UNIFIED GRAPHICS API FOR A DISPLAY-AGNOSTIC RENDERING SYSTEM

A Thesis

by

RATHINAVEL SANKARALINGAM

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Carol LaFayette
Co-Chair of Committee,	Philip Galanter
Committee Members,	Shannon Van Zandt
Head of Department,	Tim McLaughlin

December 2017

Major Subject: Visualization

Copyright 2017 Rathinavel Sankaralingam

ABSTRACT

This research focuses on development of a unified graphics application programming interface in which several types of 3D-displays that require multi-view rendering, such as Virtual Reality/Augmented Reality Headsets, Light Field Displays, and Volumetric Displays, can exist within a heterogeneous display environment. In general, GPU architecture and programming languages are not inherently well optimized for multi-view rendering. 3D displays such as stereo, volumetric and light-field displays may require a minimum of two to several hundreds or thousands of views to be rendered per display update based on the architecture of the display. Moreover, there is not a single binding software architecture standard that can exploit the common multi-view rendering needs of multi-view 3D displays. In this thesis, I extend the current multi-view light-field rendering graphics library Object Graphics Language(ObjGL), a single common standard library developed at FoVI^{3D}, to support stereo rendering in virtual reality displays including Oculus Rift, and HTC Vive, as well as explore the potential challenges in the march towards a truly heterogeneous display environment.

DEDICATION

To my mom and dad. For this wonderful life, for giving me only the best, for all the trust you both have in me, and for all the love. Thank you Amma and Appa.

ACKNOWLEDGMENTS

I want to sincerely thank my committee chair Carol LaFayette for encouraging me all along and for her constant motivation and support in helping me shape this thesis from day one. I would also like to express my sincere gratitude to Thomas Burnett, the CTO of FoVI^{3D} for giving me this wonderful opportunity to develop ObjGL, for being a great mentor and for all the technical guidance throughout this research.

I am also thankful to my committee members Philip Galanter and Shannon Van Zandt for their support and timely feedback. I cannot finish my acknowledgements without thanking fellow engineers at FoVI^{3D} who were always there to lend a hand to support me along this journey.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised and supported by a thesis committee consisting of Professor Carol LaFayette and Professor Philip Galanter of the Department of Visualization, Professor Shannon Van Zandt of the Department of Landscape Architecture and Urban Planning and Thomas Burnett, Chief Technology Officer of the Austin based start-up FoVI^{3D} . All work for this thesis was completed independently by the student.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

NOMENCLATURE

API	Application Program Interface
AR	Augmented Reality
DUPS	Display Updates Per Second
FPS	Frames Per Second
GPU	Graphics Processing Unit
GLSL	OpenGL Shading Language
HMD	Head Mounted Display
Hogel	Holographic Element
MvPU	Multiview Processing Unit
OLED	Organic Light Emitting Diodes
OpenGL	Open Graphics Library
OpenVR	Open Virtual Reality
VR	Virtual Reality

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

NOMENCLATURE vi

TABLE OF CONTENTS vii

LIST OF FIGURES ix

LIST OF TABLES xi

1. INTRODUCTION 1
   - 1.1 Introduction 1
   - 1.2 Motivation 2
   - 1.3 Goal 2

2. BACKGROUND 4
   - 2.1 Electronic Display Technologies 4
   - 2.2 3D Displays 7
     - 2.2.1 Stereoscopic Displays 8
       - 2.2.1.1 Virtual Reality Displays 11
     - 2.2.2 Autostereoscopic Displays 13
       - 2.2.2.1 Light-Field Displays 13
       - 2.2.2.2 Volumetric Displays 15
   - 2.3 OpenGL 17
     - 2.3.1 OpenGL Graphics Pipeline 18
     - 2.3.2 Fixed Function Pipeline 18
     - 2.3.3 Programmable Pipeline 20
     - 2.3.4 OpenGL Framebuffer 22
   - 2.4 Multiview Rendering and Limitations with OpenGL 22

3. METHODOLOGY AND IMPLEMENTATION 26
3.1 Heterogeneous Display Environment 26
3.2 ObjGL 27
3.2.1 Control Instructions 30
3.2.2 Cache Instructions 30
3.2.3 Render Instructions 31
3.2.4 ObjGL Shapes 33
3.2.5 ObjGL High-level Rendering Acceleration Constructs 33
3.2.5.1 Bounding Volumes and Frustum Culling 34
3.2.5.2 Critical Objects 34
3.2.5.3 Foreground and Background Segmentation 35
3.3 Implementation 35
3.3.1 Virtual Reality Rendering 35
3.3.2 OpenVR 36
3.3.3 ObjGL Host and Render Clients 38
3.3.4 ObjGL Thread Model 40
3.3.5 ObjGL Forwarding 41
3.3.6 RenderStereo 41
3.3.6.1 Init() 42
3.3.6.2 Render() 44
3.3.7 Heterogeneous Display Rendering in Action 46
3.3.8 Test Results 48
4. SUMMARY AND CONCLUSIONS 50
4.1 Future Work 50
REFERENCES 52

LIST OF FIGURES

FIGURE Page

2.1 A Segment Display ... 5
2.2 A CRT Monitor. ... 5
2.3 An LED Display Board ... 5
2.4 Flat Panel LCD TV ... 6
2.5 An iPhone Retina Display ... 6
2.6 Anaglyph 3D Glasses ... 9
2.7 Anaglyph 3D Image ... 10
2.8 Active Shutter Glasses ... 10
2.9 Virtual Reality Displays ... 12
2.10 A Virtual Reality Demo on Oculus Rift ... 12
2.11 Radiance Image generated for Light-field Display at FoVI^{3D} ... 14
2.12 Conceptual Light Field Display ... 15
2.13 Looking Glass Factory's Volumetric Display ... 16
2.14 Voxon Photonics' Voxiebox ... 17
2.15 Fixed Function Pipeline ... 19
2.16 Fixed Function Vertex Transformation flow ... 20
2.17 OpenGL Programmable Pipeline ... 21
2.18 Programmable Pipeline Vertex Transformation flow ... 21
2.19 DPS Comparison chart for single/Multiview rendering ... 25
3.1 A Model Heterogeneous Display Ecosystem ... 27

3.2 ObjGL Rendering Model in a Heterogeneous Display System 29
3.3 Vertex List Definition 31
3.4 Example ObjGL Instruction Set Ordering 32
3.5 ObjGL Shapes - Prisms, Gears, and Pyramids................................ 33
3.6 OpenVR 37
3.7 ObjGL Client-Server Render flowchart 39
3.8 ObjGL Client Render Model 40
3.9 RenderStereo JSON file... 42
3.10 Heterogeneous Display Rendering in action - Battle Field Simulation....... 47
3.11 Heterogeneous Display Rendering in action - ObjGL Custom Shape Rendering 48

LIST OF TABLES

TABLE Page

2.1 Display Update Rate Comparison for Single/Multi view Rendering. 24

3.1 ObjGL Instruction Set. 30

1. INTRODUCTION

1.1 Introduction

Digital display devices such as television sets, mobile phones and computer monitors are ubiquitous in today's world, thanks to rapid and ever-growing cutting-edge research and innovation. Cathode Ray Tubes, Liquid Crystal Displays and Light Emitting Diodes are some examples of how display technologies have evolved over the past several decades. Common to all these displays is the fact that they display 2D flat images or motion pictures. Traditional 2D digital displays present a single view of a scene that lacks third dimension of depth information. This can lead to misinterpretation and ambiguity caused by a viewer's experience or lack thereof, in reconstructing a 3D model/scene from flat projected data. This handicaps the effort to truly understand the projected information or terrain by visualizing such data on 2D displays, creating a need for coherent 3D visualization display technologies.

