A texture-based framework for improving CFD data visualization in a virtual environment

by

Gerrick O'Ron Bivins

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Mechanical Engineering

Program of Study Committee:
Kenneth Bryden, Major Professor
James Oliver
Jerald Vogel

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the master's thesis of

Gerrick O'Ron Bivins

has met the thesis requirements of Iowa State University

Major Professor

For the Major Program

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

In the field of computational fluid dynamics (CFD) accurate representations of fluid phenomena can be simulated but require large amounts of data to represent the flow domain. Most datasets generated from a CFD simulation can be coarse, ~10,000 nodes or cells, or very fine with node counts on the order of 1,000,000. A typical dataset solution can also contain multiple solutions for each node, pertaining to various properties of the flow at a particular node. Scalar properties such as density, temperature, pressure, and velocity magnitude are properties that are typically calculated and stored in a dataset solution. Solutions are not limited to just scalar properties. Vector quantities, such as velocity, are also often calculated and stored for a CFD simulation. Accessing all of this data efficiently during runtime is a key problem for visualization in an interactive application.

Understanding simulation solutions requires a post-processing tool to convert the data into something more meaningful. Ideally, the application would present an interactive visual representation of the numerical data for any dataset that was simulated while maintaining the accuracy of the calculated solution. Most CFD applications currently sacrifice interactivity for accuracy, yielding highly detailed flow descriptions but limiting interaction for investigating the field.

1.1 Visualization of computational fluid dynamics data

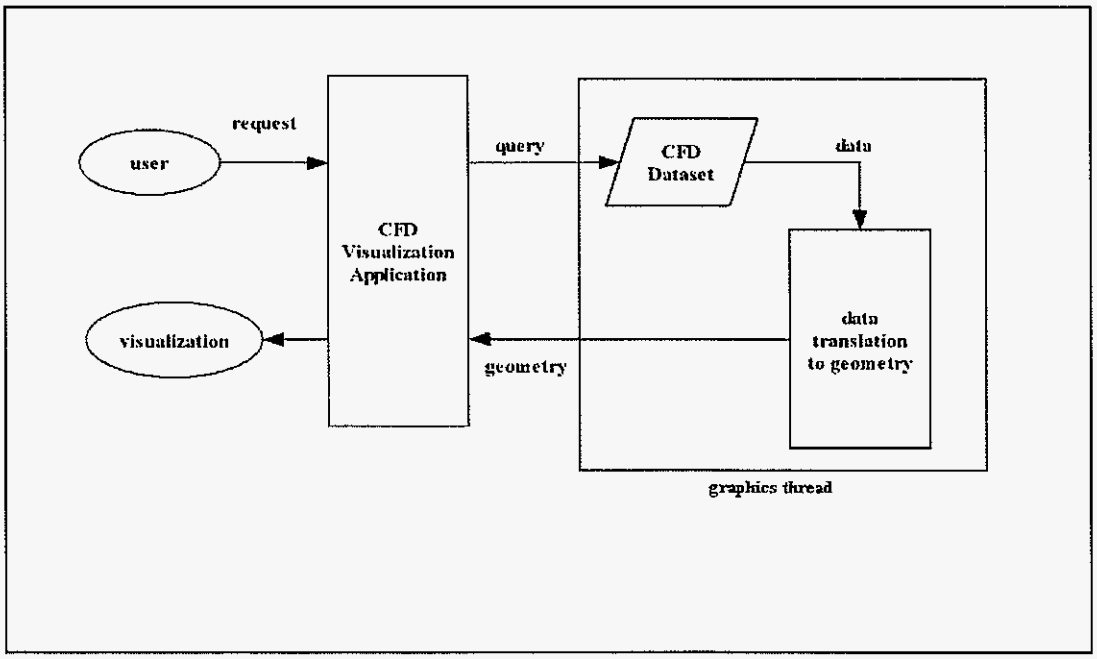
CFD post-processing generally involves two steps. First the mesh data is translated into a format that is efficiently accessible during runtime. At runtime, the dataset is queried to generate a visualization representing an engineering analysis technique. Analysis methods, such as contour planes slicing through a particular scalar property field or directional vector glyphs, are common techniques for investigating CFD data sets. Streamlines and particle

traces are also common methods for analysis of vector fields. Each of these methods gives a visual queue of a particular flow property, such as direction or magnitude.

1.2 Pipeline for generating a graphical representation

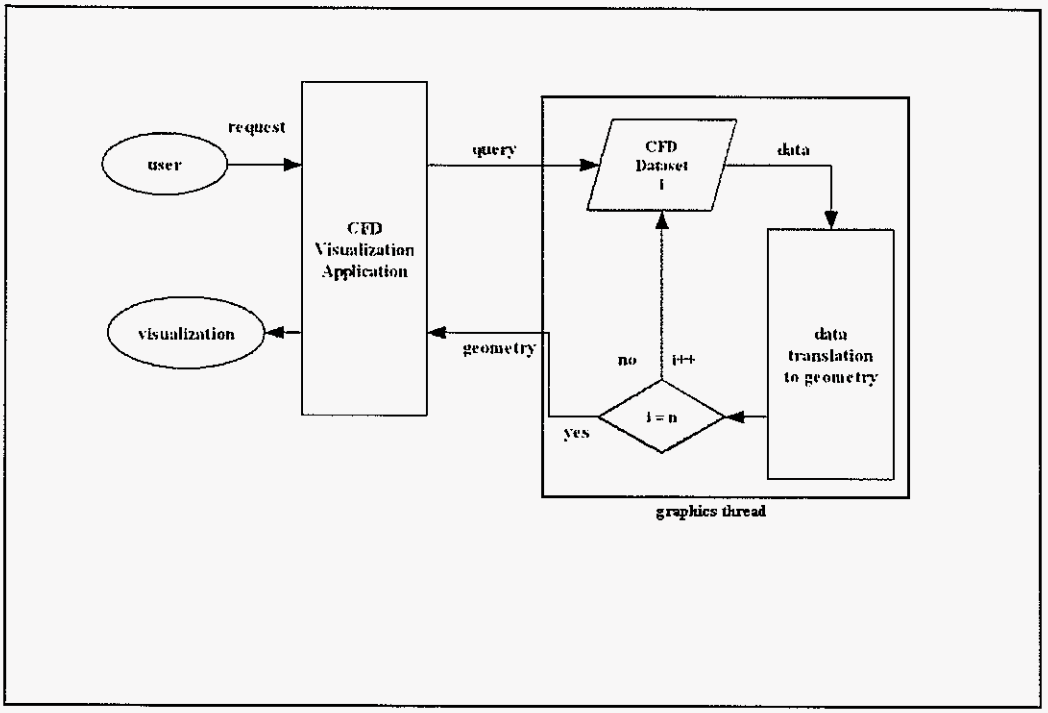
Generating graphics for a particular analysis technique relies heavily on the characteristics of the dataset being investigated. Efficient access to the dataset is required to support user interactivity. However, this is also dependent on the complexity of the dataset. Various algorithms for efficient access of large data structures have proven to be effective in reducing query time but only to a point. In an application where user interaction is a priority, it would be ideal to reduce or even eliminate the need to access the raw dataset during runtime to generate the visualization.

Figure 1.1 describes a general visualization pipeline for a CFD dataset. The process begins with the user requesting a specific visualization technique such as a contour plane. The request contains the minimal information required to generate the appropriate visuals, such as the location (percentage of the dataset bounding volume) and orientation (parallel to a Cartesian axis). This request is sent to a CFD visualization application. The application then initializes a thread to process the input user information by querying the resident dataset. The query returns information necessary for creating a graphical representation of the visualization technique. This information is then passed on to an application programming interface (API) to create the actual graphics that are to be displayed. The application waits for the thread to return the graphical representation for display. This entire process is repeated for each technique the user requests.



The issue of dataset access becomes more apparent if the user desires to investigate a time-varying dataset. Time-varying, or transient, datasets add another level of complexity to the visualization process. As is apparent from Figure 1.2, the application becomes responsible for repeating the "query-to-graphics" thread ***n*** times, where ***n*** corresponds to the number of transient time steps. As with the steady state case, the application waits for the thread to complete before sending the graphics to the display. Accordingly, the application is also responsible for setting the resident dataset at each time step. Generating graphics in time to display concurrently in the simulation quickly becomes a daunting task for the application. Traditionally most applications have supported transient visualization by processing the graphics and then running an animation or "movie type" visualization of flow phenomena. Investigative techniques such as streamlines and particle traces of transient datasets are often

unsupported in real time for transient datasets.



1.3 Recent advancements in computer graphics and CFD visualization

Current visualization research is now heading toward analysis of the data in a more efficient manner. The advancements in the technology of modern graphics processor units (GPUs) combined with their availability on consumer-level personal computers have led to texture-based methods gaining popularity. Texture-based methods are based on storing information, such as data from a CFD simulation, in a texture format and using the textures to display the data at runtime. Basic volume rendering, as described in (Wilson, 1994), of three-dimensional textures can be used to visualize various types of three-dimensional data.

The computer graphics industry has long had exposure to the GPU and its advantages. Programming at the GPU level allows the developer to have more control over special effects, such as lighting, by allowing the developer to specify operations at a per-vertex or per-fragment level.

Vertex programs, or vertex shaders, run on the vertex processor of a GPU and calculate data for each vertex of a graphics primitive. This data is then passed to the fragment processor. A fragment program is basically responsible for calculating the color for each fragment between the vertices of a primitive. Information such as color, texture coordinates, etc. is interpolated between vertices on the fragment processor. Exposure of the vertex and fragment processors to the developer gives more control over what is seen.

Because the GPU is much more efficient at processing graphics than the central processing unit (CPU), more realistic effects, such as bump mapping, and higher quality materials are achievable in real time by moving calculations from software to the graphics hardware. Most effects pre-calculate pertinent data for the desired effect, such as a normal map, and store the information in a texture. The pre-calculated data is then accessed in a fragment program, via the texture, on the GPU. The information is then used in the appropriate calculations to return a color for each fragment and therefore each pixel.

As stated previously, exposure of this functionality to the general developer has led to several algorithms that exploit the programmable functionality of the GPU. Textures, combined with textures, real-time interactive visualizations of CFD data can be achieved. Texture advection schemes such as three-dimensional image-based flow visualization (3D-IBFV), (Telea, 2003) and GPU-based three-dimensional texture advection (Weiskopf, 2004) take advantage of graphics hardware to produce interactive visualizations of large vector datasets through dye and particle injection algorithms (Laramee, 2004).

The issue of transient data handling is also resolved by using textures. Texture data representing the field can be updated using hardware acceleration. Three-dimensional graphics application programming interfaces (APIs), such as OpenGL, allow for fast

updating of texture data. This allows texture-based algorithms to handle data similarly for steady-state and transient data cases.

Integration of texture-based techniques is the next logical step for advancement of a CFD visualization application. Runtime issues of data access would be reduced by storing solution sets in three-dimensional textures. Basing an application on textures would allow the application to integrate and expand on current visualization research. A texture-based application would also improve interactivity of a CFD application, by removing unnecessary runtime dependencies for generating the visualization.

This research is organized in the following manner. Chapter Two describes the CFD visualization application chosen for integration, Virtual Engineering Suite (VE-Suite), and its current visualization process. Chapter Three describes the proposed framework and the necessary components for integrating textures with VE-Suite. Possible methods for implementing scalar and vector data analysis are discussed. Chapter Four discusses the results of the proposed implementation and shows some visual comparisons of the current visualizations and the proposed method with some applied texture-based algorithms. Chapter Five summarizes the conclusions and contains a discussion of future work.

Chapter 2: Virtual engineering software: VE-Suite

Bryden (2004) describes virtual engineering as "a user-centered process that provides access to a collaborative framework that integrates all of the models, data, and decision-support tools needed to make an *engineering decision*. The goal of virtual engineering is to develop a decision making environment that provides a first-person, immersive perspective enabling the user to interact with the engineered system in a natural way and provides the user with a wide range of accessible tools."

2.1 VE-Suite

Iowa State University's Complex Systems Virtual Engineering group, which is directed by Dr. Kenneth Mark Bryden, is developing an open source tool set to incorporate the virtual engineering process, VE-Suite. Virtual Engineering Suite (VE-Suite) is an extensible set of software tools that, combined with vrJuggler, allows a developer to easily create a virtual engineering application. vrJuggler, an open source virtual reality application development framework developed by Dr. Carolina Cruz-Neira and her research group at Iowa State University, simplifies the interface and interaction with the virtual environment for application developers.

Figure 2.1 describes the key components of VE-Suite. The toolset includes VE-Conductor, VE-Xplorer and VE-CE. VE-Conductor is the graphical user interface (GUI) that allows the user to control the virtual environment and interact with the data in the virtual environment. VE-CE, the computational engine, allows integration and data passing of experimental data streams, numerical models, algebraic equations, or any other form of data. Finally, VE-Xplorer handles the visualization and manipulation of the data through three-dimensional graphics.

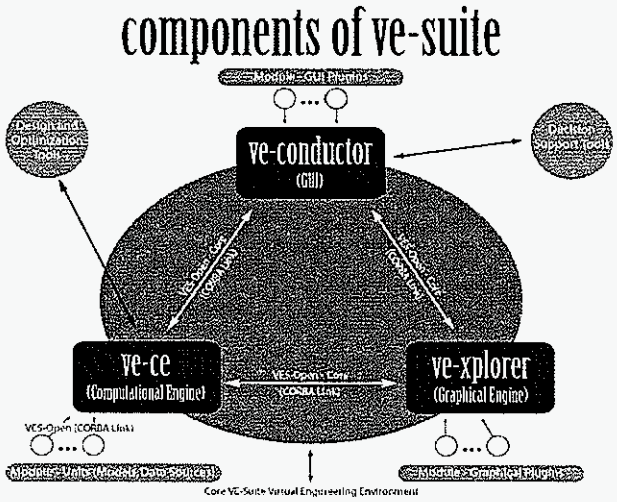
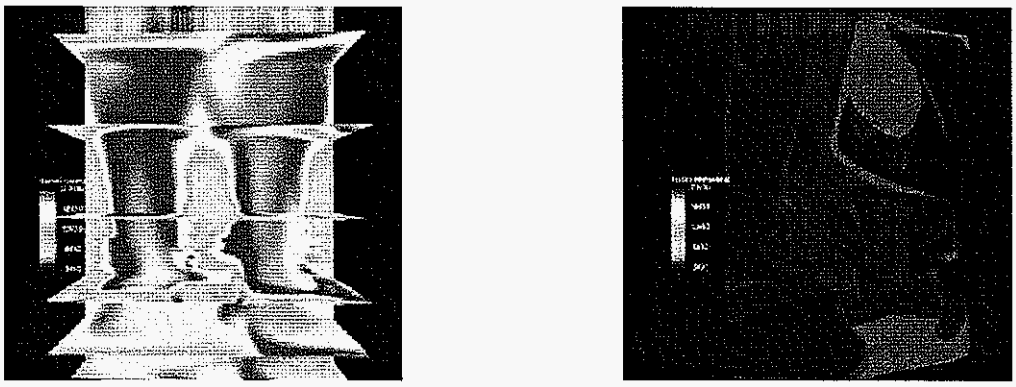


Figure 2.1 Core components of VE-Suite

2.2 CFD analysis methods in VE-Suite

Several common engineering analysis techniques are currently supported by VE-Suite. Scalar data investigation techniques typically include color contour/cutting planes at user-specified locations in the dataset. The colors correspond to the magnitude of the specific flow property and are based on a linear red-to-blue lookup table. Higher magnitudes are red while lower values are blue. Iso-surfaces with approximately the same magnitude for a particular property, are also supported for scalar data analysis in VE-Suite.

Figure 2.2 shows an example of VE-Suite's scalar capabilities. Figure 2.2a is a screen shot of contour planes of temperatures in a furnace at various locations along the y and z axes. Figure 2.2b is an iso-surface representation of the temperature of the same furnace dataset.

