A Volume Rendering Engine for Desktops, Laptops, Mobile Devices and Immersive Virtual Reality Systems using GPU-Based Volume Raycasting

Christian John Noon
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Biomedical Engineering and Bioengineering Commons, Computer Engineering Commons, and the Radiology Commons

Recommended Citation
https://lib.dr.iastate.edu/etd/12419

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
A volume rendering engine for desktops, laptops, mobile devices and immersive virtual reality systems using gpu-based volume raycasting

by

Christian John Noon

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Co-majors: Human Computer Interaction; Computer Engineering

Program of Study Committee:
Eliot Winer, Co-Major Professor
James Oliver, Co-Major Professor
Stephen Gilbert
Chris Harding
Tim Bigelow

Iowa State University
Ames, Iowa
2012

Copyright © Christian John Noon, 2012. All rights reserved.
# TABLE OF CONTENTS

## LIST OF FIGURES

v

## LIST OF TABLES

xi

## ABSTRACT

xii

## 1 INTRODUCTION

1.1 What is Volume Rendering? 1
1.2 Medical Imaging 3
1.3 Benefits of Volume Rendering 4
1.4 Real-World Volume Rendering Applications 7
1.5 Motivation 9
1.6 Dissertation Organization 12

## 2 THE VOLUME RENDERING PIPELINE

2.1 Computer Graphics and the OpenGL Rendering Pipeline 13
2.2 Volumetric Data 15
2.3 The Volume Rendering Pipeline 17
  2.3.1 Segmentation 18
  2.3.2 Gradient Computation 18
  2.3.3 Resampling 19
  2.3.4 Classification 21
  2.3.5 Coloring 22
  2.3.6 Shading 24
  2.3.7 Compositing 25
2.4 Volume Rendering Techniques 26
  2.4.1 Iso-surface Surface Rendering 27
  2.4.2 Image Splatting 28
  2.4.3 Shear Warp 29
  2.4.4 Texture Slicing 29
  2.4.5 Raycasting 30
2.5 Raycasting Execution 30
2.6 A Real-World Example of Raycasting 31

## 3 ADVANCED VOLUME RAYCASTING AND APIs

3.1 Advances in Volume GPU-based Raycasting 39
  3.1.1 Rendering Speed Optimization 39
3.1.2 GPU Texture Optimization 42
3.1.3 Lighting and Shadowing 44
3.1.4 Clipping 47
3.1.5 Rendering Multiple Volumes 49
3.1.6 Other Advancements 51
3.2 Volume Rendering APIs 51
3.2.1 Desktop APIs 52
3.2.2 Immersive Virtual Reality APIs 54
3.2.3 Mobile Device APIs 56
3.2.4 Commercial and Open Source Volume Rendering Applications 56
3.3 Research Issues 58

4 METHODOLOGY 61

4.1 Developing the Rendering Core Foundation 62
4.1.1 OpenSceneGraph 62
4.1.2 DCMTK 63
4.1.3 VR Juggler 64
4.2 The Desktop Sandbox Application 65
4.2.1 Architecture 65
4.2.2 Features 65
4.2.3 User Interface 76
4.2.4 Challenges and Contributions 78
4.3 The Immersive Sandbox Application 80
4.3.1 Architecture 80
4.3.2 Features 81
4.3.3 User Interface and Interaction 83
4.3.4 Challenges and Contributions 90
4.4 The Mobile Sandbox Application 92
4.4.1 Architecture 92
4.4.2 Raycasting Complications 97
4.4.3 Memory Limitations 101
4.4.4 GPU Fragment Operation Bandwidth 102
4.4.5 GPU Asynchronous Processing and Synchronization 111
4.4.6 Features 116
4.4.7 User Interface 128
4.4.8 Challenges and Contributions 133

5 VIPRE 135

5.1 VIPRE Architecture 135
5.1.1 The vipre Library 137
5.1.2 The vipreDICOM Library 138
5.1.3 The vipreViewer Library 138
5.1.4 The vipreRaycaster Library 139
5.1.5 The vipreOTSlicer Library 140
5.1.6 The vipreVATSlicer Library 140
5.2 Advanced Volume Raycasting Techniques 141
5.2.1 Empty Space Skipping using Octrees 141
5.2.2 Phong Illumination 146
5.2.3 Multi-Pass Rendering for Backface Depth Rasterization 148
5.3 Bridging Academic Research and Volume Rendering APIs 153

6 CONCLUSIONS AND FUTURE WORK 155
6.1 Summary and Conclusions 155
6.2 Future Work 158
6.3 Acknowledgements 158

BIBLIOGRAPHY 160
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A Full body view of a virtual forensic autopsy [71] (Top). Volume rendering of the UTCT Chameleon dataset [45] (Bottom-Right). A close up view of the lungs and throat of a three week old infant [71] (Bottom-Right).</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>An X-ray of the chest (Top-Right). A CT image of the chest (Top-Left). An Ultrasound of the abdomen (Bottom-Right). An MRI of the knee (Bottom-Left).</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Illustration demonstrates the CT process and how a set of 2D slices can generate a 3D volumetric dataset.</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Multimodal view of a head, tumor, cortical activations and fiber tracts (Right). Several views of a clipping skull for neurosurgical planning (Left). [104]</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Screenshot of the Sinus Endoscopic system’s interface.</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>Screenshot of the user interface of BodyViz, a volume rendering application designed for surgical planning and medical training.</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>A combination of DOSE and CT data. Red contours show the target volume outline, blue contours show the rectum and pink contours show the bladder (Top). Visualization of the dose distribution on areas that have high CT values. The bladder can be seen in the middle because a contrast agent was used during CT scanning to highlight softer tissues.</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>The OpenGL rendering pipeline. (<a href="http://www.songho.ca/opengl/gl_pipelin">http://www.songho.ca/opengl/gl_pipelin</a> e.html)</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>Shows the difference between an isotropic and anisotropic grid.</td>
<td>16</td>
</tr>
</tbody>
</table>
Figure 11: The volume rendering pipeline.

Figure 12: Diagram of raycasting in 2D where each ray is cast from the eye-point in a perspective projection. Image courtesy of [45].

Figure 13: Diagram of a single voxel C surrounded by 8 neighboring voxels (Left). Diagram of how Trilinear Interpolation can be used to compute the value of C (Right). ([http://en.wikipedia.org/wiki/Trilinear_interpolation](http://en.wikipedia.org/wiki/Trilinear_interpolation))

Figure 14: The same volume rendering image using two different color transfer functions with the same opacity transfer function.

Figure 15: A black pool ball in a dim room (Left). A black pool ball in a dim room with a small flashlight shown on it (Right).

Figure 16: An iso-surface surface rendering of a human skull. ([http://www.aravind.ca/images/ivis_gallery/isoColour.png](http://www.aravind.ca/images/ivis_gallery/isoColour.png))

Figure 17: Examples of image splatting on a full head dataset [190].

Figure 18: Shear-warp sampling always takes place in orthogonal direction slices.

Figure 19: Texture slicing sampling generating view-aligned slices parallel to the image plane.

Figure 20: Pseudo code of the volume rendering algorithm using raycasting.

Figure 21: Geometric representation of the volume as a surface.

Figure 22: Rendering of the volume using a grayscale color transfer function at only the ray/volume intersection points.

Figure 23: Rendering of the volume using a grayscale color transfer function and linear opacity transfer function at only the ray/volume intersections points.

Figure 24: Rendering of the volume using a grayscale color transfer function and linear opacity transfer function while resampling and compositing along the entire ray.

Figure 25: Rendering of the volume using a muscle/bone color transfer function and linear opacity transfer function while resampling and compositing along the entire ray.

Figure 26: An octree division and its tree representation.
Figure 27: Raycasting using octrees and hierarchical enumeration [47].

Figure 28: A slice of the Head dataset is partitioned using growing boxes (Left). The growing box set converted into a BSP tree (Right). [55]

Figure 29: A hand dataset rendered using Phong illumination (Left), shadow mapping (Middle) and deep shadow maps (Right). Notice deep shadow maps are the only technique to produce semitransparent shadows.

Figure 30: Interactive exploded view illustration with increasing degrees-of-explosion [110].

Figure 31: Multi-volume rendering by independently slicing each volume and depth sorting the slices into a slice stack.

Figure 32: Combination of multiple datasets using multiple rendering modes. From left to right: pre-integration with illumination, transparent isosurfaces, pre-integration with one clipping plane and a corresponding 2D slice.

Figure 33: Architecture diagram of the desktop sandbox application.

Figure 34: An image of the sandbox application rendering a bounding box encapsulating the volumetric dataset.

Figure 35: Examples of different volume rendering techniques supported by the desktop sandbox application including compositing (Top Left), MIP (Top Right) and MinIP (Bottom).

Figure 36: A chest cavity CT scan of rendering using different coloring schemes including Cardiac, Muscle and Bone, NIH and Stern from top left to bottom right respectively.

Figure 37: A close up view of a chest cavity using nearest neighbor interpolation (Top). The same close up view using trilinear interpolation (Bottom).

Figure 38: Demonstration of the clipping process. At first, the front clipping plane is positioned at the volume boundary. Next, the front clipping plane clips a portion of the front of the volume. Then, the top clipping plane clips a top portion of the volume. Finally, the right clipping plane is positioned to clip the right portion of the volume. This process is repeated each time a clipping plane is updated.
Figure 39: The general widget (Top-Left). The coloring widget (Top-Right). The windowing widget (Bottom-Left). The clipping widget (Bottom-Right).

Figure 40: Architecture diagram of the immersive sandbox application.

Figure 41: Several screenshots of the immersive sandbox application.

Figure 42: Each of the four custom widgets used for the user interface in the immersive sandbox application. Several screenshots of the immersive sandbox application.

Figure 43: A schematic of the gamepad controls used to control the immersive sandbox application.

Figure 44: Original architecture diagram of the mobile sandbox application.

Figure 45: The modified architecture using native iOS view management instead of the internal GraphicsWindowIOS implementation from OSG.

Figure 46: Screenshot of the mobile sandbox application with the pre-render camera texture displayed on top of the upscaled render.

Figure 47: A 64 pixel low resolution render of the Cardiac-CT dataset (Top-Left). A 128 pixel low-medium resolution render (Top-Right). A 256 pixel medium resolution render (Middle-Left). A 512 pixel medium-high resolution render (Middle-Right). A full resolution render at 703 pixels (Bottom).

Figure 48: A 32 slice low sampling render of the Cardiac-CT dataset (Top-Left). A 64 slice low-medium sampling render (Top-Right). A 128 slice medium sampling render (Middle-Left). A 256 slice medium-high sampling render (Middle-Right). A 355 slice high sampling render (Bottom).

Figure 49: A comparison of the medium-medium and full quality renders of the Cardiac dataset to show they are almost exactly the same despite the performance enhancements (Top). A comparison of the med-med and full quality renders of the Cardiac-CT dataset (Middle). A comparison of the med-med and full quality renders of the Manix dataset (Bottom).

Figure 50: Screenshots of each of the three custom background gradients supported in the mobile sandbox application.
Figure 51: The linear opacity transfer function (Top-Left). The linear opacity transfer function with sharpening (Top-Right). The normal opacity transfer function (Bottom-Left). The normal opacity transfer function with sharpening (Bottom-Right).

Figure 52: The “Muscle and Bone” color transfer function (Top-Left). The “Cardiac” color transfer function (Top-Right). The “Bone” color transfer function (Bottom-Left). The “Stern” color transfer function (Bottom-Right).

Figure 53: Two different examples of how the mobile sandbox application can accurately compute the intersection points with the volume bounding box.

Figure 54: A diagram of the elimination method used to sort the clipping plane bounding box intersection points.

Figure 55: Screenshot of the incorrect desktop sandbox application clipping with non-depth sorted clipping planes and bounding box (Left). Screenshot of the mobile sandbox application with proper depth sorting (Right).

Figure 56: A diagram of the scenegraph structured used to perform proper depth-sorted volume rendering.

Figure 57: The Inspector in the mobile sandbox application at launch (Left). The Inspector animating in all the widgets after the Yuria dataset was loaded (Middle). The Inspector after the animation completes (Right).

Figure 58: The Dataset view in the mobile sandbox application at launch (Left). The Dataset view while selecting a dataset (Middle-Left). The Dataset view after the progress indicator faded in and began spinning (Middle-Right). The Dataset view after the progress indicator faded out and the checkmark faded in after the dataset finished loading (Right).

Figure 59: The Clipping view in the mobile sandbox application at launch (Left). The Clipping view after turn clipping on and the widgets all faded in (Middle-Left). The Clipping view after is has been used for a while (Middle-Right). The Clipping view after hitting the reset button (Right).

Figure 60: A generic architecture diagram for all platforms supported by VIPRE.
Figure 61: The vipreDefense example rendering the Yuria dataset at 15 fps (Top). The same view and dataset with octree traversal enabled rendering at 56.8 fps (Bottom).

Figure 62: A closeup screenshot of the Cardiac dataset rendered in the vipreDefense example application (Top). The same closeup with octree traversal enabled (Bottom).

Figure 63: The vipreDefense application with the Yuria dataset loaded with default rendering (Top). The same dataset and view rendered with forward differences Phong illumination (Middle). The same dataset and view rendered with central differences Phong illumination (Bottom).

Figure 64: The desktop sandbox application demonstrating that rendering the volume in front of the clipping plane is done incorrectly (Top). The same rendering parameters with a different camera position where the volume is located behind the clipping plane resulting in the proper render (Bottom).

Figure 65: Vertex shader used for multi-pass rendering using backface rasterization.

Figure 66: Fragment shader used for multi-pass rendering using backface rasterization.

Figure 67: The vipreDefense application rendering the volume using multi-pass rendering for backface rasterization. The overlay in the bottom left is the backface depth texture generated from the first render pass.
LIST OF TABLES

Table 1: Total number of multiplications, additions and subtractions required for each interpolation method in three dimensions. Table courtesy of [4].

Table 2: A breakdown of the rendering performance when using different combinations of resolution and sampling rate for three different sized datasets.
Volume rendering is the process of visualizing characteristics and properties of three-dimensional (3D) volume data as a 3D object. The most extensive use of volume rendering takes place within the medical field. Physicians are using a combination of medical imaging technologies and volume rendering techniques to non-invasively examine patients to make critical medical decisions and diagnoses such as finding tumors, searching for blood clots and monitoring unborn fetuses. As the technological computing power continues to increase at a rapid rate, so do the opportunities to provide volume rendering solutions on new and innovative platforms such as mobile devices and immersive clustered environments. This dissertation presents a new volume rendering engine for visualizing volumetric data on multiple platforms. Three different sandbox applications were developed to investigate the challenges and architectural requirements in encapsulating the platform specific volume rendering logic inside the engine to abstract the complexity from the application level. The development of the sandbox applications resulted in the completion of the Volume Image Processing and Rending Engine, or VIPRE.

To encapsulate the platform specific implementation inside the engine, several open source application programming interfaces (APIs) were identified as worthy candidates to support the engine’s volume rendering core. OpenSceneGraph (OSG) is an open source, cross-platform graphics toolkit that supports high performance rendering through components critical to the volume rendering pipeline. The DICOM Toolkit (DCMTK) is a collection of libraries and applications
implementing a large majority of the DICOM standard capable of examining, constructing and converting DICOM image files. Finally, VR Juggler is a cross-platform, open source virtual reality software development environment designed specifically for creating and executing immersive applications. With native OSG support, application data serialization, display and device abstraction and cluster node swap barriers, VR Juggler was an ideal API for ensuring adequate performance in cluster configurations.

With the architectural design in place, three sandbox applications were developed to investigate platform specific challenges and opportunities. The desktop application was developed to create the core volume rendering algorithms for the engine such as resampling, coloring, shading and compositing. The development also produced several unique contributions including real-time windowing, a GPU compositing algorithm supported by all generic graphics cards and a convex clipping plane algorithm that supports an unlimited number of clipping planes. The immersive sandbox application was built on top of the same volume rendering core designed in the desktop application. With no modifications, the volume rendering core was successfully implemented into the immersive application resulting in the first GPU-based volume raycasting solution for immersive clustered environments. The mobile sandbox application investigation proved that despite the improved computational power of mobile devices, they are still not powerful enough to support raycasting due to the lack of 3D texture support. However, mobile devices are now fully capable of supporting orthogonal texture slicing. The development of orthogonal texture slicing required the invention of several performance enhancing features including dynamic
modification of the render resolutions, an incremental render loop, a shader-based
clipping algorithm to support OpenGL ES 2.0, and an internal backface culling
algorithm for properly sorting rendered geometry with alpha blending.

The development of the sandbox applications proved that the encapsulation
of platform specific volume rendering logic was possible with the designed
architecture. This resulted in the development of VIPRE, a unified solution for
performing volume rendering on multiple platforms. VIPRE contains many common
volume rendering features such as multiple render modes, color and opacity transfer
functions and trilinear interpolation. It also contains many more advanced features
including real-time windowing, custom CPU and GPU clipping algorithms, accurate
depth sorting, dynamic render quality modification, early ray termination and empty
space skipping, Phong illumination and multi-pass rendering for backface depth
rasterization. VIPRE is going to be released with examples and documentation to
help lower the barrier to entry for novice developers. It is going to be released under
licensing terms allowing use in both academic and commercial communities.

Future work of VIPRE includes extending the compositing algorithm to
support the insertion of surgical instruments into the volume for surgical planning.
Additionally, the integration of segmentation routines would allow new methods of
interaction for segmentation routine training to be studied for different platforms.
VIPRE will also be extended to support multiple volumes and independent clipping
for visualizing segmented data. A final area of optimization would include reusing
previous rendered textures to lazily render the volume while interacting with the user
interface in immersive environments.
1 INTRODUCTION

1.1 What is Volume Rendering?

Volume rendering is the process of visualizing characteristics and properties of volumetric data as a three-dimensional (3D) object. The volumetric data most often consists of two-dimensional (2D) images sampled at consistent intervals, then stacked sequentially to form a rectangular grid, similar to a lattice or Rubik’s Cube structure. This is fundamentally very different from surface rendering where all the polygons are rendered using exact surface representations. Volume rendering instead represents all the data as a large block of information, and dynamically interprets the data in which to render. Several examples of volume rendering can be seen in Figure 1.

The internal information contained within a volumetric dataset most often does not consist of defined surfaces or edges. In the first volume rendering implementations, surfaces within the volumetric data were approximated using geometric primitives, then rendered using well-established surface rendering techniques. The downside to this approach was that a large portion of the 3D dataset was lost due to the surface approximation. Additionally, modifying the approximated surface in any way meant the approximation computation needed to be recomputed, which would cause a large drop in rendering speed due to the high computational expense of recomputing the surface approximation.

In order to address this issue, true volume rendering techniques were developed to accommodate the entire 3D dataset into the 2D image instead of
displaying a small segmented portion as a surface. These volume rendering techniques were able to display all the 3D information in each rendered frame. However, all this new functionality came at a cost. The new techniques consisted of much more complex rendering algorithms, and significantly increased rendering times. This led to many advances in volume rendering software optimization that benefited from the continue increase of hardware acceleration. [1]

Volume rendering can be utilized by any industry or area of research involved with 3D datasets. Some of these different disciplines include medical imaging and surgical planning, nondestructive evaluation, modeling simulations, movie special effects, archaeological digs and microbiological visualization to name a few. By far
the largest area of volume rendering research and usage is performed by the medical industry. Medical imaging was one of the first applications of volume rendering, and has continued to be the driving force behind most of the volume rendering research over the past two decades.

1.2 Medical Imaging

Medical imaging began in 1895 when William Conrad Röntgen created the first x-ray of his wife's hand. This event began a whole new area of medical examination research known as medical imaging. Since then, additional medical imaging technologies such as Magnetic Resonance Imaging (MRI), Computed Tomography (CT) scans, and Ultrasound imaging have been developed. Images of each of these techniques can be seen in Figure 2.

The two most commonly used medical imaging techniques for volume rendering are CT scans and MRI images. A CT scan is a cross-sectional image obtained from different angles of the patient's body using ionizing radiation from an x-ray tube [2]. As the patient slides into the rotating X-Ray tube, 2D slice images are generated at consecutive intervals. On the other hand, an MRI is generated by the emission and absorption of electromagnetic energy in the radio frequency (RF) range of the electrostatic spectrum [3]. Different areas of the scanned object absorb and emit different variations, which form the basis of the MRI image. Today's CT scanners and MRI machines typically generate scans of 512 x 512 or 1024 x 1024 pixels. The slices can then be merged into a single 3D representation which can be used in volume rendering which can be seen in Figure 3.
1.3 Benefits of Volume Rendering

The advancement of medical imaging technologies allowed physicians to "see" inside a patient, non-invasively. The imaging techniques helped physicians discuss and examine patients as well as assist them in making diagnoses and procedure decisions. These representations have been instrumental in finding tumors, searching for blood clots and monitoring unborn fetuses. Once physicians
began using 2D imaging technologies, the need for rendering the datasets in 3D became apparent. Physicians wanted to be able to interact with the data (rotate, zoom, fly through) as well as add color and opacity to distinguish between different tissue types. Hence, volume rendering became a possible solution. Once physicians were able to interact with the data at an inspection level, the progression included embedding surgical tools such as scalpels and trocars inside the volume for surgical

**Figure 3:** Illustration demonstrates the CT process and how a set of 2D slices can generate a 3D volumetric dataset.

**Figure 4:** Multimodal view of a head, tumor, cortical activations and fiber tracts (Right). Several views of a clipping skull for neurosurgical planning (Left). [104]
planning as seen in Figure 4. Finally, haptic feedback was integrated with volume rendering technologies to create surgical simulators for surgeons to practice operations. [4]

Other areas of research that can benefit from volume rendering include complex modeling systems for simulating different phenomena such as ocean turbulence, precipitation, hurricanes and acid rain to study atmospheric trends and

anomalies [5]. Educational institutions can use volume rendering to study the internal anatomies of animals, eliminating the need for children to physically dissect them (e.g., frogs) [6]. Nondestructive imaging and visualization of mummies can help scientists study mummification techniques without damage [7]. Geologists can visualize geological information like porosity, pressure and temperature [8]. Microbiologists can visualize high-resolution datasets of microscopic organisms without disturbing them [9]. Although each of these areas of research produce very unique datasets, volume rendering is generalized enough to visualize them all (see Figure 5), allowing each area to reap the benefits that it provides.