Historically anaglyph glasses and other simple stereo techniques were used to view a slight 3D effect in still images and motion pictures projected from 2D displays. With the advent of modern GPUs and their massive parallel processing power, enormous amounts of 3D content are developed today for multiple applications in different disciplines such as gaming, movies, simulation, and architecture. Yet, visualizing a 3D object in a 2D display is not the same as seeing the object with naked eye. To better understand three-dimensional data with height or depth information, technologies are needed that can display real 3D content with depth and other cues aided by the binocular disparity of human eyes. Research and innovations in three-dimensional display technologies have been carried out for a long time, and a wide range of novel and innovative 3D displays have also been developed to visualize real 3D content. Each of these displays employ different and

innovative optical solutions to achieve similar goals of visualizing three-dimensional information. Usually these 3D displays are either stereoscopic or auto-stereoscopic. Stereoscopic (two-view) displays include Augmented Reality/Virtual Reality Head Mounted Displays(HMDs) that provide two slightly different images/views of the same scene, one for each eye, with a head tracking mechanism that aids in creating a 3D virtual environment, thus providing the user with an immersive experience. Auto-stereoscopic displays, which include novel technologies like Light-field displays and Volumetric displays, allow for the visualization of 3D aerial imagery without eye/head mounted displays or tracking.

1.2 Motivation

Despite an ever-increasing number of 3D display technologies and architectures, manufacturers typically create proprietary solutions to stream and render 3D content. This makes it increasingly difficult to attach or detach different displays to a single rendering stream. In essence, there is no common standard or single graphics specification that would allow various kinds of displays to connect to a single host and share the same content by exploiting the multi-view rendering nature of these displays. In the case of 3D displays, which require multiple views of the scene to be generated, the need becomes paramount for technologies that can compute these views quickly and efficiently. For an effective and collaborative analysis effort, a multi-display visualization system that would run on a common software specification adds immense value. Hence, there is a need for a single graphics specification that would allow streaming and rendering of data to multiple displays at the same time while allowing for the 3D display to efficiently render multiple views.

1.3 Goal

FoVI^{3D} proposes ObjGL, a software solution that can bind together multiple display technologies to a single rendering stream. The core objective of ObjGL is to relieve the

need for 3D displays to develop proprietary software technologies to stream and render data. The objective of this research is to create graphics software that can support a coherent heterogeneous display ecosystem of multi-view rendering displays. On that note, this thesis extends and validates ObjGL while streaming and rendering 3D content to stereo displays such as the Oculus Rift and HTC Vive.

2. BACKGROUND

2.1 Electronic Display Technologies

A digital display is a form of output device that can show valuable information such as text, image or video transmitted electronically. Today electronic display devices are everywhere, in every home, and to a great extent, in every individual's possession(for example, television sets, computer monitors, laptops, digital watches, mobile phones and tablets). Computer processors can perform simple operations such as addition or subtraction, with data input by the user and with the computer monitor displaying the results. Other devices like television sets just receive 2D motion pictures without much input or data processing.

Display technologies have evolved over the last century from simple segment displays(Figure 2.1) that switch on or off several segments to head-mounted immersive stereo displays and glasses-free 3D display technologies. Cathode Ray Tubes(CRT)(Figure 2.2), Light Emitting Diodes(LED)(Figure 2.3), Light Crystal Displays(LCD)(Figure 2.4) and Organic Light Emitting Diodes(OLED) are some ground-breaking display technologies that evolved at different points in the history of electronic displays. These digital displays employ different techniques for varying applications. For example, segment displays are used mainly in digital watches and pocket calculators; cathode ray tubes were used in old television sets and monitors. Today hand-held personal devices such as mobile phones and tablets come with retina displays(Figure 2.5) that can project clear high-resolution images.

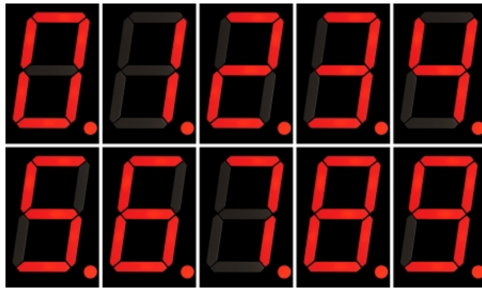
Figure 2.1: A Segment Display

Figure 2.2: A CRT Monitor.

Figure 2.3: An LED Display Board

Figure 2.4: Flat Panel LCD TV

Figure 2.5: An iPhone Retina Display

This great progress in display technologies have brought us a long way in how we receive, perceive, experience and interpret all the information. Mass media – film, entertainment, internet, gaming, architecture and data visualization – have taken new shapes around

these display technologies. One important thing to note about all the above displays is that they all show only 2D imagery. They create a two-dimensional array of colored picture elements a.k.a., pixels that stacked tightly next to each other, project image or text. But in the real-world, we see objects in three dimensions and we perceive the third dimension with our naked eye.

Traditional 2D display devices show flat images that lack depth or 3D information. This restriction greatly diminishes our ability to perceive and understand the complexity of real-world scenes when viewed from 2D displays. Moreover, it becomes difficult for some users to form accurate mental models of 3D scenes viewed on flat panel projections. It is hard to distinguish elements of interest at different heights and depths, thereby making it difficult to come to accurate conclusions and to make informed decisions. For example, when perceiving height is crucial, for example in military applications, some forms of battlefield visualization devices can make it tricky to understand nuances. The complex nature of battlefield visualization entails 3D terrain, spheres/ranges of influence, flight or trajectory paths and many other complex 3D concepts. If proper depth cues are missing, viewing high-dimensional 3D content in 2D displays can sometimes cause ambiguity and confusion in the mind of the viewer. This especially becomes an issue during collaborative exercises where individual viewers develop unique 3D mental models based on the perspective or experience.