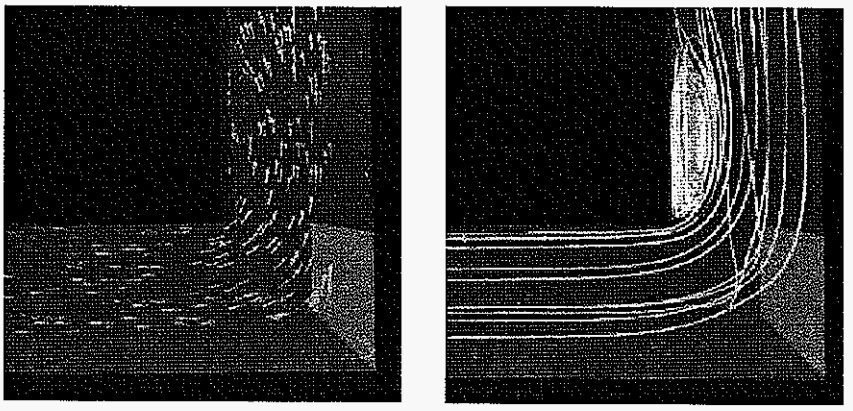


(a) Contour Representation **(b) Iso-surface Representation**

Figure 2.2 Scalar data visualization of temperature of a furnace

Vector data can be analyzed in numerous ways in VE-Suite. A simple yet insightful technique that is common in CFD visualization is the vector glyph. A vector glyph can be described as a geometrical representation, usually in the shape of an arrow head or triangle, which is oriented in the direction of the local flow property. Scalar properties can also be applied to glyphs by coloring and/or sizing the glyph geometry by the magnitude. This method is useful for giving the user a visual sense of the direction of the flow field. Streamline generation is also useful for visualizing direction of the field and is supported in VE-Suite. Characteristics of the field are visually evident.

Figure 2.3 shows an example of VE-Suite's vector visualization techniques. A vector glyph representation of the flow field (a) and streamlines generated (b) in the furnace are shown.



(a) Vector Field **(b) Streamlines**

Figure 2.3 Vector data visualization

2.3 Insight from numbers: VE-Xplorer

As stated earlier, VE-Xplorer handles the creation of graphical visualization for the numerical data passed into the application from the user. The purpose of the visualization is to provide meaningful insight to the large amounts of data produced from a CFD simulation. For the visualization to be effective, it must convey a concise summary of the numerical information in a timely fashion, without sacrificing the quality of the simulated solution. VE-Xplorer harnesses the power of scene graph APIs, such as OpenSceneGraph and Performer, for efficient organization of graphical data.

Scene graphs are "tree" structures, as defined in computer science, that organize data in a hierarchical structure of nodes. The nodes in scene graphs are related by a parent-child relationship. Each node in the scene graph can have zero or more children. A node with zero parent nodes is called the "root", while a node with zero children is called a "leaf." A child node may inherit properties from its parent node or contain its own properties.

Figure 2.4 shows a generic scene graph as described above. Each node in the graph is labeled according to its relationship to other nodes. Each node in the figure is named with a capital letter. To show its relationship to other nodes in the tree, a lower case letter is included. The lowercase letter represents the name of the node's "parent." All nodes in the figure are represented by a circle in the graph with two exceptions. The root node is labeled "A" and designated by a trapezoid. A node that is designated by a triangle is a "leaf" node. Since the root node has no parent node, its label does not contain a lowercase letter. Node "A" has two children, nodes "B" and "C". They are accordingly labeled "aB" and "aC." Node "B" has three children, which are leaf nodes. These are denoted by a lowercase "b".

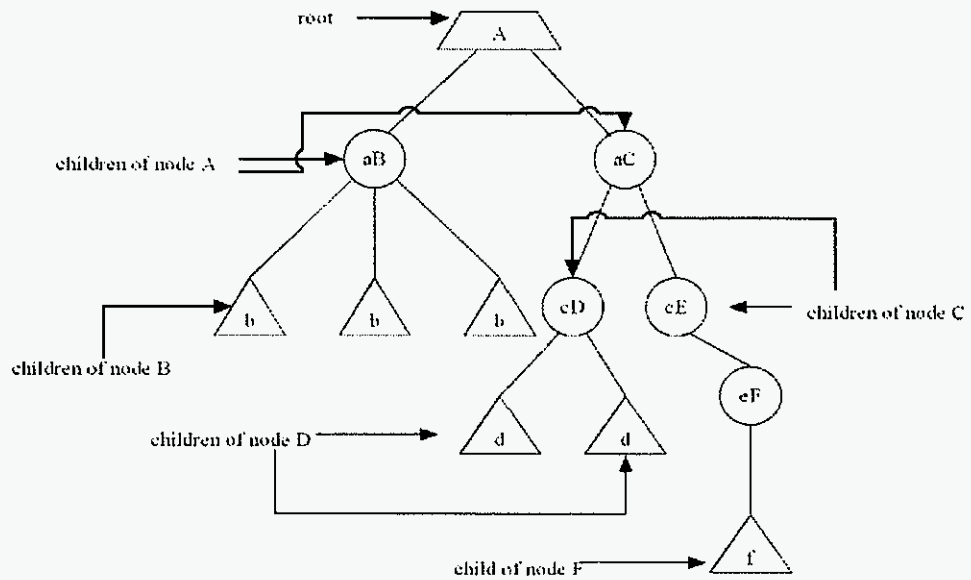


Figure 2.4 Generic scene graph

2.3.1 Initializing the scene graph from a parameter file

VE-Xplorer builds a scene graph as the dataset and its related geometry is read from a disk. VE-Xplorer begins by reading a parameter file that describes the models that are going to be visualized in the virtual environment. Information such as the location of the dataset, geometry files for the model, and geometric transformations are stored here. The information is formatted into "blocks" within the parameter file. Each block begins with a predefined *object ID* that identifies the type of object described by the block. Valid object IDs are defined in Table 2.1.

Object Ids	Description
0	Global transformation matrix
1	Scalar bar positioning properties
5	Image file (.bmp)
8	Visualization ToolKit file describing a steady state dataset
9	Geometry file
10	Transient dataset parameters
11	Sound files
12	Animated Image description
13	Indian Hills Community Colledg file description
14	Quaternion camera description

Table 2.1 Parameter file object IDs and descriptions

Depending on the objects specified in the parameter file, VE-Xplorer builds the initial scene graph and loads the data needed for runtime access. The initial scene graph is a recreation of the modeling environment and simulation scene in VR, with only geometry objects initially on the graph.

The scene graph created by VE-Xplorer consists of two main "branches," one containing geometry relating to the dataset and the other containing graphics for the visualization technique. During runtime, each of these branches is visited and manipulated based on user input.

2.4 Visualization pipeline

The process of converting a user request into a visual representation can be broken into several key components:

1. Command interpretation → Determine the appropriate visualization technique based on the user request.
2. Dataset querying → Send the appropriate information to the dataset-managing API to generate data for the requested visualization technique.
3. Graphical/Geometric representation → Convert the visualization technique information returned from the dataset managing API into a geometric representation.
4. Scene graph manipulation → Add/remove the appropriate nodes of the current scene graph to update the visualization with the requested visualization technique's geometric representation.

2.4.1 Scene graph access

After the initial scene graph is created, the underlying scene graph API traverses the newly created scene graph and then displays the resulting three-dimensional representation. This occurs for each "frame," which corresponds to a single traversal of the scene graph. During traversal, interaction with the scene graph is available through C++ functions that occur at critical times during the traversal:

1. **Pre-frame**: Function called before traversal of the root node and its children.

2. **Intra-frame**: Function called during traversal of the root node and its children.
3. **Post-frame**: Function called after traversal of the global root and its children.

VE-Explorer does the brunt of its work (scene graph management) in the *pre-frame* function. During *pre-frame*, a command queue is processed by VE-Xplorer. This command queue is simply a list of commands that are generated by user requests from VE-Conductor. As the user selects options, VE-Conductor sends the associated command to VE-Xplorer and it is added to the command queue for processing. Processing the command queue simply tells VE-Xplorer how to manipulate the scene graph.

2.4.2 Command queue processing: Handlers

During the *pre-frame* stage of scene graph traversal, VE-Xplorer checks the command queue for any commands that may be available. If present, the command is passed to a set of handlers for processing. The job of the each handler is to check the current command and, if it applies, to process the command accordingly, specific in its task.

1. **Environment Handler**: Environment interaction commands such as navigation.
2. **Model Handler**: Data model interaction commands such as setting the active dataset.
3. **Visualization Handler:** Visualization technique request commands.

The visualization pipeline is embedded in the visualization handler. Commands relative to this handler correspond to user requests to investigate various properties of the flow field dataset.

2.4.3 Creating viable representations for analysis: Visualization Handler

The visualization handler is responsible for processing user requests to update the visualization. As stated earlier, this process is defined by the following steps:

1. Query the active dataset to produce visualization technique data.
2. Process the visualization technique data to create a geometric representation.
3. Add the new geometry to the scene graph, removing old geometry if necessary.

This process is a pipeline that is dependent on Visualization Toolkit (VTK). VTK is "an open source, freely available software system for 3D computer graphics, image processing and visualization" (VTK, 2005). VTK can produce various visualizations for scalar and vector data. VE-Xplorer currently uses VTK to produce all of its visualization options, such as scalar contours and iso-surfaces, vector glyphs, streamlines, and particle traces.

The datasets used during runtime are a VTK representation of the dataset. With the dataset loaded into memory at runtime as the user selects a visualization option, the active VTK dataset is queried. This query is passed to VTK to generate a VTK representation of the selected option, which is defined in a VTK structure called an "actor" or vtkActor. This "actor" contains all the information needed to create the queried visualization option graphically.

Normally, the next step in this process would be to send the created "actor" through VTK's graphical pipeline to create the visualization. However, VTK is not designed for visualization in a VE-Xplorer-type application. Requirements such as "rendering to multiple channels (i.e. multi-headed displays)" (Rajlich, 2005) or constructing scene graphs are not

directly handled by VTK's graphical pipeline. Such things can be done using a scene graph API such as Silicon Graphics (SGI) Performer or an open source scene graph API such as OpenSceneGraph. These APIs handle virtual environment requirements efficiently and vrJuggler supports both.

To overcome this shortcoming of VTK, Paul Rajlich developed vtkActorToPF(Rajlich,), which simply takes a vtkActor and translates it to the comparable Performer geometry node representation. There is also an OpenSceneGraph version that translates a vtkActor to an OpenSceneGraph geometric node representation. VE-Xplorer uses these two utility libraries to create its graphical representations of user-queried data.

Figure 2.5 shows VE-Xplorer's visualization pipeline for a steady state dataset. vtkActorTo* refers to vtkActorToPF for a Performer based scene graph and vtkActorToOSG for an OpenSceneGraph-based scene graph.

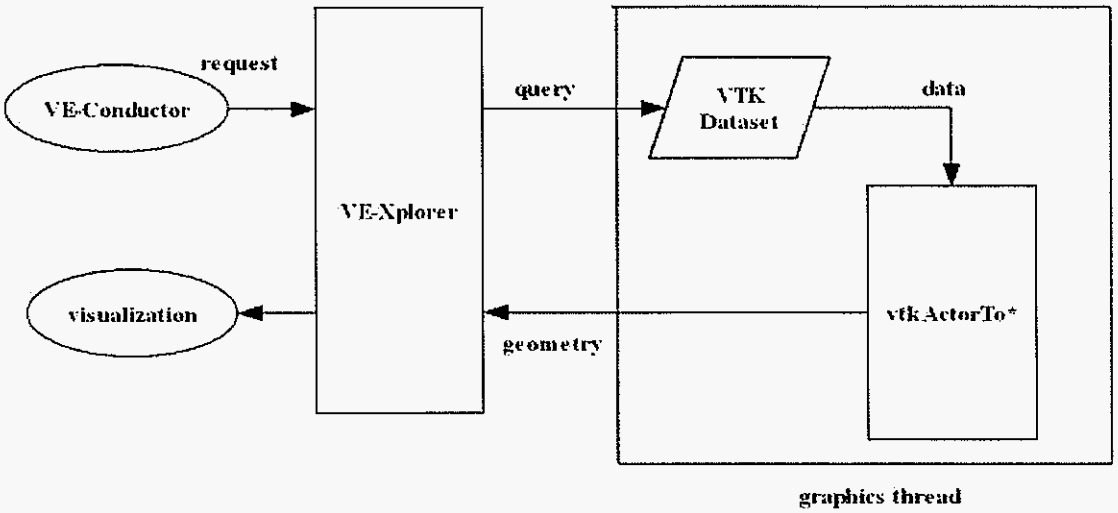


Figure 2.5 Visualization pipeline of VE-Xplorer for a steady-state dataset

The transient pipeline has a more general description. Although it is simple to understand, a few issues are revealed upon closer inspection. These issues are discussed below. Figure 2.6 displays VE-Xplorer's visualization pipeline for a transient dataset. As shown, it is very similar to the steady-state pipeline. In fact, the steady-state pipeline is a special case of the transient pipeline where ***n***, of time step number and hence number of datasets, is equal to one. In the figure, the graphics thread is repeated ***n*** times before the graphics are added to the scene graph.

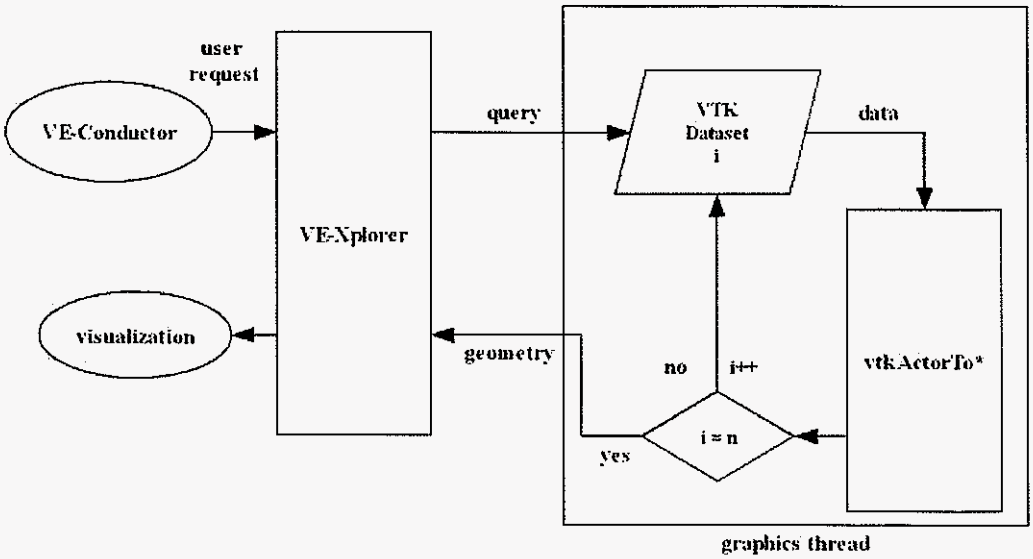


Figure 2.6 Visualization pipeline of VE-Xplorer for a transient dataset

2.5 Issues with the VE-Xplorer visualization pipeline

A couple of issues become evident during investigation of transient datasets. The first issue is related to the datasets themselves. As mentioned earlier, datasets and the information stored in them are large. For example, for a single time step, a 1,000,000 node dataset (100x100x100 grid) that contains a single scalar and a single vector, such as density and velocity, would require 4 bytes per node for a density of "float precision" values and 4 bytes * 3 values per node for a velocity vector of "float precision." That requires 16,000,000 bytes or ~ 16 megabytes of random access memory (RAM) for the dataset. This is a rare case, as typical datasets store "double precision" solutions requiring 8 bytes for each value rather than 4, thus doubling the required RAM to 32 megabytes.

These however are minimum requirements. For the dataset to be efficiently accessed during runtime, the actual C++ class created to hold the dataset in memory is larger. Extra

information such as the names for all the scalars and vectors in the dataset, connectivity lists for the grid structure, and bounding box information is allocated for each solution dataset.

Obviously, more solutions are stored on the dataset solution, along with larger grids, because the storing solution only requires hard drive space and they can be computed in reasonable amounts of time. As additional solutions are added to our solution dataset, the RAM required to efficiently access the data grows. But this is only a part of the RAM issue. For a transient dataset, this requirement must be multiplied by at least n, corresponding to each time step. If the dataset grid changes size with time, which is possible for CFD solutions due to changing of the geometry of the solution, then the RAM requirements will fluctuate accordingly. If the application requires more memory than is available in RAM, performance drops significantly, thus affecting the application's usefulness. These issues are limiting factors for most CFD applications' ability to investigate transient datasets effectively.

Because VE-Xplorer's visualization pipeline uses VTK, extra dependencies are introduced. Both the dependency on VTK's API to query datasets for data and the dependency to generate the scene graph representations result in a "delayed reaction" for datasets that are of average to large sizes. These "delayed reactions" can be interpreted as nothing happening in the display for a few moments (time varies with dataset size) and loss of response from VE-Conductor. If the application loses its interactive capability, its effectiveness is also lost.

2.6 CFD datasets as textures

A possible solution for the issue mentioned previously would be to store the dataset in a format that requires little or no access during runtime to visualize. This would remove the

dependency on external APIs to do calculations on large datasets in software, which can become slow as the complexity of the dataset increases. The format should also be small enough to handle transient datasets efficiently without losing accuracy. Three-dimensional textures, provide such a format.