1.4 Real-World Volume Rendering Applications

There are many benefits of volume rendering, and there are real-world applications in the medical industry to prove it. Three such applications are the Sinus Endoscopic system, BodyViz and a Radiotherapy dose distribution system. The Sinus Endoscopy system, seen in Figure 6, is a standalone desktop application that uses volume rendering to assist physicians with sinus surgery planning and patient education [10]. For difficult cases, careful planning of the surgery is necessary due to the reduced field of view. Therefore, the system strives to provide surgeons with realistic visualization at interactive framerates to plan the surgery before it takes place. The system was used for preoperative planning in 102 cases and claims it closely resembles the intraoperative situation.

BodyViz is a standalone volume rendering application, see Figure 7, that allows visualization of medical imaging data for preoperative surgical planning as
well as medical and anatomy student learning. The user interface is controlled by an Xbox 360 controller creating a much lower learning curve for users. BodyViz can be used to navigate under the skin, past bones, through arteries, blood vessels and organs and fly through patients’ bodies. The software can also create visual clipping planes as well as insert virtual surgical tools that can be maneuvered within the internal structures of the patients’ anatomy.

The final application is a virtual reality (VR) system (Figure 8) constructed to improve the understanding of spatial relationships between the patient anatomy and the calculated dose distribution of treatment plans used in radiotherapy (RT) [11]. The VR system uses interactive volume rendering to display the patient’s anatomy volume and the RT dose distribution volume simultaneously. Additionally, surface
and line rendering of RT structures such as target volumes and organs at risk are intermixed with the volume rendering. The system has been installed and networked in a room at Haukeland University Hospital where daily RT conferences are held, making stereo-scopic viewing of treatment planning data for clinical cases possible. These types of datasets are difficult to represent accurately as a geometric surface. Instead of representing the datasets as a defined surface, volume rendering techniques have been developed to render the volumetric data in its natural form.

1.5 Motivation

The benefits of using volume rendering in the areas such as medical imaging, surgical planning, nondestructive evaluation and simulation are immense. Students
no longer need to dissect animals to learn about their internal structure, fossils can be extracted from the ground without damage, virtual autopsies can be performed for determining cause of death, medical students can perform neurosurgery on a virtual simulator and so on. All of these scenarios are possible today, but unfortunately, the majority of those who need this technology on a day-to-day basis do not have it for several reasons.

The primary reason is that with all the advances in high-end volume rendering, the majority of them do not exist in available software applications or rendering application programming interfaces (APIs). Researchers have done a tremendous job pushing the boundaries of what is possible with volume rendering (Chapter 3), but these advances have, for the most part, remained in academic publications and limited software offerings. In order to allow all stakeholders involved

Figure 8: A combination of DOSE and CT data. Red contours show the target volume outline, blue contours show the rectum and pink contours show the bladder (Top). Visualization of the dose distribution on areas that have high CT values. The bladder can be seen in the middle because a contrast agent was used during CT scanning to highlight softer tissues.
with volume rendering to extend this work, it would be beneficial for all to have full access to the technology. These volume rendering techniques are too complex to require each stakeholder to have to implement their own rendering engine.

Another reason this technology is not widely available is that current volume rendering software and APIs are almost all designed specifically for high-end desktops. With the advancements of gaming technology and the widespread adoption of 3D movies, immersive virtual reality systems have become much more prevalent. Additionally, mobile computing devices such as iPad and Android tablets are being distributed in hospitals and schools worldwide [12, 13]. These devices are now powerful enough to drive complex volume rendering scenarios. Yet the open source community has very little native support for these different platforms.

To expand the reach of this technology, a volume rendering engine needs to be built to support multiple computing platforms from the very beginning stages of development. Therefore, when an advanced volume rendering feature is added, all the platforms benefit immediately, rather than requiring multiple volume rendering engines to support each individual platform. Additionally, these APIs are generally built directly upon OpenGL, and do not support various geometry file formats natively. This is an important consideration when developers need to intermix surgical tools, virtual environments and other intricate surface models with the volume. By considering all these API design issues before development, a volume rendering engine could benefit a larger audience with more platforms.

If such a volume rendering engine could support multiple platforms and was free to the public, developers could build unique native interfaces to support a
multitude of volume rendering applications for all disciplines. Such a volume rendering engine would lower the barrier to entry to researchers and developers alike. These individuals would be able to use the engine for advanced volume rendering techniques, and could instead focus their efforts on user experience and user interface design, as well as extending their applications to support multiple disciplines. Competition fosters innovation, and by making volume rendering more accessible to researchers and developers, everyone would benefit.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 presents a discussion of the volume rendering pipeline, various volume rendering techniques and raycasting execution. Chapter 3 presents an in-depth literature review of the advances in GPU volume rendering, different platform challenges, current volume rendering APIs and identifies the research issues. To investigate the challenges of abstracting the platform specific volume rendering core from the application level, three sandbox applications were built with a common architecture and are discussed in Chapter 4. Chapter 5 presents the Volume Image Process and Rendering Engine (VIPRE). Finally, the dissertation is concluded and summarized, with conclusions formed and future work defined in Chapter 6.
2 THE VOLUME RENDERING PIPELINE

2.1 Computer Graphics and the OpenGL Rendering Pipeline

Due to the complexities of volume rendering, it is imperative to first have a basic understanding of computer graphics as well as knowledge of the OpenGL rendering pipeline. Computer graphics, also known as computer rendering, is the process of generating an image from a 3D geometric scene. The scene can contain many different objects, each with their own individual characteristics that describe how to render them such as geometry, texture, lighting and shading. After the scene is set up, it is passed to a rendering program which processes the information into a single digital image, or frame. In a computer graphics application, the rendering process is continuous, meaning frames are rendered sequentially one after another until the application is terminated.

To better understand the rendering process, a diagram of the OpenGL rendering pipeline can be seen in Figure 9. Geometry data (vertices, lines and polygons) follow the geometry path which includes vertex operations and primitive assembly. Pixel data (pixels, images and bitmaps) instead travel through the image path that includes pixel transfer operations and texture memory allocation. Both paths are then combined at the rasterization stage, undergo fragment operations and are finally written into the framebuffer. The following is a more detailed description of the key stages of the rendering pipeline.

At the beginning of each frame, all the data is initially represented as a display list, whether it is geometry or pixels. The vertex data of the geometry is then
directed to the vertex operations stage of the rendering process. This is where each vertex is first transformed into a primitive. The vertex is also reprojected from its position in the 3D world to a position on the screen. If enabled, more complex operations are also performed such as generating texture coordinates, computing lighting characteristics and material properties. Primitive assembly handles both clipping and culling operations. Clipping removes parts of lines and/or polygons from the scene that fall on the clipped side of a plane (e.g., not viewable from a certain viewpoint). Culling is performed after clipping and removes front and/or back faces from polygons depending on which mode is specified. Once completed, the geometry primitives are complete with color, depth and texture coordinates for the rasterization step.
At the same time vertex data is being sent down the geometry path, display lists representing pixel data are sent down the image path to the pixel operations stage. There pixels from system memory are unpacked from their current format into the proper number of components. The data is then scaled, biased and processed by a pixel map. Finally, the results are written to texture memory or sent to the rasterization step.

The rasterization stage then converts the geometric and pixel data into fragments. These fragments correspond to a particular pixel in the framebuffer and are assigned color and depth. Before the fragments are stored into the framebuffer, they undergo a series of fragment operations including texturing, fog application, the scissor test, the alpha test, the stencil test, the depth-buffer test, blending and dithering. After making it through all the tests, the fragment is written into the framebuffer where it is finally displayed as a pixel of the rendered frame. For more details regarding the OpenGL rendering pipeline, please refer to the OpenGL Programming Guide [14].

2.2 Volumetric Data

Before volume rendering can be performed, one must first acquire a volumetric dataset. A volumetric dataset generally consists of a set of \( V \) samples \((x, y, z, v)\), which are also referred to as voxels. Each voxel contains location information \((x, y, z)\) as well as the value \( v \), some property of the volumetric data. The value of the voxel can vary widely between different types of datasets. For instance, the value could be a measurable property of the data such as color,
density, intensity, pressure or heat. These in particular happen to all be one-
dimensional (1D) values. The value $v$ for each voxel could also be multidimensional
for data types such as velocity $(x',y',z')$ or color $(r',g',b')$. To add another
layer of complexity, the dataset could vary with time meaning the dataset becomes a
four-dimensional (4D) set of samples $(x,y,z,t,v)$. [1]

Volumetric datasets are generally isotropic, meaning samples are taken at
regular intervals along each of the three orthogonal axes. Datasets where the
sample size varies equally between axes is referred to as anisotropic. Both types of
datasets can be seen in Figure 10. Isotropic and anisotropic datasets can be defined
on a consistent regular grid or 3D array (also known as volume buffer). The 3D array

![Figure 10: Shows the difference between an isotropic and anisotropic grid.](image)

- **Isotropic**
- **Anisotropic**
- **Rectilinear**
- **Curvilinear**
- **Unstructured**
is then used in combination with the volume rendering algorithm to produce the final 2D compositied image. In addition to regular volumetric datasets, there are also irregular datasets such as rectilinear, curvilinear and unstructured, which can also be seen in Figure 8. Most volumetric datasets consist of regular grids, yet volume rendering of irregular datasets can still be accomplished at high computational expense [1].

2.3 The Volume Rendering Pipeline

Once a volumetric dataset is acquired, there are many stages of operations required to generate a volume rendered image. Each stage of the volume rendering pipeline can be seen in Figure 11. It is important to note that this is merely a generic volume rendering pipeline, not all stages are required nor in the given order. However, most volume rendering implementations include each of these stages.

[Diagram: The volume rendering pipeline]
2.3.1 Segmentation

Segmentation is a preprocessing stage that partitions the volumetric data into multiple segments. For example, segmentation routines could be used to find a tumor, locate bone tissue or extract specific organs. In order to visualize segmented data in volume rendering, one can render the segmented volume separate from the original, render the segmented volume as a surface, or tag and store each voxel contained within a segment in the volumetric data. This information can then be used later on in the rendering process to change the visualization of the segmented voxels. This is typically accomplished by changing the color and opacity of the voxels in comparison to the rest of the volume. Segmentation is usually performed before rendering and typically only performed once. For more details on segmentation, see the following references. [15-17]

2.3.2 Gradient Computation

The next stage of the pipeline is gradient computation. This stage is responsible for finding edges or boundaries between different materials in the dataset. The gradient is a 3D vector containing orientation and magnitude that reveals the amount of variation between a voxel and its neighboring voxels. Gradients for all voxels can be computed using many different methods. Some commonly used gradient methods are the Central Difference Gradient Estimator, the Intermediate Difference Operator and the Sobel Operator [18-20]. Central and Intermediate Difference only use six of their neighboring voxels for computing the gradient and is relatively easy to implement. This allows both methods to be
executed quickly for continuous gradient evaluation for each rendered frame. However, neither of these methods is a very accurate gradient estimator. The Sobel Operator is much more accurate because it uses all 26 neighboring voxels to compute the gradient at the expense of computational efficiency. This operator is better to use in cases where the gradient for each voxel is only calculated once and stored in memory instead of being calculated each frame. Once the gradients are computed for each voxel in the dataset, the information can be reused in the classification and shading stages.

2.3.3 Resampling

Upon completion of the segmentation and gradient computations, rendering can begin. The first rendering stage is resampling. In this stage, imaginary rays are emitted from each pixel of the framebuffer screen coordinates in the view direction through the 3D scene. Rays that do not intersect with the volume will simply render the background color. The other rays will start sampling at the first intersection with the volume, or at $f_i$ as shown in Figure 12. Additional samples will then be taken and accumulated at specified intervals along the ray until it exits the volume at $I_i$.

Unfortunately, the sample location rarely correlates to an exact voxel location. For this reason, interpolation methods are used to generate approximate values for samples that lie in between a group of voxels. Some commonly used methods are Nearest Neighbor, Trilinear Interpolation, B-splines and Tricubic Interpolation [21, 22]. The computational complexity of each of these methods in three dimensions can be seen in Table 1. Nearest neighbor is the fastest method, but also produces
the worst results as it does not perform any interpolation. The Tricubic Convolution and B-spline methods produce highest-quality results, but have a high computational expense. [4]

For real-time applications, Trilinear Interpolation is often the most reasonable method due to its ability to reduce aliasing problems with very little computational overhead. Trilinear Interpolation assumes a linear relationship between an interpolation point and its neighboring points and can be performed in any particular

**Figure 12:** Diagram of raycasting in 2D where each ray is cast from the eye-point in a perspective projection. Image courtesy of [45].

**Table 1:** Total number of multiplications, additions and subtractions required for each interpolation method in three dimensions. Table courtesy of [4].

<table>
<thead>
<tr>
<th></th>
<th>Nearest Neighbor</th>
<th>Trilinear</th>
<th>Tricubic Convolution</th>
<th>B-spline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>0</td>
<td>7</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>Add/Subtract</td>
<td>3</td>
<td>14</td>
<td>39</td>
<td>39</td>
</tr>
</tbody>
</table>
order, for instance along $x$, then along $y$, and finally along $z$. To demonstrate, a voxel $C$ can be seen below in Figure 13 between 8 neighboring voxels. First, four values on the $x$-axis were computed, $C_{00}$, $C_{01}$, $C_{10}$ and $C_{11}$. Next, the values were interpolated on the $z$-axis producing $C_0$ and $C_1$. Finally, $C_0$ and $C_1$ were interpolated along the $y$-axis to produce the resulting value of $C$. Again, these operations can be computed in any order, and will always produce the same result.

![Diagram of a single voxel C surrounded by 8 neighboring voxels (Left). Diagram of how Trilinear Interpolation can be used to compute the value of C (Right).](http://en.wikipedia.org/wiki/Trilinear_interpolation)

**Figure 13:** Diagram of a single voxel $C$ surrounded by 8 neighboring voxels (Left). Diagram of how Trilinear Interpolation can be used to compute the value of $C$ (Right). ([http://en.wikipedia.org/wiki/Trilinear_interpolation](http://en.wikipedia.org/wiki/Trilinear_interpolation))

### 2.3.4 Classification

After computing the sampled voxel’s intensity using interpolation, the next step is to determine whether that voxel is going to be part of the accumulated ray voxel. This stage of the rendering process is called classification. Classification is one of the most powerful tools in volume rendering, because it allows certain structures to be visualized, even though they might be occluded by other objects. This is accomplished by creating a mapping between the range of voxel intensities
and opacity values between zero and one. The opacity is a measure of how translucent an object is. By assigning an opacity value to each sampled voxel, certain structures in the dataset can be skipped over if the opacity value is zero. Other the other hand, if the voxel opacity is not zero, the voxel moves on to be colored in the coloring stage of the volume rendering pipeline.

The mapping between voxel intensity and opacity is generated by an opacity transfer function [18, 23]. Designing the opacity transfer function can be quite complex, depending on what type of structures need to be visualized. Histograms are a useful tool in designing transfer functions as they reveal where the high frequency intensities in the dataset lie. Therefore, the opacity transfer function can be designed accordingly to expose certain parts of the data.

### 2.3.5 Coloring

To assign a color to the voxel, red, green and blue (RGB) transfer functions (referred to collectively as the color transfer function) are used to map voxel intensity to an RGB color value. Other voxel properties can be mapped to color as well, such as gradient direction or magnitude, but the most commonly used is voxel intensity. It is important to note that the goal of the color transfer function is to enhance the visual quality to interpret the volumetric data, not to achieve photo realism. Therefore, each of the colors can have their own color transfer function to define how red, green and blue each intensity value is. These RGB values are then combined to produce the final color for the voxel. Generally, each color uses a
unique transfer function. Otherwise, if they are all the same, a grayscale image is produced.

Figure 14: The same volume rendering image using two different color transfer functions with the same opacity transfer function.
Through the combination of interactive opacity and color transfer functions, one can explore the volume to reveal interesting characteristics. An example of volume rendering using two different color transfer functions can be seen in Figure 14. It is also possible to use a more automated approach to creating color transfer functions. He et al. [24] used stochastic search techniques to assist users in generating automated transfer functions. The benefit of this is that a wide range of colors can be applied to a small range of voxel intensity values for better distinction. This can be done manually, which can be difficult and time consuming, or automatically which is easier and sometimes more effective.

2.3.6 Shading

After the color is assigned, it is then time to apply the shading illumination model to the RGB voxel color. The goal of the illumination model is to simulate the reflection of light of a surface, and the effect it has on the observer while looking at that surface. For example, imagine what a black pool ball would like in a dim room. Now shine a small flashlight on it. The area of the ball where the light is shining is going to be close to white. This effect can be seen in Figure 15. The interaction of light at the surface of the pool ball affects the perception of the ball itself. It allows us to see the exact shape and contour of the surface more clearly.

In order to apply a shading illumination model, the first step is to calculate the gradient of the sampled voxel using one of the interpolation methods used in Section 2.3.3. The most commonly used method is Trilinear Interpolation. The gradient is then applied in combination with the light vector and view direction to a shading
illuminated model to compute the final RGB color of the sampled voxel. The most popular shading techniques in volume rendering are the Phong [25] and Gouraud [26] shading models. Both methods use ambient light, diffuse reflection and specular reflection in combination with the light vector, gradient vector and view direction to compute the shaded RGB color of the sampled voxel when it interacts with light. Finally, the voxel color and opacity is then accumulated and sampling continues.

**2.3.7 Compositing**

Since each ray that is cast can only represent a single pixel, every sampled voxel must be accumulated into a single RGBA color (the A in RGBA stands for the opacity of the color). This is the final stage of the volume rendering pipeline and is called compositing. To combine all the voxel values, either the front-to-back or back-to-front accumulation function is used. Front-to-back compositing is the more
commonly used as it offers performance enhancements over back-to-front. The front-to-back compositing function can be seen below in Equation 1:

\[
I(x, y) = \sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} (1 - \alpha_j)
\]  

(1)

In Equation 1, the total intensity for the voxel \( I(x, y) \) is the sum of \( I_i \) multiplied by all the transparencies \((1-a_j)\) encountered previously along the ray. The intensity \( I_i \) is generally represented by Equation 2:

\[
I_i = C_i \times \alpha_i
\]  

(2)

Equation 2 shows the intensity \( I_i \) is a function of the sample point color opacity. The front-to-back compositing function continuously evaluates the intensity of the current sampled voxel, then blends it with the accumulated voxel, and continues this process while the ray is still contained within the volume. A major advantage to the front-to-back compositing function is early ray termination where resampling is stopped when the accumulated voxel opacity reaches \( 1.0 \), or a sufficiently close value. The reason the resampling can be ended is that voxel sampled afterwards will no longer have any affect on the accumulated voxel color. This is one easy volume rendering optimization which can be made directly in the compositing function. For more details on compositing functions, please refer to [27-30].

2.4 Volume Rendering Techniques

Research has led to the development of several volume rendering techniques, each with their own advantages and disadvantages. An indirect volume
rendering technique is iso-surface surface rendering, while direct volume rendering techniques include image splatting, shear warp, texture slicing and composite raycasting. The following sections will introduce each technique and provide a brief overview.

2.4.1 Iso-surface Surface Rendering

Iso-surface surface rendering was developed to reduce the complexity of volume rendering by representing the volumetric data as a surface consisting only of geometric primitives. To represent the data as a surface, several methods exist for extracting the iso-surface from the volumetric data, the most common of which is the Marching Cubes algorithm [31]. Although this technique can be useful, there are

Figure 16: An iso-surface surface rendering of a human skull. (http://www.aravind.ca/images/ivis_gallery/isoColour.png)
several drawbacks. First, the geometric primitives can only approximate the surfaces in the original data. Highly accurate representations can require excessive amounts of geometric primitives. Accuracy is lost when visualizing small details of the dataset. This can be seen in Figure 16 in the back of the eye sockets. Another drawback is that since only a surface representation is used, the original volume’s information that is not represented by the surface is lost. Additionally, it is often quite difficult to distinguish different structures in a volume dataset shown by the surface. [1]

2.4.2 Image Splatting

Image splatting is a popular technique for direct volume rendering initially proposed by Westover [32] where voxels are represented by overlapping basis functions, commonly Gaussian kernels. The image is generated by projecting the basis functions to the screen as a superposition of pre-integrated 3D kernels, referred to as 2D footprints. A major advantage of image splatting is that only the volume points need to be rendered or stored. Image splatting ignores the empty volume space. However, image splatting can lead to color bleeding, aliasing, and

![Figure 17: Examples of image splatting on a full head dataset [190].](image-url)
blurring due to the issues associated with blending each splat as can be seen in Figure 17. For more information regarding image splatting, see [33-35].

2.4.3 Shear Warp

Shear warp volume rendering [36, 37] determines the face of the volume data that is most parallel to the viewing plane, then casts rays through each voxel of the base plane as shown in Figure 18. The resulting plane image is then projected onto the image plane using a 3D transformation and a 2D image resampling operation. The major advantage is that it is relatively fast since it only samples each voxel in the dataset once due to the orthogonal sampling. The downsides to this technique is that there is much less accurate sampling and lower image quality than other techniques.

![Figure 18: Shear-warp sampling always takes place in orthogonal direction slices.](image)

2.4.4 Texture Slicing

Texture slicing [38, 39] is a direct volume rendering technique that generates viewport-aligned slices parallel to the image plane whenever the view matrix is updated, see Figure 19. Unfortunately, every time the view matrix is updated, the viewport-aligned slices must be recomputed. To composite the slices together, the
textured polygon slices are blended using back-to-front compositing. Texture slicing is capable of producing higher quality images than the previous techniques with good performance. However, the technique still requires recomputing the view-aligned slices, contains artifacts with volumetric clipping and cannot use advanced lighting.

