University of Nevada
Reno

Vesuvius: Interactive Atmospheric Visualization in a Virtual Environment

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science

by

Michael P. Dye

Dr. Frederick C. Harris, Jr., Thesis Advisor

December, 2007

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

MICHAEL P. DYE

entitled

Vesuvius: Interactive Atmospheric Visualization In A Virtual Environment

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris Jr, Ph.D., Advisor

Sergiu Dascalu, Ph.D., Committee Member

Darko Koracin, Ph.D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

December, 2007

Acknowledgments

The work shown in this thesis has been sponsored by the Department of the Army, Army Research Office; the contents of the information does not reflect the position or policy of the federal government. This work is funded by the CAVE Project (ARO# N61339-04-C-0072) at the Desert Research Institute.

I would like to first thank my committee, in particular my advisor Dr. Frederick C. Harris Jr. Without his guidance and support I would not have imagined that I'd be here. Also to the rest of my committe, Dr. Sergiu Dascalu and Dr. Darko Koracin, for their help in making this thesis possible. You have my sincerest of graditude. I would also like to thank my family for their love and support. And last but certainly not least, my friends and fellow co-researchers for their well timed distractions during the coding/writing process.

Abstract

Atmospheric simulation is an important means of understanding the environment around us. Through the collection of large amounts of atmospheric data and computer modeling one can predict how variuos particulates such as dirt, smog, and fire can affect our cities and overall public health. However, gleaning insight from numerical data is a teduous, laborious, and time-consuming task with a very high potential for failure. Visual representations of the data can make identifying and studying atmospheric data very intuitive while reducing the chance or error. Virtual reality has long been used as a way of creating realistic visual simulations to help aid in interpreting large and complex data sets. This thesis describes a volumetric rendering library whose purpose is to address the omissions in current atmospheric rendering applications. Vesuvius allows users to visualize various large atmospheric data sets concurrently from a variety of viewpoints while interacting with normal OpenGL geometry in addition to giving the user control over the playback of temporal data.

Contents

Acknowledgements **i**

Abstract **ii**

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**

2 Background **4**
2.1 Atmospheric Simulation 4
2.2 Volume Rendering 5
2.2.1 Raycasting 7
2.2.2 Splatting 8
2.2.3 Shear-Warp 9
2.2.4 Texture-Based 10
2.2.5 Shader-Based Techniques 12
2.2.6 Large Data Sets 15
2.2.7 Transfer Functions 15
2.3 Virtual Reality 17
2.3.1 Depth Cues 18
2.3.2 Stereoscopic Display Environments 19
2.3.3 Input and Immersion 19
2.3.4 Toolkits 21
2.4 Existing Applications 23

3 Volume Library Software Design **25**
3.1 Requirements 25
3.1.1 Functional Requirements 26
3.1.2 Non-Functional Requirements 26
3.2 Use Cases 27
3.3 Architecture 28
3.3.1 Preprocessor Subsystem 28

3.3.2 Renderer Subsystem 29
3.3.3 Transfer Function Subsystem 29
3.3.4 File Subsystem 29

4 Volume Library Implementation 30
4.1 Preprocessor Implementation 30
4.2 Volume Renderer Implementation 33
4.3 Transfer Function Manager Implementation 35
4.4 File Loader Implementation 37
4.5 Interface 38
4.6 Visualization System 39
4.7 Sample Case 39

5 Conclusions and Future Work 44
5.1 Conclusion 44
5.2 Future Work 45

A User Manual 48
A.1 Preprocessor 48
A.2 Renderer 48
A.3 Transfer Function Manager 49
A.4 File Loader 49
A.5 Rendering 49

Bibliography 52

List of Figures

2.1 Example MM5 file structure. 5
2.2 Raycasting through a volume [9]. 7
2.3 Comparison between slicing distances (left) and raycasting distances (right) [45]. 8
2.4 Transforming volume to sheared object space for parallel projection [26]. 10
2.5 Transforming volume to sheared object space for perspective projection [26]. 11
2.6 Simply bricking the data results in incorrect interpolation between bricking boundaries (a). Duplicating last data across brick boundaries results in correct interpolation between bricking boundaries (b) [9]. . 12
2.7 The process a graphics card goes through to render images [9]. 13
2.8 Fixed pipline [40]. 14
2.9 Programmable pipline [40]. 14
2.10 Pre-classification (a). Post-classification (b) [9]. 16
2.11 1D transfer function results in artifacts between the dentin and the surrounding air (a). 2D transfer function accounts for this and allows the overlapping boundary values to be colored differently (b) [9]. . . . 17
2.12 Head-mounted display [42] 20
2.13 Multi screen projection system 21
2.14 Intersense tracked input system [17]. 22
2.15 Intersense head tracking system on shutter glasses [17]. 22
2.16 Atmospheric dataset shown in Vis5D. 24
2.17 Atmospheric dataset shown in Cave5D. 24

3.1 Vesuvius use case diagram. 27
3.2 Class structure of the Vesuvius volume rendering library. 28

4.1 Vesuvius file format. 31
4.2 Normal structured grid with data starting at points denoted by green line (a). MM5 file where data starts at points following the contours of the terrain (again denoted by green line (b) 32
4.3 Data distribution of Vis5D files. Notice how the data is more dense the closer it gets to ground level. 33

4.4 MM5 file rendered through Vesuvius. 35
4.5 Raycasting with default step size (a). Raycasting with increased step size (b) notice how this change requires the transfer function to be edited. 36
4.6 Creating a transfer function and storing it in the transfer function manager. 37
4.7 Root menu of Pompeii. 38
4.8 Menu that enables and disables the different volumetric layers a file may have. 39
4.9 Transfer function menu. 40
4.10 Transfer function editor menu. 41
4.11 Playback menu. 42
4.12 Process of converting an MM5 file to a Vesuvius file. 42
4.13 Vesuvius with the "Q" and "T" layers enabled. 42
4.14 "U" layer with default coloring (a). "U" layer with adjusted coloring (b). 43

A.1 Steps taken to convert an MM5 file into a Vesuvius compatible file. . 48
A.2 Initializing the renderer. 49
A.3 Creating a transfer function and adding it to the transfer function manager. 50
A.4 Loading a Vesuvius file. 50
A.5 Rendering data sets in Vesuvius 51

List of Tables

2.1 Sub header information. 6

3.1 Functional requirements for Vesuvius. 26
3.2 Non-Functional requirements for Vesuvius. 26

4.1 File header information. 31

Chapter 1

Introduction

Atmospheric simulation is an important means of understanding the environment around us. Through the collection of large amounts of atmospheric data and computer modeling one can predict how various particulates such as dirt, smog, and fire can affect our cities and overall public health. Everything from temperature to pressure to wind speed as well as a host of other data about the atmosphere is collected and is put through computer models in an attempt to predict how atmospheric changes affect the environment at large. However, the sheer number of variables involved in determining the weather make accurate weather forecasting a daunting and computationally time consuming task. One flaw in the atmospheric model can propagate through an entire simulation and rendering the final forecast useless. Because of this, a lot of time and effort is put into making accurate and comprehensive atmospheric models.

Studying both the data collected and the final computer forecast is essential to helping the scientists identify and study trends and interaction between disparate atmospheric forces as well as to validate the end forecast. However, gleaning insights from numerical data is a tedious, laborious, and time-consuming task with a very high potential for error. For example, the raw data does not give any clues regarding the shape of a given dataset. Similar problems arise when attempting to study the raw data in an attempt to find how various terrain formations interact with and affect the atmosphere. Since raw data can be hard to conceptualize, particularly when the size of the data sets can be hundreds of megabytes if not gigabytes, an alternative method of interpreting atmospheric data is needed. Visual representations of the

data make identifying and studying atmospheric data very intuitive while reducing the chance of error to the amount of error involved in converting the data from an atmospheric model to a graphical model. However, while research has been done on related fields such as volumetric rendering [4, 26, 28, 29, 48, 50] little emphasis has been put on combining these areas of research into a useful package that allows the atmospheric scientist to visualize not only the varying data but the interactions between the various data with the goal of displaying the data at interactive frame rates.

While looking at a graphical representation of atmospheric data proves to be an invaluable tool, it can often be difficult to grasp the full scope of what the data is showing from a simple desktop display. Virtual reality technology allows us to accurately and realistically model atmospheric data in a meaningful way for the user. Virtual reality has long been used as a way of creating realistic visual simulations to help aid in interpreting large and complex data sets. Recent advances in visualization and supporting technologies now offer the possibility of creating realistic, real-time atmospheric visualizations for research. By combining virtual reality technology with atmospheric simulation users are able to visually conceptualize large amounts of atmospheric data in an interactive way. However, while a package exists that combines the volumetric rendering of temporal atmospheric dataset with virtual reality [49] no package exists which allows for multiple volumetric data sets to be displayed at the same time. Furthermore, no package exists as a library that allows the rendering of other relevant data (such as topographic information or vehicular interaction).

This thesis describes a volumetric rendering library called Vesuvius whose purpose is to address the omissions in current atmospheric rendering applications. Vesuvius is intended to allow users to visualize various large atmospheric data sets from a variety of viewpoints and to allow for playback of atmospheric data sets that contain temporal information. In addition, Vesuvius allows the visualization of multiple volumetric data sets concurrently allowing the atmospheric scientist to study the interaction between the differing particulate data.

The remainder of this thesis is structured as follows: Chapter 2 discusses background information on atmospheric simulation as well as volume visualization. Chapter 3 presents Vesuvius as a solution to rendering atmospheric data in virtual reality in real-time. Chapter 4 presents the results of Vesuvius as a viable atmospheric modeling library. Finally, conclusions and future work are presented in Chapter 5.

Chapter 2

Background

2.1 Atmospheric Simulation

Atmospheric modeling and simulation is done using numerical models. Because of the sheer complexity of the model supercomputers are employed to do the calculations. The most common numerical model used in atmospheric modeling is the National Center for Atmospheric Research (NCAR)/Pennsylvania State University Mesoscale Model Version 5 (MM5) [11]. Numerical models such as the MM5 model are capable of simulating a wide range of atmospheric attributes such as mass attributes (i.e. temperature and humitidy) and momentum attributes (i.e. wind U, V, W fields). In addition to MM5 there are specialized models meant to simulate specific atmospheric attributes. One such example is the Comprehensive Air Quality Model with Extensions (CAMx) which is used to simulate the behavior of atmospheric chemicals and is commonly used in air pollution studies.

These models can be used to study past atmospheric phenomena or used as atmospheric forecasts when they are initialized with archived atmospheric data and current atmospheric data respectively. These models are also very scalable, allowing for domains ranging from a few kilometers to tens of kilometers in the same model. In the case of the MM5 format, further refining of the physics has allowed the horizontal domain to be reduced to under a kilometer.

MM5 File Format

Because the MM5 numerical model is the standard atmospheric model, this thesis focuses exclusively on visualizing this model. The MM5 format is designed to be general, allowing for addition of atmospheric attributes without the need to redesign the file. Figure 2.1 illustrates the overall structure of the MM5 file. It alternates between a flag and file data with a flag of 0 being the big header (the file header), 1 for the sub-header which contains attribute data, or 2 for the end of a time period. The big header contains basic file information contained in four arrays: an integer array, a floating-point array, and two character arrays that assign meaning to the values in the number arrays with Table 2.1 illustrating the data contained within the sub-header.

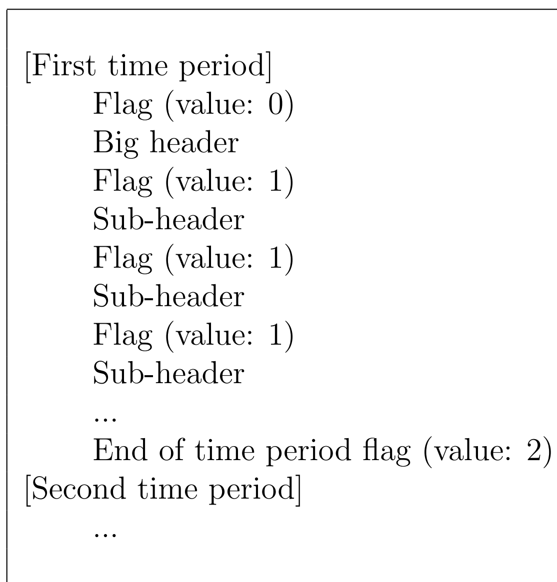


Figure 2.1: Example MM5 file structure.