2.6.1 Texture formats and data types

In computer graphics, a texture is basically an array of data. Textures are generally equivalent to two-dimensional images but can be one- or three-dimensional. A two-dimensional texture can be applied to geometry to provide a more realistic look at the object without rendering an unnecessary amount of graphics primitives. For instance, a brick wall can be drawn using a single quadrilateral with a brick texture applied as opposed to a large number of red colored rectangles for each brick in the wall.

Elements of data in a texture are called *texels*. These are similar to pixels in that they have a format and a type. The format describes how many data values are stored for each texel. For instance, a texture of the format RGBA stores four values per texel. Table 2.2 (Woo, 1999) shows valid values and the definitions for texel (usually a new term is only italicized the first time it appears) formats.

COLOR_INDEX	a single color index
RGB	red, green, blue
RGBA	red, green, blue, alpha
BGR	blue, green, red
BGRA	blue, green, red, alpha
ALPHA	a single alpha component
LUMINANCE	a single luminance component
LUMINANCE_ALPHA	luminance, alpha

Table 2.2 Common texel formats and definitions

For each texel, a data type must also be specified. The data type specifies how much data is stored per component. This, combined with the format, determines how much memory the texture will use on the GPU. Valid texel data types are similar to data types in C++. Table 2.3 (Woo, 1999) lists some common data types and their meanings.

UNSIGNED_BYTE	unsigned 8-bit integer
BYTE	signed 8-bit integer
INT	signed 32-bit integer
FLOAT	single-precision floating point
UNSIGNED_INT	unsigned 32-bit integer

Table 2.3 Common texel types and definitions

2.6.2 Other uses for textures

As GPUs become more efficient, developers are finding more uses for textures. A common usage is to store pre-calculated information that can later be accessed in a shader. An appropriate type and format are chosen based on the accuracy desired for the application.

For a CFD application, a three-dimensional texture can be used to store the property data. A scalar texture, for example, could be stored in a texture format of RGBA. The actual values could be preprocessed to map between the minimum and maximum magnitude values. If the texture data type is UNSIGNED_BYTE, valid values for texture data are [0,255]; therefore the original data must be transformed so that the minimum magnitude maps to 0 and the maximum magnitude maps to 255. Equation 1 is an equation for such a mapping, where d_q is the calculated data of type UNSIGNED_BYTE and d is the original scalar data value. The variables *min* and *max* correspond to the minimum and maximum magnitude scalar value data values.

$$d_q = 255 * (d - \min)/(\max - \min) \qquad (1)$$

The newly created values are then used as input to a user-defined color look-up table to get the three values for each of the color components. The alpha component can be constant or it can be determined by a similar look-up based on a user-defined function for opacity. The resulting RGBA vales are stored in the texture and can be directly visualized in a standard volume visualization algorithm (Wilson, 1994).

2.7 Volume visualization of three-dimensional textures

Visualizing volumetric data can be approached in several ways. Researchers introduced a common approach that harnesses texture hardware on GPUs to accelerate the visualization process to interactive speeds (Cabral, 1994). The texture-based algorithm was shown to produce rendering speeds 100 times faster than a CPU-based algorithm. The algorithm renders polygons whose normal vector is parallel to the view direction textured

with the volumetric data, basically "slicing" through it. As the "slices" are drawn back-to-front, alpha blending is calculated to accumulate the final image in the frame buffer. Tri-linear interpolation is performed on the GPU for computing visual data in the frame buffer between "slices." The result is a transparent image that represents the volume data. Figure 2.7 shows a volume visualization of a scalar dataset representing velocity magnitude of a celestial formation. A red-blue look-up table was used to calculate the color values for the scalar data, red relating to high magnitude values and blue correlating to low magnitude values. A constant alpha of .2 is used.

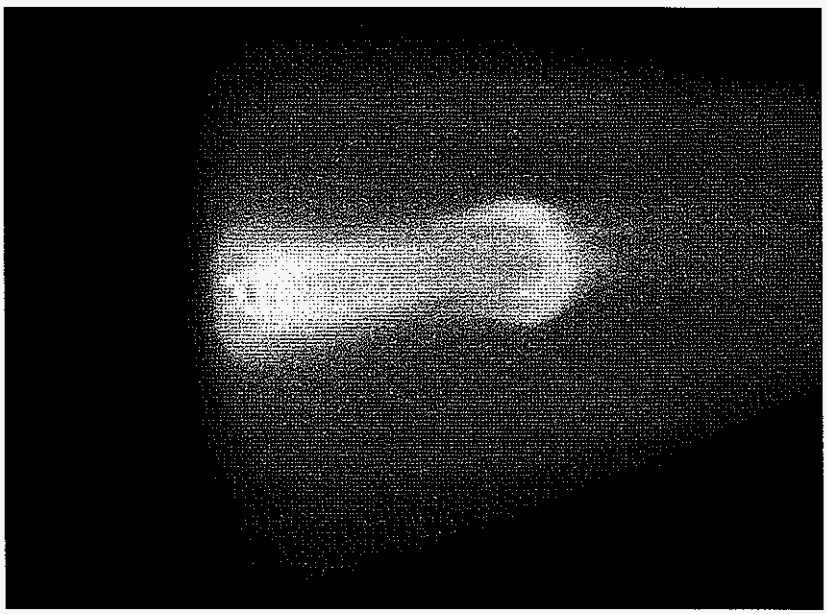


Figure 2.7 Texture-based volume visualization of a celestial formation

Chapter 3: A texture-based framework for VE-Xplorer

Developing a texture-based framework for VE-Xplorer requires the following:

1. The dataset must be converted into a suitable texture format.
2. The texture dataset must be managed efficiently during runtime.
3. The visualization of the dataset should be represented as a scene graph node.
4. Analysis techniques for the texture-based representation of the dataset should be comparable to current techniques.

With these requirements in mind, a proposed texture-based framework is described. The purpose of the framework is to complement and possibly enhance the current framework.

3.1 Representing the CFD dataset as a texture: Preprocessor

Before discussing the proposed framework, the process of generating the texture files representing the CFD data is described. The approach currently implemented is strictly used as a "first pass" effort and is by no means the most efficient or most effective. It should be noted that the implementation of the preprocessor is independent of the visualization of the data.

3.1.1 Texture file

The first step toward developing a texture-based framework for VE-Xplorer is to represent the CFD as a texture. The approach taken here is to convert the original dataset into a texture file that can be loaded during initialization of VE-Xplorer. The file contains some pertinent information for building and managing the texture at runtime:

1. Field type ➔ This is either "*s*" (scalar) or "*v*" (vector field).

2. Data range ➔ This is the valid range of values for the field. If the field is a vector, these values correspond to the range of magnitudes of the vector field.
3. Bounding box ➔ These values represent the bounding box of the CFD dataset.
4. Texture resolution ➔ These values are the x/y/z resolution values of the texture.
5. Data values ➔ Stored data values for the field.

If the field is a scalar property, one float precision value is written to the file per texel. A vector field stores four float precision values per texel (x,y,z,magnitude). Vector values (x,y,z) are normalized so that values are in the range [-1.0,1.0]. The (x,y,z) values are then transformed to map the values between [0.0,255.0] [1]. These values are written to file along with the magnitude.

3.1.2 Preprocessor algorithm

The creation of the texture file is not as straightforward as mentioned above. There are some restrictions to the definition of a valid texture that limit how information can be stored in a texel. First, a texture is simply an array of texels, a "brick-type" structure. Most CFD datasets are unstructured grids and contain various cell shapes and sizes. Textures themselves do not have a concept of shape; they are simply data. The general correlation between a CFD dataset and a texture is similar to mapping an unstructured dataset to a structured dataset. Ideally, a transformation could be found to map the unstructured data into texture space, similar to grid generation techniques that transform the original grid into computational space. This type of transformation is specific to each dataset and unless the mapping is known beforehand, requires a new transformation for each CFD dataset.

---

[1] Depending on the format, texture data should be zero or positive. The quantizing equation used is : $d_q = 255*((d_o + 1)*.5)$

The approach taken here is a simple (not necessarily the best) resampling of the original field at equidistant points within the bounding volume of the dataset. This removes the requirement calculating a transformation.

The algorithm first reads the dataset and collects the names and number of each scalar property and vector field in the dataset. Then an *octree* of the dataset is constructed to quickly to locate the cell within the dataset that is closest to the re-sampled point. After the octree is created, a true/false structure representing the sampled data point location relative to the valid dataset domain is constructed. Each re-sampled point that is in or on a cell boundary within the dataset boundary is flagged as "true." If the point is outside of the dataset domain, the point is marked as "false." This structure is later used to determine if data should be resampled at the point.

Figure 3.1 describes the algorithm for creating the true/false structure. The octree is constructed and the bounding box information is obtained from the dataset. Spacing for each sample point is calculated based on the requested texture size and the corresponding bounding box dimension for the dataset. The point for the lower corner of the bounding box is set as the starting texel.

Once the true/false structure is created, the dataset is then traversed and sampled for each scalar and vector of the dataset, using the true/false structure to effectively neglect interpolating values for points marked as "false" texels. Data for texels that are marked as "false" is set equal to zero. If a texel is marked as "true," the cell containing our sample point is located. The point is then evaluated based on its location within the cell. The weights calculated determine how much the data at each cell vertex affects the sampled point. The data stored in the texel is the sum of the weighted data values at each vertex, as

shown in equation (2), where t is the data stored for a texel, w is weight for cell vertex i, and d is the original data value for the property at vertex i.

$$t = \sum_{i=0}^{nVerts} w(i) * d(i) \quad \textbf{(2)}$$

The algorithm uses VTK for most operations described, such as creating the octree, evaluating sample point location within cells, and calculating weights for interpolated data.

This above approach is good for datasets in which the bounding box is filled with cells containing data. However, if the unstructured dataset has a small, concentrated volume of cells with non-zero data relative to the total volume of the bounding box, the texture produced will not contain enough data for effective analysis. Alternative approaches for creating the sampled texture are left for later discussion.

```
createTrueFalseStructure()
{
  octree = createOctreeFromDataset(datset);
  bbox = dataset->getBoundingBox();
  delta = (bbox.Min – bbox.Max)/( textureSize -1);

  //the bottom corner of the bbox/texture
  double pt[3] ={0,0,0};
  pt[0] = bbox[0];
  pt[1] = bbox[2];
  pt[2] = bbox[4];
  numberOfTexels = textureSize[0]* textureSize [1]* textureSize [2];
  i=0;
  j=0;
  k = 0;
  nX = textureSize [0]-1;
  nY = textureSize [1]-1;
  nZ = textureSize [2]-1;

 // loop through the bounding box, sampling
 for (i = 0;i<numberOfTexels;i++){
    pt[2] = bbox[4] + k* delta [2];
    pt[1] = bbox[2] + j* delta [1];
    pt[0] = bbox[0] + (i++)* delta [0];

    //check if the point is in a valid cell in the domain
     if(octree ->isInCellInDataset(pt)== true)
       trueFalse[i] = true
    else
       trueFalse[i] = false

  //increment counters to step through bbox
   if(i > nX){
     i = 0;
     j ++;
     if(j >nY){
       j = 0;
       k ++;
       if(k > nZ){
          k = 0;
       }
     }
   }
 }
```

Figure 3.1 True-false sampling algorithm

3.2 Managing texture files at runtime: cfdTextureManager

During runtime, an interface is needed to manage the created texture data. Upon initialization, the texture file must be read and the appropriate information extracted. During runtime, the texture data should be easily accessible so that changing texture data in the visualization is efficient. A so-called "texture manager" handles these issues in the following manner:

1. The texture manager reads files representing the texture data and stores pertinent information describing the texture and the CFD data represented by the texture file.
2. The texture manager can store information for multiple texture files.
3. The texture manager converts data read in the file into an UNSIGNED_BYTE format that is readily accessible for use in texture memory.
4. In the case that a texture manager contains data for more than one data field (i.e., transient data), the texture manager is responsible for determining the "current field" to display.

The texture manager API implementation is shown in Appendix A. It should be noted that the texture manager is responsible for managing the texture data, not visualization of the data.

3.3 Visualization of 3D textures in VE-Xplorer: cfdVolumeVisualization

A scene graph representation of volume rendering was developed using the concept as described earlier. The volume visualization node handles all aspects of visualizing a texture manager and is the basis for the proposed texture-based framework. The volume visualization node creates the appropriate polygons for "slicing" the texture data. The parameters for mapping the texture to the slices are also managed by the volume

parameters for mapping the texture to the slices are also managed by the volume visualization node. The volume visualization node also provides the basic interface for interacting with the visualization, such as creating contours via clipping planes.

3.3.1 Volume visualization scene graph structure

Figure 3.2 depicts the basic scene graph structure of the volume visualization node.

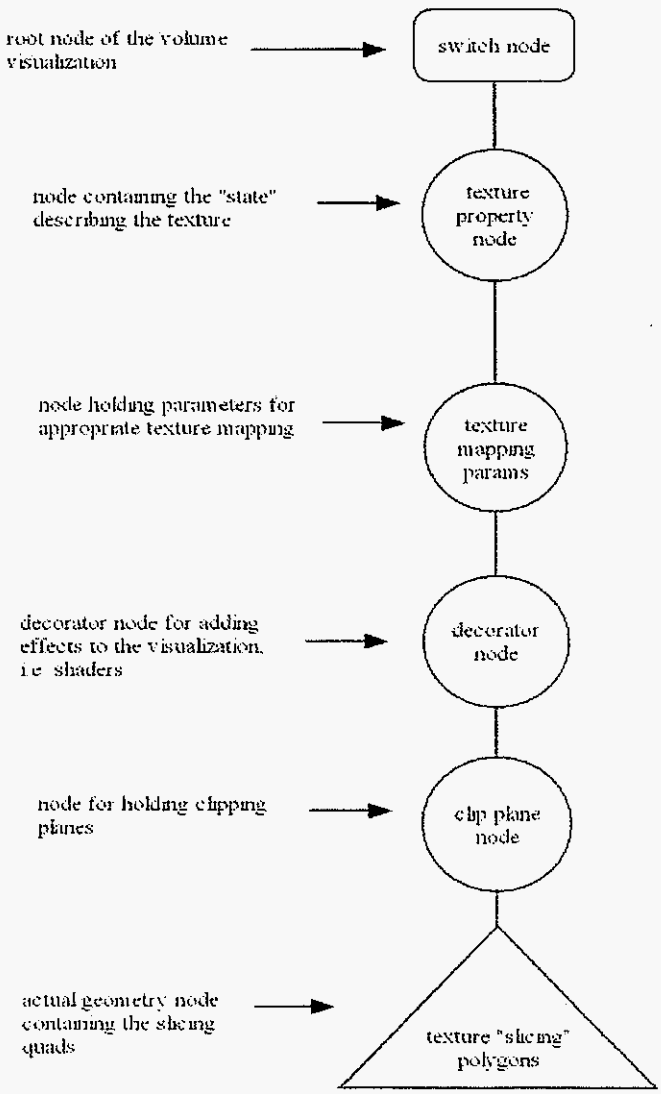


Figure 3.2 Scene graph structure of volume visualization node

Some explanation is required for the graph depicted in Figure 3.2. The root node is actually a *switch* node, which is a *group* node whose children can be switched off or on. Child nodes that are "off" are skipped during traversal. Nodes that are "on" are traversed as well as all of the root node's child nodes. The root switch node has one child, a group node containing the state describing the three-dimensional texture to be visualized. The texture property node has one child, a group node that contains the information for mapping the texture to the polygon slices appropriately. The "decorator node" is actually just a group node that serves a special purpose. If effects such as shaders are to be added to this visualization, the decorator node is used to bypass the predefined state held by the texture property node. By adding the desired effect as a child of the root switch node and attaching to the decorator node, the original visualization is kept intact but can be bypassed during traversal. This setup allows for switching between effects by simply setting the switch to the appropriate value. Figure 3.3 depicts a graph with a shader state added to bypass the original state of the volume visualization node. Multiple effects can be added and toggled in a similar fashion.

The decorator node has a single child, which holds the clipping planes. These planes are created and manipulated when the user requests a contour plane in the dataset. Finally, the actual geometry that "slices" the texture data is added as a child of the clip plane node.