2.2 3D Displays

In the real world, the objects we see are perceived as 3D due to binocular viewing that provides depth perception. Accommodation, convergence, motion parallax and binocular disparity assist in perceiving depth of three dimensional objects, along with the depth cues – occlusion, shading and specular highlights. This ability to resolve depth within in a scene either natural or virtual improves spatial understanding of the scene and reduces

the cognitive load associated with analysis and collaboration on complex tasks. With recent advancements in technology, true 3D displays are making their way into mainstream. 3D display technologies are stereoscopic or auto-stereoscopic. Auto-stereoscopic displays do not require wearing special eye glasses to view the 3D content. Light Field Displays and 3D volumetric displays are examples of autostereoscopic displays. On the other hand, binocular stereoscopic displays require special eyewear to be able to view the 3D content [1]. In many 3D display technologies, multiple views are typically used to give a sense of depth and perspective. A survey by Air Force Research Laboratory(AFRL) suggests Joint Terminal Attack Controllers(JTACs) who were shown 3D holograms and conventional 2D images rated 3D holograms to be more effective than the 2D imagery in terms of retaining height information [2]. Studies have also found that virtual reality devices can improve and enhance spatial ability and understanding by providing interactive 3D visualization [3]. In a study examining the usefulness of stereo images in elementary school education, stereoscopic 3D images were shown to reveal details that were largely ignored in 2D images [4]. 3D display technologies have also been proven to improve medical visualization by reducing ambiguity in complex volume rendered medical images [5].

2.2.1 Stereoscopic Displays

Stereo displays produce a depth effect by displaying a pair of slightly different images to each eye. Stereoscopic displays usually require viewers to wear special eye glasses. This stereo vision to create 3D illusion can be generated using different techniques. Traditional anaglyphic glasses(Figure 2.6) with red and cyan filters for each eye have been in use for a long time now. The red and cyan filters distill the two separate images for each eye from the 3D anaglyph shown in Figure 2.7. In the recent past, active shutter glasses(Figure 2.8) have brought revolutionary 3D viewing in consumer electronics. Active shutter glasses are synced with the television set which displays two images alternating

at the same rate as the glasses; the left eye glass is blacked out when the right image is shown and vice versa, effectively showing different images for each eye. Today Virtual Reality/Augmented Reality headsets create a new wave in visualizing 3D content by creating an immersive experience. Virtual Reality Head Mounted Displays like Oculus Rift and HTC Vive exploit the binocular disparity of human eyes and display two different images to each eye. This creates a depth illusion to the eyes and the head tracking mechanism in these displays gives the user a unique and immersive experience.

Figure 2.6: Anaglyph 3D Glasses

Figure 2.7: Anaglyph 3D Image

Figure 2.8: Active Shutter Glasses

2.2.1.1 Virtual Reality Displays

The last few years have been truly revolutionary as far as developments in Virtual Reality technology are concerned. Oculus Rift, PlayStation VR, and HTC Vive(Figure 2.9) with their head tracking mechanisms create an unparalleled immersive experience, with applications in a wide range of disciplines including gaming, films, education, training and simulation, and many others. Mobile VR applications have also been on the rise thanks to Samsung Gear VR and Google Cardboard(Figure 2.9). Usually these VR displays come with a head mounted display or HMD that the user wears over the head and eyes. They use sensors or base stations to determine the position and orientation of the head mounted displays to perform head and motion tracking within a confined/calibrated physical space. Inside the physical space the user can move around and get a compelling and unique immersive experience of being inside the virtual environment(Figure 2.10). In virtual reality, two virtual cameras separated by the distance between human eyes, known as the Inter Pupillary Distance(IPD), are used to capture the 3D scene for stereo rendering.

Samsung Gear VR

HTC Vive

Oculus Rift

Google Cardboard

Figure 2.9: Virtual Reality Displays

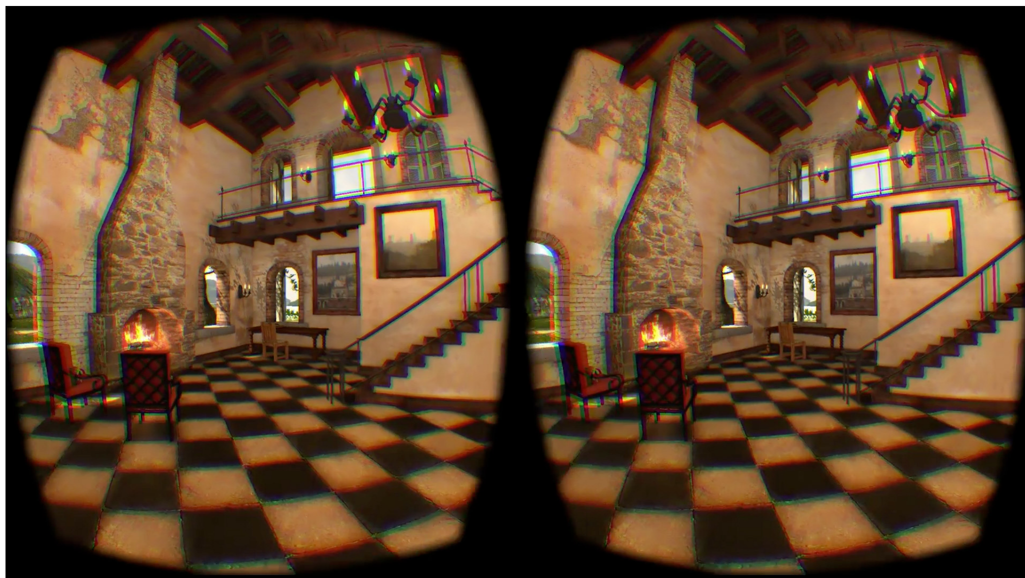

Figure 2.10: A Virtual Reality Demo on Oculus Rift

2.2.2 Autostereoscopic Displays

The term "autostereoscopic display" refers to any display technology in which the viewer perceives 3D without tracking the viewer's position and orientation. Almost all 3D devices that do not require wearing stereo goggles or stereo head-mounted displays are autostereoscopic. These are also known as glasses-less 3D or glasses-free 3D. There are multiple types of autostereoscopic devices, including light-field and holographic displays, lenticular displays and volumetric displays. These technologies employ different strategies and techniques in achieving the common goal of perceiving the third dimension without the use of any eyeglasses or head tracking gears. One of the major advantages of autostereoscopic displays is the fact that, since there is no need to wear any head gear, a group of people can stand around and experience real three-dimensional content collaboratively.

