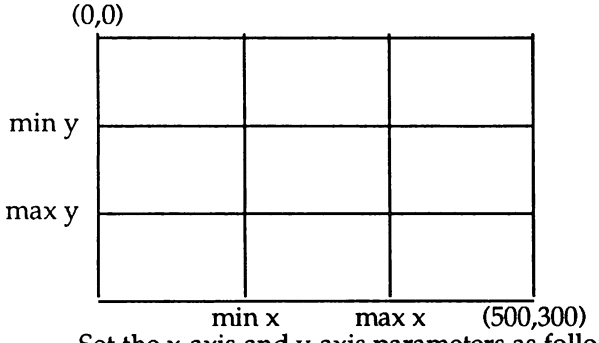
EXAMPLE 1

The following network reads a 2D field (image), crops it and displays the result:



EXAMPLE 2

Suppose you want to process the middle third of a field that contains an 500x300 pixel image:



Set the x-axis and y-axis parameters as follows:

min x = 167
max x = 333
min y = 100
max y = 200

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
filter modules

Modules that could be used in place of **crop**:

downsize
interpolate

Modules that can process **crop** output:

colorizer
gradient shade
arbitrary slicer
orthogonal slicer
any other filter module

LIMITATIONS

crop works for 2D and 3D data sets only.

SEE ALSO

The example script CROP demonstrates the **crop** module.

NAME

display image – show image in a display window

SUMMARY

Name display image

Type data output

Inputs field 2D 4-vector byte uniform

Outputs *none*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Magnification	choice	x1	x1	x16
Automag_Size *(internal)*	integer	256	50	1024
Max Image Dimension *(internal)*	integer	1280	100	4096
Dither	choice	dither		

DESCRIPTION

The **display image** modules takes an input image and displays it in a display window. This window has a pulldown menu, accessed via the small square in the window's title bar. The menu allows you to control image magnification, window resizing, and other options relating to the display window.

When the image is larger than the display window, you can scroll it with the mouse, either by "dragging" the image itself or by using horizontal and vertical scrollbars.

You can resize the display window manually, using the X Window System window manager. You can also have the window resize itself automatically, in response to a change in the image contents or a magnification selected from the display window's pulldown menu.

Note that when running avs as a remote client on a pseudocolor X terminal, **display image** has an additional choice parameter for selecting the "dithering" method. This parameter does not show up when you run avs on standard Stardent ST 1000/1500/2000/3000 systems. For details about running avs on an X server, and dithering colors on pseudocolor machines, see the discussion on "Using AVS on PseudoColor X Servers" in the manual page for avs, and in the "Starting AVS" chapter of the *AVS User's Guide.*

Note that the **display image** window can be reparented to page and stack widgets using the AVS Layout Editor.

INPUTS

Data Field (required; field 2D 4-vector byte uniform)
The input field must be in the AVS *image* format.

PARAMETERS

Magnification
A choice to specify a power of 2 (1,2,4,8,16) by which to multiply each dimension of the image.

Automag_Size
(for internal use only) This is used as a communications port to handle resizing of the image. Do not change this parameter.

Maximum Image Dimension
(for internal use only) This parameter is no longer used in AVS3. It has been kept solely for the purpose of backward compatibility. See description below.

Dither (only appears when running avs on pseudocolor X terminals)
A choice of four dithering methods. These improve the appearance of color graphics displayed on pseudocolor terminals.

- **dither** uses an internal dither mask to simulate colors that are "between" the colors actually available on a pseudocolor terminal.
- **random** uses an randomly generated dither mask to simulate colors that are "between" the colors actually available on a pseudocolor terminal.
- **monochrome** computes the luminence of the colors in the input image, by combining the red, green, and blue values for each point, according to a linear relation. The luminence values are then used to find a greyscale equivalent for each pixel. Selecting **monochrome** converts the color image into a monochrome image, resembling a black and white photograph.
- **none** each color in the input image is approximated by the closest color in the spectrum of colors actually available on a pseudocolor terminal.

MAGNIFICATION

You can magnify an image for closer examination, although the magnified image will provide no new detail. Magnification is implemented by duplicating the pixels in the original image. The result is "blockier" but provides a closer look at the image. There are several magnification levels (x1,x2,x4,x8,x16) in the pulldown menu, with the current magnification marked as (*selected*).

Since **display image** now only requests X window resources for the actual displayed window area, the **Maximum Image Dimension** parameter is no longer used.

RESIZING

The display window can be resized in several ways. You can use the X window manager's *resize* window operation to enlarge or shrink the display window. An approximate image magnification is automatically chosen that makes the image at least as large as the window. (This is now only done if the ImageAutomagnify *.avsrc* option is enabled). For a more detailed description of *.avsrc* options, see the **avs** man page.

The ImageAutomagnify parameter reenables the automatic magnification of the image to at least fill the window when the window is resized, as was the default behavior in AVS2. By default, this is disabled, because the combination of autofit and automagnification can produce unexpected window behavior.

Also see the **image viewer** module, which has continuous scale magnification.

The pulldown menu also provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Resize to Fit Image.** Resizes the window to fit the image exactly at the current magnification. (The maximum size window is the full screen window described above.) As with **Zoom Full Screen**, an embedded display window becomes a top-level window.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen** or a **Resize to Fit Image**. If the window originally was embedded in a page or stack, it will be re-embedded there.

- **AutoFit - Turn On/Off.** This toggle switch controls the automatic fitting of the display window size to its image. When this feature is enabled (the default), **display image** automatically resizes the display window whenever the image size changes. This can occur when you select a new magnification or when an entirely new image is input to **display image**. The new display window size exactly fits the new image size (unless the window is currently embedded in a page or stack).

SCROLLING

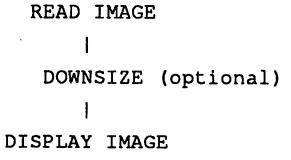
Whenever the image is larger than the display window, only a portion of the image is visible. You can "pan" over the entire image in two ways:

- Using the horizontal and vertical scrollbars that automatically appear. These scrollbars work the same way as those on File Browser windows.
- By dragging the image itself. Place the mouse cursor anywhere in the image, click and hold down any mouse button, and drag the mouse. The image moves continuously, and the scrollbars are updated when you release the mouse button. The image automatically stops scrolling when it hits its borders.

The **Scrollbars – Turn On/Off** selection on the pulldown menu allows you to disable or reenable the appearance of scrollbars along the right and bottom edges of the display window. (The "drag-the-image" method is always enabled.) You may want to suppress the scrollbars to reduce distraction or to provide additional viewing space.

The **ImageScrollbars** parameter in the AVS startup file (see Chapter 2) determines whether image windows get scrollbars by default when they contain oversize images. If you do not use this startup parameter, scrollbars are initially enabled.

EXAMPLE



RELATED MODULES

display pixmap image viewer

SEE ALSO

The example scripts ANIMATED INTEGER, FIELD MATH, and GENERATE FILTERS as well as others demonstrate the **display image** module.

NAME

display pixmap – show pixmap in a display window

SUMMARY

Name	display pixmap			
Type	data output			
Inputs	pixmap			
Outputs	*none*			
Parameters	*Name*	*Type*	*Default*	*Choices*
	Store Frames	toggle		
	Append Frame	oneshot		
	Delete Current	oneshot		
	Replay	choice	Off	Continuous Bounce Off
	Current Frame	integer		
	Max Frames	integer		
	Replace Speed	integer		
	Save Image	string		

DESCRIPTION

The **display pixmap** module displays its input pixmap in a display window. It automatically sizes the pixmap to fit the window.

display pixmap is most frequently used in conjunction with the **render geometry** module to display geometry output.

In addition, you can:

- Save the pixmap as an AVS image in a file.
- Create and play back a "flipbook" of consecutive images.

These capabilities are invoked using the module's input parameters, as described in the sections below.

Note that the **display pixmap** window can be reparented to page and stack widgets using the AVS Layout Editor.

INPUTS

pixmap The input data must be an AVS *pixmap*, typically created by the **render geometry** module.

PARAMETERS

The following parameters control a pixmap-animation capability. Note that this is independent of the animation facility in the Geometry Viewer (**render geometry** module), and works somewhat differently. See the *ANIMATION* section below for more information.

Store Frames
This toggle controls whether all new frames are automatically added to the animation sequence.

AppendFrames
Explicitly adds the currently displayed pixmap to the animation sequence. (Use when **Store Frames** is off.)

Delete Current
Deletes the currently displayed pixmap from the animation sequence.

Replay This choice widget controls how the animation sequence is to be played back: The choices are **Continuous, Bounce,** and **Off.**

Current Frame
The number of the current frame in the animation sequence (first frame = 0). This field is a typein — change the number to jump directly to another frame.

Max Frames
A typein field that specifies the ceiling for the number of frames that you can place in an animation sequence.

Replay Speed
Controls the rate at which an animation is played back. The larger the value, the greater the delay between frames.

Save Image
This is a typein field. If you type a filename or pathname into this field, the current pixmap is written to a file when you press **Return.**

RESIZING

display pixmap's pulldown menu, which is accessed by clicking on the "dimple" in the upper lefthand corner of the display window, provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen.** If the window originally was embedded in a page or stack, it will be re-embedded there.

SAVING AN IMAGE

To save an image in a file, type the filename as the value of the **Save Image** parameter. When you press **Return,** the file is created. To save another image under the same name, you can move the mouse cursor to the **Save Image** input area and press **Return** again.

ANIMATION

By changing the input data or by adjusting the parameters of upstream modules (e.g. **transform pixmap**), you can have the **display pixmap** window show a sequence of images. You can create an animation ("flip book") by designating certain images to be "frames". Then, you can play back the images, adjusting the speed with a control widget.

Because each of the images in a flip book takes up a significant amount of system memory, there is a *Max Frames* parameter. Be sure that its value is low enough so that your system can comfortably keep all of the images in memory at the same time. AVS requires roughly 4 bytes of memory per pixel of each your image. The larger the display window, the greater the memory requirements.

There are two ways to create a flip book:

- To save *all* the images that appear in the window (actually, just the last *Max Frames* that are produced — see below), turn on the **Store Frames** toggle. As each image is drawn, it will be appended to the end of the flip book. If *Max Frames* images have already been saved, this new pixmap will replace the oldest pixmap in the cycle.

- If you want to selectively add images to the flip book, modify the image until it is as you want it, then select the one-shot **Append Frame**. This appends the image to the end of the existing flip book. This method allows you to carefully construct a flipbook animation.

The **Replay** parameter controls the way in which the flip book is displayed. It has three selections:

- **Continuous** plays through all of the frames in the animation, wrapping around when it reaches the end.
- **Bounce** plays forward through the last *Max Frames* or fewer frames. When it reaches the end, it plays backwards through those frames.
- **Off** turns off the animation facility

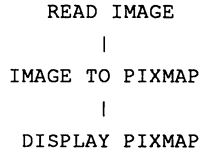
The **Replay Speed** parameter controls the rate at which flip book frames are displayed.

The **Current Frame** parameter allows you to select a particular frame "manually". It is normally updated to display the current frame, but for cases in which such updating would impact animation performance, it is not updated. Note that since only the last *Max Frames* frames are stored, the animation can begin at a frame other than 0.

After you select a particular frame, you can delete it with the one-shot **Delete Frame**.

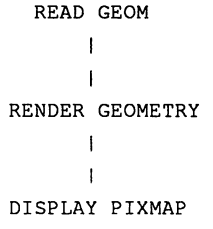
EXAMPLE 1

The following network reads in an image, converts it to a pixmap and then displays the image using **display pixmap**:



EXAMPLE 2

The following network reads in a geometry object, renders it and then displays the rendered object using **display pixmap**:



RELATED MODULES

transform pixmap, alpha blend, render geometry

LIMITATIONS

There is no way to store the "first *max frames*" frames of an animation loop.

SEE ALSO

The example scripts BRICK, FIELD LEGEND, and PROBE, as well as others demonstrate the **display pixmap** module.

NAME

display tracker - display and directly manipulate the tracer module's output

SUMMARY

Name	display tracker				
Type	data output				
Inputs	field 2D 4-vector byte uniform ("image")				
Outputs	upstream transform (invisible, optional, autoconnect)				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	scale	integer	1	1	16
	interpolate	toggle	off		

DESCRIPTION

display tracker is designed specifically to work with the modules **tracer**, and **ucd tracer**. The module **tracer** takes in volume data and performs volumetric rendering on it using ray-tracing. **tracer** outputs a 2D AVS image; **display tracker** displays this image in a window.

In addition to displaying **tracer**'s output, **display tracker** allows you to directly manipulate an image in its window using the mouse. You can rotate or translate a volume being rendered by moving the mouse, employing the "virtual spaceball" paradigm.

When you press the middle mouse button a bounding box appears superimposed around the rendered volume. Moving the mouse causes this bounding box to rotate. When the desired rotation is achieved, release the mouse button. The volume will be rendered again to show it rotated to the new position. The bounding box will disappear once the volume is redrawn. Translations are achieved in a similar way, using the right mouse button. To scale the object, use shift key in combination with the middle mouse button.

Note that **display tracker** takes AVS images as input. It can receive these images from any module that outputs an image. However, it will allow direct manipulation of images only when the module above it is equipped to receive the upstream transform that **display tracker** outputs.

INPUTS

Data Field (required; field 2D 4-vector byte uniform)
: An AVS image, typically output by the module **tracer**.

PARAMETERS

scale (integer)
: Multiplies size of input image by selected value. Scaling an image by a large amount will result in slower display times. In combination with the "width" and "height" parameters of **tracer**, you can use scale to create very large images.

interpolate (toggle)
: With interpolate off (default) the image is scaled using pixel replication. In other words, pixels are simply copied to increase the size of the image.

  With interpolate selected, bilinear interpolation is performed on the image when it is scaled. This results in smoother gradations in the color of pixels in the scaled image.

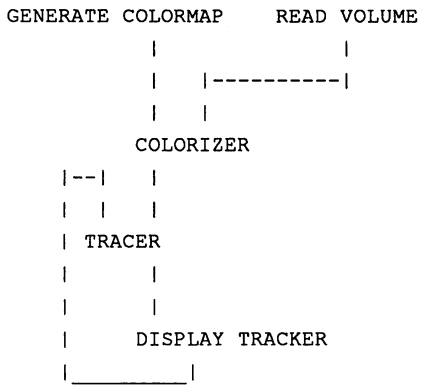
OUTPUTS

Upstream Transform (optional, invisible, autoconnect)
: The output port on the module **display tracker**, which is usually invisible, sends out a 4x4 transformation matrix describing rotations and

translations that have been applied to the image through movements of the mouse. This output port will automatically connect with **display tracker**'s invisible lefthand input port. This allows you to directly rotate and translate an image by moving the mouse in **display tracker**'s window.

EXAMPLE

The following network shows how **display tracker** is used to display the output of **tracer**. Note that **display tracker** also sends data "upstream" to **tracer**.



RELATED MODULES

Modules that could provide the **Data Field** input:

tracer
any other module which outputs an AVS image

Modules that can receive **display tracker**'s upstream transform:

tracer
gradient shade

SEE ALSO

The example script ANIMATED FLOAT demonstrates the **display tracker** module.

NAME

dot surface – generate points that define an isosurface

SUMMARY

Name dot surface

Type filter

Inputs field 3D scalar *any-data any-coordinates*

Outputs field 1D scalar (irregular 3-space)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Stepsize	real	.01	1.0E–5	1.0
Threshold	real	.02	–100	100

DESCRIPTION

The **dot surface** module accepts a 3D scalar field as input and generates a list of points that defines an isosurface. The input field is composed of cells, where each cell is defined as a subvolume composed of six faces. Each cell is processed checking for a possible intersection of the surface. If the cell does contribute to the surface it is then subdivided until the maximum physical dimension of the resulting subcell is ≤ the value of the **Stepsize** parameter. A smooth surface can be generated in this manner, given a sufficiently small **Stepsize** value.

The running time of this module is directly proportional to the number of cells processed and the number of cells that contribute to the surface. It is inversely proportional to the **Stepsize** value.

If the input field is uniform, then a physical grid is generated mapping the data volume into a canonical size. The largest dimension of the volume is mapped into the interval: [-1.0, +1.0]. Other dimensions are scaled accordingly, thus if a uniform volume consisting of 100 nodes in the x direction, 50 in the y direction and 20 in the z direction will have a bounding volume of: x=[-1.0, +1.0], y=[-0.5,+0.5], z=[-0.2,+2.0]. The distance between each node is then approximately equal to 0.02. The Stepsize parameter is relative to this length scale.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)
This module uses a scalar data value for each field element. If the input is a vector-valued field, then the first component of the vector is used as the scalar value.

PARAMETERS

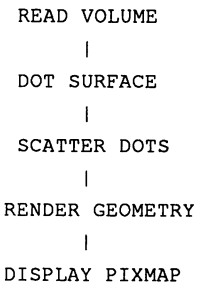
Stepsize A floating-point value that determines the resolution of the isosurface. The smaller this value, the smoother the surface.

Threshold A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the field element's data value equals the **Threshold** value.

OUTPUTS

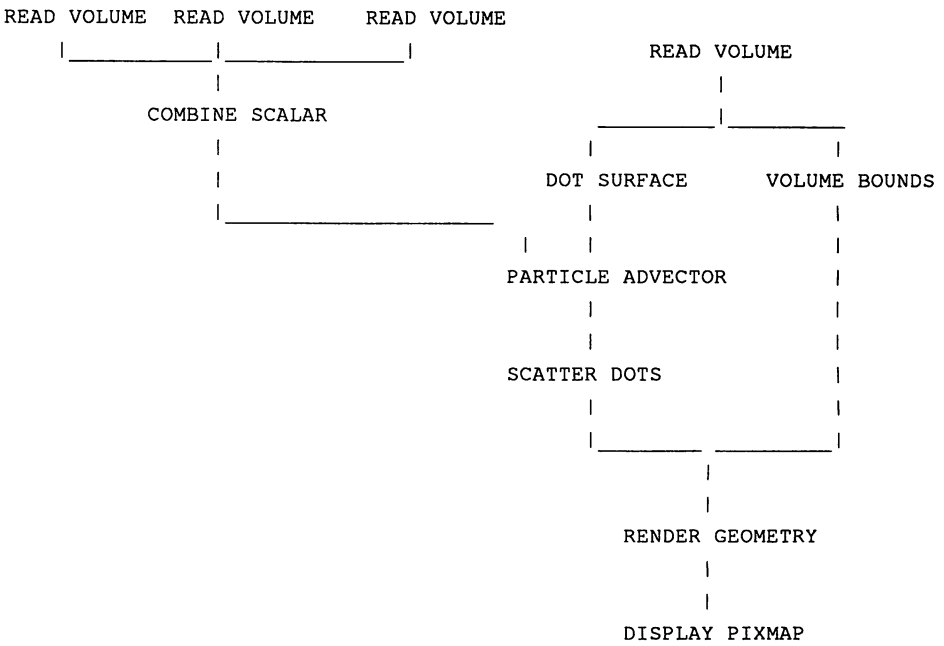
Point List (field 1D scalar irregular 3-space). The scalar data value for each output field element is unused. The only useful information is the 3D coordinate data.

EXAMPLE 1



EXAMPLE 2

In the following network, **dot surface** is used to generate a point list for use with **particle advector**. The three **read volume** modules create a vector velocity field, which is used as the driving force for the new locations of the point list originally generated by **dot surface**.



LIMITATIONS

The number of points may be inadequate to represent areas of small surface curvature with respect to the cell's local coordinate system.

A maximum of 80,000 points will be generated. Once the module calculates this number of points, it returns leaving all other cells unprocessed. Use **downsize** to avoid this if possible.

RELATED MODULES

Modules that could provide the Data Field input: read volume combine scalars

Modules that could be used in place of dot surface:

isosurface
tracer

Modules that can process dot surface output: scatter dots

SEE ALSO

The example script DOT SURFACE demonstrates the **dot surface** module.

NAME

downsize – reduce size of data set by sampling

SUMMARY

Name	downsize				
Type	filter				
Inputs	field *2D/3D any-data any-coordinates*				
Outputs	field of same type as input				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	downsize	integer	1	1	16

DESCRIPTION

The **downsize** module changes the size of the input data set by subsampling the data. It extracts every *N*th element of the field along each dimension, where *N* is the value of the **downsize factor** parameter. This technique preserves the aspect ratio of the input data.

This module is useful for operating on a reduced amount of data, in order to adjust other processing parameters interactively, or save memory. After the parameter values have been set, you can remove the **downsize** module, so that the full data set is used for final processing.

Alternatively, retain the **downsize** module in the network, so that you can interactively choose between image quality (**downsize factor** = 1 for highest-resolution data) and execution speed (**downsize factor** > 1 for lower-resolution data).

INPUTS

Data Field (required; field *2D/3D any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

downsize Determines how data elements from the field are sampled. Increasing this parameter causes more elements to be skipped over, thus *decreasing* the size of the output.

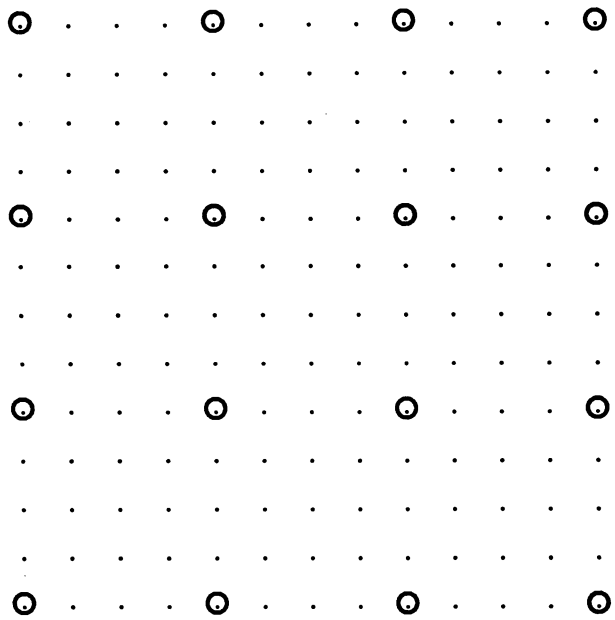
OUTPUTS

Data Field The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced by the **downsize factor**.

The **min_val** and **max_val** attributes of the output field are invalidated. Appropriate new **min_ext** and **max_ext** values are written to the output field.

EXAMPLE

The following diagram shows how a **downsize factor** of 4 reduces a 2D field. Each element of the field is represented by a dot. Only the larger dots are included in the output field.



LIMITATIONS

downsize works for 2D, and 3D data sets only.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
filter modules

Modules that could be used in place of **downsize**:

interpolate (arbitrary sampling)
crop (subset at high resolution)

Modules that can process **downsize** output:

colorizer
gradient shade
arbitrary slicer
orthogonal slicer
any other filter module

SEE ALSO

The example scripts FIELD MATH, and GRAPH VIEWER demonstrate the **downsize** module.

NAME

euler transformation - send object transformation matrix to other modules

SUMMARY

Name	euler transformation				
Type	data				
Inputs	none				
Outputs	field uniform 2D scalar float (transformation matrix)				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	theta	float	0	0	360
	phi	float	0	0	360
	rho	float	0	0	360
	scale	float	1	0	10

DESCRIPTION

euler transformation allows you to generate a 4x4 transformation matrix specifying scaling and rotations in x, y and z.

euler transformation is designed to be used with modules that can transform data in object space. This means that rotations and scaling operations are applied to a 3D data "object" before it is rendered and turned into a 2D image. **euler transform** does not supply the full "upstream transform" accepted by such modules as **brick** and **thresholded slicer**. Currently **euler transform** will work only with the modules **gradient shade** and **tracer**.

Using **euler transformation**'s dials you can select a transformation matrix that will scale and/or rotate an object. The order in which rotations are applied is x-y-z. If you rotate an object through a number of angles, it is always the original data that is transformed, i.e., transformations are not remembered and accumulated.

PARAMETERS

theta A floating point dial widget which controls rotation of the object's x axis. The x axis initially runs horizontally from negative on the left to positive on the right.

phi A floating point dial widget which controls rotation of the object's y axis. The y axis initially runs vertically from negative at the bottom to positve at the top.

rho A floating point dial widget which controls rotation of the object's z axis. The z axis initially runs perpendicular to the screen, with the positive z axis coming "out" of the screen, and the negative z axis "behind" the screen.

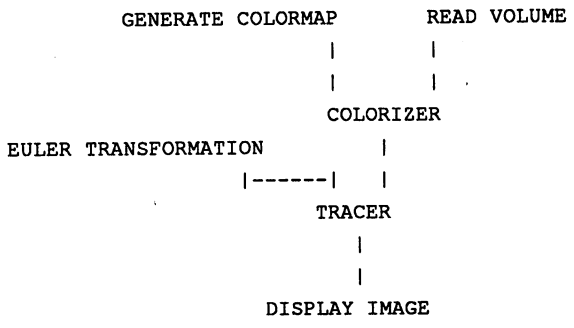
scale A floating point dial widget which controls the scaling coefficient of the transformation matrix. This makes the data "object" look larger or smaller.

OUTPUTS

Transformation Matrix (field 2D uniform scalar float)
The output is a 4x4 array of floating point values which specifies rotations and scaling operations that can be applied to transform an object around the origin of its own coordinate system.

EXAMPLE 1

The following network performs volumetric ray-tracing using **tracer**. By setting parameters in the module **euler transformation** you can rotate or scale the volume being rendered, so you can see all sides of the volume:



RELATED MODULES

Modules that accept **euler transformation**'s output:

tracer
gradient shade

SEE ALSO

The example script EULER TRANSFORMATION demonstrates the **euler transformation** module.

NAME

extract scalar – extract a scalar field from a vector field

SUMMARY

Name	extract scalar	
Type	filter	
Inputs	field *any-dimension n-vector any-data any-coordinates* (*n* = 1..25)	
Outputs	field *same-dimension* scalar *same-data same-coordinates*	
Parameters	*Name*	*Type*
	Channel	radio buttons

DESCRIPTION

The **extract scalar** module inputs a field whose data values are vectors (1D to 25D), and outputs one of the dimensions ("channels") as a scalar-valued field. The output field has the same structure as the input field, except that its data values are scalars (vector length of 1).

This module is useful for performing operations on individual channels of vector fields. It is frequently used with the **combine scalars** module, which composes vector fields from individual scalar fields.

INPUTS

Data Field (required; field *any-dimension n*-vector *any data any-coordinates*) The input data may be any field whose data values are vectors with 25 or fewer dimensions. Even scalar fields may be used, since their data values are considered to be 1D vectors.

PARAMETERS

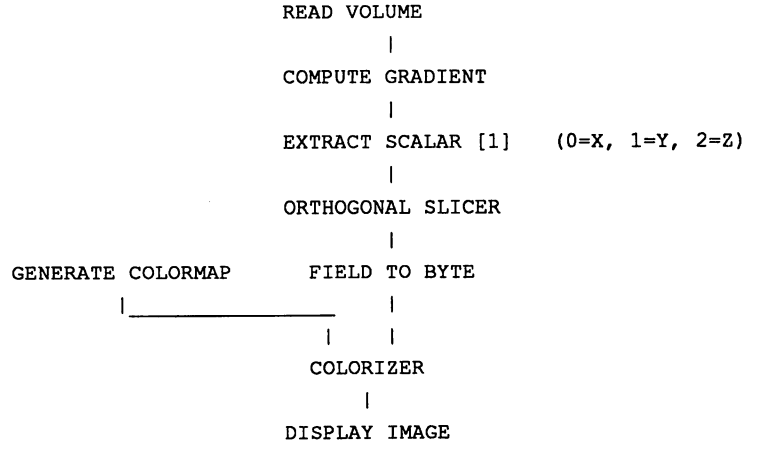
Channel Selects the dimension of the input data values to be output. A set of radio buttons appears, showing the labels that are attached to the dimensions of the *n*-vector data.

OUTPUTS

field (*same-dimension* scalar *same-data same-coordinates*)
The output field has the same dimensionality as the input field. The data for each element is reduced from a vector to a scalar.

EXAMPLE 1

This examples displays a slice of the Y-component of the gradient field of a volume:



For additional examples, see the **combine scalars** manual page.

RELATED MODULES

extract vector combine scalars

SEE ALSO

The example scripts CONTOUR GEOMETRY, CONTRAST, as well as others demonstrate the **extract scalar** module.

NAME

extract vector – subset of field vector elements as new field

SUMMARY

Name	extract vector				
Type	filter				
Inputs	field *any-dimension n-vector any-data any-coordinates*				
Outputs	field *same-dimension n-vector same-data same-coordinates*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Vector Length	integer dial	3	1	25
	Channel 0	boolean	off		
	Channel 1	boolean	off		
	Channel 2	boolean	off		
		.			
		.			
		.			
	Channel 24	boolean	off		

DESCRIPTION

The **extract vector** module takes a vector field of any dimension, coordinate system, or data type, and extracts a subset of the vector elements at each node. The output field is identical to the input field, but with only the selected vector elements at each node. This is useful, for example, with PLOT3D format data. PLOT3D data normally has seven vector elements at each node. However, only three of these, X-Momentum, Y-Momentum, and Z-Momentum, are useful if you are trying to visualize momentum vectors with the **hedgehog** module. **extract vector** is a convenient way to segregate just the vector elements needed. It is more convenient (and equivalent to) using **extract scalar** modules to extract individual vector elements and then pasting them together again with **combine scalar.**

extract vector can handle up to 25 vector elements. You can extract any subset of the ten elements.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

An AVS field with a vector of data elements at each node. The field can be any dimension, using any type of coordinate information, and any kind of data.

PARAMETERS

Vector Length

An integer dial that specifies the vector length of the *output* field. The default is 3, the minimum is 1, and the maximum is 25.

Channel 0
Channel 1
Channel 2 ...

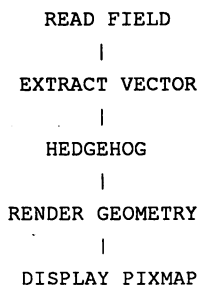
A series of on/off switches that specify which of the input vector elements to extract into the output field. If the input vector elements have been labelled, then their labels will appear instead of the default "Channel *n*". Only as many switches will appear as there are input vector elements. By default, all of the switches are "off". There is no way to change the order of vector elements; if X preceded Y in the input field, it will do so in the output field (you can change the order of vector elements by using multiple instances of the **extract scalar** module, feeding into one **combine scalars**).

OUTPUTS

Data Field (field *same-dimension n-vector same-data same-coordinates*)
The output field has the same form as the **Data Field** input, except that its vectors are shorter.

EXAMPLE 1

The following network extracts the x, y and z momentum vector elements from a field dataset, then plots their sum vector using **hedgehog**. The dataset operated on is *bluntfin.fld*, which contains PLOT3D data in field format.



RELATED MODULES

Modules that could provide the **Data Field** input:

Any module that produces a vector field output

Modules that could be used in place of **extract vector**:

extract scalar
combine scalar

Modules that can process **extract vector** output:

Any module that can process vector fields

NOTE

This **extract vector** module is *not* the same as the **extract vector** module formerly available in the AVS user-contributed module library.

SEE ALSO

The example script STREAMLINES demonstrates the **extract vector** module.

NAME

field legend - select value from scalar field using color legend

SUMMARY

Name	field legend		
Type	mapper		
Inputs	field *n-dimensions n-vector any-data any-coordinates*		
Outputs	real		
Parameters	*Name*	*Type*	*Default*
	node data	choice	<data 1>

DESCRIPTION

field legend takes a n-vector input field and a colormap and produces a "color legend" widget. The widget displays the range of values associated with one of the field's vector elements, and allows you to pick specific values of interest based on the colors associated with those values. Thus, the colors in the legend will match the colors used to display the field.

field legend displays the current colormap as a horizontal color legend. Beneath this table **field legend** prints a scale representing the range of values of one vector element of the input field. Values along this scale are displayed in scientific notation. The colormap is normalized to map to the range of values present in the input field. **field legend** behaves, in this respect, like the module **color range**. If the selected scalar has some label or unit associated with it (i.e. momentum, m/sec) **field legend** will print these as the title of the color legend.

By moving a "radio tuner" type dial along the color legend you can select specific data values. **field legend** outputs the value selected as a single floating point number.

field legend is designed to work with modules that take fields and allow you to visualize subsets of the data. Such modules include: **isosurface, thresholded slicer,** and **contour to geom.** Typically, subsets of data are selected by choosing specific values with a dial widget. For example, using **isosurface** you can select what "level" of data values to display as a surface. Manipulating colored data using **field legend**'s color legend is often more intuitive than using a floating-point parameter widget.

The module **field legend** accepts n-dimensional n-vector fields. Use the **node data** radio buttons to select one scalar element of the field to use for the color legend's range of values. If the input field is scalar to begin with **field legend** provides no buttons.

field legend outputs a single floating-point value. As a result it connects to the floating-point parameter port of another module. Before you can connect **field legend** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A purple parameter port should appear on the module icon. Connect this parameter port to the **field legend** module icon in the usual way one connects modules.

INPUTS

Data Field (required; field *n-dimensions n-vector any-data any-coordinates*)
: An AVS field which supplies the range of values displayed by field legend.

Colormap (required; colormap)
An AVS colormap which is used as the legend for selecting values from the data field.

PARAMETERS

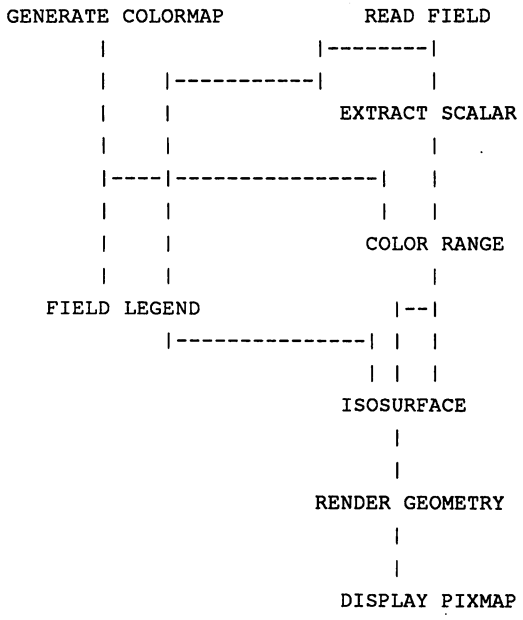
Channel Selects the vector element of the input data values to be used as the color legend's range. A set of radio buttons appears, showing the labels that are attached to the dimensions of the *n*-vector data.

OUTPUTS

Real A single floating-point value selected from the range of values in the field.

EXAMPLE 1

The following network displays isosurfaces of a 3D scalar field. **field legend** allows you to select what "level" of values should be displayed as a surface. Note that **field legend** performs the equivalent of **extract scalar** and **color range**, but these two modules still need to filter the field that **isosurface** receives. field. Also note that **generate colormap** sends the same colormap to both **field legend** and **color range**



RELATED MODULES

Modules that could provide the **Data Field** input:

read field
any other module which outputs a 3D field

Modules that could provide the **Colormap** input:

generate colormap
color range

Modules that can process **field legend's** output:

isosurface
thresholded slicer
contour to geom

SEE ALSO

The example script FIELD LEGEND demonstrates the **field legend** module.

NAME

field math – perform math operations between fields

SUMMARY

Name field math

Type filter

Inputs field *any-dimension n-vector any-data any-coordinates*
field *same-dimension same-vector any-data same-coordinates* (OPTIONAL)

Outputs field *same-dimension same-vector any-data same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
choice	choice	+		
Normalize	boolean	off		
Constant	float typein	0.0	*unbounded*	*unbounded*

DESCRIPTION

The **field math** module performs unary and binary operations upon fields.

The unary operations are +, - (minus), *, / ; logical bitwise And, Or, Xor, Not; and Square, Sqrt, and Root Mean Square (RMS). Unary operations are performed against the right-most port field when there is only one field attached. They use the value supplied by the **Constant** input parameter. Even when a second field is supplied, the Not, Square, and Sqrt operations are performed only on the first field.

When two fields are connected to the module, the **Constant** parameter is not displayed and the fields are evaluated against each other.

The input fields must be of the same dimensionality, size, vector length, and coordinate type(i.e uniform, rectilinear, or irregular). When the fields contain different data types, they are cast to the "higher order." In other words, if one field is a byte field and the other is a float, the math is done as floating point and the output field is floating point.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The rightmost input field is used as the input to unary operations.

Data Field (optional; field *same-dimension same-vector any-data same-coordinates*)
The leftmost field is the first operand in binary operations. It must have the same dimension, size, vector length, and coordinate type as the first input field.

PARAMETERS

+
-
*****
/
And (bitwise)
Or (bitwise)
Xor (bitwise)
Not (bitwise)
Square
Sqrt
RMS (Root Mean Square)
A choice of operations. With only one field, the unary operations performed are as follows:

```
+          field_value  +  Constant
-          field_value  -  Constant
```

*	field_value * Constant
/	field_value / Constant
And	field_value AND Constant
Or	field_value OR Constant
Xor	field_value XOR Constant
Not	field_value
Square	field_value * field_value
Sqrt	sqrt (field_value)
RMS	sqrt (field**2 + constant**2)

With two fields, the binary operations performed are as follows:

+	field1 + field2
-	field1 - field2
*	field1 * field2
/	field1 / field2
And	field1 AND field2
Or	field1 OR field2
Xor	field1 XOR field2
Not	NOT field1
Square	field1 * field1
Sqrt	sqrt(field1)
RMS	sqrt (field1**2 + field2**2)

Normalize Selecting **Normalize** causes the results of the operation to be normalized to between 0 and 1 for reals, 0 and 255 for bytes and 0 and 65535 for integers. **Normalize** is off by default.

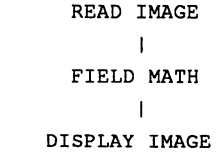
Constant A floating point typein to specify the constant value to use in unary operations. If two fields are connected to the module, **Constant** is ignored, and disappears from the control panel. The default is 0.0. There is no upper or lower limit.

OUTPUTS

Data Field (field *same-dimension same-vector any-data same-coordinates*)
The output field has the same form as the input fields. If the input fields differed in the data type, the output field will have the more elaborate data type.

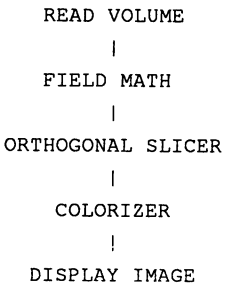
EXAMPLE 1

The following network inverts (flips the look-up table) an image using the Not function, with Normalize on. The same effect can be achieved by multiplying the image by -1.



EXAMPLE 2

This network does a logical AND on a volume against 128 (0x80) which produces a volume with only 0s and 255s based on whether the source voxel was greater or less than 128.



RELATED MODULES

Modules that could provide the **Data Field** inputs:

Any module that outputs a field

Modules that can process **field math** output:

Any module that inputs a field

SEE ALSO

Two FIELD MATH example scripts demonstrate the **field math** module.

NAME

field to byte – transform any field to an byte-valued field

SUMMARY

Name	field_to_byte			
Type	filter			
Inputs	field *any-dimension n-vector any-data any-coordinates*			
Outputs	field *same-dimension same-vector* byte *any-coordinates*			
Parameters	*Name*	*Type*	*Default*	*Choices*
	byte normalize	toggle	on	on,off

DESCRIPTION

The **field_to_byte** module takes a field of data (*integer, real, double,* or *byte*) and converts it to an *byte* field. It can be used in conjunction with volume visualization modules that have a bias towards byte fields (i.e., **compute gradient**).

By default, the input data is normalized to the range 0..255 If the toggle parameter **byte_normalize** is turned off, the data is "clamped" to that range instead. (See below for details.)

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

The input data may be any AVS field.

PARAMETERS

byte_normalize

This is a toggle parameter:

- **If on:** The data is transformed linearly into the range 0..255:

$$new\_value = \frac{(value - min) * 255}{max - min}$$

- **If off:** The data is "clamped" so that no value falls outside the range 0..255:

If *value* < 0	*new_value* = 0
If 0 ≤ *value* ≤ 255	*new_value* = *value*
If *value* > 255	*new_value* = 255

OUTPUTS

Data Field (field *same-dimension same-vector* byte *same-coordinates*

The output field has the same dimensionality as the input field, but each scalar value is forced to be a byte.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume

Modules that could be used in place of **field_to_byte**:

field to int
field to float
field to double

Modules that can process **field_to_byte** output:

read volume

SEE ALSO

The example scripts FIELD TO BYTE and FIELD TO INTEGER demonstrate the **field to byte** module.

NAME

field to double – transform any field to a field of double-precision floating point values

SUMMARY

Name field_to_double

Type filter

Inputs field *any-dimension n-vector any-data any-coordinates*

Outputs field *same-dimension same-vector* double *same-coordinates*

Parameters

Name	*Type*	*Default*	*Choices*
double normalize	toggle	off	on,off

DESCRIPTION

The **field_to_double** module takes a field of data (*byte, real, double,* or *integer*) and converts it to an *double* field. This may be useful for computing fields at greater data resolutions.

By default, the input data is simply cast (re-typed) to be double-precision floating point. If the toggle parameter **double_normalize** is turned on, the data is also normalized to the range 0..1. (See below for details.)

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

double_normalize
This is a toggle parameter:

- **If on:** The data is transformed linearly into the range 0..1:

$$\text{new\_value} = \frac{(\text{value} - \text{min})}{\text{max} - \text{min}}$$

- **If off:** The data is converted to double-precision floating point format.

OUTPUTS

Data Field (field field *same-dimension same-vector* double *same-coordinates*
The output field has the same dimensionality as the input field, but each scalar value is forced to be a double-precision number.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

read volume field to byte field to int field to float

SEE ALSO

The example script FIELD TO INTEGER demonstrates the **field to double** module.

NAME

field to float – transform any field to a field of single-precision floating point values

SUMMARY

Name	field_to_float
Type	filter
Inputs	field field *any-dimension n-vector any-data any-coordinates*
Outputs	field *same-dimension same-vector* float *same-coordinates*

Parameters	*Name*	*Type*	*Default*	*Choices*
	float normalize	toggle	off	on,off

DESCRIPTION

The **field_to_float** module takes a field of data (*byte, real, double,* or *integer*) and converts it to an *float* field. It can be used in conjunction with modules that have a bias towards *float* fields (**particle advector, samplers**).

By default, the input data is simply cast (re-typed) to be single-precision floating point. If the toggle parameter **float_normalize** is turned on, the data is also normalized to the range 0..1. (See below for details.)

INPUTS

Data Field (required; *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

float_normalize
This is a toggle parameter:

- If ON, the data is transformed linearly into the range 0..1:

$$\text{new\_value} = \frac{(\text{value} - \text{min})}{\text{max} - \text{min}}$$

- If OFF, the data is converted to single-precision floating point format (IEEE 754).

OUTPUTS

Data Field (field *same-dimension same-vector* float *same-coordinates*
The output field has the same dimensionality as the input field, but each scalar value is forced to be a single-precision number.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

read volume particle advector samplers field to byte field to int field to double

SEE ALSO

The example script FIELD TO INTEGER demonstrates the **field to float** module.

NAME

field to int – transform any field to an integer-valued field

SUMMARY

Name	field to int			
Type	filter			
Inputs	field *any-dimension n-vector any-data any-coordinates*			
Outputs	field *same-dimension same-vector* integer *same-coordinates*			
Parameters	*Name*	*Type*	*Default*	*Choices*
	int normalize	toggle	off	on,off

DESCRIPTION

The **field to int** module takes a field of data (*byte, real, double,* or *int*) and converts it to an *int* field. This may be useful for performing integer math with greater precision (–65536..65535) than that offered by byte fields (0..255).

By default, the input data is "clamped" to the range 0..65535. If the toggle parameter **int_normalize** is turned on, the data is normalized to that range instead. (See below for details.)

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

int_normalize
This is a toggle parameter:

- If ON, the data is transformed linearly into the range 0..65535:

$$\text{new\_value} = \frac{(\text{value} - \text{min}) * 65535}{\text{max} - \text{min}}$$

- If OFF, the data is "clamped" so that no value falls outside the range 0..65535:

If *value* < 0	*new_value* = 0
If 0 ≤ *value* ≤ 65535	*new_value* = *value*
If *value* > 65535	*new_value* = 65535

OUTPUTS

Data Field (field *same-dimension same-vector* integer *same-coordinat*)
The output field has the same dimensionality as the input field, but each scalar value is forced to be an integer.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

read volume field to byte field to float field to double

SEE ALSO

The example script FIELD TO INTEGER demonstrates the **field to int** module.