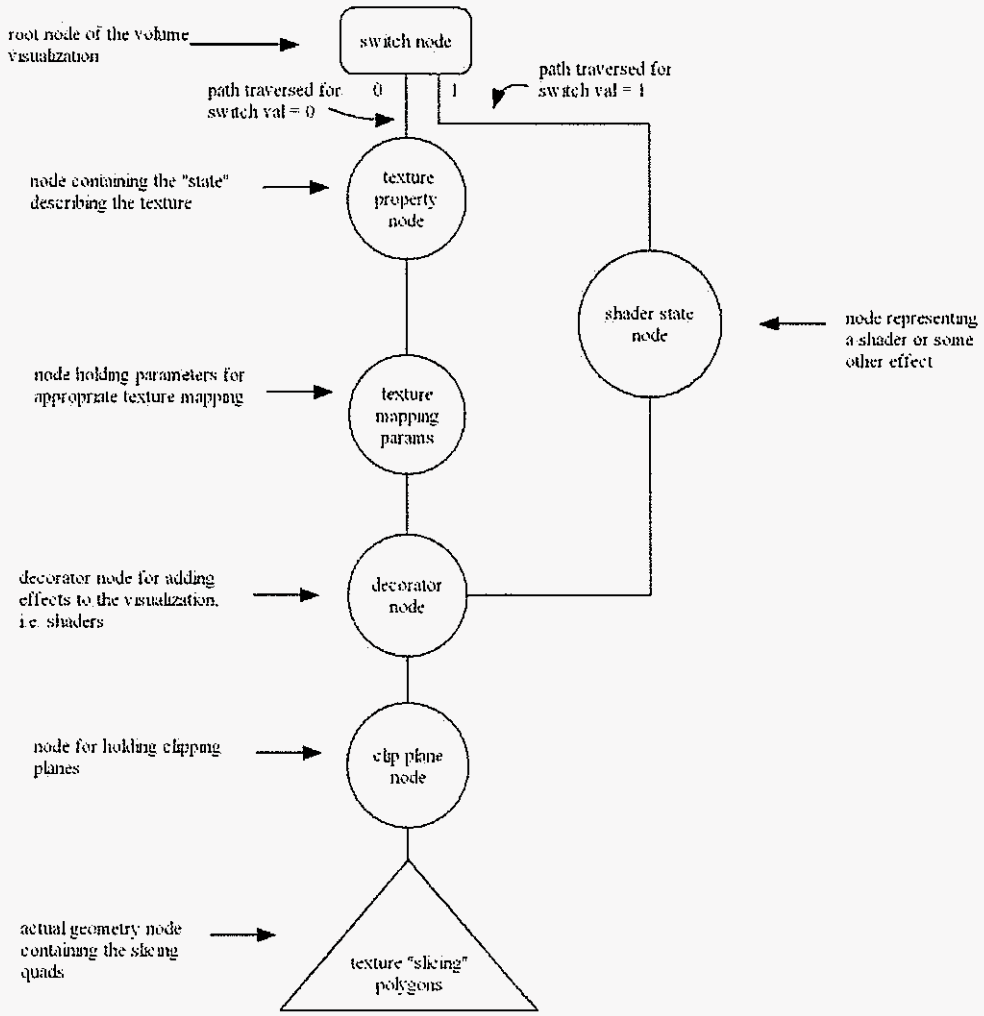


Figure 3.3 Volume visualization scene graph with a shader *state* added to by-pass the original *state*.

3.3.2 Visualization of the texture

As stated earlier, the texture manager is solely responsible for managing the texture data. The volume visualization is realized by setting a texture manager as the texture data. Once the volume visualization is created, a texture manager is set and the data is read from the texture manager. The key to efficiently switching textures is the ability of the OpenGL API

to allow the data in a resident texture to be switched without recreating the memory for the texture. This is done by a call to glTexSubImage*() (Woo, 1999). The only requirement is that the data replacing the original texture data fit within the dimensions of the original data. For example, if the original texture data has dimensions 128,128,128, valid sizes for the data replacing the original data can be no greater than 128 in each direction but can be as small as 2 . As long as the texture data supplied for updating the visualization fits these requirements, switching data is valid and a new texture need not be created. The implementation API of the volume visualization node is listed in Appendix A.

3.4 Managing datasets: cfdTextureDataset

With scene graph representation in place, the volume visualization and associated texture managers need to be organized into a structure containing all the information associated with the equivalent CFD dataset and methods for accessing the visualization. The texture dataset holds the following:

1. A single volume visualization node.
2. A texture manager for each scalar in a dataset.
3. A texture manager for each vector in a dataset.

The texture dataset provides an interface for the following:

1. Methods for setting a specific scalar or vector on the volume visualization.
2. Methods for retrieving the volume visualization associated with the particular dataset.

The texture dataset API implementation is listed in Appendix A. It should be noted that for a particular property of the flow field, scalar or vector, a single texture manager holds all the texture data. For instance, if a solution has density and velocity magnitude as two of

its scalar properties, two texture managers in the texture dataset contain the data for each of the properties. With this implementation, switching properties only requires switching the active texture manager for the volume visualization, which in effect calls glTexSubImage3D(). This applies for steady state as well as transient datasets.

3.5 VE-Xplorer interfaces for texture-based framework

The requirements to plug the texture-based framework into VE-Xplorer were mentioned previously and discussed in Chapter 1. They are restated here for clarity:

1. Parameter file block and a corresponding object ID
2. Visualization handler for user commands and interacting with the scene graph

3.5.1 Parameter file texture-based block description

The next available object ID, 15, is chosen to represent the texture dataset object. The block contains at least two information parameters for creating the texture dataset: the number of properties in the dataset and, for each property, a property texture file.

The first parameter of the property texture file is the actual number of texture files preprocessed for the property. The second parameter is a string, which is used for searching through properties of a dataset at runtime. The next parameters in the file are strings containing the name and location each of the texture files for the property. Figure 3.4 lists an example texture dataset block in a parameter file. Currently, the property texture files are manually created after the preprocessor is run. This process will eventually be moved to the preprocessor.

```
15              //texture dataset ID
 6              //number of property texture files
 ./CO.txt       // property texture file for CO
 ./gas_temp.txt // property texture file for gas_temp
 ./H2.txt       // property texture file for H2
 ./H2O.txt      // property texture file for H2O
 ./O2.txt       // property texture file for O2
 ./u_mag.txt    // property texture file for u_mag
```

Figure 3.4 Texture dataset parameter block

3.5.2 Texture-based visualization handler

The visualization handler for texture-based datasets manages all communication from the user that involves the texture dataset. The texture-based visualization handler is responsible for manipulating the volume visualization via the texture dataset interface. Currently, new commands are not added to the interface. However, the texture-based handler intercepts the currently available commands and interprets them appropriately for the volume visualization. For example, if the user issues a command to generate a contour plane, the texture-based handler interprets this command as the position for adding a clipping plane, effectively exposing the interior of the volume that correlates to the contour plane. The API implementation of the texture-based visualization handler is listed in Appendix A.

3.6 Application in VE-Xplorer

Thus far, application of volume visualization, and hence a texture-based framework, have only been described. The framework is simply a basis to build upon. A simple application would use the volume visualization as is and simply view scalar CFD data as a volume rendering. The advantages, such as transient data handling, are enough to merit using a texture-based framework. For example, the only geometry required for volume rendering of the data are the "slicing" polygons. These are created once a texture manager is set for a texture data set. To generate the standard contour plane, a clip plane is added at the desired location. The result is instantaneous because the graphics thread, and therefore the dataset query and the creation of new geometry, of the general visualization method is eliminated. The process for transient is the same. The texture manager inherently handles switching the underlying texture data based on a timing algorithm and a user-specified "delay time." The delay time simply sets the amount of time to wait before switching the texture data in the texture. Because the graphics are not generated during the query, no delay in visualization updates is evident.

Although the framework is beneficial in its current state, shaders should be used to enhance and take full advantage of its visualization capabilities.

3.6.1 Extended applications

Fragment programs, or shaders, are programs that are run on the fragment processor of the GPU. These programs are used to output the color of a fragment as they are processed for each graphics object in the scene. Fragment shaders allow the developer to control how the color of each fragment is calculated. Until very recently, fragment programs were not

easily accessible for the average developer because no high-level language existed to support such programs. Most previous shader work was done in an assembly-type language.

Fortunately, the demand for easy access to shaders has led to development of high-level shader languages such as Nvidia's Cg (C for graphics). OpenGL's architecture review board has even approved a shading language for its standard, OpenGL Shading Language. These types of languages and the APIs associated with them have led to many extended applications using shaders. A simple fragment program is listed in Figure 3.5.

```
struct f2app{
  float4 color : COLOR;
};

//simple shader to look up texture value in a 3d texture
f2app fp_volume(float4 color : COLOR,
                float3 texCoord:TEXCOORD0,
                uniform sampler3D volumeData)
{
  f2app retColor;

  //look up the value in the texture
  retColor.color = tex3D(volumeData,texCoord);
  return retColor;
}
```

Figure 3.5 Simple fragment program that applies a texture to the incoming fragment using a texture look-up.

The fragment program listed is written in Cg. At first glance, the program looks very similar to a C program. The syntax is very similar with only a few exceptions (Nvidia, 2004). The *struct* at the top of the file is simply defined to hold the color returned to the application from the fragment program, hence the name **f2app**. The program declaration

syntax is similar to a C function. The program will return an **f2app** *struct*. The parameters passed in are:

1. *float4* **color**:**COLOR** → *float4* is a vector type of four float variable, similar to an array in C. The "**:COLOR**" following the variable is a *binding semantic* (Nvidia,2004) binding the variable **color** to the interpolated color.
2. *float3* **texCoord**:**TEXCOORD**→ float3 is a vector type of three float variable. This time the "**:TEXCOORD0**" is a binding semantic that binds **texCoord** to the interpolated texture coordinated associated with the first texture unit[2].
3. *uniform sampler3D* **volumeData** → *uniform* is a type qualifier meaning that this variable cannot be modified by the shader. *sampler3D* identifies the texture. There are similar types for one- and two-dimensional textures. Textures are uniform as they cannot be modified by the shader. **volumeData** is the variable actually representing the texture data.

To begin, the program defines an *f2app* instance, **retColor**, to store the return color. The next call, *tex3D* () is used to look up a value in the three-dimensional texture, **volumeData**, at **texCoord**. The value returned from *tex3D* () is then stored in the member **color** of **retColor** with syntax similar to C. Finally, **retColor** is returned.

Of course, this is a simple program. A more complicated program could be written to enhance the visualization. For example, if the input texture represented a scalar property from our CFD dataset, an enhancement shader could be written to brighten certain ranges of scalar values and dull out others. This is usually done through the use of transfer functions.

[2] OpenGL allows the user to specify multiple textures for a single primitive to achieve special effects. This is referred to as multi-texturing (Woo, 1999).

3.6.2 Enhancing volume visualizations via transfer functions in shaders

Effective volume visualizations involve the use of transfer functions. By storing single scalar values in an ALPHA texture, transfer functions can be used to map input values to a color and an opacity value. There is quite a bit of research based on developing effective transfer functions, some of which is based on curvature (Hladuvka, 2000) or even the image itself (Fang, 1998). A simple example of a one-dimensional transfer function is a ramp that increases the input value by a gamma correction factor and linearly ramps the opacity so that as the value goes up, the opacity increases. This ramp function can be stored in a one-dimensional LUMINANCE_ALPHA texture, where the luminance value holds the value for the brightness and the alpha value holds the opacity. Figure 3.6 shows a simple function for calculating the transfer function values. Note that this algorithm would be implemented in the software to provide an interface for updating the transfer function. The shader would then access the updated transfer texture for use.

```
gammaTransferFunction(double gamma)
{
 unsigned char luminance [256];
 unsigned char alpha[256];

 //gamma table
 double gTable[256];

 y = 0;
 //calculate the gamma table
  for (int i=0; i < 256; i++)
  {
     y = (double)(i)/255.0;
     y = pow (y, 1.0/gamma);
     gTable[i] = (int) floor(255.0 * y + 0.5);
  }

  for (int i = 0; i < 256; i++)
  {
      luminance [i] = (unsigned char)gTable[i];
       alpha[i] = (unsigned char)i;
  }
}
```

Figure 3.6 Simple gamma correction transfer function

For the transfer function, the user adjusts the value of **gamma**. This affects the brightness of the input value. The alpha values are fixed for this transfer function but could just as easily be made adjustable by passing in another parameter to the function. To exhibit how transfer functions combined with shaders can be useful in volume visualization, the fragment program of Figure 3.5 is modified to include the gamma correction transfer functions. The input texture **volumeData** is of the format ALPHA. It holds a single value representing the scalar. The transfer functions are stored in one-dimensional LUMINANCE_ALPHA, UNSIGNED_BYTE textures.

```
struct f2app{
  float4 color : COLOR;
};

//simple shader to look up texture value in a 3d texture
f2app fp_volume(float4 color : COLOR,
            float3 texCoord:TEXCOORD0,
            uniform sampler3D volumeData,
            uniform sampler1D transferFunction)
{
  f2app retColor = float4(0,0,0,0);
  float4 lookUpValues;
  float4 redFrag;

  //look up the value in the texture
  lookUpValues = tex3D(volumeData,texCoord);
  //use the look up values as input to the transfer function
  redFrag = tex1D(transferFunction,lookUpValues.r);
  //set the brightened red value
  retColor.color.ra = redFrag.ra;
  retColor.color.gb = lookUpValues.gb;
  return retColor;
}
```

Figure 3.7 Fragment program exhibiting usage of transfer functions.

The program input parameters are modified to read the one-dimensional texture storing the transfer function, **transferFunction**. Two new variables are declared within the shader. A *float4* **lookUpValues** holds the values in the three-dimensional texture, which is our original data, and a *float4* holds our calculated red fragment color **redFrag**. Instead of using the values returned from the original **volumeData** texture as the final color, the transfer function is used to create a red fragment. This is termed as *dependent texture look up* because texture coordinates are not explicitly specified for the transfer function texture. Instead, the coordinates are dependent on values read from another texture or calculated within a shader.

The transfer function returns a value for the red component and an alpha value. These are then stored in the return color. The notation "." is a convenience operator defined in the Cg language as a *swizzle* operator. The swizzle operator acts on the specified components. For instance, for a *float4*, the valid components are 0-3. These are specified as "XYZW" or "RGBA" when used with the operator. *X* corresponds to the first component, *y* to the second, and so on. Similarly, *r* corresponds to the first component, *g* to the second, and so on. Since the one-dimensional texture takes a one-dimensional value as input, a *float* as opposed to a *float3*, **lookUpValues.r** is used as the look up value in the transfer function texture.

A transfer function could be defined for each color component to separately control the appearance of the final visualization. The user could then adjust the transfer functions during runtime to highlight specific ranges of scalar values.

Figure 3.8 shows some simple effects that can be achieved by using shaders on scalars of a fictional dataset. Figure 3.8 (a) shows the volume rendered without shaders. Figure 3.8(b) shows the same volume rendered using a shader that simply replicates normal volume visualization via shaders by blending the alpha value of the texture with the alpha value of the incoming fragment. Figure 3.8(c) shows the same volume with the shader modified to remove blue components, corresponding to low magnitude values, from the final display image. Finally, figure 3.8(d) is a rendering of the volume with low magnitudes removed and high (red component) gamma corrected. The effect is that red and yellow areas of the volume are brightened.