![Figure 19: Texture slicing sampling generating view-aligned slices parallel to the image plane.](image)

### 2.4.5 Raycasting

Raycasting [40] is a direct volume rendering technique that involves casting rays from each pixel in the view direction through the volume. The intersection points are computed, then resampling and compositing are used to accumulate the final pixel value. In comparison to the other techniques, raycasting is widely accepted as the best quality volume rendering technique. Additionally, it supports optimizations such as early ray termination and space leaping.

### 2.5 Raycasting Execution

Due to the parallel nature of raycasting, it is an ideal algorithm for massive parallel architectures for central processing units (CPUs) and graphics processing
units (GPUs). There are pros and cons associated with each approach. For CPU architectures, the screen is generally divided into individual sections that are processed in parallel by processing nodes or multiple threads. Once each node generates a partial image, all the images are composited into the final image and applied directly to the framebuffer. [41-43]

With the advances in GPU technology over the past decade, commodity graphics hardware is now capable of performing the entire raycasting algorithm. This was made possible by making two components of the OpenGL pipeline programmable, the vertex shader and the fragment shader. The reason GPU raycasting is so attractive is that with its intrinsic parallelism and efficient communication, the GPU can calculate much faster than the CPU [44]. However, this power comes at a price. Although GPU texture memory is continuously growing, it continues to be the bottleneck for large datasets. Not only can it be difficult to fit the 3D texture into the GPU memory, but there is the precomputed gradient information as well. GPUs offer tremendous improvements in speed and quality of raycasting volume rendering, but present difficult challenges as well. The next chapter discusses the advances of GPU raycasting volume rendering in-depth.

2.6 A Real-World Example of Raycasting

Each stage of the volume rendering pipeline is quite complex and can be difficult to comprehend without visually inspecting the results of each stage. Using pseudo code and visual comprehension, the following section will investigate a real-world example of volume rendering using raycasting. For this example,
segmentation, gradient computation and shading are not included as they are not required stages of the volume rendering pipeline. The volume rendering pseudo code can be seen in Figure 20.

To perform volume rendering using raycasting, the first stage is always to acquire the volumetric data. Lines 2 and 3 of the pseudo code import all the volume data and build a 3D texture out of the information. Once the data is loaded, the next step is to create the volume geometry. Line 6 accomplishes this by generating six geometric quads to form a 3D box object with dimensions matching the size of the volumetric dataset. This initial set up stage for volume rendering can be seen in Figure 21. Up to this point, no actual volume rendering has been performed.

The next stage of raycasting requires manual computation of the RGBA values for each pixel in the framebuffer. This is accomplished by the renderVolume() method on line 9 of the pseudo code. For each pixel, the first step is to compute the intersection point between the ray and the volume (lines 15 and 17 of the pseudo code). If the ray does not intersect the volume, the pixel is set to the background color (lines 21-25). If there is an intersection, then the next step is to extract the interpolated intensity of the voxel at the intersection point (line 38). The intensity is then converted to an RGB value using a color transfer function (line 46). The RGB value is combined with an opacity of 1.0 to produce the final pixel RGBA value (line 47). An example of this process can be seen in Figure 22. The resulting image however is not ideal. Only the outside voxels of the dataset can be visualized because there is no resampling or compositing being performed.
// First import the volume data and construct a 3D texture
importAllVolumeData();
build3DTexture();

// Create a geometric box with dimensions matching the volume
createVolumeGeometry();

// Everything is now in place to continuously render the volume
void renderVolume()
{
    // Compute the RGBA color for each pixel of the framebuffer
    for (all pixels)
    {
        // Calculate ray direction using camera matrix & pixel position
        vec3 ray_direction = computeRayDirection();

        // Compute the intersection point between the ray & volume
        vec3 intersection_point = computeRayVolumeEntryPoint();

        // If no intersection, set pixel to background color & continue
        if (intersection_point is false)
        {
            final_pixel_color = background_color;
            continue to next pixel;
        }

        // Set up the variables used for compositing
        vec4 color = dest_color = vec4(0.0, 0.0, 0.0, 0.0);
        vec4 dest_color = vec4(0.0, 0.0, 0.0, 0.0);
        float remaining_opacity = 1.0;
        vec3 pos = intersection_point;
        vec3 step = computeStepSize();

        // Step along the ray using front-to-back compositing
        while (ray.insideVolume() is true)
        {
            // Get the intensity at the sampled voxel position
            float intensity = computeIntensity(pos);

            // Get the opacity for the given intensity
            float opacity = computeOpacity(intensity);

            // Update the dest_color & remaining_opacity if voxel is opaque
            if (opacity > 0.0)
            {
                color = computeColor(intensity);
                color = color * opacity;
                dest_color = dest_color + color * remaining_opacity;
                remaining_opacity = remaining_opacity * (1.0 - opacity);
            }

            // Move to the next sample position on the ray
            pos = pos + step;
        }

        pixel_color = dest_color;
        pixel_color.a = 1.0 - remaining_opacity;
    }
}

Figure 20: Pseudo code of the volume rendering algorithm using raycasting.
Figure 21: Geometric representation of the volume as a surface.

Figure 22: Rendering of the volume using a grayscale color transfer function at only the ray/volume intersection points.
To further the raycasting process, the next stage to improving the rendering quality is to add an opacity transfer function. Instead of setting the opacity to 1.0, a linear transfer function from 0.0 to 1.0 is used on the entire range of the volumetric data (line 41). The resulting image can be seen in Figure 23. By implementing an opacity transfer function, the lower intensity values in the dataset which are mostly air are no longer rendered because they have an opacity of 0.0. The addition of the opacity transfer function exposes the internal structure of the volume. However, in this case, the internal structure of the volume is missing. This is because resampling and compositing were not included. In medical datasets, intensity correlates to tissue density. Air, a low density object, has a very low intensity while bone has a very high intensity.

**Figure 23:** Rendering of the volume using a grayscale color transfer function and linear opacity transfer function at only the ray/volume intersections points.
By adding resampling and compositing into the raycasting process, the inside of the volume can be visualized. Resampling and compositing no longer stop at the intersection point of the ray and volume, but continue along the ray until it exits the volume. For each sample point along the ray, the intensity is computed at the sample point location using trilinear interpolation (line 38). Next, the opacity of the sample point is computed by passing the interpolated intensity to the opacity transfer function (line 41). If the opacity is larger than 0.0, the sample point is used for compositing (line 44). Before the sample point can be composited, the interpolated intensity is converted to an RGB value using a color transfer function (line 46). The color is then multiplied by the opacity (line 47), and added to the accumulated destination color (line 48). The voxel opacity is then subtracted from the remaining

**Figure 24:** Rendering of the volume using a grayscale color transfer function and linear opacity transfer function while resampling and compositing along the ray.
voxel opacity (line 49). Finally, the next sample point on the ray is computed using the step vector—the \((x, y, z)\) increment for sampling along the ray (line 53), and the resampling and compositing continues until the ray exits the volume (line 35). After the ray exits the volume, the destination color and remaining opacity of the ray are combined and applied to the pixel. Once this is accomplished for each pixel in the framebuffer, an image such as shown in Figure 24 is generated.

The final improvement to the resampling and compositing process is to add color to the volume. Again remember the goal of volume rendering is to enhance the visual quality of the individual characteristics to interpret the dataset, not to achieve photo realism. In Figure 19, a grayscale color transfer function was used. Instead, Figure 25 was generated using a muscle and bone color transfer function. In

![Figure 25: Rendering of the volume using a muscle/bone color transfer function and linear opacity transfer function while resampling and compositing along the ray.](image)
comparison to the grayscale image, this image demonstrates the usefulness of color to help further enhance the visual quality of the dataset.
3 ADVANCED VOLUME RAYCASTING AND APIs

3.1 Advances in Volume GPU-based Raycasting

The advances in GPU technology over the past decade have ushered in the possibility of performing full GPU-based raycasting at interactive framerates. It will most likely replace slice-based techniques entirely in the future if the hardware capabilities continue to increase at the current rate [45]. In light of this, many advances have been made to GPU-based raycasting that are relevant to this dissertation. The advances can be broadly categorized as 1) rendering speed optimizations using techniques such as early ray termination or empty space skipping, 2) texture size optimizations such as texture compression for large, out-of-core datasets, 3) lighting and shadowing effects, 4) clipping techniques such as plane-based, hinge-slicing and exploded views, 5) multi-volume rendering with surface models using depth or opacity peeling. Each of these methods will be discussed in detail.

3.1.1 Rendering Speed Optimization

In GPU-based raycasting, early ray termination (also known as adaptive termination) is a technique used to improve rendering speed by terminating a ray before passing through the entire volume. Early ray termination can only be used when performing front-to-back raycasting due to the nature of the compositing algorithm. Whitted [46] originally proposed the idea of adaptively terminating the raytracing algorithm. Later on, Levoy [47] integrated early ray termination into front-
to-back volume raycasting by proposing two cases when it is applicable. The first case in which a ray should be terminated is if it strikes an opaque voxel. The second case also terminates ray traversal if the accumulated opacity reaches a user-specified level (generally between 0.1 and 0.01) where the color of the ray stabilizes and will no longer affect the accumulated color. Weiler et al. [48] reported that implementing early ray termination in their raycasting algorithm improved rendering speeds by up to a factor of 4 depending on the dataset.

Another technique for improving rendering speed in raycasting is called empty space skipping. This technique is built upon the fact that many datasets contain coherent regions of empty voxels, or voxels with an opacity of zero. A method for encoding 3D spatial coherence of empty voxels is to use octrees. An octree is a tree data structure that recursively subdivides a 3D volume into eight octants. At the lowest level of the octree lie the voxel cubes, a single cube enclosed by eight neighboring voxels. Each node in the octree contains a binary value representing

![Octree Division and Tree Representation](image)

**Figure 26:** An octree division and its tree representation.
whether the region contains all empty voxels. This pyramid type structure can be used in volume rendering to perform empty space skipping. An example of an octree can be seen in Figure 26.

Meagher [49] first used octrees for volume rendering by first creating a condensed representation of the volume. Then the volume was rendered by traversing the octree in a depth-first manner following a consistent direction through space. Levoy [47] extended this work by representing the volume as a complete octree and rendered the data in image order by tracing viewing rays from an observer position through the octree. An example of a single ray traversal using Levoy’s technique can be seen in Figure 27.
In addition to the methods discussed, other implementations of empty space skipping have also been investigated [50-54]. In one particularly interesting approach, Li et al. [55] partitioned the volume into sub-volumes, but did so using the growing boxes [56] approach that partitions the volume adaptively based on voxel properties, see Figure 28. The set of grown boxes is then converted into an orthogonal binary space partitioning (BSP) tree [57] to render the adaptively partitioned sub-volumes in visibility order. BSP trees are similar to octrees, except each node only contains two subregions instead of eight. This empty space skipping technique has been demonstrated to improve volume rendering by a factor of two to five.

![Figure 28: A slice of the Head dataset is partitioned using growing boxes (Left). The growing box set converted into a BSP tree (Right). [55]](image)

### 3.1.2 GPU Texture Optimization

As mentioned earlier, the bottleneck of GPU-based volume raycasting for large datasets is typically the texture memory of the GPU. Researchers have been
working on techniques to suppress the texture memory bottleneck issue such as bricking, multi-resolution volumes and compression. Each of these techniques offer a significant improvement in GPU texture memory and efficiency as well as in certain cases, rendering performance.

Bricking is a technique to divide the volume dataset into chunks, called bricks [58]. This technique is particularly suited for GPU raycasting because it can deal with datasets that exceed the available texture memory. To fit the bricks into the available texture memory, each brick must be equal to or smaller than the available texture memory on the GPU. Each brick is then loaded and unloaded to and from GPU texture memory when rendered. By rendering bricks sequentially, the texture memory is not exhausted and the entire volume can be rendered. Unfortunately, this approach leads to significantly lower frame rates, since the bus architecture connecting the GPU, CPU and main memory cannot support bricking at high frame rates [59]. In order to reduce the amount of texture switching performed on the GPU, bricking was coupled with additional techniques such as multiple resolutions, adaptive sampling and compression [60-63].

Multi-resolution rendering techniques were developed by combining several methods including bricking, octrees and unique texture caches. Lamar et al. [64] first proposed a multi-resolution sampling of octree rendering blocks at high resolution closest to the view point and lower resolution further away. Boada et al. [65] proposed a similar technique for creating an octree out of the volume, but instead set the resolution of each sub-volume using data dependent measures. An inherent problem with block-based methods, bricking and octrees, is the need to use trilinear
interpolation at block boundaries. Interpolating at block boundaries requires individual blocks to be padded, resulting in block overlaps, so interpolation can be done accurately [66, 67]. Although padding is necessary for interpolation, it is counterproductive because it results in larger block sizes. To avoid padding, Ljung et al. [68] propose an interblock interpolation technique that supports direct interpolation between block boundaries. To further improve block-based methods, other multi-resolution techniques use creative texture cache designs in combination with octrees [69, 70], bricking [71, 72] and compression [73] to accommodate for large datasets.

Another technique to improve the texture memory bottleneck is to use efficient compression schemes. Nguyen et al. [74] used blockwise compression to split the volume into small blocks equally sized and compress each block individually. Other compression techniques were proposed that operated on a wavelet representation [75-77]. Vollrath et al. [78] proposed using adaptive texture maps [79] to reduce the memory of the entire dataset, but sampling distance was not modified as the ray passed through different resolution blocks. In contrast, Guthe et al. [67, 73] used a block based wavelet compression to render the large datasets at interactive frame rates.

### 3.1.3 Lighting and Shadowing

Light interaction is an important part of volume rendering due to the major impact it has on spatial comprehension [80]. Shadows also aid spatial comprehension by serving as an important depth cue [81]. Even though the goal of
volume rendering is not to achieve photorealism, it is useful to simulate real-world lighting conditions as closely as possible. This section will discuss the Phong illumination model in combination with gradient calculations, followed by volume rendering shadowing techniques and finally ambient occlusion.

Phong illumination [25] is the most typically used illumination model for volume rendering. Due to the computational complexity of global illumination, often a simplified direction illumination model is used, that is illumination not affected by other parts of the scene that only considers light coming directly from a source. Computing the Phong illumination of a given voxel requires the current voxel position, the voxel gradient, the voxel color and the position of the light source [45]. The final voxel color is then determined after applying diffuse, specular and ambient illumination to the voxel. For additional information about the Phong illumination model, please refer to [82] for more details.

To improve the visualization of depth in volume rendering, the addition of shadows is necessary. In contrast to the shadowing techniques for slice-based volume rendering [83, 84], only a small amount of research has been done to integrate shadows into GPU-based raycasting [45]. However, it should be noted that shadows have been integrated into volume rendering raytracing systems [85-87]. Raytracing is similar to raycasting, except the ray traversal accounts for light interactions of many virtual objects. Raytracing is capable of simulating a wide variety of high-fidelity optical effects, such as reflection, refraction and scattering. However, raytracing is outside of the scope of this dissertation due to the computational complexity of performing raytracing interactively. This technique is
much more suited for non-interactive applications where images can be rendered slowly ahead of time, such as still images for animated films or special effects.

The first shadowing technique implemented into volume raycasting is shadow mapping, originally presented in 1978 by Williams [88]. Shadow mapping is an image-based approach that adds an additional render pass rendered from the light source’s point of view to determine which voxels are closest to the light source. Then in the main rendering pass, each sampled voxel undergoes a fragment-based shadow test to determine whether it should be shadowed. One benefit of shadow mapping is that soft shadows can be approximated using percentage closer filtering [89]. Shadow mapping allows for very efficient shadows on a per-fragment basis, but is not capable of generating semitransparent shadows.

To support semitransparent shadows, opacity shadow maps were developed to store alpha values instead of depth as a stack of shadow maps [90]. A more advanced technique for generating semitransparent shadows are deep shadow maps [91, 92]. Deep shadow mapping uses a stack of textures that store both depth and opacity for various layers of the shadow map. Deep shadow mapping produces much higher quality shadows than shadow mapping, but at a higher computational expense. Additionally, deep shadow mapping can produce artifacts in very thin or complex areas of the volume. These artifacts can be eliminated by generating additional shadow layers, but result in decreased performance. A comparison of Phong illumination, shadow mapping and deep shadow mapping can be seen in Figure 29.
Ambient occlusion is another shading technique that simulates global lighting by estimating the visibility of light at a given voxel. Vicinity Shading [93] is an ambient occlusion technique that pre-computes the occlusion for each voxel and stores the values in a 3D shading texture. Desgranges and Engel created a less computationally expensive version of Vicinity Shading combining ambient occlusion volumes into a composite occlusion volume [94]. Hernell et al. later proposed Local Ambient Occlusion (LAO) [85, 95] which is a technique based on casting rays in several directions from non-transparent voxels within a specified radial boundary. The LAO of each voxel increases when rays do not intersect with other voxels. Finally, Ropinksy et al. [96] proposed dynamic ambient occlusion along with color bleeding using local histograms as an alternative to Phong illumination.

3.1.4 Clipping

Clipping is a useful technique in volume rendering for exploring the internal structures of a volume. Almost all volume renderers contain at least some form of
volumetric clipping, the most basic of which are clipping planes [97, 98]. Clipping planes are artificial geometric planes that clip off the volume geometry at specified intersection points. Many more advanced clipping techniques have also been developed. McInerney and Broughton [99] used hinged slice planes to provide better contextual 3D spatial relationships. Wang et al. [100] proposed volume sculpting as a way to explore volume datasets as well as carve complex geometry out of the volume. Weiskopf et al. [101] presented a depth-based clipping technique using complex geometries to perform volume clipping. Konrad-Verse et al. [102] used deformable clipping planes for virtual resection in liver surgery planning. Additional depth-based clipping algorithms have been developed using binary clip volumes to perform complex geometric volume clipping [40, 101, 103, 104].

Another form of clipping is the use of exploded views where volume data is displaced to reveal otherwise hidden details. Niedauer et al. [105] first used clipping planes to slice geometric models into an exploded view for architectural visualization. At the same time, Chen et al. [106] used spatial transfer functions to deform volumetric data for modeling and animation purposes. Islam et al. [107] extended this work by allowing volumes to be split into many sections. McGuffin et al. [108] used deformation strategies to open up, spread apart and peel away various sections of volumetric data. Viola et al. [109] created an automated way of performing clipping based on compositing strategies that prevent an object from being occluded by a less important object. Finally, Bruckner and Gröller [110] proposed an approach for automated generation of exploded views that did not rely on extensive object information, see Figure 30.
3.1.5 Rendering Multiple Volumes

In the medical field, it is very beneficial to acquire information using multiple sources to help in medical diagnosis. However, the integration of multiple datasets into a unified 3D volume is nontrivial. The difficulty lies in how the intersecting datasets are stored in texture memory as well as how they are sampled in the raycasting process. Several techniques have been developed to find suitable strategies for integrating characteristics from multiple datasets [111-114]. Each of these techniques explore different ways of combining overlapping voxel data such as different data intermixing levels (e.g. accumulation level, illumination level, image level) as well as fusion tables where multiple properties are stored in different color channels of the 3D texture. The key differences between these techniques lie in how the volumes are combined. Manssour et al. [115] took advantage of imaging technology strengths and used an MRI volume to define the opacity transfer function while using a Positron Emission Tomography (PET) volume for the color transfer function.

The previous techniques helped build a strong foundation for storing multiple volume data, but most implementations for rendering multiple volumes used texture
slicing [116-120] due to the fact it is much easier to implement than raycasting. For multi-volume slice-based rendering, each volume is sliced as is done for view-aligned single volume rendering. The slices are then depth sorted on a shared slice stack. Finally, the slice stack is rendered in back-to-front order and blended into the framebuffer, see Figure 31. Plate et al. [121] combined bricking, octrees, depth-peeling and texture slicing to improve performance.

Figure 31: Multi-volume rendering by independently slicing each volume and depth sorting the slices into a slice stack.

To generate the highest quality multi-volume rendering, raycasting needs to be implemented instead of texture slicing. Beyer et al. [63] created a GPU-based raycasting technique to support multiple volumes, segmentation masks and view-dependent clipping and rendering modes for neurosurgical applications as shown in Figure 32. Another technique uses a combination of depth peeling [122] and dynamic shader generation to perform multi-volume rendering [104, 123]. Due to recent trends indicating that graphics programming is rapidly moving away from fixed function approaches [124], certain techniques have been built on top of the Compute Unified Device Architecture (CUDA) [125, 126]. These techniques use CUDA to exploit a sort-middle approach where volume rendering is performed using
This approach can render more than 50 arbitrarily overlapping volumes on current graphics hardware and still achieve interactive framerates.

### 3.1.6 Other Advancements

There have also been other advancements made in GPU-based volume raycasting including volume scattering [128-131], Monte-Carlo volume rendering [132-135], multiple GPU raycasting [136-140] and client/server volume rendering [141-145]. Each of these methods are not applicable to this dissertation, but are noted for presenting a full literature review on all the advances in volume rendering. These advances have brought many new possibilities to all areas able to harness to power of volume rendering. Unfortunately, almost all of these techniques still have their limitations.

### 3.2 Volume Rendering APIs

The advances in GPU-based volume raycasting in the areas of performance and visualization have been quite significant in recent years pushing the boundaries of what can be done with volume rendering. All of these advanced techniques have

**Figure 32:** Combination of multiple datasets using multiple rendering modes. From left to right: pre-integration with illumination, transparent isosurfaces, pre-integration with one clipping plane and a corresponding 2D slice.
been published, but very few are available to the general public as open source volume rendering APIs or available software. Most of today’s volume rendering APIs still rely on texture slicing to perform volume rendering. For the few APIs that truly support GPU-based volume raycasting, none have been designed and developed for any platform other than a desktop with high-end commodity graphics hardware. The following sections discuss the currently available volume rendering APIs broken down into the following categories: desktop APIs, mobile device APIs and immersive virtual reality APIs. The section is concluded with a description of popular volume rendering commercial and open source applications.

