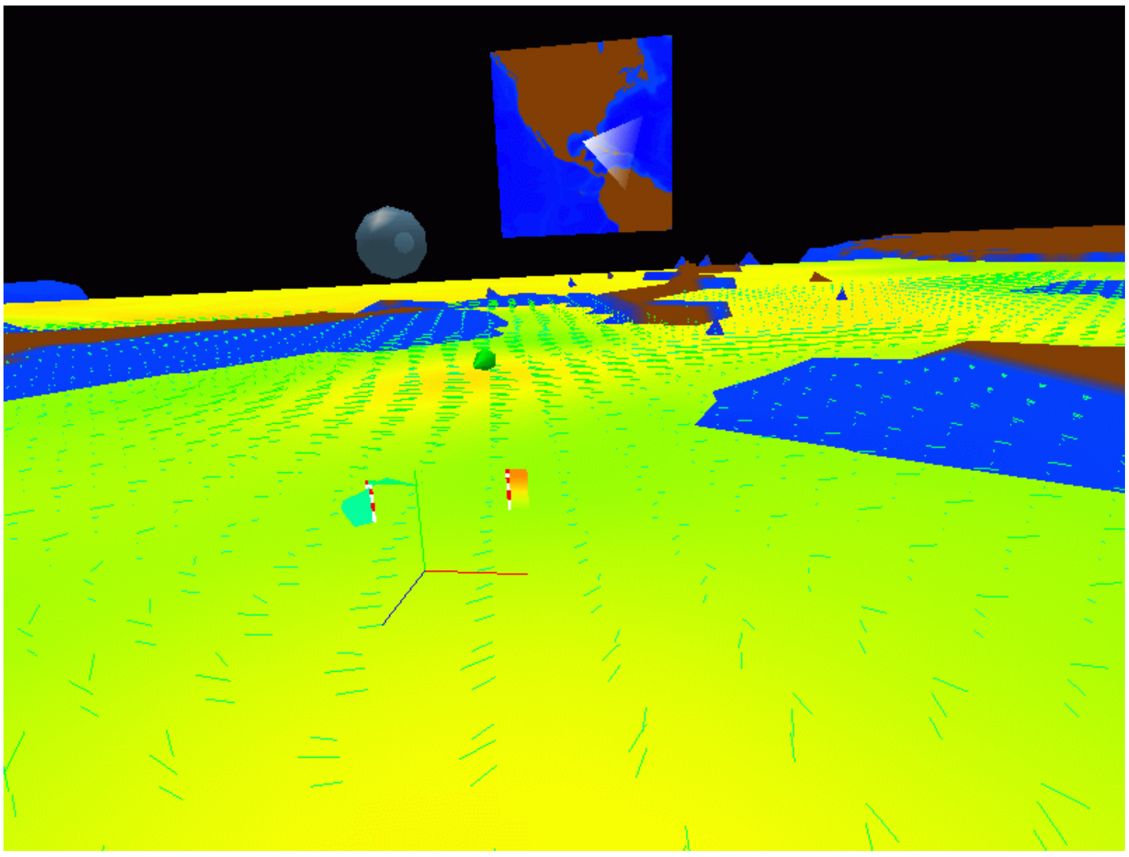
# *Triton* User Manual – version 1.0

Randall E. Hand and R. Moorhead

December 2001

ERC Technical Report # MSSU-COE-ERC-01-15

Visualization, Analysis and Imaging Lab
Engineering Research Center
Mississippi State University
Mississippi State, MS 39762



## Requirements

### *Run-Time Requirements*

*Triton* v1.0 requires:

- An SGI with on-board Texture memory
- OpenGL
- Irix 6.5
- Read access to all data
- Configuration files for VRJuggler (specific to your system)

Triton was designed to run on SGI Irix systems. VRJuggler is compatible with other platforms (such as Sun, PC, etc), but Triton has never been tested on these systems and may not run properly.

### *Compile-Time Requirements*

For compilation, you also need:

- VRJuggler 1.0+ (refer to VRJuggler documentation for additional requirements and instructions)
- GNU Compiler utilities (gcc, g++, and gmake specifically)
- MIPSpro Compiler version 7.3.1.1m or greater

## Execution

To run Triton, after it has been compiled, simply use the following syntax:

```
./roamer vrjuggler-config-files [-c config]
```

Where

- vrjuggler-config-files – all of the files (built with vjcontrol) necessary for your VRJuggler setup.
- config – the configuration script for your system.

If the configuration file is not specified, then by default config.tile will be loaded.

## User Interface

Once the system has started, you can use the following simple user interface (similar to CTHRU [1],[2])

To Rotate – simply turn the “wheel” or analog axis left or right to rotate

To Move – Use the “trigger” or other analog axis to move forward and backward.

## Configuration Basics

Configuration takes place through text files called "configuration scripts". A configuration script is formed from several blocks, each looking something like this:

```
[HEADER]
key1 value1
key2 value2
key3 value3
end
```

The Header, key, and value fields can contain any printable characters, except white spaces. The Header is enclosed in square braces, and for readability are usually all caps. Keys vary depending on the type of section this Header represents. Headers, keys, and default values are listed below, in their corresponding sections.

The "end" key is critical, as it indicates the end of a configuration block. Without it, blocks will run together. Sometimes this may be useful, to avoid duplicating sections of the file, but usually it is not.

Only two Headers are required for Triton: SYSTEM and DATA. These sections are discussed in more detail later. Keys and headers listed below are case specific.

Keys are actively searched for, not parsed. This means that if you insert unrecognized keys, there will be no error.

## Configuring Data Sources

*Triton* supports the following data formats:

- Raw Binary rectilinear data
- ASCII Fixed Data
- On-The-Fly Magnitude Data (calculated from other data sources)

These data formats are specified at run-time in the configuration files, and can be mixed and matched to form a very flexible data set. For example, U-flow and V-Flow files can be loaded through two Raw Binary data loaders, and the magnitude of the flow can be calculated through an On-The-Fly Magnitude Data loader to create colormapped flow-magnitude surfaces.