2.2 Volume Rendering

Volume visualization involves rendering volumetric data sets, which are usually stored as 3-dimensional structured grids. There are two volume rendering methodologies used to accomplish this. First is method of volume rendering is referred to as indirect

Name	Size	Description
ndim	int	Field dimensions (1D, 2D, 3D, etc)
start_index	int[4]	Starting indices in the field array
end_index	int[4]	Ending indices in the field array
xtime	float	Field forecast time (in minutes)
staggering	char[4]	Whether the field is a dot (D) or cross point (C)
ordering	char[4]	Order of the array dimensions
current_date	char[24]	Date of field
name	char[8]	Name of field
unit	char[25]	Unit of measure used in field
description	char[46]	Field description

Table 2.1: Sub header information.

volume rendering (IVR). Indirect volume rendering involves converting the volumetric point cloud into sets of polygonal iso-surfaces and rendering those as one would normal mesh data (such as Marching Cubes [31]). The second method is referred to as direct volume rendering (DVR), which involved rendering the point cloud directly to the screen and thus skipping intermediate steps such as iso-surface generation.

Indirect volume rendering assumes several things about the volumetric data set. Specifically, it assumes that iso-surfaces exist and that any such iso-surfaces can be rendered with a reasonable degree of accuracy [32]. Even if the volumetric data conforms to these two assumptions the generated iso-surfaces might be so complex that it might overwhelm the rendering capabilities of the graphics card. Because of the computational cost involved in extracting iso-surfaces (if they exist) and the potential complexity of the iso-surfaces, direct volume rendering may prove to be more efficient.

In addition, there are three classifications of volume rendering algorithms. *Image-order* algorithms involve backwards mapping the image plane onto the volumetric data (usually through the use of something such as rays). *Object-order* algorithms work on the opposite principle. They use the traditional forward mapping to map the data onto the image plane. Lastly, *hybrid* algorithms employ a combination of both of the preceeding types of algorithms. In the rest of this section, we are going

to review the major volume rendering algorithms as well as discuss their advantages and disadvantages.

2.2.1 Raycasting

Raycasting [28, 29] is a widely used image-order volume rendering algorithm for producing high quality images. In raycasting, a ray is shot from the camera through each pixel in the screen as shown in Figure 2.2. At predetermined increments along the ray a sample is taken and front-to-back or back-to-front compositing is performed and the final color and opacity for the pixel is determined. Of the popular volume rendering algorithms, raycasting produces the best images at the cost of using the most computational time. It achieves these high quality images because of how easy it is to implement trilinear filtering, though other interpolation methods can also be used [33]. In addition, unlike slicing based rendering methods the distance between samples is constant which helps avoid artifacts, as illustrated in Figure 2.3.

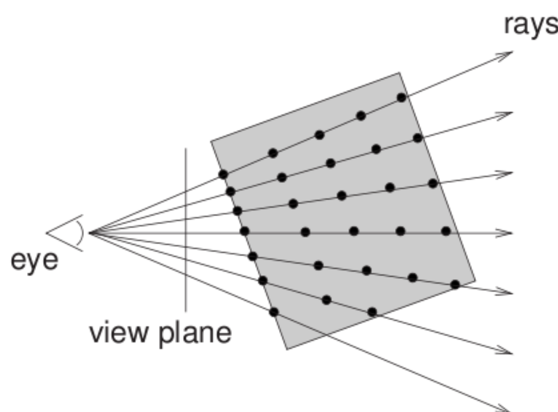


Figure 2.2: Raycasting through a volume [9].

Because a ray has to be traversed, interpolated, and composited for each pixel in the image, raycasting is the most computationally expensive of the volume rendering techniques. Because of the computational costs involved, several acceleration techniques have been created in an attempt to reduce raycasting overhead. The easiest

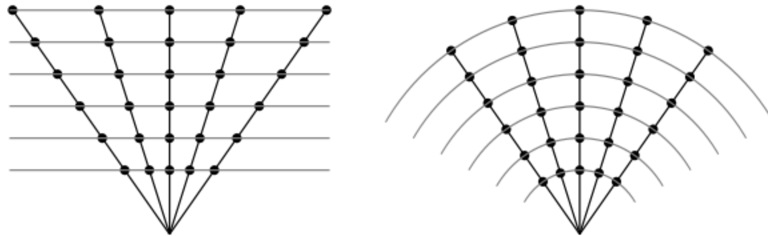


Figure 2.3: Comparison between slicing distances (left) and raycasting distances (right) [45].

method of reducing computation time is called *early ray termination*. In early ray termination rays are terminated once they reach a certain opacity level. Ray sampling can also be used to skip empty spaces and thus further minimize computational costs [51].

Another disadvantage of this pixel-by-pixel approach is that eight samples have to be loaded each time the ray steps forward to perform the trilinear interpolation, one for each corner of the cube surrounding the data point. It does not factor in previous sample data that might still be loaded into memory, which results in a lot of memory access. To minimize this object-oriented approachs to raycasting can be used to access memory in a more regular manner [34]. Various other techniques have been proposed using for commodity hardware that rely heavily on the aforementioned acceleration techniques [21].

2.2.2 Splatting

Splatting [48] is an object-order volume rendering technique designed to address some of the shortcomings of other volumetric rendering algorithms such as raycasting–specifically the high computational cost. Splatting is akin to dropping water balloons filled with paint onto a surface. Each paint filled balloon is a point in the volumetric data and the combination of all the paint splattering is thought of as the final volumetric image. The data set is first mapped to image space and then a *footprint* is used to determine to what extent a particular data point influences the other points

around it. The footprint is the integral of a *kernel* (usually Gaussian) that determines the shape of each splat as well as how much it influences the surrounding points in the final image. A footprint table can be generated to reduce the number of times integrals are computed and thus increasing frame-rate even further. Like raycasting, the volumetric data can be traversed in either a front-to-back or back-to-front order.

However, the speed increase that splatting has over raycasting comes at the sacrifice of image quality. If too small a kernel size is chosen the final image will have gaps, whereas if the kernel size is too large the final image will be blurred. Several techniques have been developed in an attempt to address the downfalls of splatting including reducing image blurring by altering the splattering pipeline [35] and reducing antialiasing through the use of varying reconstruction kernels [46].

2.2.3 Shear-Warp

Shear-warp factorization [26] is recognized as the fastest of the software volume rendering algorithms. Shear-warp is a hybrid algorithm that takes some of the best qualities of both image-order algorithms (early ray termination) and object-order algorithms (little redundant computation). It works by first transforming the volume into an intermediate coordinate system that can be easily mapped to from the object coordinate system. This *sheared object space* allows for efficient projections and is shown in Figure 2.4. For perspective projections, each volume slice must be scaled as well as sheared as shown in Figure 2.5. After the volume is transformed the slices are composited in a front-to-back order using bilinear interpolation which results in an intermediate image in sheared object space. From there the intermediate image is warped to image space which results in the final image.

Shear-warping employs a method of run-length encoding as a preprocessing step which skips transparent voxels. However, to take advantage of data coherence, an encoded volume has to be constructed for each of the three major axes. As a result, shear-warping requires up to three times the memory of other volume rendering algorithms. Furthermore, since bilinear interpolation is used and sampling distances vary,

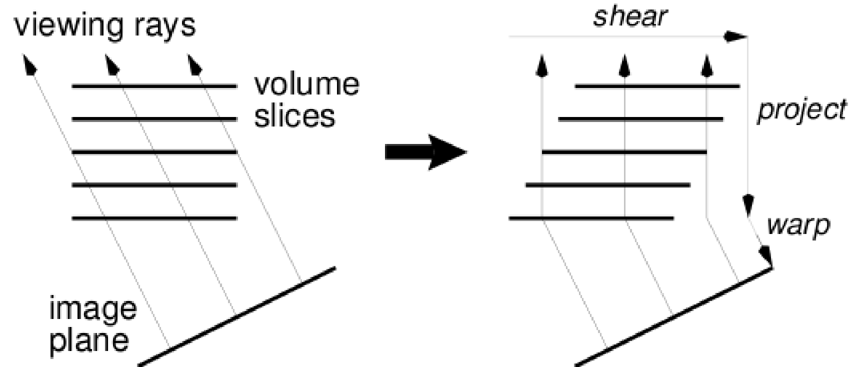


Figure 2.4: Transforming volume to sheared object space for parallel projection [26].

various aliasing effects can be seen depending on the viewing angle. Research has been done to limit these shortcomings including interpolation of intermediate slices and to reduce aliasing and reusing RLE runs for more than one viewing direction [47]. In addition, the shear-warp algorithm has been successfully adapted for use in a virtual environment [39].

2.2.4 Texture-Based

Texture-based volume rendering takes advantage of graphics hardware to accelerate the volumetric rendering process. There are two primary methods of texture-based volume rendering, one using 2D textures [4] and another using 3D textures [50].

2D texture-based volume rendering requires splitting the volume into a stack of slices so it can be stored as texture images. These slices are known as *object-aligned* or *axis-aligned* slices because the slices are defined relative to the object's coordinate system. These slices are then composited in back-to-front order to yield the final image. Among the advantages of a 2D texture-based approach are its simplicity and performance. Because it uses bilinear interpolation performed by the graphics hardware this is a very fast volume rendering algorithm. However, there are several disadvantages to 2D texture-based rendering. First of all slices have to be created for

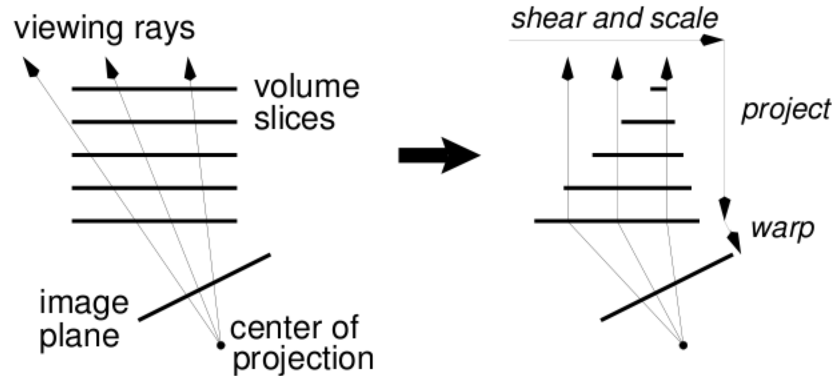
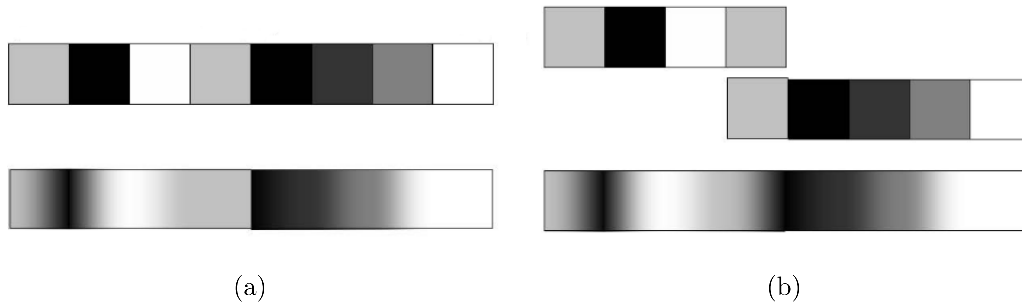


Figure 2.5: Transforming volume to sheared object space for perspective projection [26].

each major axis. This means that three times the memory is required in the graphics card to hold each set of slices. The image quality also suffers from similar drawbacks to that of shear-warping in that aliasing artifacts are often present, particularly when the volume is magnified. In addition, the sampling rate cannot be changed since it is determined by the distance between slices. Intermediate slices can be computed to decrease the distance from one slice to another which would, in effect, increase the sampling rate; however creating more slices also means using up more texture memory on the graphics card to store the textures. Another problem with storing object-aligned slices is that flickering may be detectable when the algorithm switches from one set of slices to another because of the sudden shift in sampling position.

3D texture-based volume rendering solves many of the problems introduced in 2D texture-based volume rendering. The volume is stored into a single 3D texture as opposed to a series of 2D texture slices. This allows the algorithm to take advantage of trilinear interpolation, which enables the creation of slices with an arbitrary orientation. This allows the 3D texture-based algorithm to compute *viewport-aligned* slices, which are slices parallel to the image plane. This eliminates a huge disadvantage of 2D texture-based volume rendering in that only one copy of the data needs to be stored. However, for large data sets 3D texture-based volume rendering is actually

less efficient than 2D texture-based volume rendering. This is because some sort of bricking is necessary to store a data set into a 3D texture. This bricking requirement limits the rendering by the bandwidth between the GPU and the computer memory. In addition, bricking increases the memory requirement for storing the data set because of the extra data point that needs to be duplicated across the brick boundary as shown in Figure 2.6. Brick size also plays an important part in the efficiency of a 3D texture-based approach. If a brick is too small then the duplicate voxels will end up increasing the memory requirements as well as a high number of intersection calculations to compute the viewport-aligned slices. Whereas if the brick is too large, it will be unable to fit into the texture cache.