3.2.1 Desktop APIs

The open source volume rendering APIs currently available come with a wide range of functionality and complexity. Two basic volume renderers are SIM Voleon [146], an add-on library to Coin3D [147] and eVolve [148], built directly on top of OpenGL. Both of these volume rendering APIs support 2D and 3D texture slicing for object-aligned and viewport-aligned slices rendered using back-to-front compositing. Each API supports opacity and color transfer functions as well as bricking for GPU texture memory optimizations.

Another open source volume rendering engine used by thousands of researchers around the world is the Visualization Toolkit (VTK) [149]. VTK contains many of the same functionalities of SIM Voleon and eVolve such as opacity and color transfer functions as well as multi-threaded CPU texture slicing, orthogonal and oblique clipping planes and multiple volume rendering. VTK also recently merged
the GPU-based volume raycasting library VTKEdge [150] to provide a GPU-based volume raycasting solution to its community. However, the GPU raycasting is currently limited to only NVIDIA graphics cards and has significant issues with oblique clipping planes.

ImageVis3D [151] is a much more advanced volume rendering engine that supports multiple rendering modes such as 1D and 2D transfer functions, isosurface rendering and specialized modes such as maximum-intensity projection (MIP) and slice views. ImageVis3D provides multiple rendering GPU-based implementations such as object-aligned and viewport-aligned texture slicing as well as volume raycasting. ImageVis3D supports orthogonal clipping and lighting in addition to optimizations such as bricking and multi-resolution textures to improve performance. Unfortunately, the volume rendering API for ImageVis3D is available, but is only built to support the ImageVis3D application. It is not constructed in a typical open source manner with sample applications and full documentation nor supported by a large open source community.

The most robust and full-featured open source volume rendering API available is Voreen [152]. It supports direct volume rendering (DVR), isosurface rendering, MIP rendering, Phong and tone shading illumination models, multimodal datasets, time-varying and segmented datasets, 1D and 2D opacity and color transfer functions, axis aligned clipping planes and preprocessing capabilities such as volume cropping and gradient calculations. This volume rendering API is available under the GNU General Public License (GPL) v2 and is designed for academic research purposes.
Three of these volume rendering APIs (VTK, ImageVis3D and Voreen) fully support GPU-based volume raycasting. However, even these advanced volume rendering APIs still fall short in certain areas. VTK has issues with clipping for GPU-based volume raycasting and does not support any more complex features. ImageVis3D offers no documentation for implementing their volume rendering core into another application and is not widely supported by any open source community. The final and most important shortcoming of all these volume rendering APIs is that they are only designed for single workstation desktop computers. There is no provided support for mobile devices or immersive virtual reality environments. These APIs are very complex and would very difficult to migrate to additional platforms.

### 3.2.2 Immersive Virtual Reality APIs

Immersive virtual reality systems face unique challenges such as application data serialization, device and display abstraction, renderer integration, synchronization (frame-locking) and cluster performance and overhead [153]. Due to these already daunting challenges, coupling immersive virtual reality systems with the performance challenges of volume rendering is a difficult task. Therefore, it is not surprising that the availability of open source immersive virtual reality volume rendering APIs is very limited.

Several immersive VR volume rendering solutions have been implemented over the past two decades [154-158], but only three remain that are still under active development. The first is VFIVE or the Vector Field Interactive Visualization Environment [159-161]. VFIVE was designed to visualize and analyze complicated
three-dimensional data such as flow velocities, isosurfaces, field lines, tubes and ribbons in CAVE [162] environments. The VFIVE rendering core is built on top of OpenGL and was recently expanded to support slice-based volume rendering. Stereoscopic viewing and cluster configuration are provided by CAVELib [163]. Although the source code for VFIVE is available in a limited form, the project is not an open source project with an active community.

Another immersive VR volume rendering API is FlowVR [164-167]. FlowVR is a hierarchical component oriented middleware for enabling high performance executions on parallel architectures such as clustered immersive virtual reality systems. FlowVR synchronizes rendering by transmitting graphics primitives and their rendering parameters to render network traffic between cluster nodes. Common rendering libraries can modify low-level drawing routines to use FlowVR Render objects instead of OpenGL to take advantage of FlowVR clusterization methods. VTK FlowVR is an example of such an implementation. By combining the functionality of FlowVR and VTK FlowVR, an immersive VR volume rendering application can be constructed.

Equalizer [168, 169] is a middleware API designed to handle OpenGL multi-node rendering and synchronization for high-performance visualization. It is well supported by the open source community, has built-in support for volume rendering using the eVolve API, can render both active and passive stereo and has integrated support for tracking systems. Applications built on the Equalizer framework can run unmodified on any visualization system ranging from a small workstation to a large-scale immersive virtual reality system.
All three of these APIs support immersive virtual reality volume rendering, but none have support for advanced GPU-based volume raycasting. VFIVE is not an open source project, and both VFIVE and Equalizer only support slice-based volume rendering. FlowVR indirectly supports GPU-based volume raycasting, but requires a modified version of VTK that supports FlowVR Render objects. In addition, VTK GPU-based raycasting is quite limited and does not provide advanced capabilities. In summary, there are currently not any immersive virtual reality APIs that support advanced GPU-based volume raycasting.

3.2.3 Mobile Device APIs

Previously, native volume rendering on mobile devices was simply not possible due to hardware limitations. However, the hardware of today’s mobile devices has increased significantly and is now powerful enough to support volume rendering. Unfortunately, volume rendering APIs for mobile devices do not exist. Several applications have been developed for volume rendering on the iOS platform, but the underlying volume rendering code used to build the applications is not available to the open source community. The reason a volume rendering mobile device solution has not been created is most likely because the required hardware for volume rendering on mobile devices has only very recently become powerful enough.

3.2.4 Commercial and Open Source Volume Rendering Applications

Various commercial and open source volume rendering applications have been developed to assist the medical profession. Popular commercial desktop
applications for visualizing volumetric medical data are Amira [170], Vitrea [171] and Fovia [172]. These applications are built on top of proprietary volume rendering APIs and contain many advanced features such as multimodal dataset rendering, lighting and shadowing, and GPU-based raycasting. There are also open source desktop application alternatives to the commercial products including OsiriX [173], VolView [174], ImageVis3D [151] and VoreenVE [152]. Except for VoreenVE, these software products contain less functionality than their commercial alternatives, and have much less sophisticated user interfaces.

Immersive virtual reality volume rendering solutions have also been commercialized. The Visualization Sciences Group (VSG) have developed extensions to their Open Inventor software development kit (SDK) known as VolumeViz [175] and ScaleViz [176] for rendering large volumetric datasets in immersive virtual reality environments. The Avizo [177] line of software products is built upon VolumeViz and ScaleViz to provide commercial solutions for visualizing, manipulating and understanding scientific and industrial volumetric datasets. VRVis [178] is another company that specializes in immersive virtual reality volume rendering applications for industrial partners.

Commercial volume rendering applications for mobile devices are very limited. A search for volume rendering applications on the Android App Market resulted in zero actual volume rendering applications. On the iOS platform, there are only two applications available, ImageVis3D Mobile [179] and OsiriX HD [180]. ImageVis3D Mobile is most likely built on top of the ImageVis3D rendering core while OsiriX HD is probably built on top of a custom port of VTK to support the iOS
platform. The developers have not made this information available. Regardless, even if the volume rendering APIs used to build these applications were available, one might want to think twice before doing so. The comments on these applications are quite negative. Each of them seem to crash often and most comments claim both applications are unusable.

A final human anatomy application is Grays Anatomy Premium Edition for iPad [181] complete with full interactive illustrations for anatomical exploration. The application also includes seven models in their new 3D mode which are most likely supported by surface rendering. These three applications are proof that volume rendering is on its way to mobile devices. However, there has yet to be a mobile device application that has been accepted as a viable option for performing volume rendering investigation.

3.3 Research Issues

Based on the literature review of volume raycasting, advanced GPU-based volume raycasting and volume rendering APIs, three research issues have been identified. They are:

1. **To design and construct a unified GPU-based volume rendering raycasting engine to support multiple platforms including desktops, laptops and immersive virtual reality systems on multiple operating systems.**

   Volume rendering development should not have to be performed in a custom manner for each computing platform. A unified volume rendering
engine would provide developers with a global solution for volume rendering on multiple platforms. Thus, researchers and developers would only need to familiarize themselves with a single volume rendering solution to create and deploy applications on multiple platforms.

2. To study methods to develop GPU-based volume raycasting for mobile devices supported by the iOS platform.

Mobile devices are quickly finding their way into hospitals and clinics around the world. Doctors at these facilities are using these devices to examine X-rays, write prescriptions and take notes during patient visits. These devices carry patient medical histories, triage information, allergy data and allow doctors to order treatment while they’re still with the patient. With the addition of a volume rendering solution for mobile devices, doctors could additionally use these devices for explaining ailments and anatomy to patients, collaboratively review diagnoses with other physicians and even use them for surgical planning.

3. To create a bridge between volume rendering APIs, multiple platforms and theoretical academic research.

It is common knowledge that many volume rendering APIs exist today. Except for Voreen, advanced volume rendering research performed by the academic community is not made publicly available. Additionally, Voreen only supports high-end desktop devices. By providing an open source volume rendering engine that supports multiple platforms natively,
researchers can use the engine as a bridge between academic research and open source and industrial contributions.
4 METHODOLOGY

To construct a volume rendering engine, there were many challenges and architectural design decisions that needed to be considered. The following is an initial list of requirements for the engine:

1. Must be cross-platform supporting Windows, Mac OS X and Linux
2. Must be stable
3. Must render efficiently due to the complexity of volume rendering
4. Must support desktops, laptops and immersive systems and mobile devices
5. Must encapsulate volume rendering platform customization at the engine level

Based on these requirements, the first decision that needed to be made was to choose which low-level rendering API would support the engine, DirectX or OpenGL. Since DirectX is not supported by multiple operating systems or platforms, OpenGL was chosen. OpenGL is a very stable API implemented in the C language as a state machine, thus allowing it to render very efficiently. As an API, OpenGL supports all the same platforms required of the engine through either native OpenGL or OpenGL for Embedded Systems (OpenGL ES). In order to encapsulate platform customization, the custom volume rendering code for each platform needed to be abstracted from the application level and handled inside the engine directly. To handle this type of platform encapsulation, several open source APIs were used.
4.1 Developing the Rendering Core Foundation

Before selecting open source APIs to the support critical components of the engine, the following stipulations were imposed to ensure the engine requirements were still maintained:

1. Must support free and proprietary licensing terms (LGPL, BSD, MIT, etc.)
2. Must be cross-platform supporting Windows, Mac OS X and Linux
3. Must have a large, active community of users
4. Must have been around for more than 5 years

First, all APIs needed to be released under licenses supporting both free and commercial software to allow researchers and developers to incorporate the engine into their projects. Requiring each API to support the same platforms as the engine was necessary to ensure certain APIs did not limit the scope of the engine. The final two stipulations were meant to ensure the quality of the APIs. Open source APIs with large user communities often produce the most stable and reliable codebases. Based on these stipulations, three different open source APIs were identified to support the volume rendering engine.

4.1.1 OpenSceneGraph

OpenSceneGraph (OSG) [182] is an open source, cross-platform graphics toolkit for the development of high-performance graphics applications released under the OpenSceneGraph Public License (similar to LGPL). It provides an object-oriented framework on top of OpenGL offering enhancements in performance, scalability, portability and productivity. OSG supports high performance rendering
through view-frustum culling, occlusion culling and OpenGL Shader Language and display lists which are critical to the volume rendering pipeline. Various geometry formats can also be imported directly into OSG through a dynamic plugin mechanism (osgDB) allowing intricate models such as trocars and scalpels to be rendered alongside a volumetric dataset. The rendering core of OSG is independent of the windowing system, making it easy for users to add their own window-specific libraries for various platforms such as desktops, immersive systems and mobile devices. After thirteen years of development, the user community has grown to over 2,000 users and developers who actively contribute to the development and testing of OSG. Based on all of these features, OSG was an ideal API to handle the low-level rendering of the volume rendering engine.

### 4.1.2 DCMTK

The DICOM Toolkit (DCMTK) [183] is a collection of libraries and applications implementing a large majority of the Digital Imaging and Communications in Medicine (DICOM) standard released under the BSD license. DICOM is a medical imaging standard format enabling the storage of both medical image information and pertinent patient's information into a single file for easy exchange of medical information. DCMTK is capable of examining, constructing and converting DICOM image files as well as sending and receiving images over a network connection. The DICOM library is fully cross-platform supporting Windows, Mac OS X and Linux operating systems among others. Development of the DCMTK API began in 1995, and has been under active development ever since. The DCMTK library has a large
user community and will serve as the DICOM volume loader for the volume rendering engine.

4.1.3 VR Juggler

VR Juggler [184, 185] is a cross-platform, open source virtual reality software development environment designed specifically for creating and executing immersive applications. The virtual platform of VR Juggler supports display and device abstraction allowing applications to be compiled once, and run on multiple configurations with no code changes. Multiple rendering APIs, including OSG, are able to synchronize data between each cluster node using the application data serialization mechanism. Synchronization between frames is handled by the swap barrier which ensures all cluster nodes swap their back and front buffers simultaneously. Each of these features is critical in ensuring the adequate performance in cluster configurations. VR Juggler was established in 1997 as a cross-platform API released under the LGPL license. Additionally, VR Juggler is still one of the fastest cluster synchronization APIs available today [186]. With all these features and the native support for OSG, VR Juggler was chosen to support the volume rendering engine on immersive virtual reality systems.

Once the underlying APIs for the volume rendering engine were determined, the next step was to implement the volume raycasting algorithm into sandbox applications on each platform to investigate the specific design and implementation characteristics required of each platform. Since the desktop platform presented the
smallest amount of known challenges, the desktop sandbox application was the first one developed.

4.2 The Desktop Sandbox Application

4.2.1 Architecture

The desktop application served as the initial development sandbox for constructing the volume raycasting core functionality. In Figure 33, a diagram of the software architecture of the sandbox application can be seen. The DICOM Toolkit (DCMTK) was used to load various DICOM dataset files, gather the necessary parameters pertaining to the volume, extract the intensity values from each DICOM slice and load the values into memory. The low-level volume rendering was built directly on top of OSG. User interface elements and the windowing system were provided by Qt. Rendering an OSG scenegraph in a Qt widget was handled by the QOSGWidget interface.

4.2.2 Features

The first feature built into the sandbox application was the ability to extract all the necessary information from a volumetric dataset. Using DCMTK, the DICOM data, slice resolution, rescale slope, rescale intercept, pixel spacing, slice thickness and slice location are extracted from each DICOM file. With this information, the sandbox application adjusts and reformats all the voxels in every DICOM slice, sorts them into front-to-back order and constructs the final 1D array of volumetric data used in the volume raycasting algorithm. By abstracting the volumetric data
reformatting from the user, the sandbox application hides the complexity internally which is a feature that no other volume rendering APIs provide.

Once the volumetric data is loaded into memory, the next step of process is to construct the volume bounding box geometry consisting of six quadrilaterals, or quads, encapsulating the volume as seen in Figure 34. Matching the dimensions of the bounding box to the dimensions of the volumetric dataset is the easiest approach, however this is usually inaccurate because voxels are rarely spaced equally in all three dimensions. The actual voxel spacing is defined by the pixel spacing and slice thickness extracted from the DICOM data by DCMTK. The pixel spacing creates a 3D mapping between the actual sampled dimensions and voxel dimensions. As a result, the volume geometry typically needs to be scaled along all three axes to correlate with the voxel mapping. These scale adjustments are necessary, but create sampling issues in the fragment shader. The issue is that there

Figure 33: Architecture diagram of the desktop sandbox application.
is no longer a one-to-one mapping in the fragment shader between the ray sample location and the appropriate interpolated voxel value at that location. The locations have been scaled. Therefore, the fragment shader needs to account for the inverse scale in the three axes to extract the appropriate voxel values from the 3D volume texture.

Constructing the volume geometry is the last calculation performed by the CPU. The geometry undergoes rasterization, where each geometric polygon is mapped to a pixel, or fragment, and sent to the fragment shader. The fragment shader receives either the entry or exit intersection point between the ray and volume, depending on whether the camera position is inside the volume. If the camera position is outside of the volume, the entry intersection point is precomputed.
during rasterization and received in the fragment shader. If this is the case, the exit point of the ray still needs to be computed before compositing can begin.

There are two ways of handling the exit point. The first way involves computing the ray direction, then continuing to step along the ray until the ray exits the volume. Whether it has exited the volume is computed at every step along the ray. This is the method typically used in volume rendering APIs today. The alternative to this approach is to precompute the exit point of the ray before starting the traversal. This can be done using Smits [187] ray-box intersection algorithm originally designed for raytracing. However, this algorithm produced numerical problems for rays with slopes near zero along any axis producing artifacts in these locations. Williams et al. [188] later refined Smits algorithm to properly handle the numerical instabilities eliminating the artifacts at volume borders. The fragment shader in the sandbox application uses Williams et al. version of the ray-box intersection algorithm to precompute the exit point of the ray. By precomputing the exit point, raycasting is executed faster than checking during every iteration of the resampling process if the ray has exited the volume.

The second way of calculating the exit point is performed when the ray starts inside the volume, which occurs when the camera is located inside the volume. In this particular case, no extra computation is required. The fragment shader receives the exit point location instead of the entry point. The entry point location of the ray can be computed as the location of the fragment at the camera position. In order for the fragment shader to know whether the camera is located inside the volume, it must be notified from the main application. Therefore, before the rendering process
of the frame begins, the operation determining whether the camera is inside the volume records the flag and passes it off to the fragment shader through a uniform boolean.

Computing the entry and exit points of the ray-volume intersection is all that is required to traverse the ray. At this point, the sample application supports three different types of rendering for volume raycasting: compositing, maximum intensity projection (MIP) and minimum intensity projection (MinIP). Compositing involves sampling the intensity then opacity at each sample point. If the voxel is not fully transparent, the color is computed and the sampled voxel’s color and opacity are accumulated by the global ray accumulation voxel. This process is continued until the ray exits the volume. The global ray accumulation voxel is then set as the fragment color for that fragment in the framebuffer. Since the sandbox application uses front-to-back compositing, it also is able to take advantage of early ray termination. For more details on compositing and early ray termination, please refer to sections 2.3.7 and 3.1.1 respectively.

The other two rendering techniques, MIP and MinIP, are designed for visualizing more specific aspects of the volumetric data and are computed in a similar way manner. First, the entire ray is traversed looking for either the minimum or maximum intensity value. Then, either the minimum (MinIP) or maximum (MIP) intensity value is rendered as the fragment color. MIP rendering can be used to visualize pulmonary nodules in the lungs while MinIP rendering can aid in visualizing the internal lung structure. Each of these rendering techniques (compositing, MIP, MinIP) offer unique visualizations of the same dataset providing physicians with
additional tools for volume exploration. An example of each of the three rendering techniques can be seen in Figure 35.

Figure 35: Examples of different volume rendering techniques supported by the desktop sandbox application including compositing (Top Left), MIP (Top Right) and MinIP (Bottom).

The next important feature of the sandbox application is the support for 1D opacity transfer functions. The opacity transfer function is a mapping between opacity and the full range of voxel intensity values for the volumetric dataset. The actual transfer function can be modified using many different techniques including linear blending, normal distributions, b-spline interpolation and even stochastic techniques [24]. The function values are then extracted at regular intervals into a 1D texture, generally with a 256 or 512 pixel resolution, and loaded onto the GPU for
processing by the fragment shader. The 1D opacity texture defines the opacity values for all intensity values for each sampled voxel. When determining the opacity of at a sampled voxel location, first the intensity is computed, then mapped to an opacity using the opacity texture. If the voxel opacity is larger than zero, it is accumulated. Today’s volume rendering APIs use the opacity texture to define the full range of voxel intensities. This is not an ideal approach for defining opacity when performing interactive windowing.

Interactive windowing is the process of specifying a minimum and maximum range of voxel intensities to investigate and visualize. Typically, the voxels outside the window range are not rendered at all. For example, a user may wish to examine the bone structure of a dataset where bone intensity values range from 1000 to 2000 with a global voxel range of −2000 to 3000. These windowing parameters result in a normalized focus range of voxel intensities from 0.6 to 0.8. Currently, volume rendering APIs handle modifications to the intensity range by rebuilding the entire opacity texture and loading it back onto the GPU each time window parameters are modified. The actual transfer function would be interpolated between the range of 0.6 to 0.8 instead of between 0.0 to 1.0.

The sandbox application handles interactive windowing in a much more efficient manner. There is no need to rebuild the opacity texture when performing windowing. Instead, the minimum and maximum windowing parameters can be stored directly in the fragment shader. When computing the opacity during compositing, intensities below the minimum windowing parameter are mapped to the first value in the opacity texture while intensities above the maximum windowing
parameter are mapped to the last value in the opacity texture. Thus the opacity texture is never required to be rebuilt when the windowing parameters are modified. This can be somewhat limiting though because users may want to control the voxel intensities outside the active windowing area independently, instead of setting them to the lower and upper bound values of the opacity texture. In this case, the sandbox application allows the user to override this default behavior by manually specifying the opacity values for voxel intensities below and above the active windowing area. This technique results in higher performance than current volumes rendering APIs that rebuild the entire opacity texture each frame.

The sandbox application also supports preset color transfer functions including many common coloring schemes including Bone, Cardiac, GE, Grayscale, Muscle and Bone, NIH, Red Vessels and Stern. Each of these coloring schemes use varying color channel functions to enhance different visual characteristics in various parts of the volumetric dataset. Each time the sandbox application opacity or color transfer functions are modified, the opacity and color textures are reloaded onto the GPU and updated in the fragment shader. Examples of these coloring schemes can be seen in Figure 36.

