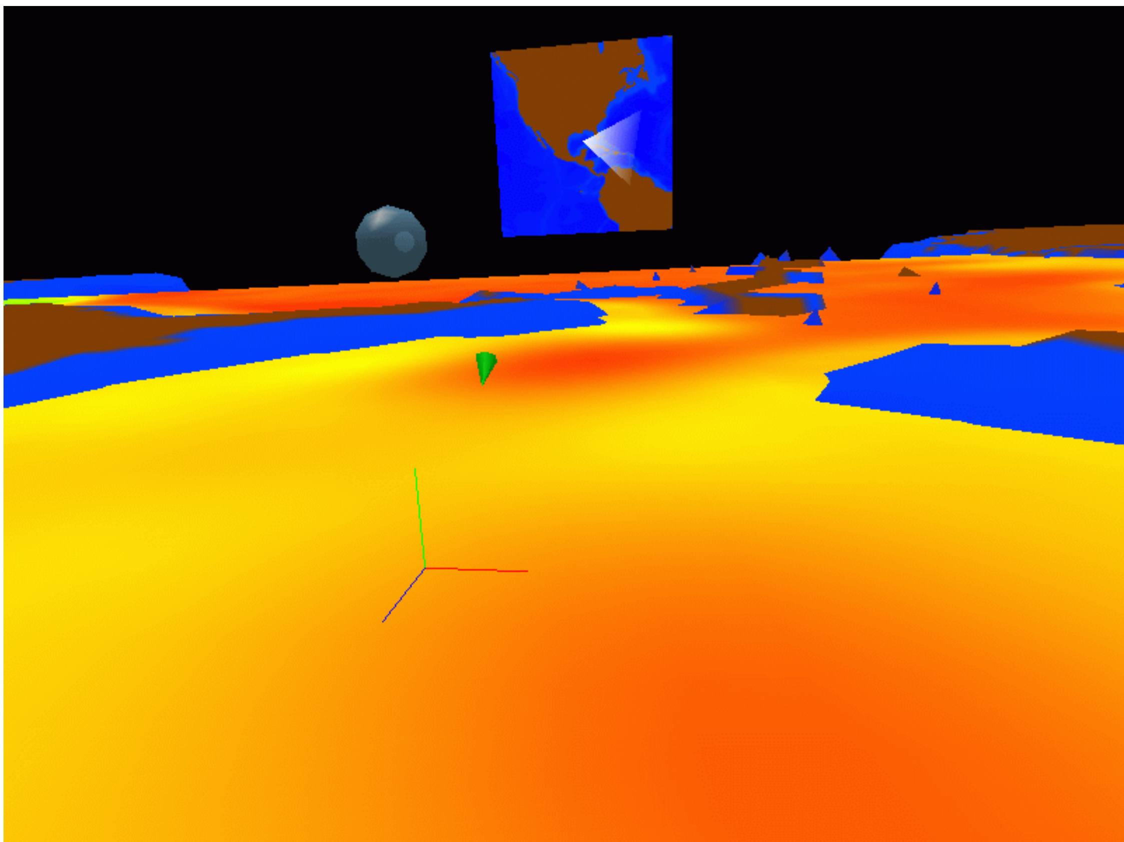
Triton Technical Overview

Randall E. Hand and Robert J. Moorhead II

December 13, 2001

ERC Technical Report #MSSU-COE-ERC-01-14

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



REQUIREMENTS **3**
RUN-TIME REQUIREMENTS 3
COMPILE-TIME REQUIREMENTS 3
FEATURES **3**
RUN-TIME CONFIGURATION 3
DATA FORMATS 3
VISUALIZATION METHODS 4
Scalar Surfaces *4*
Scalar Posts *4*
Vector Posts *5*
Vector Flow Glyph Layer *5*
ROAM Heightfields *5*
DESIGN & IMPLEMENTATION **7**
DATA SOURCES 7
VISUAL COMPONENTS 7

Requirements

Run-Time Requirements

Triton requires (for an SGI):

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

While VRJuggler will run on platforms other than Irix, none have been tested with *Triton*.

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

Features

Run-Time Configuration

Triton supports an text-file configuration setup that can be easily changed with any text editor. This file allows the user to change all aspects of the program from data sources, cache sizes, and rendering optimizations, without having to recompile or edit the program directly. At a later date, these options will also be available from inside the program.

Data Formats

Triton supports the following data formats:

- Raw Binary rectilinear data
- ASCII Fixed Data
- On-The-Fly magnitude Data (calculated from other data sources)
- DataManager Network connection data (under development)

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, a 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 it's first access, and unused data is replaced

later by new data on a later access. Also supported is multi-file tiled data (like GTOPO30), where each tile is cached individually, improving memory consumption.