(a) (b)

Figure 2.6: Simply bricking the data results in incorrect interpolation between bricking boundaries (a). Duplicating last data across brick boundaries results in correct interpolation between bricking boundaries (b) [9].

2.2.5 Shader-Based Techniques

Shaders

The job of the graphics card on modern computers is to take graphical data such as the vertices of a computer model and to translate that into a 2-dimensional image on the screen. This is done by turning the vertices into primitives (such as triangles), culling the vertices, mapping the polygon points to pixel-space, filling the polygon, and shading it as demonstrated in Figure 2.7. Traditionally, this process was fixed in that how the graphics card converted vertex information into the 2-dimensional image was unalterable and mirrored that shown in Figure 2.8.

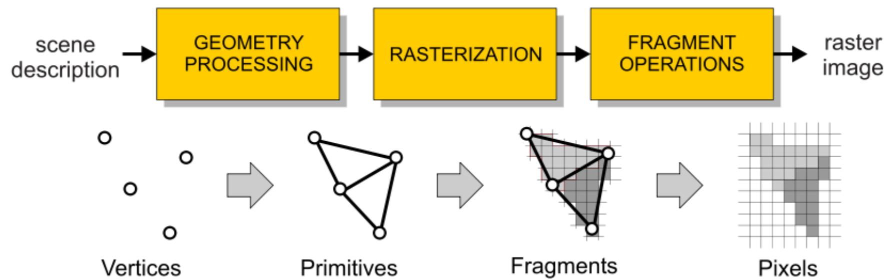


Figure 2.7: The process a graphics card goes through to render images [9].

Shaders introduced the concept of the programmable pipeline as shown in Figure 2.9. The programmable pipeline allows programmers to upload miniature programs that replace certain parts of the pipeline. Currently shaders allow the programmer to replace the fixed functionality of the *vertex processor* and the *fragment processor*.

The vertex processor is primarily responsible for linear transformations of the incoming vertices. It is the vertex processor that converts vertices from object-space to world-space to camera-space to screen-space. From there the vertices are joined together into geometric primitives which are then clipped, culled, and mapped onto the viewport before being handed to the fragment processor.

The fragment processor's responsibility is to decompose the geometric primitives into *fragments*. Each fragment corresponds to a pixel on the screen. The fragment processor then determines the final color of the fragment by interpolating the vertex attributes and the texture samples.

The ability to control parts of the rendering pipeline allows for a number of things to be done with regards to volume rendering in an effort to increase both rendering speed and image quality.

Texture-Based

Texture-based volume rendering using shaders is very similar to standard texture-based volume rendering. For 2D object-aligned texture-based volume rendering the

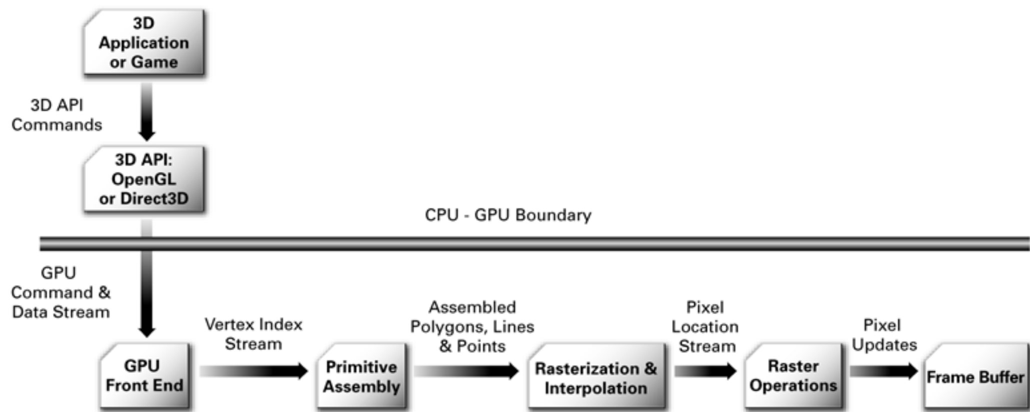


Figure 2.8: Fixed pipline [40].

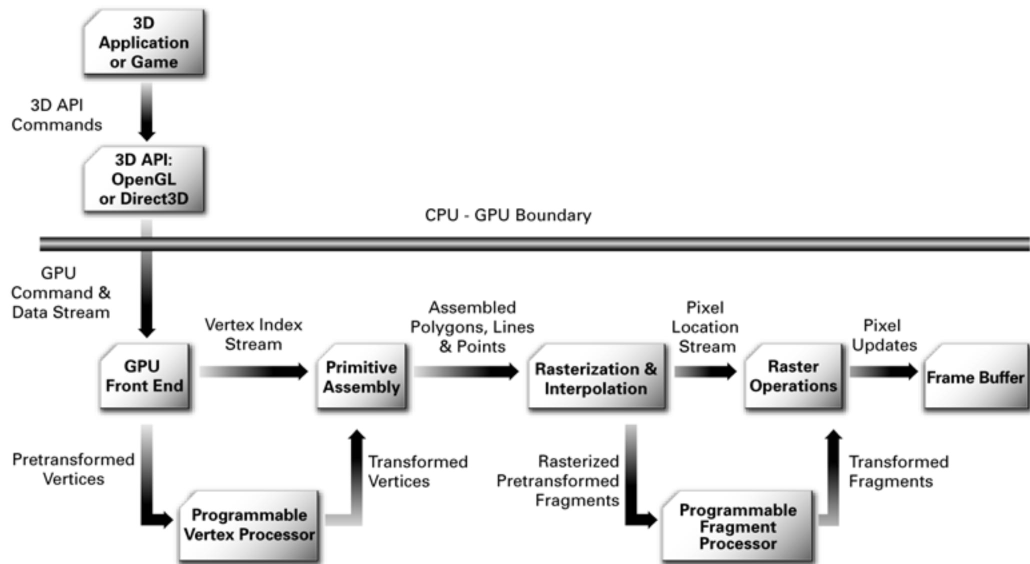


Figure 2.9: Programmable pipline [40].

difference is that the algorithm utilizes a vertex program to calculate the intermediate slices given the first and last slice. For 3D viewport-aligned texture-based volume rendering a vertex program is used to determine the polygon created from the volume's bounding box and a plane parallel to the viewport. What this does is allow the application to calculate other things while the GPU builds the slices.

Raycasting

Because of the parallel nature of the graphics card, raycasting sees the biggest benefit from shaders. Because raycasting requires the computation across the volume for every pixel on the screen, the multiple pixel pipelines offered by modern graphics cards provide a built in way of parallelizing the raycasting process. Because of how limited fragment shader functionality was, GPU-based raycasting was not presented until about 2003 [25]. However, because of how easily portable it is, it has been adapted to a number of environments including virtual environments [23].

2.2.6 Large Data Sets

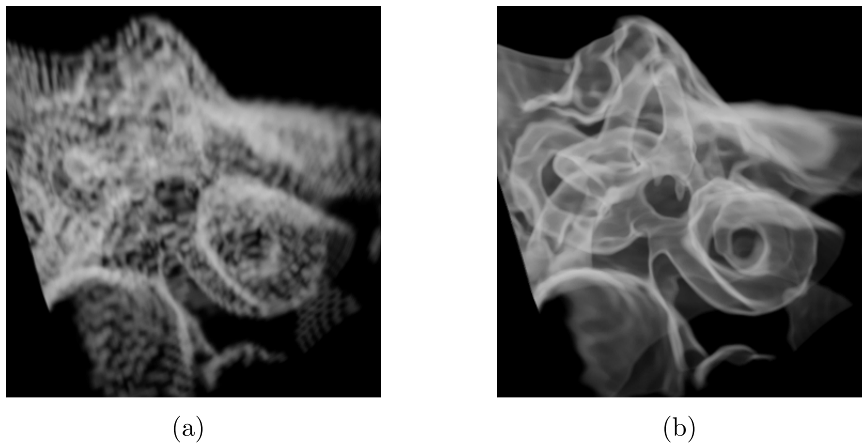
Once a renderer is capable of rendering the data the next problem becomes how to represent the data such that large data can be rendered with little or no penalty in frame rate. One method is to convert the volumetric data into a compressed heirarchical wavelet representation in a preprocessing step. That is then decompressed on-the-fly during rendering using texture maps [12]. Among the most popular approaches is to divide the volumetric data in a preprocessing step into bricks big enough to fit into memory which are passed individually as needed to the graphics system for rendering. Another slightly different approach is to arrange the octree to support multiple resolutions [2, 27, 44].

2.2.7 Transfer Functions

The remaining problem to be discussed is how volume data that spans very different and very specific ranges is converted into the colors needed for the rendering algorithm

to interpolate and composite into a final image. This is done using transfer functions. Transfer functions are usually represented as a 1-dimensional texture used as a lookup table. The value in the volumetric data is then used as the entry point to the lookup table which returns a RGBA color. However a transfer function does much more than convert volumetric data into a RGBA color value. Transfer functions are used to identify patterns and boundaries in the volumetric data. Though transfer functions can be generated automatically [19], it usually requires extensive knowledge of the data in order to properly classify the data set. For more information on classification theory and methods, please refer to [7].

There are two main ways to apply transfer functions, *pre-interpolative* and *post-interpolative*. Pre-interpolative transfer functions apply the transfer function to the discrete sample points of the volumetric data before the data is interpolated. Post-interpolative does the opposite by applying the transfer function after the data is interpolated. Figure 2.10 shows the difference between the two types of interpolations.

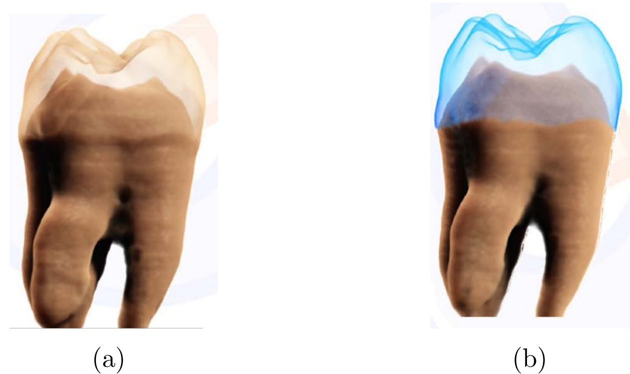


(a) (b)

Figure 2.10: Pre-classification (a). Post-classification (b) [9].

One flaw of 1-dimensional transfer functions arises from boundary discontinuities. Because the data values have to be interpolated, transfer functions attempt to smoothly transition from one data point to the next. However, for sharp boundaries (i.e. large changes between values) this attempt to smoothly transition between overlapping boundary values values results in artifacts as shown in Figure 2.11 (a).

Multi-dimensional transfer functions [20, 37] fix this by adding additional information such as a gradient that measures how much the data changes from one data point to another this artifact can be eliminated as shown in Figure 2.11 (b).



(a) (b)

Figure 2.11: 1D transfer function results in artifacts between the dentin and the surrounding air (a). 2D transfer function accounts for this and allows the overlapping boundary values to be colored differently (b) [9].

2.3 Virtual Reality

Virtual Reality is a field of computer graphics whose focus is on the mental immersion of the user. This immersion can be achieved through the use of a standard desktop and monitor such as can be found in a game, or (more typically) with specialized hardware such at those that will be discussed in the following sections. Visual feedback is the most common form of sensory feedback used by virtual reality to create a sense of immersion; however, other forms of feedback are often used including haptic (touch), olfactory (smell), and aural (sound).

Because virtual reality is a technology that can provide a sophisticated real time 3D interface for users to interact with their 3D applications it makes a good candidate for interacting with 3D atmospheric data. Research efforts of virtual reality in other contexts can be found in [5, 22, 30, 52].

Because of the technological advantages of more complex virtual reality systems, virtual reality as it applies to this thesis focuses on visual feedback using stereo-

scopic displays and employs tracking systems for interaction. What follows is a brief overview of virtual reality with a focus on the those topics relevant to this thesis. More information about virtual reality can be found in [3, 6, 42].

2.3.1 Depth Cues

Depth cues are how people take the 2-dimensional information they see in images and even in how the eye collects light onto the retina and creates the 3-dimensional spatial information that people are used to seeing. There are three main categories of depth cues that human's use to discern spatial information: motion, monoscopic, and stereoscopic depth cues.

Motion depth cues are commonly referred to as *motion parallax*. As one moves through the environment, near objects seem to move faster than objects farther away. However, motion cues are not always present (such as in pictures), so other depth cues must be used.