To improve the visualization quality when rendering close up views of the volume, trilinear interpolation was implemented in the sandbox application. This is quite easy to implement in code as the only OpenGL requirement necessary to perform trilinear interpolation on 3D textures is to pass the LINEAR flag to the texture during initialization. OpenGL will automatically perform trilinear interpolation when sampling the 3D texture in the fragment shader. Almost all volume rendering APIs
use trilinear interpolation for sampling the 3D volume texture because the computation can be performed directly in hardware offering a tremendous improvement in rendering quality with very little performance overhead. The difference in quality between nearest neighbor and trilinear interpolation can be seen in Figure 37. For more information regarding additional interpolation techniques, please refer to Section 2.3.3.

The final feature supported by the desktop sandbox application is clipping. Most volume rendering APIs support up to six orthogonal clipping planes except for VTK. VTK supports up to six orthogonal and oblique clipping planes. In the sandbox application, a custom algorithm was designed to support an infinite number of
Figure 37: A close up view of a chest cavity using nearest neighbor interpolation (Top). The same close up view using trilinear interpolation (Bottom).

orthogonal and oblique clipping planes. This would, in theory, allow a developer to
use enough clipping planes to render a volume as a sphere using a large number of adequately positioned clipping planes.

The sandbox application uses a CPU-based iterative approach when clipping the bounded volume. Each clipping plane is defined by a single point in 3D space and a clipping normal. After the clipping planes have been defined, the clipping algorithm can clip the volume geometry with each clipping plane. The algorithm uses the following steps to clip the volume geometry with each clipping plane: compute the intersection points between the volume face edges and the clipping plane,

Figure 38: Demonstration of the clipping process. At first, the front clipping plane is positioned at the volume boundary. Next, the front clipping plane clips a portion of the front of the volume. Then, the top clipping plane clips a top portion of the volume. Finally, the right clipping plane is positioned to clip the right portion of the volume. This process is repeated each time a clipping plane is updated.
rebuild all the partially clipped faces with the new clipping intersection points, remove the fully clipped faces and cap the clipped portion of the volume with a new face. This process continues until all clipping planes have had a chance to clip to the volume geometry. A demonstration of this process can be seen in Figure 38.

The sandbox application also implements lazy clipping which saves significant computation time as it only recomputes the volume geometry when clipping planes are updated. In summary, the clipping algorithm supports an unlimited number of clipping planes, but it is, however, realistically capped by intra-frame computation time. This means that only so many clipping planes can be active at a time before performance becomes an issue due to the overhead of performing the clipping operation.

4.2.3 User Interface

The user interface for interacting with the volume rendering controls in the desktop sandbox application was built using Qt. It consists of a single inspector widget that supported four different tabs (General, Coloring, Windowing and Clipping). The general tab controls features such as render quality, raycasting technique, bounding box rendering and background color. The coloring widget is very simple and allows a user to select the active color table for rendering the volume. The windowing widget controls the opacity transfer function as well as the real-time windowing controls. The most complicated and intelligent widget is the clipping widget. It controls all the logic for clipping including whether clipping is enabled, the active clipping plane, all the position and rotation controls for the active
Figure 39: The general widget (Top-Left). The coloring widget (Top-Right). The windowing widget (Bottom-Left). The clipping widget (Bottom-Right).
clipping plane and a way to reset the clipping planes. Each time the active clipping plane is modified, all the widget states are updated to represent the current state of the new active plane which can be very different from the previous state. This widget also uses a much more polished separation structure with group boxes to help make action discovery a bit more clear. Examples of all four of the inspector widgets can be seen in Figure 39.

For clarification purposes, this user interface was designed to be merely a proof-of-concept. There was little development time and no planning time spent on trying to build a useful, intuitive and professional looking user interface. It was thrown together quickly to make it easier to debug the volume rendering logic. This is, however, only the case for the desktop application. The development cycles of the other sandbox applications dedicated considerable amounts of time to breaking down use cases, generating mockups and spending additional time adding a polished look-and-feel.

4.2.4 Challenges and Contributions

The development of the desktop sandbox application certainly presented some difficult challenges along the way. The first was the computation of the exit points of the rays in the fragment shader. This was challenging because debugging equations in fragment shaders can only be done by modifying the color of the rendered fragment. Stepping through the shader logic in a debugger is simply not possible. Another issue that arose was depth sorting. By default, OSG does not provide proper depth sorting for scenegraph nodes with enabled alpha blending.
Therefore, it was impossible to get the bounding box and the clipping planes to render with the proper depth at all times. Most volume rendering libraries today have the same exact problems with proper depth rendering with alpha blending. Unfortunately, this was never properly solved in the desktop application, but future sections of the dissertation will provide more detail about this particular issue.

The final major challenge in the development of the desktop sandbox application was clipping. The development of the clipping algorithm was challenging since it is difficult to eliminate rounding errors while trying to create perfect geometrical face intersections. The algorithm took several iterations before it was working properly.

These difficult challenges led to some very unique contributions which deserve recognition. The first of which is real-time windowing directly built into the fragment shader. This allows opacity and color tables to be manipulated dynamically with zero overhead. Almost all other volume rendering APIs need to rebuild the opacity and color table textures while the desktop application simply modified a uniform in the fragment shader. A second major success is the fact that the desktop volume raycasting logic works on all modern graphics cards. It is not limited to only Nvidia or ATI cards. All the rendering is tied directly to the OpenGL specification and not to any company specific extensions.

The largest contribution of the desktop application is certainly the clipping algorithm. Other volume rendering APIs require the use of binary clip volumes to provide high-fidelity clipping. Unfortunately this approach requires an additional check in the fragment shader for every sample point along every ray for every rendered frame. This drastically increases the amount of fragment operations
necessary to clip a volume. Instead, the desktop application uses a lazily computed CPU-based algorithm that requires no additional fragment operations. This is a major performance improvement when compared to previous clipping plane implementations used in other volume rendering APIs.

4.3 The Immersive Sandbox Application

4.3.1 Architecture

The immersive sandbox application development began after the completion of the desktop sandbox application to investigate the multi-platform capabilities of the current engine design. The goal was to implement the same features into the immersive application to determine the complexities of producing a volume rendering engine capable of abstracting the volume rendering code from the platform. The system architecture for the immersive application was very similar to

Figure 40: Architecture diagram of the immersive sandbox application.
the desktop application and can be seen in Figure 40. The only real difference was that the Qt user interface API was replaced by VR Juggler. Additionally the QOSGWidget was replaced by the VR Juggler OSG App class which integrated OSG rendering into the VR Juggler DrawManager as well as the VR juggler windowing system.

4.3.2 Features

The development of the immersive application went very quickly since the original architectural design held up in all cases. The tight coupling of OSG and VR Juggler proved useful and effective for performing volume raycasting in real-time in large cluster environments. There were several small hurdles encountered along the way on the VR Juggler side, but those will be described in more detail in Section 4.3.4 or the Challenges and Contributions section.

As for the functionality in the immersive application, it contains the same features as the desktop application. These include DICOM data extraction and reformatting, a custom shader implementation of ray/volume entry and exit intersections and three different types of rendering: compositing, maximum intensity projection (MIP) and minimum intensity projection (MinIP). Additional features include color transfer functions with several presets, custom opacity transfer functions with real-time manipulation, trilinear interpolation sampling and a custom algorithm for supporting an unlimited number of orthogonal and oblique clipping planes. Several examples of different datasets and configurations can be seen below in Figure 41.
There is very little custom development to cover for the immersive application because the volume rendering logic from the desktop application reused and recycled with no code modifications. The development mostly consisted of porting the Qt portions of the desktop application to VR Juggler. Thus, the architectural design of the volume rendering engine was a multi-platform success. By using a combination of OpenSceneGraph and VR Juggler, desktops, laptops and immersive virtual reality systems are able to rely on the same core volume rendering code to perform volume rendering on these different platforms.

**Figure 41:** Several screenshots of the immersive sandbox application.
4.3.3 User Interface and Interaction

A very unique part of the immersive sandbox application is the user interface and the navigation model used to explore the volumetric dataset. To really understand the design philosophy behind the user interface for the immersive application, the challenges of creating virtual reality user interfaces must first be examined. User interface design in an immersive environment is difficult for several reasons. The first is that it needs to augment the virtual environment. Any interface embedded in a virtual environment is immediately distracting and presents a difficult challenge of not impeding on the primary goal of dataset exploration. A second challenge is interacting with the user interface. One can choose to use a secondary display device to control a user interface which sends the commands to the immersive application such as an iPad or a laptop, but requires the user to focus on the secondary device and user interface when making manipulations. This context switching is less than ideal. However, embedding the user interface inside the virtual environment requires a way for the user to manipulate the user interface with either a wand, joystick, or gamepad devices to list a few.

Another major challenge of working in a virtual reality environment is how to effectively navigate the virtual scene. There are many ways to provide this interface through 2D controls on a laptop interface (poor immersive experience), using devices with gyroscopes to control the acceleration and direction (fluid but not precise), or even wands and gamepad devices. The two most common methods of navigation in immersive virtual reality applications are the wand and the gamepad. Wand navigation is very good supporting six degrees of freedom movements, but
can be challenging to perform precise location selection for menu navigation. Additionally, wands generally come with less controls such as buttons and joysticks. Gamepads on the other hand also control six degrees of freedom motions with a wider array of functionality. User interface selection does not require point selection, but can be traversed through joystick or D-pad keys. Finally, there is a much higher likelihood of a user having prior experience with a gamepads when compared to wands due to the widespread adoption of gamepads for video game consoles.

Unfortunately, none of these user interfaces or navigation schemes is truly ideal for volume rendering, but they can still be sorted by effectiveness. To provide minimalistic context switching, the interface itself needs to be embedded within the virtual environment. By making this restriction, it was easy to select a navigation device. Since navigating a user interface in a virtual environment with a wand can be challenging to users, the natural choice was to instead use a gamepad. Once these design decisions were finalized, the next step was to create a way to display a nonintrusive user interface to the user that was controlled using a gamepad.

When coupling OSG and VR Juggler, the support for embedded user interfaces is quite limited. There is a new experimental library, osgQt, that attempts to render Qt widgets directly in the OSG scenegraph. Regrettably, it is not yet robust enough to be used in mainstream applications. The only other support is located in the osgWidget library, which is mainly designed for mouse and keyboard interaction in 2D interfaces. Due to these limitations, the user interface rendering and interaction would need to be constructed using a custom solution.
Since there was no prebuilt support for the user interface, there were no restrictions on how the user interface needed to be rendered or manipulated. The starting point was to create rendered widgets that could easily be turned on or off when necessary as the user interface is quite intrusive in the immersive environment. It also needed to be rendered on top of the virtual environment to ensure it was always visible and not occluded by the virtual scene. Other requirements included semitransparent widgets to not fully occlude the volume when enabled, quick and easy navigation to keep the learning curve low and a sharp professional look-and-feel to the widget design and theme. These stipulations resulted in the four widgets seen in Figure 42.

There were many steps to designing the final version of the interface seen in Figure 42. The first step was to use Adobe Illustrator mockup the look-and-feel of all the interaction widgets which included buttons, checkboxes, combo boxes, sliders and even double sliders. Next was to design each of the four controller widgets (Rendering, Coloring, Windowing and Clipping). Once the mockups were complete, then began the daunting task of attempting to replicate the exact look of the mockups generated by Illustrator in OSG. Unfortunately, after building a system in OSG for compositing dynamic text objects, border lines, backdrop quads and rounded polygons with gradient shading, it became apparent that the same look-and-feel could not be produced with OSG without hundreds of man hours invested. Additionally, the end goal was to create a user interface for the immersive sandbox application, not an open source user interface library for OSG. These complications led to a less robust but still quite effective solution.
Rather than dynamically rendering the user interface at runtime, what if the controller widget itself was simply an image that had been pre-rendered by an

Figure 42: Each of the four custom widgets used for the user interface in the immersive sandbox application. Several screenshots of the immersive sandbox application.
external application—i.e. Adobe Illustrator? This would eliminate the need for doing any custom rendering for the user interface other than rendering the image to a textured quad that supported alpha blending. The downside was each controller widget would have to have a prebuilt image for each possible interaction widget combination. For example, if there were two checkboxes, there would need to be four controller widget images to represent all the widget combinations. Additionally, if the controller widget changed, then all the images would need to be re-rendered to include the new change. The sliders presented a different problem because they were continuous which meant they had an infinite number of states. The only way to avoid the infinite state slider issue would be to not render them in the controller widget images, but to render them separately.

To get the best looking interface and interaction scheme with the least amount of development time, the pre-rendered controller widget approach was chosen for the final user interface in the immersive sandbox application. Each of the controller widget states were all designed and rendered using Adobe Illustrator. The images were then rescaled to power-of-two (POT) dimensions to optimize the speed in which the GPU could cycle them to and from the available texture memory. Most graphics cards only support POT textures and require rescaling the textures before loading them into the GPU memory. This rescaling can cause a significant rendering lag and it is often best to start off with POT textures in the first place. These textures could then be cycled at 60 fps with no rendering lag. The final task was to develop an interaction scheme with the pre-rendered images and the sliders to control with the gamepad.
The number one goal for designing the gamepad interaction was to use as few buttons as possible to control all the states, animations and visibility of the interaction and controller widgets at all times. With the joysticks already being used for navigation in the virtual scene as well as the left/right triggers being used to control navigation speed, the D-pad and 2 and 3 buttons were dedicated to controlling all user interface interactions. A schematic of all the gamepad controls used to control the immersive application can be seen below in Figure 43.

![Figure 43: A schematic of the gamepad controls used to control the immersive sandbox application.](image)

The most simplistic approach to navigating the user interface seemed to be to use a directed acyclic graph (DAG) approach. The user interface would start in a hidden state that was the root, or first level, of the DAG. The second step was to show the user interface which would be the second level of the DAG. This level
would contain all four of the controller widgets. The next level would step into the controller widget allowing a user to navigate to the interaction widget to be manipulated. The final level would then allow the user to directly manipulate the interaction widget. Traversing downwards through the DAG or deeper into the user interface was assigned to button 2 on the gamepad. Button 3 was assigned to traverse backwards through the DAG to step out of the user interface and eventually hide it. Navigation as well as interaction widget manipulation was then controlled using the D-pad.

To keep the user interface as un-intrusive as possible, only a single controller widget is displayed at a time. To navigate to a different controller widget, a user selects the title, then uses the left/right D-pad keys to cycle to the next widget. To add some extra polish, animations usher the exiting controller widget out using a custom fade out animation while the entering controller widget uses a contrasting fade in animation. In addition to the fading of the widgets, they are also slid left or right while they are fading. The active controller widget also fades in and out when the user interface is shown and hidden. These animations give the user interface a very fluid look-and-feel which was one of the original design goals.

In summary, it was not necessary to design such a complicated and visually appealing user interface for the immersive application. All that was really needed were a few simple controls to modify some of shader settings in the renderer. This could easily be done with a few keyboard shortcuts for the application. So then why all the extra effort? The answer lies in overall effectiveness of any software application. Sadly, users tend to notice the negatives in any software, where the best
features tend to be those which are hidden and function without thought. Since the rendering quality was already quite high, the user interface was designed to be a complimenting feature that would not impede or downgrade the volume rendering experience.

### 4.3.4 Challenges and Contributions

There were several challenges encountered throughout the development cycle of the immersive application. The first was moving the camera around the volume. VR Juggler applications coupled with OSG are recommended to move the scenegraph nodes and allow VR Juggler to control all the camera settings. This approach does not work with the current volume rendering design. The volume rendering requires the volume to stay at the origin while the camera is moved to navigate around the scene. To accomplish this, some of the draw functionality of VR Juggler had to be modified.

The next challenge came when the application reached a point where it was able to render volumes in immersive clustered environments. The best way to describe the issue was that the volume was “wiggling”. After some extensive investigation, the reason for the wiggle was due to the non-thread-safe parallel rendering of VR Juggler in multi-pipe configurations. Each pipe was rendering concurrently but sharing the same uniform in the fragment shader defining the camera view matrix. Since each pipe has a unique camera view matrix, the concurrent rendering was causing certain frames to use the other pipe’s camera view matrix. This produced the unique effect of causing the volume to wiggle when
the wrong camera view matrix was used. To temporarily solve the issue, the parallel rendering was disabled using a mutex to serialize the rendering. This is not ideal though because of the performance hit taken by eliminating parallel rendering. Two possible permanent solutions to this issue would be to extend the shader to support multiple camera view matrices or to use two unique shader programs so the uniforms would be unique.

A final issue is that all the GPU render commands are queued on all the graphics cards in the cluster. This is fine in non-cluster computing since the swap buffers command forces the GPU to execute all the commands in the queue then swap the front and back framebuffers. The downside is in the way VR Juggler tries to swap buffers. Since the swap buffers command is sent out to all nodes in the cluster at once, it is expected that all the nodes will swap the buffers when receiving the command. What this really does is tell the GPU to execute the queue of render commands on all nodes in the cluster at the same time. The problem is that the queue is so large when performing volume rendering, that almost none of the cluster nodes finish the render at the same time. With only milliseconds of difference, this produces a tearing effect in the cluster-based rendering. To ensure the buffers are swapped at precisely the same time, the GPU queue needs to be empty at the time of receiving the swap command. This would allow the buffers to be swapped immediately, thus eliminating the tearing. To ensure the GPU queue is empty, the draw command needs to implement a `glFinish()` after completion to force GPU synchronization before sending the cluster swap buffers command from the master node.
Even though some of these challenges were quite difficult to solve, it is important to note that none of them were volume rendering specific. They were all shortcomings or small design flaws in VR Juggler specific implementations. The volume rendering engine design held up very well throughout the development of the immersive application. This allowed the immersive sandbox application rendering implementation to be developed much quicker than the desktop sandbox application.

There are two main contributions based on the development of the immersive sandbox application. The first is a very custom user interface design using pre-rendered images coupled with a directed acyclic graph approach to navigation. This produces high quality widgets with a short development time. The second and possibly largest contribution of all the work in this dissertation is that according to the literature review, the immersive sandbox application is the first immersive, clustered, GPU volume raycasting application of its kind. All the other cited works use many different approaches, libraries, cluster rendering APIs and OpenGL serialization techniques to perform volume rendering, but none of them used GPU-based volume raycasting to do it. They all used some form of orthogonal or view-aligned texture slicing. This is exciting and very promising because there is still many ways to improve performance and rendering quality as will be discussed in later sections.

4.4 The Mobile Sandbox Application

4.4.1 Architecture

Performing GPU-based volume raycasting on mobile devices is a challenging task to pursue due to hardware limitations of the current generation of devices.
Currently, the most stable and reliable platform for mobile device application development is Apple’s iOS and Cocoa Touch SDK. The well designed development tools coupled with widespread adoption and dominance of the platform make it an ideal candidate for investigating volume raycasting on mobile devices. Additionally, OSG contains bindings to the latest OpenGL ES 2.0 spec to make it possible to embed an OSG scenegraph inside a iOS application.

The mobile sandbox application development began shortly after the completion of the immersive sandbox application with two major items to investigate. The first was how well mobile devices could perform under the high computational load of volume rendering. The second item was to try to determine if the volume rendering engine would be capable of abstracting the volume rendering code from the mobile platform. For this application, it was determined that an iPad 2 was the best hardware to develop with. It had the most computing power of any of the iOS devices at the time. The original system architecture for the application was quite similar to that of the desktop sandbox application and can be seen in Figure 44. This diagram is very similar to the other two architectures, except it relies on iOS and Cocoa Touch for the windowing system and user interface. It is also depends on the GraphicsWindowIOS interface in the osgViewer library that enables rendering an OSG scenegraph directly inside Cocoa Touch.

After an extensive investigation of the GraphicsWindowIOS API, it was determined that this coupling between OSG and Cocoa Touch was not robust or stable enough to support volume rendering. There were assumptions made in the development of the GraphicsWindowIOS interface block the developer from
customizing several critical features for volume rendering including the OpenGL context as well as touch interactions with gesture recognizers. These assumptions were made to provide a complete abstraction between OSG and Cocoa Touch. The class also breaks the Model/View/Controller (MVC) design pattern on the Cocoa Touch side. To better understand these issues and limitations, the issue needs to be discussed in more detail.

Cocoa Touch heavily relies on the MVC paradigm for designing user interfaces. All views should generally have a view controller which handles creation and destruction of the view, as well as interactions with the rest of the application or even the device. The view takes care of drawing all of its own internal content as well as laying out all of its child views. The view also contains the model which it is entrusted to draw. For OpenGL views, the model refers to the OpenGL context of that view. The Cocoa Touch application designer (Xcode 4) allows for quick drag and
drop construction of the views and subviews of the user interface of an application. The view controller class is used to hook up all interactions between the application, other view controllers and the specific view it manages.

Now armed with this background in MVC pattern design, let’s explore why the GraphicsWindowIOS class was not a suitable alternative. The GraphicsWindowIOS class hides from a developer the fact that the view controller and the view even exist. This is because OSG wants to keep a consistent API between all platforms (Qt, Cocoa, Cocoa Touch, .NET, etc.). Unfortunately, this cripples the functionality of an OSG widget in Cocoa Touch in several important ways. First, there is no way to attach gesture recognizers or additional views on top of the scenegraph view such as buttons, sliders, etc. Second, the OSG view cannot be integrated into the Cocoa Touch application designer to embed in other views. Finally, the class does not allow for customization of the framebuffer settings. Assumptions are made about what settings the majority of users would want, and these settings are not fully exposed to the end user. All of these major limitations led to the development of a new custom integration of OSG and Cocoa Touch built specifically for volume rendering.

Rather than hiding the view controller and view behind the OSG abstraction, the exact opposite approach was taken. The new coupling between OSG and Cocoa Touch involved three different classes: VIPREViewController, VIPREEAGLView and VIPRERenderer. This can be seen below in Figure 45. The VIPREViewController was used to handle all the gesture recognizers as well as the view management. The VIPREEAGLView was used to create a standard UIView with a CAEAGLLayer underneath. A CAEAGLLayer is what handles rendering an OpenGL context for a
UIView. The VIPREEAGLView also manages creation and destruction of its framebuffer and can be manipulated in the application designer to be used with other views. The VIPRERenderer is what handles all the manipulation of the OpenGL context rendered by the CAEAGLLayer. To do this, an osgViewer::Viewer instance is used that is created as an embedded viewer. This means that all OSG is responsible for are the internal event traversals and draw traversals. It no longer has to worry about activating the OpenGL context, creating framebuffers or even swapping buffers. These are all controlled by the VIPRERenderer.