Data is also cached in memory, to minimize memory consumption. This also helps to minimize network traffic and hard drive accesses, thereby improving performance. Data is loaded on its first access, and least recently data is replaced later by new data on an as-needed basis. Also supported is multi-file tiled data (like GTOPO30), where each tile is cached individually, improving memory usage.

Each data loader supports scaling & masking operations, meaning that none of the visualization methods need to know about them. Each loader also supports multiple layers, but does not support multiple timesteps at this time.

### *Configuring the Data Source Manager*

The DATA section configures the Data Source Manager. The following keys are supported:

- *Databases* – an integer indicating the exact number of data sources your program uses.
- *DataXName* – a header name indicating from which section to read this data source's configuration information. X is a number between 1 and *Databases*.
- *DataXType* – the type of data source to use for data source X. This must be one of:
    - TILED_FLOAT – data that is read from a series of raw binary files
    - MAGNITUDE_FLOAT – data that is calculated on the fly from two data files
    - FIXED – data that is constant across an area, and is specified in the configuration script.

A sample DATA section would look like this:

```
[DATA]
Databases 5
Data1Name BATHYMETRY
Data1Type TILED_FLOAT
Data2Name DEPTHS
Data2Type FIXED
```

```
Data3Name VECTOR_U_DATA
Data3Type TILED_FLOAT
Data4Name VECTOR_V_DATA
Data4Type TILED_FLOAT
Data5Name VECTOR_MAG
Data5Type MAGNITUDE_FLOAT
end
```

Each DataName must correspond to a Header somewhere in the configuration script. If it does not, the program will quit in an error. Data sources that you do not use may be defined, as they will only be loaded when accessed by a visualization method.

## *Configuring a FIXED Data Source*

A fixed data source is defined inline, in the configuration script. It creates a data source that returns a fixed value for any location within a layer of data. This is used most often to list depths for use by Vector Posts and Scalar Posts.

Only a few keys are used here:

- *fixed_depths_scale* – a floating point number, indicating a scale to multiply all following numbers by. (Useful if you frequently scale your data differently, and don't want to recalculate all of the depths). This key is optional, defaults to 1.0.
- *layercount* – an integer number, indicating the number of data values.
- *depthX* – a floating point number, indicating the value to return for layer X. A *depthX* key must exist for each X between 1 and *layercount*.

A sample Fixed Data source would look like this:

```
[DEPTHS]
fixed_depths_scale -0.005
layercount 45
depth1     2.5
depth2     7.5
…
depth44 5800
depth45 6100
end
```

**NOTE:** This is one of the few sections that not using the end tag can be  useful, in rare instances. If you need the same fixed data values, at two different scales, you could do this:

```
[DEPTHS1]
fixed_depths_scale -0.005
[DEPTHS2]
fixed_depths_scale +0.005
layercount 45
depth1     2.5
depth2     7.5
…
```

```
depth44 5800
depth45 6100
end
```

This will create two different data sources containing the same data, but scaled differently. In this case, they are the same quantities but different signs.

## *Configuring a TILED Data Source*

A Tiled data source currently only supports Floating Point data, so you must use TILED_FLOAT. A tiled data source usually only contains one tile (one binary file), but can support several (as in GTOPO30). Tiled data sources use the following keys:

- *cache* – an integer number, indicating how many tiles to cache in memory at once. Each layer of the data is cached independently, so *cache* may be much larger than *tilecount.* It should at least be as large as *layers*.
- *tilecount* – an integer number, indicating how many tiles there are.
- *tile*X – a header found later in the file indicating what tile to load. A *tile*X entry must exist for each X between 1 and *tilecount*.
- *layers* – an integer number indicating how many layers exist in each file.
- *landmask* – a number to mask out of the dataset. This value is optional, and if is not specified, no data is masked.
- *maskop* – an operation to use for masking. This operation is combined with *landmask* to determine what values to mask (eg. Everything greater than *landmask*, everything less than *landmask*, etc.) It can be =, >, or <. This value is optional, and defaults to = if not specified.
- *landvalue* – a floating point number. If masking is enabled then this value is written in place of masked values. This number needs to be pre-scaled. *This parameter is optional, and if not specified, masked data is simply "removed" from the dataset.*
- *flip_x* – (yes/no) If yes, the data will be inverted on the X axis of the 2-dimensional data plane, effectively flipping the data top-to-bottom. This parameter is optional, and no action is taken if not specified. Each y value will be replaced with (height-y). Another way to think of the operation is that the first line is written along the top of the data plane instead of the bottom. This allows Triton to handle data sources where the origin of the data is at the top left or the bottom left.
- *scale* – a floating point number, all data is multiplied by this scaling factor once loaded. This parameter is optional, and if not specified it defaults to 1.0.
- *minvalue* – a floating point number. If specified then the data is moved to make this the minimum value. This number needs to be pre-scaled. This parameter is optional, and if not specified no action is taken.
- *spread* – a floating point number indicating the desired range between the minimum and maximum value. If specified, it automatically calculates a scaling factor to force the minimum and maximum to be this far apart. This parameter is optional, and if not specified no action is taken.

The data is multiplied by *scale* before the minimum and maximum is calculated. Masked values are ignored during this calculation. If *landvalue* is specified, a separate pass is made next to write this value in, so the *landvalue* needs to be scaled in the configuration block. Another pass is made through the scaled and masked data, to set the *minvalue* and *spread,* if specified.