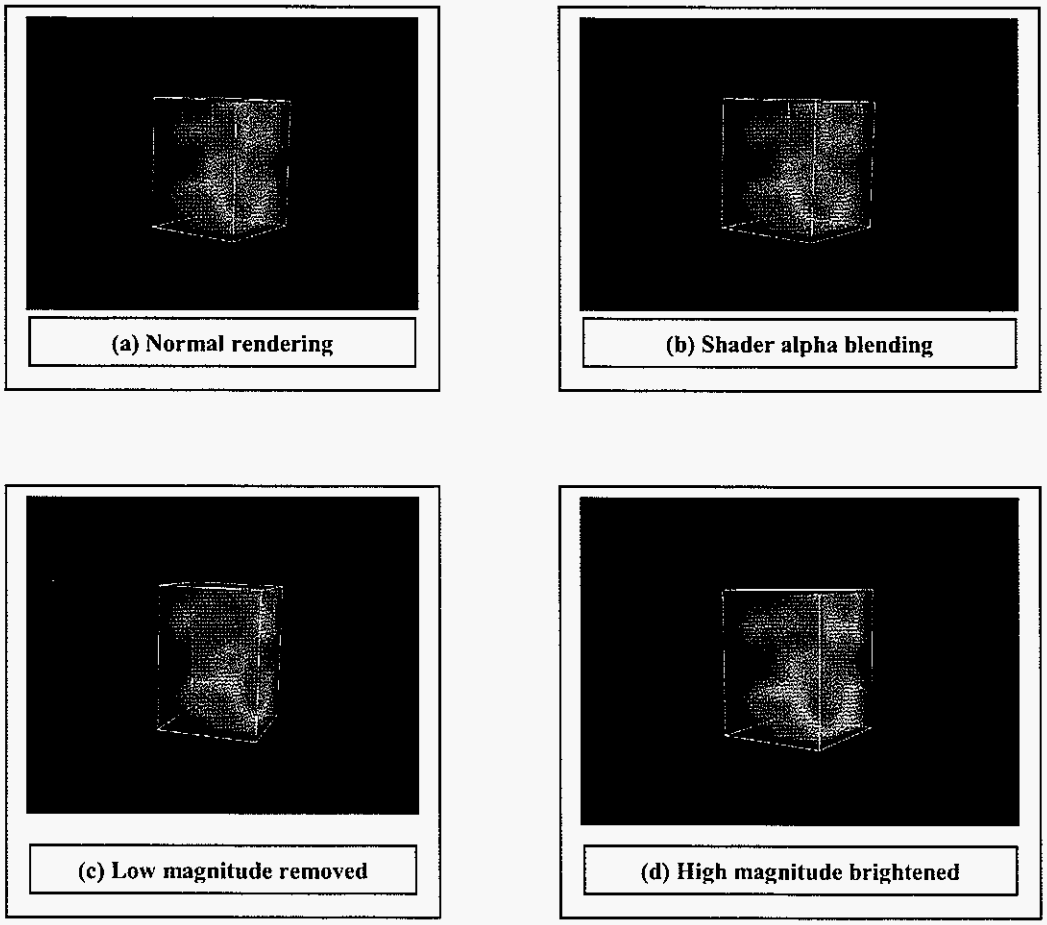


Figure 3.8 Simple effects on a scalar data set using shaders

3.6.3 Managing states in the scene graph

When discussing scene graph managing APIs, *state* refers to the properties, such as color, that can be active when geometries are rendered. Properties that can also be modes such as equations for blending, texturing, lighting, shaders, etc. are all such properties in a state. Until a state is changed, it remains in effect. For example, if a texture is defined as an

active state for drawing, each object that is rendered after the texture is activated is rendered with that texture until it is deactivated.

Scene graph APIs allow a developer to efficiently manage different states in a single graph. To apply a state to a certain node in the scene graph, the state must be set to the node. In the volume visualization node, the node labeled "texture property node" is actually a group node with a state containing the texture. It is important to note that in scene graphs, if a parent node has a state defined, its children will inherit that state unless it is specified to be overridden or ignored. This allows the switch structure to selectively apply any added effects to geometry by switching the geometry's parent node structure.

3.7 Adding shaders to the volume visualization node

Referring to Figure 3.3, the volume visualization node's structure provides a simple interface for adding "decorators," or effects, such as shaders. Adding a shader such as the transfer shader described in 3.5.2 requires components to do the following:

1. Create the shader state.
2. Create the group node for owning the shader state.
3. Add the group node to the volume visualization node graph.

To accomplish these tasks, two interfaces are developed: a shader manager and a visualization node handler.

3.7.1 Shader manager

The shader manager is an interface that creates a state containing the shader. The manager initializes the state from the actual file and creates the necessary properties for the shader. As an example, the transfer shader described in Figure 3.7 requires two textures from the application, a one-dimensional texture for the transfer function and a three-dimensional

texture holding the scalar data. The shader manager is responsible for creating these textures and setting them on the state, which is the shader. Since the shader parameters will differ for different shaders, the shader manager is customizable via C++ inheritance. The implementation of the base shader manager class is listed in Appendix A. An example implementation of a derived shader manager for the shader generating Figure 3.8 is listed.

3.7.2 Volume visualization handler

The volume visualization handler creates the group node containing the state that describes the shader to be added to the volume visualization node. The volume visualization handler is also responsible for adding the shader group node to the graph. This interface is also customizable, via C++ inheritance, so the developer can create different effects as needed. The implementation of the volume visualization handler is listed in Appendix A along with the derived visualization handler used for generating Figure 3.8.

3.8 Vector visualization: Texture advection

So far, only scalar investigative methods have been discussed for the proposed framework. The framework is based on visualizing a property of the flow, which directly correlates to scalars but not to vectors. To integrate vector analysis tools in the proposed framework, texture advection methods could be utilized. Texture advection methods transport a collection of particles represented by a property texture, according to the vector field. The path a particle travels due to the vector field can be described by the ordinary differential equation:

$$\frac{dr(t)}{dt} = u(r(t),t) \qquad (3)$$

The variable **r** (t) represents the path of a particle at an instance of time t, and **u**(**r**, t) is the vector field (Weiskopf, 2004). A first-order backward difference explicit Eulerian scheme (Tannehill, 1997), applied to Equation (3), yields the equation for particle positions:

$$r(t-\Delta t) = r(t) - \Delta t u(r(t), t) \qquad (4)$$

The actual positions of the particles, **r**, are represented in texture advection schemes by the texture coordinates, **c**, of the property texture, T(**c**). To solve for the property texture T at a given time, Equation (4) is applied to the texture coordinates and the property texture to yield:

$$T_t(c) = T_{t-\Delta t}(c - \Delta s v_t(c)) \qquad (5)$$

In Equation (5), T_t represents the property texture at the current time step while $T_{t-\Delta t}$ represents the property texture at the previous time step. Δs corresponds to the size of the time step in texture space and v_t is the vector field at an instance in time (Weiskopf, 2004).

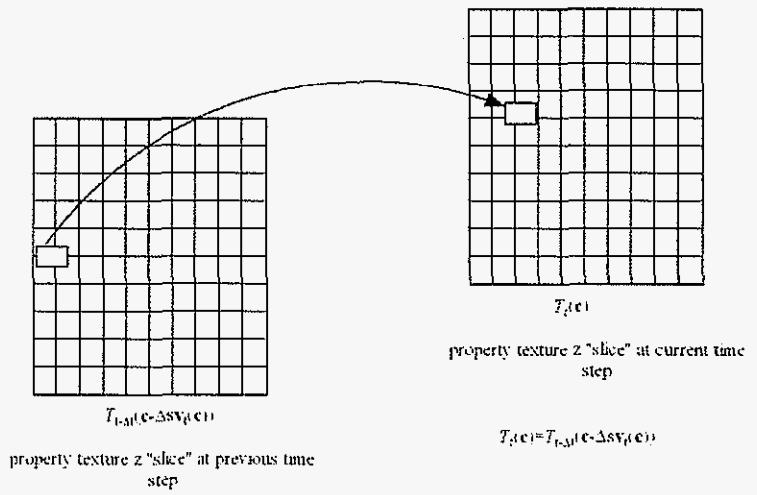


Figure 3.9 Texture advection

Figure 3.9 depicts the basic idea of texture advection. To calculate the value stored in the property texture at the current time step, Equation (4) is applied to the original texture coordinates of the current property texture yielding advected coordinates. These coordinates represent the location of the property before the advection due to the vector field. The advected coordinates are used to look up the property values from the previous time step. The property values are then stored at the new location (the original texture coordinates) in the property texture of the current time step.

Texture advection methods such as line integral convolution (LIC) (Cabral, 1993), Lagrangian-Eulerian advection (LEA) (Jobard, 2002) and image-based flow visualization (IBFV) (van Wijk, 2002) are restricted to two-dimensional fields. IBFV combines the effectiveness of particle injection and dye advection-type investigative techniques at interactive frame rates for steady and unsteady flow fields. For each time step, the vector

field warps the current property texture, or underlying mesh, and a filtered noise image is blended with the advected image to produce animated images, even for steady state datasets.

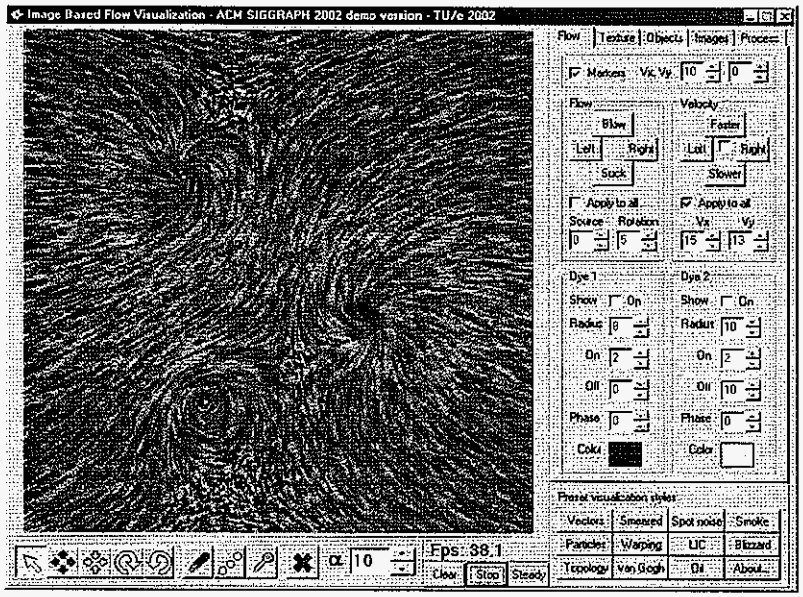


Figure 3.10 Image-based Flow Visualization. A screen capture of the IBFV demo program from ACM Siggraph 2002

3D-IBFV (Telea, 2003) is an extension to IBFV that uses the graphics hardware to accelerate the visualizations and achieve interactive frame rates for steady and unsteady flow fields. For 3D-IBFV, the original IBFV algorithm is applied to each z "slice" of the property field. For each "slice," the values of the previous z and next z "slice" of the property texture are interpolated onto the current z slice to advect the property field (Telea, 2003). The updated property texture is then visualized in a volume rendering. Streaklines, dye, and particle injections can all be visualized in real time. The algorithm of 3D-IBFV is restricted to vector fields that contain small z components and also require multiple passes to update single slices of the property texture.

GPU-Based 3D Texture Advection (Weiskopf, 2004) expands on 3D-IBFV by implementing the advection calculations in shader programs, storing the vector field in a three-dimensional texture, therefore eliminating the z component restriction of 3D-IBFV. This algorithm maps Equation (5) directly to a shader program that calculates texture coordinates for a dependent texture lookup in the property texture from the previous time step. This is done in a single pass of a z "slice" of the property texture to achieve interactive frame rates.

Chapter 4: Results

In this chapter, various datasets are compared in the texture-based framework with the current framework of VE-Xplorer. The CFD visualization application used is VE-Suite's VE-Xplorer. All runs were done on a Windows XP machine with a 3.0 gigahertz Intel Pentium 4 processor with 512 megabytes of random access memory and an Nvidia 5200FX graphics card, with 128 megabytes of memory, supporting OpenGL 1.5.

4.1 Scalar datasets

The following dataset is used to test basic visualization techniques of VE-Xplorer. Figure 4.1 shows the dataset with no cutting planes specified. Note that the picture shown in (a) is actually a cutting plane specified at x = 0. For the initial visualization of the current framework, a cutting plane must be specified; otherwise a visual representation of the dataset is not available. In contrast, (b), the texture-based version, shows the entire volume on initial viewing.

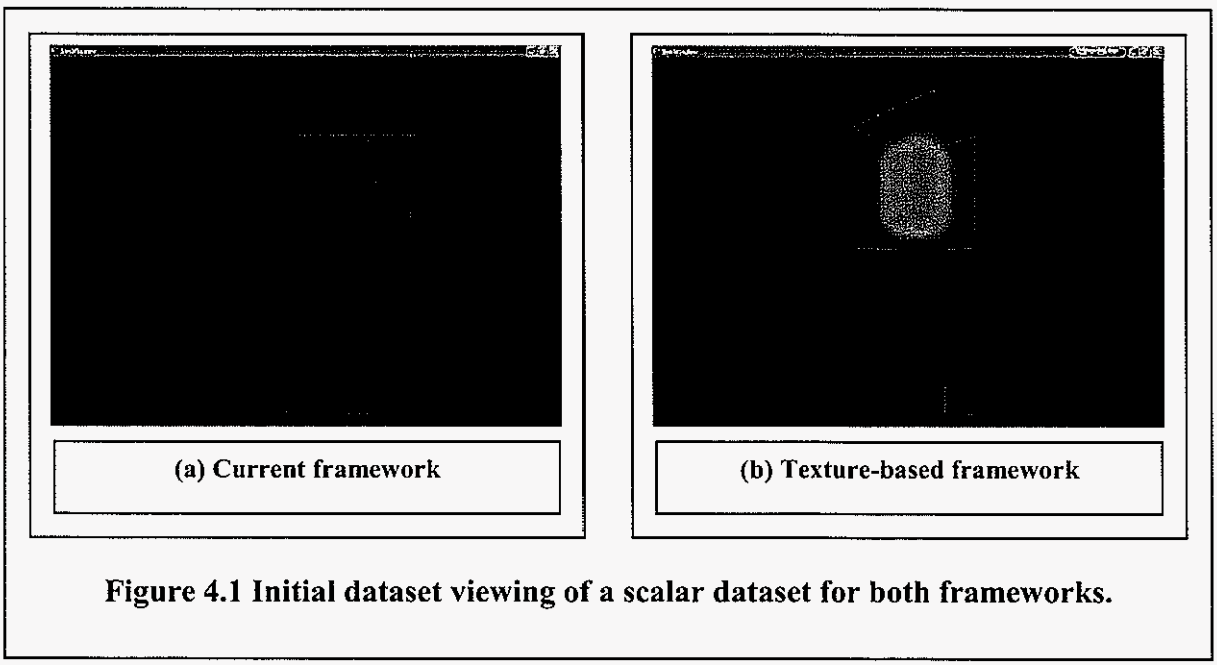


Figure 4.1 Initial dataset viewing of a scalar dataset for both frameworks.

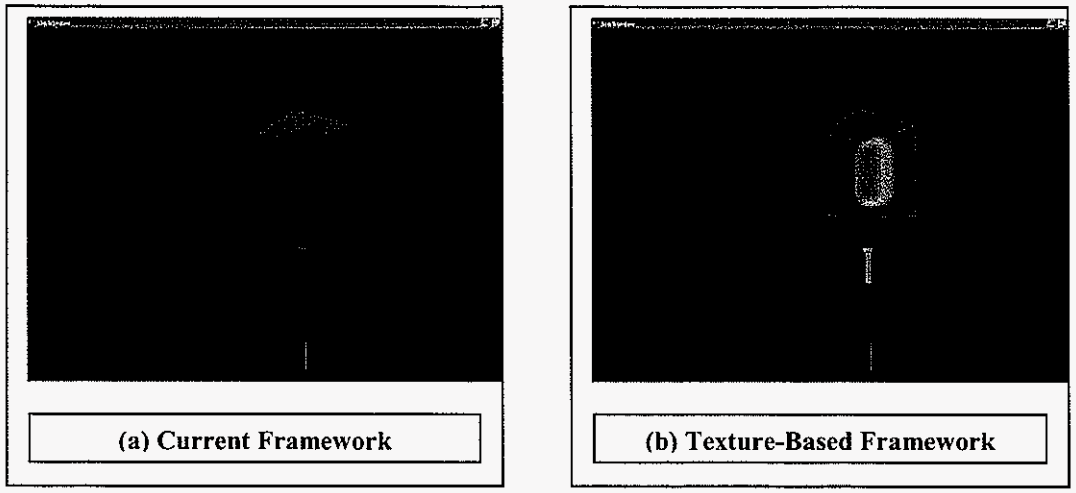
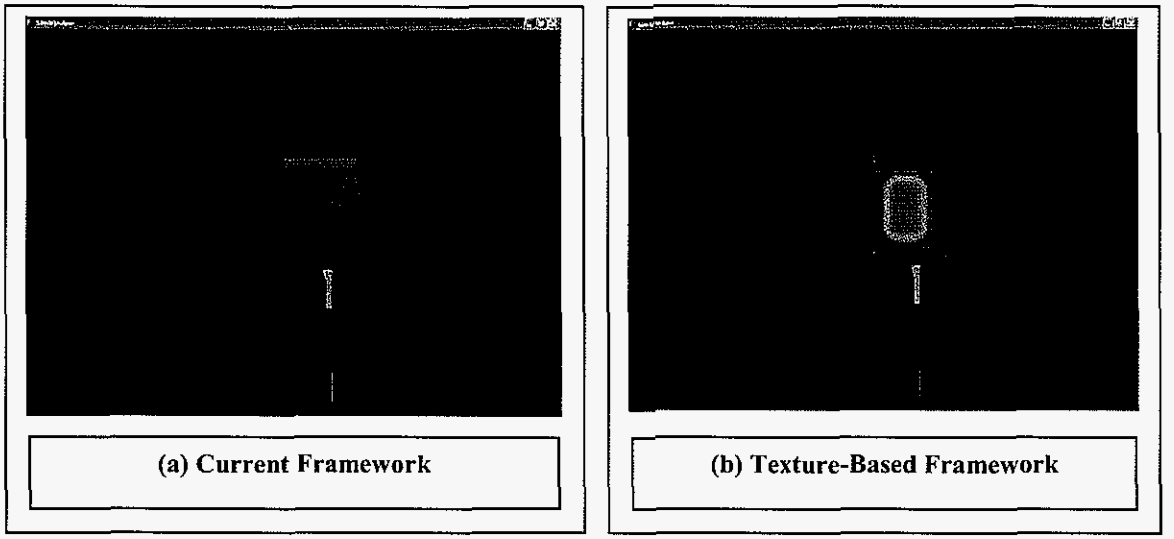


Figure 4.2 Rotated x contour plane visualization at $x = .5*xmax$.