![Diagram](https://via.placeholder.com/150)

**Figure 45:** The modified architecture using native iOS view management instead of the internal GraphicsWindowIOS implementation from OSG.

This customization was necessary for many reasons and the results were outstanding. There are many more details about how the VIPRERenderer works to perform volume rendering, but that will be discussed in more details in sections to
come. The important thing to keep in mind about this sandbox application is that every part of the coupling between OSG and Cocoa Touch is completely custom. This customization is something that many developers in the future can benefit from, even those not using volume rendering.

4.4.2 Raycasting Complications

Once the custom classes between Cocoa Touch and OSG were completed, the next logical step was to attempt to get volumes rendering in the scenegraph using GPU-based volume raycasting. The initial approach was to try to use the same design for volume rendering as was used in the two previous sandbox applications. That design included using a 3D texture to store the voxel data along with 1D lookup tables for opacity and color in the fragment shader. Fortunately, this approach is fully supported by the OpenGL ES 2.0 specification. Unfortunately, there are almost no mobile devices on the market today that support 3D textures. The specification is only a set of guidelines. It is up to the hardware manufacturer to decide what parts of the specification they wish to support and implement. Along with not supporting 3D textures, the iOS implementation of the OpenGL ES 2.0 specification also does not support 1D textures. Therefore, the entire data structure design for the fragment shader used in the previous sandbox applications would not work in the mobile sandbox application.

The next approach was to use a large number of 2D textures to store all the voxel data to perform raycasting. To do this, multiple slices of voxel data would need to be stored in a single 2D texture. For example, the largest 2D supported texture on
an iPad 2 is 4096 x 4096 which was found by querying the \texttt{GL\_MAX\_TEXTURE\_SIZE} of the OpenGL context. Therefore, if one slice of voxel data is 512 x 512, then a single texture could accommodate 64 slices of voxel data. Unfortunately, another limitation of the iPad 2 is that it only supports the instantiation of eight 2D textures in a fragment shader at a single time. This would cap the total amount of voxel slices to 512. This seemed like a reasonable limitation given that most medical datasets contain between 200-500 slices of data.

In theory, raycasting using 2D textures seemed to at least be possible. This led to the development of a raycasting implementation using 2D textures. In summary, it worked. In reality, it did not work fast enough. Stacking the voxel data side-by-side in a single texture was straightforward, but extracting the voxel data out of the texture became cumbersome in the fragment shader. To extract a single intensity value out of the texture, the pixel coordinates first needed to be transformed into voxel coordinates. This is the same process used by the desktop and immersive sandbox applications. However, since the data is not stored in a 3D texture, the voxel coordinates then need to be transformed into the 2D stacked texture coordinates. This transformation process is quite slow and adds many additional fragment operations which would not be necessary using 3D textures. On top of this, there are two significant problems with interpolation. The first problem is that all the data at the borders of each slice is incorrect due to interpolation. Each slice needs to be padded to get accurate values from interpolation at the slice borders. In order to pad the slices accurately, at least a one voxel border must be placed around each slice. This greatly reduces the number of slices that can then fit into a single 2D
texture. With slices now being 514 x 514, the total number of slices that fit into a 2D texture is 49 which only allows for 392 total slices to be rendered in a single fragment shader.

In the end the texture memory limitation proved to not even be an issue due to the second interpolation problem. The second problem was that 2D textures do not provide 3D interpolation. As a result, volumes will only be accurately interpolated for two of the six orthogonal views. The other four will suffer from nearest neighbor interpolation. One could certainly implement neighbor voxel sampling and interpolate the voxel value correctly, but there is simply not enough fragment operations available for interpolation in the fragment shader. Again this did not prove to be the limiting factor either.

The reason raycasting does not work currently on an iPad 2 is that there are simply not enough fragment operations available in software to raycast in real-time. Transforming the pixel position to a voxel position to then the mapped position in one of the eight 2D textures is too costly to be performed in real-time. An implementation of raycasting using the stacked 2D texture approach with only 64 slices was only reaching about 1 frame per second. With this poor performance, it was deemed impossible with the current iPad 2 OpenGL ES 2.0 hardware support to perform GPU-based volume raycasting in real-time. Development of this approach and investigation was discontinued because it is not yet possible with the current hardware.

The exciting part about this failure is that even though raycasting is not possible right now, it probably will be soon. There are currently three different
Android devices on the market today that do support 3D textures. Now that does not mean that they would be able to do volume raycasting, but it does mean that the capabilities are on the horizon. If the mobile device hardware capabilities continue to grow as they have historically grown, with a few more iterations, volume raycasting will absolutely be possible using the same approach the other two sandbox applications took.

Now the investigation of volume rendering on mobile devices could have concluded with the fact that volume raycasting cannot be performed in real-time on mobile devices. Instead, the development continued with an approach more suitable to the current state of today’s hardware, orthogonal texture slicing (Section 2.4.4). View-based texture slicing was not considered due to the lack of 3D hardware interpolation of 2D texture data. There again would have been too many fragment operations required to render in real-time. Orthogonal texture slicing does not require a conversion of pixel coordinates to voxel coordinates. It also only requires one 2D texture to be stored in the fragment shader for the slice. Since this approach eliminated a large portion of the fragment operations which made the other approaches not possible, it moved forward into the next stage of development.

To accomplish real-time orthogonal texture slicing on an iPad 2, there were still many performance issues to overcome. These included memory bandwidth, GPU fragment operations and GPU asynchronous processing and synchronization. The following sections will describe each of these in more detail.
4.4.3 Memory Limitations

The process of performing orthogonal texture slicing requires a number of quad polygons to be stacked in order creating a rectilinear prism equal to the size of the volume. The most accurate way to represent the voxels is to create a slice for each voxel section of the dataset. For example, for a 512 x 512 x 256 sized dataset, the most accurate representation of this data would be to render 256 slices with dimensions of 512 x 512 in the z-direction, 512 slices with dimensions of 512 x 256 in the y-direction, and 512 slices with dimensions of 256 x 512 in the x-direction.

Representing the voxel data in this fashion requires three unique sets of textures, one for each axis. Technically six are required, but only three are necessary because the texture order is simply reversed in opposite directions of the same axis. Creating these sets of textures requires a few steps. The first step is to read in all the voxel data from each DICOM image. The second step is to normalize all the voxel data to the 0–255 range (must be a single byte in OpenGL ES 2.0) and construct a 1D array that represents the entire 3D dataset. Finally, each of the three axes’ textures are extrapolated from the 3D dataset. This process has a very large memory footprint and can very quickly exceed the amount of memory an iPad 2 application is allowed to have. Any iOS application is only allowed a certain amount of memory before being automatically terminated by iOS due to excessive memory usage. This required an in-depth investigation of the memory management schemes to keep the application from being closed by the operating system at runtime.

The first step was to free memory along the way that was no longer necessary. For example, for a 512 x 512 x 512 dataset, it was never possible to load
all the DICOM images, then construct the 3D dataset without the application being
terminated for exceeding the allocated memory. However, if the DICOM image data
was released when the data had been transferred into the 3D dataset, then the
application was allowed to continue.

The next step was to build the texture sets for each axis. Unfortunately, there
is no way to remove any of the 3D dataset data until all three texture sets have been
created. It is not possible to allocate each of the three texture sets for 512 x 512 x
512 voxels. Each time the final texture set is allocated the application is terminated
prematurely. The only way to stay under the allowed memory footprint for the
application at this point is to downsample the image data for each texture. Since iOS
does not allow non power-of-two (NPOT) textures, the image data must be scaled
down to at least 256 x 256 before being allocated into the texture. By scaling the
data down, the memory capacity is not exceeded and the application is allowed to
continue.

4.4.4 GPU Fragment Operation Bandwidth

After the three sets of textures for each axis were constructed, three different
sets of quads were built for each axis to render the textures. Each of the sets of
quads was placed under two different parent switches: a positive switch with all
quads in the default order and negative switch where all the quads were in the
reverse order. A switch is a scenegraph node capable of quickly enabling/disabling
all child nodes. By storing all six sets of quads in a manner where they could quickly
be enabled or disabled, this allowed the visible quads to be quickly flipped whenever
the orthogonal view closest to the camera’s viewpoint would change. It was also noticed that the positive xyz switches all need to be enabled on the first rendered frame to cache all the textures on the GPU. Otherwise, the first time the view direction switches, there will be a significant lag when caching the textures on the GPU.

With the mechanism in place for dynamically cycling the visible quads to the closest orthogonal view, basic volume rendering was fully implemented. With the addition of opacity and color transfer function lookups, the application was only able to render approximately 1 frame per second (fps). This was far more promising than raycasting, but still was nowhere near real-time.

The only way at this point to possibly speed up the rendering process was to simply render less while interacting with the volume. The application was entirely GPU bound because there was not enough processing power available to render 60 fps. From this point, it was determined there were two main ways to improve the rendering speed during interaction. The first way was to render a smaller viewport and use OpenGL to upscale the result. The second way would be to render less quads and turn up the opacity contribution linearly with the number of quads removed. Both methods would result in much higher frame rates, but lower quality renders. When not interacting with the volume, the full quality volume would be rendered to provide the highest quality image possible. This was a sacrifice that had to be made to produce a real-time interactive volume renderer with today’s hardware.
Rendering a smaller viewport meant that the rendering process needed to be split into multiple cameras. The idea was to render a much smaller viewport in a pre-render camera, then use the main camera to render a single quad that was pinned to the bounds of the main camera’s viewport. The main camera quad renders the resulting texture from the pre-render camera. It would use OpenGL’s bilinear hardware interpolation to upscale the much smaller image to fit the viewport of the main camera. The way this can be accomplished is by attaching the pre-render camera to a framebuffer object. A framebuffer object is generally used when

Figure 46: Screenshot of the mobile sandbox application with the pre-render camera texture displayed on top of the upscaled render.
rendering needs to be done off-screen. Then the resulting texture of the framebuffer object can be attached to the fragment shader of the main camera quad. The main camera quad then samples the texture for each pixel and uses bilinear interpolation to upscale. An example of this can be seen in Figure 46.

Depending on the size of the dataset being rendered, different resolutions of the pre-render camera resulted in much different framerates. In Figure 47 below, the five supported pre-render resolutions can be seen. These resolutions are 64, 128, 256, 512 pixels and 703 pixels. As is expected, the higher the resolution, the better the quality of the render. Something to keep in mind is that the aspect ratio of the pre-render camera viewport must match that of the main camera viewport. Otherwise the upscaling aspect ratio would not be 1:1 and would result in additional inaccuracies.

The second way of speeding up the rendering was to render less quads. If there are less quads being rendered, there are significantly less GPU fragment operations taking place. To render less quads, an additional level of switches was required for each negative and positive axis switch. For example, the positive z-axis switch would contain five child switches, each containing a different number of quads. The maximum number of quads for each level was 32, 64, 128, 256, and 512 respectively. If a dataset contained 512 slices, then the positive z-axis child switches would contain exactly the preset number of quads for each level. If the dataset only contained say 355 slices, the z-axis child switches would contain 32, 64, 128, 256, and 355 quads. For examples of the other four lower quality switches, see Figure 48.
Figure 47: A 64 pixel low resolution render of the Cardiac-CT dataset (Top-Left). A 128 pixel low-medium resolution render (Top-Right). A 256 pixel medium resolution render (Middle-Left). A 512 pixel medium-high resolution render (Middle-Right). A full resolution render at 703 pixels (Bottom).
Figure 48: A 32 slice low sampling render of the Cardiac-CT dataset (Top-Left). A 64 slice low-medium sampling render (Top-Right). A 128 slice medium sampling render (Middle-Left). A 256 slice medium-high sampling render (Middle-Right). A 355 slice high sampling render (Bottom).
It is easy to see how the quality changes with the number of quads being rendered. What is not shown in those images is the fact that all the resulting renders have exactly the same opacity as the others, even though they are rendering a significantly different number of quads. This is due to the opacity correction put into the fragment shader. For example, if four semitransparent gray quads were rendered on top of one another, the resulting composited quad would be black and opaque. However, if only one of the quads were rendered, the resulting quad would still be gray and semitransparent.

The same effect happens when removing slices from the volume. If less quads are being rendered, the opacity of each quad needs to be increased to keep the composited volume render roughly the same opacity. If the opacity correction is not used, the rendered volume will be much more transparent. Therefore, the opacity contribution of a quad in the high quality render is 1.0 whereas the opacity contribution of a quad in the medium-high quality render is 4.0. This is assuming of course that the medium-high quality render contains \( \frac{1}{4} \) of the amount of quads as its high quality counterpart. This simple correction in the fragment shader allows the volume to be rendered at the same opacity even when significantly changing the amount of quads representing the volume.

Both pre-render resolutions and volume quad quality play an important role in improving the speed at which the volume can be rendered. Individually, neither approach solves the issue of being able to render volumes interactively at 60 fps because the quality is too poor. The benefit is when they are used in combination with each other. Table 2 breaks down three different sized datasets and the
Table 2: A breakdown of the rendering performance when using different combinations of resolution and sampling rate for three different sized datasets.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Quality</th>
<th>Cardiac (201 slices)</th>
<th>Cardiac-CT (355 slices)</th>
<th>Manix (460 slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>60 fps</td>
<td>60 fps</td>
<td>60 fps</td>
</tr>
<tr>
<td>Low</td>
<td>Low - Medium</td>
<td>60 fps</td>
<td>60 fps</td>
<td>60 fps</td>
</tr>
<tr>
<td>Low</td>
<td>Medium</td>
<td>60 fps</td>
<td>55 fps</td>
<td>50 fps</td>
</tr>
<tr>
<td>Low</td>
<td>Medium - High</td>
<td>55 fps</td>
<td>50 fps</td>
<td>40 fps</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>45 fps</td>
<td>45 fps</td>
<td>30 fps</td>
</tr>
<tr>
<td>Low - Medium</td>
<td>Low</td>
<td>60 fps</td>
<td>60 fps</td>
<td>60 fps</td>
</tr>
<tr>
<td>Low - Medium</td>
<td>Low - Medium</td>
<td>60 fps</td>
<td>60 fps</td>
<td>60 fps</td>
</tr>
<tr>
<td>Low - Medium</td>
<td>Medium</td>
<td>50 fps</td>
<td>55 fps</td>
<td>48 fps</td>
</tr>
<tr>
<td>Low - Medium</td>
<td>Medium - High</td>
<td>42 fps</td>
<td>45 fps</td>
<td>35 fps</td>
</tr>
<tr>
<td>Low - Medium</td>
<td>High</td>
<td>30 fps</td>
<td>30 fps</td>
<td>25 fps</td>
</tr>
<tr>
<td>Medium</td>
<td>Low</td>
<td>60 fps</td>
<td>60 fps</td>
<td>60 fps</td>
</tr>
<tr>
<td>Medium</td>
<td>Low - Medium</td>
<td>60 fps</td>
<td>60 fps</td>
<td>55 fps</td>
</tr>
<tr>
<td>Medium</td>
<td>Medium</td>
<td>45 fps</td>
<td>50 fps</td>
<td>45 fps</td>
</tr>
<tr>
<td>Medium</td>
<td>Medium - High</td>
<td>30 fps</td>
<td>30 fps</td>
<td>28 fps</td>
</tr>
<tr>
<td>Medium</td>
<td>High</td>
<td>20 fps</td>
<td>15 fps</td>
<td>10 fps</td>
</tr>
<tr>
<td>Medium - High</td>
<td>Low</td>
<td>58 fps</td>
<td>55 fps</td>
<td>55 fps</td>
</tr>
<tr>
<td>Medium - High</td>
<td>Low - Medium</td>
<td>45 fps</td>
<td>45 fps</td>
<td>40 fps</td>
</tr>
<tr>
<td>Medium - High</td>
<td>Medium</td>
<td>30 fps</td>
<td>30 fps</td>
<td>20 fps</td>
</tr>
<tr>
<td>Medium - High</td>
<td>Medium - High</td>
<td>10 fps</td>
<td>10 fps</td>
<td>10 fps</td>
</tr>
<tr>
<td>Medium - High</td>
<td>High</td>
<td>4 fps</td>
<td>4 fps</td>
<td>3 fps</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
<td>50 fps</td>
<td>50 fps</td>
<td>40 fps</td>
</tr>
<tr>
<td>High</td>
<td>Low - Medium</td>
<td>35 fps</td>
<td>35 fps</td>
<td>30 fps</td>
</tr>
<tr>
<td>High</td>
<td>Medium</td>
<td>20 fps</td>
<td>20 fps</td>
<td>15 fps</td>
</tr>
<tr>
<td>High</td>
<td>Medium - High</td>
<td>5 fps</td>
<td>5 fps</td>
<td>2 fps</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>1 fps</td>
<td>1 fps</td>
<td>1 fps</td>
</tr>
</tbody>
</table>
Figure 49: A comparison of the medium-medium and full quality renders of the Cardiac dataset to show they are almost exactly the same despite the performance enhancements (Top). A comparison of the med-med and full quality renders of the Cardiac-CT dataset (Middle). A comparison of the med-med and full quality renders of the Manix dataset (Bottom).
performance metrics for all possible combinations of pre-render resolution and sampling quality. The important information has been highlighted in red and green. The red lines represent the medium resolution used in combination with the high sampling rate as well as the medium sampling rate used in combination with the high resolution. The resulting image quality with these combinations of settings is still very high, but the performance is quite low with framerates ranging from 10-20 fps. The green line represents the combination of medium resolution and medium sampling rates resulting in framerates ranging from 45-50 fps as well as high image quality.

Even for very large datasets, interactive framerates can be achieved by using a combination of medium resolution and medium sampling rates. Through much trial-and-error, these combinations result in the highest quality images with the most interactive framerates. To help aid the data in Table 2, Figure 49 places the medium resolution, medium sampling rate interactive render next to the full quality render for the Cardiac, Cardiac-CT and Manix datasets. These comparisons make it clear that the render quality is still quite high even when both the resolution and sampling rate performance optimizations are in use. For small datasets, higher resolutions and sampling rates can be used and still achieve interactive framerates.

4.4.5 GPU Asynchronous Processing and Synchronization

Even with all this optimization in place to reach real-time framerates when interacting with the volume, there was still a major limitation for the end user. They had to wait for the high quality render to complete before being able to interact with
the volume again. This resulted in a frustrating experience for the user as investigating the volume is not a stop-and-go procedure. For example, while changing the minimum and maximum slider values for real-time windowing, the user would have to keep moving the slider continuously. If they paused even for a moment, the high resolution render would begin and real-time windowing would no longer update. This would happen often because it is very difficult to move sliders every so slightly on iOS. Another example was when a user was navigating around the volume, as soon as the navigation was paused, the high resolution render would begin. This would cause the user to have to wait until it completed to interact with the volume again. Every continuous volume rendering operation was subject to this stop-and-go behavior. This flat out made the mobile sandbox application unusable. As a result, fixing this limitation became the final focus for performance improvements in the mobile sandbox application.

Of all the platform specific issues encountered while developing this application, this particular one was the most challenging. To understand how the issue was solved, one must understand the complexity of the issue first. The main reason the high quality render could not be stopped once it was started lies in the design of the OSG render loop. OSG allows a user to set up the scenegraph exactly how they wish, then render it. Under normal circumstances, where the framerate should stay consistently high and continuous, this is the behavior expected of a scenegraph. However, this does not fit the paradigm of using low quality rendering in combination with high quality rendering where very different framerates are produced.
As an example, let's say a low quality render takes 20 milliseconds, and a high quality render takes 1 second. Now as soon as a user stops interacting with the volume (releases the touch), the high quality render begins and will take an entire second to complete. OSG will first run the event traversal, then the update traversal and finally the cull/draw traversal. Once it finishes the draw traversal, the buffers are swapped and the final image is displayed to the screen. This process is a black box that cannot be interrupted in any way. So in summary, once a frame starts rendering in OSG, it cannot be stopped until it completes the frame and swaps buffers.

Unfortunately, the issue gets even worse. All the draw operations from the OSG draw traversal are submitted to the GPU queue controlled by iOS. For high quality renders, this queue gets large very quickly. At a certain point, iOS flips an internal switch that says the GPU is behind, application execution on the CPU needs to be paused until the GPU can catch up. This was discovered through many hours of debugging the iOS application run loop. This results in all application events such as touch events or button presses to be delayed until the GPU queue is exhausted. What this boils down to is that even if the OSG render loop could be stopped or paused in some manner (which it cannot), iOS would not let a pause or stop event get executed until the GPU state and CPU state were fully synchronized. Now that the problem has been laid out in detail, let's discuss the solution.

There were three drastic modifications required to eliminate this issue. The first was to move all rendering off the main thread and onto worker threads so the main thread can continue to receive application events from iOS. The second was to modify OSG so the render loop could be paused or stopped. The third and most
critical modification was to keep the GPU queue small enough so iOS would not start blocking application events. All three of these modifications were required to stop the high quality render process before it completed.

Moving the rendering off the main thread and onto a worker thread was not a trivial task. First, the application needed to use lazy rendering to not waste necessary computing power. Lazy rendering refers to the idea of only rendering when necessary to save CPU cycles and battery life. If no changes to the scenegraph take place, then there is no reason to re-render the scene. The second requirement is to maintain a consistent draw rate when drawing is necessary. The final requirement is to be as efficient and minimalistic with threads as possible because the iPad 2 only contains two processing cores. Additional threads will only cause additional context switching on the CPU.

To implement lazy rendering, all events that modify the scenegraph in any way are submitted to the renderer queue. As long as there is an event in the queue, the renderer will render a low quality frame. Rendering a low quality frame pops the event from the render queue. After the queue has been exhausted, the high quality render begins. Upon completion of the high quality render, the renderer is paused until a new event is submitted to the queue.