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.

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. 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 6 (5 Vector Flow Glyphs), and several Scalar Posts scattered around, all while rendering the context bathymetry using a ROAM Heightfield.

Scalar Surfaces

A scalar surface renders a colormapped scalar value for all data points in a square subsection of a single layer, centered around the user. The user defines the following:

- the data source to use for colormapping (D_C)
- the layer to render (L)
- the data source to use for height (D_H)
- the colormap (C)
- and the size (S)

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

Scalar Posts

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

- the data source to use for colormapping (D_C)
- the data source to use for heights (D_H)
- the colormap (C)
- and a location relative to the user (x,y)

The data in D_C at (x,y, each D_H) 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 where the exact layers of D_H are at, a striped post is drawn next to it, with the transitions indicating the exact layers. D_H 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 a value, such as temperature, that can vary significantly from layer to layer.

Vector Posts

A vector post renders a colormapped scalar value for a single point, relative to the user's position, in all layers. Instead of being a flat post through, the data values wrap around the striped post to reflect two more scalar fields defined by the user. The user defines the following:

- the data source to use for colormapping (D_C)
- the data source to use for heights (D_H)
- the two data sources to use for directions (D_U,D_V)
- the colormap (C)
- a location relative to the user (x,y)
- and a magnification factor to use for the direction (M)

The data in D_C at (x,y, each D_H) is rendered as a ribbon penetrating all layers of the data, stretched in the direction ($M*D_U$,$M*D_V$). Smooth shading is used to make the colors smoothly transition. So that the user can tell where the exact layers of D_H are at, a striped post is drawn next to it, with the transitions indicating the exact layers. D_H 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 point in all layers.

Vector Flow Glyph Layer

A vector flow glyph layer renders a colormapped scalar value onto a line indicating a vector value for all points in a square around the user, in a single layers. The user defines the following:

- the data source to use for colormapping (D_C)
- the data source to use for heights (D_H)
- the two data sources to use for directions (D_U,D_V)
- the colormap (C)
- a Size (S)
- a Layer to enable (L)

For each point in a square (S) around the user in the layer (L), a line is drawn from the grid point in the direction of (D_U,D_V). The line is colormapped by the values in D_C. D_C is usually a On-The-Fly-calculated magnitude data source, calculating the magnitude of the vector (D_U,D_V) at each point.

ROAM Heightfields

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 hit. By changing this upper bound, you can ensure that the framerate stays high no matter how complex the geometry is.

The user defines the following: (A short list of the most important values)

- the data source to use for colormapping (D_C)
- the data source to use for heights (D_H)
- the colormap (C)
- A Patch Size (P_s)
- A Starting Variance (V)
- A Maximum number of Triangles (T)
- Among several others

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 direction.

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. This implementation 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.

This method is usually only used to render the bathymetry for contextual information.

Design & Implementation

Data Sources

Triton takes a slightly different approach to data loaders than most other programs. A DataManager object is instantiated at program startup, and configured from the configuration file. It maintains a list of all databases, along with what type of database they are. It instantiates DataSource objects, which are the base object of all the file loaders. Each DataSource is required to implement a few basic functions such as LoadData, LoadPoint, SetLayer, etc. Each DataSource is also given a text name. Each DataSource internally caches frequently and recently used data to improve performance.

Because of this design, a Visualization method designed to use a DataSource instead of a global array or such, automatically works with any file loader without requiring re-compilation or editing. Unfortunately, this imposes the restriction of rectilinear file formats.

Visual Components

Triton uses a similar structure for it's visual components. On startup, it reads a list of Visual Components from the configuration file, and instantiates and configures a VisualComponent object for each one. The VisualComponent object is the base object for all of the Visualization methods, and has a few simple functions to override such as PreFrame, PostFrame, PostRender, Render, etc. When the VisualComponent is created, it's passed a reference to the DataManager where it can obtain it's data through the text data names. This means a visualization method doesn't need to know how to access the data, just how to render the data. It also receives a reference to the UserData object, which contains information like the user's current location and orientation. This is used for interacting with the environment.

Because of this design, it's very simple to add new visualization methods into the system. All of the data-loader work has been removed to an entirely separate code base, meaning the programmer can focus on OpenGL rendering and algorithms for optimizations. Also, Visualization methods can be designed in template classes allowing for a single class that can load floating point, integer, or byte data. Then they can be encapsulated a VisualComponent object for each data type.

This is the case for ROAM Heightfields. The ROAM Heightfield class is actually a template class ROAMLandscape. But through the VisualComponent Manager, I have added VCROAMFloat, VCROAMInt, and VCROAMByte with a minimum of extra code.