Figure 4.2 shows standard contour plane visualization in (a), the current framework, and (b), the texture-based framework. The visualization is rotated approximately 45 degrees about the z-axis from its original orientation. This comparison shows the basic difference in the contour plane engineering analysis technique for the two frameworks. The current framework generates an actual plane at a user-specified position, whereas the texture-based framework cuts through the volume at the user-specified position, exposing the interior of the volume corresponding to the contour plane, leaving the rest of the volume intact.

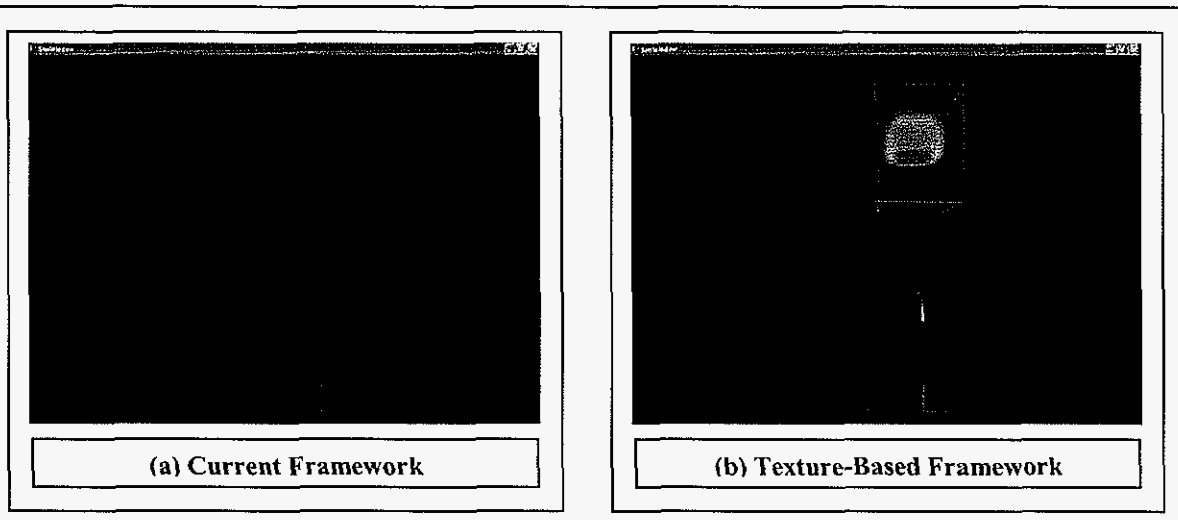
Figures 4.3 and 4.4 are similar comparisons of contour plane visualizations in the current framework and the texture-based framework. Figure 4.3 displays y contours and 4.4 displays z contours.



(a) Current Framework

(b) Texture-Based Framework

Figure 4.3 y contour plane visualization at $y = .5*ymax$.

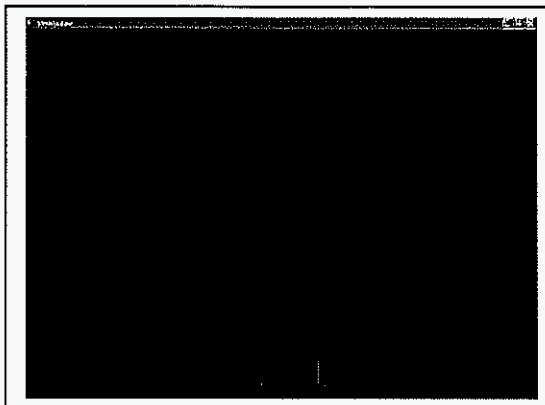


(a) Current Framework

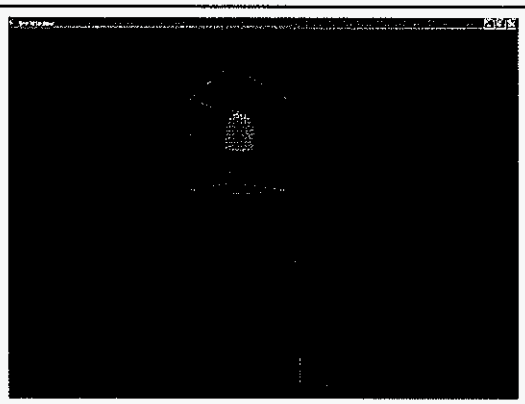
(b) Texture-Based Framework

Figure 4.4 z contour plane visualization at $z = .5*zmax$

Figure 4.5 shows the same scalar dataset with multiple contour planes. The three planes are located at 50% of the maximum value of the bounding box dimension.



(a) Current Framework



(b) Texture-Based Framework

Figure 4.5 Three contour planes at 50% along each axis.

Two differences in the visualization are apparent. First, the colors of this dataset seem to be different. The green and yellow areas in the texture dataset are a result of the transformation of the scalar values to the RGBA values, via the linear red-blue lookup table, resulting in a smoother transition of the colors from high to low magnitude. The blue region, representing low magnitudes, is barely visible in the texture-based version whereas it is a dominant feature in the current framework.

The second difference is the actual visualization itself. For the current framework, two-dimensional planes are used to represent various contour planes. The perception of depth of the dataset can be lost, making the visualization confusing. Intersections of multiple contour planes can give a misrepresentation of the actual data that is present.

In the texture-based framework, the visualization exposes the interior of the volume at the specified contour locations, leaving the rest of the volume intact. The perception of depth is not distorted, even with intersecting contour planes, as can be seen in the figures 4.2-

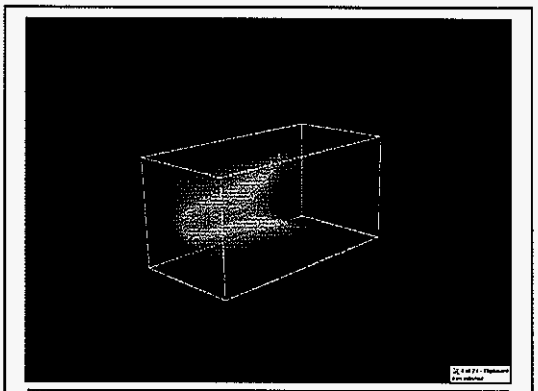
4.2 Transient scalar datasets

The dataset used in this comparison is a rectilinear grid of dimensions 200x120x120, simulating a celestial formation. The textures representing the scalar properties in the dataset are of the dimensions 128x128x128. There are currently twelve total time steps, each containing data solutions for the four scalar properties and the two vector fields listed below:

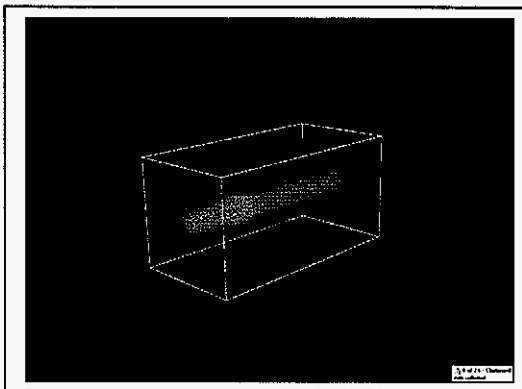
1. Velocity magnitude ➔scalar
2. Magnetic field magnitude➔scalar
3. First internal energy ➔scalar
4. Density➔scalar
5. Velocity field➔vector
6. Magnetic field➔vector

The entire transient solution cannot be processed under the current framework. Also, for interactive frame rates, the contour planes must be pre-computed before loading the dataset into the application, but not all time steps for all solutions can be loaded simultaneously for investigation. The texture-based framework loads the entire dataset and scalar analysis methods, specifically contour planes, are achieved at interactive frame rates.

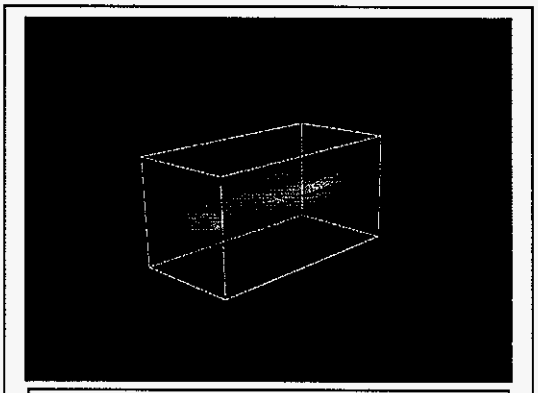
Figure 4.6 shows various time steps of the dataset for the velocity magnitude with a y contour plane.



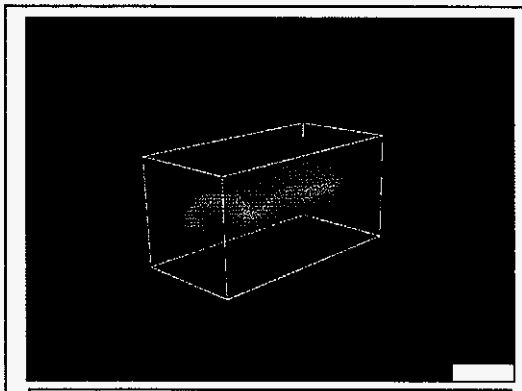
(a) $t = 1$



(b) $t = 4$



(c) $t = 8$



(d) $t = 10$

Figure 4.6 Velocity magnitude of a transient dataset at various time steps with a contour plane at $y = .35*ymax$

Figure 4.7 shows the magnetic field magnitude of the same dataset with the same y contour plane as Figure 4.6.

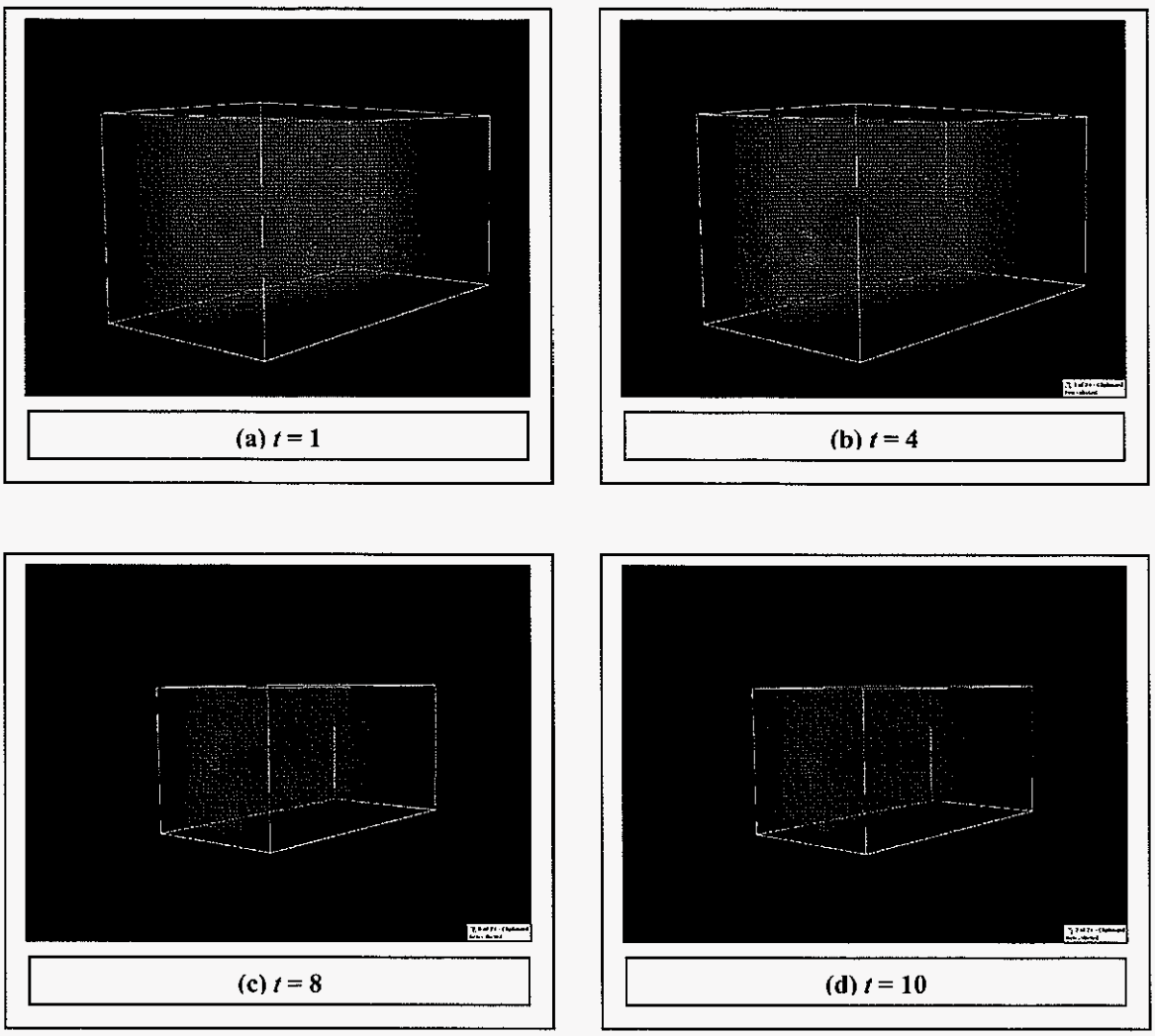


Figure 4.7 Magnetic magnitude of a transient scalar dataset at various time steps with a y contour plane at $y = .35 * ymax$

The property textures for the two remaining scalar properties, first internal energy and density, did not produce visualizations. This is due to the current implementation of creating the RGBA values for the texture. As discussed in Chapter 3, the minimum and maximum

scalar values of each property are linearly mapped to a red-blue color lookup table. If the values are concentrated in the low end, the algorithm returns black pixels and an opacity value close to zero. If the values are concentrated in the high end, the algorithm returns mostly red pixels that are opaque. Improvements for this problem could easily be overcome by implementing some simple shaders, representing various transfer functions, and are discussed in Chapter 5.

4.3 Vector data analysis

For analysis of vector fields, the three-dimensional GPU advection algorithm presented in (Weiskopf, 2004) is implemented. Currently, this implementation is under development. To implement this algorithm correctly requires four key components:

1. Pixel buffer rendering for the advection routine.
2. Three-dimensional texture updates via glCopyTexSubImage3D() to update the property texture one "slice" at a time.
3. A mechanism for swapping textures of the current time step with the previous time step in a "ping-pong" scheme.
4. Transfer functions to interpret the property texture data into the appropriate color values for the final visualization

The proposed framework successfully implements the pixel buffer, or *pbuffer*, (Wynn, 2001) and the accompanying shader for implementing the advection. The framework also properly updates the property texture for the current time step as well as the "ping-pong" scheme for the animation updates.

The basic algorithm for visualization begins by rendering a quad off-screen. The off-screen rendering accesses the advection shader and draws the quad with values representing

the results of the advection algorithm on the property texture of the previous time step. These values are then copied to the property texture of the current time step for the appropriate z elements of texels. This process is repeated for each "z slice" of texels for the property texture for the current time step.

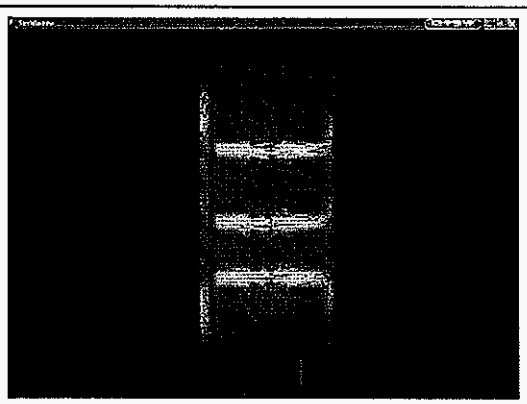
The updated property texture is then volume rendered. A transfer function shader is applied to the rendering to allow the user to adjust the visualization during runtime. Finally, "ping-pong" of the two property textures occurs to prepare for the next time step. To "ping-pong" simply means to switch the rolls of the two textures, making the current property texture the previous property texture for the next time step.