Monoscopic depth cues are depth cues that only require the information from one eye. Monoscopic depth cues can be dicerned from anything, including pictures. There are many different types of monoscopic depth cues which can be found in [36], a few of which are outlined below:

- Perspective – The appearance of two lines converging in the distance. A classic example is looking down a long straight road.
- Occlusion – When one object is covering or partially blocking another object.
- Texture Gradient – The farther away an object is, the less detail can be seen.

Stereoscopic cues are commonly referred to as *binocular vision* and requires the information gathered from both eyes. These cues use the slight differences in perspective due to the offset of the eyes (known as *binocular disparity*) to discern spatial information. The main problem with stereoscopic cues are that from a distance farther than six meters the binocular disparity is so minimal that the difference between the perspectives is virtually insignificant.

2.3.2 Stereoscopic Display Environments

Stereoscopic displays allow for an added sense of immersion over standard desktop displays in that they allow for stereoscopic cues as discussed in the previous section.

In order to achieve stereoscopic depth cues in a virtual environment, the viewpoint of each eye has to be rendered and displayed. There are two main types of stereoscopic displays: head mounted displays (HMD) and projection displays. Head mounted displays like those shown in Figure 2.12 are devices that contain small, lightweight displays (one for each eye) that are designed to fit on a persons head and block out the outside world. Head mounted displays suffer from a couple immersion problems such as the inability to see oneself in the simulation. Projection displays like those in Figure 2.13 solve this problem but are much more expensive.

There are two main ways to achieve stereo on projection based displays. The first is through a passive system that utilizes special glasses with polarized filters. On passive systems, each screen is rendered using two projectors (one for each eye) each of which has a special polarized filter corresponding to the filters being used in the glasses. The second method of achieving stereo is by using an active system. Active stereo systems like the one in Figure 2.13 use a high-speed projector and special glasses that shutter open and closed each eye in quick succession synced with the rendering of the projection.

2.3.3 Input and Immersion

Different input devices allow for varying levels of immersion. Joysticks and gamepads are a common input device used with computers. However, because they wre not a common type of input defice found in real-life, they take away from the overall sense of immersion

To add a level of immersion to the standard joystick, their position and orientation is often tracked. This allows the user to utilize the joystick–commonly called a wand–much like they would their hand. They can move it around and manipulate the environment much in the same way they would in the real world. An example of

Figure 2.12: Head-mounted display [42]

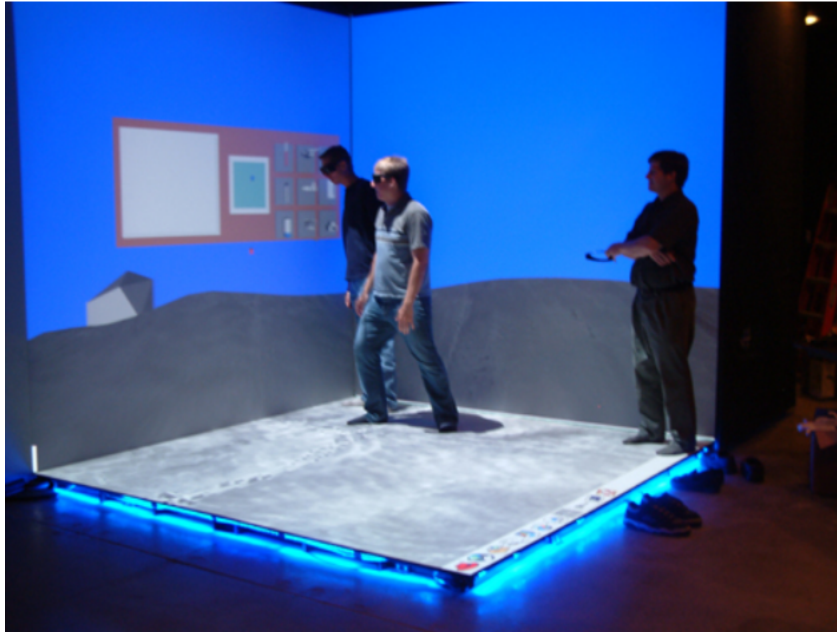


Figure 2.13: Multi screen projection system

a standard wand used in a virtual environment can be seen in Figure 2.14.

To provide motion cues (motion parallax) some type of system is required to track head position and orientation. Orientation is essential for HMD systems as it determines what's being rendered. Projection based systems depend much less on orientation since the environment is being constantly rendered for each viewpoint. Position is required to maintain a certain level of immersion as position allows an application to render from the users point of view, allowing them to inspect objects by moving their head much like they would in real life. An example of the type of head tracking used in a virtual environment can be seen in Figure 2.15.

2.3.4 Toolkits

Because virtual reality systems contain specialized input and tracking hardware as well as specialized screen hardware, writing applications for a virtual environment can be a difficult task. In addition, hardware configurations can change between systems which requires software changes for each of these different systems. Virtual

Figure 2.14: Intersense tracked input system [17].

Figure 2.15: Intersense head tracking system on shutter glasses [17].

reality toolkits attempt to provide a layer of abstraction to the hardware so that applications only need to be written once and can be used on these varying systems. The biggest difference between the different toolkits is in the types of input hardware and computational configurations that they support. Oliver Kreylos' VRUI [24] supports a variety of input systems including the mouse and high-end tracking systems such as the Intersense IS-900™ tracking system as well as allows the user to add input systems. In addition, VRUI supports both shared-memory and cluster-based computational systems. Other virtual reality toolkits include VRJuggler [15] and FreeVR [41].

2.4 Existing Applications

Currently, very few applications exist designed for atmospheric visualization. The most popular being Vis5D [18] and its virtual environment port Cave5D [49]. The problems with Vis5D/Cave5D are that they are designed soley to display atmospheric data. They are not designed to display anything other than atmospheric data and thus unable to be used as part of a larger visualization application. In addition, while they can display atmospheric data including data containing temporal data, they are incapable of displaying more than one dataset from the model at a time. Figure 2.16 shows an example of a dataset being rendered in Vis5D with Figure 2.17 showing Cave5D.

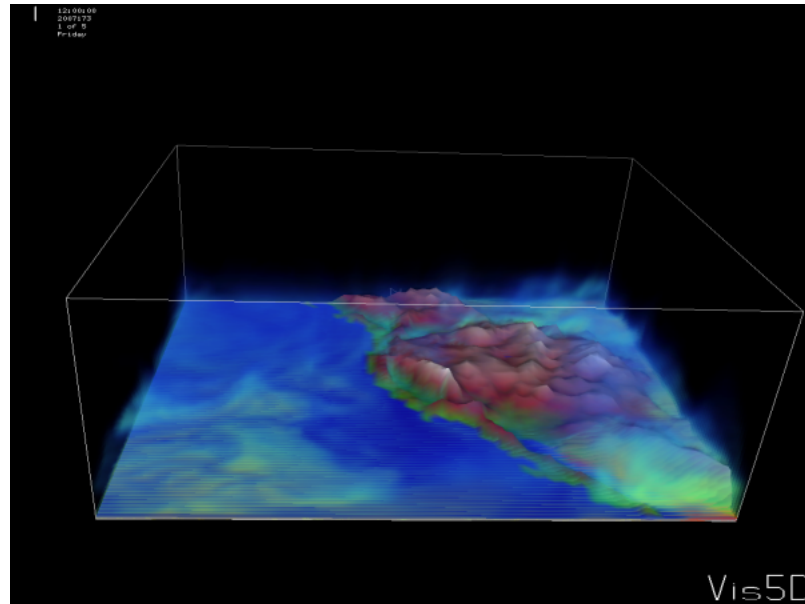


Figure 2.16: Atmospheric dataset shown in Vis5D.

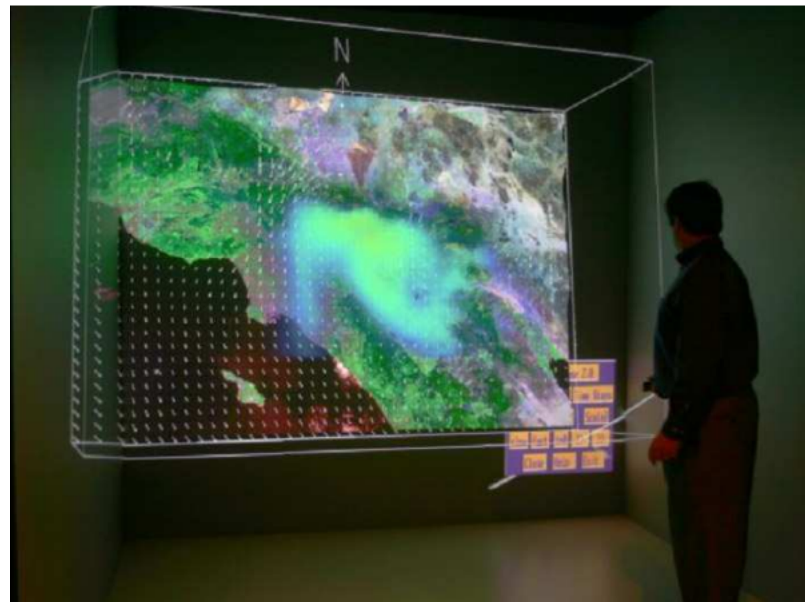


Figure 2.17: Atmospheric dataset shown in Cave5D.

Chapter 3

Volume Library Software Design

Chapter 2 outlined the research that has been done in the various aspects of atmospheric rendering. In addition, it introduced the leading atmospheric rendering application and explained the weaknesses and disadvantages of the application.

This chapter will formally introduce the Vesuvius volume rendering library by presenting its requirements, use cases, and overall structure. The requirements outline the basic features of the library as well as the technology needed to support the library. The use cases describe the basic functionality of the library. More detailed descriptions of the library along with the overall structure are presented later in the chapter. Please refer to [1, 8, 43] for more information on the software engineering practices and principles used to develop Vesuvius.

3.1 Requirements

The goal of Vesuvius is to provide the ability to visualize a large amount of volumetric data (with an emphasis on atmospheric data) across a large variety of applications and environments. It must be modular to give it the capability to plug into other applications where it would used as part of a larger application. In addition, it must be able to render temporal data on both desktops and virtual environments. Lastly, it should allow for the rendering of multiple overlapping volumetric datasets at once.

3.1.1 Functional Requirements

Table 3.1 outlines the basic functionality and features that are the basis for Vesuvius. These features were created based on an analysis of scientific requirements as well as by what was lacking in other volumetric rendering systems.

F01	Vesuvius shall give the ability to display any type of volumetric data
F02	Vesuvius shall display atmospheric data via the MM5/v5d file formats
F03	Vesuvius shall allow for temporal datasets
F04	Vesuvius shall allow the user to start a simulation
F05	Vesuvius shall allow the user to stop a simulation
F06	Vesuvius shall allow the user to render overlapping volumetric datasets
F07	Vesuvius shall allow the user to toggle which volumetric datasets to render
F08	Vesuvius shall allow the user to specify transfer functions for each dataset
F09	Vesuvius shall allow the user to adjust transfer functions on-the-fly
F10	Vesuvius shall allow the user to load and save transfer functions
F11	Vesuvius shall allow the user to enter the volume and look around
F12	Vesuvius shall allow the user to control rendering detail vs. frame-rate

Table 3.1: Functional requirements for Vesuvius.

3.1.2 Non-Functional Requirements

Non-functional requirements are outlined in Table 3.2 and show the supporting technologies and software supported by Vesuvius. This primarily covers the interface and the features that are presented to the developer.

N01	Vesuvius shall be written in C++
N02	Vesuvius shall render in OpenGL
N03	Vesuvius shall be cross-platform compatible
N04	Vesuvius shall be composed of many modular components
N05	Vesuvius shall be independent of any windowing/VR toolkit
N06	Vesuvius shall implement raycasting as its rendering algorithm
N07	Vesuvius shall use VRUI for its prototype application
N08	Vesuvius shall take advantage of shaders
N09	Vesuvius shall render at interactive frame-rates

Table 3.2: Non-Functional requirements for Vesuvius.

3.2 Use Cases

Figure 3.1 shows the functionality provided by the end user. These are developed using the functional requirements outlined in the previous section. Only major features visible to the user are represented as use cases in this figure. There are two actors that influence Vesuvius. The user provides input and controls the simulation. Time updates the simulation with the relevant data as directed by the user.