NAME

field to mesh – transform a 2D scalar field to a surface in 3D space

SUMMARY

Name	field_to_mesh				
Type	mapper				
Inputs	field 2D scalar *any-data any-coordinates* colormap (optional)				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Z scale	float	1.0	-25.0	25.0

DESCRIPTION

The **field to mesh** module converts a two-dimensional field into surface in 3D space, represented as a GEOM-format *mesh*. Each element of the field is mapped to a point in a base plane. The height of the mesh above each point in this plane is proportional to the scalar value of the field.

For irregular fields, the "base plane" need not actually be planar. The 2D grid of field elements is mapped into 3D space using the coordinate array included in the field description.

INPUTS

Data Field (required; field 2D scalar *any-data any-coordinates*)
The input data must be a 2D field with a scalar data value at each element. The data value may be of any primitive type: byte, integer, float, or double, and have uniform, rectilinear, or irregular coordinates.

Colormap (optional)
Colors each vertex of the mesh, according to the data value at that point. If no colormap is supplied, the vertices are colored white.

PARAMETERS

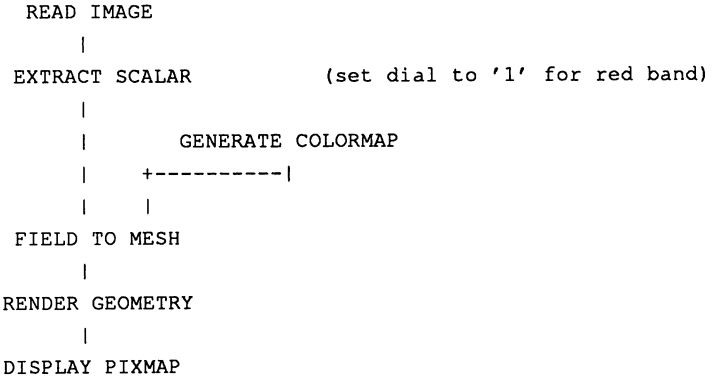
Z scale With uniform input fields, determines the height of the mesh. With rectilinear and irregular input fields, this parameter is unused.

OUTPUTS

Geometry The output is an AVS *geometry*.

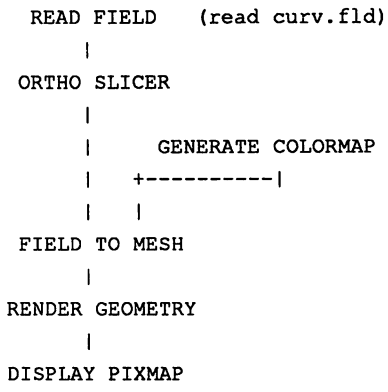
EXAMPLE 1

This example uses the "red band" (red component of the RGB color) of an image as a 2D field. It then converts this field to a mesh, using a colormap, and displays the mesh.



EXAMPLE 2

This example shows how to take orthographic slices through a curvilinear data set, showing them as <XYZ> meshes:



LIMITATIONS

This module can output meshes that are too big for the **render geometry** module to handle, causing AVS to crash. Use the **downsize** filter module to reduce the size of the input data.

NOTE

The normals in the *geometry* that **field to mesh** outputs are reversed from their direction in AVS 2.0. This is a correction made in the AVS 3.0 release.

SEE ALSO

The example script COLOR RANGE demonstrates the **field mesh** module.

NAME

field to ucd – convert AVS field to unstructured cell data format

SUMMARY

Name	field to ucd				
Type	filter				
Inputs	field 3D *n-vector any-data any-coordinates*				
Outputs	ucd structure				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	field comp	choice	<data 1>		
	cell connect	boolean	off		

DESCRIPTION

field to ucd converts a 3D AVS field into a UCD structure. Note, that to initiate the conversion you must press the "field comp" button. This is the case even if the button is highlighted from a previous operation.

field to ucd converts a scalar value at each location in the input field into the value of a node in the UCD structure. **field to ucd** can receive an n-vector field, but its output UCD structure can only have a scalar value at each node. Note that the cells of the output structure will be hexahedra.

An AVS field is an array with a vector of values at each location. On the other hand unstructured cell data (UCD) has a hierarchical structure, consisting of structure data, cell data, and node data. Both structure data and cell data are optional, i.e., UCD structures may often contain only node data.

Structure data refers to data that holds for the entire structure. For example, in a simulation of forces on an object, the location of loads could be stored as structure data. Cell based data is particular to each cell in the structure.

At the lowest level are the nodes, which are the vertices of the cells. Each node can have several data components associated with it. Furthermore, each of these components may itself be either a vector or a scalar. Adjacent cells may share the same nodes. Conversely, nodes will tend to belong to several cells. For each node there is a list of cells it belongs to. This is the "node connectivity list". For a detailed description of the unstructured cell data format, see Appendix E in the *AVS Developer's Guide.*

field to ucd computes the min and max extents of the structure. If the **cell connect** parameter is selected, **field to ucd** also computes the node list for each cell in the output structure. This connectivity list is necessary if the output UCD structure is to be processed by **ucd streamline**.

Thus, if the input field has dimensions *width * height * depth*, there will be width * height * depth nodes in the output structure. The number of cells in the structure output by **field to ucd** would be *(width - 1) * (height - 1) * (depth -1).*

If the type of the input field is irregular, the coordinates associated with each field data element become the coordinates of the UCD structure's nodes. If the input field is rectilinear, node coordinates are computed using the field's "points" information. If the input field is uniform, node coordinates are computed based on the implicit organization of the field array.

INPUTS

Data Field (required; field 3D *n-vector any-data any-coordinates*)

The input data must be a 3D field, with an n-vector of values at each location in the field. The field can be uniform, rectilinear, or irregular.

PARAMETERS

field comp Initiates the conversion of the field to a UCD structure.

cell connect

A boolean switch. When **cell connect** is selected the cell connectivity list is calculated. For each node in the UCD structure, this lists which cells the node belongs to. This is needed when using the **ucd streamline** to produce streamlines.

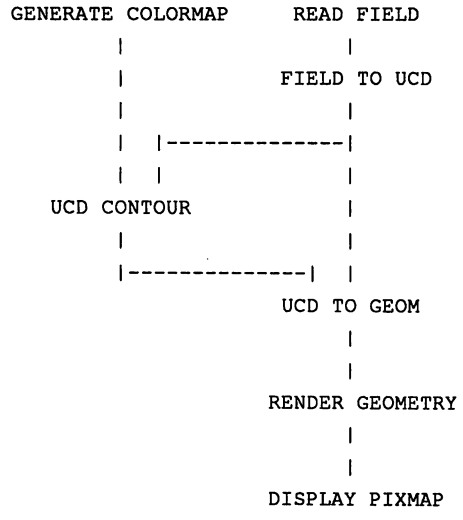
OUTPUTS

UCD Structure

The output structure is in AVS unstructured cell data (UCD) format.

EXAMPLE

The following network reads in an AVS field, converts it into a UCD structure, then into a geometry, and renders it:



RELATED MODULES

Modules that could provide the **Data Field** input:

read field
Any module that outputs a 3D AVS field.

Modules that can process **field to ucd**'s output:

ucd to geom, ucd crop, ucd threshold, ucd extract, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d, ucd legend, ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script FIELD TO UCD demonstrates the **field to ucd** module.

NAME

file browser – send a filename to one or more module(s) filename parameter port(s)

SUMMARY

Name	file browser		
Type	data		
Inputs	none		
Outputs	string		
Parameters	*Name*	*Type*	*Default*
	File Browser	browser	NULL

DESCRIPTION

The **file browser** module sends a single user-specified filename string to one or more string parameter ports on one or more receiving modules. Its purpose is to allow you to simultaneously control filename parameter input to more than one module using only a single File Browser input widget.

Before you can connect **file browser** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A light blue parameter port should appear on the module icon. Connect this parameter port to the **file browser** module icon in the usual way.

PARAMETERS

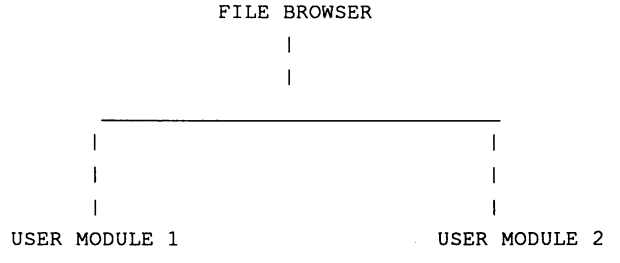
File Browser (string)
: The single filename string, specified through a File Browser widget, to be sent to the receiving module(s) filename string parameter port(s). The default value is NULL.

OUTPUTS

filename (string)
: The filename string value is sent to all modules with filename string-type parameter ports that are connected to the **file browser** module.

EXAMPLE 1

The following network inputs the same data file simultaneously to two user-written modules.



RELATED MODULES

Modules that can process **file browser** output:

all modules with filename string parameter ports

SEE ALSO

The example script FILE BROWSER demonstrates the **file browser** module.

NAME

flip normal – change direction of each vertex normal for a geometry object

SUMMARY

Name	flip normal
Type	filter
Inputs	geometry
Outputs	geometry
Parameters	none

DESCRIPTION

The **flip normal** module transforms an AVS *geometry* so that all the vertex normals point in the opposite direction. This is most often used to correct normals that have been calculated incorrectly.

When its normals are backwards, a 3D object appears unaffected by light sources; it frequently appears as a grey silhouette.

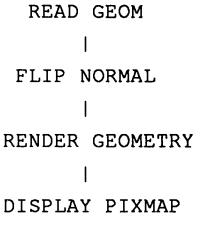
INPUTS

Geometry The input can be any AVS *geometry*.

OUTPUTS

geometry The output is an AVS *geometry* that represents the same object.

EXAMPLE



RELATED MODULES

read geom, offset, shrink, tube, render geometry

NOTES

Some filter modules (e.g. **offset**) sometimes produce bad normals, which can be corrected with **flip normal**.

SEE ALSO

The example script FLIP NORMALS demonstrates the **flip normal** module.

NAME

float – send a floating point number to one or more module(s) floating point parameter port(s)

SUMMARY

Name	float				
Type	data				
Inputs	none				
Outputs	float				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Float Value	dial	0.0	*unbounded*	*unbounded*

DESCRIPTION

The **float** module sends a single user-specified floating point value to one or more float-type parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control floating point parameter input to more than one module using only a single input widget (whether the default dial, or a typein).

Before you can connect **float** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A purple parameter port should appear on the module icon. Connect this parameter port to the **float** module icon in the usual way.

PARAMETERS

Float Value (float)

The single user-supplied floating point value to be sent to the module(s) floating point parameter port(s). The default value is 0.0. There is no minimum or maximum restriction on the value. You should be aware of the range of numbers that it is reasonable to send to the receiving modules. The default widget type is a dial. If you change this to a typein widget, then you should type the value as a real number, e.g., .55 or -100.25.

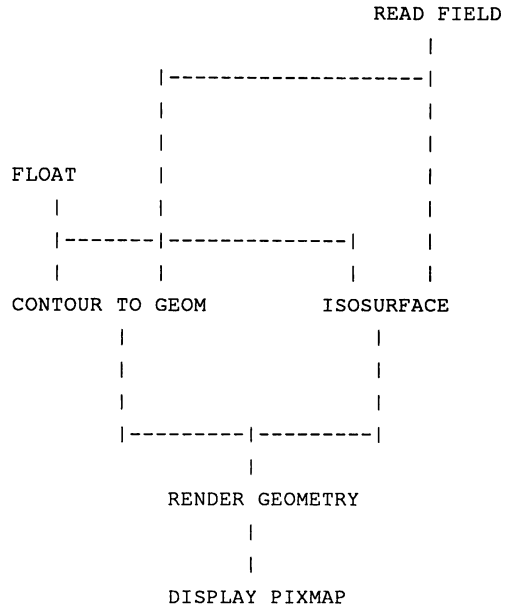
OUTPUTS

Float Output (float)

The floating point value is sent to all modules with floating point-type parameter ports connected to the **float** module.

EXAMPLE 1

The following network reads a field, then produces both a contour and an isosurface for the same floating point value, with both outputs composited in the render geometry display window.



RELATED MODULES

Modules that can process **float** output:

all modules with float-type parameter ports

NAME

generate colormap – output AVS colormap

SUMMARY

Name generate colormap

Type data

Inputs none

Outputs colormap

Parameters

Name	*Type*	*Default*	*Min*	*Max*
lo value	float	0	none	none
hi value	float	255	none	none
hue				
saturation				
brightness				
opacity				
composite				
edit	popup window			
read				
write				

DESCRIPTION

The **generate colormap** module produces an AVS *colormap* data structure, for use by modules that transform input data into color values. These modules include:

colorizer
arbitrary slicer
bubbleviz
field to mesh
isosurface

Note that when the range of values in the input field is not evenly distributed between 0 and 255, or if much of the data lie outside the 0 to 255 range, you can use the **color range** module to effectively scale the output colormap to the range of your data. For a more detailed description, see the man page for **color range**.

This module bases its output colormap on the state of the *colormap editor* control widget, which is invoked by clicking the **edit** button in the control panel. The colormap editor uses a *hue-saturation-brightness* (HSB) color space model:

hue	0.00 = red
	0.16 = yellow
	0.33 = green
	0.50 = cyan
	0.66 = blue
	0.83 = magenta
saturation	0.00 = white
	1.00 = **hue**
brightness	0.00 = black
	1.00 = **hue**

The HSB color space can be thought of as an inverted cone:

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

- The **brightness** axis runs from the tip of the cone (black) to the base (white).

PARAMETERS

The state of the colormap editor control widget specifies the colormap to be generated. This widget is a popup window that includes four *editing panels* and eight buttons. The editing panels are:

hue Raises the **hue** editing panel. The default panel is a linear ramp: 0=blue through 255=red.

saturation Raises the **saturation** editing panel. The default panel has all colors fully saturated: 0–255 = 1.0.

brightness Raises the **brightness** editing panel. The default panel has all colors at full brightness: 0–255 = 1.0.

opacity Raises the **opacity** editing panel. (The **opacity** value is placed in the *auxiliary* field of the colormap.) The default panel is a linear ramp: 0=0.0 through 255=1.0.

The following buttons apply to the editing panel that is currently visible:

composite
This is a toggle — when ON, the editing panel becomes a composite of the hue, saturation, and brightness panels. A line through the composite panel display indicates the status of the currently-selected panel: hue, saturation, brightness, or opacity.

edit
Press this button to pop up an editing window for the current panel. The editing window includes these settings:

Min
Max In the HSB color model, the hue is represented as a circle. By default, the colormap produces hues between 0† and 240† around this circle. This is the hue range from red to blue. The **Min** and **Max** parameters allow you to select another hue range.

From/Value
To/Value
do interpolation
These controls provide precise numeric control over the mapping of input values to output colors. This is an alternative to scribing a freehand mapping with the mouse. For example, suppose the input values range from 0 to 175, but the values in the range 160–165 are critical. It would be desirable to have the values in the critical range be mapped to a contrasting hue (or range of hues). To accomplish this, set **From** to 160 and **To** to 165. Set the two **Value** settings to numbers that produce a contrasting hue, e.g. 0.0 (bright red) as the From Value and 0.1 (semi-bright red) as the To Value. Then press the **do interpolation** button to redefine the portion of the colormap specified by the above settings as a linear ramp.

invert Inverts the current editing panel along a horizontal axis. The hue (or saturation, etc.) assigned to the *lo value* becomes assigned to the *hi value*, and vice-versa.

flip Flips the current editing panel along a vertical axis. Each input value is mapped to the complementary output value (e.g. an opacity of 0.667 is becomes 0.333).

cycle Performs a circular shift on the current editing panel. For example, with a **Step** value of 10, pressing the **cycle** button effectively moves the image in the editing panel down by 10 slots (out of 255). Subsequent presses of **cycle** move the image again and again.

ramp Generates a linear ramp on the currently raised editing panel: *lo value* =0.0 through *hi value*=1.0.

smooth Smooths the curves of a hand-scribed editing panel.

read

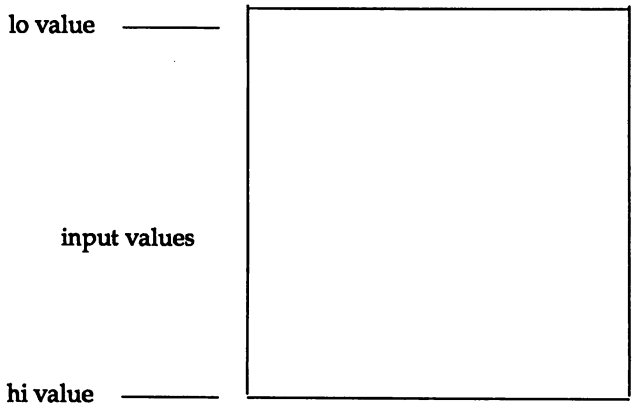
Reads a colormap from disk storage. Pressing this button pops up a File Browser widget, allowing you to specify a filename. You can also change the working directory.

write

Writes the current colormap to a disk file. Pressing this button pops up a File Browser widget, allowing you to specify a filename. You can also change the working directory.

You can change an editing panel from its current setting by scribing a curve with the mouse. Place the mouse cursor anywhere within the editing panel, hold down any mouse button, and drag upward or downward.

Each editing panel is organized as follows:



output values: 0-1

lo value (see *LIMITATIONS* below) a floating point dial which specifies the minimum data value that can be used as input to the colormap (the value at the top of the editing panel). The default low value is 0.

hi value (see *LIMITATIONS* below) a floating point dial which specifies the maximum data value that can be used as input to the colormap (the value at the bottom of the editing panel). The default high value is 255.

OUTPUTS

colormap The output is an AVS *colormap*.

COLORMAP FILE FORMAT

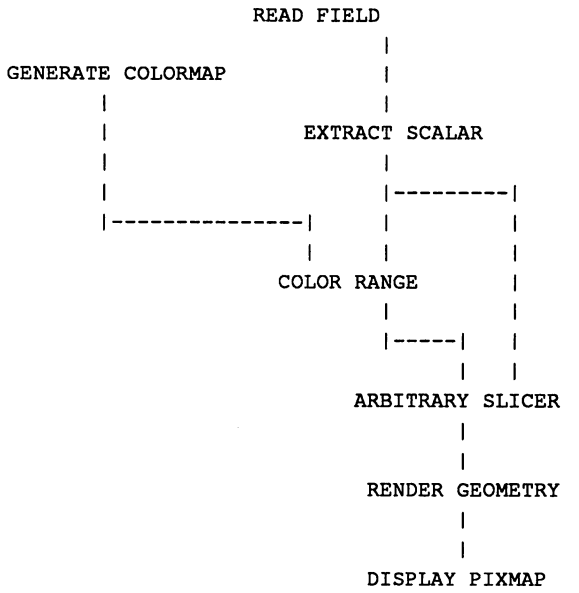
Colormaps are stored on disk as ASCII files, in the following format:

```
number_of_entries
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
low_value high_value
```

The hue, saturation, brightness, and opacity values are normalized to the range 0.0 – 1.0. The default colormap has 255 entries, with the hue, saturation, brightness, and opacity default values as described above.

EXAMPLE

The following network reads in a 3-vector field, i.e. every field location has 3 values associated with it. The **extract scalar** module selects one of the fields values. **color range** stores the field's min and max values so that the colormap can be scaled to the range of data in the field:



LIMITATIONS

The **generate colormap** module can only generate colormaps with 255 entries.

SEE ALSO

The example scripts COLOR RANGE, PROBE, as well as others demonstrate the **generate colormap** module.

NAME

generate filters - generate 2D filters for image processing

SUMMARY

Name	generate filters				
Type	data				
Outputs	field 2D scalar float				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	selection	choice	Gaussian		
	Size	integer	3	1	65
	focus1	float	0.5	0.0	10.0
	focus2	float	0.25	0.0	10.0
	power	float	1.0	0.0	10.0
	angle	float	0.0	0.0	360.0
	scale	float	0.5	0.0	1.0

DESCRIPTION

generate filters produces 2D scalar fields of floating point values. These can be used as convolution filters in image processing by feeding them into the **convolve** module.

generate filters outputs the following filters: Gaussian, Laplacian, Power, Ellipse, Line, Random, dx, and dy. All filters, except Laplacian and Random, are normalized to the range 0.0 to 1.0.

PARAMETERS

selection Sets the function used to produce the image processing filter. Each functions has a number of parameter dials associated with it. Only the dials associated with a given function will be visible when you select that function. There are eight options:

Gaussian
Generates filters using a normal-distribution, bell-shaped, function. The Gaussian operator is typically used as a low-pass filter to smooth or blur images.

Laplacian
Generates "mexican hat" shaped function. The Laplacian function produces a high-pass filter. A Laplacian function is produced as the difference between two Gaussian functions. This is why there are two foci for the Laplacian functions: one for each of the two component Gaussians. Laplacian filters are not normalized to the range of 0.0 to 1.0.

Power
Generates an exponential function.

Ellipse
Generates an elliptical function, with two foci.

Line
Generates a filter that has the effect of blurring an image along a given line.

Random
Generates a uniformlly distributed random filter that is not normalized.

dx Generates the x component of the Sobel operator (see **sobel**), which detects changes in the image in the x direction. This can be used to locate vertical edges in images. The **dx** filter is 3x3 and cannot be resized.

dy Generates the y component of the Sobel operator (see **sobel**), which detects changes in an image in the y direction. This can be used to locate horizontal edges in images. The **dy** filter is 3x3 and cannot be

resized.

Size Determines the length of the filter's sides. Filters are squares. NOTE: convolving a filter with an image is a N x M operation, where N is the number of elements in the convolution filter and M is the number of elements in the image. Consequently, filters of sizes over 16 require a great deal of computation. The size parameter is used by all of the functions.

focus1 Used in Gaussian, Power, and Line filters to control the width and amplitude of the filter function, which are inversely related. In the Laplacian filter, this controls the width and amplitude of one of the two component Gaussian functions. In the Ellipse filter, this controls the ellipse's first focus.

focus2 In the Laplacian filter, this controls the width and amplitude of the second component Gaussian function. In the Ellipse filter, this controls the ellipse's second focus.

power Value between 0.0 and 10.0, used in the Power filter to set the exponent of the function.

angle Value between 0.0 and 360.0, used in the Line filter to set the angle of the line relative to the horizontal.

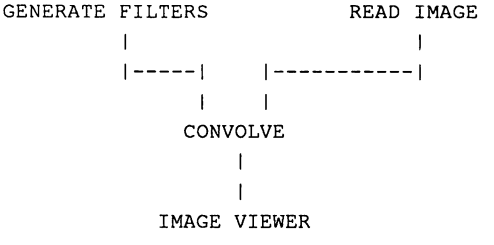
scale Value between 0.0 and 1.0, used with the Laplacian or random filters to reduce the range of the function's values.

OUTPUTS

Filter The output is a 2D field of scalar floats, i.e. a grid where every location contains one floating point value.

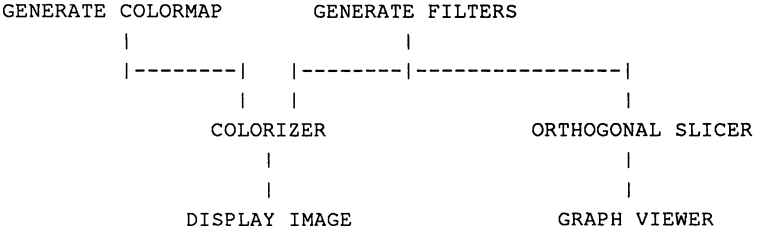
EXAMPLE 1

The following network generates a filter, convolves it with an image, then displays the result:



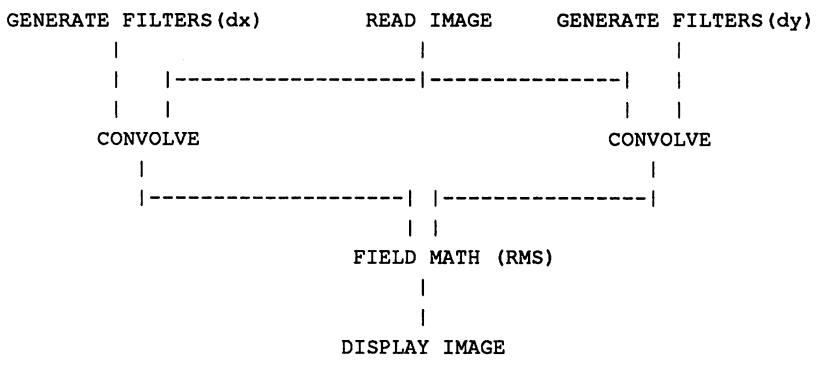
EXAMPLE 2

The following network shows what the convolution filters produced by **generate filters** look like, both as an image, and as an x-y graph. The module **colorizer** makes an AVS *image* out of the filter and colors it with a colormap output by **generate colormap** (NOTE: the colormap's max value must be changed to some small number, such as 0.03, using the Dial Editor). At the same time, **orthogonal slicer** generates a cross section through the filter, which can then be displayed as a histogram using the **graph viewer** module. (NOTE: set **orthogonal slicer** to slice through the middle of the filter.)



EXAMPLE 3

The following network shows how you can combine the **dx** and **dy** filters into the equivalent of a "sobel" edge detecting operator:



RELATED MODULES

Modules that can process **generate filter**'s output:

convolve
colorizer
orthogonal slicer

SEE ALSO

The example script GENERATE FILTERS demonstrates the **generate filter** module.

NAME

generate histogram – plot distribution of data values in a scalar field

SUMMARY

Name	generate histogram
Type	filter
Inputs	field *any-dimension* scalar *any-data any-coordinates*
Outputs	field 1D scalar integer uniform

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Number of Bins	integer dial	256	1	1024
Min Bin	float dial	0.0	*unbounded*	*unbounded*
Max Bin	float dial	255.0	*unbounded*	*unbounded*
Choice	choice	histogram		
Normalize	boolean	on		

DESCRIPTION

The **generate histogram** creates an output field that characterizes the distribution of data values in a scalar field. This output field is intended to be plugged into the **graph viewer** module to be plotted, either as a curve or a bar graph.

Picture an "empty" bar graph. The **Min Bin** and **Max Bin** dial settings determine the range of data values that will be counted. **Number of Bins** determines how many discrete chunks ("bins") the whole range of data values in the input field will be divided into. (**Max Bin** - **Min Bin**) / **Number of Bins** determines the range of each chunk.

generate histogram reads the input field and examines each value. It decides which sub-data range bin the value would fit in, and increments the integer count for that bin by one. If the value is below **Min Bin** or above **Max Bin**, it is discarded.

The result of this is a 1D field of **Number of Bins** elements, with each element an integer count of the number of original data values that fell into that range.

Alternatively, if **cumulative** was selected instead of **histogram**, each bin count reflects its own count *plus* the count of all previous bins.

In either case, the output field should be connected to the **graph viewer** module's rightmost "linear plot" port.

INPUTS

Data Field (required; field *any-dimension* scalar *any-data any-coordinates*)
: A scalar AVS field whose distribution of data values is to be counted.

PARAMETERS

Number of Bins
: An integer dial that determines how many chunks the range of data values is to be divided into. The default is 256. The minimum allowable is 1, the maximum is 1024.

Min Bin
Max Bin
: Two floating point dials that set the endpoints of the range of data values to count. If **Normalize** (default) has been selected, the **Min Bin** and **Max Bin** dials will be initially set to the actual minimum and maximum data values in the input data. Without **Normalize** **Min Bin** is initially set to 0.0, and **Max Bin** to 255.0. This parameter is unbounded.

Normalize
: The **Normalize** switch determines whether the **Min Bin** and **Max Bin** dials will be automatically set to the actual minimum and maximum data values in the field. Without **Normalize**, you would need to have some idea of the real data value range in the input field so that you could set the dials in a way that would not inadvertently discard data. With **Normalize** on, **generate histogram** examines the input field's data structure to see if minimum and maximum values have been specified. If they are present, it uses them. If they are not present, it calculates the actual

minimum and maximum in order to set the dials.

When **Normalize** is on the **Min Bin** and **Max Bin** dials can not be used; if they are moved, they will "snap back" to their original values. **Normalize** is on by default.

histogram
cumulative

A choice that decides how the data values are counted. If **histogram** (the default) is chosen, each bin contains a count of the number of data values that fell into its sub-range. If **cumulative** is selected, each bin contains a count of the number of data values that fell into its sub-range, *plus* the total of all bins preceding it.

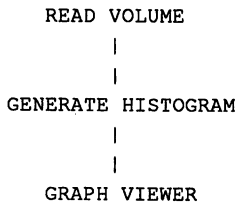
OUTPUTS

Data Field (field 1D scalar integer uniform)

The output field is a 1D field, **Number of Bins** long, with each element an integer count of the number of data values that fell into its range. It is used as "One Column" input to the **graph viewer** module's rightmost input port.

EXAMPLE 1

The following network reads in a volume (byte data in the range 0 to 256), calculates the distribution of values, and graphs the result:



RELATED MODULES

Modules that could provide the **Data Field** inputs:

Any module that outputs a field

Modules that can process **generate histogram** output:

graph viewer

See also **statistics**

SEE ALSO

The example scripts GENERATE HISTOGRAM and GRAPH VIEWER demonstrate the **generate histogram** module.

NAME

gradient shade – apply lighting and shading to colored data set

SUMMARY

Name gradient shade

Type filter

Inputs field 2D/3D 4-vector byte uniform *(colorized data)*
field 2D/3D 3-vector real uniform *(gradient supplied by* **compute gradient***)*
field 2D scalar float (transformation matrix) (optional)

Outputs field *same-dimension* 4-vector byte uniform *(shaded version of colorized data)*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
ambient		0.1	0.0	1.0
diffuse		0.8	0.0	1.0
specular	float	0.0	0.0	1.0
gloss		20.0	0.0	50.0
lt theta	float	0.0	none	none
lt off-ctr	float	0.0	none	none

DESCRIPTION

The **gradient shade** module accepts a colored 2D or 3D data set, along with its gradients (supplied by the **compute gradient** module). It applies a single light source to the colored data, then shades it.

The gradient at each location in the data field substitutes for the *surface normal*, which is used in traditional algorithms for lighting and shading surfaces. (A surface normal at a particular point on a surface is a vector perpendicular to the surface.)

Various shading styles are achievable using the lighting controls (see *PARAMETERS* below). These include creating shiny and matte surfaces, and controlling the location of the light source.

INPUTS

Data Field (required; field 2D/3D 4-vector byte uniform)
The input field is an image (2D pixel array) or a block of voxels (3D pixel array).

Gradient (required; field 3D 3-vector real uniform)
This field is the gradient of the **Data Field. Transformation Matrix** (optional, field 2D scalar float) The transformation matrix is applied to **gradient shade**'s light source, and is used to control the location of the light. This input has the same effect as the **lt theta** and **lt off-ctr** parameters.

PARAMETERS

The way in which all the following parameters determine the coloring of an object is described below.

ambient The contribution of ambient (uniform background) lighting to the color. When this is set to 0.0, all surfaces facing away from the light source are black. When this is set to 1.0, surfaces appear in their own colors, with no shading information present.

diffuse The contribution of diffuse (directional) lighting to the color.

specular The contribution of specular lighting to the color.

The ST1500 and ST3000 versions of this module do not generate specular highlights.

gloss The sharpness of the specular highlight. The larger this value, the smaller and sharper the specular highlights.

lt off-ctr The angle between the light source and the positive Z axis (which comes out of the screen at a right angle).

lt theta The angle between (1) the projection of the light source on the X-Y plane and (2) the positive Y axis. This value measures how much an off-center light source "swings around" the Z-axis.

With *lt theta* = 0.0 and *lt off-ctr* = 0.0, the light source is coming straight from the eye perpendicular to the data. A positive *off-ctr* value moves the light source "up" (in the positive Y direction); a negative value moves it "down".

The equation for calculating the intensity of light reflected by a spot of surface is:

$(int_{amb} * ambient) + (int_{diff} * diffuse * \cos(phi)) + (int_{diff} * specular * \cos^{gloss}(lt\ off\text{-}ctr))$

In performing this computation, **gradient shade**:

- Assumes that int_{amb} and int_{diff} are both maximal (1.0).
- Uses *lt theta* and *lt off-ctr* to compute *phi*, the angle between the surface normal (gradient vector) and the light source. The quantity cos(*phi*) is the attenuation (reduction) factor for the directional (diffuse) light.
- Computes the quantity $\cos^{gloss}(\alpha)$, the attenuation factor for the specular highlight.

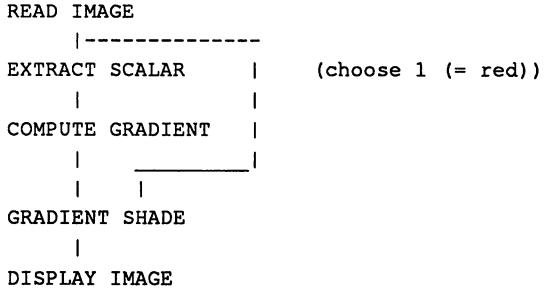
OUTPUTS

Data Field (field *same-dimension* 4-vector byte uniform)
The output field has the same form as the **Data Field** input.

The **min_val** and **max_val** attributes of the output field are invalidated.

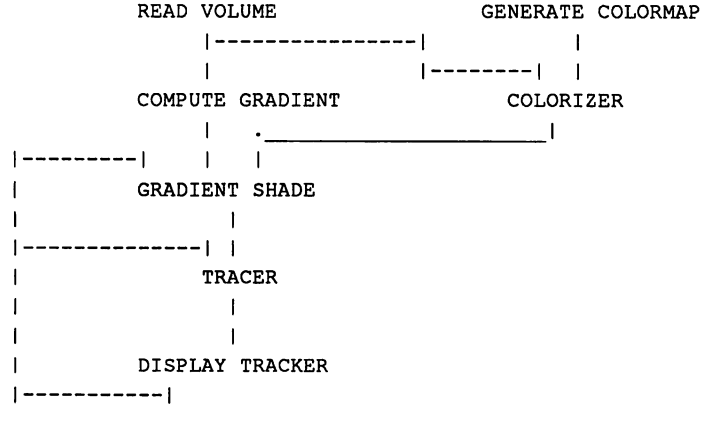
EXAMPLE 1

The following network shades a 2D image:



EXAMPLE 2

The following network shades a 3D image:



RELATED MODULES

Modules that could provide the **Data Field** input:

read volume

Modules that could provide the **Gradient** input:

compute gradient

Modules that could be used in place of **gradient shade**:

colorizer

Modules that can process **gradient shade** output:

display image (2D data)

Modules that can supply transformation matrices:

display tracker
euler transformation

See also **extract scalar**, which gets a single scalar height field from an image.

SEE ALSO

The example script ANIMATED FLOAT demonstrates the **gradient shade** module.

NAME

graph viewer – create XY and contour plots of data (Graph Viewer)

SUMMARY

Name	graph viewer
Type	data output
Inputs	field *any-dimension* scalar *any-data any-coordinates* (*linear data, optional*) field *any-dimension* scalar *any-data any-coordinates* (*contour data, optional*) field 2D 4-vector byte uniform (*background image, optional*)
Outputs	geometry
Parameters	none

DESCRIPTION

The **graph viewer** module provides access within an AVS network to the complete Graph Viewer subsystem. Many different modules can supply input data. That is, many *field*-format outputs can be connected to **graph viewer**'s input ports. Depending upon how **graph viewer** is set up, successive sets of incoming data will either replace an existing graph, be added to the graph, or be drawn in a new graph window.

You can also invoke **graph viewer** with no inputs, so that the graph is initially empty. Plots can be added to a graph either by upstream modules or by the various **Read Data** selections on the **graph viewer** control panel. Data sent by upstream modules can be saved to files in a variety of forms using the **Write ASCII XY Data**, **Write AVS Plot Data**, or **Write AVS Geometry Data** selections. In this way, you can save data plots and retrieve them later with **Read Data** selections. In addition, a PostScript image of the plot can be saved with the **Write PostScript** selection.

Note that the **graph viewer** window can be reparented to page and stack widgets using the AVS Layout Editor.

SPECIAL CONSIDERATIONS

This module is the module representation of the Graph Viewer subsystem. Instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level menu of the Graph Viewer subsystem. Thus, when you add the **graph viewer** icon to a network, no page is added to the Network Control Panel. There are two ways to access the Graph Viewer menu:

- Click the "dimple" in the **graph viewer** icon with the left mouse button.
- With the cursor positioned over the **Data Viewers** button located at the top of the Network Control Panel, press and *hold down* any mouse button. When the "AVS Data Viewers" pop-up menu appears, roll the mouse down to "Graph Viewer" and release the mouse button. This **Data Viewers** button is always visible, even when there is no active network.

In some circumstances, it is useful to be able to access both the Graph Viewer control panel and the Network Control Panel simultaneously. They both occupy the same screen position, along the left edge of the screen. In these cases, use the X Window System window manager to move one of these menu windows out of the way.

The **graph viewer**'s control panel also differs from that of other modules in these ways:

- The Network Editor's **Layout Editor** cannot be used to rearrange Graph Viewer controls.
- If a network includes more than one instance of **graph viewer**, AVS does *not* create a separate control panel for each instance. Each **graph viewer** sends its output to a different window, but the same Graph Viewer application menu controls all the windows. The module whose output window is currently highlighted in red is the one being controlled. To switch the *focus* to another **graph viewer** output window, just click in it with any mouse button.

RESIZING

The **graph viewer**'s pulldown menu, which is accessed by clicking on the "dimple" in the upper lefthand corner of the display window, provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen**. If the window originally was embedded in a page or stack, it will be re-embedded there.

INPUTS

Data Field (optional, field *any-dimension* scalar *any-data any-coordinates*)

The rightmost input port is for *linear* data that is to be made into an XY plot. If the input field is 1D, the values are taken to be Graph Viewer "plot as Y" data, meaning that they are interpreted as Y values that will be graphed against an evenly-spaced set of X values. If the input field is 2D, the values are taken to be Graph Viewer "plot as Y" data, meaning that they are interpreted as X and Y values. Although the **graph viewer** will accept fields of more than 2D, it will only graph the first two dimensions and ignore the rest. Many modules can create 2D subsets of fields (**orthogonal slicer** is an example). If such a module is used twice in succession (Example 2 below) a 1D subset of the field is created. The simplest example is **orthogonal slicer**. Note that the values at each point must be scalar. If you have a vector field, you must use **extract scalar** or a module with similar effect to produce a scalar version of the field.

Data Field (optional, field *any-dimension* scalar *any-data any-coordinates*)

The center input port is for *contour* data that is to be made into a contour plot. If the input field is 2D, the values are taken to be Graph Viewer "plot as contour" data that is interpreted as X and Y values. There is no size limit on the input file, but if it is large you will get a warning message. The real limit is the size of available memory. Note that the values at each point must be scalar. If you have a vector field, you must use **extract scalar** or a module with similar effect to produce a scalar version of the field.

Image (optional, field 2D 4-vector byte uniform)

The leftmost input port accepts an AVS image. **graph viewer** normally plots its graphs against a black background. If you send an image into this port, it will be used as the background instead, and the plot window will be resized to match the image size.

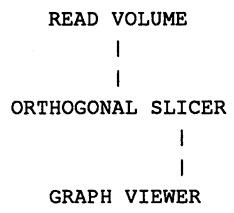
OUTPUTS

Geometry (optional; geometry)

graph viewer can produce PostScript file versions of plots for hardcopy printing with its **Write Postscript** selection. If you want to create output that will print or display correctly on a different device, this output port leaves the option open for a module that converts AVS geometry-format files to the format of another type of device. You could also use the **render geometry** module followed by the **pixmap to image** module to produce an image version of a **graph viewer** plot for the **image viewer**.

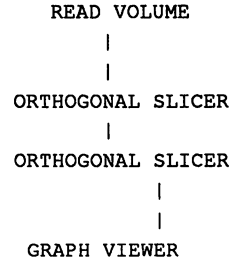
EXAMPLE 1

This network reads a volume, then uses **orthogonal slicer** to section out a 2D slice of the volume for plotting as X and Y data. Note that if **graph viewer** is set up to *add* each additional set of data to an existing plot, one could then manipulate the **orthogonal slicer's slice plane** dial to get a single graph with multiple plot lines showing successive slices through the volume.



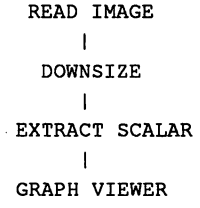
EXAMPLE 2

This network reads a volume, then uses the **orthogonal slicer** module *twice* to extract a 1D slice through the volume data:



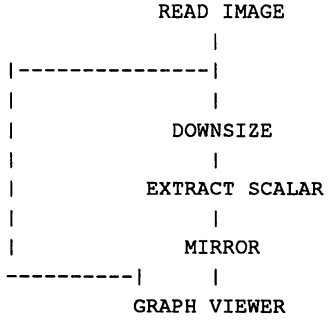
EXAMPLE 3

This network reads an image, downsizes the image to a reasonable resolution for graphing, then extracts the "red" data channel from the 4-vector image representation. This data is fed to **graph viewer's** middle (contour) input data port, and a contour plot of the reds in the image is displayed.



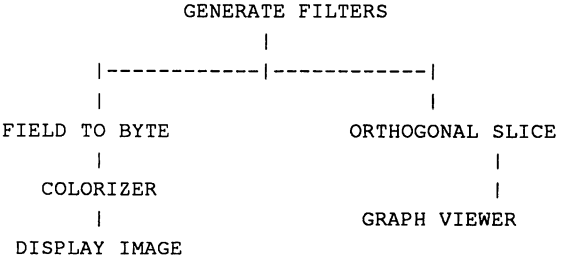
EXAMPLE 4

This network does the same as above, but displays the contour plot on top of the *mandrill.x* image it is a contour of. As with the network above, downsize the image to some reasonable size, and extract either the red, green, or blue bytes from it. The **mirror** module is necessary because 0,0 for an image is located in the upper left corner, while 0,0 for a graph is located in the lower left corner. To flip the image top to bottom, like this, select Y on the **mirror** module. The contour data is fed to **graph viewer's** middle (contour) input data port, and the image is fed in **graph viewer's** leftmost (image) input data port.



EXAMPLE 5

This network plots a section through the Gaussian image-processing filter produced by **generate filters:**



RELATED MODULES

generate histogram

SEE ALSO

The *Graph Viewer* chapter of the *AVS User's Guide.*

Two example GRAPH VIEWER scripts demonstrate the **graph viewer** module.

NAME

hedgehog – show vectors in a 3D 3-vector field

SUMMARY

Name hedgehog

Type mapper

Inputs field 3D 3-vector *any-data* uniform
field irregular 3-space (optional, from samplers module)
upstream transform *(optional, invisible, autoconnect)*

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*MaxValues*
Vector Scale	float	1.0	0.01	10.0
N segments	integer	16	2	64
Method	radio	point		point, trilinear
Sample	radio	point		point, line, circle, plane, space

DESCRIPTION

The **hedgehog** module takes as input a 3D uniform field whose values are 3-vectors of any primitive data type. That is, the data represents a volume of lattice points, each point having a 3D vector of *float* values. This 3D-vector value can be visualized as a small line segment with a particular length and direction.

The **hedgehog** module takes an arbitrarily-oriented (user-controlled) sample of locations within the volume. The sample object can be moved like any other geometry object. To select it, click on it with the left mouse button, or enter the Geometry Viewer and make it the current object. You can choose this sample to be:

- A single point
- A set of points on a line segment
- A set of points on a circle
- A set of points on a plane
- A volume of points

The module outputs the line segment(s) representing the values of the vector field at the sample location(s). The lines have arrows at their ends, showing the direction of the vectors. Often, this collection of line segments resembles the coat of a hedgehog — hence the module's name.