The data keys *minvalue* and *spread* are useful when you do not know the range of the data that you are loading, and want to force it within a certain range. (ie. force a minimum value of *minvalue* and a maximum value of *minvalue+spread)*

For each *tile*X, a configuration block needs to exist detailing the exact specifications of the file. The keys for each of those blocks looks like this:

- *data* – the actual path and filename of the data file (no spaces)
- *startx, starty* – integer numbers, the x and y location of the data, in object space
- *width, height* – integer numbers, indicating the width and height of the data in object space.

In this context, object space refers to the “plane” below the user. Usually *startx* & *starty* are 0. However, in a system with multiple adjacent data tiles, *startx* & *starty* will indicate each tile’s starting point within the whole data set. All following examples use a single-tile data system.

A sample tiled data source block would look like:

```
[BATHYMETRY]
scale 0.005
landmask 1
tilecount 1
cache 1
tile1 PIPS
layers 1
flip_x yes
end

[PIPS]
data NCOM/model/bathymetry.bin
startx 0
starty 0
width 1024
height 640
end
```

## Configuring a MAGNITUDE Data Source

A Magnitude data source takes two data sources, and uses them to calculate a scalar magnitude point-by-point to generate another data source. This is useful if your using vector data and want to colormap by magnitude, but don’t have a separate magnitude dataset.

A magnitude data source uses the following keys:

- *data_u* – a data source (listed BEFORE this one) containing scalar u values.
- *data_v* – a data source (listed BEFORE this one) containing scalar v values.
- *startx, starty* – integer numbers indicating the starting points within *data_u* & *data_v*
- *width, height* – integer numbers indicating the area to read from *data_u* & *data_v*
- *layercount* – an integer number indicating the number of layers to generate from *data_u* & *data_v*

If *startx* and *starty* are not 0, or *width* and *height* are not equal to the width and height of *data_u* and *data_v*, then the resulting dataset will not be the same size. This could cause problems for certain visual components.

The entire dataset is generated on the first access, so there is no noticeable delay during run time for static data.

A sample Magnitude data source would look like this:

```
[VECTOR_MAG]
data_u VECTOR_U_DATA
data_v VECTOR_V_DATA
startx 0
starty 0
width 1024
height 640
layercount 20
end
```

## Configuring Visualization Methods

*Triton* currently supports the following visualization methods:

- Scalar Surfaces
- Scalar Posts
- Vector Posts
- Vector Flow Glyph Layer
- Heightfields (Rendered with ROAM)

Each of these methods will load data from any of the available data loaders, and supports ISTV-style colormaps [3][4]. Any combination of these visualization methods can be operating at the same time, and any one can be duplicated as often as wished. For example, the user could have a Scalar Surface at layer 7 (One Scalar Surface), Vector Flow Glyph layers for layers 1 through 5 (5 Vector Flow Glyphs), and several Scalar Posts scattered around, all while rendering the context bathymetry using a ROAM Heightfield.

### *Configuring the Visualization Manager*

The visualization manager is responsible for creating and maintaining all of the visual components. It is setup in the other required block, SYSTEM. It uses the following keys:

- *startx, starty* – integer numbers indicating the starting x and y location of the user on load.
- *ComponentCount* – an integer number greater than or equal to the number of visual components.
- *AddComponent*X – a header name indicating where to get information for this component. X must be between 1 and *ComponentCount (inclusive)*.

Each header listed in an *AddComponent*X key must appear later in the configuration script, or the program will exit with an error. Each corresponding configuration block must also contain the key *type*, and it must be one of the following:

- WAND_POINTER – a simple line indicating the direction of the wand. Not very useful except for VRJuggler configuration of the wand and screens.
- SCALAR_POST – a scalar post
- SCALAR_QUAD – layers of scalar values
- VECTOR_POST – a vector post
- VECTOR_GLYPHS – layers of vector flow glyphs
- ROAM_FLOAT – a ROAM heightfield

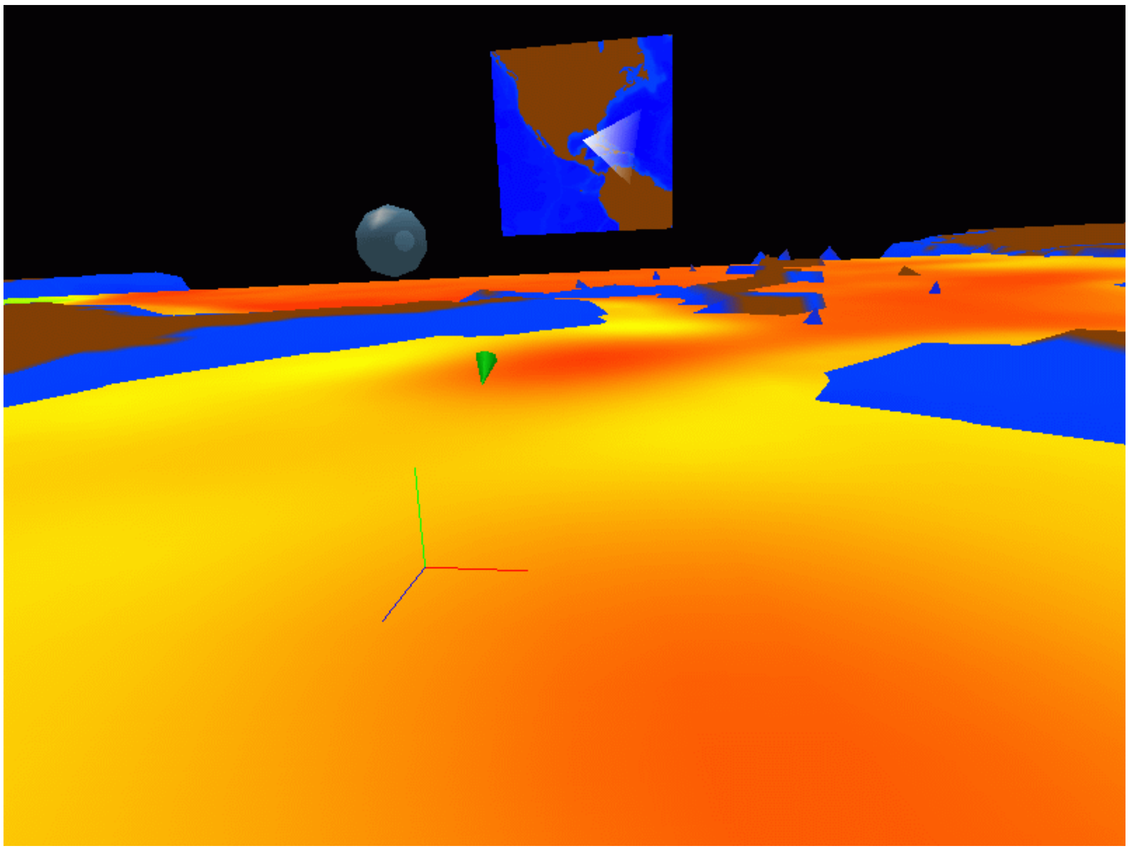
The last five types correspond to the 5 currently supported visualization methods listed below.