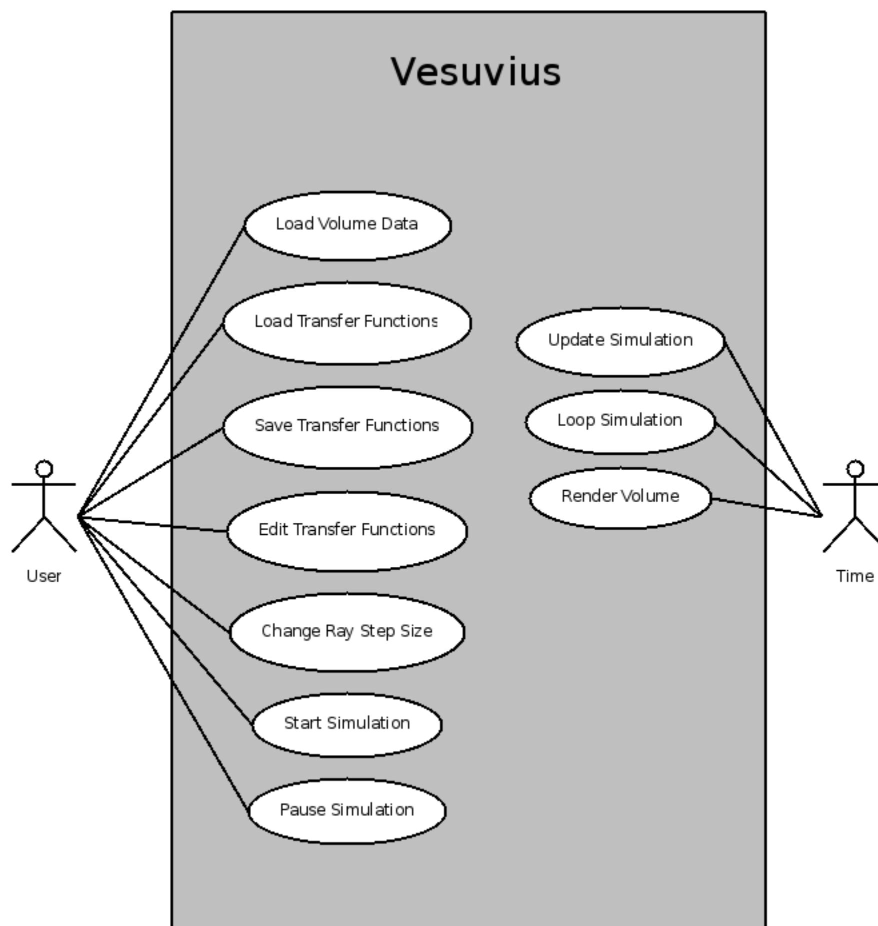


Figure 3.1: Vesuvius use case diagram.

3.3 Architecture

Figure 3.2 shows the class heirarchy of the Vesuvius volume rendering library. It is split into four parts: the preprocessor, the renderer, the transfer function manager, and the file loader. The goal of the design was for it to be modular and flexible, allowing the user to replace any subsystem with their own custom system.

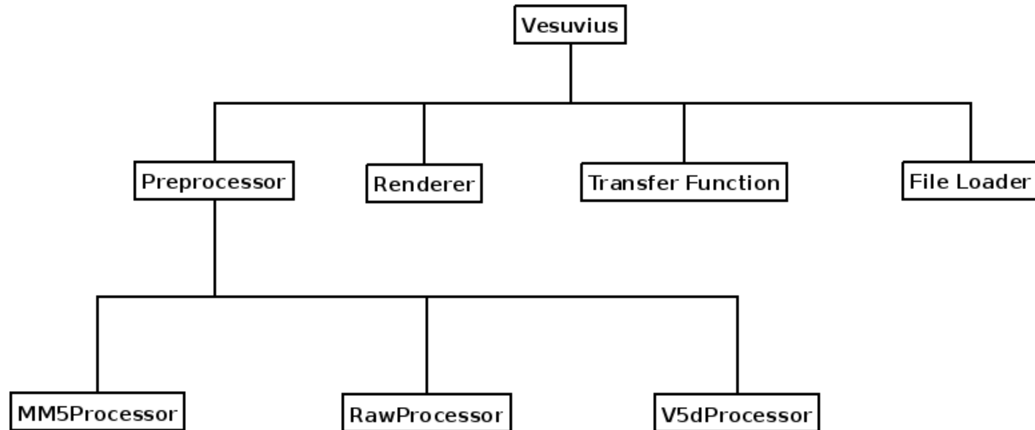


Figure 3.2: Class structure of the Vesuvius volume rendering library.

3.3.1 Preprocessor Subsystem

The preprocessor subsystem is responsible for converting various other atmospheric data sets such as RAW, V5D, and MM5 data, into a custom file format that Vesuvius is able to read. Because of the many types of data that could potentially be rendered a single file format simplifies the rendering process. In addition, the preprocessor is designed to brick out the data into manageable chunks to allow the renderer to handle larger data sets as well as correctly blending multiple overlapping data sets. By doing this in a preprocessing step it minimizes the amount of computation that is required during run-time, increasing the overall frame-rate.

3.3.2 Renderer Subsystem

The renderer subsystem is responsible for taking the data given to it and rendering it onto the screen. It uses a GPU-based raycasting algorithm that takes a data segment (in a 3-dimensional texture) and transfer function (in a 1-dimensional texture) and determines how to render it. It also determines what to render in cases where there is existing geometry occluding the volume. The renderer subsystem is designed to be very basic in that it simply renders what's given to it, doing little in the way of interpreting or validating the data or transfer functions provided.

3.3.3 Transfer Function Subsystem

The transfer function subsystem is responsible for keeping track of any and all transfer functions that a user might define for a data set or multiple data sets as well as loading and saving them. The subsystem allows transfer functions to be defined via code or through a graphical front-end to help facilitate the creation of transfer functions.

3.3.4 File Subsystem

The file subsystem is responsible for loading in volumetric data sets and in the case of data sets containing temporal data, performing memory management in that it has to cache out old time step and load in newly requested ones. The file loading subsystem also is responsible for determining what data sets to render as well as determining the correct rendering order for the front-to-back compositing that takes place inside the rendering subsystem.

Chapter 4

Volume Library Implementation

Pompeii is the name of the test application that demonstrates the capabilities of the Vesuvius volume rendering library. Pompeii is written using the VRUI toolkit to allow Vesuvius to be rendered in a virtual environment. Through VRUI, Pompeii adds graphical transfer function editing capabilities to make editing transfer functions easier. Furthermore, because of VRUI's built-in menuing system, Pompeii adds menu items to encapsulate the functionality provided by Vesuvius.

Each subsystem was designed to work independently from each other. This allows for greater modularity. Therefore, if the user wants to use their own transfer function manager or switch to a different rendering algorithm they may do so without having to rewrite everything. What follows is an explanation of each individual subsystem along with implementation modifications and notes on use.

4.1 Preprocessor Implementation

It is the job of the preprocessor to convert the volumetric data from its native format into one which Vesuvius can read. Currently supported formats include RAW data, Vis5D files, and MM5 files (through Vis5D conversion). Because of how much difference there can be in what is stored in the various file formats, the file format supported by Vesuvius has to be both robust yet simple. Figure 4.1 details the file format that Vesuvius supports with Table 4.1 detailing all the information that Vesuvius keeps.

First the length, width, and height of the overall volume are stored so the user

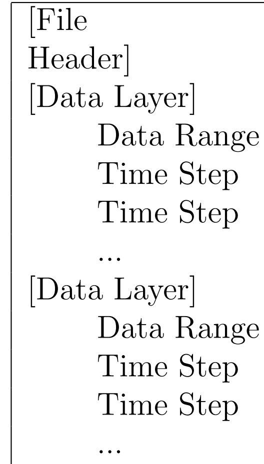


Figure 4.1: Vesuvius file format.

Name	Size	Description
dimensions	int[3]	Dimensions of the volume (x, y, z)
brickSize	int	Size of each brick (x, x, x)
numBricks	int[3]	Brick dimensions of volume (bX, bY, bZ)
geospatial	float[2]	Geospatial coordinates
numTimeSteps	int	Number of time steps in each data layer
numLayers	int	Number of data layers in file
layerNames	string[numLayers]	Names of the data layers

Table 4.1: File header information.

knows how large the overall volume is going to be. Secondly, the size of the bricks are given (including the overlap voxel as mentioned in Section 2.2.7). Then geospatial coordinates are stored (zero if the original data file does not have geospatial coordinates). Then the number of different layers in the data file followed by their name. Following that are the number of time steps each individual layer has. The actual bricked out data is last. Because the size of the layers and time steps are known, it is easy to calculate where in the file any particular layer/time step is located, making retrieval of information quick.

Preprocessing MM5 files is a little more complex than simply converting the MM5 files to Vesuvius files. Rather than MM5 volume information being contained inside a structured grid as shown in Figure 4.2 (a), the initial starting position of the data

follows the contours of the terrain as in Figure 4.2 (b). Although this saves space by not including no data markers for points below the terrain, visualization doesn't like non-structured grids. For this reason the MM5 file is converted first into a Vis5D file. This is done instead creating custom conversion routines because while the Vis5D conversion utility is lacking (data near the terrain is jagged), it is the widely used and commonly accepted way to convert MM5 data into a structured grid. From there the file can be converted from a Vis5D file into a Vesuvius file.

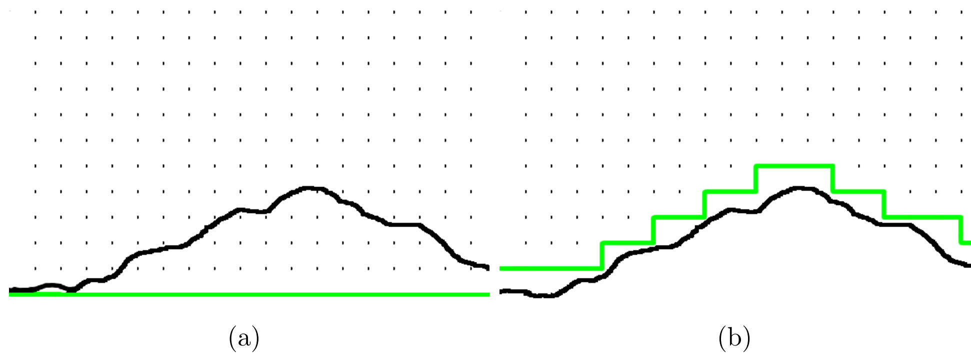


Figure 4.2: Normal structured grid with data starting at points denoted by green line (a). MM5 file where data starts at points following the contours of the terrain (again denoted by green line (b)

Very little conversion is necessary to convert a Vis5D file into a Vesuvius file. The primary reason that any conversion is necessary comes from the difference in algorithms used. Vis5D uses 2D texture-based volume rendering whereas Vesuvius uses GPU-based raycasting. So while Vis5D through 2D textures is able to render a structured grid, Vesuvius requires a regular structured grid. With Vis5D files, more detail is given to the data near the terrain. This is done by having the data grouped closer together near the terrain and increasing the distance between the data as it gets farther from the ground as shown in Figure 4.3. While texture-based volume rendering via 2-dimensional texture slices allows for this, GPU-based raycasting–and any other volume rendering using 3D textures–requires regular structured grids. So what the preprocessor does is determine a reasonable constant distance between data layers (often adding layers in the process) and interpolating between layers.

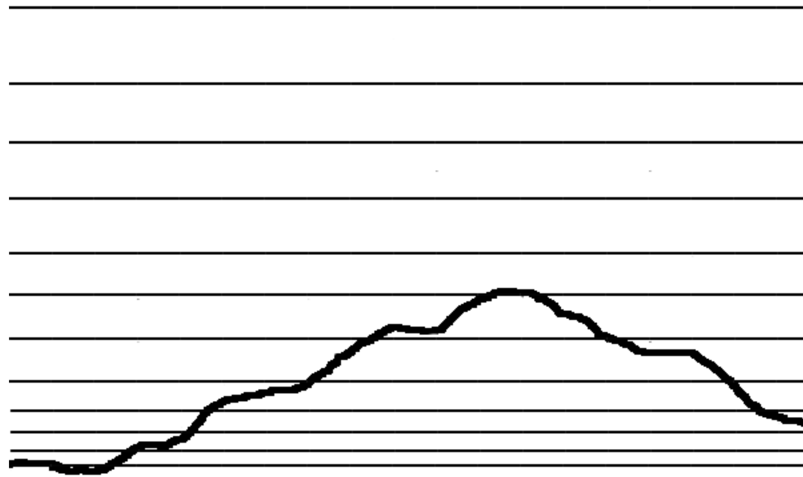


Figure 4.3: Data distribution of Vis5D files. Notice how the data is more dense the closer it gets to ground level.