Since arbitrarily oriented sample locations do not, in general, coincide with the lattice points in the data volume, an interpolation method is used to determine a field value based on the values of one or more nearby lattice points.

hedgehog can optionally receive input from the **samplers** module. **samplers** outputs a list of points in space, and these points become the starting location for advecting particles. When **hedgehog** receive input from the **samplers** module, the **N Segment** dial, and the **Sample** buttons disappear from the **hedgehog**'s control panel.

INPUTS

Volume Data (required; field 3D 3-vector float)
The input data must be a 3D field, representing a volume of points. The data value for each point must be a 3D vector of *floats*.

Sample Input (field irregular 3-space)
This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **hedgehog** will take these locations and display the data values associated with them. This input can be used instead of **hedgehog**'s Sample parameter.

Upstream Transform (optional, invisible, autoconnect)
When the **hedgehog** and **render geometry** modules coexist in a network, they communicate through a normally-invisible data port. "Hedgehog" shows up as an object in the Geometry Viewer. When you select the hedgehog object and move it, **render geometry** informs the **hedgehog** module what the sample's new location is, and the **hedgehog** module recalculates the location and data it is displaying accordingly. This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **hedgehog** module's sample of locations.

PARAMETERS

Vector Scale
The lengths of the line segments output by this module are proportional to this value.

N segments
An integer value which determines the number of points sampled by the line, circle, plane, or space sampling probe. This controls the density of line segments output by **hedgehog**.

Method (radio buttons) Controls the way in which the field value is determined at each sample location:

- If **point**, a nearest-neighbor algorithm is used. Each mesh vertex is assigned the value of the nearest point in the lattice.
- If **trilinear**, a trilinear interpolation is performed. The value at each vertex depends on the values at the eight lattice points that are the corners of the "enclosing cube". The trilinear interpolation method is more accurate but takes longer to compute, particularly at higher resolutions.

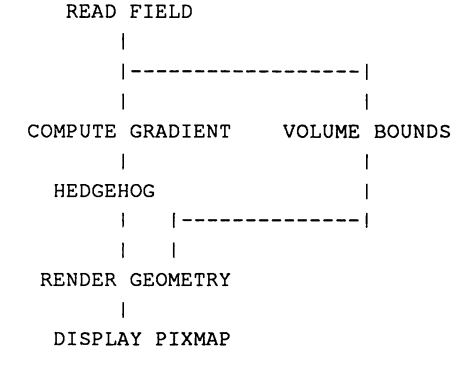
Sample (radio buttons) Specifies the type of sample taken from the vector field: point, line, circle, plane, or space.

OUTPUTS

Hedgehog (geometry)
The output *geometry* is a collection of line segments that represent the 3D-vector values at the sample locations. The line segments have arrows at their ends, indicating the direction of the vectors.

EXAMPLE

The following network visualizes the output of the **compute gradient** module.



RELATED MODULES

Data input:
read volume, volume manager
Gradient computation:
compute gradient
Vector operations:

vector curl, vector div, vector grad, vector mag, vector norm

Additional geometries:

volume bounds, isosurface

Geometric rendering:

render manager,
render geometry,
display pixmap

Sample Input:

samplers

SEE ALSO

The example script HEDGEHOG demonstrates the **hedgehog** module.

NAME

histogram stretch – balance the histogram of a data set

SUMMARY

Name	histogram stretch				
Type	filter				
Inputs	field *any-dimension* scalar byte *any-coordinates*				
Outputs	field of same type as input				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	histr_min	int	0	0	255
	histr_max	int	255	0	255

DESCRIPTION

histogram stretch is an image/volume processing module that balances the "histogram" of a data set between specified values. This operation combines histogram balancing (also called "histogram normalization" or "histogram equalization") and contrast stretching.

Finding the *histogram* of an image (or volume) consists of tallying the number of pixels (voxels) of each value into "bins". Byte data typically generates 256 bins (1 bin for each possible data value).

The *histogram equalization* process consists of trying to establish the same number of pixels (voxels) per bin by translating the pixel (voxel) values, using a well-chosen lookup table. This has the effect of creating an even distribution of values throughout the data set. It typically used to enhance low-contrast images (volumes) or images in which the data is "bunched up" at one end of the spectrum.

Equalization is applied only to values within the range specified by the parameters **histr_min** and **histr_max**. Data outside this range is not included in the histogram generation, and is eliminated.

INPUTS

Data Field (required; field *any-dimension* scalar byte *any-coordinates*)
The input data may be an AVS field of any dimensionality, each of whose values is a scalar *byte*.

PARAMETERS

histr_min Specifies the bottom of the range of input values that will be histogrammed, then transformed.

histr_max Specifies the top of the range of input values that will be histogrammed, then transformed.

OUTPUTS

Data Field The output field has the same form as the input field.

Appropriate new **min_val** and **max_val** values are written to the output field.

LIMITATIONS

This module works for *byte* fields only. (For other data types, there is no general way to determine the "right" number of bins to generate.) To apply this module to non-*byte* data, use the **field_to_byte** module to pre-process the data.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
field to byte

Modules that could be used in place of **histogram stretch**:

contrast stretch

Modules that can process **histogram stretch** output:

field to integer

field to float
field to double
any other filter module

SEE ALSO

The example script HISTOGRAM STRETCH demonstrates the **histogram stretch** module.

NAME

image compare – display two images together

SUMMARY

Name	image compare				
Type	filter				
Inputs	field 2D uniform 4-vector byte *(image)* field 2D uniform 4-vector byte *(image)*				
Outputs	field 2D uniform 4-vector byte *(image)*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Select	choice	vert_slice		
	Switch	toggle	off		
	valuator	float	0.5	0.0	1.0

DESCRIPTION

The **image compare** module lets you visually compare two images by displaying portions of those images together in one rectangular area in eight different ways—e.g. as two vertical slices, as two horizontal slices, in a checker pattern, etc. The main intent is to let you see "before" and "after" versions of the same image. One image is designated the "primary image," the other the "secondary image". You can flip back and forth between the dominant and secondary image using the **switch** parameter. In most cases, the **valuator** parameter controls the ratio of image 1 to image 2 appearing in the rectangle.

Both input images must have the same dimensions.

INPUTS

Image (required; field 2D uniform 4-vector byte)
One of the two images to compare.

Image (required; field 2D uniform 4-vector byte)
The other of the two images to compare.

PARAMETERS

selection Sets the way the two images are displayed together in the same rectangle.

vert_slice
vertical bands of the two images are displayed side by side.

horiz_slice
horizontal bands of the two images are displayed, one above the other.

diag_slice
slices from the upper left corner diagonally from one image to the next.

solid
disables the **valuator** dial described below.This lets you flicker between the images using the **switch** toggle described below.

circle
transforms the **valuator** dial to control the radius of a circle centered at the center of the image.

checker
creates a checkerboard pattern between the two images. The smaller the value showing on **valuator**, the more checks in the checkerboard.

venetian
creates alternating horizontal bands of image 1 and image 2.

random
randomly dithers between one image and the other based on the probability assigned by the **valuator** dial.

valuator The **valuator** dial controls the proportion of the rectangle viewing space that each image occupies. Allowable values are from 0.0 to 1.0, with the default 0.5 meaning "show half of one image and half of the other". As you move the dial, one or the other of the images gets more rectangle space.

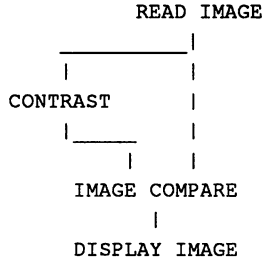
switch A toggle switch that exchanges the proportions of the screen given to image 1 and image 2.

OUTPUTS

Image (field 2D uniform 4-vector byte)
The output image is the patchwork combination of image 1 and image 2, with the same dimensions.

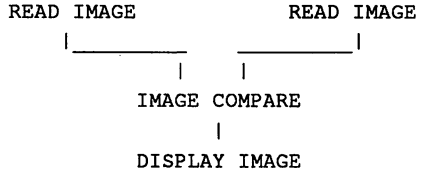
EXAMPLE 1

The following network compares an image with a contrasted version of itself:



EXAMPLE 2

The following network compares two images and displays the result through the **image viewer**. The images must be the same size.



RELATED MODULES

Modules that could provide the **image compare** inputs:

read image
Any module that produces an image as output

Modules that can process **image compare** output:

image viewer
display image
Any module that takes image input

See also **field math, constant blend**

SEE ALSO

The example scripts IMAGE COMPARE, and IMAGE II demonstrate the **image compare** module.

NAME

image manager – share images among subnetworks

SUMMARY

Name image manager

Unsupported this module is in the unsupported library

Type data

Inputs none

Outputs field 2D 4-vector byte (image)

Parameters

Name	*Type*	*Choices*
IMAGMGR Select	choice	Select, Replace
Image Manager	browser	
Image Choices	choice	

DESCRIPTION

The **image manager** module reads an image file from disk and outputs the image as a "field 2D 4-vector byte". It works like the **read image** module, except that it has both a cacheing mechanism and a way of sharing data among **image manager** modules in separate subnetworks.

See the **read image** manual page for a description of the image format.

PARAMETERS

IMAGMGR Select

A choice that determines how newly-read images will be placed to the list of currently active images:

- If **Select** is chosen, a new image is added to the end of the list.
- If **Replace** is chosen, a new image replaces the currently selected member on this list.

In either case, the change is reflected in *all* the *image manager* modules in all active subnetworks.

Image Manager

A file browser that allows you to select an image file to read.

Image Choices

A set of choices, listing each of the currently active images.

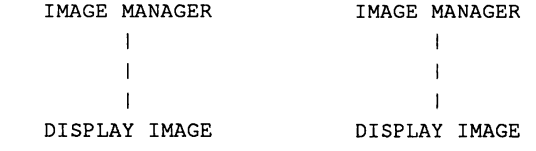
OUTPUTS

Data Field (field 2D 4-vector byte)

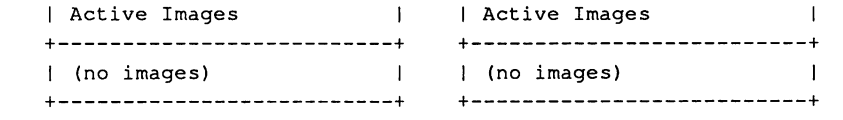
The output is an AVS *image*.

EXAMPLE

The following subnetworks might be used to display two images:



In this case, both **image manager** managers would contain "select/replace" choice buttons, a file browser, and an area below the browser:



Once a file (e.g. *heart_slice_22*) is selected with the browser in the **image manager** on the left, these buttons would look like this:

```
+--------------------------+    +--------------------------+
| * heart_slice_22         |    |   heart_slice_22         |
+--------------------------+    +--------------------------+
```

If a different file (e.g. *heart_slice_10*) is chosen from the browser in the **image manager** on the right, the buttons would look like this:

```
+--------------------------+    +--------------------------+
| * heart_slice_22         |    |   heart_slice_22         |
|   heart_slice_10         |    | * heart_slice_10         |
+--------------------------+    +--------------------------+
```

By selecting the same active image, you can have both networks display the same image:

```
+--------------------------+    +--------------------------+
| * heart_slice_22         |    | * heart_slice_22         |
|   heart_slice_10         |    |   heart_slice_10         |
+--------------------------+    +--------------------------+
```

Now, if you want to replace this image with a new one, click on the **Replace** buttons above the browser, then select a new file (e.g. *kidney_slice_04*) in just one of the **image manager** browsers. The result is that all **image manager** modules with the old image (*heart_slice_22*) selected as their active image will be automatically updated with the new image (*kidney_slice_04*):

```
+--------------------------+    +--------------------------+
| * kidney_slice_04        |    | * kidney_slice_04        |
|   heart_slice_10         |    |   heart_slice_10         |
+--------------------------+    +--------------------------+
```

RELATED MODULES

Same as for **read image**.

LIMITATIONS

The cached images are not freed until all *image manager* modules are destroyed. This is not the case with **read image** — the old data is freed whenever a new file is read.

NAME

image to pixmap – convert image to pixmap

SUMMARY

Name	image to pixmap			
Type	mapper			
Inputs	field 2D 4-vector byte uniform			
Outputs	pixmap			
Parameters	*Name*	*Type*	*Default*	*Choices*
	Approximation Technique (16-plane system only)	choice	none	none, dithering random, monochrome

DESCRIPTION

The **image to pixmap** module takes as input an *image* ("field 2D 4-vector byte") and outputs the same image as a *pixmap*. It is useful for converting the output of modules that produce images into modules that require pixmaps.

The *image* and *pixmap* data types differ in these major ways:

- Images are allow for efficient direct manipulation by a module, whereas pixmaps allow for efficient manipulation by the display device.
- Pixmaps are directly usable by a display device (under control of the X server). In X terminology, pixmaps contain "pixel values", images contain "colors". This difference is important only for 16-plane pseudo-color systems, in which pixmap values are interpreted as indices into the system's color lookup table. An image contains 24-bit color values, which cannot be used on such systems, which have only 12 color planes.
- A pixmap is represented by an X Window System *resource id* (an integer). This means that transferring a pixmap from one module to another is more efficient than transferring all the data that defines an image.

See the **read image** manual page for a description of the AVS image format.

INPUTS

Data Field (required; field 2D 4-vector byte uniform)
The input field must be an AVS *image*.

PARAMETERS

This module has the following parameter only when running on a 16-plane system.

approximation technique (16-plane systems only)
Controls the conversion of color values to pixel values. There are four approximation techniques:

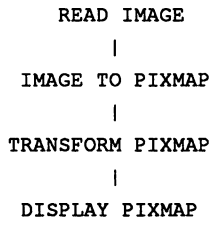
- **dithering**: uses a dither matrix to approximate each color
- **random**: uses a random number dither to approximate each color
- **monochrome**: uses the luminance of the color as an index into a greyscale ramp
- **none**: takes the closest approximation for each color

OUTPUTS

pixmap The output is an AVS *pixmap*.

EXAMPLE

This network allows an image to be displayed in an arbitrary-sized window:



RELATED MODULES

pixmap to image, transform pixmap, display pixmap

SEE ALSO

The example script OUTPUT POSTSCRIPT demonstrates the image to pixmap module.

NAME

image viewer – display and manipulate collections of images (Image Viewer)

SUMMARY

Name	image viewer
Type	data output
Inputs	field 2D 4-vector byte uniform (*image, optional, multiple*)
Outputs	none
Parameters	none

DESCRIPTION

The **image viewer** module provides access within an AVS network to the complete Image Viewer subsystem. Many different modules can supply the input images. That is, many *image*-format outputs can be connected to the **image viewer**'s image input port. All the images will be combined into a single current scene.

You can also invoke **image viewer** with no inputs, so that the "scene" is initially empty. Images can be added to a scene either by upstream modules or by the **Read Image** selection on the **image viewer** control panel. Images sent by upstream modules can be saved to files using the **Write Image** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Image**.

The Image Viewer's Action submenu can create simple "flip book" animations. You can send a series of images from upstream modules into the **image viewer** and have it turn them into a simple animation.

Note that the **image viewer** window can be reparented to page and stack widgets using the AVS Layout Editor.

RESIZING

The **image viewer**'s pulldown menu, which is accessed by clicking on the "dimple" in the upper lefthand corner of the display window, provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen**. If the window originally was embedded in a page or stack, it will be re-embedded there.

SPECIAL CONSIDERATIONS

This module is special: instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level menu of the Image Viewer subsystem. Thus, when you add the **image viewer** icon to a network, no page is added to the Network Control Panel.

There are two ways to access the Image Viewer menu:

- Click the small square in **image viewer** icon with the left mouse button.
- With the cursor positioned over the **Data Viewers** button located at the top of the Network Control Panel, press and *hold down* any mouse button. When the "AVS Data Viewers" pop-up menu appears, roll the mouse down to "Image Viewer" and release the mouse button. This **Data Viewers** button is always visible, even when there is no active network.

In some circumstances, it is useful to be able to access both the Image Viewer control panel and the Network Control Panel simultaneously. They both occupy the same screen position, along the left edge of the screen. In these cases, use the X Window System window manager to move the one of these menu windows out of the way.

The **image viewer**'s control panel also differs from that of other modules in these ways:

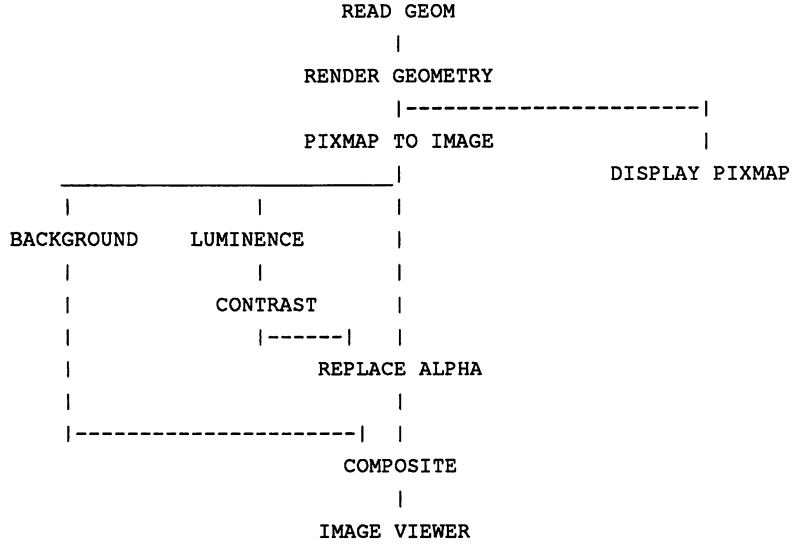
- The Network Editor's **Layout Editor** cannot be used to rearrange Image Viewer controls.
- If a network includes more than one instance of **image viewer**, AVS does *not* create a separate control panel for each instance. Each **image viewer** sends its output to a different window, but the same Image Viewer menu controls all the windows. The module whose output window is currently highlighted in red is the one being controlled. To switch the *focus* to another **image viewer** output window, just click in it with any mouse button.

INPUTS

Image (optional, multiple; field 2D 4-vector byte uniform)
The input data can be any AVS *image*. More than one image can be input to this port. All the images will be combined into the same "scene".

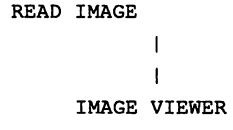
EXAMPLE 1

This network receives a series of images of what were originally AVS geometry objects, composites them over a background image, and creates a simple animation as the user manipulates the geometry object:



EXAMPLE 2

The following network reads in an image and then sends it to the **image viewer** module. This lets you apply all of the imaging techniques of the **image viewer** to the image.



RELATED MODULES

display image
read image
pixmap to image

SEE ALSO

The Image Viewer chapter in the *AVS User's Guide*.

NAME

integer – send a user-entered integer to one or more module(s) integer parameter port

SUMMARY

Name	integer				
Type	data				
Inputs	none				
Outputs	integer				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Integer Value	dial	0	*unbounded*	*unbounded*

DESCRIPTION

The **integer** module sends a single user-specified integer value to one or more integer-type parameter ports on one or more receiving modules. Its purpose is to make it possible for you to simultaneously control integer parameter input to more than one module using only a single input widget (whether the default dial, or a typein).

Before you can connect **integer** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **integer** module icon in the usual way.

PARAMETERS

Integer Value (integer)

The single user-supplied integer value to be sent to the module(s) integer parameter port(s). The default value is 0. There is no minimum or maximum restriction on the value. You should be aware of the range of numbers that it is reasonable to send to the receiving modules. The default widget type is a dial.

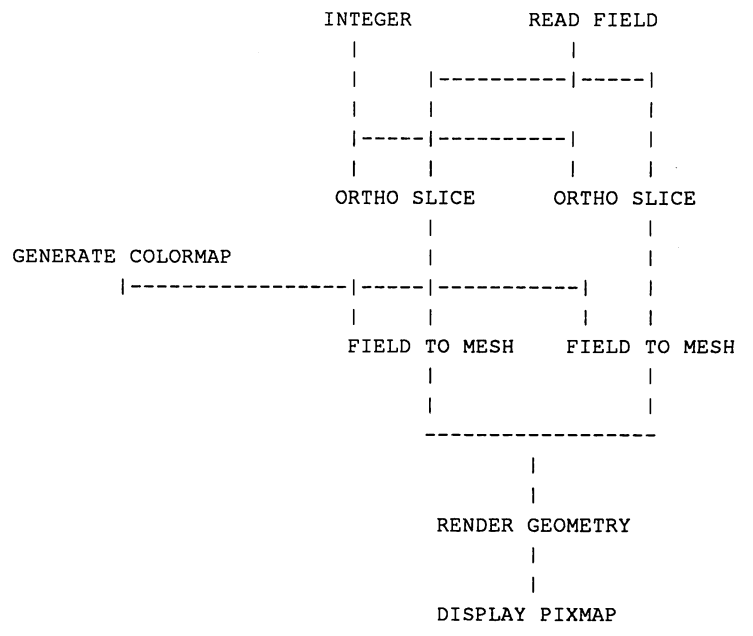
OUTPUTS

Integer (integer)

The integer value is sent to all modules with integer-type parameter ports connected to the **integer** module

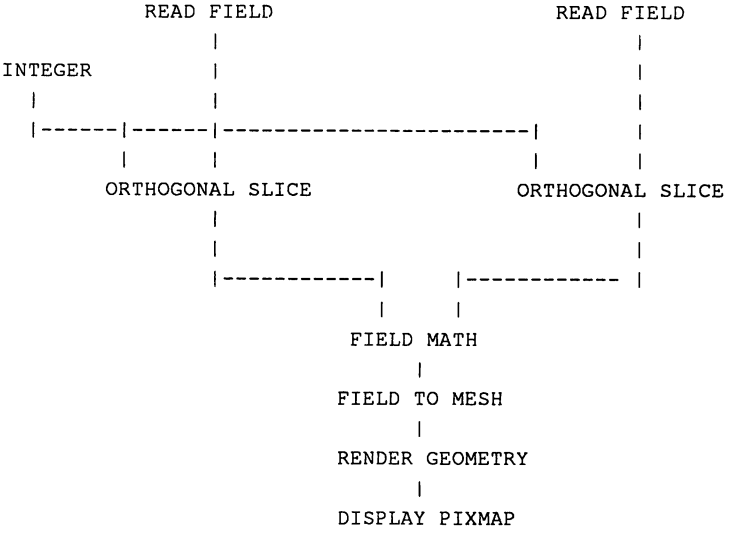
EXAMPLE 1

The following network reads a field, then creates two orthogonal slices through the field in different planes (one in I and one in J) using the **integer** module to specify the same offset slice plane to both slicers. The resulting planes are converted to meshes and composited together in the render geometry window.



EXAMPLE 2

This example reads two different fields, uses the **integer** module to specify the same slice plane in both to the **orthogonal slicer** modules, then uses **field math** to produce a new field that is the difference between them.



RELATED MODULES

Modules that can process **integer** output:

all modules with integer-type parameter ports

SEE ALSO

The example scripts INTEGER, FIELD TO BYTE, as well as others demonstrate the **integer** module.

NAME

interpolate – compute intermediate values to change the size of a field

SUMMARY

Name interpolate

Type filter

Inputs field 2D/3D scalar byte *any-coordinates*

Outputs field *same-dimension* scalar byte *same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*MaxChoices*
interp_sx	float	1.0	0.0	4.0
interp_sy	float	1.0	0.0	4.0
interp_sz	float	1.0	0.0	4.0
sampling	choice	Point		Point, Bi/Trilinear

DESCRIPTION

The **interpolate** module arbitrarily changes the size of its input data, either by sub-sampling or interpolating it. This module is useful for smoothly scaling the data arbitrarily up and down.

The interpolation algorithm first selects, for each output point, its real (floating-point) position in the input data set:

```
New X = Old X * interp_sx
New Y = Old Y * interp_sy
New Z = Old Z * interp_sz
```

With the *point sampling* method, it then selects the closest pixel (voxel) to the computed one. With *bilinear* (in 2D) or *trilinear* (in 3D) sampling, it finds the four pixels (2D) or eight voxels (3D) around the computed point and does a linear sampling for in-between pixels.

The point sampling mode is much quicker than the linear sampling and should be used when interactivity is more important than image quality.

INPUTS

Data Field (field 2D/3D scalar byte *any-coordinates*)
The input field may be 2D or 3D. The data for each element must be a single byte. The field can be uniform, rectilinear, or irregular.

PARAMETERS

interp_sx
interp_sy
interp_sz (does not appear for 2D input fields)
The interpolation factors for the coordinate dimensions.

sampling This choice determines the sampling method, **Point** or **Bi/Trilinear**, as described above.

OUTPUTS

Data Field The output field has the same form as the input field. Appropriate new **min_ext** and **max_ext** values are written to the output field.

RELATED MODULES

This module is similar to **downsize** (which does uniform, stride-based point sampling) and **crop** (which selects a subset of the data but doesn't change the resolution). Some advantages to using this module are: it can scale non-uniformly in each dimension; it can do high-quality linear sampling; and it can scale data up instead of only down.

LIMITATIONS

This module does the wrong thing when down-sampling (going from a large image to a small one) in the Bi/Trilinear mode. What it should do is "average" appropriately chosen regions down to each pixel. What is does is to choose the four pixels around the center of that region and interpolate between them. This is not a huge

error, but it is not strictly correct.

SEE ALSO

The example script INTERPOLATE demonstrates the **interpolate** module.

NAME

isosurface – generate an isosurface for a volume of data

SUMMARY

Name	isosurface
Type	mapper
Inputs	field 3D scalar *any-data any-coordinates* field 3D scalar *any-data* (optional) colormap (optional)
Outputs	geometry

Parameters

Name	*Type*
Isosurface Level	float
Optimize Surface Description	toggle
Optimize Wireframe Description	toggle
Flip Normals	toggle

DESCRIPTION

The **isosurface** module inputs a volume data set (3D field of values, either curvilinear, rectilinear, or uniform). It produces a geometric object that represents an isosurface of this object. An *isosurface* is a 3D generalization of a 2D contour line — it connects all field elements that have the same parameter-controlled data value.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)
The input data must represent a volume, with a single value of any primitive data type for each field element.

Auxiliary Data Field (optional; field 3D scalar *any-data*)
This port can be used to generate a colored isosurface; the color at each point on the surface indicates the value of another attribute of the volume. For instance, you could generate a pressure isosurface with colors indicating the temperature at each point on the surface.

In this case, the **Data Field** would be used to input the pressure data, and the **Auxiliary Data Field** would be used to input the temperature data. In all cases, both volume data sets must have the same dimensions.

Colormap (optional; colormap)
If you use an **Auxiliary Data Field**, you must also specify a colormap. Since the auxiliary volume data is floating-point, you must adjust the **lo value** and **hi value** parameters of the **generate colormap** module to correspond to the minimum and maximum data values of the auxiliary field.

For the pressure-temperature example described above, the temperature data set might have data values in the range 0.0–100.0 degrees. In this case, set the **lo value** to 0 and **hi value** to 100 in **generate colormap**.

PARAMETERS

Isosurface Level
A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the field element's data value equals the **Isosurface Level** value.

Optimize Surface Description
Optimize Wireframe Description
These two toggle parameters allow you to control a tradeoff between how efficiently the isosurface is computed and how efficiently it can be rendered. If you turn on **Optimize Surface**, extra time will be spent generating a more optimal surface description, containing fewer triangles.

Turn on **Optimize Wireframe** to generate a wireframe representation for the isosurface along with the shaded surface representation. If you want

to view your surface as a wireframe (using the **Objects** selection in the **render geometry** control panel), you must toggle this on.

Flip Normals

Reverses the direction of each surface normal in the generated isosurface. If the normals point in the wrong direction, the outside of the isosurface will appear at the ambient light intensity. In this case, click this button or specify bi-directional lighting in the **render geometry** control panel (**Lights** selection).

OUTPUTS

Isosurface (geometry)

A shaded surface, optionally with an associated wireframe representation.

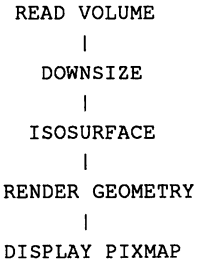
NOTES

The most important parameter is the **Isosurface Level** (threshold), which is defined in the unbounded floating-point data space of the volume. It is not always easy to know in what range the data is defined. Often, the data is defined some well-known real-world domain (e.g. temperature in degrees). In some cases, the data has been converted from *byte* data, and therefore must lie within the range 0.0–255.0.

Because **isosurface** is compute-intensive, it is often advisable to include a **downsize** module in the network. This allows you to quickly select a proper isosurface level before generating one at full resolution.

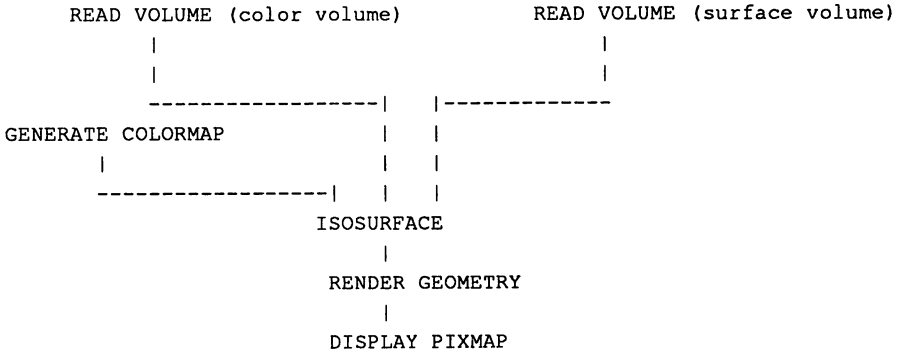
Another technique is to use the **Action** capability of the Geometry Viewer (**render geometry** module) to save and play back a sequence of isosurfaces at different value levels.

EXAMPLE 1



EXAMPLE 2

This example uses an auxiliary data set.



RELATED MODULES

render geometry, downsize, generate colormap, read field, read volume

LIMITATIONS

In some circumstances, the generated isosurface may have some of its normals pointing inward and some outward. There is no way to correct this situation, but usage of bi-directional lighting (**Lights** selection of the Geometry Viewer/**render geometry**) may be helpful.

SEE ALSO

The example script FIELD LEGEND demonstrates the **isosurface** module.

NAME

local area ops - image processing based on pixel neighborhoods

SUMMARY

Name	local area ops				
Type	filter				
Inputs	field 2D 4-vector byte uniform (*image*) OR field 1-3D scalar *any-data any-coordinates*				
Outputs	field of same type as input				
Parameters	*Name*	*Type*	*Default*	*Min*	*MaxChoices*
	kernel width	integer	3	3	31
	choice	choice	Min		Min, Max, Median, Mean

DESCRIPTION

local area ops contains four operations used in image processing, each of which takes an input field and computes an output image using some function. In a "local area operation" the value of each pixel in the output image is based on the values of pixels in its immediate neighborhood. The kernel is the NxN neighborhood of pixels surrounding each pixel used to calculate each new pixel value. The "width" of the kernel thus determines the size of this neighborhood.

In the operation **Min**, for example, using a filter width of 3, the value of each pixel in the output image becomes the minimum value of the pixel and the 8 pixels surrounding it.

In the case of an image, which is a 2D field of 4-byte vectors, **local area ops** disregards the alpha bytes and separates the red, green, and blue bytes. Then it applies the operation separately to each color byte, before reassembling the bytes into 4-vector image format. The status bar shows the module processing three times, once for each color byte.

Apart from AVS images **local area ops** handles only scalar values of any data type. All data-types are converted to floats during computation and then converted back in the output of **local area ops**.

In order to handle edge effects, a border around the perimeter of the image is not operated on. The border is half the width of the kernel.

INPUTS

Data Field (required; field 2D 4-vector byte uniform (*image*)) OR
Data Field (required; field 1-3D scalar *any-data any-coordinates*) Typically, the input will be an AVS *image*, which is a 2D field of 4-vector bytes.

The input may be any 1-3D field of scalar values of *any-data any-type*.

PARAMETERS

choice sets which local area operation to apply. There are 4 options:

Min
In the **min** operation each pixel in the output image becomes the minimum of the pixels in its immediate neighborhood. This has the effect of shrinking light regions of an image, and is refered to as a "region shrinking" operation.

Max
In the **max** operation each pixel in the output image becomes the maximum of the pixels in its immediate neighborhood. This has the effect of enlarging light regions of an image, and is refered to as a "region growing" operation.

Median
In the **median** operation the pixels in the neighborhood are sorted. Then the pixel at the center of the neighborhood gets the value that is in the middle value of the sorted array. This has an effect similar to

the mean operation, but it can be especially useful in removing noise from an image, since anomalies are not likely to effect the output image. Note: since the **median** calculation requires a sort, it is very compute intensive, especially when the filter width is large. AVS puts up a warning message when the **median** operation is selected.

Mean
In the **mean** operation each pixel in the output image becomes the average of the pixels in its immediate neighborhood. This has the effect of reducing the contrast of an image between the light and the dark regions.

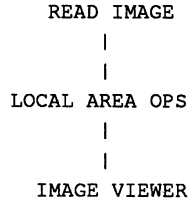
kernel width
Determines the size of the neighborhood of pixels contributing to the value of each pixel in the output image.

OUTPUTS

Output Field
The output field is the same type as the input data field.

EXAMPLE 1

The following network reads in an image, applies the local area operations to it, and displays the resulting image:



RELATED MODULES

Modules that could provide the **Data Field** input:

read image
pixmap to image
orthogonal slicer
any other module which outputs a field of scalars or an image

Modules that can process the output of **local area ops**:

display image
image viewer
any other module which takes a 2D field as input

SEE ALSO

The example script LOCAL OPS demonstrates the **local area ops** module.

NAME

luminence – compute the luminence of an image

SUMMARY

Name	luminence
Type	filter
Inputs	field 2D uniform 4-vector byte *(image)*
Outputs	field 2D uniform scalar byte
Parameters	none

DESCRIPTION

The **luminence** module computes the luminence (brightness) of an image, then outputs a 2-dimensional field of the same dimensions, but with a *scalar* byte value for each pixel in the original image instead of the full four-byte alpha, red, green, blue vector.

The **luminence** (I) is calculated as follows:

I = (0.299 * red) + (0.587 * green) + (0.114 * blue)

This luminence byte value can be used to produce a black and white version of the original image (with **colorizer**), or substituted back into the alpha byte of the original image (with **replace alpha**) to produce transparency effects.

INPUTS

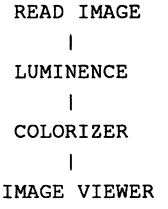
Image (required; field 2D uniform 4-vector byte)
The image whose luminence to calculate.

OUTPUTS

Data Field (field 2D uniform scalar byte)
The output field has the same dimension as the input image, but with a scalar byte value representing the image luminence at each original pixel instead of color value.

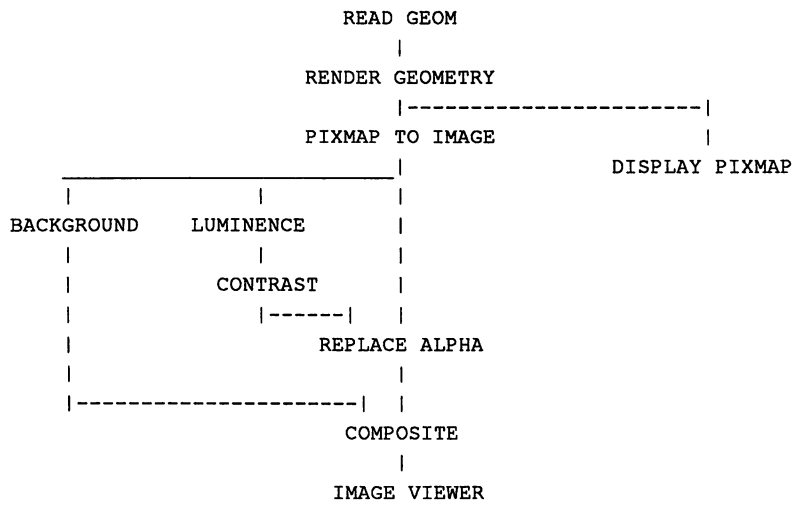
EXAMPLE 1

The following network reads an image, computes its luminence, colorizes the resulting field with the default black and white colormap, producing a black and white version of the original image. The result is displayed through the **image viewer**.



EXAMPLE 2

This network takes a geometry, displays it on the screen, then converts the screen pixmap to an image, computes its luminence, uses that to create an alpha mask, renders a shaded background and composites the rendered image over the shaded background. The **contrast** modules controls should be set to : minimum and maximum input contrast, both 1; minimum output contrast 0, and maximum output contrast, 255. If the original geometry were */usr/avs/data/geometry/jet.geom* and the **background** module were set to produce a sky-like pattern, this would produce a jet over a sky field.



RELATED MODULES

Modules that could provide the **Image** input:

Any module that produces an image as output

Modules that can process **luminence** output:

colorizer
contrast
Any modules that can process a 2D scalar field

See also **background, composite, replace alpha,** and **extract scalar**

SEE ALSO

The example script LUMINENCE demonstrates the **luminence** module.

NAME

mirror – reverse array indices in a 2D or 3D data set

SUMMARY

Name	mirror
Type	filter
Inputs	field 2D/3D *n-vector any-data any-coordinates*
Outputs	field of same type as input

Parameters

Name	*Type*	*Default*	*Choices*
axis	choice	Original	Original, X, Y, Z

DESCRIPTION

The **mirror** module reverses the array indexes along one dimension of a 2D or 3D field. This has the effect of creating a mirror image of the data set. In a 50 x 100 field, applying **mirror** to the X dimension does the following (in FORTRAN array notation):

```
INPUT(1,i) ---> OUTPUT(50,i)     (for all 100 values of i)
INPUT(2,i) ---> OUTPUT(49,i)
INPUT(3,i) ---> OUTPUT(48,i)
INPUT(4,i) ---> OUTPUT(47,i)
  ...
INPUT(50,i) ---> OUTPUT(1,i)
```

mirror can be used to change the orientation of the data for display and/or processing purposes.

To perform a reversal in two or more dimensions, use two or more **mirror** modules in succession.

INPUTS

Data Field (field 2D/3D *n-vector any-data any-coordinates*)
The input may be any 2D/3D scalar AVS *field*.

PARAMETERS

axis The choices for exchanging the data are:

- **Original** Copies the input to the output; no transformation is performed.
- **X** Reverses the array indices in the X dimension (first dimension).
- **Y** Reverses the array indices in the Y dimension (second dimension).
- **Z** Reverses the array indices in the Z dimension (third dimension). (Equivalent to **Original** for a 2D field.)

OUTPUTS

Data Field The output field as the same form as the input field.

RELATED MODULES

This module combined with **transpose** can re-orient the data in any desired way.

SEE ALSO

The example script GRAPH VIEWER demonstrates the **mirror** module.

NAME

offset – deform, or "blow up" a geometry object based on vector values at each node

SUMMARY

Name	offset				
Type	filter				
Inputs	geometry				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	offset	float	0.0	none	none

DESCRIPTION

The **offset** module transforms an AVS *geometry*, so that each vertex of each polygon is translated along its vertex normal. It is useful for emphasizing surface discontinuities (e.g. cusps) and for producing "blow ups" of objects.

INPUTS

Geometry (required; geometry) An AVS geometry, created with the *libgeom* library or by another AVS module.

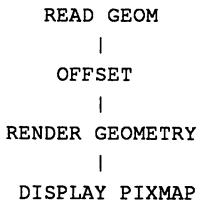
PARAMETERS

offset The amount by which each vertex is translated along its normal. Positive values create a "blow-up" of the geometry. Negative values collapse it.

OUTPUTS

Geometry A geometry that represents that same object(s) as the input data.

EXAMPLE



RELATED MODULES

read geom, flip normal, tube, render geometry

LIMITATIONS

This module works only for polytriangle strips and meshes, not for polyhedra. It has no effect on objects that do not have surface normals.

SEE ALSO

The example script OFFSET demonstrates the **offset** module.

NAME

oneshot - send a oneshot value to one or more module(s) "oneshot" parameter port(s)

SUMMARY

Name	oneshot				
Type	data				
Inputs	none				
Outputs	oneshot				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	oneshot	oneshot	0	0	unbounded

DESCRIPTION

The **oneshot** module sends a single user-specified "oneshot" value to one or more "oneshot" parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control "oneshot" parameter input to more than one module using only a single "oneshot" input widget.

oneshot outputs an integer which represents the number of times that **oneshot**'s parameter button was clicked in a certain time period. The length of the time period is not user controllable, but depends on the speed with wich AVS executes the network to which **oneshot** is connected. Thus, if AVS were executing a compute intensive network, you could click **oneshot**'s button 10 times. Then, **oneshot** will output the number 10 the next time it executes. Typically, **oneshot** is used as a signal to perform some operation.

Since oneshot data-type is not identical to an integer, **oneshot** can not be used to pass integer parameters.

Before you can connect **oneshot** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **oneshot** module icon in the usual way one connects modules.

PARAMETERS

oneshot (integer)
: The single "oneshot" value, specified through a "oneshot" button, to be sent to the receiving module(s) oneshot parameter port(s). The default value is zero.

OUTPUTS

oneshot (integer)
: The "oneshot" value is sent to all modules with oneshot-type parameter ports that are connected to the **oneshot** module.

RELATED MODULES

Modules that can process **oneshot**'s output:

all modules with oneshot-type parameter ports

SEE ALSO

The example scripts WRITE VOLUME and WRITE IMAGE demonstrate the **oneshot** module.

NAME

orthogonal slicer – slice through 3D or 2D field with plane perpendicular to coordinate axis

SUMMARY

Name	orthogonal slicer					
Type	mapper					
Inputs	field 3D or 2D *n-vector any-data any-coordinates*					
Outputs	field 2D or 1D *n-vector same-data same-coordinates*					
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*	*Choices*
	slice plane	int	0	0	255	
	axis	choice	K			I, J, K

DESCRIPTION

The **orthogonal slicer** module takes a 2D slice from a 3D array, or a 1D slice from a 2D array. It does so by holding the array index in one dimension constant, and letting the other index(es) vary. For instance, a data set might include a volume of 5000 points, arranged as follows (using FORTRAN notation):

```
DATA(I,J,K)          I = 1,10
                     J = 1,20
                     K = 1,25
```

You can take a 2D "I-slice" from this data set by setting *I*=4 and letting the other indices vary:

```
DATA(4,J,K)          J = 1,20
                     K = 1,25
```

The notation used in the example above assumes that the field's data values are scalars (in FORTRAN, DATA(4,5,6) must be a scalar). If fact, however, the **orthogonal slicer** module can takes slices of vector-valued fields, also. It passes through whatever data type is presented to it; e.g. if the input is a "field 3D 3-vector float", the output is a "field 2D 3-vector float".

INPUTS

Data Field (field 2D/3D *n-vector any-data any-coordinates*)
The input may be any 3D or 2D *field*.

PARAMETERS

slice plane Determines the value of the array index to be held constant. This value is reset to zero each time a new data field is input.

axis Selects the dimension (I, J, or K) in which the array index is to be held constant.

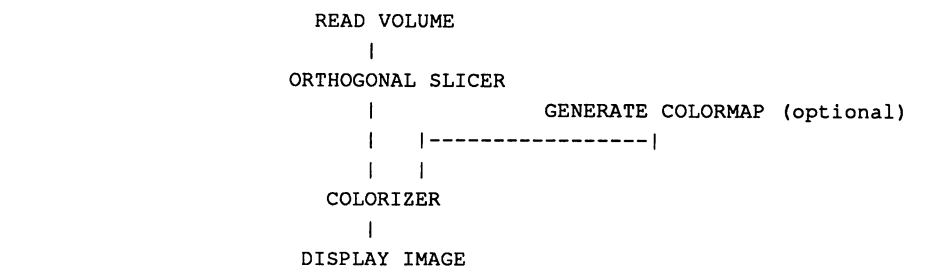
OUTPUTS

Data Field (field 1D/2D *n-vector any-data any-coordinates*)
The output field is 2D instead of 3D (or 1D instead of 2D), and has the same type of data as the input field.

Appropriate new values for **min_ext** and **max_ext** are written to the output field.