To keep consistent draw execution when the renderer is running, a separate worker NSThread was instantiated. Attached to the run loop of this worker thread is an NSTimer that fires 300 times a second. A display link, a timer that is synchronized with the refresh rate of the display, is the desired way of running an asynchronous render loop on iOS, but the high quality rendering required a much faster update to
be effective. The iOS display link for an iPad 2 is only executed at 60 fps making it too slow to support the high quality rendering. The most efficient way to pass off the rendering commands to be rendered off the main thread without spinning up additional resources was to submit them to a Grand Central Dispatch (GCD) queue. Grand Central Dispatch is a technology developed to optimize multi-core operations using thread pools at the iOS level resulting in much higher performance than typical multi-threaded scenarios. This allows iOS to optimize its thread pool resources to execute the rendering commands in as few operations as possible.

Once the scene was being rendered lazily off the main thread, the next step was to modify OSG to be paused or stopped in the middle of the rendering process. This meant redesigning the OSG render loop at its core. Instead of traversing the entire scenegraph structure at once, it needed to be done incrementally. This required splitting the scenegraph into render chunks. A chunk is simply a portion of the scenegraph to be rendered. In orthogonal texture slicing, this refers to a small set of textured quads. Each chunk needed to be small enough that it could be rendered quickly (less than 10 ms). Through much trial-and-error, the ideal size of a render chunk on iOS was 16 quads. Once OSG had finished rendering the chunk, it would allow the application to process incoming application events. After the application events had been processed, the next chunk was rendered into the same buffer. This process would continue until all chunks had been rendered and the resulting buffer was swapped and displayed to the user.

After completing lazy rendering, asynchronous rendering and incremental rendering, it was expected that the high resolution render would be able to be
stopped mid-render. However, after all these optimizations, the application events were still not being delivered until the buffers were swapped. After several weeks debugging the render loop, the solution was stumbled upon by accident. The GPU was not executing any of the render commands until the buffers were being swapped. Even though the scenegraph was being rendered in chunks and the application events were allowed to come in through the main thread run loop, they were being delayed by iOS because the GPU queue had grown too large. The GPU was not being “forced” to render each chunk as it was submitted. OpenGL contains a function `glFinish()` that forces the CPU to wait until the GPU synchronizes its queue. It is widely regarded as a function to stay away from in almost all situations. However, it was designed for the exact purpose of forcing the CPU and GPU to stay in sync. After adding the `glFinish()` call to the end of each rendering chunk, the application events began immediately coming into the main thread while the high resolution render was running. When the renderer receives a new application event, the renderer finishes the current chunk, then exits the high resolution render loop, clears the framebuffer and begins rendering a low quality render. This was a monumental change to the mobile sandbox application architecture and resulted in immediately being able to cancel high quality renders to greatly improve the usability of the application.

### 4.4.6 Features

The mobile sandbox application contains almost all the same features as the desktop and immersive sandbox applications. The first shared feature is gradient
backgrounds. It supports three different gradient backgrounds to give a much
classier look and feel to the scenegraph. This is a technique used in almost every
modeling and animation software package today. Images of the three different
background gradients can be seen below in Figure 50. This was much more
complicated in the mobile application because the pre-render camera used to render
the gradient background needs to be attached to the pre-render camera for the low-
quality renders as well as to the high quality render. It is still implemented in the
same manner between sandbox applications.

Figure 50: Screenshots of each of the three custom background gradients
supported in the mobile sandbox application.

Due to the fact that the mobile application no longer uses raycasting, it is
limited to composite rendering. It is not able to perform MIP and MinIP renderings.
There is no possible way in the fragment shader to compute the minimum or
maximum voxel intensity along the ray. This is because there is no ray at all.
Orthogonal texture slicing relies on OpenGL to perform all the compositing in
hardware. It is widely accepted that orthogonal texture slicing is of much lower
quality than raycasting, but, due to hardware limitations, is the only method possible on the current generation of hardware.

The mobile sandbox application does however fully support all the same opacity transfer functions as the desktop application. These include linear and normal transfer functions as well as the sharpen option. What the sharpen option

Figure 51: The linear opacity transfer function (Top-Left). The linear opacity transfer function with sharpening (Top-Right). The normal opacity transfer function (Bottom-Left). The normal opacity transfer function with sharpening (Bottom-Right).
does is increase the slope of the transfer function to make voxels appear opaque more quickly. Figure 51 shows the differences between four combinations of opacity transfer functions.

In addition to opacity transfer functions, the mobile sandbox application also supports all eight color transfer functions. Both the opacity and color transfer functions could not be done in the same manner as the desktop and immersive applications. The reason for this was that the iPad 2 OpenGL ES 2.0 specification does not support 1D textures. However, as was mentioned earlier, the specification does support 2D textures. Consequently, the opacity and color transfer function textures were modified to 2D textures to be read correctly in the fragment shaders. Several examples of the different color transfer functions can be seen in Figure 52.

Two features which exist solely in the mobile sandbox application are preset views and multitouch gestures. The application contains six orthogonal and eight isometric preset views to help quickly navigate to a particular view direction of the volume. These are all labeled using medical terminology. In addition, the application supports four multitouch gestures for intuitive navigation around the volume. A single touch pan gesture recognizer is used to rotate the volume around the center trackball position. A double touch pan gesture recognizer is used to pan around the volume dataset. A pinch gesture recognizer is used to control the zoom level of the camera as well as pan while zooming. Finally, a double tap gesture recognizer is used to center the volume without changing the zoom level. These types of interactions are not possible with the default OSG and Cocoa Touch GraphicsWindowIOS implementation.
Another very unique feature of the mobile sandbox application is clipping. Clipping exists in the other two sandbox applications, but uses a very different algorithm. Clipping planes are not supported in OpenGL ES 2.0 through the `glClipPlane()` interface. Instead, clipping must occur at the shader level. Therefore, each of the six clipping planes are stored as `uniform vec4` values in the

![Image](image.png)

**Figure 52:** The “Muscle and Bone” color transfer function (Top-Left). The “Cardiac” color transfer function (Top-Right). The “Bone” color transfer function (Bottom-Left). The “Stern” color transfer function (Bottom-Right).
vertex shader. Inside the vertex shader, the dot product of the clipping plane normal and the vertex position are computed to determine whether the vertex is clipped off. Then in the fragment shader, if clipping is enabled, the vertex position is compared against all six clipping planes to see if it is clipped by any of them. If it is clipped, then it is discarded. Otherwise it is passed off to the compositing algorithm.

Rendering of the clipping planes is also done much better in the mobile application in comparison to the other sandbox applications. The clipping planes are bound to the constraints of the volume bounding box. This means that the intersections of the clipping plane and the volume bounding box are accurately computed, then rendered in a single plane regardless of the number of intersections with the bounding box. Two examples of this can be seen in Figure 53 below. This greatly improves the visual perception of the clipping planes in reference to the volume.

**Figure 53:** Two different examples of how the mobile sandbox application can accurately compute the intersection points with the volume bounding box.
To create a perfect plane through the volume that was capped at the volume bounds, a custom algorithm had to be developed. The first step was to compute the intersection points between the clipping plane and the bounding box edges using the following two equations:

\[ u = \frac{A \cdot x_1 + B \cdot y_1 + C \cdot z_1 + D}{A \cdot (x_1 - x_2) + B \cdot (y_1 - y_2) + C \cdot (z_1 - z_2)} \]  

(3)

\[ P = P_1 + (P_2 - P_1) \cdot u \]  

(4)

In Equation 3, \( ABCD \) represented the equation of the clipping plane. The points \( P_1 \) and \( P_2 \) represented the start and end points of the bounding box edge. If \( u \) was between 0.0 and 1.0, then Equation 4 was computed to find the intersection point between the clipping plane and the bounding box edge.

After the intersection points were computed, they needed to be sorted into the perimeter points if there were more than three points. If not properly sorted, the ends of the clipping planes would form a crisscross pattern over the plane instead of an encapsulating border. Several different approaches were taken to try to use angular sweeps to put the border points in order. Sadly, this technique was not suited to the task for several reasons. First of all, it was not able to handle the small edge cases where the floating point mathematical precision would break down. Additionally, the anchor point for the plane was not always guaranteed to be within the bounds of the clipping plane perimeter, thus causing very large discrepancies when this would occur.
The solution used was to create all the possible edge combinations from the border points, then eliminate those edges that were crossing. The total edge combinations for a given set of points $P$ was calculated using Equation 5:

$$E_n = (P_n - 1)!$$  (5)

For example, there are 6 different edge combinations given four points (Figure 54), ten edge combinations for five points, etc. To eliminate crossing edges, the first step was to create several vectors. If the first line segment consisted of $P_1$ and $P_2$, and the second line segment consisted of $P_3$ and $P_4$, then the following three vectors were created:

$$\vec{V}_1 = P_2 - P_1$$  (6)
$$\vec{V}_2 = P_3 - P_1$$  (7)
$$\vec{V}_3 = P_4 - P_1$$  (8)

After all three vectors were created, the two cross products, $D_1$ and $D_2$, were computed using Equations 9 and 10:

$$\vec{D}_1 = \vec{V}_1 \wedge \vec{V}_2$$  (9)
$$\vec{D}_2 = \vec{V}_1 \wedge \vec{V}_3$$  (10)

If $D_1$ and $D_2$ were in opposite directions, this meant that the end points of the second line segment were on different sides of the first line segment which implied that the two edges were crossing. This logic is demonstrated in Figure 54 for further clarification.
After the border edges were identified, they needed to be converted into a list of sorted border points. This was done by start end point matching between edges. Once all the border points were sorted, an OSG line loop was used to render the border while an OSG polygonal fill was used to render the plane.

Another very important feature built into the mobile sandbox application is the ability to perform proper depth sorting. This is a major advancement over the other applications because they are not able to do this properly. For example, proper depth sorting refers to all objects appearing in the proper depth order regardless of whether they are transparent or not. For example, a clipping plane that is behind the volume should appear directly behind the volume, then the volume should be

Figure 54: A diagram of the elimination method used to sort the clipping plane bounding box intersection points.
rendered on top of it. Without performing proper depth sorting before the rendering traversal, the depth appearance of clipping planes with the volume is incorrect. In Figure 55, the incorrect depth rendering from the desktop sandbox application can be seen in the left image while proper depth sorting from the mobile sandbox application can be seen in the right image.

![Figure 55: Screenshot of the incorrect desktop sandbox application clipping with non-depth sorted clipping planes and bounding box (Left). Screenshot of the mobile sandbox application with proper depth sorting (Right).](image)

To properly depth sort all the drawables within the scene, they must be rendered from back-to-front when using alpha blending. Some scenegraphs are capable of doing proper depth sorting while blending, unfortunately, OSG is not one of them. The only way to properly depth sort while using alpha blending in OSG is to do it dynamically within the scenegraph. To do this in the mobile sandbox application, there were several modifications that needed to be made to the scenegraph. First off, the bounding box was split into six different drawables with inward facing normals and attached to a pre-rendered `osg::Switch`. Next, the same
six drawables were also attached to a post-rendered `osg::Switch`. Therefore, the bounding planes were attached to the scenegraph twice. The same approach was taken with the clipping plane geometry. An example of the scenegraph structure that resulted can be seen below in Figure 56.

![Figure 56: A diagram of the scenegraph structured used to perform proper depth-sorted volume rendering.](image)

Once the scenegraph structure was in place, the application needed a mechanism to inform the scenegraph whether to render the pre or post-render bounding box and clipping planes. For polygonal planes this is simple because back-face culling can be used. However, for the mobile application, this only covers the clipping plane geometry. It does not cover the bounding box nor the clipping plane
border as back-face culling only works for polygons, not line loops. This led to the development of an internal mechanism for computing back-face culling for line loop planes as well as polygonal planes. By using Equation 11, it can be computed whether the plane is facing the camera regardless of the projection matrix in use:

$$u = (P - E) \cdot N$$  \hspace{1cm} (11)

In Equation 11, $P$ represents a border point on the plane while $N$ is the plane normal. The $E$ variable represents the camera eye position. If $u$ is positive, that means the plane is facing the camera. So each time the camera is now updated in the mobile application, each individual bounding box plane and clipping plane computes the value of $u$ and renders either the pre or post-rendered geometry depending on whether $u$ is positive or negative. This approach works well for bounding box and clipping planes, but is not robust enough to handle objects being rendered directly inside the volume.

The final custom feature of the mobile application is the ability to serialize datasets to quickly reload them. Due to the limited hardware capabilities of the iPad 2 as well as the fact that three different texture sets need to be generated for orthogonal texture slicing, it takes a long time to load and process the data. However, once the data has been processed, there’s no need to reprocess the same data again if it can be saved in an efficient manner. To help clarify, it takes almost three minutes to load the Cardiac-CT dataset which consists of 355 slices with a resolution of 512 x 512. This is a considerable amount of time to ask the user to wait before interacting with the volume.
To help address this issue, a serialization scheme was built to write all the textured quads directly out to a binary file. This was accomplished by leveraging the osgDB serialization mechanism built into OSG. With a very small amount of code and logic, the entire volume scenegraph can be written out to disk. Then using the same serialization mechanism, the binary data can be read back in, extracted and applied to an empty scenegraph in a much more efficient manner. For example, after implementing this serialization into the mobile application, reloading the already processed Cardiac-CT dataset has been reduced from three minutes to four seconds. Put another way, datasets can now be loaded approximately 45 times faster than before. This is major performance improvement which could greatly benefit the other sandbox applications as well.

4.4.7 User Interface

When designing the user interface for the mobile sandbox application, it was necessary to keep the most commonly used controls accessible to the user at all times. The reasoning behind this logic is that the exploration of a volumetric dataset requires many fine-grained adjustments to extract specific information. Users should not be burdened by cumbersome, context switching actions. As a result, the user interface was designed in a manner that encourages exploration by allowing quick access to the commonly used controls while always keeping the volume on-screen. This design philosophy was then coupled with a native iOS look-and-feel to give the user interface a minimalistic and intuitive behavior. For those who have already used an iPad, there is virtually no learning curve to using the mobile sandbox application.
The layout of the application was constructed using a UISplitViewController which creates a master view and detail view on the left and right respectively. For this application, the master view was used in combination with a UINavigationController to build a dynamic user interface while the detail view was used only to render the volume. The following sections break the user interface down into three major sections: the Inspector view, the Dataset view and the Clipping view.

**Figure 57:** The Inspector in the mobile sandbox application at launch (Left). The Inspector animating in all the widgets after the Yuria dataset was loaded (Middle). The Inspector after the animation completes (Right).
The Inspector view, or Inspector, is home to the most commonly used features such as selecting a dataset, adjusting the window settings and selecting a color table. It also contains additional widgets to quickly navigate the user to the less commonly used features such as preferences, preset views and clipping planes. Initially when the application launches, only the Dataset table cell is visible. This directs the user to the Dataset view to select a dataset to investigate. After the dataset is loaded, the additional table cells are animated into the view. Four different states of the Inspector can be seen below in Figure 57.

Some of the notable customizations added to the Inspector are the display of the current selection in the Dataset, Opacity TF and Color Table cells. The current selection is chosen in sub-views, but sent back to the Inspector to display the current state of the volume. The integration of UISwitch andUISlider views into certain table cells is a nice way to divide up user interface controls. Additionally, the minimum and maximum intensity sliders are not allowed to cross over each other. Although the look-and-feel of the Inspector seems like a basic implementation of stock user interface views, it is embedded within a custom UITableView and UINavigationController to provide a simple and intuitive navigation hierarchy. The approach was used to provide the maximum amount of information with as minimalistic user interface as possible.

The next view worth describing in detail is the Dataset view. The Dataset view uses a UITableView to display all the datasets preloaded into the application. It displays both the name of the dataset as well as a short description including the total number of slices as well as the physical area of the body the dataset contains.
When a dataset is selected, a UIActivityIndicatorView (spinner) is faded into the selected table cell. As the dataset loads into the scenegraph, the activity indicator continues to spin on the main thread. Once the dataset is loaded into the scenegraph, the activity indicator is quickly faded out and a checkmark is quickly faded in. This particular view is all about the small details. Several different examples of these animations can be seen below in Figure 58.

Figure 58: The Dataset view in the mobile sandbox application at launch (Left). The Dataset view while selecting a dataset (Middle-Left). The Dataset view after the progress indicator faded in and began spinning (Middle-Right). The Dataset view after the progress indicator faded out and the checkmark faded in after the dataset finished loading (Right).

The most complex view is the Clipping view. When loaded initially, it contains a lone Clipping table cell with a UISwitch for enabling the clipping functionality built into the application. When enabled, the rest of the table cells are animated into place. Several examples of the ClippingView can be seen below in Figure 59. There
are custom controls for everything from selecting the current clipping plane to rotating and positioning it in all directions. An UIActionSheet is used to reset either the active clipping plane or all clipping planes. The most interesting behavior of the Clipping view is when the Clipping table cell is disabled, the user interface is retracted back into the table cell. However, all the settings are still intact. If clipping is re-enabled, all the same clipping information is restored and the volume is again clipped as before. This is helps eliminate unnecessary options from the user and falls in line with the minimalistic design philosophy.

Figure 59: The Clipping view in the mobile sandbox application at launch (Left). The Clipping view after turn clipping on and the widgets all faded in (Middle-Left). The Clipping view after is has been used for a while (Middle-Right). The Clipping view after hitting the reset button (Right).
4.4.8 Challenges and Contributions

Of all the sandbox applications, the mobile application presented the largest amount of challenges as well as resulted in the largest number of unique contributions. Let’s first begin with all the interesting challenges that arose throughout development. Initially, it was difficult to even get iOS and OSG working together because of the shortcomings of the GraphicsWindowI0S implementation in OSG. The next major challenges were no 3D texture support and no 3D interpolation using 2D textures. These two challenges combined eliminated the possibility of performing raycasting on a mobile device in real-time with today’s hardware. After moving to orthogonal texture slicing, application memory limitations quickly became a problem when having to load three different texture sets simultaneously. Next up was the most complicated of all, canceling the high resolution render. The last two challenges were the lack of glClipPlane() support in OpenGL ES 2.0 and dataset loading times.

Even with this list of challenges, the mobile application was still completed. This is due to all the unique contributions invented throughout the development cycle. Before examining the contribution list, it should be noted that no library currently exists today to perform volume rendering on mobile devices. Thus, each feature in this application is in fact a substantial contribution to the community. With that said, let’s dig into the truly innovative contributions that make this application unique.

The first contributions involve dynamically modifying both the pre-render camera resolution and the textured quad sampling rate at runtime to increase the
interactivity of the rendering, producing a much more fluid interaction. The next major contributions are a highly optimized asynchronous iOS rendering mechanism for OSG applications, in addition to an incremental OSG render loop designed specifically for volume rendering. To perform clipping in OpenGL ES 2.0, a pure shader-based volume clipping algorithm was designed along with an algorithm for calculating the precise clipping plane geometry rendered at the bounding box intersections. A final contribution was the development of an internal back-face culling algorithm to perform depth sorted rendering using native OSG data structures.
5 VIPRE

The culmination of volume rendering research presented in this dissertation resulted in a multi-platform volume rendering solution known as the Volume Image Processing and Rendering Engine (VIPRE). VIPRE is designed in such a manner that it abstracts the volume rendering logic away from the platform. Essentially, to end users (developers), the same volume rendering methods exposed will work on all different platforms such as desktops, laptops, immersive clusters and mobile devices. To better understand how this abstraction was made possible, the following section describes the VIPRE architecture in depth.

5.1 VIPRE Architecture

Designing the VIPRE architecture was an incremental process. It began with the development of three sandbox applications to investigate whether a volume rendering platform abstraction was possible. Upon the completion of the sandbox applications, it became apparent that not only was such an abstraction possible, but could be done in a manner where platform specific implementations could be avoided at the library level. For example, the orthogonal texture slicing solution, designed specifically for the mobile application, also works very well in desktop and immersive environments. Rather than developing platform specific implementations behind the VIPRE libraries, the specific implementations were expanded to all platforms. The only situation where platform specific implementation could not be expanded was the coupling between user interfaces and the VIPRE rendering window. For this, example applications were developed to allow end developers to
customize the implementations, rather than bury them deep within the VIPRE libraries.

Since the rendering logic of all the sandbox applications was able to be extrapolated to all platforms, it made sense to split the library by rendering technique. Each technique has a common traversal interface, but the implementation of certain parts of the rendering are very different. Thus, VIPRE was split into six different unique libraries including vipre, vipreDICOM, vipreViewer, vipreRaycaster, vipreOTSlicer and vipreVATSlicer. An architectural diagram of

**Figure 60:** A generic architecture diagram for all platforms supported by VIPRE.

Since the rendering logic of all the sandbox applications was able to be extrapolated to all platforms, it made sense to split the library by rendering technique. Each technique has a common traversal interface, but the implementation of certain parts of the rendering are very different. Thus, VIPRE was split into six different unique libraries including vipre, vipreDICOM, vipreViewer, vipreRaycaster, vipreOTSlicer and vipreVATSlicer. An architectural diagram of
VIPRE can be seen below in Figure 60. The following sections will discuss what each of the VIPRE libraries are used for.

5.1.1 The VIPRE Library

The vipre library is a low-level support library that provides common functionality for the other libraries as well as a base mechanism for building new volume rendering technique implementations. A unique notification system was invented to send abstract notifications and objects in a type-safe manner without the need to couple the sending and receiving objects together. This is useful for making low-level scenegraph changes at the application level. For example, users can send notifications directly from their application to modify fragment shader uniforms without ever having the extract the uniform objects which are buried deep within the libraries.