4.2 Volume Renderer Implementation

The algorithm chosen for Vesuvius was a GPU-based raycasting algorithm. This was done because of the scalability of the algorithm with modern graphics cards. The ability of a graphics card to compute multiple pixels at a time combined with raycasting's pixel-by-pixel approach means that as graphics cards increase in power and pixel pipelines, the performance of the application will increase. In addition, raycasting is a more robust algorithm, capable of scaling to multiple GPUs with a minimum of modifications.

The algorithm departs from that presented in [25] in that a color cube is not rendered to determine the ray direction. Instead, the shader capitalizes on knowledge about the OpenGL ModelView Matrix and uses the inverse of it to determine the camera position. From there it takes the difference between the camera position and the position of the pixel being rendered to determine the ray direction. By moving the ray direction calculation to the shader, a framebuffer for the front-face culled color cube is not necessary as actual location calculations are being computed instead of computing direction from the difference of two color cubes.

The implementation of the algorithm includes early ray termination. So when the ray exits the volume, or the opacity reaches a certain point, the raycasting for that pixel will end. Additions had to be made to the original algorithm to allow standard OpenGL objects with the volume library as well as to be able to fly inside the volume and are presented in [38]. For interaction with solid OpenGL objects, the raycaster needs knowledge of what opaque objects are in the scene so when the ray hits a solid object it knows to terminate early. This is done by obtaining the depth buffer before each rendering pass and sending it as a texture to the fragment shader. From there world coordinates are extracted from the depth buffer to determine where the solid object is in relation to the volume, which determines how far the ray should go (if it should start at all). The second addition is if the camera is inside the volume. In this case a polygon has to be rendered directly in front of the camera so the ray has a starting position. This is done computationally before each render so the shader does not have to calculate a starting position for each pixel.

The raycasting is split into two stages. In the first stage, the volume data given to the raycaster is rendered into a framebuffer object. In the second stage, the framebuffer object is blended in with the current framebuffer to create the final image. This two-stage process is done to support bricking. Each brick can be rendered with the correct blending equations into the framebuffer object which can then be blending using different blending equations into the final images as in Figure 4.4.

Choosing the right blending equations proved to be crucial to correctly render the volume. Two blending functions are needed. One to blend the volumes correctly together and a second to blend the final volumetric image with the existing image in the framebuffer. Because of how crucial blending functions are to the final image, some restrictions are imposed with regards to rendering order. First, the volumetric data must be rendered in front-to-back order to correctly blend the varying volumetric data. Second, the volumetric data must be the last thing rendered. This is done because the final image is rendered as a 2-dimensional polygon blended with the framebuffer. If it is not the last thing rendered then parts of the volume will be

incorrectly covered by any additional OpenGL objects rendered later on. In addition, objects rendered later are not in the depth buffer for the raycaster to use to factor early ray termination.

One issue is that as the step size changes all of the transfer functions need to be changed to reflect the change in step size. This is because changing the step size changes the number of samples that are taken of the volume and thus affect how much compositing is done within the volume. So by decreasing the step size (more samples) the values in the transfer function have to be decreased and vice-versa. Figure 4.5 shows what happens when the step size is changed without modifying the transfer function. Because of a higher sampling rate, the image reaches a high opacity much faster than with a lower step size, resulting in a darker image.

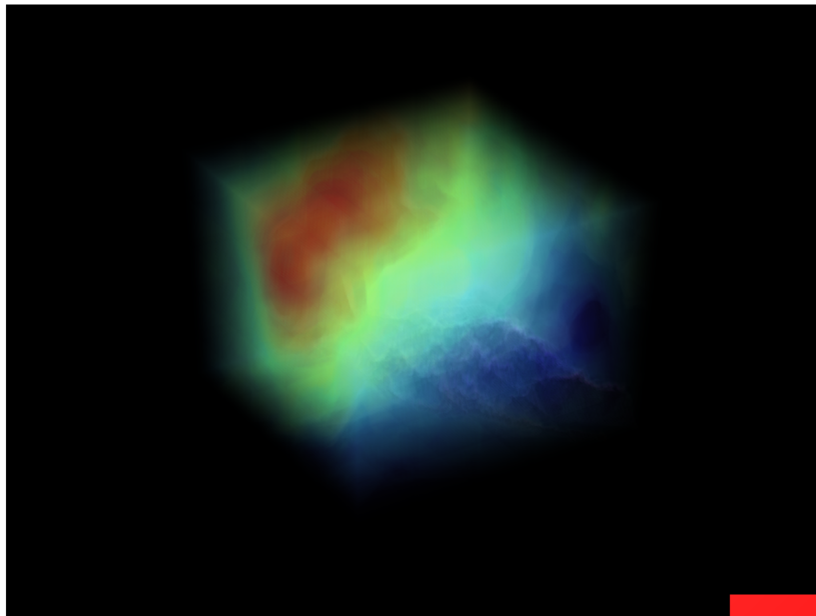
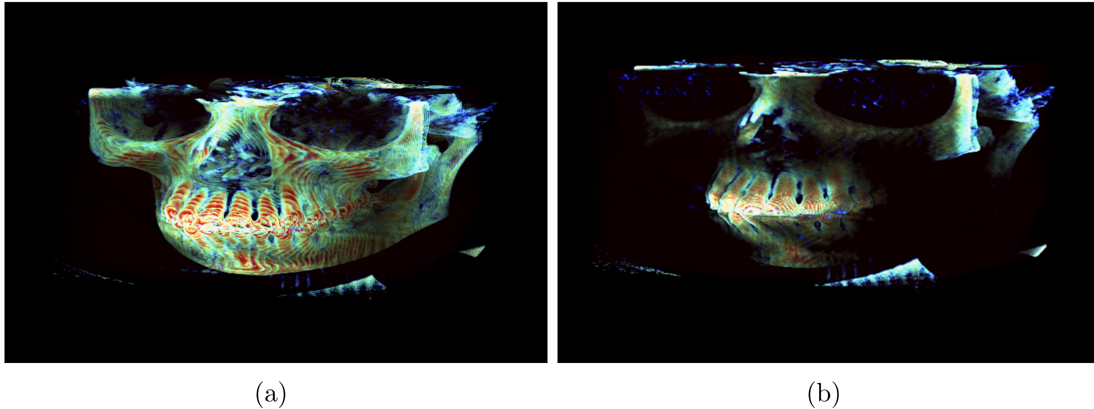


Figure 4.4: MM5 file rendered through Vesuvius.

4.3 Transfer Function Manager Implementation

The transfer function manager is designed to give the user the ability to create, load, and save transfer functions. A transfer function is implemented as a structure



(a) (b)

Figure 4.5: Raycasting with default step size (a). Raycasting with increased step size (b) notice how this change requires the transfer function to be edited.

containing–among other information–red, green, blue, and alpha control channels. These points range from a fixed domain (usually spreading the domain of the actual volumetric data set) to an arbitrary range (which can be different for each color channel). These control channels are then combined to form a 1-dimensional transfer function which can then be passed to the renderer along with the volumetric data to be rendered. The transfer function manager has the capability of storing an arbitrary number of transfer functions, each one with its own unique name as well as modifying specific transfer functions, save/load individual transfer functions from file, or save/load an entire list of transfer functions.

The transfer function manager does not require the individual control channels to have control points at the same domain. For example, if the red control channel has a control point at (10, 0.5), the blue control channel does not have to have a control point with it's domain being 10. This allows for a greater degree of flexibility when designing transfer functions in that all the channels are completely independent of each other. To account for this, linear interpolation is used to match up the arbitrarily defined control channels when the texture is being created. An example of how to create a simple transfer function and store it is found in Figure 4.6.

```
transferFunction.fBounds[0] = 0.0f;
transferFunction.fBounds[1] = 255.0f;

transferFunction.redChannel.push_back( pair<float, float>( 0.0f , 0.500f ) );
transferFunction.redChannel.push_back( pair<float, float>( 255.0f, 0.090f ) );

transferFunction.greenChannel.push_back( pair<float, float>( 0.0f , 0.510f ) );
transferFunction.greenChannel.push_back( pair<float, float>( 85.0f , 0.470f ) );
transferFunction.greenChannel.push_back( pair<float, float>( 170.0f, 0.300f ) );
transferFunction.greenChannel.push_back( pair<float, float>( 255.0f, 0.090f ) );

transferFunction.blueChannel.push_back( pair<float, float>( 0.0f , 0.500f ) );
transferFunction.blueChannel.push_back( pair<float, float>( 180.0f, 0.230f ) );
transferFunction.blueChannel.push_back( pair<float, float>( 255.0f, 0.080f ) );

transferFunction.alphaChannel.push_back( pair<float, float>( 0.0f , 0.000f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 115.0f, 0.000f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 150.0f, 0.050f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 175.0f, 0.065f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 195.0f, 0.300f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 225.0f, 0.450f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 255.0f, 1.000f ) );

transferFunctionManager.AddTransferFunction( "name", transferFunction );
transferFunctionManager.SetActiveTransferFunction( "name" );
```

Figure 4.6: Creating a transfer function and storing it in the transfer function manager.

4.4 File Loader Implementation

The file loader is responsible for loading in the parts of the volumetric file that are needed at any given time as well as offering the user general information about the file. It provides general information such as the number of layers in the data set, the bounds of the data set, the number of time steps in the data set, as well as geospatial coordinates for use if the user wishes to geospatially align the volume.

The user specifies what time step of what data layers they wish to load and the loader will read them in. When it's time to render the individual bricks, it is the

loader's responsibility to determine rendering order based on camera position. It does this by computing the global position of each brick and treats it as a bounding box. From there it goes through and finds the distance between the camera and the center of each brick and sorts those in order from closest to farthest. What is returned is the ordering that is required to correctly render the volume. By grabbing the correct bricks of the correct data layers in the correct order and passing them from the file loader to the renderer correct visual results are assured.

4.5 Interface

Through the VRUI toolkit Pompeii offers a menu front-end to the Vesuvius volume rendering library. It offers controls to adjust various settings that the library exposes. Figure 4.7 shows the root menu containing all the options available to the user. There are other menues to change the step size of the raycaster as well as to enable and disable the different layers in an MM5 file like shown in Figure 4.8.

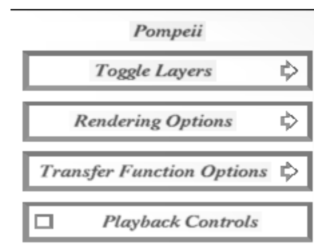


Figure 4.7: Root menu of Pompeii.

The transfer function menu shown in Figure 4.9 allows the user to load, save, and edit transfer functions. It is the menu that allows the user to adjust the look of each volumetric layer. The transfer function editing menu shown in Figure 4.10 contains options to edit each of the four color channels as well as to add and remove control points. A color map is shown to give real-time feedback into how the changes to the transfer function will effect the volume. In addition to the color map, the volume itself will adjust in real-time reflecting the change in transfer function.

Pompeii also includes a playback menu like the one in Figure 4.11. The menu

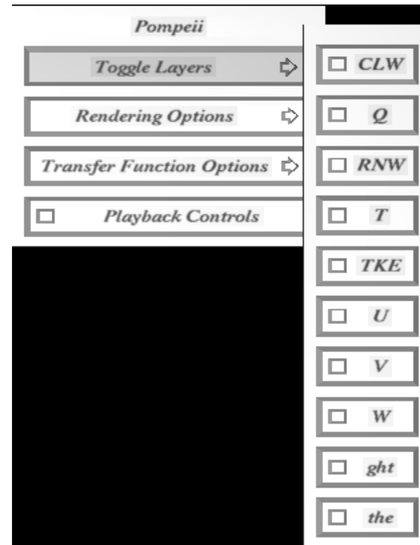


Figure 4.8: Menu that enables and disables the different volumetric layers a file may have.

allows the user to move between frames on any volumetric data set that contains temporal information. In addition, it provides the user with the ability to play and pause the simulation.

4.6 Visualization System

Pompeii runs on our immersive visualization hardware that is composed of a four-screen CAVE™-like Fakespace FLEX™ [14] system running quad AMD Opterons™ and an NVIDIA Quadro® FX 4500 X2. The system is capable of rendering active stereo on all four screens and is tracked using an Intersense IS-900™ head tracking unit and wand unit [16]. However, based on it's structure Vesuvius is able to run on any visualization system from four-screen CAVE-like systems to normal desktop displays.