EXAMPLE 1

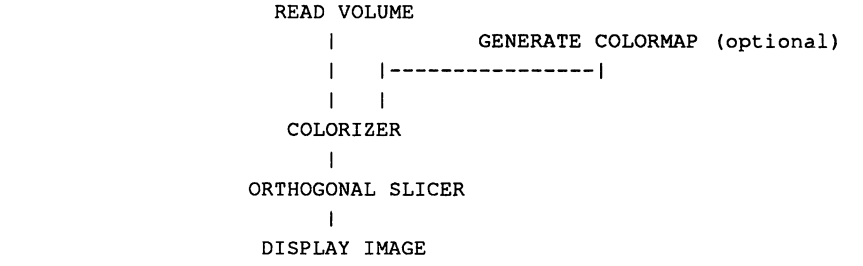
The following network takes a slice from a scalar volume and displays it:



The **colorizer** module is necessary because the output of **orthogonal slicer** is a "field 2D scalar byte", which must be cast into an AVS *image* field for display.

EXAMPLE 2

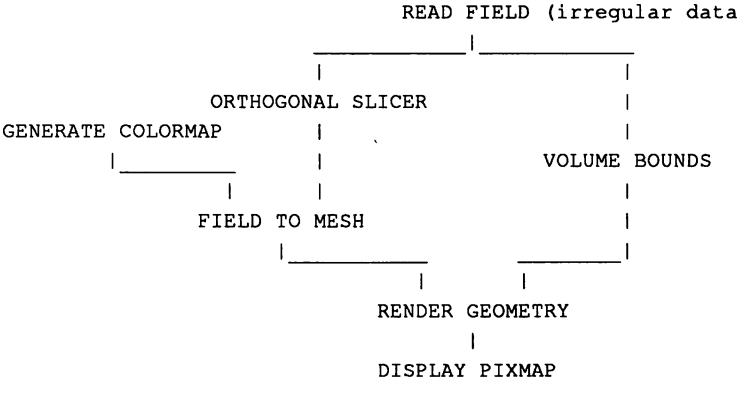
For reasonably small volumes, a better way to construct this network is:



This network has the effect of colorizing the entire volume once, which make the slicing operation more efficient. It does this at the expense of allocating more memory up front.

EXAMPLE 3

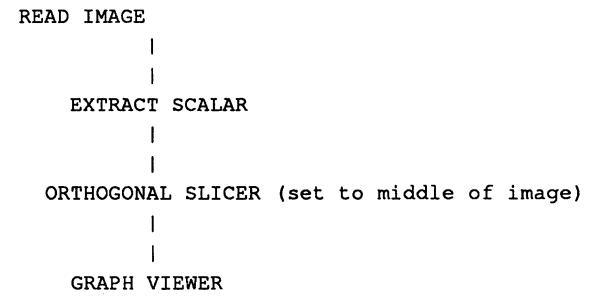
Irregular Data: orthogonal slicer supports the passing of "points" data for *rectilinear* and *irregular* data. This is an important module for visualizing curved data sets. For example:



(This is the reason for labeling the axis control with "I, J, and K": frequently, the data is *not* aligned to the X, Y, and Z axes. **orthogonal slicer** takes slices through the logical data set, not the physical one.)

EXAMPLE 4

The following network shows how to use **orthogonal slicer** to plot the values of one scan-line of an image:



RELATED MODULES

field to mesh
colorizer

SEE ALSO

The example scripts ANIMATED INTEGER, COLOR RANGE, and VECTOR CURL demonstrate the **orthogonal slicer** module.

NAME

output postscript – convert pixmap to PostScript™ and store in file

SUMMARY

Name	output postscript	
Type	data output	
Inputs	pixmap (required; pixmap)	
Outputs	*none*	
Parameters	*Name*	*Type*
	filename	browser
	mode	choice
	monochrome	toggle
	8 bit	toggle
	compress	toggle
	dither	toggle

DESCRIPTION

The **output postscript** module converts its input pixmap to the PostScript™ page description language and stores it in a file.

In the ST1500 and ST3000 versions, the window containing the picture to be output is mapped before the picture is saved.

After the file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

Two types of PostScript output are supported:

- Suitable for sending to a PostScript-compatible laser printer.
- Mathematica™ compatible.

INPUTS

pixmap (pixmap)
Any AVS pixmap.

PARAMETERS

filename A file browser that allows you to specify the name of the PostScript file to be created. After the file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

Mode Selects the type of PostScript output: **laserwriter** or **mathematica.**

The following toggle parameters control the creation of Mathematica PostScript files:

monochrome
If ON, produces monochrome output. If OFF, produces color output.

8 bit If ON, produces 8-bit output. If OFF, produces 4-bit output.

compressed
If ON, produces compressed output. If OFF, produces uncompressed output.

dither
If ON, produces dithered output. If OFF, produces undithered output.

EXAMPLE

This example converts an image to a PostScript file:

```
 READ IMAGE
     |
IMAGE TO PIXMAP
     |
OUTPUT POSTSCRIPT
```

RELATED MODULES

image to pixmap, render geometry, alpha blend

LIMITATIONS

The Mathematica compress option is not supported in any released version of Mathematica.

The dither option produces visual artifacts on some images.

There is no color option supported in this module.

COPYRIGHT

SEE ALSO

The example script OUTPUT POSTSCRIPT demonstrates the **output postscript** module.

NAME

particle advector – release grid of particles into velocity field

SUMMARY

Name particle advector

Type mapper

Inputs field 3D 3-vector float *any-coordinates*
field irregular 3-space (optional, from samplers module)
upstream transform *(optional, invisible, autoconnect)*

Outputs particles geometry
tracers geometry

Parameters

Name	*Type*	*Default*	*Min*	*MaxValues*
Mesh Res	integer	16	2	128
Tracer Length	integer	0	0	100
Time Step	float	0.0	0.0	1.0
Size	float	0.0	0.0	1.0
Advect Batch	oneshot			
Stop Advection	toggle	off		
Replay Advect	toggle	off		
Reset Particles	oneshot			
Show Bounds	toggle	on		
Color	toggle	off		
Surface	toggle	off		
Method	radio	Euler		Euler, Runge-Kutta
Tracer Style	choice	cap		cap, cycle, add

DESCRIPTION

The **particle advector** module takes as input a 3D 3-vector field of *float*s (e.g. fluid flow simulation data), and treats it as a velocity field. A batch of zero mass (the "sample") particles is *advected* (placed into the field at various initial positions with no initial direction or speed). The particles move through the velocity field according to the magnitude and direction of the vectors at the nodes in the volume. A forward differencing method is used to estimate the next position of each particle as a function of the current position and velocity.

This module is an AVS *coroutine* — it generates new data continuously, rather than waiting for a module upstream to pass it new data.

The starting position of the sample of particles is user controlled. If **particle advector**'s **Show Bounds** parameter is turned on, and **particle advector** is not connected to the **samplers** module (see description of **Upstream Transform input, below**), **the sample object, from which particles are advected, is visible. This object can be manipulated like any other geometry object. To select it, click on it with the left mouse button, or enter the Geometry Viewer and make it the current object.**

particle advector can receive input from the **samplers** module. **samplers** outputs a list of points in space, and these points become the starting location for advecting particles. When **particle advector** receives input from the **samplers** module, the **Mesh Res** dial, and the **Show Bounds** and **Surface** buttons disappear from the control panel. If **particle advector** does not receive input from the **samplers** module, particles can only be advected from a plane sample; the point, circle, and space options are not available.

Note that, using the **Stop Advection** button, it is possible to advect a batch of particles, stop their progress, reposition the sample plane, and then advect another batch with new parameter settings from a different location. Turn **Stop Advection** off to set

both groups of particles in motion.

INPUTS

Data Field (required; field 3D 3-vector float *any-coordinates*)
The input data must be a 3D field, representing a volume of points. The data value for each point must be a 3D vector of *floats*. The input field can be uniform, rectilinear, or irregular.

Sample Input (optional; field irregular, from **samplers** module)
This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **particle advector** uses these locations as the starting positions for advecting particles. If **particle advector** does not receive input from the **samplers** module, particles can only be advected from a plane sample; the point, circle, and space options are not available.

Upstream Transform (optional, invisible, autoconnect)
When the **particle advector** and **render geometry** modules coexist in a network, they communicate through a normally-invisible data port. "particle.advect" shows up as an object in the Geometry Viewer. When you select the particle.advect object and move it, **render geometry** informs the **particle advector** module what the sample's new location is, and the **particle advector** module recalculates the location and data it is displaying accordingly. This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **particle advector** module's sample of locations. Note that, when **particle advector** receives sample input from the **samplers** module, the bounds of the "particle.advect" object are not visible, and **particle advector's Show Bounds** parameter is disabled.

PARAMETERS

Various aspects of the particle advection process can be adjusted interactively.

Mesh Res The umber of particles is controlled by the **mesh res** parameter. The total number in each batch is **mesh_res * mesh_res.**

Tracer Length
Integer dial which controls the length of the tracer output which shows the trajectory of each advected particle. The default is 0; higher numbers produce longer tracers.

Time Step Adjusts a scalar that multiplies the magnitude of the vector along which each particle is travelling. This causes successive positions of particles to be more widely spaced. (See also the **Color** parameter.)

size Controls the radius of the particles, which are rendered as spheres.

The default *size* is zero; this causes the particles to be rendered as points (individual pixels).

Advect batch
Triggers the release of a batch of particles.

Stop Advection
Temporarily halts this module.

Replay Advection
Restarts the advection using the current settings of all parameters.

Reset Particles
Sets the total number of particles to zero.

Show Bounds
(toggle) Controls the visibility of the mesh of particles.

Color (toggle) If ON, colors the line segments to indicate how fast the particles are travelling through the velocity field:

red	fastest
yellow	
green	
cyan	
blue	stopped

Surface Creates a solid shaded mesh. The coloring scheme is the same as that used with the **Color** parameter.

method (radio buttons) The buttons **Euler** and **Runge-Kutta** select the method used to calculate the next position of a sample particle. The **Euler** method is faster, involving a single vector in the input field. The **Runge-Kutta** method involves an interpolation, and produces considerably more accurate results.

Tracer Style
(radio buttons) Specifies the form of the **tracers** output:

cap Short lines that show the beginning trajectory of each advected particle. The particles eventually "break free" of these lines, after which the particles continue to move, but the lines do not.

cycle Short lines that show the last few interations of the flow. These lines appear to be "tails" attached to the advected particles.

end Continuous lines that show the entire trajectories of the particles.

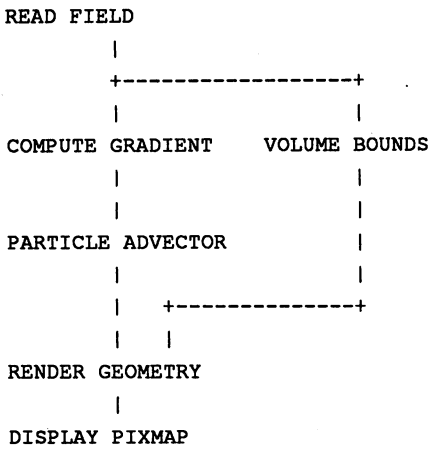
OUTPUTS

Particles Geometry
This output is an AVS *geometry* that represents the batch of particles advected into the input vector field.

Tracers Geometry
This output is a set of tracer lines (analogous to stream lines) produced by the sample particles. The **tracer style** parameter controls the form that these lines take.

EXAMPLE

In the following network, **read field** reads in a 3D scalar field, and **compute gradient** calculates a 3-vector for every field location.



RELATED MODULES

Vector operations:

vector curl, vector div, vector grad, vector mag, vector norm

Additional geometries:

volume bounds
arbitrary slice
isosurface

Geometric rendering:

render manager
render geometry
display pixmap

SEE ALSO

The example script PARTICLE ADVECTOR demonstrates the **particle advector** module.

NAME

pdb to geom – create molecule geometry from Protein Data Bank(PDB) file

SUMMARY

Name	pdb to geom		
Type	data		
Inputs	none		
Outputs	geometry		
Parameters	*Name*	*Type*	*Choices*
	Data file	browser	
	Render Mode	choice	ball and stick, ball, stick, colored stick, colored residue

DESCRIPTION

The **pdb to geom** module reads the description of a molecule from a file in the Brookhaven Protein Data Bank (PDB) data format. Typically, such files have a *.pdb* filename suffix. The output is an AVS *geometry* description of the molecule.

PARAMETERS

Data File A file browser allows you to specify the name of the *.pdb* file containing the molecule description.

Mode The type of geometry produced:

ball and stick
Small spheres represent the atoms, and white lines represent the bonds.

ball
Large spheres represent the atoms.

stick
White lines represent the bonds.

colored stick
Colored lines represent the atoms and their bonds.

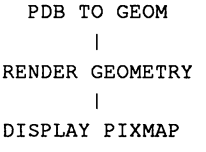
colored residue
Colored lines represent the atoms and their bonds. The color of the lines represents the type of amino acid that the molecule is in.

OUTPUTS

Molecule (geometry)
An AVS *geometry* description of the molecule.

EXAMPLE

This example shows a simple application of **pdb to geom**:



RELATED MODULES

render geometry

LIMITATIONS

If you read in the same *.pdb* file name twice, you will get only one instance of the geometry, not two.

Since the *.pdb* file does not contain any bond information, bonding is determined by the distances between atoms.

SEE ALSO

The example script PDB TO GEOM demonstrates the **pdb to geom** module.

NAME

pixmap to image – transform AVS pixmap to AVS image

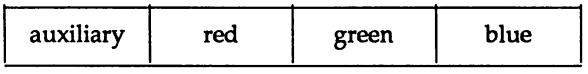
SUMMARY

Name	pixmap to image
Type	mapper
Inputs	pixmap
Outputs	image (field 2D 4-vector byte)
Parameters	none

DESCRIPTION

The **pixmap to image** module takes an AVS pixmap as input and outputs an AVS image ("field 2D 4-vector byte"). The pixmap is an X Window System resource used to store image data in the X server. This reduces the amount of data AVS must pass between modules: a pixmap id and window id.

The 4-vector byte representation for the image consists of pixels that look like this:



this field interpreted as pixel's opacity value

these three fields make up pixel's color value

The high-order byte field (auxiliary) is generally unused, but sometimes contains alpha (opacity) information on a per-pixel basis.

INPUTS

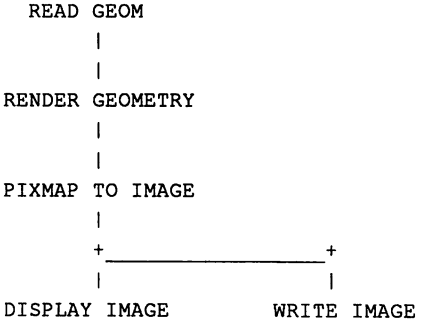
pixmap (required; pixmap)
: The input is any AVS *pixmap*.

OUTPUTS

image (field 2D 4-vector byte)
: The output data is a 2D block of pixels. The data set at each point of the 2D field will be a 4-vector of bytes in the AVS *image* format.

EXAMPLE

This module is useful for converting the output of data output modules (e.g. **render geometry**) into images for writing to a file.



RELATED MODULES

Image processing:

contrast, threshold, histogram stretch, clamp, interpolate, colorizer, generate colormap

Renderers which generate pixmaps:

render geometry

Display an image:

display image

Pixmap manipulation and display:

transform pixmap, display pixmap

LIMITATIONS

The "Refine" function in a **transform pixmap** module that is upstream of a **pixmap to image** module does not work.

SEE ALSO

The example script BACKGROUND demonstrates the **pixmap to image** module.

NAME

print field – create an ASCII printable/readable version of an AVS field

SUMMARY

Name print field

Type data output

Inputs field *any-dimension n-vector any-data any-coordinates*

Outputs none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Display Header	switch	on		
Display Data	switch	off		
Max Elements	integer	-1	-1	1000
Output File	typein	*/tmp/pfield...*		
Min X	typein	0	0	1000
Max X	typein	-1	-1	1000
Min Y	typein	0	0	1000
Max Y	typein	-1	-1	1000
Min Z	typein	0	0	1000
Max Z	typein	-1	-1	1000
Min W	typein	0	0	1000
Max W	typein	-1	-1	1000

DESCRIPTION

The **print field** module creates a human-readable version of the contents of an AVS field. The information takes two forms: it is displayed in an **Output Browser** widget on the AVS control panel, and it is written to a online file. **print field** is useful whenever you need to inspect the actual contents of an AVS field. For example, if you are using the **import to field** module, **print field** can show whether you importing the data correctly.

By default, **print field** displays just the header information, showing the number of dimensions (Ndim), the size of each dimension (Dims), the number of coordinate dimensions (Nspace), the vector length (Veclen), the data type (real, integer, byte, etc.), the size of each data element in bytes (Size), the coordinate type (uniform, rectilinear, or curvilinear), and the minimum and maximum data extent. If the information is present, it will also display any labels and minimum or maximum data values associated with the field.

If the **Display Data** switch is toggled, **print field** also displays the data contents of the field and its coordinate values. An integer dial regulates how many values (to a maximum of 1000) are shown. A scrollbar lets you scroll vertically through the data elements outside the the normal scope of the display widget.

By default, **print field** starts at X, Y, Z values 0, 0, 0 and starts counting up with the Z value turning over most quickly. However, you can display any rectangular section of the data by setting the minimum and maximum coordinate values for X, Y, Z, and (if present) W.

Whenever you change any of the parameter settings, **print field** rewrites the **Output File**, as well as changing the display in the **Output Browser** widget.

The window in which **print field** displays its output can be resized, like any other widget, using the AVS Layout Editor. For a detailed description of how to do this, see the section titled "Layout Editor," in the chapter The Network Editor Subsystem of the *AVS User's Guide*.

INPUT

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input AVS field can be 1, 2, 3, or 4 dimensional.

PARAMETERS

Display Header
A toggle switch that controls whether **print field** displays and writes the field's header information (dimensionality, type, etc.) It is on by default.

Display Data
A toggle switch that controls whether **print field** displays and writes the field's data and coordinate information. It is off by default.

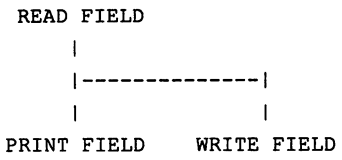
Max Elements
An integer dial that controls how many elements of the field are displayed and written to the output file. The default is 1, which displays and writes one value. The maximum for any one display and file write is 1000 elements. You can use the scrollbar at the side of the **Output Browser** widget to see values vertically outside the window. You can look at the file output version of the field if too much data is clipped horizontally by the **Output Browser** widget. or resize the widget using the Layout Editor.

Output File
An ASCII typein for specifying the output file. By default, **print field** writes to a file in the */tmp* directory called *pfield_nnnn*, where nnnn is the process id of the **print field** module. The **Output File** is rewritten whenever any of the other parameters change.

Min X
Max X
Min Y
Max Y
Min Z
Max Z
Min W
Max W
Integer typeins that define a rectangular section of the field to display and write to the **Output File**. Whatever values are entered here, **Max Elements** regulates the total number of elements that will be output. **print field** does not check to see that the values entered are within the actual dimensions of the field, or that the number of dimensions match, but it will not exceed the actual dimensions of the field. 1, 2, 3 and 4 dimensional fields are supported. By default, minimum values are set to 0, while the maximum values are -1, causing as much of the field in that dimension to be displayed as **Max Elements** allows.

EXAMPLE 1

The following network converts some data into an AVS field, displays the contents of the new field, and gives the person the option of writing the new AVS field permanently to disk. For details on converting data into AVS field format, see the man page for **read field**.



RELATED MODULES

compare field

LIMITATIONS

print field writes to /tmp by default. This can cause problems if: (1) there is no /tmp mounted on your system, (2) the /tmp directory does not have very much room in it or has inaccessible protections, or (3) the module is being run remotely.

SEE ALSO

The example scripts PRINT FIELD, and FIELD MATH demonstrate the **print field** module.

NAME

probe – interactively show numeric data values in a geometry rendered field

SUMMARY

Name	probe
Type	mapper
Inputs	field 3D *n-vector any-data any-coordinates* colormap *(optional)* field irregular *(optional, from samplers module)* upstream transform *(optional, invisible, autoconnect)* upstream geometry *(optional, invisible, autoconnect)*
Outputs	geometry upstream transform *(optional, invisible, autoconnect)*

Parameters

Name	*Type*	*Default*
Sampling Style	choice	Point
Probe Type	choice	Cursor
Pick Geometry	boolean	off

DESCRIPTION

Scientific visualization converts numbers into colored pictures. However, after you have a picture, you often want to be able to get back and examine the numbers that are producing it.

The **probe** module displays the numeric data values in a field at a location in space. It works for fields that have been rendered as an AVS geometry. It works for uniform, rectilinear, and irregular coordinates, and for any data type. It works for both scalar and vector fields.

probe works by creating a cursor-like object titled "probe" that coexists in the Geometry Viewer window with the rendered version of the field data. Its initial position is 0,0,0; the origin. You deal with this probe object just like any other object in the Geometry Viewer. As you move the "probe" object through space, it reports its location and the data value at that location.

There are two major ways to use the **probe**:

- With the **Pick Geometry** option *off*, the "probe" object in the Geometry Viewer acts like any other object. To find a data value at a particular location in space, you make "probe" the current object and move it to that location. The movement can be direct manipulation using the usual Geometry Viewer mouse-button commands (e.g., right button moves object left and right); or, if that is too awkward and imprecise, you can use the Geometry Viewer's "Transformation Selection" panel and have the "probe" object jump to any absolute or relative point in space. As the **probe** travels, it continuously reports its location and the data value beneath it.
- With the **Pick Geometry** option *on*, data sampling is more a "point the mouse cursor and click" technique. Select "probe" as the current object in the Geometry Viewer, point at the object surface you want to sample with the *mouse* cursor, then press the left mouse button. The probe object snaps *to the surface* beneath the cursor and reports the data value.

  The Geometry Viewer tells the **probe** module what vertex the mouse cursor was over when the button was pressed, and **probe** reports the original data value at that vertex.

  When reporting data values for vector fields, **probe** lists the values of all the vector elements. If the **probe** is being colored with the data values., the color shown is SQRT(vec0**2 + vec1**2 + vec2**2 ...), in other words, the magnitude of the data vector, mapped to the range of the current colormap.

INPUTS

Data Field (required; field 3D *n-vector any-data any-coordinates*)
The input field is 3D, scalar or vector, uniform or rectilinear or irregular, of any data type.

Colormap (optional)
If an AVS colormap is supplied to the center input port, the color of the probe object in the Geometry Viewer will change according to the data value it is pointing at. I.e., if it is pointing at a "low" value with the default colormap from **generate colormap**, the probe object will be blue; it it is pointing at a "high" value, it will be red.

Data Field (optional; field irregular)
This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **probe** will take these locations and display the data values associated with them.

Upstream Transform (optional, invisible, autoconnect)
When the **probe** and **render geometry** modules coexist in a network, they communicate through a normally-invisible data port. "Probe" shows up as an object in the Geometry Viewer. When you select the probe object and move it, **render geometry** informs the **probe** module what the probe's new location is, and the **probe** module recalculates the location and data it is displaying accordingly. This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **probe** module's "probe" object.

Upstream Geometry (optional, invisible, autoconnect)
Used by the **Pick Geometry**'s "point cursor and click" technique, this normally invisible port is what the **render geometry** module uses to inform **probe** of the geometry vertex selected so it can display the data value for it. The module connection occurs automatically.

PARAMETERS

Sampling Style
A pair of radio buttons that specify what sampling technique to use to report the data values.

point means that, if the probe/cursor is pointing *between* actual nodes on the data lattice, it will display the *real* data value for the *nearest* node. This is the faster sampling technique.

Trilinear means that, if the probe/cursor is pointing between actual nodes on the data lattice, it will *calculate* a data value that is a trilinear interpolation of the *eight* nearest real node data values.

Probe Type
A set of radio buttons that control what the "probe" object looks like in the Geometry Viewer.

Cursor creates a probe that looks like a miniature XYZ axis.

Crosshair creates a probe that looks like half of a miniature XYZ axis. The crosshair stays aligned with the axis, and its endpoints lie in the XY, YZ, and XZ planes.

Probe creates a probe that looks like an electronic probe or a dissecting needle.

Pick Geometry
A boolean switch that controls whether one moves the "probe" object like any Geometry Viewer object by selecting it as the current object and translating it with mouse button commands or the Transformation Selections panel (the default, off); or whether one selects data by pointing to an object's verticies with the mouse cursor and pressing the left mouse

button.

OUTPUTS

Geometry (geometry)

The output geometry has two parts:

The rendering of the "probe" object, and;

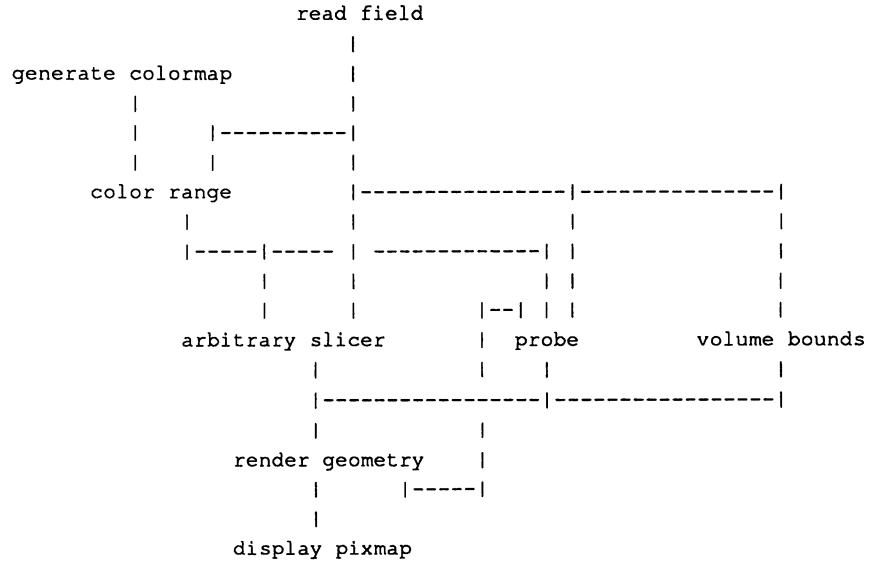
The rendering of the "Text for Probe" that lists the data value and coordinate position.

Upstream Transform (optional, invisible, autoconnect)

If **probe** is connected to the **samplers** module, it uses this port to relay movement information from **render geometry** back up the network to **samplers**.

EXAMPLE 1

The following network inputs a curvilinear scalar field, scales the color values to the actual data range, displays it through **arbitrary slicer**, with a colorized "probe" object, surrounded by volume bounds:



RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
read field
read plot3d

Modules that could provide the **colormap** input:

generate colormap
color range

Modules that could provide the Sample field input:

samplers

Modules that can process **probe** output:

render geometry
render manager

SEE ALSO

The example script PROBE demonstrates the **probe** module.

NAME

read field – read AVS field from a disk file, or import data files into AVS field format

SUMMARY

Name	read field	
Type	data	
Inputs	none	
Outputs	field *same-dimension same-vector same-data same-coordinates*	
Parameters	*Name*	*Type*
	Read File	browser

DESCRIPTION

The **read field** module has two input modes:

- In its first input mode, it reads an AVS *field* data structure from a disk file into a network. The format of an AVS *field file* is discussed below in section *AVS Field File Format*.
- In its second input mode, it converts data stored in ASCII, Fortran unformatted, or pure binary data files into AVS field format. **read field** can thus be used to import *some* datasets into the AVS system.

The two input modes — "native field input" and "data-parsing input" — are described separately in the sections below.

PARAMETERS

Read File A file browser window to specify the name of the file to be read.

NATIVE FIELD INPUT

read field can read files in the native AVS field file format into an AVS network. An AVS field file (suffix *.fld*) has the following components:

- An ASCII header that describes the field
- Two separator characters that divide the ASCII header from the data and coordinate information
- A binary area containing the data and coordinate information

The **write field** module creates files in this format.

ASCII HEADER

The ASCII header contains a series of text lines, each of which is either a comment or a *TOKEN=VALUE* pair. For example, the following header defines a field of type "field 2D 4-vector byte", which is the AVS image format:

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

The first two lines are comments, indicated by the # character. Note that the first line of the header *must* begin as follows:

```
# AVS
```

In this example, comments also occur at the end of the 3rd and 6th lines. Any characters following (and including) # in a header line are ignored. Comments are not required.

SEPARATOR CHARACTERS

The ASCII header must be followed by two formfeed characters (i.e. **Ctrl-L**, octal 14, decimal 12, hex 0C), in order to separate it from the binary area. This scheme allows

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

BINARY AREA

The size (in bytes) of the binary area depends on the field type:

- For **uniform** fields, the binary area contains data values followed by the coordinate values.

  Coordinate information is limited to minimum and maximum extent fullword values for each physical dimension (n-space) of the data. The minimum and maximum extent values in the coordinate binary area are copies of the **min_ext** and **max_ext** values in the field data structure, *except* when the field has been cropped, downsized, or interpolated. Then 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 subsetted data. Mapper modules can use this additional extent information to properly locate their geometric representation of the subsetted data in world coordinate space. The extents in the coordinate binary area are stored in this order: minimum x, maximum x, minimum y, maximum y, minimum z...etc.

  Thus, the size of the binary area is the product of the following numbers:

value of **dim1**	(product of sizes of computational dimensions
value of **dim2**	yields total number of field elements)
...	
value of **dim***x*	
value of **veclen**	*(number of data values per field element)*
size of **data**	*(byte size of primitive data type)*

  Plus:

8 * value of **nspace**	*(2 coordinates per dimension, 4 bytes per coordinate)*

  In the stream of data values:

  - All the data values for a field element are stored together.
  - The first array index varies most quickly (*FORTRAN-style*).

- For **rectilinear** fields, the binary area contains both data values and coordinates for each scalar data value or vector of data values. The data values occupy the same amount of space as for a **uniform** field. Each coordinate is a single-precision floating-point number (4 bytes), and there is one coordinate for each array index in each dimension of computational space. Thus, the size of the coordinates area is:

  (*dim1* + *dim2* ... + *dimx*) * 4

  All of the X-coordinates are stored together, at the beginning of the coordinates area. Following these are all the Y-coordinates, and so on.

- For **irregular** fields, the data area contains both data values and coordinates. The data values occupy the same amount of space as for a **uniform** field. Each coordinate is a single-precision floating-point number (4 bytes), and each field element is mapped to a point in *nspace*-dimensional physical space. Thus, the size of the coordinates area is:

  (*dim1* * *dim2* ... * *dimx*) * *nspace* * 4

  As with **rectilinear field**, all of the X-coordinates are stored together, at the beginning of the coordinates are. Following these are all the Y-coordinates, and so on.

EXAMPLE 1

The following ASCII header describes a volume (3D uniform field) with a single byte of data for each field element. This format might be used to represent CAT scan data.

```
# AVS field file
ndim=3          # number of dimensions in the field
```

```
dim1=64          # dimension of axis 1
dim2=64          # dimension of axis 2
dim3=64          # dimension of axis 3
nspace=3         # number of physical coordinates per point
veclen=1         # number of components at each point
data=byte        # data type (byte, integer, float, double)
field=uniform    # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

(64 * 64 * 64) * 1 * 1 = 262,144 bytes

The coordinates area occupies (2 * 4) * 3 bytes. The total binary area occupies 262,168 bytes.

EXAMPLE 2

The following ASCII header describes a volume (3D uniform field) whose data for each field element is a 3D vector of single-precision values. This format might be used to represent the wind velocity at each point in space.

```
# AVS field file
ndim=3           # number of dimensions in the field
dim1=27          # dimension of axis 1
dim2=25          # dimension of axis 2
dim3=32          # dimension of axis 3
nspace=3         # number of physical coordinates per point
veclen=3         # number of components at each point
data=float       # data type (byte, integer, float, double)
field=uniform    # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

(27 * 25 * 32) * 4 * 3 = 259,200 bytes

The coordinates area occupies (2 * 4) * 3 bytes. The total binary area occupies 259,224 bytes.

EXAMPLE 3

The following ASCII header describes an irregular volume (3D irregular field) with one single-precision value for each field element. The binary area includes an (X,Y,Z) coordinate triple for each field element, indicating the corresponding point in physical space. This format might be used to represent fluid flow data.

```
# AVS field file
ndim=3           # number of dimensions in the field
dim1=40          # dimension of axis 1
dim2=32          # dimension of axis 2
dim3=32          # dimension of axis 3
nspace=3         # number of physical coordinates per point
veclen=1         # number of components at each point
data=float       # data type (byte, integer, float, double)
field=irregular  # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

(40 * 32 * 32) * 4 * 1 = 163,840 bytes

The coordinates area occupies this amount of space:

(40 * 32 * 32) * 4 * 3 = 491,520 bytes

DATA-PARSING INPUT MODE

In its second input mode, **read field** can convert a certain class of data stored in ASCII, Fortran unformatted, or pure binary data files into AVS field format. To import data into AVS, you must create an ASCII description file that defines the structure of the AVS field to make. The first part of this description file is identical in format and meaning to the ASCII header file described above.

The second part of this file contains commands that specify which files contain the data or coordinate information, its data type (ASCII, binary, or Fortran unformatted), and simple parsing instructions. **read field** can read a file that is parseable by this general scheme:

skip *n* lines or bytes
move over an **offset** of *m* columns on this line (ASCII only)
read the value
do until # *of values needed*
{
take *p* **stride**(s) to the next value
}

The ASCII description file, data, and coordinate information for rectilinear and irregular data can all be read from different files. If the resulting AVS field contains a vector of data values at each point, each vector element can also be read from a separate file.

The ASCII description file must have a *.fld* file suffix or the **read field** file browser will not display the file.

read field data parsing capablility is meant to be used only once, in order to convert data to AVS field format. The parsing activity makes **read field** run more slowly than when it reads a file that is already in AVS field format. Once you have read your data using **read field**'s data-parsing mode, you should use the **write field** module to store it permanently on disk in AVS field file format.

Suggestion: While experimenting with **read field**'s ASCII description file, connect its output port to the **print field** module's input port and use **print field**. This allows you to examine the results online, to see whether the data is being interpreted correctly.

read field chronicles its progress in a status display below the file browser widget as it works through the input files to assemble the AVS field.

ASCII DESCRIPTION FILE

As the example below shows, the ASCII description file contains a series of text lines that define the AVS field to construct. Each line is either:

- A comment
- A required line in the form *token=value*
- An optional line in the form *token=value*
- A **variable** or **coord** parsing specification

The following ASCII description file imports three dimensional curvilinear data with a vector of values at each point into an AVS field of type "field 3D 3-vector irregular float". This type of data often occurs in computational fluid dynamics applications. The data and coordinate information are in separate files, both of which were written as straight binary data. Both files happen to have a serial organization. In the data file, all of vector element 1's values appear, then all of vector element 2's, then all of vector element 3's values. In the X, Y, Z coordinate file, all the X coordinate values appear, then all the Y's, then all the Z's.

Each line's meaning is explained in detail below.

```
# AVS field file      the string "# AVS" must be the first
#                     five characters in the file
#                     when a '#' character appears in a line,
#                     the rest of the line is a comment
#
ndim=3                      # REQUIRED--the number of dimensions in the field
dim1=40                     # REQUIRED--dimension of axis 1
dim2=32                     # REQUIRED--dimension of axis 2
dim3=32                     # REQUIRED--dimension of axis 3
```

```
nspace=3                    # REQUIRED--number of coordinates per point
veclen=3                    # REQUIRED--number of components at each point
data=float                  # REQUIRED--data type (byte,integer,float,double)
field=irregular             # REQUIRED--field type (uniform, rectilinear,irregular)
min_ext=-1.0 -1.0 -1.0      # OPTIONAL--coordinate space extent
max_ext=1.0 1.0 1.0         # OPTIONAL--coordinate space extent
label=x-velocity            # OPTIONAL--component label for variable 1
label=y-velocity            # OPTIONAL--component label for variable 2
label=z-velocity            # OPTIONAL--component label for variable 3
unit=miles-per-second       # OPTIONAL--describes unit of measure for variable 1
unit=miles-per-second       # OPTIONAL--describes unit of measure for variable 2
unit=miles-per-second       # OPTIONAL--describes unit of measure for variable 3
min_val=-2.18 -0.32 -3.73   # OPTIONAL--minimum data values per component
max_val=5.79 3.54 1.50      # OPTIONAL--maximum data values per component
#
# For each coordinate X, Y, and Z, where to find it and how to read it
#
coord 1 file=/usr/userid/data/wing.bin filetype=binary skip=12
coord 2 file=/usr/userid/data/wing.bin filetype=binary skip=163852
coord 3 file=/usr/userid/data/wing.bin filetype=binary skip=327692
#
# For each value in the vector, where to find it and how to read it
#
variable 1 file=/usr/userid/data/wdata.bin filetype=binary skip=28
variable 2 file=/usr/userid/data/wdata.bin filetype=binary skip=163868
variable 3 file=/usr/userid/data/wdata.bin filetype=binary skip=327708
```

Any characters following (and including) # in a header line are ignored.

NOTE: The first five characters in the ASCII description file *must* be "# AVS" or **read field** will not recognize the file as valid.

The example above shows all of the required *TOKEN=VALUE* token names: an ASCII description file that is missing one or more of these lines causes **read field** to generate an error. Required *TOKEN=VALUE* pairs are stored in the AVS field that **read field** produces as output.

Optional *TOKEN=VALUE* pairs are stored in the output AVS field as well, if they are provided. **min_ext** and **max_ext** are stored in the output AVS field even if they are not specified, as **read field** calculates them if they are not provided.

The **variable** and **coord** lines are not stored in the output AVS field. They are only instructions to **read field**.

With the exception of filenames, ASCII description file specifications are *not* case-sensitive. The following features are new in AVS 3.0:

- You can surround the = character with any amount of white space (including none at all). For example, "dim2 = 32", "DIM 2 =32", and "Dim2=32" are all equivalent.
- Value strings *do not* have to be padded out to 11 characters.

ndim = *value* (required)
: The number of computational dimensions in the field. For an image, **ndim** = 2. For a volume, **ndim** = 3.

dim1 = *value* (required)
dim2 = *value* (required, depending on total number of dimensions)
dim3 = *value* (required, depending on total number of dimensions)
...
: The dimension size of each axis (the array bound for each dimension of the computational array). The number of **dim***x* entries must match the value of **ndim**. For instance, if you specify a 3D field (**ndim**=3), you must specify the length of the X dimension (**dim1**), the length of the Y dimension (**dim2**), and the length of the Z dimension (**dim3**).

Note that counting is 1-based, not 0-based.

nspace = *value* (required)
The dimensionality of the physical space that corresponds to the computational space (number of physical coordinates per field element).

In many cases, the values of **nspace** and **ndim** are the same — the physical and computational spaces have the same dimensionality. But you might embed a 2D computational field in 3D physical space to define a manifold; or you might embed a 1D computational field in 3D physical space to define an arbitrary set of points (a "scatter").

veclen = *value* (required)
The number of data values for each field element. All the data values must be of the same primitive type (e.g. **integer**), so that the collection of values is conceptually a **veclen**-dimensional vector. If **veclen**=1, the single data value is, effectively, a scalar. Thus, the term *scalar field* is often used to describe such a field.

data = byte (one of the four options is required)
data = integer
data = float
data = double
The primitive data type of all the data values.

field = uniform (one of the three options is required)
field = rectilinear
field = irregular
The field type. A **uniform** field has no computational-to-physical space mapping. The field implicitly takes its mapping from the organization of the computational array of field elements.

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

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

min_ext = *x-value [y-value] [z-value]...* (optional)
max_ext = *x-value [y-value] [z-value]...* (optional)
The minimum and maximum coordinate value that any member data point occupies in space, for each axis in the data. If you do not supply this value, **read field** calculates it and stores it in the output AVS field data structure. This value can be used by modules downstream to, for example, size the **volume bounds** drawn around the data in the Geometry Viewer or put minimum and maximum values on coordinate parameter manipulator dials (**probe**). Values can be separated by blanks and/or commas.

If you do not know the extents, don't guess — let **read field** calculate them. Most downstream modules use whatever values are supplied, without checking their validity. If the wrong numbers are specified, incorrect results will be computed.

label = *string1 [string2] [string3]...* (optional)
Allows you to title the individual elements in a vector of values. These labels are stored in the output AVS field data structure. Subsequent modules that work on the individual vector elements (for example, **extract scalar**) will label their parameter widgets with the strings provided here instead of the default "Channel 0, Channel 1...", etc. You can either use one **label** line as shown here, or separate label lines as shown in the example above. In either case, the labels are applied to the

elements of the vector in the order encountered. You can also label single scalar values, though downstream modules may ignore such a label. Any alphanumeric string is acceptable. Strings can be separated by blanks and/or commas.

unit = *string1 [string2] [string3]...* (optional)
Allows you to specify a string that describes the unit of measurement for each vector element. You can either use one *unit* line as shown here, or separate unit lines as shown in the example above. In either case, the unit specifications are applied to the elements of the vector in the order encountered. You can also specify the unit for a single scalar value, though downstream modules may ignore it. Any alphanumeric string is acceptable. Strings can be separated by blanks and/or commas.

min_val = *value [value] [value]...* (optional)
max_val = *value [value] [value]...* (optional)
For each data element in a scalar or vector field, allows you to specify the minimum and maximum data values. These values are stored in the output AVS field data structure. This is used by subsequent modules that need to normalize the data. Values can be separated by blanks and/or commas.

read field does not calculate these values if you do not supply them (unlike **min_ext** and **max_ext**). If you do not know these values, don't guess — just leave these optional lines out. In this case, the **write field** module will compute these values when it creates an AVS field file. Most downstream modules use whatever values are supplied, without checking their validity. If the wrong numbers are specified, incorrect results will be computed.

variable *n* **file=***filespec* **filetype=***type* **skip=***n* **offset=***m* **stride=***p*
coord *n* **file=***filespec* **filetype=***type* **skip=***n* **offset=***m* **stride=***p*
variable specifies where to find *data* information, its type, and how to read it.

coord specifies where to find *coordinate* information, its type, and how to read it. It is used when the data is **rectilinear** or **irregular.**

The individual parameters are interpreted as follows:

n An integer value that specifies which element of a data vector or which coordinate (1 for x, 2 for y, 3 for z, etc.) the subsequent read instructions apply to. **n** does not default to 1 and must be specified.

file = *filespec*
The name of the file containing the data or coordinates. If no directory information is given, only a filename, the directory defaults to */usr/avs/data*, or the value given by the DataDirectory environment variable, or to the directory specified with the **–data** option on the **avs** command line, with increasing precedence. It does not default to the file browser widget's current directory.

filetype = ascii
filetype = unformatted
filetype = binary
ascii means that the data or coordinate information is in an ASCII file. In ASCII files, float data can be specified in either real (0.1) or scientific notation (1.00000e-01) format interchangeably.

unformatted means that the data or coordinate information is in a file that was written as Fortran unformatted data. (Fortran unformatted data is binary data with additional

words written at the beginning and end of each data block stating the number of bytes (ST1000/2000 machines) or words (ST1500/3000 machines) in the data block.). When you are figuring out the **skip** and **stride** values below, ignore size words in your calculation.

binary means that the file is written in straight binary format. such as that produced by Unix output routines, write and fwrite.

Note the warning on binary compatibility among different hardware platforms in the introductory chapter of the *AVS Developer's Guide*.

In each case, **read field** will use the data type specified in the earlier **data={byte,float,integer,double}** statement when it interprets the file.

skip = *n* For **ascii** files, **skip** specifies the number of *lines* to skip over before starting to read the data. Lines are demarked by newline characters.

For **binary** or **unformatted** files, **skip** specifies the number of *bytes* to skip over before starting to read the data.

There are two motivations for **skip**. First, data files often include header information irrelevant to the AVS field data type. Second, if the file contains, for example, all X data values, then all Y data values, **skip** provides a way to space across the irrelevant data to the correct starting point.

skip can only be used once at the start of the file. There is no way to **skip**, read, **stride**, then **skip** again.

You must simply know what value to use for **skip** based on your knowledge of the software that produced the original data file, the number of data elements, and the type (byte, float, double, integer, etc.)

skip defaults to 0.

offset = *m* **offset** is only relevant to ASCII files; it is ignored for binary or unformatted files. **offset** specifies the number of columns to space over before starting to read the first datum. (The **stride** specification determines how subsequent data are read.) Hence, to read the fourth column of numbers in an ASCII file, use **offset=3**.

In ASCII files, columns must be separated by one or more blank characters. Commas, semicolons, TAB characters, etc., are *not* recognized as delimiters. If necessary, edit ASCII files to meet this restriction.

offset defaults to 0 (the first column, no columns spaced over).

stride = *p* **stride** assumes you are "standing on" the data value just read. **stride** specifies how many "strides" must be taken to get to the next data value. In ASCII files, **stride** means stride forward *p* delimited items. In binary and unformatted files, **stride** means stride forward $p \times$ *the size of the data type* (byte, float, double, integer). In a file where the data or coordinate values are sequential, one after the other, the **stride** would be 1. Note that this presumes homogeneous data in binary and unformatted files — double-precision values could not be intermixed with single precision values.

stride defaults to 1.

The stride value will be repeatedly used until the number of data items indicated by the product of the dimensions (e.g. dim1 × dim2 × dim3) have been read.

Here are some **skip, offset,** and **stride** examples for ASCII data. "A's" are vector component 1; "B's" are vector component 2. There are more examples at the end of this manual page.

ASCII file organization 1:

```
X     Y     Z     A     B
1     1     1     A1    B1
2     2     2     A2    B2
3     3     3     A3    B3
4     4     4     A4    B4
5     5     5     A5    B5
```

to read A: skip=1, offset=3, stride=5
to read B: skip=1, offset=4, stride=5

ASCII file organization 2:

```
A1    A2    A3    A4    A5
A6    A7    A8    A9    A10
A11   A12   A13   A14   A15
B1    B2    B3    B4    B5
B6    B7    B8    B9    B10
B11   B12   B13   B14   B15
```

to read A: skip=0, offset=0, stride=1
to read B: skip=3, offset=0, stride=1

ASCII file organization 3:

```
A1      B1      A2      B2      A3      B3
A4      B4      A5      B5      A6      B6
A7      B7      A8      B8      A9      B9
A10     B10     A11     B11     A12     B12
```

to read A: skip=0, offset=0, stride=2
to read B: skip=0, offset=1, stride=2

ASCII file organization 4:

```
TEMP1=A1 TEMP2=A2 TEMP3=A3 TEMP4=A4
TEMP5=A5 TEMP6=A6 TEMP7=A7 TEMP8=A8
PRESS=B1 PRESS=B2 PRESS=B3 PRESS=B4
PRESS=B5 PRESS=B6 PRESS=B7 PRESS=B8
```

read field cannot read this file until the data labels and equal signs are edited out.

EXAMPLE 4

You have some 3-dimensional, curvilinear data that projects the amount and location of wood that will be eaten after five years by a colony of termites that has entered a 14th century Scandanavian grain silo structure at a particular spot in its base. The data is in one ASCII file, *decay.dat*, as a long sequential, numbered list of 1250 consumed-wood values that looks like this:

```
1,1002.707;
2,1443.971;
3,1307.069;
4,1240.354;
5,1778.715;
   ...
```

The coordinates that correspond to the data values are in a separate ASCII file,

where.coord, that looks like this:

```
LOC,1,0,0.2500000,0.0000000e+00,1.105255,0.0000000e+00;
LOC,2,0,0.2500000,0.0000000e+00,1.000000,0.0000000e+00;
LOC,3,0,0.5000000,0.0000000e+00,1.552552,0.0000000e+00;
LOC,4,0,0.5000000,0.0000000e+00,1.442042,0.0000000e+00;
LOC,5,0,0.5000000,0.0000000e+00,1.331531,0.0000000e+00;
    ...
```

In the data file, the second column represents the data. In the coordinate file, the fourth through sixth columns are the x, y, and z coordinates, respectively.

First, to read this data, you must use a text editor to globally edit out the commas and semi-colons, changing them to spaces. The files now look like:

```
1 1002.707
2 1443.971
  ...