A sample SYSTEM block would look like this :

```
[SYSTEM]
startx 900
starty 400
```

```
ComponentCount 10
AddComponent1 MAINROAM
AddComponent2 DATA_POSTS
AddComponent3 DATA_FLOW_GLYPHS
AddComponent4 DATA_FLOW_POST
AddComponent5 DATA_SURFACE
end
```

## Scalar Surfaces



A scalar surface renders a colormapped scalar value for all data points in a rectangular subsection of a single layer, centered around the user. It uses the following keys:

- *colormap* – an ISTV colormap file to map to this layer
- *x_length, y_length* – how large to make this layer. It will extend away from the user *x_length*/2 in both the positive and negative X directions, and *y_length*/2 in both the positive and negative Y directions. (X & Y are the data axes, not visualization axes). Each number must be a positive integer power of 2 (32,64,128,256, etc)
- *data_values* – a data source to retrieve the scalar values used for rendering
- *depth_values* – a data source to retrieve the values for the depth of each layer.

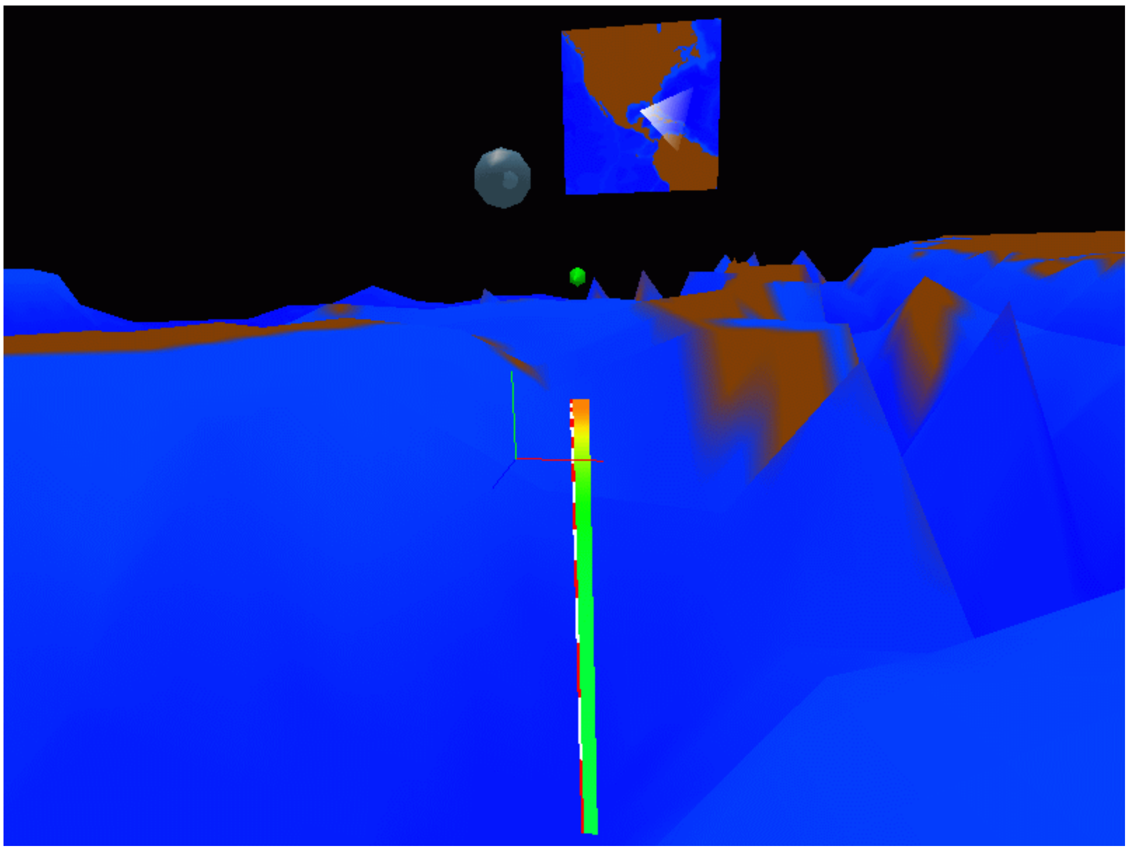
- *layer*X – (on/off) if on, then a scalar surface will be generated and rendered for this layer.

The entire area (*x_length* * *y_length*) is generated as an RGBA texture and drawn on a rectangle at the desired location. The height of the quad is taken from *depth_values* at the user's location, and the quad is centered around the user. *depth_values* is usually an ASCII Fixed-Data source, to keep the data layers at constant heights independent of user position.