The vipre library also contains many rendering classes to act as the building blocks for the various rendering techniques. Some examples of these rendering classes are pre and post-render cameras for multi-resolution rendering, background gradient cameras, color and opacity tables, shader controllers, trackball manipulators and a clipping plane renderer. Finally, the library provides a common rendering interface for designing a custom rendering technique library. This interface is exposed through the Volume and Renderer classes in the vipre library. Each technique is required to use the rendering interface to work together with all the other vipre libraries.
5.1.2 The vipreDICOM Library

The vipreDICOM library was designed as a bridge between DCMTK and OSG because the voxel data format of DCMTK is not the same as used for rendering by OSG. There were two main design goals put forth when creating the vipreDICOM library. The first was to provide a simple and intuitive way to build unique series objects and extract all the DICOM data from them. The second goal was to easily be able to extract voxel data and prepare it in an OSG friendly data structure for rendering. With these design goals in mind, Object, Slice and Series classes were built to extract all the DICOM header data and expose it in an intuitive manner to the user. A SeriesBuilder class was also designed to make it simple to build unique series objects from a list of files or even a directory. This portion of the vipreDICOM library satisfies the first development goal. To transform the voxel data into OSG data structures, several classes were built to handle all the different possible combinations of types of voxel data and convert them into data structures that can be used to create osg::Image objects. These are the interface OSG provides to get the voxel data into textures and onto the GPU for rendering. All of this functionality can be used with very few lines of code to eliminate the challenge of manipulating DICOM data.

5.1.3 The vipreViewer Library

The vipreViewer library is designed on top of the osgViewer library with the same end goal of rendering the scenegraph. However, the two have very different render loop implementations. The osgViewer library uses a render loop that first
executes all the events registered with the event queue, then calls all registered update callbacks, next traverses the scenegraph rendering each node it encounters and finally, swaps buffers pushing the rendered frame to the display. The vipreViewer is similar in many ways, except for the rendering traversal. This portion of the render loop is performed incrementally. Depending on the hardware used and the volume size and complexity, the time it takes to render a full quality image can vary greatly. It is sometimes necessary to stop a full quality render to provide immediate feedback to users in volume rendering applications. This was very apparent in the mobile sandbox application. In the vipreViewer library, when a chunk of the volume is finished rendering, the render loop checks to see if the full quality render has been cancelled. If it has, then full quality rendering ends and the low quality render begins. This allows developers to use different rendering qualities during the static and interactive states of their applications.

5.1.4 The vipreRaycaster Library

The vipreRaycaster library contains the rendering implementation for performing GPU-based volume raycasting. It implements the Volume and Renderer interfaces from the vipre library to be picked up by the vipreViewer render loop. As for the raycasting implementation, the library is built upon the same core rendering logic used to develop the desktop sandbox application. It consists of the renderer and clipping classes for handling volume reconstruction when clipping is being used. It also contains the ability to perform advanced volume rendering techniques such as
empty space skipping using octrees, Phong illumination and multi-pass rendering for back-face depth rasterization.

5.1.5 The vipreOTSlicer Library

The vipreOTSlicer (vipre orthogonal texture slicer) library is also a rendering technique library built using orthogonal texture slicing. Since it is a rendering technique library, it is built on top of the vipre and vipreViewer interfaces just as the vipreRaycaster library. The rendering logic for this library was borrowed from the implementation of the mobile sandbox application. This library is still under development, but when completed, will have abstracted the rendering core in a way that can be used on all platforms. Therefore, any VIPRE application will be able to use the vipreOTSlicer library to perform volume rendering.

5.1.6 The vipreVATSlicer Library

The vipreVATSlicer (vipre view-aligned texture slicer) library is the final supported rendering technique within VIPRE. It has yet to be completed, but when finished, will be a full implementation of volume rendering using view-aligned texture slicing. It will implement the vipre and vipreViewer interfaces for tight integration with the render loop. It will use a CPU-based slicing algorithm to generate the slice polygons dynamically, then the same shader-based clipping algorithm used by the vipreOTSlicer library.
5.2 Advanced Volume Raycasting Techniques

The sandbox applications were a critical component to the development of VIPRE, but do not implement many advanced volume rendering techniques. Once the foundation of VIPRE was constructed, several advanced volume rendering techniques were explored. These techniques were empty space skipping, Phong illumination and multi-pass rendering for back-face depth rasterization. The following sections describe the development of each of these advanced volume rendering techniques in detail.

5.2.1 Empty Space Skipping using Octrees

There are many ways to improve performance when raycasting. The initial version of VIPRE, built on top of the sandbox applications, was already using front-to-back compositing and early ray termination. It was also using several fragment shader optimizations to reduce the number of fragment operations required during traversal. These are all proven ways to greatly improve the performance of raycasting. However, none of these optimizations are designed to handle large amounts of transparent voxels. Raycasting through these transparent regions is very wasteful. Even though a ray may end up not accumulating a single voxel and discarding the fragment, the ray is still required to step through the entire volume. The algorithm does not have a way to skip over the transparent regions. To improve performance in these cases, an empty space skipping system was designed for VIPRE using octrees.
An octree is a tree data structure where each internal node of the data structure contains exactly eight children. It is constructed by recursively splitting 3D space into eight octants. For more information about octrees, please refer to Section 3.1.1 of this dissertation. To integrate octrees into VIPRE, there were two approaches considered. The first was to use a recursive octree to represent all the possible LODs in a single 3D texture. This approach is robust and results in a single data structure to store all the LODs as well as the original voxel data. The downside to this approach is that it requires border padding resulting in a much larger representation of the original voxel data. This data structure is commonly used when rendering very high polygonal models using volume rendering. The second approach was to use an individual texture for each LOD level stored independently from the original voxel data texture. This approach requires more logic in the fragment shader, but no additional border padding. It also requires fewer fragment operations to perform the indexing conversions between the LOD level traversals. The second approach was used to implement empty space skipping in VIPRE.

The first step was to build a single LOD texture representing the first LOD level of the original voxel data. It was \( \frac{1}{8} \) the size of the original voxel data. For each node in the LOD texture, the minimum and maximum intensities of the eight child voxels were computed and stored in the red and green channels of the LOD texture respectively. The octree LOD texture was then pushed onto the GPU using nearest neighbor interpolation. Once this was completed, the fragment shader needed to be instructed to skip the empty space.
Skipping empty space in the fragment shader required the compositing algorithm to be modified. Instead of always sampling and attempting to accumulate each voxel along the ray, it needed to be done in blocks according to the octree texture. First, the start point of the ray was used to lookup the minimum and maximum values of the block in the octree texture. If the maximum intensity of the block was lower than the minimum intensity being rendered, the block could be skipped because the sampled voxels inside the block would be transparent. If the block was not able to be skipped, then two normal samples along the ray were traversed and composited.

With only a single LOD level being used, the results were very promising. Using octrees in the compositing algorithm resulted in virtually no performance hit when rendering volumes with few transparent voxels. Additionally, in volumes with large areas of transparent voxels, the performance is roughly 3-4 times faster when compared to not using empty space skipping. This performance could also be improved by using more LOD levels in the fragment shader. An example of empty space skipping performance can be seen below in Figure 61.

The only downside to using octrees was that it tended to introduce small artifacts in the rendered image in certain view directions. These artifacts can be seen below in Figure 62. In general, the artifacts were only introduced in oblique view directions due to a shortcoming in the development of the compositing algorithm. In order to ensure only transparent voxels are being skipped, each sample point needs to compute whether it is still within the octree block. In oblique directions, non-transparent voxels are sometimes being skipped causing the small
Figure 61: The vipreDefense example rendering the Yuria dataset at 15 fps (Top). The same view and dataset with octree traversal enabled rendering at 56.8 fps (Bottom).
Figure 62: A closeup screenshot of the Cardiac dataset rendered in the vipreDefense example application (Top). The same closeup with octree traversal enabled (Bottom).
rectangular artifacts. This addition will be added to the final version of VIPRE before being released.

5.2.2 Phong Illumination

Another advanced volume rendering technique is the introduction of an illumination model into the compositing algorithm to produce a lit volume. This can greatly improve the depth perception of surfaces within the volume render. The most common illumination model used in volume rendering today is the Phong Illumination model. In order to compute the Phong illumination at a given voxel requires the voxel position, gradient and color along with the position of the light source. The final voxel color is then determined by compositing the diffuse, specular and ambient illumination with the voxel color. Additional information about Phong illumination can be found in Section 3.1.3 of this dissertation.

In order to use Phong illumination in VIPRE, the fragment shader needed to be extended to compute the gradient at a given voxel as well as compute the diffuse, specular and ambient light interactions. Computing the gradient was done using two different techniques, central differences and forward differences. These are two common gradient calculation methods which can be performed directly in the fragment shader. The results of both types of gradient calculations in addition to Phong illumination can be seen below in Figure 63.

It is easy to tell that the addition of light interaction with the volume greatly improves depth perception of surfaces within the volume. It also introduces small artifacts due to the inaccuracies of the gradient estimators. It is common knowledge
Figure 63: The vipreDefense application with the Yuria dataset loaded with default rendering (Top). The same dataset and view rendered with forward differences Phong illumination (Middle). The same dataset and view rendered with central differences Phong illumination (Bottom).
that the gradient estimator used in volume rendering is one of the most important
calculations affecting the quality of the render. To further improve the rendering
quality, a better gradient estimator such as the Sobel operator needs to be used.
Unfortunately, the better gradient estimators need to be pre-calculated and stored in
a gradient texture due to the computational complexity of the calculation.
Precomputing the gradients will remove the requirement of the calculation to be
done dynamically in the fragment shader, but then requires the memory allocation of
a second texture the same size as the voxel data texture. This essentially doubles
the amount of texture memory required to render the volume. It will however result in
much higher quality volumes. This tradeoff cannot be generalized and must be
considered individually for each volume rendering application.

5.2.3 Multi-Pass Rendering for Backface Depth Rasterization

One major problem with the desktop and immersive sandbox applications
was the inaccuracies of the volume render when using oblique clipping planes. If the
oblique clipping plane is located behind the volume, the volume is rendered
incorrectly. This is due to the inability for the ray exit point calculation to
accommodate non-orthogonal volume geometry. What happens is the exit point is
calculated at the bounding box, but the volume is clipped off before the bounding
box at an oblique angle. This results in proper rendering when the clipping plane is
in front of the volume, and improper renders when behind. Figure 64 attempts to
point out the rendering issue.
Figure 64: The desktop sandbox application demonstrating that rendering the volume in front of the clipping plane is done incorrectly (Top). The same rendering parameters with a different camera position where the volume is located behind the clipping plane resulting in the proper render (Bottom).
This is much easier to comprehend when interacting with the volume and is difficult to point out through static images. The way current applications and libraries handle this problem is to construct a binary clip texture. Inside the clip texture is a binary value for each voxel storing whether that voxel should be clipped. This is not ideal because computing this texture is very expensive and makes it difficult to perform interactive clipping at high framerates. It also requires that another texture be introduced in the fragment shader increasing the memory footprint. A final issue with this approach is that each sampled point along the ray needs to perform an additional lookup with the clip texture to see if it should be clipped. These three combined greatly decrease the functionality and interactivity of clipping with volume rendering.

To avoid using binary clip textures, a different approach was developed. Since the current clipping plane algorithm only produces a convex polygonal volume, there is no need to create a binary clip volume. What needs to be calculated are the rasterized locations of the backfaces of the volume. These backface locations need to then be made available in the fragment shader for compositing. This can be accomplished using multiple render passes in combination with frontface culling and frame buffer objects.

There were several steps involved with using multiple render passes to rasterize the backfaces in VIPRE. The first was to create an additional render pass in OSG which involves using a pre-render camera. Then the volume geometry node is attached and rendered with the camera using frontface culling. Instead of rendering to the default framebuffer, this camera is rendered to a framebuffer object.
(FBO). The resulting FBO texture is then bound to the main render pass shaders as an additional texture. Once the structure is in place, the pre-render pass camera needs to use a custom fragment shader that writes the rasterized backface xyz locations into the rgb values for the fragment color. Examples of the vertex and fragment shaders can be seen in Figures 65 and 66.

```
// Vertex shader varying values
varying vec4 vertex;

void main()
{
    vertex = gl_Vertex;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

**Figure 65:** Vertex shader used for multi-pass rendering using backface rasterization.

```
// Vertex shader varying values
varying vec4 vertex;

// Uniforms from the main program
uniform vec3 cuboidDimensions;

void main()
{
    float red = vertex.x / cuboidDimensions.x;
    float green = vertex.y / cuboidDimensions.y;
    float blue = vertex.z / cuboidDimensions.z;
    gl_FragData[0] = vec4(red, green, blue, 1.0);
}
```

**Figure 66:** Fragment shader used for multi-pass rendering using backface rasterization.

Once the pre-render pass is completed, the backface (exit points) positions are available in the texture used for compositing through the FBO. Then in the fragment shader of the main render pass, the fragment coordinates are used to extract the backface position from the backface texture. This eliminates the need to use the ray to bounding box algorithm as discussed in Section 4.2.1. An image of the
resulting volume render with the pre-render pass texture overlaid on top of the render can be seen below in Figure 67.

![Image of volume render with pre-render pass texture overlaid](image)

**Figure 67:** The vipreDefense application rendering the volume using multi-pass rendering for backface rasterization. The overlay in the bottom left is the backface depth texture generated from the first render pass.

The results of this alternate approach are very promising. By introducing approximately a 10% decrease in performance, all of the improper rendering has been eliminated. This approach coupled with the custom clipping plane implementation of the desktop and immersive sandbox applications is much faster than all other approaches using binary clip volumes and is truly a unique contribution.
5.3 Bridging Academic Research and Volume Rendering APIs

Expanding VIPRE to support the advanced volume rendering techniques was straightforward due to the vast amount of documentation and example applications ranging from DICOM extraction to full volume raycasting. Because of this, VIPRE is going to serve as a platform for bridging the gap between academic research and open source volume rendering APIs. Currently, Voreen is the only available volume rendering API actively contributed to by academic researchers. There are many issues with this. First off, Voreen is not a widely supported open source API with a large community dedicated to development and testing. Secondly, Voreen only supports desktop computers, so researchers looking for immersive and mobile device solutions are left to develop their own implementations. Finally, Voreen is released under the GNU GPL license, so it can only be used for non-commercial purposes. VIPRE is a proposed solution to address these issues directly.

To help bridge the gap between theoretical research and real world applications, volume rendering APIs need to be robust and openly available. VIPRE is a key component in making this happen. By bringing the advanced GPU-based raycasting technology to multiple platforms, researchers and developers will no longer have to build their own internal volume rendering solutions for systems outside the domain of desktop computers.

Another major benefit of the development of VIPRE is that the research community will be provided an open source solution built around a community that supports and welcomes volume rendering enhancements and new contributions. For example, Foo [189] used Fuzzy Logic to perform tumor segmentation, but his work is
not currently available to the research community. The main reason for this is that most volume rendering open source communities do not accept public submissions. In addition, most of these communities are supported by commercial institutions that govern the internal development and future scope of the APIs. In order to avoid this type of situation, VIPRE will serve as a unified hub for volume rendering advancements for the public community. This will help ensure the best technology is available to all on all platforms.
6 CONCLUSIONS AND FUTURE WORK

6.1 Summary and Conclusions

A new volume rendering engine was developed to support multiple platforms through a unified interface. This provides developers with a global volume rendering solution for deploying applications on desktops, laptops, immersive clusters and mobile devices. Before a multi-platform engine could be created, an investigation of the complexities and challenges of performing volume rendering on each platform was required along with a common architecture for performing the core volume rendering. This led to the development of three unique sandbox applications, each of which provided major contributions to the final volume rendering engine.

The desktop sandbox application was developed to build the core volume rendering algorithms for the engine such as resampling, coloring, shading and compositing. The resulting application included all of the core volume rendering algorithms as well as some unique features. Real-time windowing was built directly into fragment shader uniforms to dynamically modify the opacity transfer function. The GPU compositing algorithm was implemented using only the OpenGL specification and not any specific extensions allowing the library to render on any commodity graphics card. Finally, a convex clipping plane algorithm was designed to allow any number of clipping planes to be used to clip the volume in all directions.

The immersive sandbox application was built directly on top of the core volume rendering logic developed in the desktop application. The main focus of the immersive investigation was whether the core volume rendering logic could be
extrapolated to and immersive environment. Based on the current research reviewed, the resulting immersive application is the first application capable of performing GPU-based volume raycasting in large immersive clustered environments. Additionally, the core volume rendering logic did not have to be customized at all to accommodate the immersive platform.

The mobile sandbox application began as an investigation into the capabilities of today’s mobile device hardware. The initial results proved that raycasting cannot yet be performed in real-time due to the lack of 3D texture support. This shortcoming led to the development of an orthogonal texture slicing implementation. To create an interactive volume rendering solution, many performance enhancing features were built into the mobile application, each of which could be utilized by both rendering algorithms in the engine. These features include a method for dynamically modifying the render resolution, an incremental rendering loop for canceling high resolution renders, a shader-based clipping algorithm for OpenGL ES 2.0 and an internal backface culling algorithm to perform depth sorted rendering with alpha blending.

The completion of the sandbox applications verified that the common architecture used could support multi-platform volume rendering. This led to the development of VIPRE, or the Volume Image Processing and Rendering Engine. VIPRE contains the following features:

- Two volume rendering algorithms: raycasting and orthogonal texture slicing
- Three rendering modes: composite, MIP and MinIP
- Four preset opacity transfer functions and eight color transfers functions
- Real-time windowing using a dynamic shader-based opacity controls
- Bilinear and trilinear interpolation for 2D and 3D fragment shader textures
- Custom CPU and GPU-based clipping algorithms
- Accurate depth sorted rendering with alpha blending
- Dynamic render quality modification using multiple render passes
- Early ray termination and empty space skipping
- Phong illumination supporting multiple gradient operators
- Multi-pass rendering for backface depth rasterization
- Support for all commodity desktop graphics cards in addition to iOS devices

VIPRE was designed from the beginning to provide a unified solution for performing volume rendering on multiple platforms. This produced a robust volume rendering core which is able to be extended to support more complex forms of volume rendering. To help enable the extension of VIPRE, simplified versions of the sandbox applications as well as a robust set of documentation are included in the library to provide a cohesive starting point for novice and intermediate developers. Additionally, VIPRE is going to be released under licensing terms to allow it to be used in both the academic and commercial communities. The intention of this is to provide researchers and developers the ability to create new inventive methods of interaction with volumetric data without having to build their own volume renderer. Today’s technology is ever advancing, and VIPRE is an attempt to lower the barrier to entry for volume rendering to usher in a new generation of volume rendering applications. Competition fosters innovation, and by making volume rendering more accessible to researchers and developers, everyone can benefit.
6.2 Future Work

For the future development of VIPRE, there are several areas of focus. The first focus area is surgical planning. Surgical planning requires inserting instruments into the volume. VIPRE currently only supports proper depth rendering for geometry located outside of the volume. To render geometry inside the volume properly, an additional render pass needs to be implemented to build a depth map of the internal volume geometry. This depth map can then be integrated into the compositing algorithm in the main rendering pass to stop the ray traversal at the proper location.

Another area of focus is segmentation. By integrating a segmentation library into VIPRE, new methods of interaction for training segmentation routines could be studied in different platforms such as immersive clustered displays or mobile devices. Additionally, VIPRE will be extended to support multiple volumes and independent volume clipping to help visualize the internal structures of a segmented dataset such as a tumor. By reusing textures generated by previous render passes, more interactive user interfaces could be designed for immersive applications.

6.3 Acknowledgements

I would like to take this opportunity to thank everyone involved with the work presented in this dissertation as well as those who supported me throughout my graduate career. First and foremost, I would like to thank my fiancé, Rachael Gonzales. Without you I never would have had the strength to pull through the difficult times. I would also like to thank my family for their love, support and understanding through all the years.
Next I would like to thank my advisor Dr. Eliot Winer. Not only did your guidance help make me the person I am today, but without you I don’t know if I would have ever attended graduate school.

To the faculty and staff of the Virtual Reality Applications Center at Iowa State University, thank you for all the assistance you have given me and for making my graduate school experience as meaningful and worthwhile as possible. It has truly been a wonderful experience that I will never forget.

And last but not least, my research colleagues Dr. Eric Foo and Brandon Newendorp, your time and input throughout many design discussions and late night debugging sessions was essential to the development of VIPRE.
BIBLIOGRAPHY

Publishing Co.

37. Wu, Y., et al. Shear-image order ray casting volume rendering. in Proceedings
York, USA: ACM.

38. Rezk-Salama, C., et al. Interactive volume on standard PC graphics hardware
using multi-textures and multi-stage rasterization. in Proceedings of the ACM
SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware. 2000. New
York, New York, USA: ACM.

39. Engel, K., M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering
using hardware-accelerated pixel shading. in Proceedings of the ACM
SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware - HWWS '01.

40. Roettger, S., et al. Smart hardware-accelerated volume rendering. in
Eurographics Association.

41. Hsu, W.M. Segmented Ray Casting for Data Parallel Volume Rendering. in

42. Ma, K.L., Parallel volume ray-casting for unstructured-grid data on distributed-
memory architectures, 1995.

43. Ma, K.L. and T.W. Crockett, A Scalable Parallel Cell-Projection Volume
95.

44. Heng, Y. and L. Gu. GPU-based Volume Rendering for Medical Image
Visualization. in Conference proceedings : ... Annual International Conference
of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering

Volume Raycasting. in ACM SIGGRAPH 2009 Courses. 2009. ACM.

46. Whitted, T., An improved illumination model for shaded display.

47. Levoy, M., Efficient ray tracing of volume data. ACM Transactions on

48. Weiler, M., et al., Hardware-based ray casting for tetrahedral meshes. IEEE
333-340.


51. Wan, M., et al. Volume Rendering based interactive navigation within the

52. Wan, M., A. Sadiq, and A. Kaufman. Fast and reliable space leaping for
interactive volume rendering. in Proceedings of the conference on


178. VRVis. 2011.
179. ImageVis3D Mobile. 2011.
189. Foo, J.L., A Framework for Tumor Segmentation and Interactive Immersive Visualization of Medical Image Data for Surgical Planning, in Isis2010, IOWA STATE UNIVERSITY.