LOC 1 0 0.2500000 0.0000000e+00 1.105255 0.0000000e+00
LOC 2 0 0.2500000 0.0000000e+00 1.000000 0.0000000e+00
  ...
```

The following ASCII description file, *decay.fld*, would import the data into AVS field format:

```
# AVS Field File
#
# Termite Decay after Five Years
#
  ndim=3            # number of dimensions in the field
  dim1=25           # dimension of axis 1
  dim2 =10          # dimension of axis 2
  dim3 =5           # dimension of axis 3
  nspace=3          # number of physical coordinates
  veclen=1          # number of elements at each point
  data=float        # data type (byte, integer, float, double)
  field=irregular   # field type (uniform, rectilinear, irregular)
coord 1 file = /name/Termites/where.coord filetype=ascii offset = 3 stride = 7
coord 2 file = /name/Termites/where.coord filetype=ascii offset = 4 stride = 7
coord 3 file = /name/Termites/where.coord filetype=ascii offset = 5 stride = 7
variable 1 file = /name/Termites/decay.dat filetype=ascii offset =1 stride = 2
```

EXAMPLE 5

The following ASCII description file specifies how to convert the volume data in the file */usr/avs/data/volume/hydrogen.dat* into an AVS field. *hydrogen.dat* is a series of binary byte values that represent the probability of finding an electron at various locations around a hydrogen nucleus. The first three bytes in the file give the X, Y, and Z dimensions of the data—however, this information is not part of the actual data and must be skipped over. You could examine these three bytes and determine what to use for the dimensions in the ASCII description file. Thereafter, it is just a matter of reading successive bytes. **offset** is not used because this is not an ASCII file. **stride** is allowed to default to 1.

```
# AVS field file
ndim=3              # number of dimensions in the field
dim1=64             # dimension of axis 1
dim2=64             # dimension of axis 2
dim3=64             # dimension of axis 3
nspace=3            # number of physical coordinates per point
veclen=1            # number of components at each point
data=byte           # data type (byte, integer, float, double)
field=uniform       # field type (uniform, rectilinear, irregular)
variable 1 file=/usr/avs/data/volume/hydrogen.dat filetype=binary skip=3
```

EXAMPLE 6

This ASCII description file specifies how to use **read field** to convert the image data in */usr/avs/data/image/mandrill.x* into an AVS field. The first two words in *mandrill.x* are 32-bit integers that specify the horizontal and vertical dimensions of the image. This information must be skipped over — you must supply it in the ASCII description file. Thereafter, *mandrill.x* is a succssion of 32-bit straight binary words, one word per pixel. However, in AVS, each of these words is considered to be a vector of 4 bytes. The first byte is the "alpha" (or "transparency") value for the pixel, and the second through fourth bytes are the red, green, and blue values for each pixel. Thus, this whole file is treated as a series of binary bytes.

```
# AVS field file
#
ndim = 2                          # number of dimensions in the field
nspace=2                          # number of physical coordinates
dim1=500                          # dimension of axis 1
dim2=480                          # dimension of axis 2
veclen=4                          # number of components at each point
data=byte                         # data type (byte, integer, float, double)
field=uniform                     # field type (uniform, rectilinear, irregular)
label = alpha, red, green, blue      # labels the vector elements
variable 1 file=/usr/avs/data/image/mandrill.x filetype=binary skip=8 stride=4
variable 2 file=/usr/avs/data/image/mandrill.x filetype=binary skip=9 stride=4
variable 3 file=/usr/avs/data/image/mandrill.x filetype=binary skip=10 stride=4
variable 4 file=/usr/avs/data/image/mandrill.x filetype=binary skip=11 stride=4
```

RELATED MODULES

The **write field** module will take the AVS field produced by **read field** and write it to disk as a permanent AVS field file. The **read field** module can then read the data much more quickly whenever you need to use it.

The **print field** module displays the ASCII header and contents of an AVS field interactively on the screen. Connect it to **read field**'s output port while experimenting with ASCII description files to verify that the data is being read correctly.

ERROR CHECKING

read field performs a significant amount of error checking. If an error is detected while reading the field, an error dialog box appears on the screen, indicating the line in which the error occurred (if it was in the ASCII header), along with the type of error.

SEE ALSO

The example scripts PRINT FIELD, CONTRAST, FIELD MATH, as well as others demonstrate the **read field** module.

NAME

read geom – reads a data file containing an AVS 'geometry'

SUMMARY

Name	read geom
Type	data
Inputs	*none*
Outputs	geometry
Parameters	*Name* *Type* Read Geometry browser

DESCRIPTION

The **read geom** module reads a file containing an AVS *geometry* and outputs the geometry to one or more modules connected to its output port. The resulting object will be named after the file from which it was read. Since AVS replaces geometries based on the object name, if you read in the same filename twice, you will only get one representation of the object.

Since the Geometry Viewer subsystem (also accessible as the **render geometry** module) has a built-in **Read Object** function, you rarely need to use this module. It is most useful when used in conjunction with a filter module that processes geometric data (e.g. **shrink**).

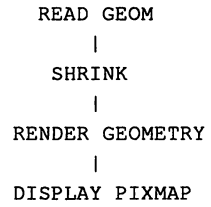
PARAMETERS

filename A file browser allows you to specify the name of the file that contains an AVS *geometry*.

OUTPUTS

geometry The output is the *geometry* that was read from the specified file.

EXAMPLE



RELATED MODULES

shrink, offset, render geometry, wireframe, tube

LIMITATIONS

This module reads GEOM-file files only. It cannot read *.obj* script files or *.scene* scene files that can be created with the Geometry Viewer Script Language (see Appendix B).

The object is always named after the file from which it is read. This makes it awkward to create animation loops, for which you might want to direct multiple files to the same name or to read in multiple instances of the same object.

SEE ALSO

The example scripts CONTRAST, OFFSET, PROBE, as well as others demonstrate the **read geometry** module.

NAME

read image – read image file from disk into a field

SUMMARY

Name	read image				
Type	data				
Inputs	none				
Outputs	field 2D 4-vector byte				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Read Image	Browser	not applicable		

DESCRIPTION

The **read image** module reads an image file from disk and outputs the image as a "field 2D 4-vector byte". Each field element represents a pixel. The data value for each element is a 4D vector of bytes, laid out as follows:

auxiliary	red	green	blue

this field interpreted as pixel's opacity value

these three fields make up pixel's color value

The auxiliary field ("alpha") is sometimes used to store opacity information on a per-pixel basis.

PARAMETERS

Read image

A file browser window that allows you to specify the name of the image file to be read.

OUTPUTS

Data Field The output data is a 2D block of pixels. The data set at each point of the 2D field will be a 4-vector of bytes in the AVS *image* format.

IMAGE FILE FORMAT

read image expects its input file to be in the following format:

```
4-byte integer      nx: number of pixels in X dimension
4-byte integer      ny: number of pixels in Y dimension
nx * ny * 4 bytes   pixel data (4 bytes per pixel)
```

RELATED MODULES

Image processing:

contrast, threshold, histogram stretch, clamp, interpolate
luminence, generate filters, sobel, convolve, local area ops

Decompose/compose images from separate bands:

extract scalar
combine scalars

Display picture:

display image

Turn image data into a pixmap for more powerful viewing techniques:

image to pixmap
transform pixmap
display pixmap

SEE ALSO

The example scripts CONTRAST, FIELD IMAGE, PRINT FIELD, as well as others demonstrate the **read field** module.

NAME

read plot3d – read a PLOT3D format file into an AVS field

SUMMARY

Name read plot3d

Unsupported this module is in the unsupported library

Type data

Inputs *none*

Outputs field 1D, 2D, or 3D irregular 3-, 4-, or 5-vector float

Parameters

Name	*Type*	*Default*
X[YZ] Grid File	browser	
Q Solution File	browser	
Multigrid	boolean	false
w/IBLANK	boolean	false
Data Format	choice	binary
Organization	choice	3D/whole
Grid number	integer	1

DESCRIPTION

The **read plot3d** module reads computational fluid dynamics data files in the National Aeronautics and Space Administration's PLOT3D format (see reference) and converts them into AVS field format. There are two types of PLOT3D files, the XYZ grid files that specify the irregular coordinate information, and the 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. The data within the files can be in either binary, or FORTRAN formatted or unformatted format. 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 (1D), Y momentum (1D and 2D), Z momentum (1D, 2D, and 3D), 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 displays a control panel with a set of radio button switches for specifying the multigrid attribute, the IBLANK attribute, dimensionality and organization, a set for the input file type, and an integer dial for the grid number (this dial is not displayed for single-grid files). You specify the Q solution file and XYZ grid file through two separate file browsers. The file selections are cancelled whenever the selection of data format or organization is changed. In addition, if the module has successfully produced an output field, and subsequently one of the file browsers is used to select a file, the file selection for the other browser is cancelled. These actions prevent the module from attempting to mesh unrelated XYZ and Q files when you change from one data set to another.

PARAMETERS

multigrid A toggle that specifies whether the file has a single grid or multiple grids.

grid number
Which grid, in multi-grid files, to use to produce the AVS field.

w/IBLANK
A toggle that specifies whether or not the XYZ file contains an array of IBLANK values for each point in the grid.

data format
A set of radio buttons to specify how *both* the X[YZ] grid file and Q solution file are organized:

binary
The file is written in binary format, that is, the machine's native representation for integers (for the indices) and single precision floating point (for the points and values).

formatted
The file is written as FORTRAN formatted ASCII output.

unformatted
The file is written as FORTRAN unformatted output, including any framing values used by the machine's native FORTRAN compiler.

Organization
A set of radio buttons to specify the dimensionality and organization of the data for both the X[YZ] grid file and the Q solution file.

1D Input files are each a sequence of 1-dimensional arrays of values.

2D Input files are each a sequence of 2-dimensional arrays of values, stored in natural FORTRAN order.

3D/whole
Input files are each a sequence of 3-dimensional arrays of values, stored in natural FORTRAN order.

3D/planes
Input files are each a sequence of sets of 2-dimensional arrays of values, where each set of arrays corresponds to a single plane from the entire array.

X[YZ] File A file browser widget for specifying the grid file.

Q (solution) File
A file browser widget for specifing the solution file.

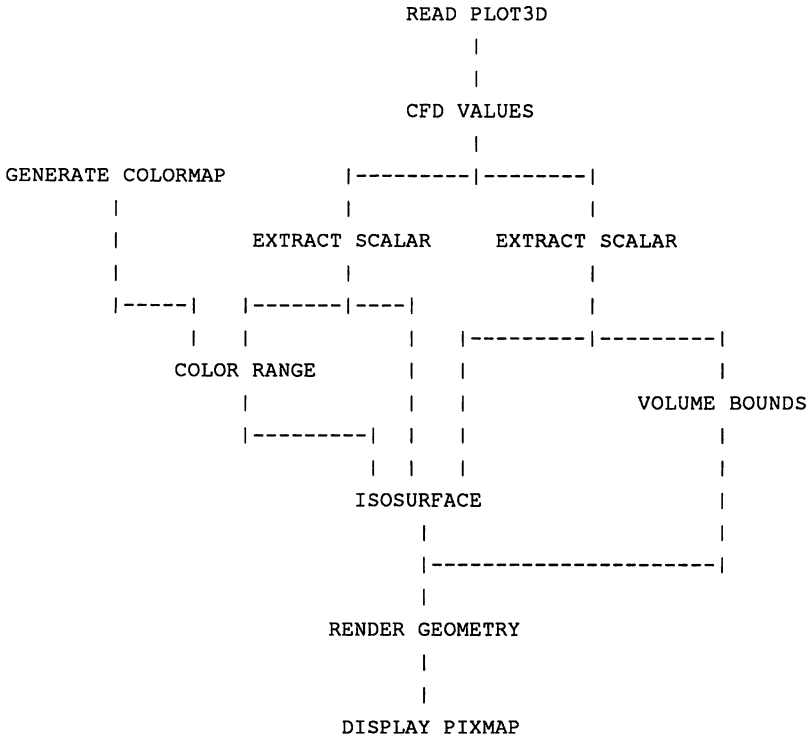
OUTPUTS

Data Field (field irregular float 1D, 2D, or 3D of 3-, 4-, or 5-vector)
The AVS field output will match the dimensionality of the original PLOT3D dataset. 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 AVS field represents only the one specified grid of multi-grid parameter files. There is no way to pack multiple grids into an AVS field.

EXAMPLE

The following example shows how **cfd values** and **read plot3d** can be used. The **extract scalar** on the right extracts one value from the 12-vector that **cfd values** outputs. **isosurface** computes the isosurface for this scalar output, and **volume bounds**

is used to draw a bounding box for the data. The left hand **extract scalar** module extracts another value from **cfd values** output. This second scalar field is used to color the isosurface. The **color range** module is used to scale the colormap to the range of the extracted cfd value. This network will allow you, for example, to generate an isosurface of the density in a field, and then color this isosurface based on the temperature values at each point on the isosurface.



RELATED MODULES

The **cfd values** modules is particularly designed to compute 7 common CFD values such as temperature, pressure, enthalpy, mach number, and energy from the five values provided by this and any other CFD input modules.

Modules that can process **read plot3d** output:

cfd values
extract scalar
extract vector
volume bounds
isosurface
arbitrary slicer

REFERENCES

Pieter Buening, **PLOT3D Reference Manual.**

SEE ALSO

The example scripts READ PLOT3D and CFD VALUES demonstrate the **read plot3D** module.

NAME

read ucd – read UCD structure from a disk file

SUMMARY

Name	read ucd				
Type	data				
Inputs	none				
Outputs	ucd structure				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Read UCD	browser			
	Cell Connect	boolean	off		
	Node Data	choice	<data 1>		
	Cell Data	choice	<data 1>		
	Model Data	choice	<data 1>		

DESCRIPTION

read ucd 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. Specifically, if a file is binary then it is assumed that it is in the format output by the module **write ucd**.

ASCII UCD files have a simple format which will be described below, in the section titled *ASCII File Format*. For a more detailed description of both ASCII and binary file formats, see Appendix E of the *AVS Developer's Guide*.

PARAMETERS

Read UCD A file browser window to specify the name of the UCD file to be read.

Cell Connect
A boolean switch. When **Cell Connect** is selected the connectivity list is calculated. For each node in the UCD structure, this lists which cells the node is incident on. This is needed when using the **ucd streamline** module to produce streamlines.

Node Data A set of radio buttons that shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure, "<no data>" will be displayed on the button. These buttons are used only to display the data components at the nodes; pressing them has no effect on the module.

Cell Data A set of radio buttons that shows the label attached to each cell data component. If there are no labels, the default "<data 1>, <data 2>, ..." is displayed. If there is no cell data in the structure, "<no data>" will be displayed on the button. These buttons are used only to display the data components at the cells; pressing them has no effect on the module.

Model Data
Data associated with the structure as a whole is also known as "model data". A set of radio buttons that shows the label attached to each model data component. If there are no labels, the default "<data 1>, <data 2>, ..." is displayed. If there is no whole-structure data, "<no data>" will be displayed on the button. These buttons are used only to display the model data components; pressing them has no effect on the module.

OUTPUTS

UCD structure
The output structure is in AVS unstructured cell data format.

ASCII FILE FORMAT

If a UCD file is in ASCII, it has the following format. For a more complete description of UCD file formats, as well as a discussion of UCD data in general, see Appendix E of the *AVS Developer's Guide*.

```
# <comment 1>
   .
   .
   .
# <comment n>
<num_nodes> <num_cells> <num_ndata> <num_cdata> <num_mdata>
<node_id 1> <x> <y> <z>
<node_id 2> <x> <y> <z>
   .
   .
   .
<node_id num_nodes> <x> <y> <z>
<cell_id 1> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
<cell_id 2> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
   .
   .
   .
<cell_id num_cells> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
<node_data_name 1>
<node_data_name 2>
   .
   .
   .
<num_comp> <size comp 1> <size comp 2> ... <size comp num_comp>
<comp_name 1> <units_name 1>
<comp_name 2> <units_name 2>
   .
   .
   .
<comp_name num_comp>, <units_name num_comp>
<node_id 1> <node_data 1> ... <node_data num_ndata>
<node_id 2> <node_data 1> ... <node_data num_ndata>
   .
   .
   .
<node_id num_nodes> <node_data 1> ... <node_data num_ndata>
```

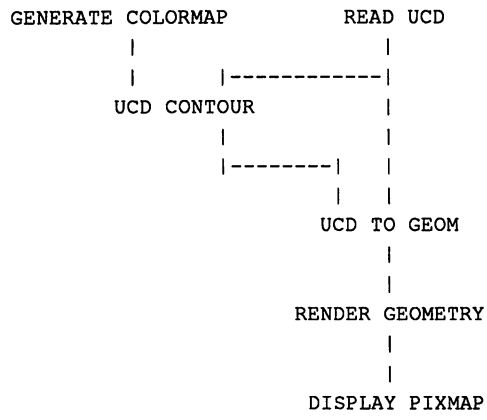
SAMPLE UCD FILE

The following is an example of a simple UCD file:

```
8 1 1 0 0                    #i.e. 8 nodes, 1 cell, 1 component of node data
1  0.000   0.000   1.000     #node coordinates
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     #cell id, material id,cell type, cell vertices
1 1                          #num data components, size of each component
stress, lb/in**2             #component name, units name
1  4999.9999                 #data value for each node component
2  18749.9999
3  37500.0000
4  56250.0000
5  74999.9999
6  93750.0001
7  107500.0003
8  5000.0001
```

EXAMPLE

The following network reads in a UCD ASCII file (*.inp* suffix), and displays it:



RELATED MODULES

Modules that can process **read ucd**'s output:

ucd to geom, ucd crop, ucd threshold, ucd extract, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d, ucd legend, ucd probe, ucd streamline, write ucd, ucd tracer.

SEE ALSO

The example script READ UCD demonstrates the **read ucd** module.

NAME

read volume – read volume file from disk into a field

SUMMARY

Name	read volume	
Type	data	
Inputs	none	
Outputs	field 3D scalar byte	
Parameters	*Name*	*Type*
	Read Volume	Browser

DESCRIPTION

The **read volume** module reads a disk file in *volume data* format and outputs the data as a "field 3D scalar byte". It is used to read data files containing scalar-valued volume data (e.g. CAT scan data, NMR data).

PARAMETERS

read volume

A file browser allows you to specify the name of the file that contains the volume data set.

OUTPUTS

Data Field (field 3D scalar byte)

The output is the byte data cast as the scalar data in a 3D *field*.

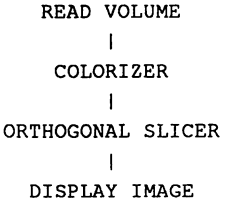
VOLUME DATA FILE FORMAT

read volume expects its input file to be in the following format:

```
(1 byte)    nx:         number of voxels in X
(1 byte)    ny:         number of voxels in Y
(1 byte)    nz:         number of voxels in Z

(nx * ny * nz bytes): volume data elements
```

EXAMPLE

This simple example displays a volume data set.



RELATED MODULES

Colormaps:

generate colormap, read colormap

Filters:

clamp, contrast, crop, downsize, field to byte, field to double, field to float, field to int, histogram stretch, interpolate, mirror, offset, transpose, colorizer, compute gradient, gradient shade

Mappers:

dot surface, arbitrary slicer, bubbleviz, orthogonal slicer, field to mesh, isosurface, volume bounds

Renderers:

alpha blend, display image, render geometry, vbuffer

SEE ALSO

The example scripts ANIMATED FLOAT, BRICK, and THRESHOLDED SLICER demonstrate the **read volume** module.

NAME

render geometry – convert geometric description to image (Geometry Viewer)

SUMMARY

Name	render geometry
Type	data output
Inputs	geometry (optional, multiple) field 2D/3D 4-vector byte (optional, hardware texture mapping systems only)
Outputs	pixmap
Parameters	*Name* add to object transform

DESCRIPTION

The **render geometry** module provides access within an AVS network to the complete Geometry Viewer subsystem. Many different modules can supply input geometries. That is, many *geometry*-format outputs can be connected to **render geometry**'s geometry input port. All the objects will be combined into a single scene. Each module providing input to **render geometry** can define attributes and geometries for any number of objects. Each of these modules can also define a hierarchical relationship among its objects.

You can also invoke **render geometry** with no inputs, so that the "scene" is initially empty. Objects can be added to a scene either by upstream modules or by the **Read Object** selection on the **render geometry** control panel. Geometries and descriptions sent by upstream modules can be saved to files using the **Save Object** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Object**.

SPECIAL CONSIDERATIONS

This module is special: instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level application menu of the Geometry Viewer subsystem. Thus, when you add the **geometry viewer** icon to a network, no page is added to the Network Control Panel. There are two ways to access the Geometry Viewer menu:

- Click the small square in **render geometry** icon with the left mouse button.
- Click the **Geometry Viewer** button located at the top of the Network Control Panel. This button is always visible, even when there is no active network.

In some circumstances, it is useful to be able to access both the Geometry Viewer control panel and the Network Control Panel simultaneously. They both occupy the same screen position, along the left edge of the screen. In these cases, use the X Window System window manager to move the one of these menu windows out of the way.

The **geometry viewer**'s control panel also differs from that of other modules in these ways:

- The Network Editor's **Layout Editor** cannot be used to rearrange Geometry Viewer controls.
- If a network includes more than one instance of **render geometry**, AVS does *not* create a separate control panel for each instance. Each **render geometry** sends its output to a different window, but the same Geometry Viewer application menu controls all the windows. The module whose output window is currently highlighted in red is the one being controlled. To switch the $_{\text{focus}}$ to another **render geometry** output window, just click in it with any mouse button.

INPUTS

Geometry (optional, multiple; geometry)

The input data can be any AVS *geometry*. More than one geometry can be input to this port. All the geometries will be combined into the same "scene".

Texture (optional; field 2D/3D 4-vector byte uniform)

This input port is only supported on systems which have both software and hardware support for 2D/3D texture mapping. The Stardent GS-series machines have this feature. Other systems (e.g., the Stardent Titan series machines) may not.

The optional input provides a way to perform "dynamic texture mapping". The AVS 2D or 3D field of color values input to this port it is available as a dynamic texture. From within the "Edit Texture" submenu under **Objects**, you can bind this texture map to a particular object.

PARAMETERS

add to object transform

This parameter can be attached to the dialbox or the Spaceball, allowing these devices to control object transformations. In such cases, you can still control transformations using the mouse:

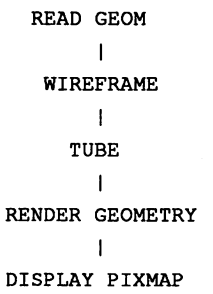
Mouse	Transform
middle	rotate
right	translate in plane of screen
middle with SHIFT key	scale
right with SHIFT key	translate perpendicular to plane of screen

OUTPUTS

pixmap The output is a pixmap containing a *scene* that includes all the input objects.

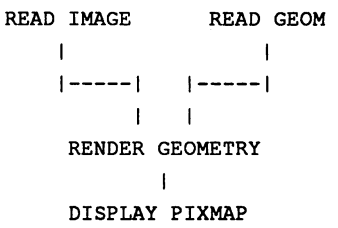
EXAMPLE 1

This network creates a tube version of an object:



EXAMPLE 2

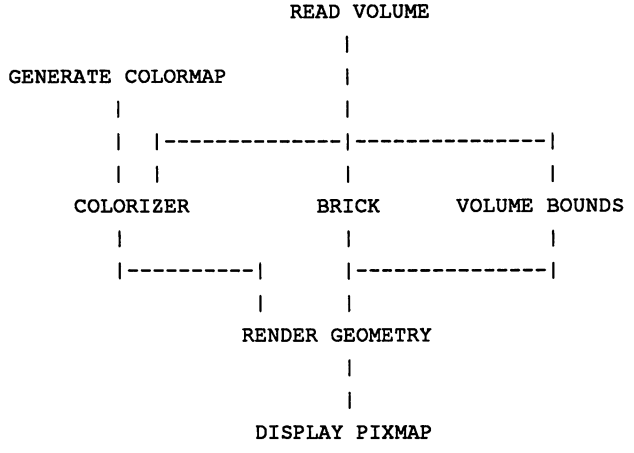
This network implements dynamic textures:



EXAMPLE 3

The following network shows how **render geometry**'s left input port is used to perform 2D/3D texture mapping using the **brick** module. The network reads a byte

volume which is sent to **colorizer** to paint the byte values as colors, to **brick** to map the surfaces, and to **volume bounds** to draw a box around the limits of the volume. The **generate colormap**, and **colorizer** create the 3D texturemap, which is fed to **render geometry** through the left input port.



RELATED MODULES

display pixmap, read geom, pdb to geom, render manager

SEE ALSO

The *Geometry Viewer* chapter of the *AVS User's Guide.*

The example scripts BRICK, FLIP NORMALS, PDB TO GEOM, as well as others demostrate the **render geometry** module.

NAME

render manager – share geometries among subnetworks

SUMMARY

Name	render manager
Unsupported	this module is in the unsupported library
Type	data output
Inputs	geometry
Outputs	none

Parameters

Name	*Type*
Create New Window	one shot
Active Windows	choice

DESCRIPTION

The **render manager** module takes geometries as input, uses the AVS Geometry Viewer to render them, and displays the results in one or more windows. This module is very similar to the **render geometry** module, with these differences:

- **render manager** creates its own pixmap and window on the screen, rather than relying on **display pixmap**. An initial window is created by default.
- **render manager** has a built-in mechanism for creating and selecting output windows. A set of windows is shared among **render manager** modules in separate subnetworks. At any moment, one of them — the *current output window* — is shared by all the **render manager** modules in all subnetworks. This window displays the combined results of all these modules.

It is possible to create a new output window, which automatically becomes the shared current output window. This provides a powerful capability for exploring differences between datasets, or different mappings of the same dataset. See the **Create New Window** parameter below.

This module is used by the AVS Image Viewer and Volume Viewer subsystems.

INPUTS

Geometry (geometry)
Any AVS *geometry*.

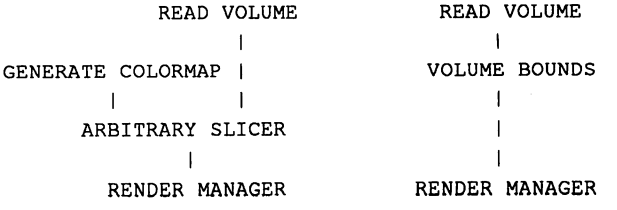
PARAMETERS

Create New Window
Click this button to create a new output window, which becomes the current output window. Subsequent geometric input is rendered into this window, until such time as you change the current output window again (perhaps by creating yet another window).

Active Windows
A choice menu that lists all the output windows, showing which one is current. You can also make an output window current by pressing any mouse button in the window itself.

EXAMPLE

Suppose you have built the following two networks:



When you select a volume dataset (e.g. *hydrogen.dat*) for the **arbitrary slicer** subnetwork, the slice is rendered by the Geometry Viewer, and a window is created to display the picture. If you select the same dataset in the **volume bounds**

subnetwork, the bounds are rendered and displayed in the same window.

If you click **Create New Window**, and then select a new dataset was selected in the **arbitrary slicer** subnetwork, it (and it alone) is displayed in the new window. The geometries in the original window do not change.

RELATED MODULES

Same as for **render geometry**.

NOTES

The output window(s) are not destroyed until *all* **render manager** modules are destroyed.

NAME

replace alpha – replace the alpha channel (transparency) in an image

SUMMARY

Name	replace alpha
Type	filter
Inputs	field 2D uniform 4-vector byte *(image)* field 2D uniform scalar byte *(new alpha*
Outputs	field 2D uniform 4-vector byte *(image)*
Parameters	none

DESCRIPTION

The **replace alpha** module replaces the alpha (opacity) byte of all the pixels in an image with the byte value from a 2D uniform *scalar* field of the same dimensions. This 2D uniform scalar field is usually produced by passing the image through the **luminence** or **extract scalar** module, then perhaps performing further imaging techniques on the scalar value (e.g. **contrast**), The modified alpha is then rejoined with the original image using **replace alpha**.

INPUTS

Image (required; field 2D uniform 4-vector byte)
The image whose alpha byte will be replaced. This is the right input port on the **replace alpha module.**

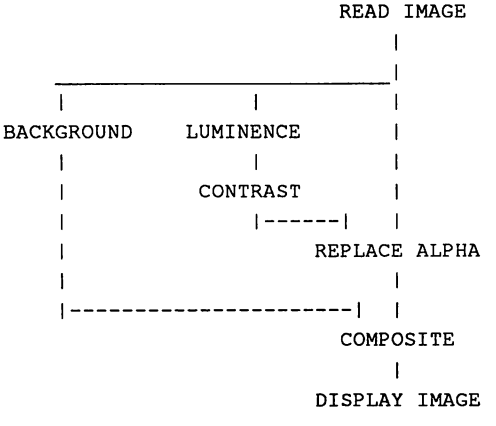
Data Field (required; field 2D uniform scalar byte)
The field of byte values, with the same dimensions as the input image, to use as the replacement alpha values. This is the left input port on the **replace alpha** module.

OUTPUTS

Image (field 2D uniform 4-vector byte)
The output image has the same dimension as the input image.

EXAMPLE 1

The following network reads an image, computes its luminence, uses that to create an alpha mask, generates a shaded background, and composites the rendered image over the shaded background image.



RELATED MODULES

Modules that could provide the **Image** input:

contrast
pixmap to image
read image
threshold
Any module that produces an image as output

Modules that could provide the 2D scalar field:

luminence
extract scalar
Any modules that can output a 2D scalar field

Modules that can process **replace alpha** output:

composite
write image
image to pixmap

See also **background, luminence**

SEE ALSO

The two example BACKGROUND scripts demonstrate the **replace alpha** module.

NAME

samplers – extract a subset of locations from a 3-vector 3D field

SUMMARY

Name	samplers				
Type	data				
Inputs	field 3D float *any-coordinates* upstream transform (*optional, invisible, autoconnect*)				
Outputs	field 3D irregular *(locations)*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Choice	choice	point		
	N Segment	integer dial	16	2	64

DESCRIPTION

The **samplers** modules extracts a subset of coordinates from a 3D AVS field of floating point data, producing an output field that is "3-space irregular," i.e., it contains a series of coordinates (also called "scattered data") in 3-space (which can correspond to a uniform, rectilinear, or irregular grid) but without any data values associated with them.

samplers's main purpose is to simultaneously control two or three of the **hedgehog/particle advector/stream lines** modules. For example, you can show the streamlines and hedgehog vectors for the same sample set of points together.

samplers can extract a single location coordinate point, a series of points along a line through the 3D field, a series of points along a circle in a 3D field, a series of points on a plane in a 3D field, or a series of points in a volume of a 3D field.

How many points **samplers** extracts (the sample resolution) depends upon the N **Segment** dial setting.

When the output "field of locations" is connected to the *left* input port of the three volume-of-vectors mapping modules (**hedgehog, particle advector,** and **stream lines**), these modules will calculate and display only the subset of points in the input field.

If you don't connect **samplers** to the left input port on **hedgehog/particle advector/stream lines,** these modules create their own internal parameters that function identically to the **samplers** module, like the other parameters-as-data modules (**integer,** etc.),

When **samplers** and **render geometry** coexist in a network, the two are connected automatically through an invisible "upstream transform" port. "samplers" becomes a selectable object in the Geometry Viewer. If you select and move the "samplers" geometry object, **render geometry** informs the **samplers** module of the new location of the sample subset, **samplers** recalculates the "field of locations" and the **hedgehog/particle advector/stream lines** module redraws the data at the new location. The effect is direct mouse-manipulation control over a line, circle, plane, or volume sampling subset.

If you want less than a whole plane or whole volume sample, use the **crop** module on the input to **samplers,** while letting the full field through to **hedgehog/particle advector/stream lines**'s *right* input port. You can then move the subset volume around the whole volume of the field.

INPUTS

Data Field (required; field 3D 3-vector *any-data any-coordinates*)
The input field is a 3D 3-vector of any coordinate type and any data type.

Upstream Transform (optional, invisible, autoconnect)
When the **samplers** module coexists with the **render geometry** module in a network, **render geometry** feeds information on how the "samplers" object has been moved in the Geometry Viewer back to this input port on the **samplers** module. The information is relayed through the **hedgehog, particle advector,** or **stream lines** module. The modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over the **samplers** sample set.

PARAMETERS

point
line
circle
plane
space A set of radio button choices that determines what type of geometric construct the sample locations will be taken from. You can move each of the structures listed below around the volume of data using the Geometry Viewer's transformations.

point causes a single data location to be output, no matter what the **N Segment** parameter value is. This is the default.

line causes **N Segment** sample locations to be taken along a line through the volume.

circle causes **N Segment** sample locations to be taken around a "ring" within the volume space.

plane causes **N*N Segment** sample locations to be taken along a plane slice through the volume space.

space causes **N*N*N segment** sample locations to be taken throughout the whole volume space. The only way to subset the volume is to pass it through the **crop** module before it reaches **samplers.**

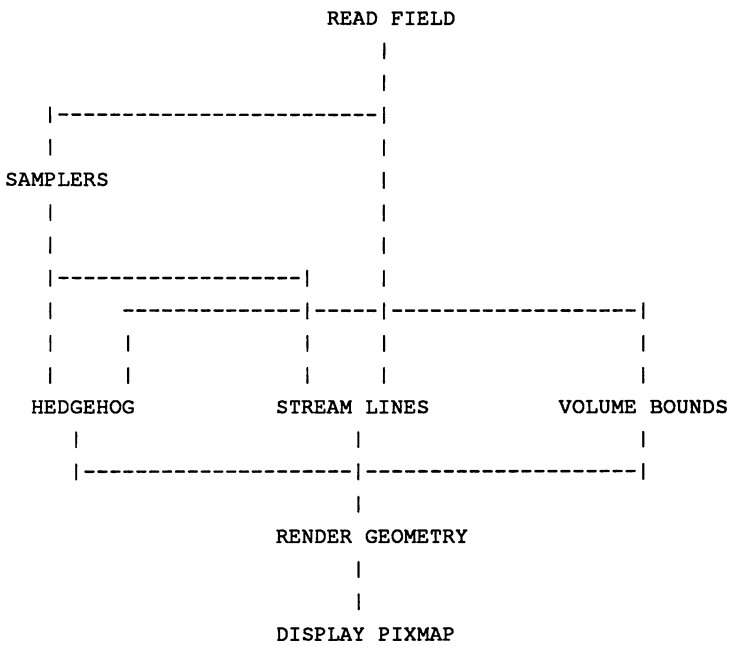
N Segment An integer dial that determines how many sample locations to extract from the volume. It is ignored for **point.** The default is 16, the minimum is 2, and the maximum is 64.

OUTPUTS

Data Field (field 3D irregular)
The output field is a 3D lattice of locations from the original input field, with no data values at each node. It is passed down to the **hedgehog, particle advector,** or **stream lines** *left* input port, telling them what subset of their complete data to map.

EXAMPLE 1

The following network reads in a 3-vector field, extracts a sample subset, then maps it as both a **hedgehog** and **stream lines** representation, finally displaying it surrounded by volume bounds.



RELATED MODULES

Modules that could provide the **Data Field** input:

read field

Modules that can process **sampler** output:

hedgehog
particle advector
stream lines

SEE ALSO

The example script PARTICLE ADVECTOR demonstrates the **sampler** module.

NAME

scatter dots – generate spheres at points in 3D space

SUMMARY

Name scatter dots

Type mapper

Inputs field 1D real 3-space irregular (a "scatter" field)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Choices*
Connect the dots	toggle	off			on, off
Radius	Real	0.0	0.0	1.0	

DESCRIPTION

The **scatter dots** module generates spheres of various radii at the coordinate locations in a specified field. For a scalar field, each sphere's radius is proportional to the scalar value, and the sphere is always colored white. If the field is a 4-vector float (such as that produced by the **bubbleviz** module), only the first element of the vector determines the sphere's radius. The other three elements are interpreted as red-green-blue color values (normalized to the range 0..1).

INPUTS

Point List (required; field 1D 3D float *irregular*) The input field must be a list of points in 3D space, with a *float* value specified at each point.

PARAMETERS

Connect the dots (toggle)

- If OFF, a sphere is drawn at each point in the field. The radius of the sphere is specified by the field element's scalar data value. (If the field has vector data, the value of the first vector element is used and the other values determine the sphere's color.
- If ON, the points are represented as dots, connected with a single polyline (in the order specified by the 1D array). If the input field has 4-vector float data, the last three vector elements are ignored. No spheres are drawn in this case.

Radius (real)

Radius is a floating-point multiplier factor for the sphere radii.

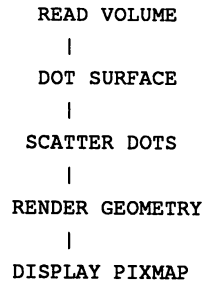
OUTPUTS

Geometry (geometry)

The output is an AVS *geometry*.

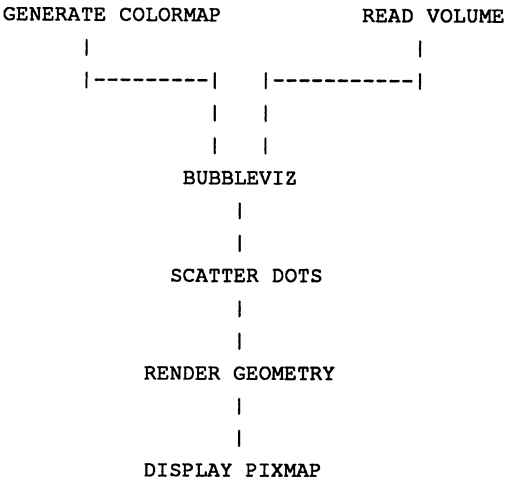
EXAMPLE 1

The **scatter dots** module can be used in combination with the **dot surface** module as follows:



EXAMPLE 2

The **scatter dots** module is required to make **bubbleviz** work properly:



RELATED MODULES

read geom, geom to scatter, tube, wireframe, render geometry

SEE ALSO

The example scripts BUBBLEVIZ, and DOT SURFACE demonstrate the **scatter dots** module.

NAME

shrink – make polygons of a geometry object smaller

SUMMARY

Name shrink

Type filter

Inputs geometry

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
offset	float	1.0	0.0	1.0

DESCRIPTION

The **shrink** module transforms an AVS *geometry*, so that each vertex of each polygon is translated towards (or away from) the polygon's centroid (center of gravity). This has the effect of creating spaces between polygons, and is useful for visualizing the internal geometry of an object.

INPUTS

Geometry (required; geometry) An AVS geometry, created with the *libgeom* library or by another AVS module.

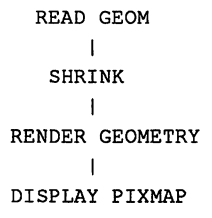
PARAMETERS

offset The amount by which each vertex is translated. Positive values collapse the geometry inward. Negative values create a "blow-up" of the geometry.

OUTPUTS

Geometry A geometry that represents the same object(s) as the input data.

EXAMPLE



RELATED MODULES

read geom, flip normal, tube, render geometry

LIMITATIONS

This module works only for polytriangle strips and meshes; it does not work for polyhedra.

This module doesn't copy UV data, used in texture mapping.

This module can increase the size of the data: it can generate up to five times the number of triangles for polytriangle objects, and up to three times the number of vertices for meshes.

SEE ALSO

The example script SHRINK demonstrates the **shrink** module.

NAME

sobel - apply an edge detecting filter to 2D field

SUMMARY

Name	sobel
Type	filter
Inputs	field 2D *n-vector any-data any-coordinates* ("image")
Outputs	field of same type as input
Parameters	none

DESCRIPTION

sobel uses the "sobel operator" for finding edges in a 2D byte field. The typical use is to find edges in images prior to some segmentation operation, such as dividing the image into regions that correspond to the individual objects in the picture. The Sobel operator consists of two 3x3 filters. One detects changes in an image in the x direction; thus detecting vertical edges. The other detects changes in the y direction, and thus is used to detect horizontal edges.

sobel takes the two sobel filters and applies them to a source field to produce a destination field. Both the source and destination fields must be 2D. Typically, the source and destination fields will be AVS *images*, but they might also be 2D slices of 3D fields.

sobel accepts vectors of any size containing data of any type. In the case of an image, which is a 2D field of 4-byte vectors, **sobel** disregards the alpha bytes and separates the red, green and blue bytes. Then it applies the filter separately to each color byte, before reassembling the bytes into 4-vector image format.

In the case of non-image data, for example a 2D field of 5-vector floats, **sobel** handles one component of the vector at a time. All data-types are converted to floats during computation and then converted back in **sobel**'s output.

In order to handle edge effects, a border around the perimeter of the source field is not operated on. The border is one pixel wide.

INPUTS

Data Field (required; field 2D *n-vector any-data any-coordinates*)
A 2D AVS field, typically an image, to be operated on.

OUTPUTS

Output Field
The output field is the same type as the input data field.

EXAMPLE 1

The following network reads in an image, applies the sobel operation to it, and displays the resulting image:

RELATED MODULES

Modules that could provide the **Data Field** input:

read image
pixmap to image
orthogonal slicer
any other module which outputs a 2D field

Modules that can process **sobel**'s output:

display image

image viewer
any other module which takes a 2D field as input

Also related:

generate filters
convolve
local area ops

SEE ALSO

The example script SOBEL demonstrates the **sobel** module.

NAME

statistics – display statistics on AVS field contents including min and max values

SUMMARY

Name	statistics			
Type	data output			
Inputs	field *any-dimension n-vector any-data any-coordinates*			
Outputs	none			
Parameters	*Name*	*Type*	*Default*	
	Compute Median		switch	off

DESCRIPTION

The **statistics** module displays global statistical information about field data. **statistics** scans the input field and produces a small output table like the following:

```
Field Statistics
================
Dimensions:          628 184 (x4)
Min/Max:             0.000000 255.000000
Mean:                58.934429
Median:
Standard Deviation:  76.030327
Skewness:            1.328104
Kurtosis:            0.686514
```

The output is displayed in an output text widget. Calculating the Median value is compute-intensive; it is only calculated if the **Compute Median** switch is turned on.

Use the **statistics** module when you need to know what a field's min/max are. This information is often useful if you wish to scale the dials in downstream modules which are operating on the same input field. The output values mean:

Dimensions
The dimensions of the field, with vector length, if applicable.

Min/Max
The lowest and highest values in the data set.

Mean
The average of the data.

Median
The center value of a sorted list of the data.

Standard Deviation
The square root of the sum of the squares of the deviations.

The next two values are derived from comparing the distribution of the values to an ideal Gaussian "standard" distribution.

Skewness
When positive, the right side of the distribution curve is "steeper" than the left. When negative, the left side is "steeper."

Kurtosis
When positive, the data is more "spikey" than a standard distribution. When negative, the data is more broadly-distributed than a standard distribution.

INPUT

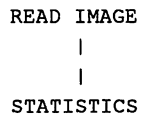
Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input AVS field can be any dimension, with any vector length, and of any data type.

PARAMETERS

Compute Median
A toggle switch that makes **statistics** also go through the compute-intensive calculation of the field's median. It is off by default.

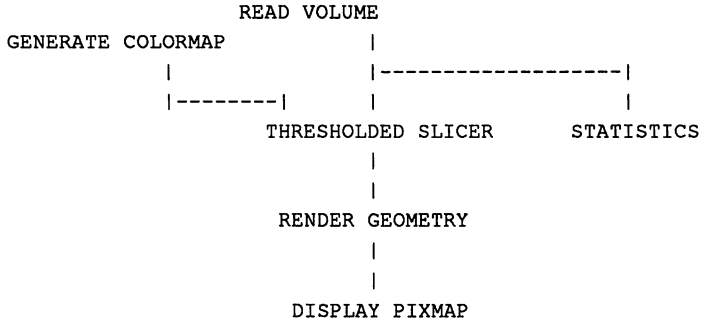
EXAMPLE 1

The following network computes statistics on an image.



EXAMPLE 2

The following network shows how you might use the **statistics** module to determine the min and max values in a 3Dfield, so that you could scale the dials on the **thresholded slicer** module accordingly.



RELATED MODULES

print field
compare field

SEE ALSO

The example script STATISTICS demonstrates the **statistics** module.

NAME

stream lines – generate stream lines for a vector field

SUMMARY

Name stream lines

Type mapper

Inputs field 3D 3-vector *any-data any-coordinates*
field irregular *(optional, from samplers module)*
upstream transform *(optional, invisible, autoconnect)*

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*MaxValues*
width	integer	12	4	32
length	integer	12	4	128
step	float	0.02	0.0	1.0
N segment	integer	16	2	64
Sample	radio	point		point, line, circle, plane, space
Mode	radio	lines		lines, mesh
Method	radio	Euler		Euler, Runge-Kutta

DESCRIPTION

The **stream lines** module generates streamlines based on a *field* that is a volume of 3D vectors. It places a "sample" of points at a parameter-controlled starting location in the volume. The number of points is also parameter-controlled; their orientation is mouse-controlled, using the same "virtual trackball" paradigm as the Geometry Viewer.

Then, for every time step, **stream lines** advances each sample point through space, based on the interpolated value of the vector field at its present position. The result is a set of stream lines showing the progress of massless particles through a vector field.

This module is similar to the **particle advector** module, except that the result is a static set of lines (or a surface) instead of a dynamically updated set of spheres.

INPUTS

Data Field (required; field 3D 3-vector *any-data any-coordinates*)

The input field must be a 3D uniform field. The data for each field element must be a 3D vector of any primitive data type, representing the components of a velocity vector.

Sample Field (optional; field irregular)

This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **stream lines** will take these locations and use them as the sample of starting for points for the stream lines.

Note that, when the **stream lines** module receives input locations from **samplers**, **stream lines's N segments** dial, and its **Sample** buttons disappear from the control panel.

Upstream Transform (optional, invisible, autoconnect)

When the **stream lines** and **render geometry** modules coexist in a network, they communicate through a normally-invisible data port. "streamline" shows up as an object in the Geometry Viewer. When you select the streamline object and move it, **render geometry** informs the **stream lines** module what the sample's new location is, and **stream lines** recalculates the location and streamlines it is displaying, accordingly.

This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **stream lines** module's sample of locations.

PARAMETERS

Width The density of points in the sample set.

Length A scale factor, which multiplies the length of the streamline segments generated during each time step.

Step Determines the time step for the interactive computation. The larger the value, the greater the interval.

N segments
An integer value which determines the number of points for which stream lines are computed. This controls the density of the stream lines output by **stream lines**.

Sample (radio buttons) Specifies the configuration of points from which stream lines will be drawn: point, line, circle, plane, or space.

mode (radio buttons) The default mode, **lines**, causes a stream line to be produced from each point in the sample set. The **mesh** mode applies only to line and circle samples. In this mode, a sample line or circle sweeps out a surface (manifold or cylinder) instead of a set of stream lines. If plane or space is selected as the sample, the **lines**, and **mesh** buttons disappear from the control panel. This is true even when the sample is received from the **samplers** module.

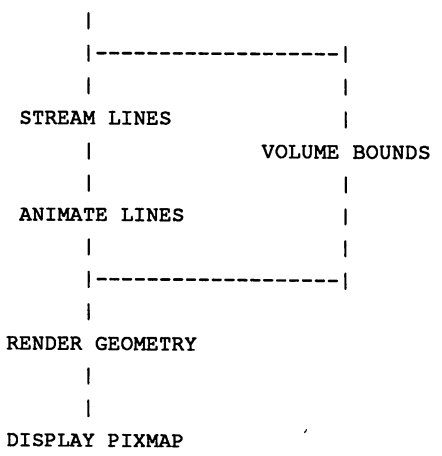
method (radio buttons) The buttons **Euler** and **Runge-Kutta** select the method used to calculate the next position of a sample particle. The **Euler** method is faster, involving a single vector in the input field. The **Runge-Kutta** method involves an interpolation, and produces considerably more accurate results.

OUTPUTS

Streamlines (geometry)
A set of disjoint lines.

EXAMPLE

The following network reads in a 3D vector field, and calculates streamlines for the field. **animate lines** is used to dynamically represent the output of **stream lines**.



RELATED MODULES

hedgehog, particle advector, stream lines

RELATED MODULES

animate lines hedgehog particle advector samplers

SEE ALSO

The example script STREAMLINES demonstrates the **stream lines** module.

NAME

threshold – restrict values in data field

SUMMARY

Name	threshold				
Type	filter				
Inputs	field *any-dimension n-vector any-data any_coordinates*				
Outputs	field of same type as input				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	thresh_min	float	0.0	none	none
	thresh_max	float	255.0	none	none

DESCRIPTION

The **threshold** module transforms the values of a field as follows:

- Any value less than the value of the **threshold_min** parameter is set to 0.
- Any value greater than the value of the **threshold_max** parameter is set to 0.
- All values within the **threshold_min**-to-**threshold_max** range are not changed.

After being **threshold**'ed, a data set's values are all either zero, or in this range:

thresh_min ≤ *value* ≤ **thresh_max**

Note the difference between the **clamp** and **threshold** modules:

- **threshold** sets values outside the specified range to be zero.
- **clamp** sets values outside the specified range to be the range's minimum and maximum values.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any_coordinates*)
The input data may be any AVS field.

PARAMETERS

thresh_min
The minimum threshold value.

thresh_max
The maximum threshold value.

OUTPUTS

Field Data The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
any other filter module

Modules that could be used in place of **threshold**:

clamp

Modules that can process **threshold** output:

colorizer
any other filter module

SEE ALSO

The example scripts CONTOUR GEOMETRY, and THRESHOLDED SLICER demonstrate the **threshold** module.

NAME

thresholded slicer – slice through volume data with high/low values invisible

SUMMARY

Name thresholded slicer

Type mapper

Inputs field 3D scalar *any-data any-coordinates* (volume)
upstream transform (optional, invisible, autoconnect)
colormap (required)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Resolution	int dial	12	12	64
Distance	float dial	0.0	unbounded	unbounded
Low Threshold	float dial	0.0	unbounded	unbounded
High Threshold	float dial	255.0	unbounded	unbounded
Sampling Style	choice	point	point,trilinear	
Refine	choice	coarse	coarse, fine	

DESCRIPTION

The **thresholded slicer** module extracts a 2D slice from a 3D volume of data. It differs from the **arbitrary slicer, orthogonal slicer,** and **brick** modules in that one can establish that numerical values below a **Low Threshold** value and above a **High Threshold** value will not be mapped—they will be given zero values in the output 2D slice. One can thus "edit out" or "crop" high and low values from a volume rendering.

thresholded slicer's slice plane is moveable through the Z axis with its **Distance** parameter dial.

It is also possible to move the slice plane arbitrarily within the volume using the mouse or the Geometry Viewer's transformation panel This is because **thresholded slicer** has an invisible "Upstream Transform" input port that allows it to automatically receive information from the **render geometry** module about how the "thresholded slice" object has moved,

The mapping technique for **thresholded slicer** is the same as **arbitrary slicer.** That is, the volume of data is represented as a 3D scalar field, defining a lattice within the volume. The slice plane is represented as a 2D grid, with parameter-controlled resolution. The intersection of the volume and the grid is a *mesh* of vertices in 3D space.

Each vertex in the mesh is assigned a color (with the input from **generate colormap** or the **colormap manager**) that corresponds to one or more values of the scalar field. Values below and above the **Low Threshold** and **High Threshold** settings are set to zero. Since, in general, the mesh vertices *do not* coincide with the original lattice points, an interpolation method can be used — see the *trilinear* input parameter below.

By default, the volume is placed at the origin and the slice plane is an X-Y plane placed midway through the Z dimension of the data.

You can control the resolution of the mesh using the **Resolution** parameter. At lower resolutions, fewer original data points are used in the computations; at higher resolutions, more points are used.

The optimal way to use this module is to start off with a low resolution mesh, position it as desired, then increase the resolution and turn on trilinear mapping and the Fine level of refinement.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)
The input data must be a 3D field, with a byte value at each location in the field. The field must be uniform.

Upstream Transform (optional, invisible, autoconnect)
When the **thresholded slicer** module coexists with the **render geometry** module in a network, **render geometry** feeds information on how the "thresholded slice" object has been moved in the Geometry Viewer back to this input port on the **thresholded slicer** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **thresholde slicer's** slice plane.

Colormap (required; colormap)
By default, the value computed for each vertex of the mesh is used as the hue in HSV space. The values are transformed to the range 0..255, and are then used as indexes into the colormap.

PARAMETERS

Resolution An integer dial that controls how many sampling points are taken through each dimension of the volume data. The default is a fairly low resolution 12. The maximum value is 64.

Distance A floating point dial widget that controls the movement of the slice surface in the Z direction. The 0.0 initial value is defined to be *midway* through the volume. Hence, a volume with a Z dimension of 64 has 0.0 in the middle, with +32.0 and -32.0 in either direction. The dial itself is unbounded. If you enter a value outside the actual volume, the slice surface disappears.

Low Threshold
A floating point dial, set by default to 0.0. Values in the volume below this dial setting do not generate any polygons.

High Threshold
A floating point dial, set by default to 255.0. Values in the volume above this dial setting do not generate any polygons.

Refine The intersection of the contour with the voxel is computed in a refinement loop. This selection chooses how many levels of refinement are performed. **coarse** is 2; **fine** is 8. Fine gives more accurate contours.

Sampling Style
A choice of two styles that control how each vertex in the output mesh is assigned a color:

- **If Point,** a nearest-neighbor algorithm is used. Each mesh vertex is assigned the byte value of the nearest point in the lattice.
- **If Trilinear,** a trilinear interpolation is performed. The value at each vertex depends on the byte values at the eight lattice points that are the corners of the "enclosing cube".

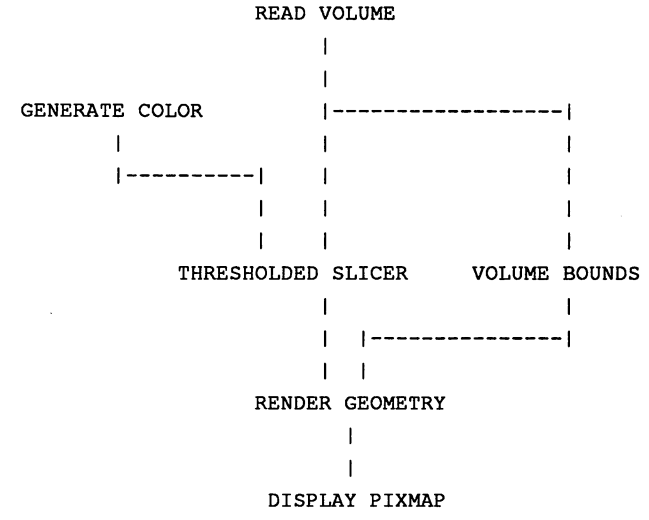
The trilinear interpolation method is more accurate but takes longer to compute, particularly with larger meshes.

OUTPUTS

Geometry (geometry)
The output is an AVS *geometry*.

EXAMPLE

This example shows the typical usage of the **thresholded slicer** module for byte data in the range 0-255:



The **volume bounds** modules gives a reference frame for orienting the slice plane. Often, an **isosurface** is also input to the **render geometry** module.

SEE ALSO

The example script THRESHOLDED SLICER demonstrates the **thresholded slicer** module.

NAME

tracer - perform ray-traced volumetric rendering on volume data

SUMMARY

Name tracer

Type mapper

Inputs field uniform 3D 4-vector byte (volume)
upstream transform (optional, autoconnect)

Outputs field 2D 4-vector byte (image)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
alpha scale	float	1.0	0.0	1.0
perspective	float	0.0	0.0	1.0
width	integer	64		
height	integer	64		
interpolate	toggle	off		

DESCRIPTION

tracer belongs to a family of modules (along with the unsupported modules **vbuffer** and **alpha blend**) that render volume data. **tracer** takes a volume, which can be visualized as a block of cubic "voxels" (volume elements), and generates a 2D image using ray-tracing. Each voxel in the volume has color and opacity values associated with it, consequently **tracer**'s input must be a 3D field of color values. In other words, the data at each point of the volume must be a 4-vector of bytes in the alpha-red-green-blue format used in AVS images.

Use the **colorizer** or **gradient shade** modules to generate such color fields. You can use **tracer** with non-byte data, because **colorizer** and **gradient shade** take in a field containing any data (byte, integer, float, double), and output a 3D field of color values.

The ray tracing method is as follows. For each pixel in the output image a ray is "shot" into the volume. Each voxel the ray passes through makes some contribution to the color of the pixel. How much a voxel contributes depends on its opacity. The ray travels through the volume until the opacity of all the cubes it has passed through adds up to 1.0. This is an "additive light model", because the rays accumulate voxel color contributions as they travel through a volume.

For example, if a ray were to hit a completely opaque red voxel then it would travel no further, and the pixel associated with that ray would be colored red. On the other hand, if the voxel were nearly transparent, then it would confer only a fraction of its color to the pixel, and the ray would pass deeper into the volume, summing the color values of the other voxels it intersects.

Volumetric rendering such as this allows you to penetrate beneath the surface of 3D data, and see depths surrounded by "translucent" outer layers. The degree of opacity of the volume can be controlled by changing the alpha scale parameter, or by using **generate colormap**'s widget to edit the opacities associated with the data.

The method used by **tracer** is considerably faster than that used by **vbuffer**, and it avoids the image anomalies that **alpha blend** and **vbuffer** display when volumes are rotated. For a detailed description of the ray_tracing method used by tracer, see: Garrity, M., "Raytracing Irregular Volume Data," (Proceedings of the 1990 San Diego Workshop on Volume Visualization), *Computer Graphics*, Volume 24, Number 5, November 1990, pp. 35-40. ACM SIGGRAPH.

INPUTS

Data field (required; field 3D 4-vector byte)

The input data must be a 3D block of voxels. That is, the data at each point of the 3D volume field must be a 4-vector of bytes in the alpha-red-green-blue format used in AVS images. Typically, voxel data is produced by passing 3D field data through the module **colorizer.**

Upstream Transform (optional, autoconnect)

The lefthand port on the module **tracer,** can receive a 4x4 transformation matrix describing rotations and translations to apply to the volume data. This matrix (field 2D scalar float) can come from an appropriate downstream module, or from the modules **euler transformation** and **display tracker.** These mechanisms allow you to rotate the volume in 3-space.

For example, when the **tracer** module is connected to the **display tracker** module in a network, **display tracker** sends a transformation matrix back to this port on **tracer.** This allows you to directly manipulate the volume by moving the mouse in **display tracker's** window, using the "virtual spaceball" paradigm. For a more detailed description of direct manipulation see the section titled "Transforming Objects" in Chapter 5 of the *AVS User's Guide.*

PARAMETERS

alpha scale (float)

A floating point value between 0.0 and 1.0 which is multiplied by the alpha byte of every voxel in the volume. This determines how transparent the the volume will seem. The default of 1.0 results in all the voxels' alpha bytes remaining unchanged. As the value of alpha scale approaches 0.0 the volume becomes more transparent, allowing rays to penetrate deeper into the volume, and making inner regions visible.

perspective (float)

With perspective set to the default 0.0, the rays sent into the volume emanate from an "eye point" at infinity. This means that when a ray passes through the image plane it is orthogonal to that plane, resulting in a parallel projection (i.e. non-perspective) view of the volume. As the perspective value increases the point from which rays emmanate moves closer to the image plane, resulting in an increase in perspective. Selecting a high value for perspective may result in part of the volume moving outside the bounds of the image window.

width (integer)

Value which determines the width in pixels of the output image. Another way of thinking of this is the width determines the number of rays that will be projected into the volume along the x direction. This changes the shape of the window through which you view the volume. With perspective on, changing the width can bring clipped regions of the window back into view.

height (integer)

Value which determines the height in pixels of the output image. Another way of thinking of this is the height determines the number of rays that will be projected into the volume along the y direction. This changes the shape of the window through which you view the volume. With perspective selected, changing the height can bring clipped regions of the window back into view.

interpolate (toggle)

Allows you to choose between two ray-tracing algorithms:

Voxel approximation (default)

The 3D field is broken into cells, or voxels, as described above, i.e. the volume is decomposed into blocks, each with eight corners. Each voxel has a single opacity, color, and set of shading parameters. These values are taken from the vertex at the voxel's upper lefthand corner, and are assumed to be uniform throughout the voxel.

The length of a ray's path through a voxel is computed. Thus if a ray just nicks the corner of a green voxel, only a little green is added to the ray's accumulated color. This method is faster than the trilinear interpolation method. Use it to get a quick look at the data.

Trilinear Interpolation

In this algorithm it is not assumed that each voxel has a uniform color and opacity. Rather, the field values of the voxel's eight corners are interpolated. These interpolated values are then used to determine the actual opacity and color values of the points at which a ray enters and exits a voxel. As in the voxel approximation method, the length of the rays path through the voxel affects that voxel's contribution to the ray's color. This method produces a more accurate rendering of the volume.

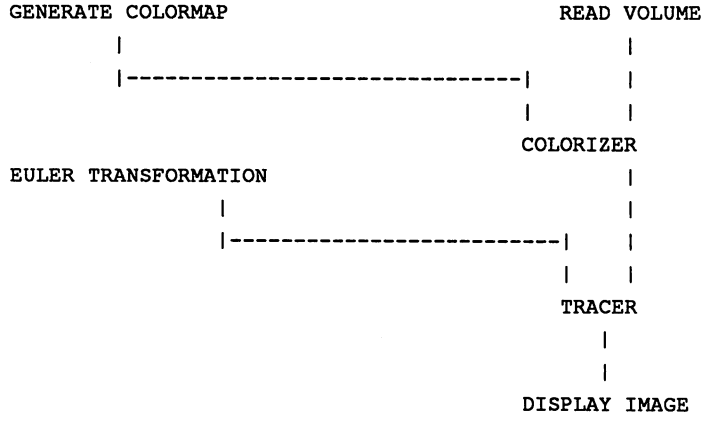
OUTPUTS

Data Field (field 2D 4-vector byte)

The output field is an AVS image.

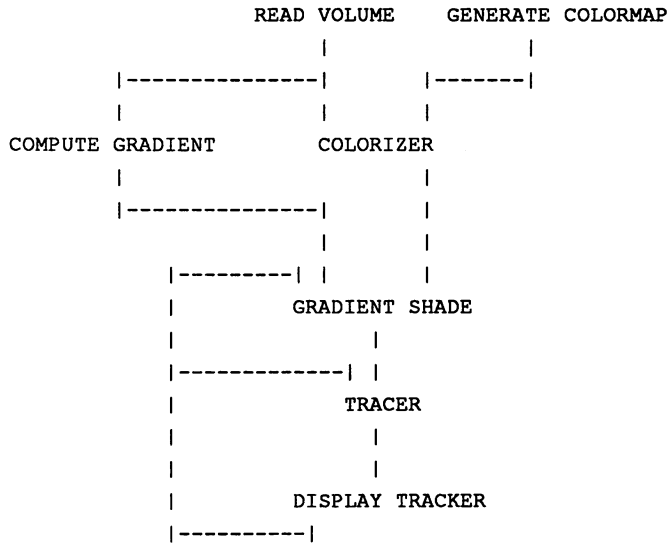
EXAMPLE 1

The following network reads in volume data, assigns color values to each element of the data and passes this field to **tracer**. The module **euler transformation** allows you to rotate the volume to produce views from any angle:



EXAMPLE 2

Another interesting technique is to apply a light source to the data. In order to do this the gradient of the data (which approximates the "surface normal") must be computed. Note that the **gradient shade** module has been modified to accept a transformation matrix. This prevents the light source from rotating relative to the data, when the object is rotated using **display tracker** or **euler transformation**. Without connecting **display tracker** (or **euler transformation**) to **gradient shade**, the light source would appear "attached" to any object transformations. A network for doing this gradient shading is:



Note that this network uses the module **display tracker**, which allows you to directly manipulate the volume being viewed by moving the mouse. **display tracker** feeds information on the mouse's movements back to **tracer** through its lefthand data port.

RELATED MODULES

Modules that could provide the **Data Field** input:

read field
colorizer
gradient shade
any other module which outputs a 3D field of 4-vector bytes.

Modules that can process **tracer**'s output:

display tracker
display image
image viewer
any other module which takes an AVS image as input.

SEE ALSO

Garrity, M., "Raytracing Irregular Volume Data," (Proceedings of the 1990 San Diego Workshop on Volume Visualization), *Computer Graphics*, Volume 24, Number 5, November 1990, pp. 35-40. ACM SIGGRAPH.

A DEMO script, tracer.scr, demonstrates the **tracer** module.

NAME

transform pixmap – perform 3D transformation on pixmap (hardware texture mapping systems only)

SUMMARY

Name	transform pixmap	
Type	data output	
Inputs	pixmap	
Outputs	pixmap	
Parameters	*Name*	*Type*
	image transform	4x4 matrix
	transform image	toggle
	reset	toggle
	refine	toggle

DESCRIPTION

The **transform pixmap** module maps its pixmap input onto a rectangle that has been arbitrarily transformed in three dimensions. The resulting pixmap is then output. The transformation allows for rotation, scaling, translation, or shearing of the image (or any combination thereof).

A benefit of using **transform pixmap** in a network is that it automatically scales its output pixmap size to fit the output window of a **display pixmap** module downstream. For example, if you read in a 512x512 pixmap, you can display the entire pixmap in any size window.

For **transform pixmap** to work, and for it to appear in the module palette, the system it is running on must support texture mapping in both graphics software and hardware. The Stardent ST1000 and ST2000 machines have this feature. Other systems (e.g., the Stardent ST1500 and ST3000 machines) may not.

INPUTS

pixmap The input can be any AVS *pixmap*.

PARAMETERS

image transform

Controls the 3D transform to be applied to the pixmap. The control widget is a window containing a colored cube, annotated with coordinate axis information. Transforming this cube with the following mouse buttons causes the pixmap to be transformed accordingly:

Mouse	Transform
left	cycle among three views: along X-axis, along Y-axis, along Z-axis
middle	rotate
right	translate in plane of screen
middle with SHIFT key	scale
right with SHIFT key	translate perpendicular to plane of screen

The mouse button mapping is the same as in the Geometry Viewer (or the **render geometry** module).

transform image (toggle)

This toggle parameter controls whether you can transform the image directly (i.e. in its window), or must use the **transformation** widget described above.

- **If ON:** The **transform pixmap** module "grabs" button press events in the associated output window, allowing you to transform the image directly.

  NOTE: For pixmaps generated by a **render geometry** module, button clicks in the window will no longer transform the geometry, but will transform the pixmap instead.

- **If OFF:** The mouse buttons have the same meanings, but you cannot "grab" the image in the output window directly. Instead, you must transform the cube in the transform control widget, which appears in the module's control panel.

refine (toggle)

Controls the use of point sampling to improve the quality of the output pixmap.

- **If ON,** A "successive refinement" algorithm is used to improve picture quality. When there is no other work left to do, **transform pixmap** applies nine refinement passes, each of which incrementally improves the picture. This is especially useful when small images are to be displayed in very large windows, or vice-versa.
- **If OFF,** the transformation applied to the image uses a "point sampling" algorithm.

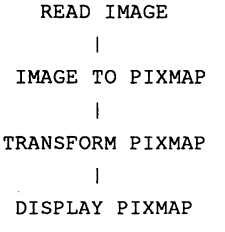
reset (one-shot)

Resets the transformation of the image to be the identity transformation.

OUTPUTS

pixmap The output is a pixmap containing a *scene* that includes all the input objects.

EXAMPLE



RELATED MODULES

image to pixmap, transform pixmap, display pixmap

LIMITATIONS

This module only works properly when running locally on Stardent ST1000, ST2000 systems.

When you transform an image directly (**transform image** toggle) or use the **Reset** function, the transform control widget is not updated.

SEE ALSO

The example script CONTOUR GEOMETRY demonstrates the **pixmap tp image** module.

NAME

transpose – exchange dimensions in a 2D or 3D data set

SUMMARY

Name	transpose			
Type	filter			
Inputs	field 2D/3D *n-vector any-data any-coordinates*			
Outputs	field of same type as input			
Parameters	*Name*	*Type*	*Default*	*Choices*
	axis	choice	Original	Original, YZ, XZ, XY

DESCRIPTION

The **transpose** module exchanges the data in two dimensions of a 2D or 3D field. It can be used to change the orientation of the data for display and/or processing purposes.

INPUTS

Data Field (required; field 2D/3D *n-vector any-data any-coordinates*)
The input data may be any 2D or 3D AVS field.

PARAMETERS

axis The choices for exchanging the data are:

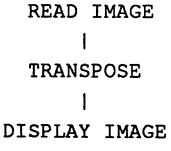
- **Original** Copies the input to the output; no transformation is performed.
- **YZ** Swaps the Y and Z dimensions. (Equivalent to "Original" for a 2D field.)
- **XZ** Swaps the X and Z dimensions. (Equivalent to "Original" for a 2D field.)
- **XY** Swaps the X and Y dimensions.

OUTPUTS

Data Field (field 2D/3D *n-vector any-data any-coordinates*)
The output field has the same dimensionality and type as the input field.

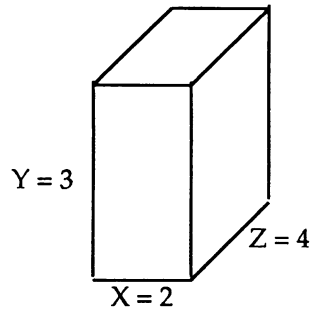
EXAMPLE 1

The following network reads in an image and then swaps the XY dimensions:

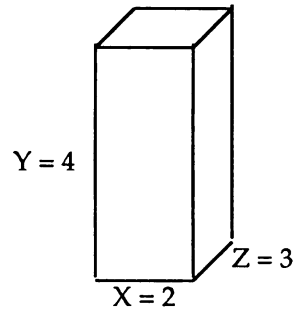


EXAMPLE 2

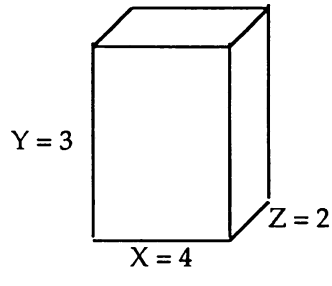
These drawings illustrate the transposition choices:



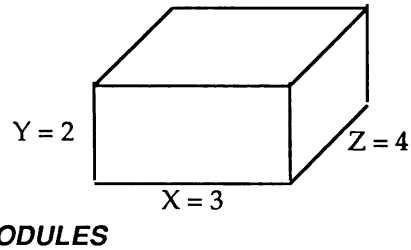
Original Data



After YZ Transpose



After XZ Transpose



After XY Transpose

RELATED MODULES

This module combined with **mirror** can re-orient the data in any desired way.

NAME

tristate - send a tristate value to one or more module(s) tristate parameter port(s)

SUMMARY

Name	tristate				
Type	data				
Inputs	none				
Outputs	tristate				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	tristate	tristate	0	0	2

DESCRIPTION

The **tristate** module sends a single user-specified tristate value to one or more tristate parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control tristate parameter input to more than one module using only a single tristate input widget.

The tristate data-type is a variant of the boolean data-type. A tristate variable has three possible values: 0, 1 or 2. It is used to make selections when there are only three possible choices.

Before you can connect **tristate** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **tristate** module icon in the usual way one connects modules.

PARAMETERS

tristate (integer)
: The single tristate value (0, 1, or 2), specified through a tristate widget, to be sent to the receiving module(s) tristate parameter port(s). The default value is zero.

OUTPUTS

tristate (integer)
: The tristate value (0, 1, or 2) is sent to all modules with tristate-type parameter ports that are connected to the **tristate** module.

RELATED MODULES

Modules that can process **tristate**'s output:

modules with tristate-type parameter ports

NAME

tube – convert lines to cylindrical tubes

SUMMARY

Name	tube				
Type	filter				
Inputs	geometry				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	radius	float	0.1	0.0	4.0

DESCRIPTION

The **tube** module transforms an AVS *geometry*, replacing a set of disjoint lines with "tubes" constructed out of eight polygons.

INPUTS

Geometry (required; geometry) An AVS geometry, created with the *libgeom* library or by another AVS module.

PARAMETERS

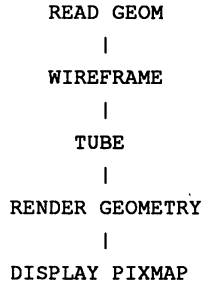
radius The radius to be used for the tube. Only values in the range 0.0 – 0.4 produce an acceptable result.

OUTPUTS

Geometry (geometry)
The output is an AVS *geometry*, representing each input line as a set of polygons.

EXAMPLE

In this example, the original geometry includes no disjoint lines. The **wireframe** module is used to add disjoint lines, which are then converted to tubes.



RELATED MODULES

read geom, offset, shrink, flip normal, wireframe, render geometry

LIMITATIONS

Only **radius** values in the range 0.0 – 0.4 produce acceptable results.

The cylinders are not capped and adjacent line segments are not joined. For thick cylinders, there may be quite a bit of surface intersections at the joins.

SEE ALSO

The example script TUBE demonstrates the **tube** module.

NAME

ucd anno – show data values of cells or nodes of a UCD structure

SUMMARY

Name ucd anno

Type mapper

Inputs ucd structure
upstream geometry (optional, invisible, autoconnect)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
cell data	choice	<data 1>		
label id	boolean	on		
label value	boolean	off		
cell nodes	boolean	off		
title	boolean	off		
Text Size	integer	2	1	5

DESCRIPTION

ucd anno makes it possible to see the values of specific cells and nodes of a UCD structure simply by clicking on the structure. The cell or node values of the cell that is clicked on are output as geometry labels, and can be viewed along with the UCD structure using the **render geometry** module. The **ucd anno** module, thus, provides a way to directly view data values contained in a UCD structure.

In a UCD structure, nodes and cells may have an arbitrary number of data components associated with them. **ucd anno** displays the values of one data component at a time, whether it is a scalar or a vector. Use the **node data** and **cell data** radio buttons to select which data component you want **ucd anno** to display. (see *LIMITATIONS* below).

ucd anno takes two inputs: a UCD structure, and an upstream geometry which it receives when it is in a network with **render geometry**. When you click the left mouse button on the image of the UCD structure the **render geometry** module sends information upstream telling **ucd anno** where on the structure the mouse was clicked. From this information **ucd anno** calculates which cell or node is being selected, and displays the data for that cell or node.

The labels that **ucd anno** outputs appear as geometry objects in 3-space attached to the nodes they are associated with. If the UCD structure is rotated the node and cell labels will rotate along with it. As they rotate they remain oriented parallel to **display pixmap**'s window. This may cause a label to intersect the volume of the UCD structure and be partly or wholly hidden by the structure. Rotating the structure further will usually bring the label above the structure's surface.

INPUTS

UCD Structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

upstream geometry (optional, invisible, autoconnect)
When the **ucd anno** module coexists with the **render geometry** module in a network, **render geometry** feeds information on where the mouse has been clicked back to this input port on the **ucd anno** module. The two modules connect automatically, through a data pathway that is normally invisible. This makes it possible to see the values of specific cells and nodes simply by clicking on them.

PARAMETERS

node data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

cell data Selects which of the cell's data components to display. A set of radio buttons shows shows the label attached to each cell data component. Before the module has received data, the default "<data 1>, <data 2>, ..." is displayed. If there is no cell based data in the structure "<no data>" is displayed on the button.

label id When **label id** is selected the integer or string that identifies a cell or node is displayed.

label value When **label value** is selected the floating point value associated with one data component of a cell or node is displayed.

cell nodes When **cell nodes** is selected, **ucd anno** displays the data for all the nodes of the cell that has been clicked on. Thus, for a hexadehron, **ucd anno** would display the node data at each of the cell's 8 nodes.

title When **title** is selected, if the UCD structure has a title, it is displayed in the top-left corner of **display pixmap**'s window.

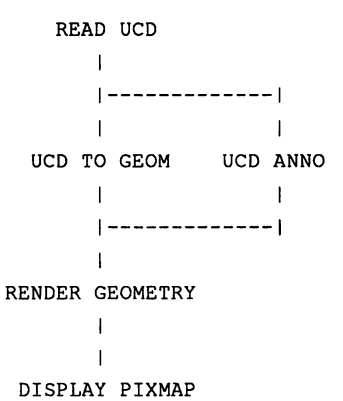
Text Size An integer dial that controls the font size of the output strings.

OUTPUTS

Geometry **ucd anno**'s outputs consist of the selected UCD structure values output as a geometry.

EXAMPLE

The following network reads in a UCD structure and annotates it. The selected values are displayed by **render geometry** along with the UCD structure itself:



RELATED MODULES

Modules that could provide the **UCD structure** input:

field to ucd
ucd crop
ucd threshold
ucd extract
ucd hex to tet
Any module that outputs a UCD structure.

Modules that can process **ucd anno**'s output:

render geometry

LIMITATIONS

ucd anno does not yet support the display of cell based data.

SEE ALSO

The example script UCD ANNO demonstrates the **ucd anno** module.

NAME

ucd contour – generate list of color values associated with unstructured cell data

SUMMARY

Name	ucd contour		
Type	mapper		
Inputs	ucd structure colormap		
Outputs	field 1D 3-vector real		
Parameters	*Name*	*Type*	*Default*
	node data	choice	<data 1>

DESCRIPTION

ucd contour is used to create a color contour of a UCD structure. Its output is passed to **ucd to geom** to produce a colored representation of a UCD structure. Essentially, **ucd contour** associates colors with the values at each node of a UCD structure.

Typically a UCD structure has a number of nodes. Each of these nodes may have an arbitrary number of data components associated with it. Furthermore each of these components itself can be a vector or a scalar.

ucd contour can only color the values of scalar node components. By using the **node data** radio buttons you can select a scalar data component for **ucd contour** to color. If a UCD structure has both scalar and vector components, only the scalar components will be displayed. The labels associated with the data components will be displayed on the radio buttons.

ucd contour takes each node value and colors it in proportion to the range of values in the structure using the formula:

$$\text{color\_index} = \frac{\text{node\_value} - \text{min\_node\_value}}{\text{max\_node\_value} - \text{min\_node\_value}} * \text{max\_colormap\_value}$$

The "color index" is an index into the input colormap, and is used to compute the 3-vector real value for a given color.

Thus **ucd contour** scales the colormap to the range of values of the node component that has been selected. In other words, the lowest node value present in the structure will get colored with the lowest colormap value, and the highest node value will get colored with the highest colormap value. Of course you may change the input colormap using **generate colormap**'s colormap widget. The **Color Field** output by **ucd contour** does not include the "alpha" or opacity information contained in an AVS colormap.

It should be noted that the **Color Field** output by **ucd contour** is not an AVS colormap.

INPUTS

UCD structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Colormap (required; colormap)
An AVS colormap. **ucd contour** maps node values in the input structure to colors in the colormap.

PARAMETERS

node data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is

displayed. If there is no node data in the structure "<no data>" is displayed on the button.

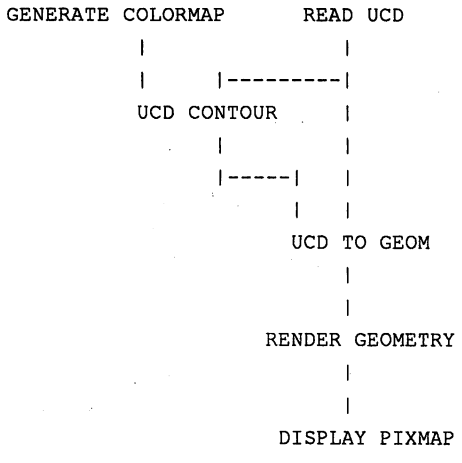
OUTPUTS

Color Field (field 1D 3-vector real)

The output field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green, and blue.

EXAMPLE

The following network reads in a UCD structure, colors each node based on the node's value, and displays the result:



RELATED MODULES

Modules that could provide the **UCD Structure** input:

read ucd
ucd crop
ucd threshold
Any module that outputs a UCD Structure.

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd contour**'s output:

ucd iso
ucd probe
ucd rslice
ucd to geom
ucd slice 2D

SEE ALSO

The example scripts UCD ISO, UCD PROBE, UCD EXTRACT, as well as others demonstrate the **ucd contour** module.

NAME

ucd crop – subset UCD structure data using slice plane or box

SUMMARY

Name	ucd crop		
Type	filter		
Inputs	ucd structure upstream transform (invisible, invisible, autoconnect)		
Outputs	ucd structure geometry		
Parameters	*Name*	*Type*	*Default*
	Crop tool	choice	plane
	Do Crop	boolean	off

DESCRIPTION

ucd crop allows you to cut away portions of a ucd structure leaving behind the cells you are interested in. You can use either a slice plane or a wireframe box as your tool for subsetting UCD structures. Two notes: First, before cropping a UCD structure, the subsetting tool must be moved from its default location. Second, to initiate the actual cropping operation, you must press the "Do Crop" button.

The slice plane is initially oriented in the xy plane. If you rotate the slice plane, you will see that one side has a highlighted area. The highlighted surface is on the side that will be cropped. In other words, any cells in the input structure which lie on the highlighted side of the slice plane will not appear in the structure output by **ucd crop**. If a cell has even one node lying on the outside of the slice plane, that cell will be cropped from the output. Similarly, when using the cubic "space" tool, any cells that are outside the bounds of the wireframe box are cropped from the output structure.

The **ucd crop** module is similar to the module, **ucd threshold**. **ucd crop**, however, eliminates nodes from a UCD structure based on their x, y, z coordinates — **ucd threshold** eliminates nodes based upon their values.

ucd crop outputs both the cropped ucd structure and a geometry which represents the subsetting tool currently selected. Typically, the **ucd to geom** module is used to convert the structure output by **ucd crop** to a geometry so it can be visualized using **render geometry**.

Since **ucd crop** outputs the slice plane and box subsetting tools as geometry objects, they can be sent directly to **render geometry**, and they can be manipulated directly using the mouse just like any other geometry objects; simply enter the Geometry Viewer and select the crop tool object as the current object. When **ucd crop** is linked in a network to **render geometry**, manipulating the subsetting tools with the mouse causes **render geometry** to send an upstream transform to **ucd crop**. This tells **ucd crop** how the slice plane or box tool has been reoriented relative to the input structure. Then **ucd crop** can recalculate what portions of the structure to cut away.

INPUTS

Structure (required)
: The input structure is in AVS unstructured cell data (UCD) format.

Upstream Transform (invisible, invisible, autoconnect)
: When the **ucd crop** module coexists with the **render geometry** module in a network, **render geometry** feeds information on how the "plane" or "space" subsetting object has been moved in the Geometry Viewer back to this input port on the **ucd crop** module. The two modules connect

automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **ucd crop**'s subsetting tools.

PARAMETERS

Plane A radio button that selects the slice plane as the subsetting tool.

Space A radio button that selects the wireframe box as the subsetting tool.

Do Crop A boolean switch that initiates the cropping function. This button allows you to manipulate the subsetting tool until you are satisfied with its position, and only then perform the cropping.

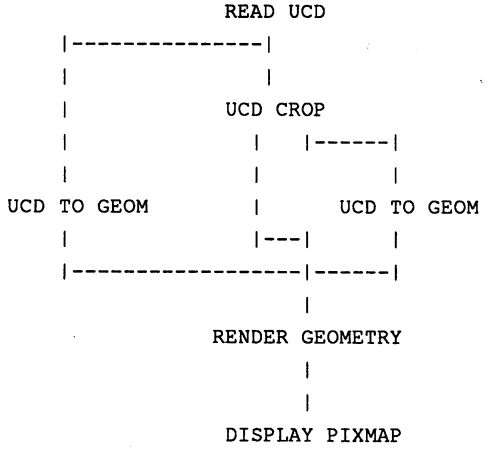
OUTPUTS

Structure The output structure is the cropped AVS unstructured cell data (UCD).

Geometry The geometry object that **ucd crop** outputs represents the subsetting tool currently selected, i.e. either the slice plane or the box tool.

EXAMPLE

The following network reads in a UCD structure and crops it. The **ucd crop** module outputs a geometry which gets passed directly to **render geometry**; it also outputs the cropped UCD structure from which a geometry is formed. Note that **read ucd** also sends the UCD structure to a second **ucd to geom** module. If you switch to the **geometry viewer** and specify a wireframe representation mode for the geometry output by this module, then the structure output by the second **ucd to geom** functions as a wireframe bounding volume around the structure.



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
ucd_extract
Any module that outputs a UCD structure.

Other modules that subset UCD structures:

ucd threshold
ucd rslice

Modules that can process the cropped UCD structure output:

ucd to geom
Any module that inputs a UCD structure.

Modules that can process the subsetting tool output:

render geometry

SEE ALSO

The example script UCD CROP demonstrates the **ucd crop** module.

NAME

ucd extract – extract single node component from a UCD structure

SUMMARY

Name	ucd extract		
Type	filter		
Inputs	ucd structure		
Outputs	structure		
Parameters	*Name*	*Type*	*Default*
	node data	choice	<data 1>
	Do Extract	boolean	off

DESCRIPTION

The **ucd extract** module takes a ucd structure that has several data components at each node and outputs a structure that has only one data component at each node. The output UCD structure is identical to the input structure, except for the extraction. Note, to initiate the extraction, you must press the "Do Extract" button.

Each node in a UCD structure may have an arbitrary number of data components associated with it. Furthermore each of these components itself can be a vector or a scalar. For example, a UCD structure may have 100 nodes. Each node consists of 3 components, labeled "temperature", "pressure", and "velocity". The first two components are scalar float values, but velocity is represented as a vector of three values.

ucd extract will extract any single component of the node data, whether that component is a vector or a scalar. If **ucd extract** takes a vector component, it extracts the entire vector of values. This means that **ucd extract** does not let you take a single element from a vector component.

INPUTS

UCD structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

node data Selects which of the node's data components to extract. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure, "<no data>" is displayed on the button.

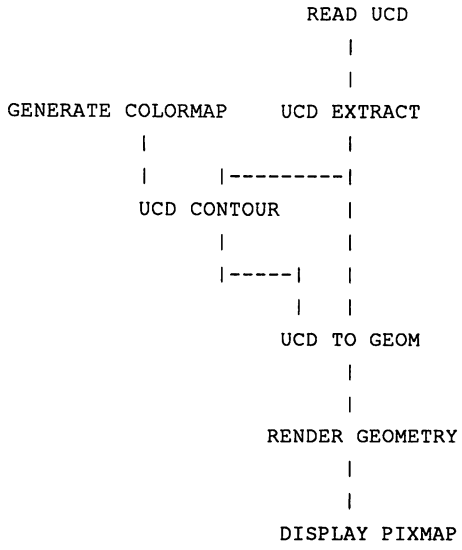
Do Extract A boolean switch that initiates the extraction once the node component has been selected.

OUTPUTS

UCD structure
The output structure is the same as the input structure, except that the node data is reduced to one component.

EXAMPLE

The following network reads in a UCD structure, extracts one component of the node data, and colors each node based on the value of that component:



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that can process **ucd extract**'s output:

ucd to geom, ucd crop, ucd threshold, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d, ucd legend, ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD EXTRACT demonstrates the **ucd extract** module.

NAME

ucd hex to tet– convert a UCD structure from hexahedral cells to tetrahedral cells

SUMMARY

Name	ucd hex to tet		
Type	filter		
Inputs	ucd structure		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	24 Tet	boolean	off
	Node Data	choice	<data 1>

DESCRIPTION

The module **ucd hex to tet** takes a UCD structure with hexahedral cells and converts it to a structure with tetrahedral cells.

To perform the conversion, **ucd hex to tet** must recompute the structure's node connectivity list. Hexahedral cells can be subdivided into 5 tetrahedra or into 24 tetrahedra. When data cannot be properly decomposed into 5 tetrahedra, it needs to be divided into 24 by adding a new node at the center of each face in the cell. These new nodes are added to the UCD structure, and data for them is computed by averaging the values at the corners of the face they are in.

ucd hex to tet is designed to work with the module **ucd tracer**, which performs ray-traced rendering on UCD structures. **ucd tracer** requires that its input structure contain tetrahedral cells.

INPUTS

Structure (required)
: The input is a UCD structure which has cells that are hexahedral.

PARAMETERS

24 Tet (boolean)
: When **24 Tet** is selected, hexahedral cells are decomposed into 24 tetrahedra, instead of the default, which is 5.

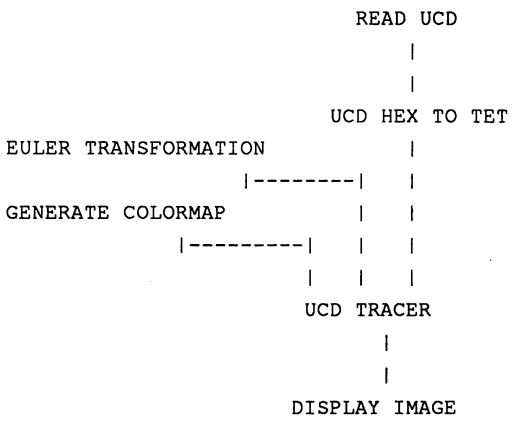
node data Selects which of the node's data components to use . A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button. When the **24 Tet** option is being used to subdivide hexahedral cells into 24 tetrahedra, the **node data** parameter determines which component is used to compute the data associated with the new nodes.

OUTPUTS

Structure The output is a UCD structure which has cells that are tetrahedral.

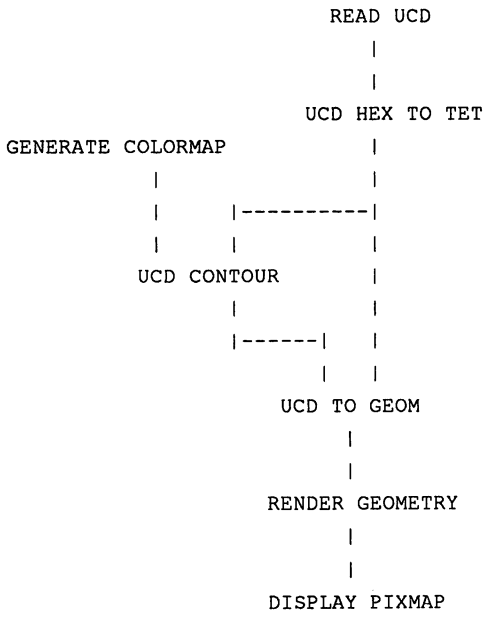
EXAMPLE 1

The following network reads in a UCD structure, which is converted from hexahedral cells to tetrahedal cells. This structure is then passed to **ucd tracer**. The module **euler transformation** allows you to rotate the volume to produce views from any angle:



EXAMPLE 2

The following network shows how **ucd hex to tet** can be used with modules other than **ucd tracer.**



RELATED MODULES

Modules that could provide the **ucd structure** input:

read ucd
any other module which outputs a hexahedral UCD structure.

Modules that can process **ucd hex to tet**'s output:

ucd tracer

NAME

ucd hog – show UCD node vector values as line segments in 3D space

SUMMARY

Name	ucd hog				
Type	mapper				
Inputs	ucd structure colormap upstream transform (optional, invisible, autoconnect)				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	node data	choice	<data 1>		
	scale	float	0.2	0.0	10.0
	arrows	boolean	off		
	N segments	integer	16	2	64
	Sample	choice	point		
	Vector Magnitude		boolean	off	

DESCRIPTION

ucd hog takes in a ucd structure whose node values include a 3-vector float component. **ucd hog** interprets the 3 values of the vector as the x, y, z components of a vector in space and then displays these 3D vectors as small line segments with a particular length, direction and color. "hog" is short for "hedgehog", a reference to the bristly appearance of the output geometry vectors.

ucd hog gives you a sample probe, which you can manipulate in the object space of the UCD structure. To move the sample probe, select it by clicking on it with the left mouse button. Alternately you can enter the Geometry Viewer, and make it the current object. Then the probe can be moved like any other geometry object. As it moves, **ucd hog** will recompute the line segments it outputs.

ucd hog only operates on vector components, thus it complains if the input structure has only scalar values at the nodes. If the nodes of a structure have more than one 3-vector component, use the **node data** radio buttons to select which component to use in calculating the hedgehog.

By default, **ucd hog** does not display the vector for every node in the structure. Instead **ucd hog** takes an arbitrarily-oriented (user-controlled) sample of locations within the bounds of the UCD structure. You can choose this sample to be:

- A single point
- A set of points on a line segment
- A set of points on a circle
- A set of points on a plane
- A volume of points

The module outputs the line segment(s) representing the node value at the sample location(s).

ucd hog uses the input colormap to associate a color with each line segment vector based on the magnitude of the vector. The colormap is scaled to the range of values in the structure.

Since arbitrarily oriented sample locations do not, in general, coincide with the position of the UCD structure's nodes in space, an interpolation method is used to determine which nodes are nearest to the sample locations.

INPUTS

UCD structure (required)
The input data must be a UCD structure. The structure must include a node data component which is a 3-vector of floats to be interpreted as vectors in 3-space.

colormap (required; colormap)
An AVS colormap which is used by **ucd hog** to associate colors with vector values. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd field legend**.

Upstream Transform (optional, invisible, autoconnect)
When the **ucd hog** module coexists with the **render geometry** module in a network, **render geometry** feeds information on how the point, circle or other sampling probe has been moved back to this input port on the **ucd hog** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **ucd hog**'s sampling probe.

PARAMETERS

node data Selects which of the node's data components to represent as vectors. A set of radio buttons shows shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

Vector Scale
The lengths of the line segments output by this module are proportional to this value.

arrows When **arrows** is selected the line segments are drawn with arrows at their heads, indicating their direction.

N segments
An integer value which determines the number of points sampled by the line, circle, plane, or space sampling probe. This controls the density of line segments output by **ucd hog**.

Sample (radio buttons) Specifies the type of sample taken from the vector field: point, line, circle, plane, or space.

Vector Magnitude
When **Vector Magnitude** is selected the magnitude of the vectors is not indicated by the length of their line-segment representation. Instead, the vectors are all the same length, and only their color indicates their magnitude.

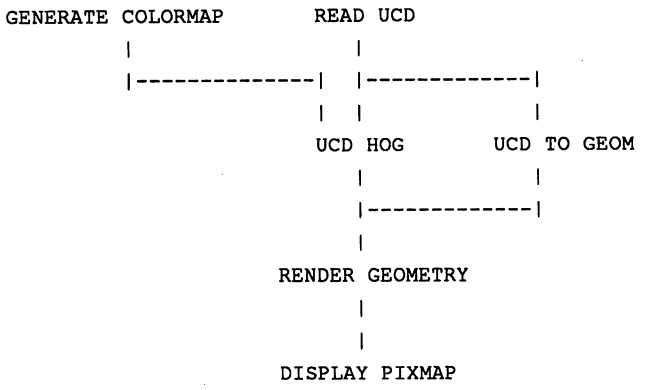
OUTPUTS

hog (geometry)
The output *geometry* is a collection of line segments representing the 3-vector component of nodes near the sample locations.

EXAMPLE

The following network reads in a UCD structure with a 3-vector float value as one of the components of the node data. **ucd hog** displays the values as line segment vectors. Note that the module **ucd to geom** is used to provide a frame within which to view the hedgehog of vectors. To do this, switch into the Geometry Viewer and change the rendering mode for the geometry output by **ucd to geom** to "wireframe". Also, edit the color properties for this object to make it dimmer and more transparent. This will improve your view of the line segments output by **ucd hog**. You

may want to similarly edit the properties of the sample probe, especially if it is a plane.



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that can process **ucd hog**'s output:

render geometry
Any module that inputs a geometry.

SEE ALSO

The example script UCD HOG demonstrates the **ucd hog** module.

NAME

ucd iso – generate an isosurface for a UCD structure with scalar node data

SUMMARY

Name ucd iso

Type mapper

Inputs ucd structure
colormap (optional)
info (from **ucd legend**; optional)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Node Data	choice	<data 1>		
Iso Level	float	1.0	1.0	9.0
map scalar	boolean	off		
Wireframe	boolean	off		

DESCRIPTION