A sample scalar surface would look like this:

```
[DATA_SURFACE]
type SCALAR_QUAD
colormap temperature.ct
x_length 128
y_length 128
data_values SCALAR_DATA
depth_values DEPTHS
layer7 on
end
```

## Scalar Posts



A scalar post renders a colormapped scalar value for a single point, relative to the user's position, in all layers. It uses the following keys:

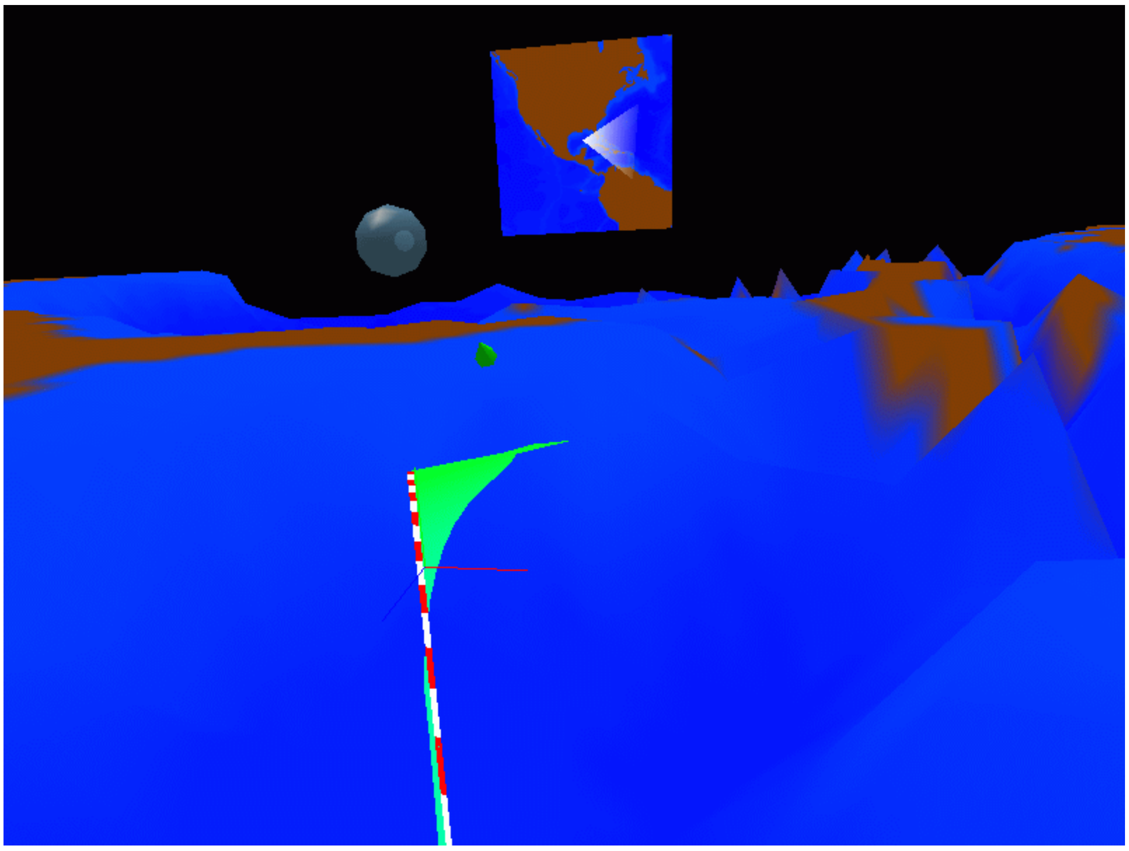
- *x_offset, y_offset* – floating point numbers that indicate where to place the post relative to the user's position. These coordinates are in the data's object space, so the post will move around the user as the user rotates.
- *colormap* – an ISTV colormap to use for rendering
- *data_values* – a data source containing the actual data values to render
- *depth_values* – a data source containing the depths of each data layer.

The data in *data_values* at (x,y, each *depth_values*) is rendered as a post penetrating all layers of the data. Smooth shading is used to make the colors smoothly transition. So that the user can tell the exact extent of the layers of depth_values, a striped post is drawn next to it, with the alternating color transitions indicating the exact layer extents. *data_heights* is usually a Fixed-Data source, but can be a varying data source and the transition points will change to match the current position. This visualization method is typically used to visualize a variable, such as temperature, that can vary significantly from layer to layer in value. When the user is between data points, the data is clamped to the nearest data point.

A sample scalar post block would look like this :

```
[DATA_POSTS]
type SCALAR_POST
x_offset 3.0
y_offset 1.0
colormap temperature.ct
data_values SCALAR_DATA
data_depths DEPTHS
end
```

## Vector Posts



A vector post renders a colormapped scalar value for a single x-y position, relative to the user's position, in all layers. Instead of being a flat fixed-width post though, the surface bends around the striped post to reflect two other scalar fields defined by the user. It has the following keys:

- *x_offset, y_offset* – floating point numbers that indicate, relative to the user's position, where to place the post. These coordinates are in the data's object space, so the post will move around the user as the user rotates.
- *colormap* – an ISTV colormap to use for rendering
- *color_values* – a data source containing the data values to which to map the colors
- *depth_values* – a data source containing the depths of each data layer.
- *scaling* – a floating point number by which to multiply the widths to make them more visible. The screenshot above used 5.0. This parameter is optional, and defaults to 1.0
- *flowu_values* – a data source containing u flow values
- *flowv_values* – a data source containing v flow values