4.3.1 Noise injection

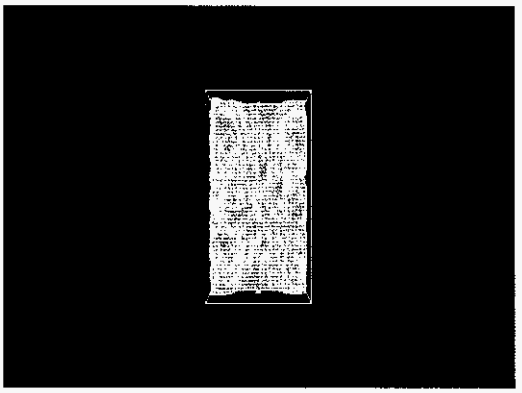
Because the entire celestial dataset could not be loaded in the current framework, a fermentor data set is used for vector analysis comparisons. The dataset contains 12 total time steps. The 10^{th} time step is being visualized below. The fermentor geometry contains three internal rotating propellers aligned along the z-axis. The two dataset property textures for the "ping-pong" of the algorithm have a resolution of 128^3, as does the velocity field texture. The noise particle injection texture has a resolution of 32^3 and is repeated, or tiled, to cover the entire domain of the flow field. Scaling can be applied to the injection texture coordinates to modify the density of the particles within the flow field.

Figure 4.8 shows the fermentor dataset, comparing (a) the current visualization technique to (b) the texture advection algorithm (Weiskopf, 2004) integrated in the proposed framework. The current technique displays a plane of vector glyphs at grid points in the dataset. The glyphs are colored by magnitude and can be scaled by the magnitude. The user is also able to adjust the number of glyphs displayed and the size, if desired. The advection

algorithm displays a cut-away of the volume dataset. The data is viewed as "whispy smoke" traveling in the direction of the flow field. This is a direct result of the transfer functions used in the volume rendering portion of the algorithm. These could ideally be adjusted to improve the final visualization, but this is left for future work. The idea here is to show that the basic algorithm can be easily incorporated into the proposed framework.



(a) Vector Glyphs



(b) 3D-GPU Texture Advection

Figure 4.8 Fermentor dataset with a contour/cutting plane located at $y = .35ymax$.

It should be noted that the visualization in (a) includes a scalar contour plane slightly offset of the vector plane. This is only included in the picture to aid in seeing the glyphs in this paper as opposed to the application viewed when running in the virtual environment. The glyphs have also been uniformly scaled for the same reasons.

Both visualizations clearly show the swirling of the vector field caused by the rotation of the propellers; however, the vector glyphs are static, while the advection visualization is continuously updated with particle injections and dissipation, creating a time-dependent animation of the steady-state dataset.

Although these visualizations are similar, the advection visualization combines depth with motion to provide an intuitive general description of the field, while the glyph version can be harder to interpret due to its two-dimensional nature.

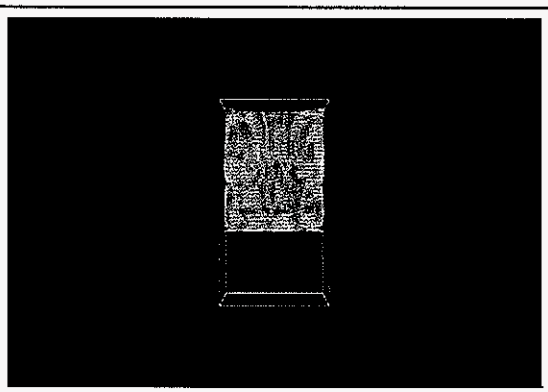
The advection algorithm inherently causes a slowdown in overall navigation and performance of the application compared to the vector glyphs as noted by (Weiskopf, 2004), but applying visualization techniques such as streamlines and cutting/contour planes causes no performance penalties. Interactive frame rates are still achieved but navigation is noticeably slower.

4.3.2 Streamlines

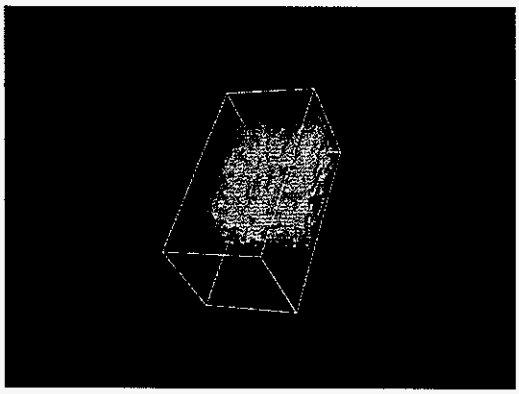
As mentioned, streamlines are generated naturally in texture advection algorithms. The current framework uses an external library to generate geometry representing streamlines. This is done in the software and causes the application to become non-responsive to user input while the streamlines are being calculated. Results are similar to the advection algorithm, but it is time consuming to generate streamlines representing interesting flow patterns. There is also an element of "trial-and-error" when generating streamlines in the current framework because interesting flow patterns are only detected if the user selects an appropriate seed point.

The advection algorithms, however, simply advect a "dye" material as another property, creating the visualizations of streamlines or streak line. Figure 4.9 shows the fermentor dataset with a dye emitter placed at various locations in the flow along with noise injection. The dye is modeled by a small, cube-shaped, three-dimensional ALPHA texture which represents the amplitude of the dye, as in (Weiskopf, 2004). A simple transformation is used to change the location of the dye and can be scaled by the user without a performance

hit. This allows the user to freely explore the flow field without having to guess at seed points to pick up interesting aspects of the field.



(a) Dye inject in upper left corner



(b) Dye injected near the center

Figure 4.9 Texture advection with a red dye and particle injection in the texture-based framework.

Figure 4.9 shows the texture advection visualization with a red dye injected at (a) [.1, .35, .9] and (b) [.5,.35,.5]. The emitter was relocated real-time without causing a delay, or pause, in the application response. The dye clearly shows the rotation in flow field but is partially occluded by the particle injections. A complete implementation of Weiskopf's algorithm allows the user to adjust the amount of particles injected from the GUI, but this implementation is left for future work due to time constraints. However, the particles can be completely removed from the visualization leaving only the dye injection as shown in Figure 4.10.

Figure 4.10 shows the same dye injection as Figure 4.9 (a) from various view locations without the particle injections. The three-dimensional dye allows easy location and

visualization of interesting flow patterns without the "trial-and-error" methods of seed point placement in the current framework.

Figure 4.11 shows different views of advected dye when the emitter is placed at an "interesting" position in the fermentor flow field.

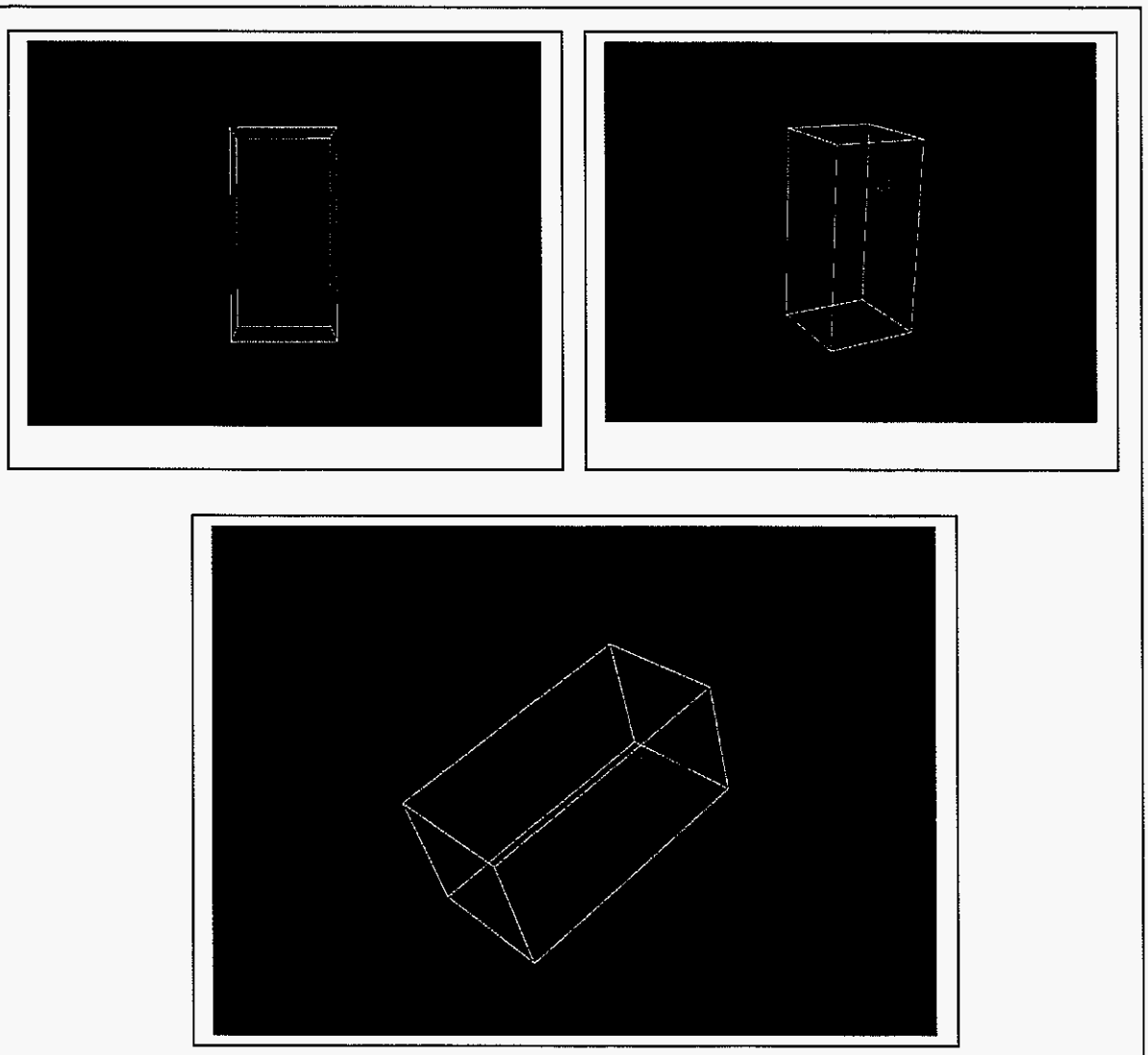


Figure 4.11 Interesting flow pattern visualized by texture advection of dye in the fermentor dataset.

Chapter 5: Conclusions and future work

A basic framework for integrating texture-based techniques into a scene graph-based CFD visualization application, VE-Xplorer, has been presented. The framework allows the application to remove runtime dataset access dependencies by visualizing preprocessed three-dimensional textures representing the dataset. The framework also generally handles any dataset, steady state or transient, removing time-consuming scene graph management issues from the current framework. The framework incorporates visualization of standard engineering scalar data analysis techniques such as contour planes. Hardware shaders are easily integrated in the framework, allowing enhancements and extensions to scalar property visualizations.

The framework is also shown to be easily extensible to integrate various texture-based techniques for analysis of CFD vector data. Progressive algorithms, such as those implementing texture advection, can be easily integrated in the framework, allowing the application to stay "up-to-date" with current data visualization techniques. Incorporating such methods could improve the current vector field analysis capabilities of a CFD visualization application. Animation and depth add visual queues to the visualization, providing intuitive insight to the flow field.

Hardware accelerated advection algorithms, such as 3D-GPU texture advection (Weiskopf, 2004), allow the user to visualize normally time-consuming techniques such as streamlines nearly instantaneously. Also, by combining interactive exploration techniques such as dye and noise injection, the user is able to visualize interesting flow field characteristics on a large and small scale simultaneously.

5.1 Future work: Preprocessor

As mentioned earlier, to create the textures used as the basis for the volume rendering and therefore the framework, the preprocessor resamples the original dataset. Although the framework is not dependent on the preprocessor, improvement can be made to provide better comparisons to the current framework. The method presented here for resampling is only effective in specific cases where the data in the dataset is spaced relatively evenly throughout the bounding volume of the dataset, or the magnitude of the data is evenly distributed through the range of the magnitude. The "structured" resampling method is purposely chosen in this work as a "first-pass," i.e., only to develop. An initial idea for improvement is to divide the original dataset bounding volume into sub-volumes in the preprocessor to create smaller sub-datasets in respect to the entire bounding volume. By applying the original texture resolution to the sub-volumes, the sampling frequency for each sub-volume would be increased, producing more accurate textures. VE-Xplorer currently supports investigation of multiple datasets, so the sub-volumes could still be investigated as separated datasets.

5.2 Future work: Transfer functions

In the existing basic framework, any scalar data is read in and converted to RGBA values in the texture manager. This is a limitation that removes the user's ability to adjust the scalar visualizations once the texture is loaded. The problem can be seen in a dataset where the range of values is large but most of the data values are concentrated in a much smaller range. This problem is compounded by the preprocessor sampling algorithm problem. A simple solution to the problem would be to store the scalar data directly in an ALPHA texture, and then apply a transfer function that returns RGBA values for each fragment based on a user-adjustable scalar range.

Iso-surfaces currently are not implemented in the proposed framework, but a separate shader could be written to easily handle this. The shader would read in a scalar property as an ALPHA texture and use a transfer function that describes the "iso-value" the user desires to extract. The value in the property texture would be used to perform a dependent texture lookup in the transfer function. If the value lies in the "iso-value" range of the transfer function, appropriate RGBA values would be returned; otherwise, the fragment is not colored. Transfer function development is also needed to fully implement the advection algorithm presented in (Weiskopf, 2004).

5.3 Future work: Interface

For the proposed framework to be effective, a GUI interface needs to be developed. VE-Conductor currently handles user communication with VE-Xplorer, so it would need to be extended. Most of the scalar interaction can be handled through the current interface. As shader capabilities, mainly transfer functions, are added to the texture-based interface, the accompanying GUIs will need to be developed. Similar interfaces will need to be developed for the currently implemented vector analysis algorithm (Weiskopf, 2004). When fully implemented, the user will be able to inject up to two noise materials and a dye emitter in the flow field for analysis. The user should also be able to adjust properties of the injection materials as discussed in the algorithm (Weiskopf, 2004).

5.4 Future work: Multiple scene graphs

The proposed framework is currently implemented in one scene graph API. VE-Xplorer currently fully supports visualization in two scene graph APIs, so the framework should support two as well.

5.5 Limitations

The proposed framework has only one limitation: the support for three-dimensional textures is required. Volume rendering can be accomplished by rendering stacks of two-dimensional textures, but visualization quality is affected. The current framework does not support volume rendering via two-dimensional textures simply because most consumer graphics cards have supported three-dimensional textures since the release of OpenGL 1.2 (current release is 1.5).