The **ucd iso** module takes a UCD structure as input. The structure must have at least one component of its node data that is a scalar value. It produces a geometry object that represents an isosurface of this structure. An isosurface is a 3D generalization of a 2D contour line — it connects all structure elements that have the same value. You can use the **Node Data** buttons to select which component of the node data to use when computing the isosurface.

The **Iso Level** value can be set in two ways. The value can be set using **ucd iso**'s floating-point parameter dial. **ucd iso** also can accept an info input from the module **ucd legend**.

By default, the isosurface generated by **ucd iso** is not colored. To color the isosurface, **ucd iso** must receive its optional colormap input, and the **map scalar** parameter must be selected. If the input field has more than one scalar component of its node data, you can use the buttons beneath **map scalar** to select which component's values to use in determining the isosurface's color.

For example, if a structure's node data consisted of three scalars, temperature, pressure, and density, you might compute an isosurface for a given temperature throughout the structure. It would be intuitive to color this isosurface based on the temperature variable. However, it is also possible to color the temperature isosurface using the values of the pressure or density node data, thus indicating the pressure or density that hold for a fixed temperature.

Note: **ucd legend** outputs either a single float value or two float values representing a range. **ucd iso** can only use **ucd legend**'s single float output. Also, when **ucd iso** is connected to **ucd legend**, the selections of **ucd legend**'s node data buttons override **ucd iso** settings.

INPUTS

UCD structure (required)
The input data must be a UCD structure. The structure must include a scalar node data component.

colormap (optional)
An AVS colormap which is used by **ucd iso** to associate colors with the output isosurface. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd field legend**.

Info **ucd iso** can receive input from the module **ucd legend** through its left-most input port. This tells **ucd iso** what the value of the isosurface level should be.

PARAMETERS

node data Selects which of the node's scalar data components to use in constructing the isosurface. A set of radio buttons shows the label attached to each scalar node data component. Before the module receives any data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure, "<no data>" is displayed on the buttons.

Iso Level A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the UCD structure's data value equals the **Iso Level** value.

map scalar When the "map scalar" parameter is selected, and the optional colormap input is received, the isosurface that **ucd iso** outputs is colored using the values of the selected node component. By default it is off, and the isosurface is uncolored.

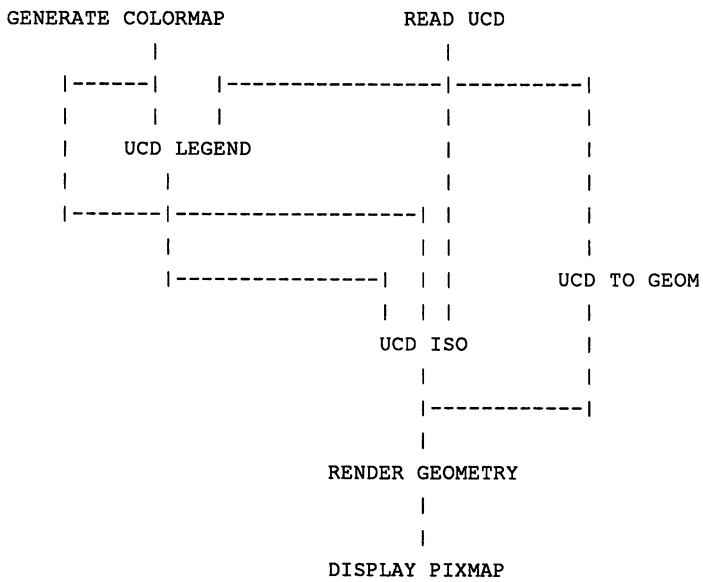
Wireframe If **Wireframe** is selected, the geometry output by **ucd iso** can be rendered as a wireframe object. To do this, switch into the Geometry Viewer and change the rendering mode to "wireframe".

OUTPUTS

Isosurface (geometry)
A shaded surface which represents the isosurface.

EXAMPLE

The following network reads in a UCD structure and generates an isosurface for some node value. The **generate colormap** module provides a colormap to color the isosurface, and the module **ucd legend** provides a widget for picking values for the **Iso Level** parameter. Note that the module **ucd to geom** is used to provide a frame within which to view the isosurface. To do this, switch into the Geometry Viewer and change the rendering mode for the geometry object produced by **ucd to geom** to "wireframe".



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Module that provides **Color Field** and **Info** inputs:

ucd legend

Modules that can process **ucd iso**'s output:

render geometry

SEE ALSO

The sample script UCD ISO demonstrates the **ucd iso** module.

NAME

ucd legend - creates a color legend relating UCD data to a color scale

SUMMARY

Name	ucd legend				
Type	mapper				
Inputs	ucd structure colormap				
Outputs	range (struct_ucd_legend) color field (field 1D 3-vector real)				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	node data	choice	<data 1>		
	range	boolean	off		
	value	float	0.0	0.0	100.0
	hi value	float	(not applicable)		
	lo value	float	(not applicable)		

DESCRIPTION

The **ucd legend** module performs two functions. First it is used to color unstructured cell data (UCD). To do this it takes in an AVS colormap, and outputs an array of colors — one for each node in the UCD structure.

Second, **ucd legend** creates a "color legend" widget relating UCD data to a color scale. This widget displays the input colormap as a horizontal spectrum. Beneath this color table **ucd legend** prints the range of the node values of the UCD structure. Like a "legend" on a map, the color legend shows you which color represents each value. This widget is used, like a floating-point dial, to pick specific values.

ucd legend works with modules that take UCD structures and allow you to visualize subsets of the data, or specific values in the data. Such modules include: **ucd iso**, and **ucd thresh**. Typically, using a dial, you specify a single value, or a range of values, say from 1.6 to 4.3. With **ucd legend** you can specify the subset by numerical value or, by color range, for example, as ranging from green to blue. Manipulating colored data using **ucd legend**'s color legend is often more intuitive than using a floating-point parameter widget.

By dragging a "radio tuner" dial along the color legend you select a specific value for **ucd legend** to output. If the **range** parameter is selected you can move two radio tuner dials along the color legend to select both minimum and maximum values for the range that **ucd legend** outputs. The middle mouse button controls the maximum dial; the left controls the minimum dial.

Typically a UCD structure has a number of nodes. Each of these nodes may have an arbitrary number of data components associated with it. Furthermore each of these components itself can be a vector or a scalar.

ucd legend only works with scalar node components. By using the **node data** radio buttons you can select a scalar data component for **ucd legend** to use in its color legend. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. When data has been input the labels associated with the node components are displayed on the buttons. If there is no node data in the structure "<no data>" is displayed on the button.

ucd legend prints the scale representing the range of values associated with the selected node component, e.g. temperature, in scientific notation. The input colormap is normalized to the range of values of the selected node component. The label associated with the selected scalar is printed as title of the color legend.

INPUTS

UCD structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Colormap (required; colormap)
An AVS colormap. **ucd legend** uses the colormap to associate colors with each node in the input UCD structure. The colormap is also used to generate the "color legend" widget.

PARAMETERS

node data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

range A boolean switch. If it is selected **ucd legend** outputs two values representing the minimum and maximum of a range. If it is off, **ucd legend** outputs a single floating point value. By default it is off.

value If the range parameter is selected, a single floating-point dial appears. This functions in an identical manner to the **ucd legend**'s color widget; you can use it to select specific output values. In particular you can use the dial to type in specific values, by opening the dial's Dial Editor.

lo value
hi value If the range parameter is not selected, two floating-point dials appear. Using them you can specify the minimum and maximum values of the range that **ucd legend** outputs. The values shown on these dials are scaled to the range of values present in the input structure.

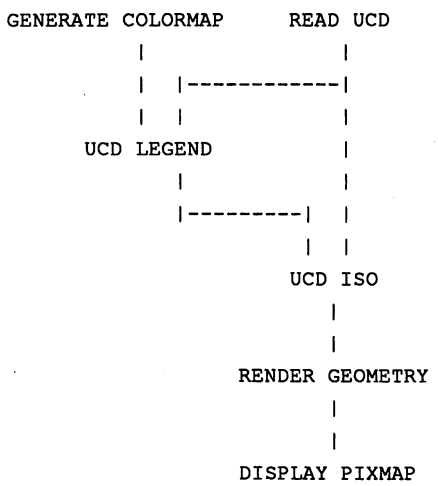
OUTPUTS

Selection (struct_ucd_legend)
ucd legend outputs either a single floating-point value, or two values representing the minimum and maximum of a range.

Color Field (field 1D 3-vector real; optional)
The color field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green and blue. Note that the **Color Field** is not the same as an AVS colormap. **ucd contour** can also be used to generate Color Fields.

EXAMPLE

The following network reads in a UCD structure. **ucd legend** generates a Color Field that associates a color with each node in the structure. The Color Field is then used to color an isosurface of the structure.



RELATED MODULES

Modules that could provide the **UCD Structure** input:

read ucd
field to ucd
any other module which outputs a UCD structure

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd legend**'s output:

ucd iso
ucd thresh

Modules that can produce Color Fields:

ucd contour

SEE ALSO

The example script UCD THRESHOLD, as well as others, demonstrates the **ucd legend** module.

NAME

ucd offset – deform a UCD structure based on vector values at each node

SUMMARY

Name	ucd offset				
Type	filter				
Inputs	ucd structure				
Outputs	ucd structure				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Do Offset	boolean	off		
	offset factor	float	0.25	0.0	1.0
	node data	choice	<data 1>		

DESCRIPTION

ucd offset "physically" deforms a ucd structure based upon the values at each of the structure's nodes.

The nodes of a UCD structure may contain several data components. Each of these components may itself be either a vector or a scalar value. **ucd offset** only operates on vector components, thus it complains if the input structure has only scalar values at the nodes. If the nodes of a structure have more than one 3-vector component, then use the **node data** radio buttons to select which component to use in calculating the deformation.

ucd offset takes the selected 3-vector component of each node and uses the three elements of that vector to translate the node in space. The first element of the vector translates the node's x coordinate, the second translates the y coordinate, and the third translates the z coordinate. The magnitude of each translation is proportional to the values at the nodes scaled by an **offset factor** between 0.0 and 1.0.

For example, if an unstructured cell dataset has a node component which is a 3-vector of values representing velocity in the x, y, z directions, **ucd offset** translates the x, y, and z location of each node proportional to the velocity values at that node.

INPUTS

Structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

Do Offset A boolean switch that initiates the deformation of the structure.

offset factor
A floating point value that is used to scale the magnitude of the deformation.

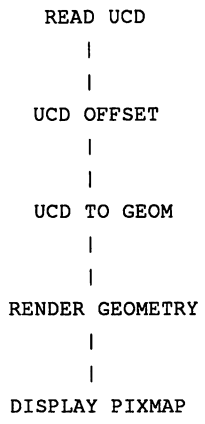
Node Data A set of radio buttons shows the label attached to any vector components of the node data. Before the module receives data, the default "<data 1>, <data 2>,..." is displayed. If there are no vector components of the node data **ucd offset** complains. If there are several vector data components, these buttons let you select which component to use in calculating the offset of the UCD structure.

OUTPUTS

Structure The output structure is the deformed UCD structure.

EXAMPLE

The following network reads in a UCD structure and deforms it, then displays the result:



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that can process **ucd offset**'s output:

ucd to geom, ucd crop, ucd threshold, ucd anno,
ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d,
ucd legend, ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD OFFSET demonstrates the **ucd offset** module.

NAME

ucd probe – interactively show numeric data values in a geometry rendered UCD structure

SUMMARY

Name ucd probe

Type mapper

Inputs
ucd structure
color field *(field 1D 3-vector float; optional)*
field irregular *(optional, from samplers module)*
upstream geometry *(optional, invisible, autoconnect)*

Outputs
geometry
upstream transform *(optional, invisible, autoconnect)*

Parameters

Name	*Type*	*Default*		
Probe Type	choice	Cursor		
node data	choice	<data 1>		
cell data	choice	<data 1>		
label id	boolean	on		
label value	boolean	off		
cell nodes	boolean	off		
Text Size	integer	2	1	7

DESCRIPTION

The **ucd probe** module displays the numeric data values associated with a specific cell in a UCD structure. It works for structures that have been rendered as AVS geometries. The **ucd probe** module lets you see the values in a UCD structure by simply pointing the mouse and clicking on the cell you are interested in.

ucd probe works by creating a cursor-like object titled "probe" that coexists in the Geometry Viewer window with the rendered version of the UCD structure. Its initial position is aligned with the first cell in the structure.

You deal with this probe object slightly differently than other objects in the Geometry Viewer. You cannot rotate or translate the probe using the middle or left mouse buttons. This is the case even if "probe" is selected as the current object in the Geometry Viewer.

To move the "probe" object through space, and have it display the values of UCD cells, press the left mouse-button. The probe object will "snap" to the UCD cell which is below the mouse cursor. This is a "point the mouse and click" technique of data sampling. Alternately, if you hold the left mouse-button down as you move the mouse, the probe "follows" the cursor around the display window, continuously reporting its position and the values of cells it passes over.

The Geometry Viewer tells the **ucd probe** module what UCD cell the mouse cursor was over when the button was pressed. **ucd probe** then reports the data values of the nodes which make up the vertices of the selected cell.

When reporting values for nodes with several data components, **ucd probe** lists the values of all the node components.

ucd probe outputs a geometry object representing the cell that has been selected. It is usually helpful to view the selected cell together with a wireframe rendering of the structure it belongs to. This can be achieved by adding the module **ucd to geom** to your network. **ucd to geom** outputs the entire UCD structure as a geometry object. By setting the rendering mode for this geometry to "lines", you can produce a wireframe representation of the structure. The example network, below,

demonstrates this.

INPUTS

UCD structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Color Field (field 1D 3-vector real; optional)
The color field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green and blue. The color field input is used by **ucd probe** to render the geometry object which represents the selected UCD cell. Two modules output the color field data type, **ucd contour** and **ucd legend**. Note that the color field is not the same as an AVS colormap.

Sample Field (optional; field irregular)
This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **ucd probe** takes these locations and displays the data values associated with them.

Upstream Geometry (optional, invisible, autoconnect)
Used by the **ucd probe**'s "point cursor and click" technique, this normally invisible port is what the **render geometry** module uses to inform **ucd probe** of the geometry vertex selected so it can display the data value for it. The module connection occurs automatically.

PARAMETERS

Probe Type
A set of radio buttons that control what the "probe" object looks like in the Geometry Viewer.

Cursor creates a probe that looks like a miniature XYZ axis.

Crosshair creates a probe that looks like half of a miniature XYZ axis. The crosshair stays aligned with the axis, and its endpoints lie in the XY, YZ, and XZ planes.

Probe creates a probe that looks like an electronic probe or a dissecting needle.

node data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

cell data Selects which of the cell's data components to display. A set of radio buttons shows shows the label attached to each cell data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no cell based data in the structure "<no data>" is displayed on the button.

label id When **label id** is selected the integer or string which identifies a cell or node is displayed.

label value When **label value** is selected the floating point value associated with one data component of a cell or node is displayed.

cell nodes When **cell nodes** is selected, **ucd probe** displays the data for all the nodes of the cell that has been clicked on. Thus, for a hexadehron, **ucd probe** displays the node data at each of the cell's 8 nodes.

Text Size An integer dial that controls the font size of the output strings.

OUTPUTS

Geometry (geometry)

The output geometry has three parts:

The rendering of the UCD cell that was selected,

The rendering of the "probe" object,

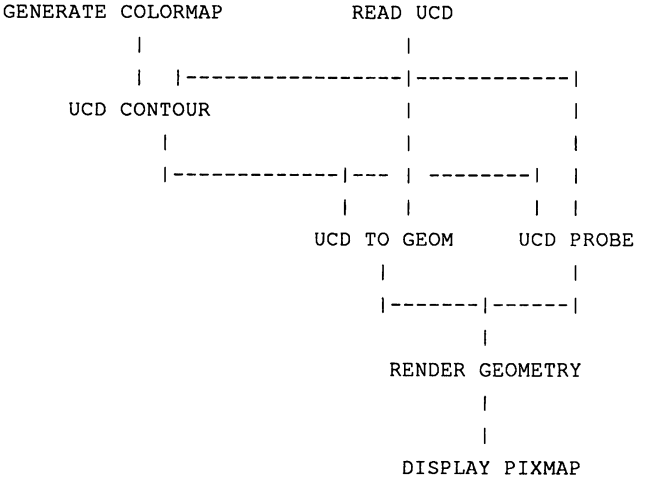
The rendering of the "Text for Probe" that lists the data values and coordinate position.

Upstream Transform (optional, invisible, autoconnect)

If **ucd probe** is connected to the **samplers** module, it uses this port to relay movement information from **render geometry** back up the network to **samplers**.

EXAMPLE

The following network reads in a UCD structure, which is passed both to **ucd to geom** and to **ucd probe**. The **ucd probe** outputs a geometry object representing the cell that has been selected. The **ucd to geom** outputs the entire UCD structure. By setting the representation mode for the entire structure to "lines", you can produce a rendering of the selected cell within a wireframe model of the structure:



RELATED MODULES

Modules that could provide the **Data Field** input:

read ucd
field to ucd

Modules that could provide the **Color Field** input:

ucd contour
ucd legend

Modules that could provide the **Sample** field input:

samplers

Modules that can process **probe** output:

render geometry

SEE ALSO

The example script UCD PROBE demonstrates the **ucd probe** module.

NAME

ucd rslice – slice away portions of a UCD structure

SUMMARY

Name	ucd rslice				
Type	mapper				
Inputs	ucd structure color field (field 1D 3-vector real)				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Do Slice	boolean	off		
	x-rot	float	0.0	0.0	360.0
	y-rot	float	0.0	0.0	360.0
	Distance	float	0.0	-2.0	2.0

DESCRIPTION

The **ucd rslice** module cuts through a UCD structure along an arbitrarily positioned slice plane. **ucd rslice** outputs the structure minus the portions that have been sliced away. The slice plane can be rotated around the x and y axes, and moved back and forth along the normal to the plane. Note that to initiate the slicing operation you must press the "Do Slice" button.

ucd rslice is similar to the modules **ucd crop** and **ucd threshold**, which also subset UCD structures. However, these two modules cut away the cells that make up the UCD structure; they do not cut *through* cells. **ucd rslice**, on the other hand, slices through any cells which the slice plane intersects. When you slice through hexahedral cells, for example, you may produce cells that do not look like hexahedrons. This is especially true if the UCD structure is being rendered as a wireframe.

By default, the UCD structure is placed at the origin and the slice plane is in the X-Z plane. The orientation of the slice plane is controlled by two floating point parameter dials, **x-rot** and **y-rot**. If you rotate the slice plane, you will see that one side has a highlighted area. The highlighted surface is on the side that will be removed.

Each time the slice plane is reoriented the boolean parameter **Do Slice** is turned off. This lets you adjust the slice plane until it is where you want, and only then perform the slicing operation. The slice plane can be moved back and forth through the UCD structure along the normal to the plane, using the **Distance** floating-point dial. This lets you take a series of parallel slices through a UCD structure in any direction.

INPUTS

Structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Color Field (field 1D 3-vector real)
This input field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green, and blue. Note that the color field is not a regular AVS colormap. Two modules output color fields: **ucd contour** and **ucd legend**.

PARAMETERS

Do Slice A boolean switch that initiates the slicing operation. This button allows you to manipulate the slice plane until you are satisfied with its position, and only then slice the UCD structure.

x-rot A floating point value which rotates the slice plane around the UCD structure's x axis.

y-rot A floating point value which rotates the slice plane around the UCD structure's y axis.

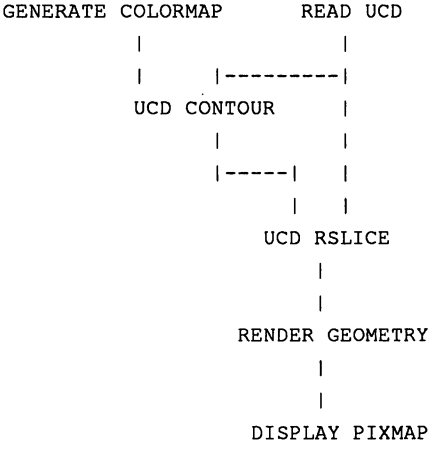
Distance A floating point value between -2.0 and 2.0 which moves the slice plane back and forth in the direction of the normal to the plane. This value is scaled by the largest dimension of the UCD structure. Consequently, you can move the slice plane along the normal from -(2 * max dimension) to (2 * max dimension).

OUTPUTS

Geometry **ucd rslice** outputs a geometry which includes the slice plane, and the portion of the UCD structure remaining after the slice operation is performed.

EXAMPLE

The following network reads in a UCD structure and slices it. The **ucd rslice** module outputs the sliced structure and the slice plane as geometries, which are rendered using **render geometry**:



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
ucd_extract
Any module that outputs a UCD structure.

Other modules that subset UCD structures:

ucd threshold
ucd crop

Modules that can process **ucd rslice**'s output:

render geometry

SEE ALSO

The example script UCD RSLICE demonstrates the **ucd rslice** module.

NAME

ucd slice 2D – extract 2D slice from a UCD structure

SUMMARY

Name	ucd slice 2D				
Type	mapper				
Inputs	ucd structure Color Field (field 1D 3-vector real)				
Outputs	geometry geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Do Slice	boolean	off		
	x-rot	float	0.0	0.0	360.0
	y-rot	float	0.0	0.0	360.0
	Distance	float	0.0	-1.0	1.0
	Transform Slice	boolean	on		

DESCRIPTION

The **ucd slice 2D** module extracts a 2D colored slice from a UCD structure. The slice plane can be rotated around the x and y axes, and moved back and forth along the normal to the plane.

By default, the UCD structure is placed at the origin and the slice plane is in the X-Z plane. The orientation of the slice plane is controlled by two floating point parameter dials, **x-rot** and **y-rot.**

Each time the slice plane is reoriented the boolean parameter **Do Slice** is turned off. This lets you adjust the slice plane until it is where you want, and only then perform the slicing operation. Once the slice plane is oriented as desired, and **Do Slice** is selected, the slice plane can be moved back and forth through the UCD structure along the normal to the plane. **Do Slice** remains "on" as you take successive slices along the normal. This lets you rapidly take a series of parallel slices through a UCD structure in any direction.

ucd slice 2D outputs two geometry objects. One is the slice plane, the other is the 2D slice of the UCD structure.

There are two different ways to use **ucd slice 2D**. In the first way, as shown in the first example network, **ucd slice 2D** sends both the slice plane and the 2D slice to **render geometry** via its righthand output port. **ucd to geom** is used to produce a wireframe representation of the UCD structure. To view the slice plane, the 2D slice, and the wireframe model all together, **ucd slice 2D**'s "transform slice" parameter must be turned **off**. In this configuration, when you move the slice plane, the 2D slice will move with it. Note that in this setup, **ucd slice 2D**'s lefthand output port is not connected to anything. If this port is connected to **render geometry** the results will be unpredictable.

In the second configuration, two **render geometry** modules are used. The first one receives the output of **ucd to geom**. To make **render geometry** display this as a wireframe model, switch to the **geometry viewer** and specify a wireframe representation mode for the structure. **ucd slice 2D** sends the slice plane geometry object to this first module through its righthand output port. This lets you orient the slice plane relative to the whole structure. **ucd slice 2D** then sends the 2D slice output to the second **render geometry** module via its lefthand output port. For the 2D slice to appear in the second **display pixmap** window, **ucd slice 2D**'s "transform slice" parameter must be turned **on**. The 2D slice will be displayed parallel to the viewplane, i.e. viewed straight-on. When you move the slice plane in the first

display pixmap window the 2D slice will change in the second window.

INPUTS

Structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Color Field (field 1D 3-vector real)
This input field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green, and blue. Note that the **Color Field** is not a regular AVS colormap.

PARAMETERS

Do Slice (boolean)
A boolean switch that initiates the slicing operation. This button allows you to manipulate the slice plane until you are satisfied with its position, and only then extract the slice.

x-rot A floating point value which rotates the slice plane around the UCD structure's x axis.

y-rot A floating point value which rotates the slice plane around the UCD structure's y axis.

Distance A floating point value between -1.0 and 1.0 which moves the slice plane back and forth in the direction of the normal to the plane. This value is scaled by the largest dimension of the UCD structure. Consequently, you can move the slice plane along the normal from - **max dimension** to + **max dimension.**

Transform Slice (boolean)
When this is selected the 2D slice of the UCD structure is transformed so that it is parallel to the viewing plane. This must be turned **off** when **ucd slice 2D** is sending both its output geometries to a single **render geometry** module. It must be turned **on** when **ucd slice 2D** is sending its slice plane output to one **render geometry** module and its 2D slice output to another **render geometry** module.

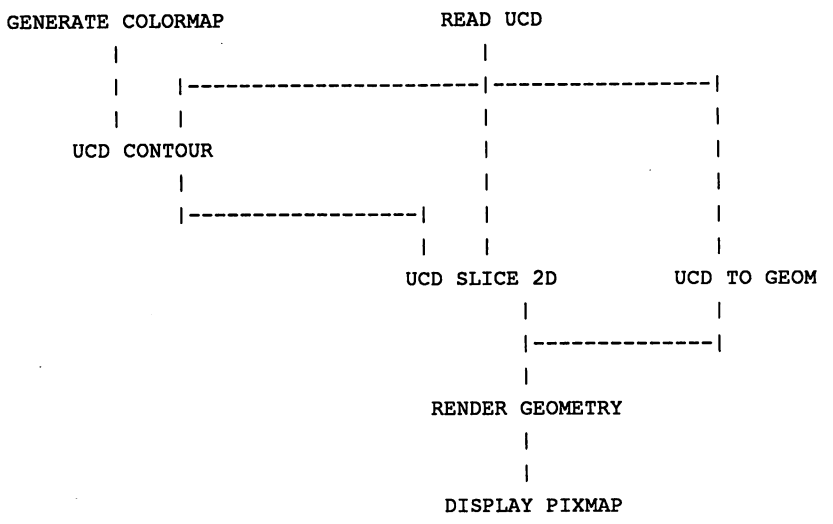
OUTPUTS

Geometry The geometry object that **ucd slice 2D** outputs from the left output port represents the 2D slice of the UCD structure.

Geometry The geometry object that **ucd slice 2D** outputs from the right output port represents the slice plane.

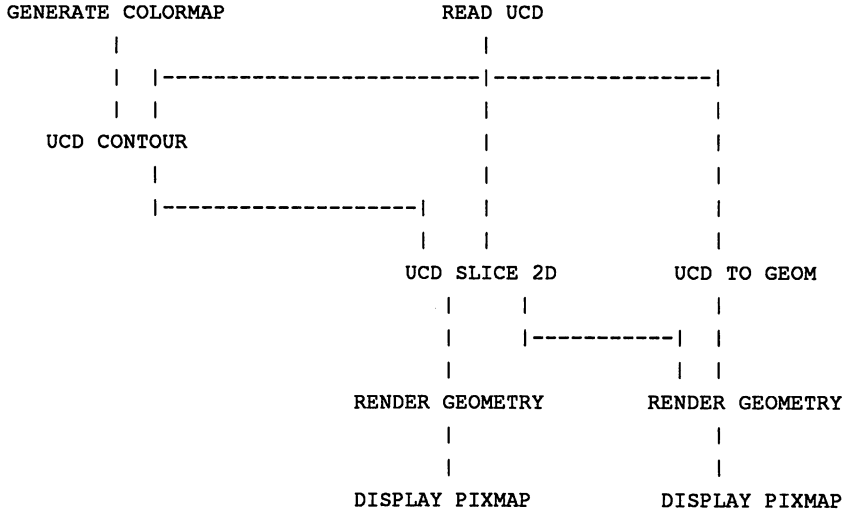
EXAMPLE 1

In the following network **ucd slice 2D** sends both the slice plane output and the 2D slice output to a single **render geometry** module. This module also receives a wireframe model of the entire UCD structure from the module **ucd to geom**. These three geometries are all rendered together. Note that for the 2D slice to be correctly oriented, the **Transform Slice** parameter must be off.



EXAMPLE 2

In the following network **ucd slice 2D** outputs both the 2D slice of the UCD structure and the slice plane. The 2D slice is viewed alone using the lefthand **render geometry** module. The 2D slice is transformed so that it is parallel to the view plane. This is done by selecting **ucd slice 2D's Transform Slice** parameter. The slice plane itself is sent to the righthand **render geometry** module, where it is rendered along with a wireframe representation of the structure as a whole. The wireframe model provides a framework within which you can move the slice plane. To produce the wireframe model, switch to the **geometry viewer** and specify a wireframe representation mode for the structure.



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that could provide the **Color Field** input:

ucd contour
ucd legend

Modules that can process **ucd slice2D's** output:

render geometry
Any module that inputs a geometry

SEE ALSO

The example script UCD SLICE 2D demonstrates the **ucd slice2D** module.

NAME

ucd streamline – generate stream lines for a UCD structure with vector node data

SUMMARY

Name ucd streamline

Type mapper

Inputs ucd structure
colormap
upstream transform *(optional, invisible, autoconnect)*

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
Probe Res	integer	16	2	64
sample style	choice point			
N segment	integer	2	2	10
Integ Method	choice	1st order		
Backwards	boolean	off		
Color Streams	boolean	off		
Start Streams	boolean off			

DESCRIPTION

The **ucd streamline** module generates colored streamlines based on the vector node data in a UCD structure. It places a "sample" of points at a starting location in the UCD structure. The number of points is parameter-controlled. The orientation of the points is controlled by the "sample probe", which can be moved around in space like any other geometry object, using the "virtual trackball" paradigm.

To initiate the calculation of stream lines, you must press the **Start Streams** button. To move the probe, select it by clicking on it, or by entering the Geometry Viewer and making it the current object. After each run through, the probe will be hidden from view. Moving it, or changing a parameter will make it visible again.

A UCD structure consists of cells with nodes at their vertices. Each node may have data associated with it. **ucd streamline** only works with structures that have a vector component in their node data, thus it complains if the input structure has only scalar values at the nodes. If the nodes of a structure have more than one 3-vector component, use the **node data** radio buttons to select which component to use in calculating the stream lines.

The vectors at each node can be viewed as exerting force on the sample points. For every time step, **ucd streamline** advances each sample point through space, based on the interpolated value of the node vectors surrounding the point. The result is a set of stream lines showing the progress of massless particles moving under the influence of the vectors at the nodes of the UCD structure.

Note that for streamlines to be calculated, the input UCD structure must have had its node connectivity list computed. Both **read ucd** and **field to ucd** have "cell connect" parameters to perform this operation. It is useful to turn these parameters on before reading in a new UCD structure. Otherwise, **ucd streamline** complains when it receives the structure.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format. It must have at least one node component which is a 3D vector, representing the components of a velocity vector.

colormap (required; colormap)
An AVS colormap that is used by **ucd streamline** to associate colors with vector values. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd legend**.

Upstream Transform (invisible, optional, autoconnect)
When the **ucd streamline** module coexists with the **render geometry** module in a network, **render geometry** feeds information on how the point, circle or other "sample probe" has been moved back to this input port on the **ucd streamline** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **ucd streamline**'s sample probe.

PARAMETERS

Node Data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there are no vector components of the node data **ucd streamline** complains. If there are several vector data components, these buttons let you select which component to use in calculating the stream lines. If there is no node data in the structure "<no data>" is displayed on the button.

Probe Res Integer dial that controls the density of points in the sample set.

Sample Style (radio buttons)
Specifies the configuration of points from which stream lines are drawn: point, line, circle, plane, or space.

N Segment Integer dial that specifies the number of increments for which stream lines are computed within each cell of the UCD structure. As the number of segments increases, so does the accuracy of the stream lines.

Integration Method
Selects the integration method used to advance sample points through space: **1st order** uses an euler integration method, **2nd order** uses a 2nd order Runge-Kutta method, and **3rd order** uses a 3rd order Runge-Kutta method.

Backward (boolean)
If **Backward** is selected, stream lines are extrapolated in the opposite direction that the UCD structure's vectors are pointing. By default this switch is off.

Color Stream (boolean)
If **Color Stream** is selected, the stream lines are colored based on the magnitude of the interpolated vectors used to generate the stream lines. By default this switch is off.

Start Stream (boolean)
A boolean switch that initiates the calculation of stream lines . This button allows you to manipulate the sample probe until you are satisfied with its position, and only then begin computing stream lines.

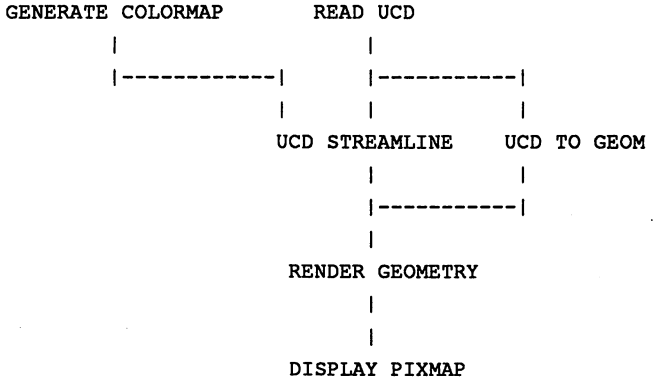
OUTPUTS

Stream Lines (geometry)
A set of colored disjoint lines.

EXAMPLE

The following network reads in a UCD structure with a 3-vector float value as one of the components of the node data. **ucd streamline** displays colored stream lines. Note

that the module **ucd to geom** is used to provide a frame within which to view the streamlines. To do this, switch into the Geometry Viewer and change the rendering mode for the geometry object produced by **ucd to geom** to "wireframe".



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd streamline**'s output:

render geometry

SEE ALSO

The example script UCD STREAMLINE demonstrates the **ucd streamline** module.

NAME

ucd threshold – get subset of UCD structure based on node values

SUMMARY

Name	ucd threshold		
Type	filter		
Inputs	ucd structure info (required; from **ucd legend**)		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	below	boolean	off
	inclusive	boolean	off

DESCRIPTION

ucd threshold takes a subset of the cells in a ucd structure based on the values at cell nodes. Input structure cells with nodes values that fall outside a user specified range do not appear in the structure which **ucd threshold** outputs.

The input received from **ucd legend** tells **ucd threshold** what range to restrict values to. This information can either be a single floating point number representing the cut-off value, or it can be two floating point numbers representing both a high and a low threshold.

The **ucd threshold** module is similar to the module, **ucd crop**. **ucd crop**, however, eliminates nodes from a UCD structure based on their x, y, z coordinates — **ucd threshold** eliminates nodes based upon their values.

INPUTS

Structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Info (from **ucd legend**)
ucd threshold must receive input from the module **ucd field legend** through its left input port. This tells **ucd threshold** what range to restrict values to.

PARAMETERS

below A boolean switch, which has meaning only when the **info** input is a single floating-point value. If it is selected, **ucd threshold** outputs the subset of the UCD structure that is below the threshold value. If it is not selected **ucd threshold** outputs the subset of the UCD structure that is above the threshold value.

inclusive A boolean switch; if it is selected, then **all** the nodes of a given cell must satisfy the threshold condition for that cell to be passed to the output. In other words, if a cell has even one node whose value falls outside the threshold range, that cell is eliminated from the output. If the **inclusive** switch is turned off, only one node of a given cell needs to satisfy the threshold condition for the cell to be included in the output structure.

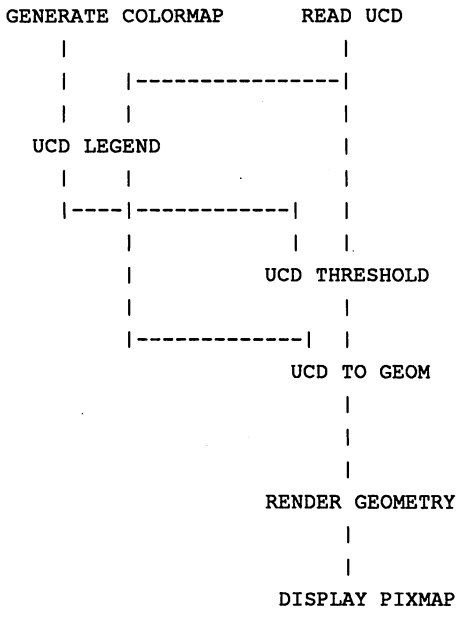
OUTPUTS

Structure The output structure is the threshold filtered AVS unstructured cell data (UCD).

EXAMPLE

The following network reads in a UCD structure. The structure is passed to the **ucd field legend** module, which outputs a threshold value or range. It is also passed to the **ucd threshold** module, which restricts the structure's values to the threshold

range. Note that **ucd legend** also outputs a color field that gets passed to **ucd to geom** so that the data is colored.



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that provides the **info** input:

ucd legend

Modules that can process **ucd threshold**'s output:

ucd to geom, ucd crop, ucd anno,
ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d,
ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD THRESHOLD demonstrates the **ucd threshold** module.

NAME

ucd to geom – convert a UCD structure into an AVS geometry

SUMMARY

Name	ucd to geom				
Type	mapper				
Inputs	ucd structure color field (field 1D 3-vector real)				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Shrink	boolean	off		
	shrink factor	integer	10	0	100
	Remove In. Face		boolean	off	

DESCRIPTION

ucd to geom converts a ucd structure into an AVS geometry that can be rendered using the **render geometry** module.

At the lowest level, unstructured cell data consists of nodes located in 3-space. These nodes may have a vector of values associated with them. Nodes form the vertices of polyhedral cells, which themselves may have cell based data associated with them.

ucd to geom takes the input structure's node location data, as well as a node connectivity list telling which nodes connect to form which cells. Each cell thus defined is converted into geometry format and is added to the geometry object that the module outputs.

A UCD structure may have hundreds of nodes and cells, many of which are likely to be "interior" and thus hidden. You can use the **Remove Interior Faces** parameter to restrict **ucd to geom**'s output to the "exterior", visible faces of the UCD structure's cells. This makes converting the structure to a geometry and rendering it much faster.

The cells can be shrunk using the **shrink factor** parameter. If the cells in a structure are packed close together, this creates gaps between cells and lets you see how cells are really shaped.

ucd to geom can receive an optional color field through its left input port. The color field is an array of color values - one color for each node in the input UCD structure. This results in the structure being rendered as a colored geometry object. The color field can be generated by the modules **ucd contour** or **ucd legend**.

INPUTS

Structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Color Field (field 1D 3-vector real; optional)
The color field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green and blue. The **Color Field** input is produced by the modules **ucd contour** and **ucd legend**. Note that it is not the same as an AVS colormap.

PARAMETERS

Shrink (boolean)
When this is selected each cell in the UCD structure is shrunk by the factor specified by the **shrink factor** parameter. By default **Shrink** is off.

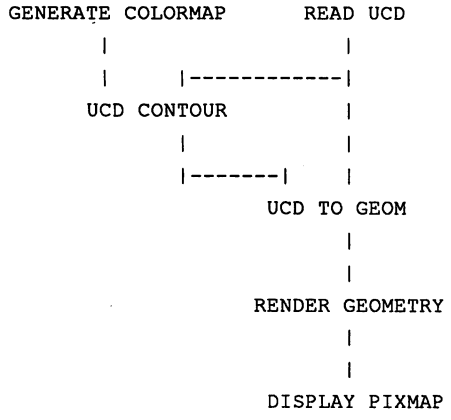
shrink factor
An integer is used to scale the cells of the UCD structure. Values of this parameter range from 1 to 100, representing percentages. The default shrink factor of 10 results in cells that are shrunk by 10 percent.

Remove Interior Faces (boolean)
When **Remove Interior Faces** is selected **ucd to geom** only adds "exterior", visible cell faces to the output geometry. This makes converting to a geometry and rendering much faster.

OUTPUTS

Geometry The geometry that **ucd to geom** outputs represents the cells of the input UCD structure colored according to the values of the input color field.

EXAMPLE



RELATED MODULES

Modules that could provide the **UCD Structure** input:

read ucd
field to ucd
ucd extract
Any module that outputs a UCD Structure.

Modules that could provide the **Color Field** input:

ucd contour
ucd legend

Modules that can process **ucd to geom**'s output:

render geometry
Any module that takes an AVS geometry.

SEE ALSO

The example scripts READ UCD, UCD THRESHOLD, UCD CROP, as well as others, demonstrate the **ucd to geom** module.

NAME

ucd tracer - perform ray-traced volumetric rendering on a UCD structure

SUMMARY

Name ucd tracer

Type data output

Inputs ucd structure
tracker info (field 2D scalar float)
colormap (required)

Outputs field 2D 4-vector byte (image)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Size	integer	128	0	1024
alpha scale	float	1.0	0.0	10.0
exterior	boolean	on		

DESCRIPTION

ucd tracer belongs to a family of modules that render volumetric data. **ucd tracer** takes a UCD structure, consisting of tetrahedral cells, and generates a 2D image using ray-tracing. Each cell in the structure has data values associated with its nodes. These values are used to assign a color and opacity value to every node in the structure. Note that, by default, **ucd tracer** "exterior" parameter is **on**, and therefore only an object's surface is ray-traced.

The ray tracing method is as follows. For each pixel in the output image a ray is "shot" into the UCD structure. Each cell the ray passes through makes some contribution to the color of the pixel. The color is calculated by interpolating between the color of the point at which the ray enters the cell and the color of the exit point. How much color a cell contributes depends on its opacity. The ray travels through the volume until the opacity of all the cells it has passed through adds up to 1.0. This is an "additive light model", because the rays accumulate cell color contributions as they travel through a volume.

For example, if a ray hits a completely opaque red tetrahedron then it travels no further, and the pixel associated with that ray is colored red. On the other hand, if the tetrahedron is nearly transparent, then it confers only a fraction of its color to the pixel, and the ray passes deeper into the volume, summing the color values of the other cells it intersects.

Volumetric rendering such as this allows you to penetrate beneath the surface of 3D unstructured cell data, and see depths surrounded by "translucent" outer layers. The degree of opacity of the volume can be controlled by changing the alpha scale parameter, or by using **generate colormap**'s widget to edit the opacity values in the input colormap.

ucd tracer only works with UCD structures that have tetrahedral cells. You can convert hexahedral data to tetrahedral using the module **ucd hex to tet**.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format. The structure's cells must be tetrahedrons.

tracker info *(field 2D scalar float)*

The middle input port on the module **ucd tracer** can receive a 4x4 transformation matrix describing rotations and translations to apply to the UCD structure. The matrix (field 2D scalar float) can come from the module **euler transformation** or **display tracker**. This allows you to

rotate the structure in 3-space.

colormap (required; colormap)
An AVS colormap which is used by **ucd tracer** to associate colors with UCD node values. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd legend**.

PARAMETERS

Size (integer)
Value which determines the height and width of the output image measured in pixels. Another way of thinking of this is that the width determines the number of rays that are projected into the volume along the x and y directions. This changes the size of the square window through which you view the volume,.

alpha scale (float)
A floating point value between 0.0 and 10.0 which is multiplied by the alpha value of every node in the structure. This determines how transparent the the structure will seem. As the value of alpha scale approaches 0.0 the volume becomes more transparent, allowing rays to penetrate deeper into the volume, and making inner regions visible.

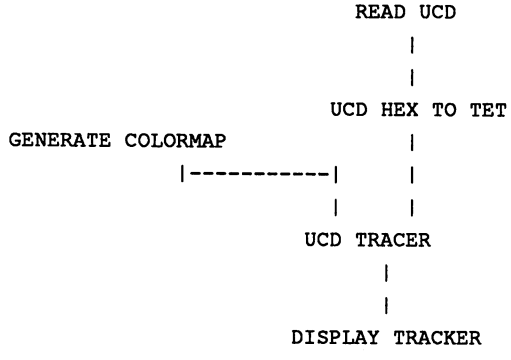
exterior (boolean)
If **exterior** is selected, then only the surface of the UCD structure is ray-traced. Note that this is the default.

OUTPUTS

Field (field 2D 4-vector byte)
The output field is an AVS image.

EXAMPLE

The following network reads in a UCD structure, which is converted from hexahedral cells to tetrahedal cells. This structure is then passed to **ucd tracer**. The module **display tracker** allows you to rotate the volume to produce views from any angle. Objects are manipulated using the usual mouse buttons.



RELATED MODULES

Modules that could provide the **ucd structure** input:

read ucd
ucd hex to tet
any other module which outputs a tetrahedral UCD structure.

Modules that can process **ucd tracer**'s output:

display tracker
display image
image viewer
any other module which takes an AVS image as input.

SEE ALSO

Garrity, M., "Raytracing Irregular Volume Data," (Proceedings of the 1990 San Diego Workshop on Volume Visualization), *Computer Graphics*, Volume 24, Number 5, November 1990, pp. 35-40. ACM SIGGRAPH.

The example script UCD TRACER demonstrates the **ucd tracer** module.

NAME

vbuffer – perform volumetric rendering on volume data

SUMMARY

Name vbuffer

Unsupported this module is in the unsupported library

Type data output

Inputs field 3D scalar any-data uniform
colormap

Outputs pixmap

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Xrotation	real	0.0	-180	180
Yrotation	real	0.0	-180	180
Zrotation	real	0.0	-180	180
ScaleX	real	0.5	0.0	5.0
ScaleY	real	0.5	0.0	5.0
ScaleZ	real	0.5	0.0	5.0
Ztrans	real	-2.0	-10	10
Imagesize	int	200	10	1024
Background	real	0.0	0.0	1.0
Cell-Based	logical	off	off	on
Re-use Scripts	logical	on	off	on
Script File	string	/tmp/vbuf.inp		
Scalar File	string	/tmp/vbuf.dat		
Image File	string	/tmp/vbuf.image		

DESCRIPTION

The **vbuffer** module creates a volumetric rendering of a 3D uniform scalar field, using RGB color and opacity transfer functions. The technique employed uses two levels of interpolation:

- A cell-averaged or voxel representation produces lower quality and lower apparent resolution images at moderate execution speeds.
- A trilinear or cell-based interpolation results in very high quality images with a slower execution time.

The **vbuffer** module is actually a wrapper for a standalone program, **vbuf**, which performs the image computation:

1. **vbuffer** places the parameters and transfer functions in a disk file for **vbuf** to read.
2. **vbuffer** executes **vbuf** and waits for it to exit.
3. **vbuf** reads the data from the disk file, computes an image, writes it to disk, and exits.
4. **vbuffer** reads the image that was written to disk and places it on the output port.

You can choose between two rendering algorithms:

- **Trilinear interpolation:** This algorithm uses an inverse-mapping scheme to generate images. The 3D field data is decomposed into 8-node *cells*, with one field value at each vertex of the cell. Each cell is processed by accumulating color and opacity along an integration volume which maps into one or more pixels.

  At several locations through this volume, both the value of the scalar field and its gradient are interpolated. The sampled scalar field value is used to map into

transfer functions, which determine the color and opacity at the point. The color for the pixel is determined by a product of the diffuse illumination (due to the dot product of the light source and gradient vectors), the opacity, and the sampled color as accumulated along the volume. After all the pixels that the cell projects into are processed, the algorithm moves on to the next cell. Partial pixel contributions are saved into an in-memory frame buffer.

- **Voxel Approximation** (default): The 3D field is decomposed into cells, as described above. But no interpolation is performed within the cell. Each cell has a single opacity, color, texture color, surface gradient, and set of shading parameters. This method is much faster than the trilinear interpolation method. Use it to get a quick (albeit ragged) look at the data. It is most useful for selecting the opacity and color transfer functions.

INPUTS

Data Field (required; field 3D scalar any-data uniform)
The input field must be 3-dimensional. The data for each field element must be a scalar.

Colormap (required; colormap)
Any AVS *colormap*.

PARAMETERS

Xrotation The x rotation of the data field in degrees.

Yrotation The y rotation of the data field in degrees.

Zrotation The z rotation of the data field in degrees.

ScaleX The x scale factor of the data field. By default the highest resolution axis of the data is scaled between -1.0 and 1.0, and other axes are scaled accordingly by this.

ScaleY The y scale factor of the data field.

ScaleZ The z scale factor of the data field.

Ztrans The z translation of the field. The coordinate system is right handed, so only that data which has a negative z coordinate will be visible.

Imagesize The resolution in pixels of the computed image. The image is always square. The algorithm complexity scales with the number of pixels and the number of cells which have a non-zero opacity.

Background
The background brightness. Zero corresponds to a black background, 1.0 to a white background.

Cell-Based A toggle switch between the voxel approximation algorithm (default) and the trilinear interpolation algorithm (cell-based and substantially slower).

Re-use Scripts
vbuffer generates a script to run **vbuf**. This toggle switch allows you either to reuse these scripts, data files and images, or to build new scripts and files with each new invocation.

Script File The name of the vbuf script file.

Scalar File The name of the vbuf data input file.

Image File The name of the vbuf output image file.

OUTPUTS

Pixmap (pixmap)
An AVS pixmap containing the 2D image rendered by **vbuffer.**

RUNNING VBUF AS A STANDALONE PROGRAM

vbuf itself can be run as a process separate from AVS, using the input files and datasets generated by **vbuffer. vbuf** recognizes these command-line options:

–v1 Send a message to *stdout* each time a new plane of the data is computed.

–v2 As each new cell is processed, lists the *ijk* value. (This is extremely verbose.)

–voxel *number*
When run standalone, the default rendering method uses a trilinear interpolation algorithm. This options invokes the voxel approximation algorithm instead.

The standard command line for running **vbuf** as a standalone program is:

vbuf –v1 < *input_file*

The *input_file* can be the script produced by the **vbuffer** module. Input files contain a small set of commands:

load *datafile* **bin res** *ni nj nk* **aspect** *dx dy dz* [**grid** *gridfile*]
load *datafile* **ascii res** *ni nj nk* **aspect** *dx dy dz* [**grid** *gridfile*]
Data description commands. The optional grid file contains the coordinate values for rectilinear fields. This ASCII file contains all the X coordinates (one per line), followed by all the Y coordinates, then all the Z coordinates.

image *imagefile* **res** *ires jres* **bg** *red green blue* **encode frame** *num*
Image description command.

rotate *rx ry rz* **trans** *tx ty tz* **fov** *angle*
Frame composition command.

color dcue *znear zfar exponent* **ambient** *redA greenA blueA*
scalar(1) red(scalar(1)) green(scalar(1)) blue(scalar(1))
...
scalar(n) red(scalar(n)) green(scalar(n)) blue(scalar(n))
<blank line>
Object color description and transfer function command. Note that this is a multi-line command whose last line is blank. The blank line terminates the command.

opacity
scalar(1) opacity(scalar(1))
...
scalar(n) opacity(scalar(n))
<blank line>
Opacity transfer function command. Note that this is a multi-line command whose last line is blank. The blank line terminates the command.

shade light *xl yl zl xd yd zd falloff*
scalar(1) shade(scalar(1))
...
scalar(n) shade(scalar(n))
Illumination command. The light points from *(xl, yl, zl)* to *(xd, yd, zd).* Note that this is a multi-line command whose last line is blank. The blank line terminates the command.

map *mapfilename* **res** *ires jres kres*
3D texture mapping command.

view Image creation command.

LIMITATIONS

Only uniform fields can be rendered by **vbuffer**.

Some image anomalies can occur along cell boundaries. This is view-orientation dependent.

The execution time can be dramatically reduced by limiting the number of cells that contain a non-zero amount of opacity.

RELATED MODULES

Modules that could provide the Data Field input: read volume

Modules that could be used in place of **vbuffer**:

isosurface
dot surface
alpha blend

Modules that can process **vbuffer** output:

transform pixmap
display pixmap

SEE ALSO

"Vbuffer: Visible Volume Rendering," C. Upson, M. Keeler, *Computer Graphics*, V 22, N 4, August 1988, pp 59-65.

NAME

vector curl – compute the curl of a vector field

SUMMARY

Name	vector curl
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D 3-vector float uniform
Parameters	none

DESCRIPTION

The **vector curl** module accepts a vector field as input and computes the curl of that field as output. This is related to the divergence as follows:

$curl = (DEL \times F)$
$div = (DEL \bullet F)$

... where F is the vector input field.

The equation used to compute the curl is:

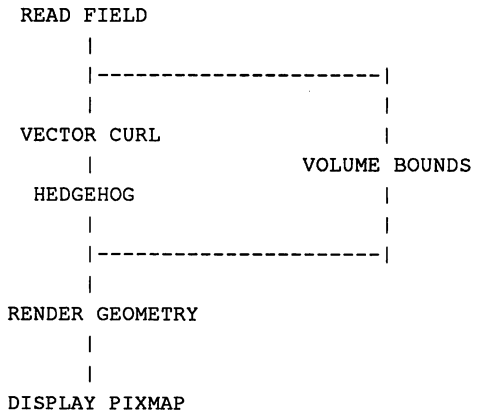
```
new_dx[X][Y][Z] = (dz[X][Y+1][Z] - dz[X][Y-1][Z]) -
                  (dy[X][Y][Z+1] - dy[X][Y][Z-1])