2.2.2.1 Light-Field Displays

Light-field display technology aims at providing a continuous head parallax experience over a wide viewing zone leading to the perception of 3D imagery that is visible to the unaided eye i.e., without the use of special glasses or tracking mechanisms [6]. This involves a continuous reconstruction of the light-field from a 3D scene. It allows for perspectivally correct visualization of the 3D scene within the display's projection volume. Light field rendering is the process of generating an enormous number of views within a confined 3D cuboidal space known as the view volume. These views are projected as micro-images known as Holographic Elements(Hogels) [7] in a large 2D pixel array known as the radiance image or the plenoptic image(Figure 2.11). This plenoptic image of hogels is then projected through a micro-lens array thereby creating a continuous 3D light-field. Each hogel represents the position, orientation and intensity of light rays passing through the micro-lens. However, the process of creating the radiance image is a complex

activity because rendering the 3D geometry is unique for each hogel/micro-lens. Light-field rendering requires an enormous number of views of the same scene to create a level of realism for perspectivally correct visualization and hence requires more processing power than stereo rendering which requires only two views. Figure 2.12 shows a conceptual light-field display.

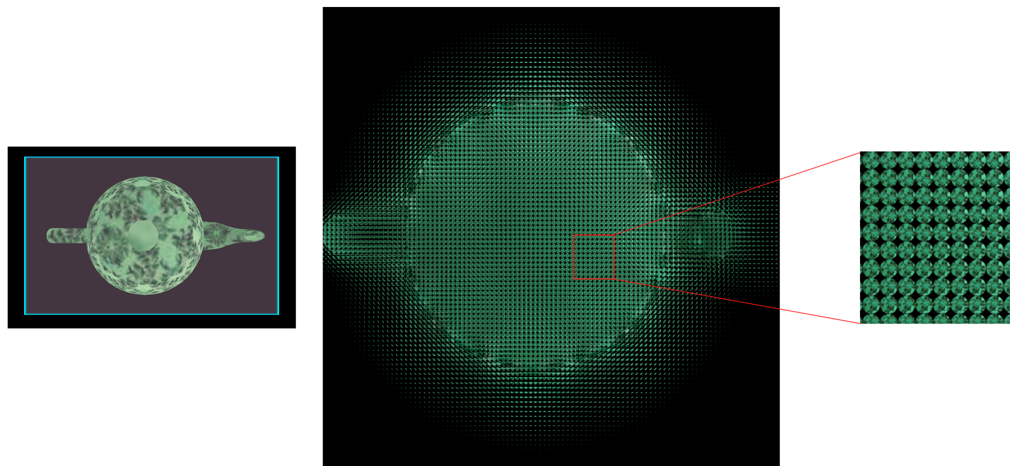

Figure 2.11: Radiance Image generated for Light-field Display at FoVI^{3D}

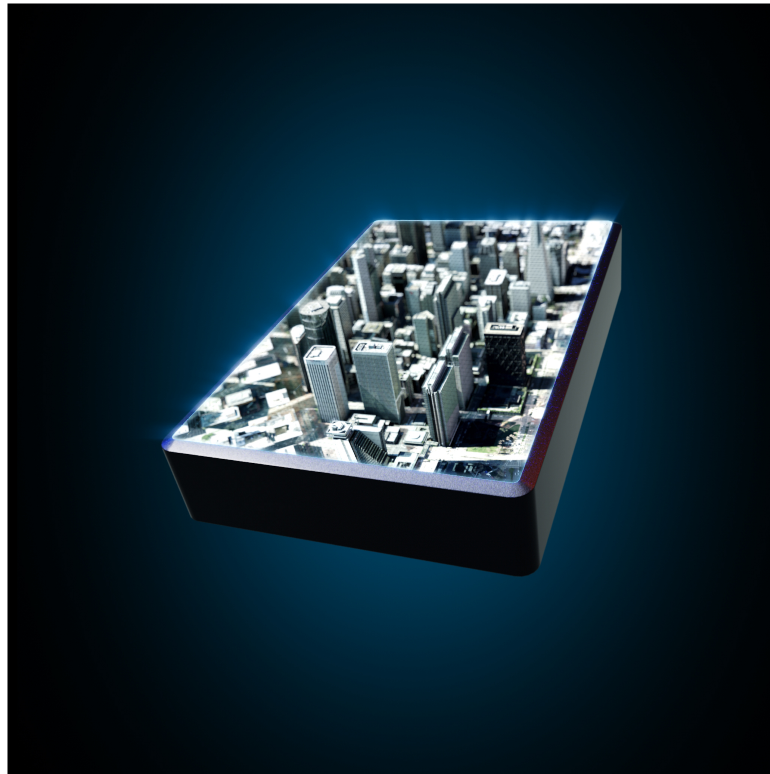

Figure 2.12: Conceptual Light Field Display

2.2.2.2 Volumetric Displays

Over the last few decades, multiple companies and universities have developed volumetric display solutions. Most of them differ in the techniques and optics behind the volumetric projection. Volumetric display technologies are usually static screen(solid-state upconversion, gas medium, voxel array, layered LCD stack) or swept-screen(rotating LED array, cathode ray sphere, varifocal mirror, rotating helix and so on) [8]. Laser based volumetric displays use lasers to create luminous points of light at desired locations in air or in water to create a floating 3D object. Other displays often employ projection onto a multi-planar display volume where glass panels are stacked up one after another. Like light-field displays, these volumetric displays also aid in creating a three-dimensional per-

ception without the aid of special glasses. Holovect, Looking Glass Factory(Figure 2.13), Voxon(Figure 2.14), and Holografika are some of the notable manufacturers who have created Volumetric displays.