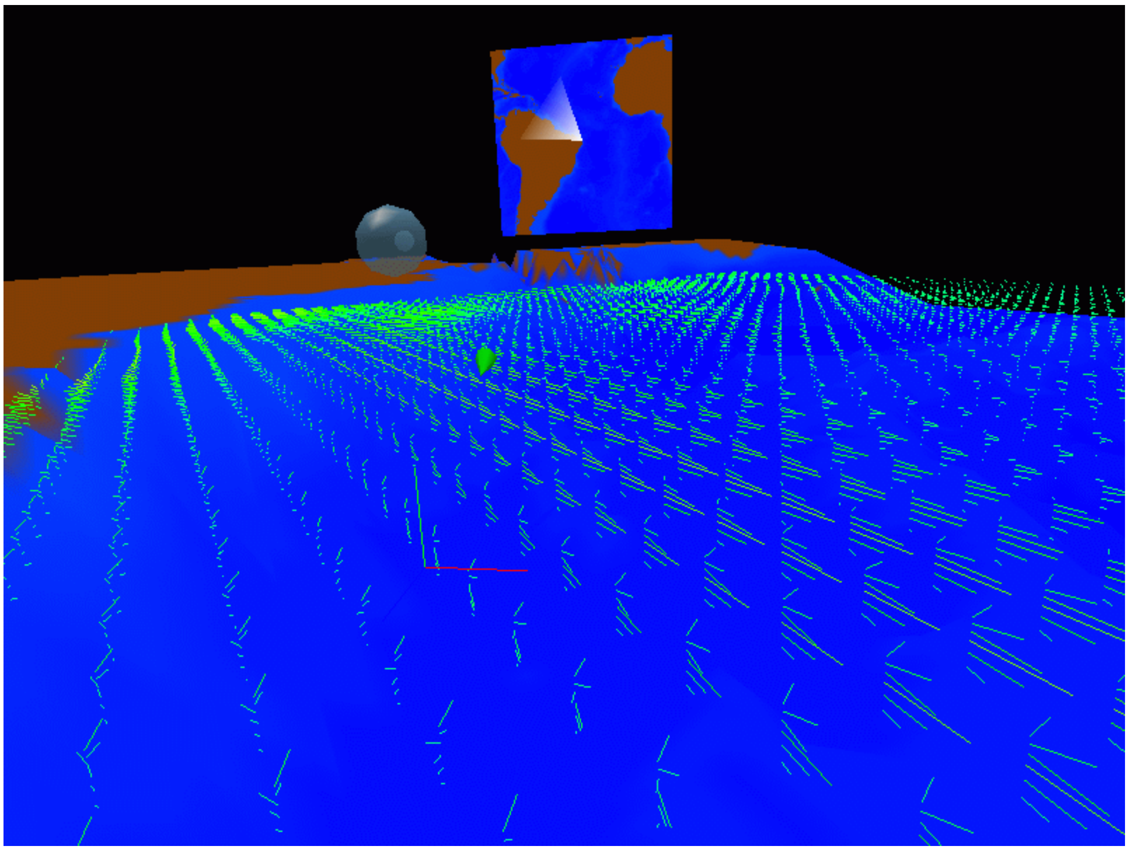
The data in *color_values* at (x,y, each *depth_values*) is rendered as a ribbon spanning all layers of the data, with the width and direction determined by (*scaling*flowu_values*, *scaling*flowv_values*). Smooth shading is used to make the colors smoothly transition. So that the user can tell the exact extent of the layers of *depth_values*, a striped post is drawn next to it, with the transitions indicating the exact layers. *depth_values* is usually a Fixed-Data source, but can be a varying data source and the transition points will change to match the current position. This is typically used to indicate the flow at a single lat/long position in all layers.

When the user is between data points, the data from the nearest point is used.

A sample vector post block would look like:

```
[DATA_FLOW_POST]
type VECTOR_POST
colormap temperature.ct
flowu_values VECTOR_U_DATA
flowv_values VECTOR_V_DATA
depth_values DEPTHS
color_values VECTOR_MAG
x_offset 3
y_offset -1
scaling 5.0
end
```

## Vector Flow Glyph Layers



A vector flow glyph layer renders a colormapped scalar value onto a set of oriented lines indicating a vector value for all points in a rectangle around the user, in a single layer. It uses the following keys:

- *colormap* – An ISTV colormap to use for rendering
- *x_length, y_length* – integers indicating the size of the area to render ( a rectangle centered around the user)
- *color_values* – a data source containing the data values to which to map the colors
- *depth_values* – a data source containing the depths of each data layer.
- *flowu_values* – a data source containing u flow values
- *flowv_values* – a data source containing v flow values
- *layer*X – (on/off) if on, then this layer is rendered. X must be between 1 and the number of layers in the above data sources.

For each data point in a rectangle (*x_length*y_length*) around the user in the *layerX*, a line is drawn from the grid point in the direction of (*flowu_values, flowv_values*). The line is colormapped by the values in *color_values*, and is the length

**sqrt((*flowu_values***flowu_values*) + (*flowv_values***flowv_values*)).**

*color_values* is usually a magnitude data source, calculating the magnitude of the vector at each point.

A sample vector flow glyph block would look like:

```
[DATA_FLOW_GLYPHS]
type VECTOR_GLYPHS
colormap temperature.ct
x_length 64
y_length 64
flowu_values VECTOR_U_DATA
flowv_values VECTOR_V_DATA
depth_values DEPTHS
color_values VECTOR_MAG
layer1 on
layer3 on
layer5 on
end
```

## ROAM Heightfields

(ROAM was used to render the bathymetry in the previous images)

Probably the most complex of all the visualization methods, ROAM Heightfields let you render a heightfield of any user-defined quality at interactive framerates. ROAM is a run-time level of detail algorithm that stands for Real-Time Optimally Adapting Meshes. It recursively subdivides a surface into triangles, until either a certain detail is achieved or a certain number of triangles is obtained. By changing this upper bound, one can ensure that the framerate stays high no matter how complex the geometry is. It recognizes the following keys:

- *colormap* – an ISTV colormap mapped onto the surface.
- *data* – the data source from which to read the heightfield.
- *data_layer* – the layer of *data* to use. This parameter is optional and defaults to 1.
- *color* – the data source to use for generating colors. **NOTE:** if “shared” is specified here, then *color* will be the same as *data*. This is much faster than actually listing *color* to be the same as *data*.
- *color_layer* - the layer of *color* to use. This parameter is optional and defaults to 1.
- *patchsize* – the patch size to use. Must be a positive integer power of 2 (16, 32, 64, etc). This sets the minimum resolution (the surface will always render these points). Large patch sizes will degrade visual quality, while small patch sizes will increase computation time.
- *viewsquare* – the area to visualize. It must be an odd integer. A square of *viewsquare*viewsquare* patches will be rendered, centered around the user.