new_dy[X][Y][Z] = (dx[X][Y][Z+1] - dx[X][Y][Z-1]) -
                  (dz[X+1][Y][Z] - dz[X-1][Y][Z])

new_dz[X][Y][Z] = (dy[X+1][Y][Z] - dy[X-1][Y][Z]) -
                  (dx[X][Y+1][Z] - dx[X][Y-1][Z])
```

INPUTS

Data Field (required; field 3D 3-vector float uniform)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

EXAMPLE

The following network reads in a 3D vector field and computes its curl, then displays the field vectors using **hedgehog**:



OUTPUTS

Data Field (field 3D 3-vector float uniform)
The output field is in the same format as the input field.

The **min_val** and **max_val** attributes of the output field are invalidated.

RELATED MODULES

gradient shade
tracer

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example scriptVECTOR CURL demonstrates the **vector curl** module.

NAME

vector div – compute the divergence of a vector field

SUMMARY

Name	vector div
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D scalar float uniform
Parameters	none

DESCRIPTION

The **vector div** module accepts a vector field as input and computes the divergence of that field as output. This is related to the curl as follows:

$curl = (DEL \times F)$
$div = (DEL \bullet F)$

... where F is the vector input field.

The equation used to compute the divergence is:

```
divergence[X][Y][Z] = (dx[X+1][Y][Z] - dz[X-1][Y][Z]) +
                      (dy[X][Y+1][Z] - dy[X][Y-1][Z]) +
                      (dz[X][Y][Z+1] - dz[X][Y][Z-1])
```

INPUTS

Data Field (required; field 3D 3-vector float uniform)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

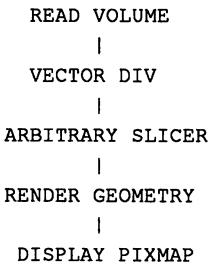
OUTPUTS

Data Field (field 3D scalar float uniform)
The output field has a single floating-point value for each input field element.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE

The following network reads in a 3D vector field and computes its divergence:



RELATED MODULES

vector curl, vector div, vector norm, vector mag, hedgehog, stream lines, stream mesh

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR DIV demonstrates the **vector div** module.

NAME

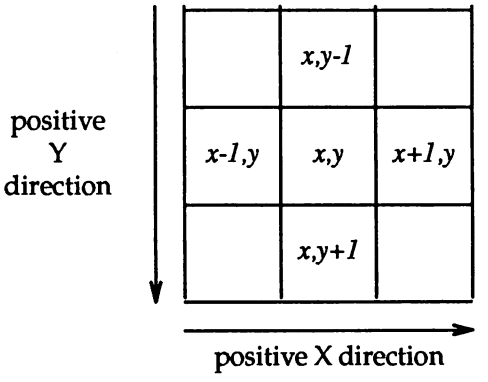
vector grad – compute the vector gradient of a 3D scalar field

SUMMARY

Name	vector grad
Type	filter
Inputs	field 3D scalar float uniform
Outputs	field 3D 3-vector float uniform
Parameters	none

DESCRIPTION

The **vector grad** module computes the gradient of a 3D field. The gradient is treated by some other modules as a "pseudo-normal" to the "surface" for each data element. A "nearest neighbor" algorithm is used to compute the gradient: the difference between the next data value (in each direction) and the previous data value. In two dimensions, this can be represented as follows:




```
Delta_x[X][Y][Z] = data[X+1][Y][Z] - data[X-1][Y][Z]
Delta_y[X][Y][Z] = data[X][Y+1][Z] - data[X][Y-1][Z]
Delta_z[X][Y][Z] = data[X][Y][Z+1] - data[X][Y][Z-1]
```

The **min_val** and **max_val** attributes of the output field are invalidated.

This module is identical to the **compute gradient** module, except that it does *not* normalize the output. **compute gradient** is designed for gradient shading fields, whereas this module is designed for input into the other vector field modules: **vector curl, vector div, vector mag,** and **vector norm**. Note that **vector grad** followed by **vector norm** produces the same results as **compute gradient**.

INPUTS

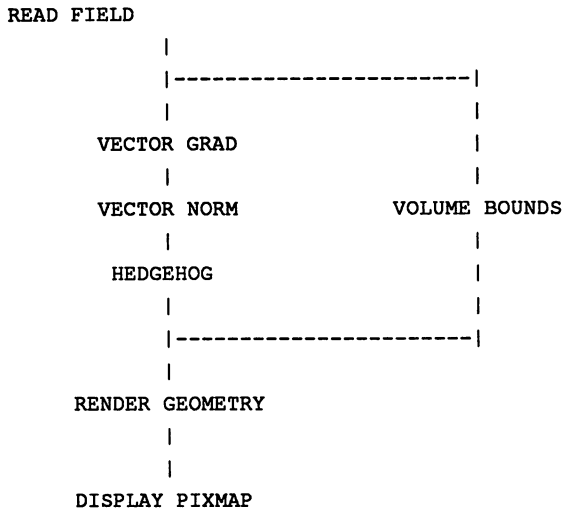
Data Field (field 3D scalar float uniform)
The input field must represent a volume of elements, with a single floating-point value for each input field element.

OUTPUTS

Data Field (required; field 3D 3-vector float uniform)
The output field has a 3D vector of floating-point data values for each element.

EXAMPLE

The following network reads a 3D scalar field, computes its gradient and then uses the **hedgehog** module to display the resulting vector field:



RELATED MODULES

vector curl, vector div, vector norm, vector mag, hedgehog, particle advector, stream lines, stream mesh

LIMITATIONS

There may be algorithms better than "nearest-neighbor" for computing the gradient.

This module produces 12 bytes per pixel (voxel). For example, a 128 x 128 x 128 byte volume is about 2.1 MB before the gradient is computed. The **compute gradient** module produces a 25.2 MB internal data set from this data. This will have an adverse performance effect on systems whose physical memory is 32 MB or less.

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR GRAD demonstrates the **vector grad** module.

NAME

vector mag – compute the magnitude of a vector field

SUMMARY

Name	vector mag
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D scalar float uniform
Parameters	none

DESCRIPTION

The **vector mag** module accepts a vector field as input and computes the magnitude of each vector data value. The output is a scalar field consisting of the magnitudes.

The magnitude equation is:

```
Magnitude[X][Y][Z] = sqrt((dx[X][Y][Z]*dx[X][Y][Z]) +
                          (dy[X][Y][Z]*dy[X][Y][Z]) +
                          (dz[X][Y][Z]*dz[X][Y][Z]) )
```

INPUTS

Data Field (required; field 3D 3-vector float uniform)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

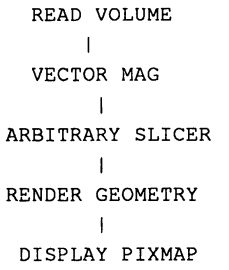
OUTPUTS

Data Field (field 3D scalar float uniform)
The output field has a single floating-point value for each input field element.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE

The following network reads in a 3D vector field and computes the magnitude of the vectors:



RELATED MODULES

vector curl, vector div, vector norm, vector mag. hedgehog, particle advector, gradient shade, stream lines, stream mesh

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR MAG demonstrates the **vector mag** module.

NAME

vector norm – normalize a vector field

SUMMARY

Name	vector norm
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D 3-vector float uniform
Parameters	none

DESCRIPTION

The **vector norm** module accepts a vector field as input, and produces a normalized version of that vector field as output. The normalization equation looks like:

```
Magnitude = sqrt((dx*dx) +  (dy*dy) + (dz*dz))
New_dx    = dx / Magnitude
New_dy    = dy / Magnitude
New_dz    = dz / Magnitude
```

INPUTS

Data Field (required; field 3D 3-vector float uniform)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

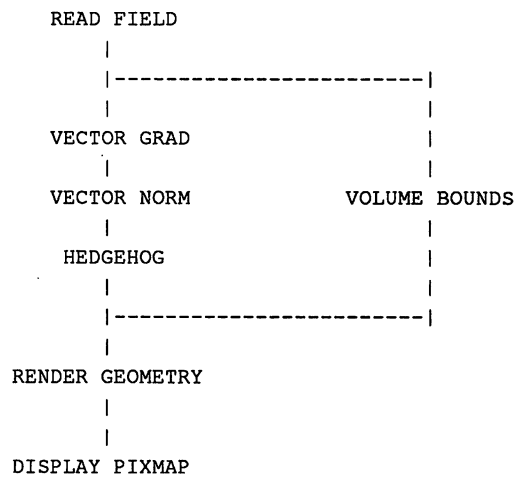
OUTPUTS

Data Field (field 3D 3-vector float uniform)
The output field is in the same format as the input field.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE

The following network reads a 3D scalar field, computes its gradient and then uses the **hedgehog** module to display the resulting vector field:



RELATED MODULES

vector curl, vector div, vector norm, vector mag, hedgehog, particle advector, gradient shade, stream lines, stream mesh

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR NORM demonstrates the **vector norm** module.

NAME

volume bounds – generate bounding box of 3D 3-vector field

SUMMARY

Name	volume bounds	
Type	mapper	
Inputs	field 3D *n-vector any-data any-coordinates*	
Outputs	geometry	
Parameters	*Name*	*Type*
	Hull	toggle
	Min I	toggle
	Max I	toggle
	Min J	toggle
	Max J	toggle
	Min K	toggle
	Max K	toggle

DESCRIPTION

The **volume bounds** module generates lines that indicate the "bounding box" of a 3D data set (field). It is frequently used in conjunction with other geometry-based volume-visualization modules (e.g. **bubbleviz, isosurface, hedgehog, arbitrary slicer**), since it provides some volumetric context for the data.

INPUTS

Data Field (required; field 3D *n-vector any-data any-coordinates*) The input data must be a 3D field, but may have any kind of data at each location in the field.

OUTPUTS

Bounds (geometry) The output *geometry* consists of the lines that form the bounding box.

PARAMETERS

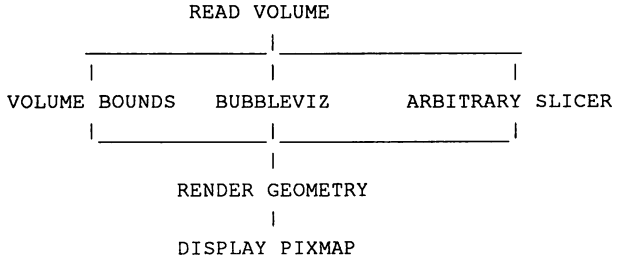
Hull Draws lines for the perimeter of the data set.

Min I
Max I
Min J
Max J
Min K
Max K

These toggle switches provide further help is visualizing the way the computational space is mapped into physical space. Each one fills in one of the six faces of the hull. For example, turning on **Min I** draws a mesh showing the 2D slice of field elements with the minimum index value in the first dimension; turning on **Max K** draws a mesh showing the 2D slice of field elements with the maximum index value in the third dimension.

EXAMPLE

The following network showing a typical usage of **volume bounds**:



RELATED MODULES

read volume, volume manager

SEE ALSO

The example scripts BRICK, HEDGEHOG, PROBE, as well as others demonstrate the **volume bounds** module.

NAME

volume manager – share volumes among subnetworks

SUMMARY

Name volume manager

Unsupported this module is in the unsupported library

Type data

Inputs none

Outputs field 3D scalar byte

Parameters

Name	*Type*	*Choices*
VOLUMGR select	choice	Select, Replace
Volume Manager	browser	
Volume Choices	choice	

DESCRIPTION

The **volume manager** module reads an volume file from disk and outputs the volume as a "field 3D scalar byte". It works like the **read volume** module, except that it has both a caching mechanism and a way of sharing data among **volume manager** modules in separate subnetworks.

See the **read volume** manual page for a description of the volume format.

PARAMETERS

VOLUMGR Select

A choice that determines how newly-read volumes will be placed to the list of currently active volumes:

- If **Select** is chosen, a new volume is added to the end of the list.
- If **Replace** is chosen, a new volume replaces the currently selected member on this list.

  In either case, the change is reflected in *all* the *volume manager* modules in all active subnetworks.

Volume Manager

A file browser that allows you to select an volume file to read.

Volume Choices

A set of choices, listing each of the currently active volumes.

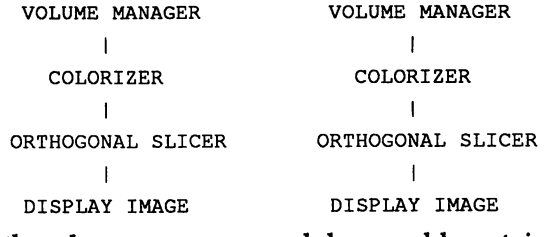
OUTPUT

Data Field (field 3D scalar byte)

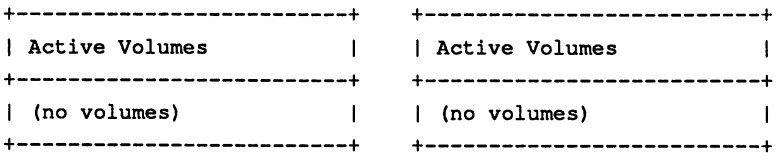
The output is the byte data cast as the scalar data in a 3D *field*.

EXAMPLE

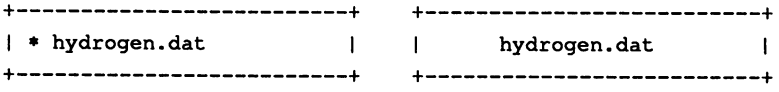
The following subnetworks might be used to display two volumes:



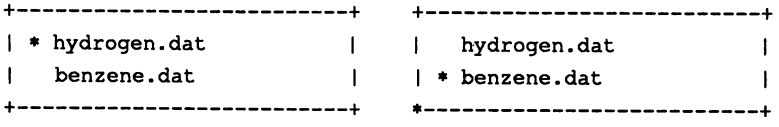
In this case, both **volume manager** modules would contain "select/replace" choice buttons, a file browser, and an area below the browser:



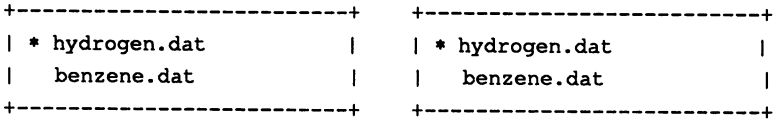
Once a volume (e.g. *hydrogen.dat*) was selected from the browser in the **volume manager** on the left, these buttons would look like this:



If a different file (e.g. *benzene.dat*) is chosen from the browser in the **volume manager** on the right, the buttons would look like this:



By selecting the same active volume, you can have both networks display the same volume:



Now, if you want to replace this volume with a new one, click on the **Replace** buttons above the browser, then select a new file (e.g. *methane.dat*) in just one of the **volume manager** browsers. The result is that all **volume manager** modules with the old volume (*hydrogen*) selected as their active volume will be automatically updated with the new volume (*methane.dat*).

RELATED MODULES

Same as for **read volume**.

LIMITATIONS

The cached volumes are not freed until all **volume manager** modules are destroyed. Because volume data can be large, caching multiple volume datasets can use a lot of memory.

NAME

wireframe – convert object from surface to wireframe representation

SUMMARY

Name	wireframe
Type	filter
Inputs	geometry
Outputs	geometry
Parameters	none

DESCRIPTION

The **wireframe** module transforms an AVS *geometry*, replacing all surfaces defined as polytriangle strips with wireframe representations. This is useful for constructing a wireframe version of an object that has been defined as a shaded surface.

INPUTS

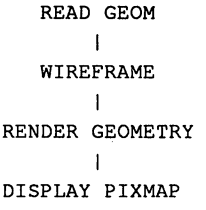
Geometry (required; geometry) Any AVS *geometry*, created with the *libgeom* library or produced by another AVS module.

OUTPUTS

Geometry A geometry that represents the same object as the input data.

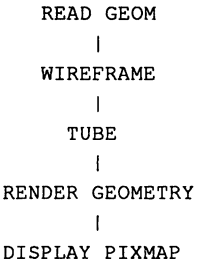
EXAMPLE

This example shows the use of the **wireframe** module to generate a wireframe version of a polygonal object:



EXAMPLE 2

This example uses the **wireframe** and **tube** modules to have a geometry involving spheres drawn with cylinders instead of lines:



RELATED MODULES

read geom, offset, shrink, flip normal, tube, render geometry

LIMITATIONS

The **wireframe** module generates lines based on the order of the vertices of a polytriangle strip. Sometimes, the resulting object is not exactly what you want. It may have "cobwebs" and other (usually invisible) data inconsistencies of the original polytriangle strip. You may need to regenerate the original data in order to produce the desired wireframe representation.

SEE ALSO

The example scripts TUBE, and WIREFRAME demonstrate the **wireframe** module.

NAME

write field - write a field description to disk

SUMMARY

Name	write field				
Type	data output				
Inputs	field *any-dimension n-vector any-data any-coordinates*				
Outputs	none				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Write Field	browser			

DESCRIPTION

The **write field** module writes an AVS *field* description to disk. The field format on disk includes two parts, an *ASCII header* and a *binary area*. This format is described in detail in the manual page for **read field**.

INPUTS

Data Field (field *any-dimension n-vector any-data any-coordinates*)
The input can be any AVS *field*.

PARAMETERS

Write Field

A file browser that allows you to specify the name of the field file to be created. The file suffix *.fld* is appended to the name automatically. If the file already exists, **write_field** issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

After the field file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

EXAMPLE 1

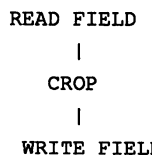
Following is an example of a file produced by **write field**:

```
ndim=3          # number of dimensions in the field
dim1=64         # dimension of axis 1
dim2=64         # dimension of axis 2
dim3=64         # dimension of axis 3
nspace=3        # number of physical coordinates per point
veclen=1        # number of components at each point
data=byte       # data type (byte, integer, float, double)
field=uniform   # field type (uniform, rectilinear, irregular)

262,144 bytes of data
```

The field has three dimensions (it's a volume), 64x64x64. There is a single byte at each point, and the field is uniform.

EXAMPLE 2

The following network reads in a field, crops it and then writes the resultant field to a file:



RELATED MODULES

print field compare field

write field writes *any* AVS field file.

AVS data input modules that produce field output:

image manager
read field
read image
read volume
volume manager

AVS filters that produce field output:

clamp
colorizer
combine scalars
compute gradient
contrast
crop
dot surface
downsize
extract scalar
field to byte
field to double
field to float
field to int
geom to scatter
gradient shade
histogram stretch
interpolate
mirror
threshold
transpose
vector curl
vector div
vector grad
vector mag
vector norm

AVS mappers that produce field output:

bubbleviz
orthogonal slicer
pixmap to image

ERRORS

Write field complains if it can't open the file, or if there isn't enough space to write the complete file.

SEE ALSO

The example script WRITE FIELD demonstrates the **write field** module.

NAME

write image – store image data in a file

SUMMARY

Name	write image
Type	data output
Inputs	field 2D 4-vector byte uniform
Outputs	none
Parameters	*Name* *Type* Write Image browser

DESCRIPTION

The **write image** module writes an AVS *image* data structure to a file. This structure takes the form of a "field 2D 4-vector byte". See the **read image** manual page for a detailed description of the image format.

INPUTS

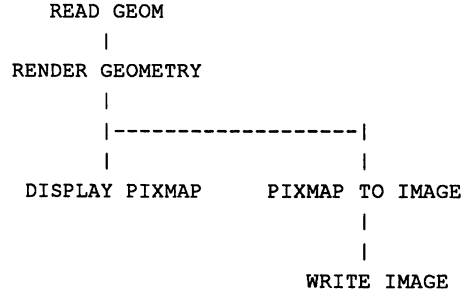
Data Field (required; field 2D 4-vector byte uniform)
The input can be any AVS *image*.

PARAMETERS

Write image
A file browser that allows you to specify the name of the image file to be created. The file suffix *.x* is appended to the name automatically. If the file already exists, **write_image** issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

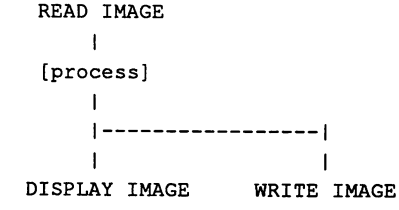
After the image file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

EXAMPLE



AVS IMAGE VIEWER

All of the data pre-processing networks in the AVS Image Viewer have the following structure:



... where <process> is one of the following modules **crop, downsize, mirror, transpose,** or **interpolate.** These networks are in the *image_viewer* subdirectory of the */usr/avs/networks* directory.

RELATED MODULES

Image processing:

contrast, threshold, histogram stretch, clamp, interpolate

Decompose/compose images from separate bands:
extract scalar, combine scalars
Show image:
display image

Take output from data output module, and write the data out as an image:
render geometry, transform pixmap, pixmap to image

SEE ALSO

read image
image viewer

The example script WRITE IMAGE demonstrates the **write image** module.

NAME

write ucd - write unstructured cell data to disk

SUMMARY

Name	write ucd	
Type	data output	
Inputs	ucd structure	
Outputs	none	
Parameters	*Name*	*Type*
	Write UCD	browser

DESCRIPTION

The **write ucd** module writes a UCD structure to disk.

write ucd outputs a binary file. This file format is read by the module **read ucd**. The format of this binary file, however, is not the same as the ASCII UCD format which is also read by **read ucd**.

The format of UCD structure, as well as the format of ASCII and binary UCD files is described in detail in the manual page for **read ucd**, and in Appendix E of the *AVS Developer's Guide*.

INPUTS

ucd structure

The input can be any UCD structure.

PARAMETERS

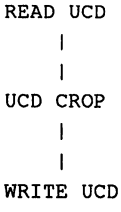
Write UCD

A file browser that allows you to specify the name of the ucd file to be created.

After the UCD file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

EXAMPLE

The following network reads in a UCD structure, crops it, and writes the resulting structure to disc:



RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

ERRORS

write ucd will complain if it can't open the file, or if there isn't enough space to write the complete file.

SEE ALSO

The example script WRITE UCD demonstrates the **write ucd** module.

NAME

write volume – write volume data to a file

SUMMARY

Name	write volume	
Type	data output	
Inputs	field 3D scalar byte uniform	
Outputs	none	
Parameters	*Name*	*Type*
	Write Volume	browser

DESCRIPTION

The **write volume** module writes volume data to a file. The volume is in the AVS format "field 3D scalar byte". The data format on disk is:

1 byte: number of voxels in X 1 byte: number of voxels in Y 1 byte: number of voxels in Z nx * ny * nz * 1 byte: voxel data

Each time the file is written, the filename is reset to NULL. This prevents successive changes upstream in the network to automatically trigger a volume data file to be written. A new filename must be entered each time the file is to be written out.

If the file to be written exists, the following warning appears:

```
File FILENAME
  already exists.  Do you want to overwrite it?
```

Two choices are presented. If you select **Cancel**, the write operation is aborted. If you select **Overwrite**, the existing file on disk is replaced with the new volume data.

This module is commonly used to pre-process a volume database for later use. For example, the input data might be very low-contrast. You could construct a network that includes the **contrast** module and the **write volume** module. Once you select appropriate settings for the contrast, the data could be written to a file, and used later for other types of processing.

INPUTS

Data Field (required; field 3D scalar byte uniform)
: The input data must be a 3D field, with a byte value at each location in the field.

PARAMETERS

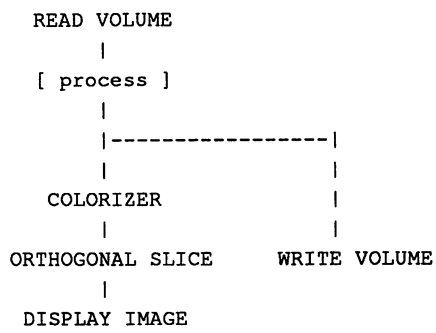
Write Volume
: A file browser that allows you to specify the name of the volume data file to be created. The file suffix *.dat* is appended to the name automatically. If the file already exists, **write_volume** issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

RELATED MODULES

read volume, clamp, contrast, crop, downsize, histogram stretch, interpolate, mirror, threshold, transpose

EXAMPLE

The data pre-processing networks in the AVS Volume Viewer all use the the following model:



where [process] is one of the following: **crop, downsize, mirror, transpose,** or **interpolate.** These networks are in the *volume_viewer* subdirectory of the *avs/networks* directory.

LIMITATIONS

The format of volume databases on disk is severely limiting. The dimensions are restricted to a maximum of 255 in x, y and z. The data also must be in the range 0 - 255.

SEE ALSO

read volume

The example script WRITE VOLUME demonstrates the **write volume** module.