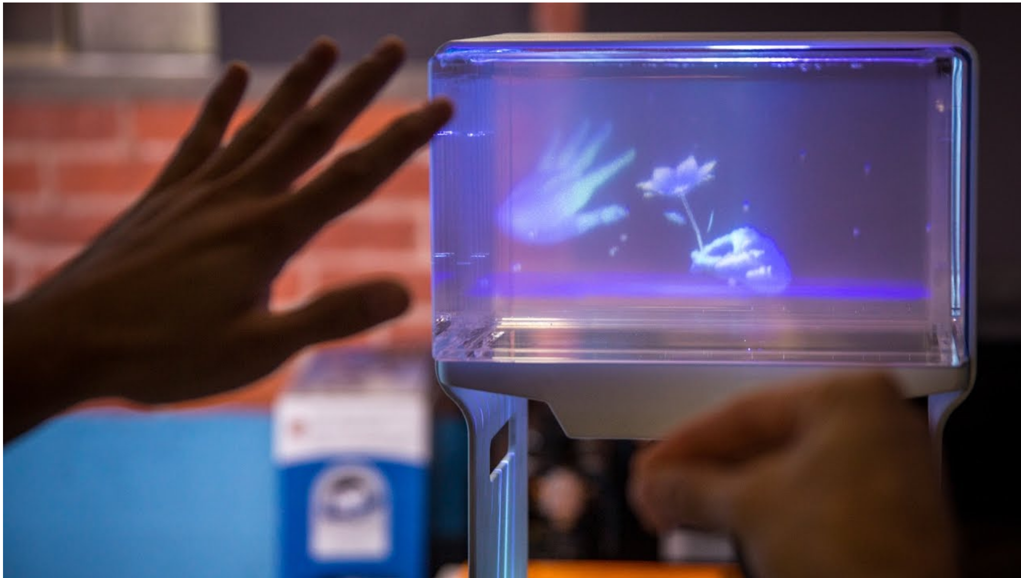

Figure 2.13: Looking Glass Factory's Volumetric Display

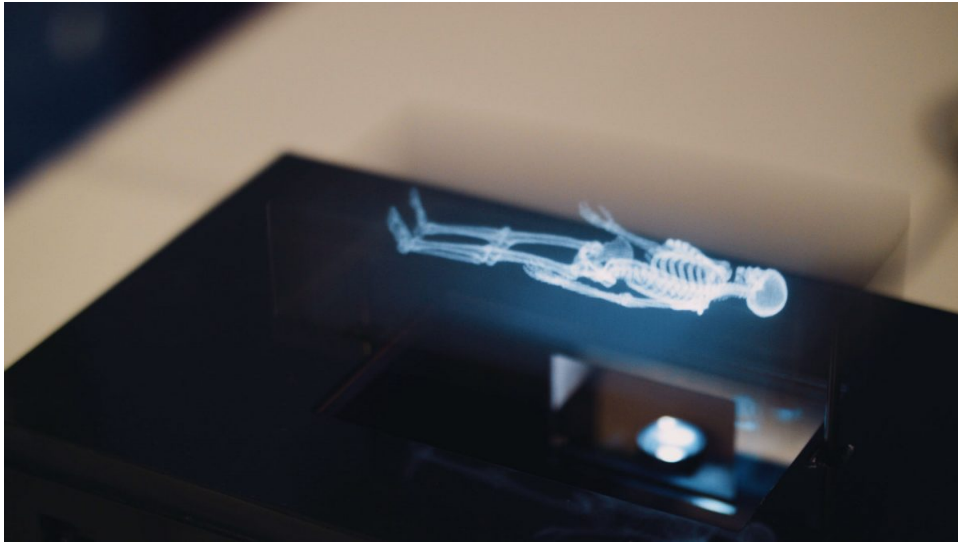

Figure 2.14: Voxon Photonics' Voxiebox

2.3 OpenGL

Open Graphics Language(OpenGL) is a low-level cross-platform application programming interface for rendering 2D and 3D graphics. OpenGL interacts with the Graphics Processing Unit(GPU) to facilitate hardware accelerated rendering. OpenGL was developed and released by Silicon Graphics, Inc.(SGI) in January 1992. Since then, OpenGL has become the most-widely used open graphics standard in the world. Currently OpenGL specification is managed by the non-profit technology consortium Khronos Group. OpenGL has found its application in numerous fields that require maximum performance such as CAD, entertainment, medical imaging and virtual reality to produce compelling graphics. Microsoft's proprietary Direct3D, which is often compared with OpenGL as a competing API, targets only Windows platforms. This gives OpenGL an obvious advantage of catering to cross platform applications and hence a ubiquitous presence. Mantle developed by AMD and Metal by Apple are other notable graphics programming libraries.

2.3.1 OpenGL Graphics Pipeline

The OpenGL pipeline represents a sequence of steps and mathematical operations that transform input 3D attributes like vertices and normals along with textures/texture coordinates and a virtual camera to a 2D array of pixels. The rendering pipeline consists of various stages that implement core graphics algorithms.

OpenGL has evolved radically over the past two decades and has gone through several architectural changes. As the hardware capabilities increased over the years, enabling faster matrix computations, which is a prerequisite for 3D graphics, OpenGL has become a much more sophisticated rendering library and an obvious choice for performance critical and interactive applications. The major development in OpenGL architecture in recent years is the ability to directly program and manipulate graphics hardware using short programs known as "Shaders." Shaders carry out very specific well-defined operations like vertex transformation or pixel manipulation. This programmable pipeline gives more flexibility to developers to carry out complex and custom operations with vertices and colors which were previously hard to accomplish with the legacy fixed function pipeline. The fixed function pipeline had built-in functions that users can use directly to specify operations. But this lacked the flexibility that comes with the shaders in manipulating the vertices, colors or geometric primitives.

2.3.2 Fixed Function Pipeline

In early versions of OpenGL, the API had built-in function calls to handle operations such as matrix transformations and lighting. The term "fixed function" refers to the fact that most of the commands in the pipeline act as fixed function entry points and carried out only the designated operations. These operations were built into the hardware, and there were very little to no means to modify how the mathematical operations were implemented. While it was easy to set up and render objects, it limited

developers to only the built-in functionalities. As can be seen in Figure 2.15 [9], the fixed function pipeline consists of rigid placeholder functions that carried out only a series of designated operations. In fixed function OpenGL, the model and view matrix transformations are always defined and collated using the function definition glMatrixMode(GL_MODELVIEW)(Figure 2.16), which makes the pipeline rigid to application of custom transformations.