- *variance_depth* – an integer indicating the depth of the variance tree. This should be sqrt(*patchsize)* +1. Setting it lower will reduce the number of trianges tessellated correctly. Setting it higher simply wastes memory, but does not impact performance. If this number is set lower than recommended, then when a surface is split *variance_depth* levels, it will automatically split it the remaining levels, regardless of requirements.
- *max_nodes* – an integer, sets the maximum number of splits. This should be much higher than the number of triangles that you want.
- *render* – an integer, specifying the rendering method to use.
    - 0 – filled and textured.
    - 1 – filled and lit, no texture
    - 2 – filled. No lights, no texture.
    - 3 – wireframe.

  This parameter is optional, and defaults to 0 (filled and textured)
- *map_size* – an Integer, indicating the “size” of the map, used to adjust the resolution.
- *gDesiredTris* – the Desired number of triangles per frame.
- *FrameVariance* – the starting allowable variance before splitting another level.
- *MinVariance* – The lowest value FrameVariance can be. This parameter is optional, and defaults to 0.
- *AutoLOD* – (1/0) if 1, then the *FrameVariance* is automatically changed each frame, to optimally hit *gDesiredTris.*

ROAM renders the data as a surface beneath the user. Each area is generated to a full-detail texture, then the texture is mapped on the lower-detail geometry, to ensure the color data is accurate, even if the geometry is not. These textures are also used to render a map in front of the user, with a small cone indicating the current view.

This method also implements View-Frustrum Culling on a per-wall basis. This means each wall draws only the parts of the scene visible from that side. This vastly increases the framerate, because each side wall can only see ¼ of the scene, and the floor can only see a tiny area when the user is bound close to the ground.

Several articles can be found on ROAM, but this implementation has been heavily modified to support the following:

- Dynamic loading of data – As the user moves around, data moves off one edge of the meshable-area, and new data must be loaded. This implementation only loads new data when required where most load all data at startup.
- Separate tessellation and rendering – Most methods tessellate into triangles in a binary tree, and render directly from the tree. The implementation herein achieves an order of magnitude speedup from converting the tree into a simple array list before rendering, and rendering from that.

- Per-Context View Frustrum Culling – Textures are only generated and loaded for areas that can be seen from the current wall, and surfaces are only rendered if it is possible they could be seen from there.
- Several other optimizations through the use of lookup tables.

ROAM is usually used to render the bathymetry for contextual information.

A sample ROAM block would look like this:

```
[MAINROAM]
type ROAM_FLOAT
colormap terrain.ct
data BATHYMETRY
data_layer 1
color shared
patchsize 64
viewsquare 9
variance_depth 9
max_nodes 50000
render 0
map_size 65536
gDesiredTris 8000
FrameVariance 50
MinVariance 0.0
AutoLOD 1
end
```

## Sample Configuration Files

### *Sample PIPS configuration*

This configuration file loads up bathymetry, one scalar post, and one scalar surface.

```
[SYSTEM]
startx 900
starty 400
ComponentCount 10
AddComponent1 MAINROAM
AddComponent2 DATA_POSTS
AddComponent3 DATA_SURFACE
end

[DATA]
Databases 3
Data1Name SCALAR_DATA
Data1Type TILED_FLOAT
Data2Name DEPTHS
Data2Type FIXED
Data3Name BATHYMETRY
Data3Type TILED_FLOAT
end

[POINTER]
type WAND_POINTER
width 0.3
length 3.0
color_red 1.0
color_green 0.0
color_blue 0.0
end

[DATA_SURFACE]
type SCALAR_QUAD
colormap temperature.ct
x_length 128
y_length 128
data_values SCALAR_DATA
depth_values DEPTHS
layer12 on
end

[DATA_POSTS]
type SCALAR_POST
x_offset -2.0
y_offset 1.0
colormap temperature.ct
data_values SCALAR_DATA
data_depths DEPTHS
end

[MAINROAM]
```

```
type ROAM_FLOAT
colormap terrain.ct
data BATHYMETRY
data_layer 1
color shared
patchsize 32
viewsquare 9
variance_depth 9
max_nodes 50000
render 0
map_size 65536
gDesiredTris 8000
FrameVariance 50
MinVariance 0.0
AutoLOD 1
end

[BATHYMETRY]
scale -0.005
landmask -1
tilecount 1
cache 1
tile1 PIPS
layers 1
end

[PIPS]
data PIPS/bathymetry.bin
startx 0
starty 0
width 1280
height 720
end

[SCALAR_DATA]
cache 45
tilecount 1
tile1 TEMPERATURE
layers 45
landmask 1.e+30f
end

[TEMPERATURE]
data PIPS/flat/potential_temp.bin
startx 0
starty 0
width 1280
height 720
end

[DEPTHS]
fixed_depths_scale -0.005
layercount 45
depth1     2.5
depth2     7.5
depth3    12.5
depth4    17.5
```

```
depth5    23.015
depth6    29.67
depth7    37.7
depth8    47.385
depth9    59.065
depth10   73.15
depth11   90.14
depth12  110.635
depth13  135.36
depth14  165.19
depth15  201.17
depth16  244.57
depth17  296.925
depth18  360.08
depth19  436.26
depth20  528.15
depth21  639.195
depth22  775
depth23  950
depth24 1150
depth25 1350
depth26 1550
depth27 1750
depth28 1950
depth29 2150
depth30 2350
depth31 2550
depth32 2750
depth33 2950
depth34 3150
depth35 3375
depth36 3625
depth37 3875
depth38 4125
depth39 4375
depth40 4625
depth41 4900
depth42 5200
depth43 5500
depth44 5800
depth45 6100
end
```