4.7 Sample Case

The following section illustrates a sample case of atmospheric rendering through Pompeii using Vesuvius. The MM5 file contains atmospheric data covering the western

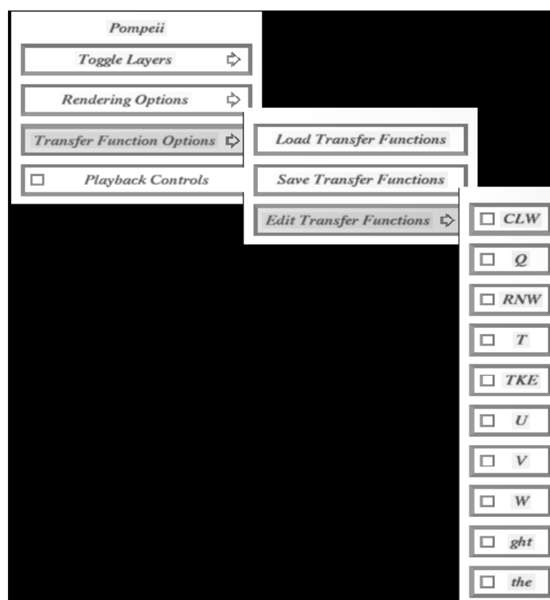


Figure 4.9: Transfer function menu.

United States in an 867Mb file. The file is first processed using a freely available Fortran program tovis5d which converts the volumetric data in the MM5 file into a Vis5D file with a size of 13Mb. From there it is run through the Vesuvius preprocessor to tweak the data further and brick it. The size of the bricks is specified as a command-line argument and the exact size is data dependent. Figure 4.12 shows the steps that the user goes through to convert the MM5 file into a Vesuvius compatible file.

Once the volume is converting into a Vesuvius file, it can be run in Pompeii. When Pompeii is run, all data layers the file may contain are disabled by default, resulting in an empty screen. The user can then toggle layers on and off by right-clicking, hovering over the "Toggle Layers" menu and clicking each layer that they wish to show. The button will indicate whether a layer is enabled or disabled as shown in Figure 4.13.

The user can also edit the look of a volumetric layer by right-clicking, and going to "Transfer Function Options→Edit Transfer Functions" and choosing the layer you wish to edit. When a transfer function is being edited, if the layer is enabled the

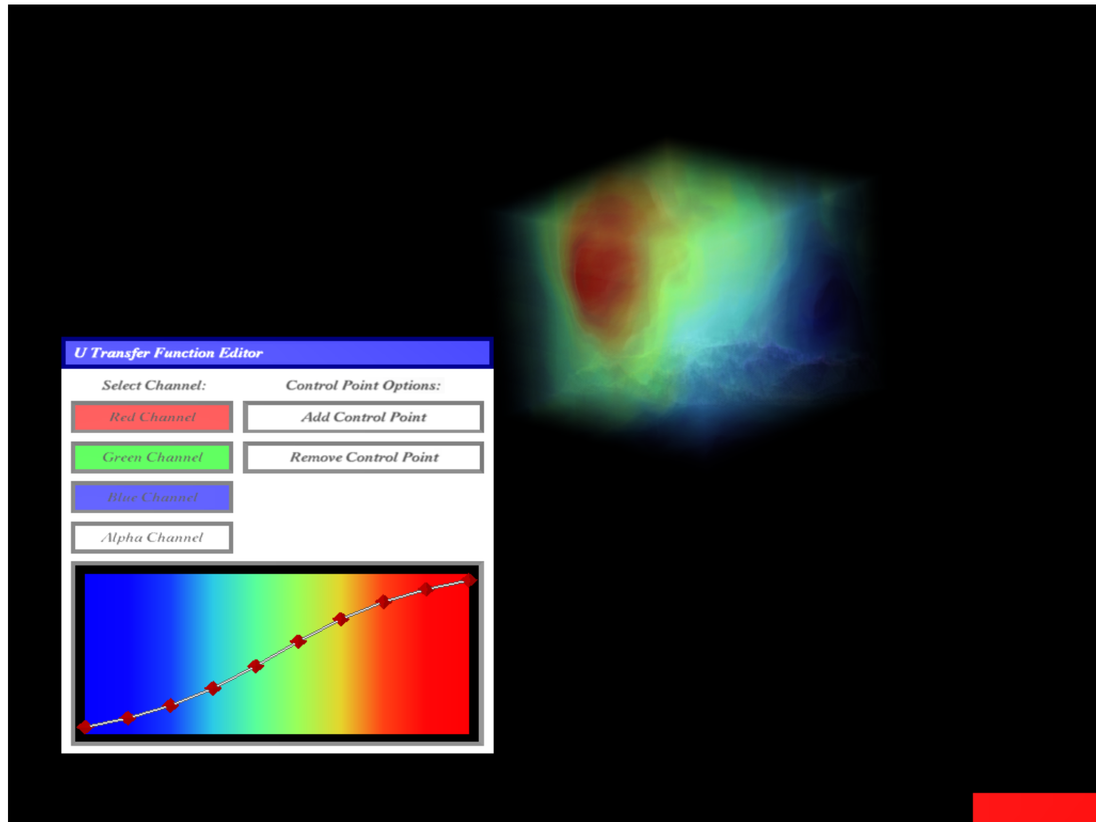


Figure 4.10: Transfer function editor menu.

volume will immediately reflect the changes done to the transfer function. Figure 4.14 shows a volumetric layer with default coloring and compares it to an image with adjusted coloring.

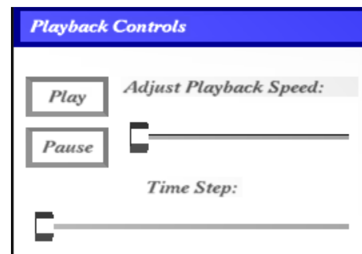


Figure 4.11: Playback menu.

```
~:./tovis5d MMOUT_DOMAIN1
~:./preprocessor -b 20 mm5.v5d
```

Figure 4.12: Process of converting an MM5 file to a Vesuvius file.

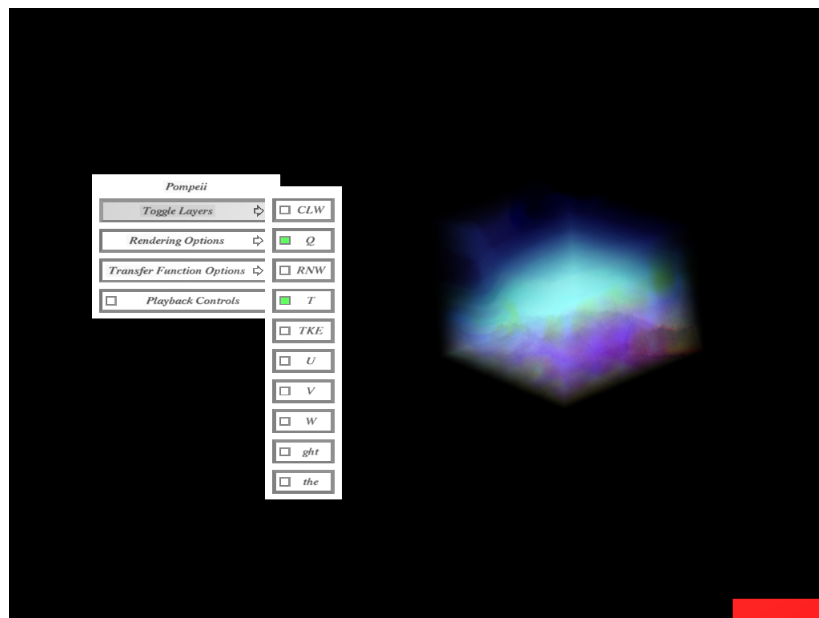


Figure 4.13: Vesuvius with the "Q" and "T" layers enabled.

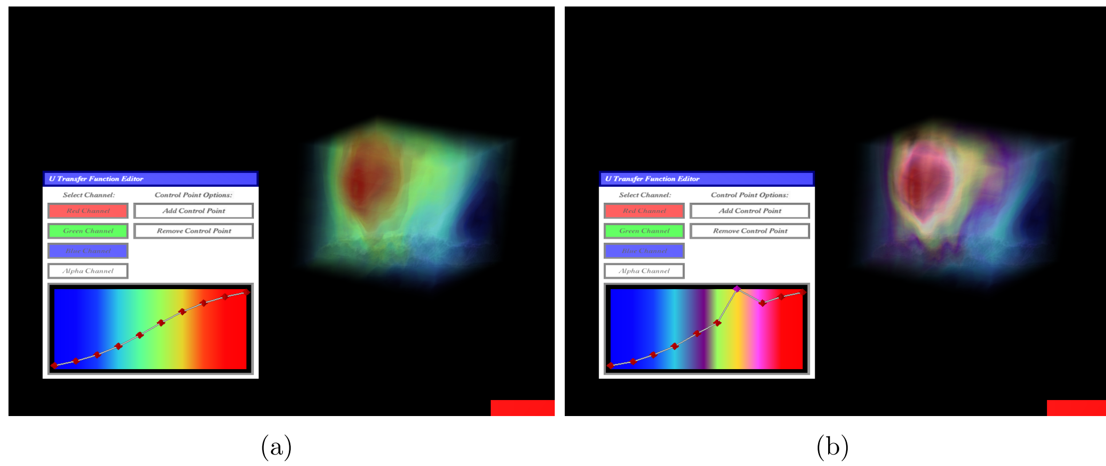


Figure 4.14: “U” layer with default coloring (a). “U” layer with adjusted coloring (b).

Chapter 5

Conclusions and Future Work

5.1 Conclusion

Though there are many ways to render a volume [4, 26, 28, 29, 48, 50], little work has been done on volumetric rendering in a virtual environment [23, 39, 49] and even less on rendering atmospheric data [18, 49]. This thesis introduced a modular library for rendering atmospheric data in a virtual environment. Because of its modular approach, it allows for the addition or modification of the varying subsystems without affecting the overall library.

Vesuvius encompasses the previous work done in the areas of atmospheric visualization, virtual reality, and volume rendering and combines them in a new way to arrive at a robust volume rendering library capable of rendering data ranging from static volumes such as engine blocks to temporal playback of atmospheric data in an existing OpenGL application. It is modular and adaptable enough to be used on a wide variety of hardware environments ranging from a standard desktop application to a virtual environment.

Initial tests of the library have been promising. Vesuvius has shown that it can render atmospheric data from MM5 files at interactive frame-rates on a virtual environment utilizing the VRUI toolkit. Vesuvius promises to be a crucial tool for atmospheric scientists in identifying and observing trends in atmospheric data, particularly over time.

5.2 Future Work

Vesuvius encompasses many difficult problems from a wide range of disciplines in computer science and the atmospheric sciences. While the initial library has been successful in displaying atmospheric data, much work could be done to make the visualizations more accurate and render faster.

Preprocessor

The preprocessor is the subsystem which could use the most work. It implements a simple bricking algorithm to split up larger data sets into sizes the GPU can render. However since MM5 files can have nested data containing different resolutions, an alternative method would be required. Creating a more complex multiresolution octree such as that found in [2] would not only support multiple resolutions but would create a level-of-detail effect which would make the overall renderer more robust and efficient.

MM5 files have to go through two conversions for Vesuvius to be able to render them. First they must be converted into Vis5D files, and then into Vesuvius' proprietary format. Two levels of interpolation result in data that is visually acceptable but hardly accurate. Eliminating all possible interpolation and converting what is necessary straight from the MM5 file as opposed to converting from an intermediary Vis5D file would make for a visualization that is much more scientifically accurate.

Renderer

Most of the work done in Vesuvius comes from the renderer as it creates the volumetric image. Because of how computationally expensive raycasting is by nature anything that can be done to optimize it would result in a noticeable speed increase. The renderer employs early ray termination to speed up the rendering process, however it would see added benefit from employing empty space skipping. By incorporating certain markers in the bricks–or octree–it is possible to determine what bricks or nodes are empty, and thus ignore those bricks in the rendering process. This would

dramatically increase the performance where it is given porous volumetric data.

The renderer also does a poor job of representing the volume realistically. This is because it does not take light and shadows into account when rendering the volume. A type of local illumination using the Blinn-Phong shading model and a 3-dimensional gradient field could be used to add illumination. However, it does not account for volumes where the material properties are such that light scatters nor for objects that are translucent [10].

Phase functions calculate the distribution of light after scattering. By combining phase functions with a diffuse lighting model such as Blinn-Phong or Lambertian shading it is possible to realistically shade a volume. Furthermore, by incorporating a second raycasting pass that goes back-to-front as opposed to front-to-back, it might be possible to calculate how much light is obscured by a volume, allowing for the shading of other OpenGL geometry.

Transfer Function Manager

While the current implementation of transfer functions is elegantly simple and performs well, 1-dimensional transfer functions can lead to artifacts as discussed in Section 2.2.7. Implementing a multi-dimensional transfer function system such as that in [20] would eliminate these artifacts however they would also result in a more complex user interface. To help ease the burden on the user to create a transfer function for each data layer, automatically generating a base transfer function such as proposed in [19] would go a long way in aiding the user. While the resulting transfer function would not be perfect, it would serve as a good starting point for the user to then edit.