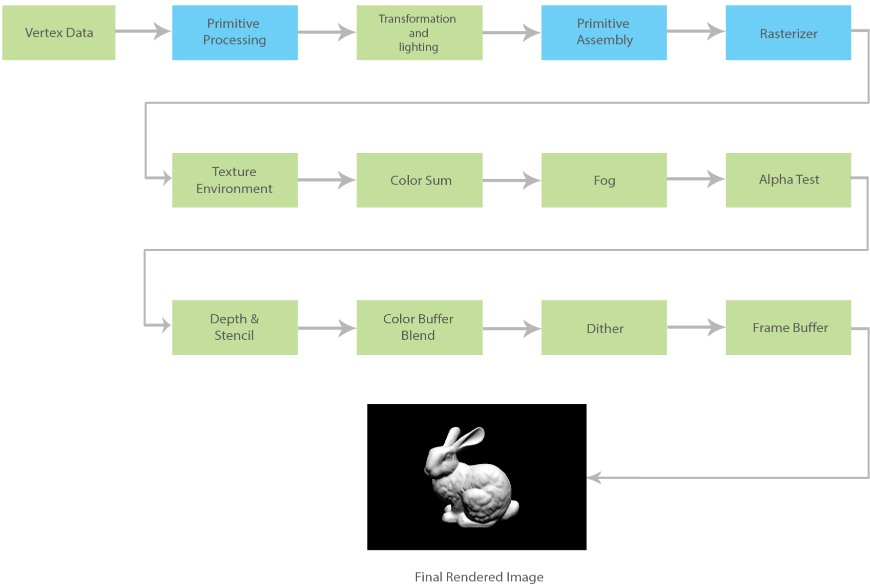

Figure 2.15: Fixed Function Pipeline

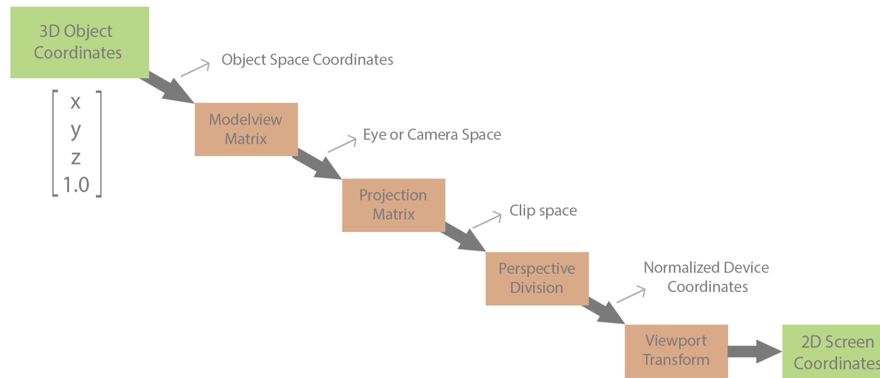

Figure 2.16: Fixed Function Vertex Transformation flow

2.3.3 Programmable Pipeline

As GPUs evolved, several parts of the fixed function pipeline were replaced by a shader based architecture. OpenGL 2.0 and later versions come with a programmable pipeline where the programmable part is achieved with shaders written in GLSL(OpenGL Shading Language) a C-like language with backward compatibility for a fixed function pipeline. GLSL was created to give graphics developers more direct control of the graphics pipeline. While the programmable pipeline is a lot more flexible, it is less intuitive than the fixed function pipeline and requires a lot of code to render even relatively simple geometry like a triangle. The modern OpenGL pipeline has different programmable stages such as vertex shaders, tessellation shaders, geometry shaders, and fragment shaders. Each of these stages acts as the input of the other; however, each stage executes in parallel, keeping GPU utilization very high. In Figure 2.17 [10], the blue rounded edge boxes show different shading stages where users have more control over operations. In a stark contrast from fixed function OpenGL, shaders allow users to manipulate model and view matrices separately allowing for more flexibility as seen in Figure 2.18.

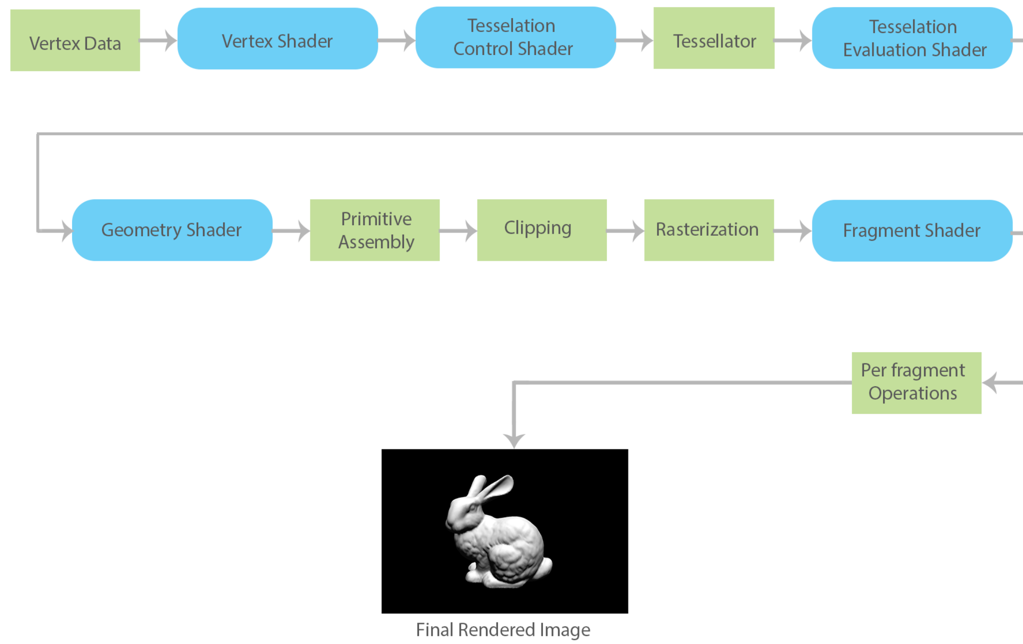

Figure 2.17: OpenGL Programmable Pipeline

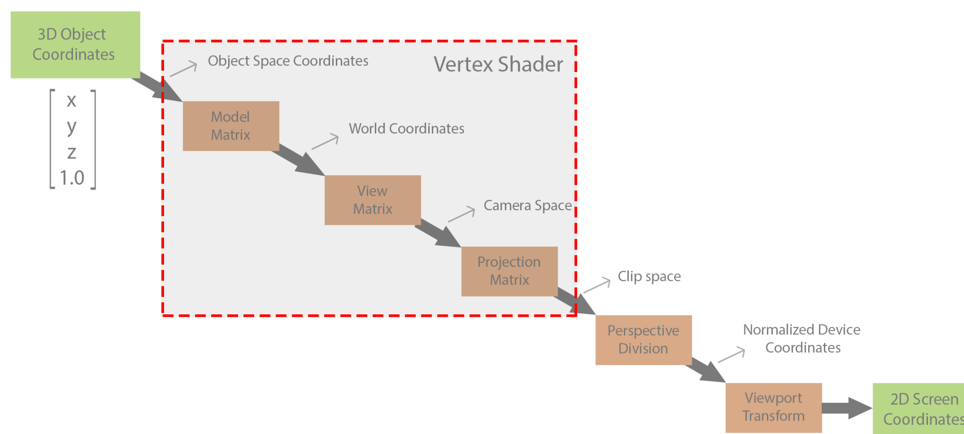

Figure 2.18: Programmable Pipeline Vertex Transformation flow

2.3.4 OpenGL Framebuffer

OpenGL always renders into a region of memory known as the framebuffer. A framebuffer is a collection of buffers that can be used as the destination for rendering [11]. OpenGL has two kinds of framebuffers: the default framebuffer, which is provided by the OpenGL context; and user-defined framebuffers known as Framebuffer Objects or FBOs. The default framebuffer usually represents a window or a display device and is bound automatically when a new context is created. Framebuffer objects are generally used for off-screen rendering that allows applications to perform operations like shadow mapping, reflections, post processing and many other effects.

The framebuffer is a combination of a color buffer, depth buffer and a stencil buffer. It is a container for these buffers, where the depth and stencil buffers can be optional. There are three types of attachment points to a framebuffer in OpenGL.