## ***Sample NCOM Configuration:***

This configuration displays the bathymetry, one scalar post, 4 layers of data flow glyphs, a vector post, one scalar surface.  It also uses several data sources.

```
[SYSTEM]
startx 900
starty 400
ComponentCount 10
AddComponent1 MAINROAM
AddComponent2 DATA_POSTS
AddComponent3 DATA_FLOW_GLYPHS
AddComponent4 DATA_FLOW_POST
AddComponent5 DATA_SURFACE
end

[DATA]
Databases 6
Data1Name SCALAR_DATA
Data1Type TILED_FLOAT
Data2Name DEPTHS
Data2Type TILED_FLOAT
Data3Name BATHYMETRY
Data3Type TILED_FLOAT
Data4Name VECTOR_U_DATA
Data4Type TILED_FLOAT
Data5Name VECTOR_V_DATA
Data5Type TILED_FLOAT
Data6Name VECTOR_MAG
Data6Type MAGNITUDE_FLOAT
end

[POINTER]
type WAND_POINTER
width 0.3
length 3.0
color_red 1.0
color_green 0.0
color_blue 0.0
end

[DATA_FLOW_GLYPHS]
type VECTOR_GLYPHS
colormap temperature.ct
x_length 64
y_length 64
flowu_values VECTOR_U_DATA
flowv_values VECTOR_V_DATA
depth_values DEPTHS
color_values VECTOR_MAG
layer1 on
layer2 on
layer3 on
layer5 on
end
```

```
[DATA_FLOW_POST]
type VECTOR_POST
colormap temperature.ct
flowu_values VECTOR_U_DATA
flowv_values VECTOR_V_DATA
depth_values DEPTHS
color_values VECTOR_MAG
x_offset 3
y_offset -1
scaling 5.0
end

[DATA_SURFACE]
type SCALAR_QUAD
colormap temperature.ct
x_length 128
y_length 128
data_values SCALAR_DATA
depth_values DEPTHS
layer7 on
end

[DATA_POSTS]
type SCALAR_POST
x_offset 3.0
y_offset 1.0
colormap temperature.ct
data_values SCALAR_DATA
data_depths DEPTHS
end

[MAINROAM]
type ROAM_FLOAT
colormap terrain.ct
data BATHYMETRY
data_layer 1
color shared
patchsize 32
viewsquare 9
variance_depth 9
max_nodes 50000
render 0
map_size 65536
gDesiredTris 8000
FrameVariance 50
MinVariance 0.0
AutoLOD 1
end

[BATHYMETRY]
scale 0.005
landmask 1
tilecount 1
cache 1
tile1 PIPS
layers 1
```

```
flip_x yes
end

[PIPS]
data NCOM/model/bathymetry.bin
startx 0
starty 0
width 1024
height 640
end

[SCALAR_DATA]
cache 20
tilecount 1
flip_x yes
tile1 TEMPERATURE
layers 20
maskop <
landmask -9.9999998e+30
end

[TEMPERATURE]
data NCOM/t3d/t3d_20000313_00.bin
startx 0
starty 0
width 1024
height 640
end

[VECTOR_U_DATA]
cache 20
tilecount 1
tile1 FLOW_U
flip_x yes
layers 20
maskop <
landmask -9.9999998e+30
end

[FLOW_U]
data NCOM/u3d/u3d_20000313_00.bin
startx 0
starty 0
width 1024
height 640
end

[VECTOR_V_DATA]
cache 20
tilecount 1
flip_x yes
tile1 FLOW_V
layers 20
maskop <
landmask -9.9999998e+30
end
```

```
[FLOW_V]
data NCOM/v3d/v3d_20000313_00.bin
startx 0
starty 0
width 1024
height 640
end

[DEPTHS]
cache 20
tilecount 1
flip_x yes
tile1 DEPTHS_TILE
scale 0.005
layers 20
maskop <
landmask -9.9999998e+30
end

[DEPTHS_TILE]
data NCOM/model/zm.bin
startx 0
starty 0
width 1024
height 640
end

[VECTOR_MAG]
data_u VECTOR_U_DATA
data_v VECTOR_V_DATA
startx 0
starty 0
width 1024
height 640
layercount 20
end
```

## References

[1] K. Gaither, R. Moorhead, S. Nations, and D. Fox, "Visualizing Ocean Circulation Models Through Virtual Environments," IEEE Computer Graphics and Applications, Vol. 17, No. 1 , Jan.-Feb. 1997 pp. 16-19

[2] S. Nations, R. Moorhead, K. Gaither, S. Aukstakalnis, R. Vickery, W. C. Couvillion Jr., D. N. Fox, P. Flynn, A. Wallcraft, P. Hogan, O. M. Smedstad, "Interactive Visualization of Ocean Circulation Models," IEEE Visualization '96, Oct. 1996, pp. 429-432.

[3] M.A. Chupa, R.J. Moorhead, S. Nations, A. Johannsen, K. Gaither, and R. Vickery, "ISTV: Interactive Structured Time-varying Visualizer," Proceeding of Oceans 99, Sept. 1999, pp. 938-944.

[4] A. Johannsen and R. J. Moorhead, "Flow Visualization of Basin-Scale Ocean Data," IEEE Visualization '94, Washington, D.C., Oct. 1994, pp. 355-358.