Appendix : Framework interface

The following C++ classes represent the interface developed for implementing the framework proposed in this work:

1. cfdTextureManager → texture manager class
2. cfdTextureDataset → texture dataset class
3. cfdVolumeVisualizationNode → volume visualization class
4. cfdVolumeVisNodeHandler → volume visualization handler class
5. cfdShaderManager → shader manager class
6. cfdTextureBasedVisHandler → texture-based visualization handler

cfdTextureManager

```
#ifndef _CFD_TEXTURE_MANAGER_H_
#define _CFD_TEXTURE_MANAGER_H_
#ifdef VE_PATENTED
#ifdef WIN32
#include <windows.h>
#endif
#include <iostream>
#include <vector>

class cfdTextureManager{
public:
  cfdTextureManager();
  cfdTextureManager(const cfdTextureManager& tm);
  virtual ~cfdTextureManager();

  enum DataType{SCALAR,VECTOR};
  enum PlayMode{PLAY,STOP};

  //add a vector field from a file
  void addFieldTextureFromFile(char* textureFile);

  void setPlayMode(PlayMode mode){_mode = mode;}
  //forwardBackward == -1 backward
  //forwardBackward == 1 forward
  void setDirection(int forwardBackward);

  //set the current frame
  void SetCurrentFrame(unsigned int whichFrame);

  float* getBoundingBox(){return _bbox;}
  int timeToUpdate(double curTime,double delay);

  //get the vector field at a given timestep
  unsigned char* dataField(int timeStep){return_dataFields.at(timeStep);}

  //get the next vector field
  unsigned char* getNextField(/*int plusNeg*/);
  unsigned int getNextFrame();

  //get the number of vector fields
  int numberOfFields(){return _dataFields.size();}

  //the resolution of the fields
```

```
  int* fieldResolution(){return _resolution;}

  //the current frame
  unsigned int GetCurrentFrame();

  //the data ranges
  float* dataRange(){return _range;}
  float* transientRange(){return _transientRange;}
  DataType GetDataType(int whichField){return _types.at(whichField);}

  //equal operator
  cfdTextureManager& operator=(const cfdTextureManager& tm);

protected:
  int _curField;
  int* _resolution;
  std::vector<DataType> _types;
  float _bbox[6];
  float _range[2];
  float _transientRange[2];
  std::vector<unsigned char*> _dataFields;
  double _prevTime;
  int _direction;
  PlayMode _mode;
};
#endif
#endif // _CFD_TEXTURE_MANAGER_H_
```

cfdTextureDataset

```
#ifndef CFD_TEXTURE_DATA_SET_H
#define CFD_TEXTURE_DATA_SET_H
#ifdef VE_PATENTED
#ifdef _OSG

class cfdVolumeVisualization;
class cfdTextureManager;
#include <vector>
#include <map>
#include <iostream>
#include <string>

class TextureDataInfo{
public:
  TextureDataInfo();
  TextureDataInfo(const TextureDataInfo& tdi);
  ~TextureDataInfo();
  void SetName(std::string name);
  void SetTextureManager(cfdTextureManager* tm);

  const char* GetName();
  cfdTextureManager* GetTextureManager();
  TextureDataInfo& operator=(const TextureDataInfo& tdi);
protected:
  std::string _name;
  cfdTextureManager* _tm;

};

class cfdTextureDataSet{
public:
  cfdTextureDataSet();
  virtual ~cfdTextureDataSet();

  void SetActiveScalar(char* name);
  void SetActiveVector(char* name);
  void SetFileName(char* name);
  void CreateTextureManager(char* textureDescriptionFile);
  void AddScalarTextureManager( cfdTextureManager*, const char* );
  void AddVectorTextureManager( cfdTextureManager*, const char* );

  int FindVector(char* name);
  int FindScalar(char* name);
```

```
  cfdTextureManager* GetActiveTextureManager();
  cfdVolumeVisualization* GetVolumeVisNode();
protected:
  unsigned int _nScalars;
  unsigned int _nVectors;
  char* _fileName;
  cfdVolumeVisualization* _volVisNode;
  cfdTextureManager* _activeTM;

  typedef std::vector<TextureDataInfo*> TextureDataList;

  std::vector<std::string> _scalarNames;
  std::vector<std::string> _vectorNames;
};
#endif
#endif
#endif
```

cfdVolumeVisualizationNode

```
#ifndef CFD_VOLUME_VISUALIZATION_H
#define CFD_VOLUME_VISUALIZATION_H
#ifdef VE_PATENTED
class cfdGroup;
#ifdef _PERFORMER
#elif _OPENSG
#elif _OSG
namespace osg
{
  class Node;
  class Geometry;
  class Texture1D;
  class Texture3D;
  class TexGen;
  class TexEnv;
  class Geode;
  class ClipNode;
  class TexGenNode;
  class Material;
  class Shape;
  class Image;
  class Switch;
  class StateSet;
  class Group;
  class BoundingBox;
  class Billboard;
}
class cfdTextureMatrixCallback;
#include <osgUtil/CullVisitor>
#include <osg/TexMat>
#include <osg/Vec3>
#include "cfdUpdateTextureCallback.h"
#include "cfdTextureManager.h"
#ifdef CFD_USE_SHADERS
#include "cfdUpdateableOSGTexture1d.h"
#endif
class cfdVolumeVisualization{
public:
   cfdVolumeVisualization();
   cfdVolumeVisualization(const cfdVolumeVisualization&);
   virtual ~cfdVolumeVisualization();

   enum VisMode{PLAY,STOP};
```

```
  enum Direction{FORWARD,BACKWARD};
  enum ClipPlane{XPLANE=0,YPLANE,ZPLANE,ARBITRARY};

  void SetPlayDirection(Direction dir);
  void SetPlayMode(VisMode mode);
  void SetSliceAlpha(float alpha = .5);
  void SetVeboseFlag(bool flag);
  void SetShaderDirectory(char* shadDir);
#ifdef _OSG
  void SetStateSet(osg::StateSet* ss);
  void SetState(osg::State* state);
  void Set3DTextureData(osg::Texture3D* texture);
  void SetBoundingBox(float* bbox);
  void SetNumberofSlices(int nSlices = 100);
  void SetTextureManager(cfdTextureManager* tm);
  void SetCurrentTransientTexture(unsigned int ct);
  void DisableShaders();
  void CreateNode();
  void AddClipPlane(ClipPlane direction,double* position);
  void RemoveClipPlane(ClipPlane direction);
  void UpdateClipPlanePosition(ClipPlane direction,double* newPosition);

  bool isCreated(){return _isCreated;}
  unsigned int GetCurrentTransientTexture();
  cfdUpdateTextureCallback* GetUpdateCallback(){return _utCbk;}
  osg::Vec3f GetBBoxCenter(){return _center;}
  float* GetTextureScale(){return _scale;}
  osg::ref_ptr<osg::StateSet> GetStateSet();
  osg::ref_ptr<osg::Texture3D> GetTextureData();
  osg::ref_ptr<osg::Switch> GetVolumeVisNode();
  osg::ref_ptr<osg::Group> GetDecoratorAttachNode();
  cfdVolumeVisualization& operator=(const cfdVolumeVisualization& rhs);
#endif
protected:
  VisMode _mode;
  Direction _traverseDirection;
  bool _verbose;
  bool _isCreated;
  bool _useShaders;
  bool _volShaderIsActive;
  bool _transferShaderIsActive;
  unsigned int _nSlices;
  unsigned int _tUnit;
  float _alpha;
  void _createVisualBBox();
```

```
  void _createClipCube();
  void _buildGraph();
  void _createClipNode();
  void _createStateSet();
  void _attachTextureToStateSet(osg::StateSet* ss);
  void _createTexGenNode();
  void _createVolumeSlices();
  void _buildAxisDependentGeometry();
  void _buildSlices();

  char* _shaderDirectory;
  cfdTextureManager* _tm;
  osg::Vec3 _center;
  float _transRatio[3];
  float _diagonal;
  float _scale[3];
#ifdef _OSG

  osg::ref_ptr<osg::Switch> _volumeVizNode;
  osg::ref_ptr<osg::TexGenNode> _texGenParams;
  osg::BoundingBox* _bbox;
  osg::ref_ptr<osg::ClipNode> _clipNode;
  osg::ref_ptr<osg::StateSet> _stateSet;
  osg::ref_ptr<osg::Billboard> _billboard;

  osg::ref_ptr<osg::Group> _noShaderGroup;
  osg::ref_ptr<osg::Group> _decoratorAttachNode;
  osg::ref_ptr<osg::Texture3D> _texture;
  osg::ref_ptr<osg::Image> _image;
  osg::ref_ptr<osg::State> _state;
  cfdUpdateTextureCallback* _utCbk;
#endif

};
#endif//OSG
#endif// CFD_VOLUME_VISUALIZATION_H
#endif
```

cfdVolumeVisNodeHandler

```
#ifndef CFD_VOLUME_VIZ_NODE_HANDLER_H
#define CFD_VOLUME_VIZ_NODE_HANDLER_H
#ifdef VE_PATENTED
#ifdef _OSG
#include <osg/BoundingBox>
#include <osg/ref_ptr>
namespace osg
{
  class Group;
  class Switch;
  class TexGenNode;
}

class cfdTextureManager;
class cfdVolumeVisNodeHandler{
public:
  cfdVolumeVisNodeHandler();
  cfdVolumeVisNodeHandler(const cfdVolumeVisNodeHandler& vvnh);
  virtual ~cfdVolumeVisNodeHandler();

  void SetSwitchNode(osg::Switch* vvn);
  void SetAttachNode(osg::Group* attachNode);
  void SetCenter(osg::Vec3f center);
  void SetTextureScale(float* scale,bool isInverted = true);
  void SetTextureManager(cfdTextureManager* tm);
  void SetBoundingBox(float* bbox);
  void SetBoundingBoxName(char*name);
  void SetDecoratorName(char* name);
  bool IsThisActive();
  virtual void Init();
  void TurnOnBBox();
  void TurnOffBBox();
  void EnableDecorator();

  cfdVolumeVisNodeHandler& operator=(const cfdVolumeVisNodeHandler& vvnh);
protected:
  void _createVisualBBox();
  //set up the stateset for the decorator
  virtual void _setUpDecorator()=0;
  virtual void _applyTextureMatrix()=0;
  virtual void _updateTexGenUnit(unsigned int unit=0);
  void _createTexGenNode();
  unsigned int _whichChildIsThis;
```

```
  unsigned int _whichTexture;
  bool _autoTexGen;
  cfdTextureManager* _tm;
  osg::ref_ptr<osg::Switch>_bboxSwitch;
  osg::ref_ptr<osg::Group> _visualBoundingBox;
  osg::ref_ptr<osg::Switch> _vvN;
  osg::ref_ptr<osg::Group> _decoratorGroup;
  osg::ref_ptr<osg::Group> _byPassNode;
  osg::ref_ptr<osg::TexGenNode> _texGenParams;
  osg::BoundingBox _bbox;
  osg::Vec3f _center;
  float _scale[3];
};

#endif // _OSG
#endif// CFD_VOLUME_VIZ_NODE_HANDLER_H
#endif
```

cfdOSGShaderManager

```
#ifndef CFD_OSG_SHADER_MANAGER_H
#define CFD_OSG_SHADER_MANAGER_H
#ifdef VE_PATENTED
#ifdef _OSG
#include <osg/StateSet>
#ifdef CFD_USE_SHADERS

class cfdOSGShaderManager{
public:
  cfdOSGShaderManager();
  cfdOSGShaderManager(const cfdOSGShaderManager& sm);
  virtual ~cfdOSGShaderManager();

  virtual void Init() = 0;

  void SetShaderDirectory(char* dir);
  osg::StateSet* GetShaderStateSet();
  void SetBounds(float* bounds);
  unsigned int GetAutoGenTextureUnit(){return _tUnit;}

  virtual cfdOSGShaderManager& operator=(const cfdOSGShaderManager& sm);
protected:
  virtual void _setupCGShaderProgram(osg::StateSet* ss,
                                     char* progName,
                                     char* funcName);
  osg::ref_ptr<osg::StateSet> _ss;
  char* _shaderDirectory;
  unsigned int _tUnit;
  float* _bounds;
};
#endif //CFD_USE_SHADERS
#endif // _OSG
#endif
#endif// CFD_OSG_SHADER_MANAGER_H
```

cfdTextureBasedVizHandler

```
#ifndef CFD_TEXTURE_BASED_MODEL_HANDLER_H
#define CFD_TEXTURE_BASED_MODEL_HANDLER_H
#ifdef VE_PATENTED
#include <vpr/Util/Singleton.h>

class cfdDCS;
class cfdGroup;
class cfdCursor;
class cfdNavigate;
class cfdCommandArray;
class cfdSwitch;
class cfdTextureManager;
#include <vector>
#ifdef _PERFORMER
#elif _OPENSG
#elif _OSG
namespace osgUtil { class SceneView; }
class cfdPBufferManager;
class cfdVolumeVisualization;
class cfdTextureDataSet;
class cfdVolumeVisNodeHandler;

#ifdef CFD_USE_SHADERS
class cfdVectorVolumeVisHandler;
class cfdScalarVolumeVisHandler;
#endif

class cfdTextureBasedVizHandler: public vpr::Singleton< cfdTextureBasedVizHandler >
{
  public:
     void PreFrameUpdate( void );
     void CleanUp( void );
     void SetParameterFile(char* paramFile);
     void SetCommandArray(cfdCommandArray* cmdArray);
     void SetWorldDCS(cfdDCS* dcs);
     void SetParentNode(cfdGroup* parent);
     void SetNavigate(cfdNavigate* navigate);
     void SetCursor(cfdCursor* cursor);
     void SetActiveTextureDataSet(cfdTextureDataSet* tdset);
     void ViewTextureBasedVis(bool trueFalse);

#ifdef CFD_USE_SHADERS
     void SetPBuffer(cfdPBufferManager* pbm);
```

```
    void PingPongTextures();
#endif

    cfdPBufferManager* GetPBuffer();
    //bool InitVolumeVizNodes( void );
    cfdVolumeVisualization* GetVolumeVizNode(int index);
    cfdVolumeVisualization* GetActiveVolumeVizNode( void );

  protected:
    void _updateScalarVisHandler();
    void _updateVectorVisHandler();

    char* _paramFile;
    cfdCommandArray* _cmdArray;
    cfdDCS* _worldDCS;
    cfdNavigate* _nav;
    cfdCursor* _cursor;
    cfdTextureDataSet* _activeTDSet;
    cfdTextureManager* _activeTM;

    //std::vector<cfdVolumeVisualization*> _volumeVisNodes;
    cfdVolumeVisualization* _activeVolumeVizNode;
    cfdGroup* _parent;
    cfdPBufferManager* _pbm;
    osgUtil::SceneView* _sceneView;
    cfdVolumeVisNodeHandler* activeVisNodeHdlr;

#ifdef CFD_USE_SHADERS
    cfdVectorVolumeVisHandler* _vvvh;
    cfdScalarVolumeVisHandler* _svvh;
#endif

    //cfdSwitch* _visOptionSwitch;
    float* _currentBBox;
    bool _cleared;
    bool _textureBaseSelected;
private:
    // Required so that vpr::Singleton can instantiate this class.
    friend class vpr::Singleton< cfdTextureBasedVizHandler >;
    cfdTextureBasedVizHandler( void );
    ~cfdTextureBasedVizHandler( void ){ ; }// Never gets called, don't implement
};
#endif //OSG
#endif //
#endif// CFD_TEXTURE_BASED_VIZ_HANDLER_H
```

References

Bryden, K., (2005) "VE-Suite". Accessed 4/4/2005. http://www.vrac.iastate.edu/%7Ekmbryden/VE-Suite.htm.

Cabral, B., Cam, N. & Foran, J. (1994) "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *Proceedings of ACM Symposium on Volume Visualization*, pg 91-98.

Fang, S., Biddlecome, T. & Tuceryan, M. (1998) "Image-Based Transfer Function Design for Data Exploration," *Proceedings of IEEE conference on Visualization,* pg. 319-326.

Hladuvka, J., Konig, A. & Groller,E. (2000) "Curvature-Based Transfer Functions for Direct Volume Rendering," In *Spring Conference on Computer Graphics 2000* , pg. 58-65.

Kitware, Accessed 3/18/2005. <http://www.vtk.org>.

Larmee, R.S., Hauser, H., Doleisch, H., Vrolijk, B., Post, F.H., & Weiskopf, D. (2004) "The State of the Art in Flow Visualization: Dense and Texture-Based Techniques," *Computer Graphics Forum*, Vol. 23, No. 2, 203-221.

Rajlich, P. (1998) "An object oriented approach to developing visualization tools portable across desktop and virtual environments," *Master's Thesis, University of Illinios at Urbana-Champaign.*

Shen, H., & Kao, D.L. (1998) "A New Line Integral Convolution Algorithm for Visualizing Time-Varying Flow Fields," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 4, No.2, pg. 98-108.

Tannehill, J.C., Anderson, D.A., & Pletcher, R.H., (1997) *Computational Fluid Mechanics and Heat Transfer*. Philadelphia, PA: Taylor & Francis.

Telea, A., & van Wijk, J.J. (2003) "3D-IBFV: Hardware-Accelerated 3D Flow Visualization," *Proceedings of the 14th IEEE Visualization Conference,* pg. 233-240.

Van Wijk, J.J. (2002) "Image Based Flow Visualization," *ACM Transactions on Graphics,* Vol 21, No. 23, pg 745-754.

Van Wijk, J.J. (1991) "Spot noise texture synthesis for data visualization," *Proceedings of ACM SIGGRAPH,* Vol. 25, No. 4, 309-318.

Weiskopf, D., & Ertl, T. (2004) "GPU-Based 3D Texture Advection for the Visualization of Unsteady Flow Fields," *WSCG Short Communication Papers Proceedings*, pg. 181-188.

Weiskopf, D. (2004) "Dye Advection Without the Blur: A Level-Set Approach for Texture-Based Visualization of Unsteady Flow," *Computer Graphics Forum*, Vol. 23, No. 3, pg. 479-488.

Wilson, O., Van Gelder, A. & Wilhelms, J. (1994) "Direct Volume Rendering via 3D Textures." Technical Report UCSC-CRL-94-19. Univ. of Calif. Santa Cruz

Woo, M., Neider, J., Davis, T. & Shreiner D. (1999) *OpenGL Programming Guide.* Boston: Addison- Wesley.

Wynn, C., (2001) "Using P-Buffers for Off-Screen Rendering in OpenGL," *Nvidia* 8/9/2001. Accessed 8/18/2004. <http://developer.nvidia.com/object/PBuffers_for_OffScreen.html>.

Acknowledgements

This work was performed at Ames Laboratory under Contract No. W-7405-Eng-82 with the U.S. Department of Energy. The United States government has assigned the DOE Report number IS-T 2575 to this thesis.