1. *Color*: the outputs from the fragment shader

2. *Depth*: Z buffer for the framebuffer object

3. *Stencil*: the stencil buffer used for storing per-pixel masks

Each framebuffer may have multiple color attachments but only one depth and stencil buffers. The framebuffer can be used for both drawing and reading purposes. It can be bound for drawing purposes using GL_DRAW_FRAMEBUFFER when draw commands are issued and it can be bound for reading purposes using GL_READ_BUFFER when glReadPixels() function is used.

2.4 Multiview Rendering and Limitations with OpenGL

As we discussed earlier in the 3D displays section, light field rendering and virtual reality technologies are based on multiple view point rendering to generate the sense of depth

and perspective to the imagery. Different Multiview rendering applications have been developed for autostereoscopic displays using brute force techniques, geometry shaders and so on [12] [13]. OpenGL and other graphics APIs are well optimized for utilizing the massive parallel processing power of modern GPUs when rendering from a single view point and have been widely used for interactive applications in general. Yet, OpenGL as well as the GPUs are suitable neither for processing multiple views in parallel nor for efficiently rendering multiple views for Multiview displays [1]. Many developers of novel display architectures use fixed-function OpenGL shims to intercept draw commands and forward them to a display specific renderer. However, there is no collective agreement on which version to shim, and each vendor creates their own proprietary graphics solutions [14]. Hence, applications developed using a particular version of OpenGL may or may not be usable from such devices, causing compatibility issues.

Even when the modern OpenGL's programmable pipeline allows more flexibility in handling creative rendering using shaders, only one projection transform, view transform and viewport transform can be active at a given time. Thus, the enormous processing power of the GPU is allocated to rendering a single view to a framebuffer target. Hence, for multi-view rendering, multiple passes of the geometry must be made serially, each with a unique view transform and a small render target within a large framebuffer. This serially increases the number of vertex transforms by the number of viewpoints rendered into the framebuffer. Even in current stereo rendering techniques, vertices must be transformed twice using view and projection matrices to produce two views of the same scene. In such a scenario, the host application must wait for the processor to finish processing one view to move on to the next viewpoint. This can create a bottleneck situation especially when there is a need for generating multiple views of the scene. This bottleneck created by lack of native multi-view processing for light-field displays in OpenGL is best explained in Table 2.1 and Figure 2.19. The data below shows the rendering performance of an NVidia

GPU GTX Titan X for multi-view rendering seen from a monitor with 60Hz refresh rate. When the number of vertices to be processed increases, we see only negligible difference in the update rate for a single view-point rendering. We can clearly see that multi-view rendering requires significantly more time to update the display, even though the number of pixel operations is roughly the same. This is because the scene must be dispatched and rendered multiple times to update the light-field display once. Even when the number of pixels in both single-view and multi-view renders remain the same, the drop in display updates per second clearly shows that scene transformations for each view creates a major bottleneck situation. This will indeed affect rendering complex scenes, for example, large models with millions of triangles, and will hinder interactivity of the application.

Geometry	No. Vertex	Single view(4M pixel)	Multiview(80x16 views 4M pixel)
1 - Torus	672	60.04	7.90
2 - Torus	4392	59.92	3.55
3 - Prism	468	60.00	9.40
4 - Prism	3468	59.98	3.90

Table 2.1: Display Update Rate Comparison for Single/Multi view Rendering.

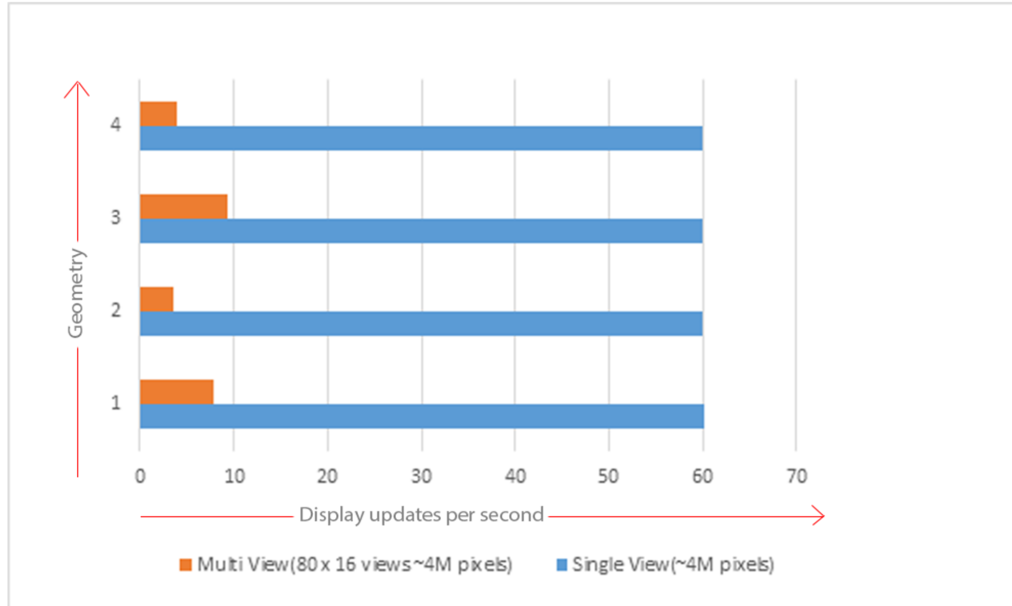

Figure 2.19: DPS Comparison chart for single/Multiview rendering

3. METHODOLOGY AND IMPLEMENTATION

3.1 Heterogeneous Display Environment

A heterogeneous display environment is a client-server rendering system where various 2D/3D displays can connect to a single host application and receive 3D graphics commands; in this environment, it becomes each display or client's responsibility to render graphics in a manner appropriate for that display's projection needs. This setup comes with several challenges that need to be addressed before a truly heterogeneous display ecosystem can be established. Different display manufacturers create and use proprietary solutions for how graphics are generated, transmitted and displayed. This can be a major burden for the manufacturers and users because the content that corresponds to those propriety software standards can only be viewed from the respective displays. But if a consensus can be achieved on developing and using technologies that can cater to multiple displays it can be a mutual win for the manufacturer and the consumer. ObjGL was designed to solve two primary challenges to achieve the heterogeneous display environment.

The first of the two challenges in achieving a heterogeneous display system is addressing how three-dimensional data is stored and transmitted for rendering at the displays. The second is how this dataset is potentially rendered for different displays in the heterogeneous display ecosystem. The rendering process needs to be less coupled with display specific constructs, to abstract any such constraints, and to provide a common interface for writing any application. This abstraction is necessary to create a system where the displays are loosely coupled to the host application and the host application has little to no knowledge of display particulars. This requires that all the displays agree on a common API that defines the scene in a display agnostic manner without favoring any particular display technology. In Figure 3.1 we see a conceptual model of a heterogeneous display

ecosystem with a light field display, a virtual reality headset and an augmented reality goggle.