File Loader

A lot of work could be done to optimize the file loader. A smarter caching algorithm could be used to pre-cache volumetric data that is going to be needed in order to minimize lag as the data is accessed from the hard drive. By keeping track of

more information during the simulation process it can also do a more efficient job of determining rendering order, which is one of the bigger bottlenecks CPU-wise for Vesuvius.

Another way to speed up the rendering process is to give OpenGL the volumetric texture through pixel buffer objects (PBOs). This allows for asynchronous data upload that does not block either the CPU nor the GPU. In addition, data upload when dealing with non power of two textures is considerably faster. However, the data must be in a GPU-native format before being sent to the graphics card, which would be done in the preprocessor.

Other Enhancements

There are various other enhancements that would prove useful to atmospheric scientists. The ability to probe the data and return the raw data value would help atmospheric scientists determine what precisely they're looking at. The ability to automatically create iso-surfaces for a particular range of data through automatically generated transfer functions would also aid in isolating specific particulates.

In addition, a detailed user manual complete with documentation and sample programs would aid end-users in understanding and using Vesuvius.

Appendix A

User Manual

This appendix shows a brief overview of how to initialize and run the Vesuvius volume rendering system.

A.1 Preprocessor

To preprocess a file is a simple matter of finding an appropriate brick size for the particular data set, converting it to vis5d format (if MM5), and running the preprocessor with the appropriate brick size as shown in Figure A.1.

```
~:./tovis5d MMOUT_DOMAIN1
~:./preprocessor -b 20 mm5.v5d
```

Figure A.1: Steps taken to convert an MM5 file into a Vesuvius compatible file.

A.2 Renderer

To set up the rest of vesuvius requires the OpenGL Extension Wrangler Library [13]. To initialize the renderer the user needs to know two things: the size of the screen (or size of the area they wish to render to) and the path to the shaders that the user wishes to use. The process of initializing the renderer is shwon in Figure A.2.

```
CRaycaster *renderer;
renderer = new CRaycaster( xScreenSize, yScreenSize );

// METHOD 1
renderer->Initialize();
renderer->LoadShaders( "vertexShader.glsl", "fragmentShader.glsl" );

// METHOD 2
renderer->Initialize( "vertexShader.glsl", "fragmentShader.glsl" );
```

Figure A.2: Initializing the renderer.

A.3 Transfer Function Manager

Next, at least one transfer function needs to be specified so the renderer knows how to color the data. Figure A.3 shows the process of creating a transfer function through code. Once a transfer function is created, it needs to be added to the transfer function manager.

A.4 File Loader

Lastly, a file needs to be loaded, which is demonstrated in Figure A.4

A.5 Rendering

Once the different subsystems of Vesuvius are initialized, they need to be rendered. Figure A.5 shows how to retrieve rendering information from the file loader and render it correctly using the initialized renderer.

```
STransferFunction transferFunction;
CTransferFunctionManager transferFunctionManager;

transferFunction.fBounds[0] = 0.0f;
transferFunction.fBounds[1] = 255.0f;

transferFunction.redChannel.push_back( pair<float, float>( 0.0f , 0.500f ) );
transferFunction.redChannel.push_back( pair<float, float>( 255.0f, 0.090f ) );

transferFunction.greenChannel.push_back( pair<float, float>( 0.0f , 0.510f ) );
transferFunction.greenChannel.push_back( pair<float, float>( 85.0f , 0.470f ) );
transferFunction.greenChannel.push_back( pair<float, float>( 170.0f, 0.300f ) );
transferFunction.greenChannel.push_back( pair<float, float>( 255.0f, 0.090f ) );

transferFunction.blueChannel.push_back( pair<float, float>( 0.0f , 0.500f ) );
transferFunction.blueChannel.push_back( pair<float, float>( 180.0f, 0.230f ) );
transferFunction.blueChannel.push_back( pair<float, float>( 255.0f, 0.080f ) );

transferFunction.alphaChannel.push_back( pair<float, float>( 0.0f , 0.000f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 115.0f, 0.000f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 150.0f, 0.050f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 175.0f, 0.065f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 195.0f, 0.300f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 225.0f, 0.450f ) );
transferFunction.alphaChannel.push_back( pair<float, float>( 255.0f, 1.000f ) );

transferFunctionManager.AddTransferFunction( "TransferFunction", transferFunction );
transferFunctionManager.SetActiveTransferFunction( "TransferFunction" );
```

Figure A.3: Creating a transfer function and adding it to the transfer function manager.

```
CFileLoader fileLoader;
fileLoader.LoadFile( "file.vsv" );
```

Figure A.4: Loading a Vesuvius file.

```
float *data;
int iBrickSize;
int iNumBricks[3];
vector<int> order;
vector<string> activeLayers;


order        = fileLoader.OrderBricks();
iBrickSize   = fileLoader.GetBrickSize();
activeLayers = fileLoader.GetEnabledLayers();
fileLoader.GetNumBricks( iNumBricks );

// Go through and render in correct order
for ( i = 0; i < (int)order.size(); ++i )
{
    // Go through each ACTIVE layer and render them
    for ( j = 0; j < (int)activeLayers.size(); ++j )
    {
        // Set the transfer function as active so it creates the texture
        transferManager.SetActiveTransferFunction( "TransferFunction" );

        // Set the transer function for the renderer
        renderer>SetTransferFunction( transferManager.TEXTURE WIDTH,
                                      transferManager.GetTexture() );

        // Get data for specific layer
        data = m fileLoader.GetBrick( activeLayers[j], order[i] );

        // Render brick into offscreen buffer
        glPushMatrix();
            renderer>RaycastBrick( data, order[i], iBrickSize, iNumBricks );
        glPopMatrix();

        delete data;
    }
}

// Render composited image to screen
renderer>Render();
```

Figure A.5: Rendering data sets in Vesuvius

Bibliography

[1] Jim Arlow and Ila Neustadt. *Uml and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, 2001.

[3] F. P. Brooks. Grasping reality through illusioninteractive graphics serving science. In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[4] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM Press.

[5] Jian Chen, Yung-Chin Fang, R. Bowen Loftin, Ernst L. Leiss, CChing yao Lin, and Simon Su. An immersive virtual environment training system on real-time motion platform. *Proc. of the Computer Aided Design and Computer Graphics*, 2:951–954, 2001.

[6] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. The cave: audio visual experience automatic virtual environment. *Commun. ACM*, 35(6):64–72, 1992.

[7] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[8] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories*. Pearson Addison-Wesley, 2003.

[9] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama, and Daniel Weiskopf. Real-time volume graphics. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 29, New York, NY, USA, 2004. ACM Press.

[10] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. A K Peters, Ltd., 2006.

[11] Georg A. Grell, Jimy Dudhia, and David R. Stauffer. *A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5)*. National Center for Atmospheric Research, Boulder, CO, USA, 1994.

[12] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 53–60, Washington, DC, USA, 2002. IEEE Computer Society.

[13] Milan Ikits and Marcelo Magallon. Glew: The opengl extension wrangler library. http://glew.sourceforge.net. Accessed October 22, 2007.

[14] Fakespace Systems Inc. Fakespace systems inc. cave. http://www.fakespace.com/flexReflex.htm. Accessed October 24, 2007.

[15] Infiniscape. Vrjuggler. Accessed October 21, 2007.

[16] Intersense. Intersense is-900 wireless tracking modules. http://www.intersense.com/products/prec/is900/wireless.htm. Accessed October 24, 2007.

[17] Intersense. Is-900 wireless tracking modules. http://www.intersense.com/products/prec/is900/wireless.htm. Accessed October 21, 2007.

[18] Steven G. Johnson and Jim Edwards. Vis5d. http://vis5d.sourceforge.net. Accessed October 21, 2007.

[19] Gordon Kindlmann and James W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86, New York, NY, USA, 1998. ACM Press.

[20] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[21] Gunter Knittel. The UltraVis system. In *Proceedings of the IEEE Symposium on Volume Visualization*, pages 71–79, 2000.

[22] David Koller, Peter Lindstrom, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner. Virtual gis: A real-time 3d geographic information system. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 94, Washington, DC, USA, 1995. IEEE Computer Society.

[23] Andrea Kratz, Markus Hadwiger, Anton Fuhrmann, Rainer Splechtna, and Katja Bühler. Gpu-based high-quality volume rendering for virtual environments. In *International Workshop on Augmented Environments for Medical Imaging and Computer Aided Surgery (AMI-ARCS) 2006*, 2006.

[24] Oliver Kreylos. Vrui. Accessed October 21, 2007.

[25] Jens Krüger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(Annual Conference Series):451–458, 1994.

[27] Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 355–361, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[28] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[29] Marc Levoy. Efficient ray tracing of volume graphics. *ACM Transaction on Graphics*, 9(3):245–261, 1990.

[30] R. Bowen Loftin, B. Montgomery Pettitt, Simon Su, Chris Chuter, J. Andrew McCammon, Chris Dede, Brenda Bannon, and K. Ash. Paulingworld: An immersive environment for collaborative exploration of molecular structures and interactions. In *NORDUnet'98, 17th Nordic Internet Conference*, 1998.

[31] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[32] Michael Meißner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis. A practical evaluation of popular volume rendering algorithms. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 81–90, New York, NY, USA, 2000. ACM Press.

[33] Torsten Möller, Raghu Machiraju, Klaus Mueller, and Roni Yagel. Evaluation and design of filters using a taylor series expansion. *IEEE Transactions on Visualization and Computer Graphics*, 03(2):184–199, 1997.

[34] Benjamin Mora, Jean Pierre Jessel, and René Caubet. A new object-order ray-casting algorithm. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 203–210, Washington, DC, USA, 2002. IEEE Computer Society.

[35] Klaus Mueller, Torsten Möller, and Roger Crawfis. Splatting without the blur. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 363–370, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[36] J. O. Robinson. *The Psychology of Visual Illusion*. Dover Publications, Mineola, NY, USA, 1998.

[37] Stefan Roettger, Michael Bauer, and Marc Stamminger. Spatialized transfer functions. In *IEEE VGTC Symposium on Visualization (2005)*, 2005.

[38] Henning Scharsach. Advanced gpu raycasting. In *Proceeding of Central European Seminar on Computer Graphics (CESCG05)*, pages 69–76, 2005.

[39] Jürgen P. Schulze, Roland Niemeier, and Ulrich Lang. The perspective shear-warp algorithm in a virtual environment. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 207–214, Washington, DC, USA, 2001. IEEE Computer Society.

[40] Ren Shao. Ren shao - work. http://blogs.vislab.usyd.edu.au/index.php/RenShao?cat=136. Accessed October 24, 2007.

[41] William R. Sherman. Freevr. http://www.freevr.org. Accessed October 21, 2007.

[42] William R. Sherman and Alan B. Craig. *Understanding Virtual Reality: Interface, Application, and Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[43] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[44] Rajagopalan Srinivasan, Shiaofen Fang, and Su Huang. Volume rendering by template-based octree projection. In *Proceedings of the Eurographics Workshop on Visualization and Scientific Computing*, pages 155–163, 1997.

[45] Simon Stegmaier, Magnus Strengert, Thomas Klein, and Thomas Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting 2005. In *Fourth International Workshop on Volume Graphics*, pages 187–195, 2005.

[46] J. Edward Swan, Klaus Mueller, Torsten Möller, Naeem Shareef, Roger Crawfis, and Roni Yagel. An anti-aliasing technique for splatting. In *Proceedings of IEEE Visualization 97*, pages 197–204, 1997.

[47] Jon Sweeney and Klaus Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 95–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[48] Lee Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, New York, NY, USA, 1990. ACM Press.

[49] Glen Wheles, Cathy Lascara, Bill Hibbard, and Brian Paul. Cave5d. http://www-unix.mcs.anl.gov/ mickelso/CAVE2.0.html. Accessed October 21, 2007.

[50] Orion Wilson, Allen VanGelder, and Jane Wilhelms. Direct volume rendering via 3d textures. Technical report, Santa Cruz, CA, USA, 1994.

[51] Roni Yagel and Zhouhong Shi. Accelerating volume animation by space-leaping. In *Proceedings of Visualization '93*, pages 62–69, 1993.

[52] Ching yao Lin, R. Bowen Loftin, Ioannis A. Kakadiaris, David T. Chen, and Simon Su. Interaction with medical volume data on a projection workbench. In *The Proceedings of the 10th International Conference on Artificial Reality and Telexistence*, pages 148–152, Taipei, Taiwan, 2000.