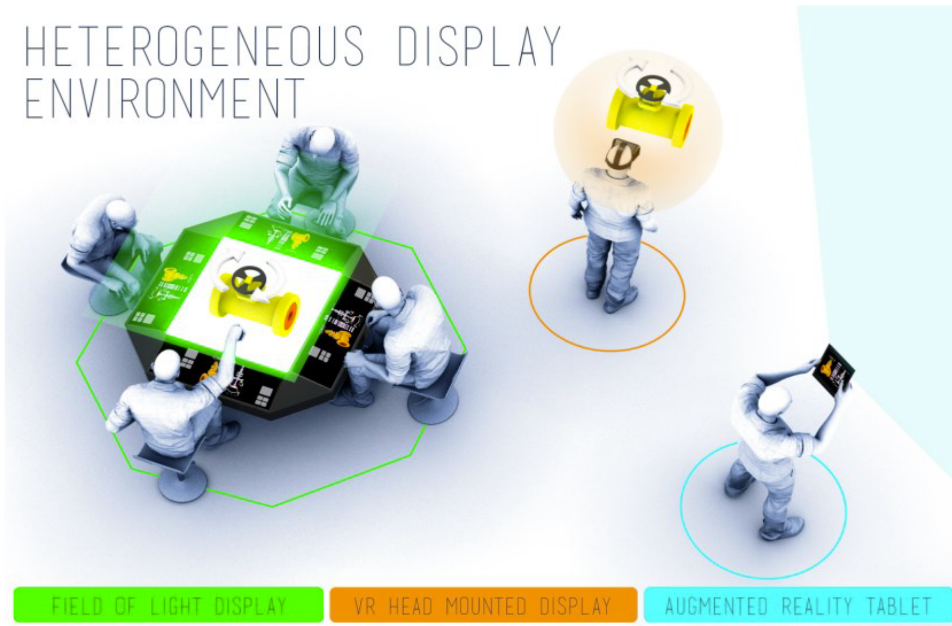

Figure 3.1: A Model Heterogeneous Display Ecosystem

3.2 ObjGL

FoVI^{3D}'s Object Graphics Language(ObjGL) is designed to be a high-level cross-platform display agnostic application programming interface that facilitates 3D rendering on a wide variety of Field of Light Display(FoLD) architectures. ObjGL draws heavily from Open Graphics Language(OpenGL), yet it is streamlined for fast rendering for multi-view display systems. ObjGL is designed to facilitate display agnostic rendering where the rendering host application does not have any knowledge of the connecting client displays.

Versions of OpenGL differ in how the vertex lists of geometry, textures and texture coordinates are defined and stored. For example, in early versions of OpenGL there were

no vertex buffer objects or vertex array objects to store vertex information. Hence, the problem of portability with applications designed for a specific version to run on older computers or devices that support only older versions arises. The same problem can also be said of mobile and embedded devices that use OpenGL ES which has slightly different constructs than their desktop counterparts. ObjGL abstracts the version specific definitions of OpenGL and let the user define input information in a common standard format using custom classes VertexLst, Texture, Material and so on. The device or version specific constructs are handled at the backend on the API level and can be added as and when needed.

ObjGL can render in both first-person view perspective and display-centric perspective. The first-person view perspective renders the scene from a single viewpoint. The first-person view is the normal single reference point of view that we see in 2D monitors and VR/AR head-mounted displays. The display-centric view volume perspective defines a position and orientation where the scene is rendered from the perspective of the display. Light field displays' radiance image rendering employs display centric view volume definition. It essentially creates an appearance of a window into a three-dimensional object or a scene. The displays connecting to ObjGL host can render either first-person or display centric views based on the display type. In ObjGL, huge emphasis is laid on the ease of plug and play use in a multi-display environment. In Figure 3.2 we see how ObjGL fits in a heterogeneous display system.

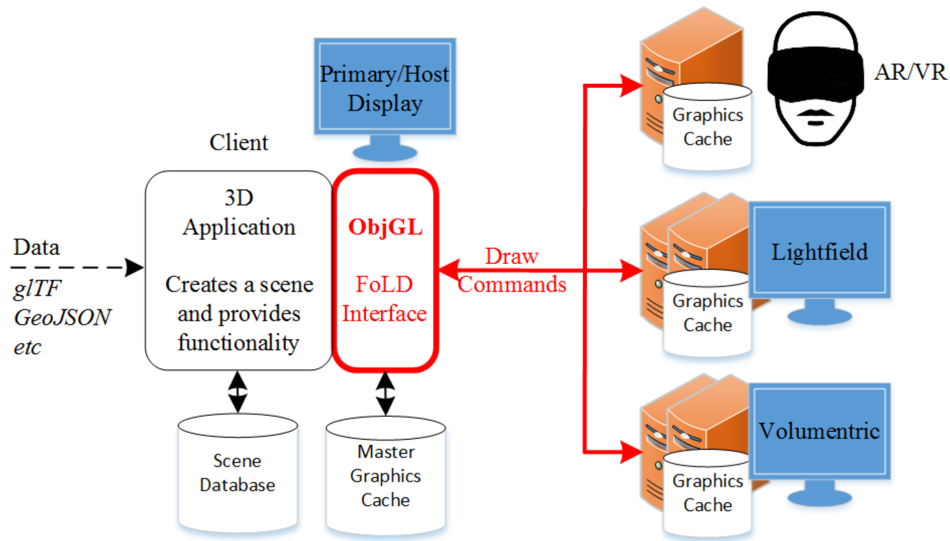

Figure 3.2: ObjGL Rendering Model in a Heterogeneous Display System

ObjGL API is specifically designed to support a customized extreme Multi-view rendering and its instruction set has been classified into three types. The instructions are as shown in Table 3.1

1. *Control*: Instructions used to initiate and delineate frames.
2. *Cache*: Instructions used to cache and remove geometry or textures from the graphics hardware.
3. *Render*: Instructions for controlling viewpoints/view volumes/lights, binding materials and rendering vertex list.

Control	Cache	Render
Clear	Cache Viewpoint	Activate Light
Finish	Cache View Volume	Bind Material
	Cache Light	Render VertexLst
	Cache Material	Deactivate Light
	Cache VertexLst	Bind Texture
	Cache Texture	Look
	Remove Viewpoint	Survey
	Remove View Volume	
	Remove Texture	
	Remove Light	
	Remove VertexLst	

Table 3.1: ObjGL Instruction Set.

3.2.1 Control Instructions

The control instructions Clear and Finish are used for clearing active back buffers and initializing render pipelines; and delineating the end of a render frame respectively. They are called before the start and after the end of rendering respectively.

3.2.2 Cache Instructions

In general, cache instructions are responsible for pre-caching information such as view point, view volume, lights, materials, and vertex lists. A vertex list defines a triangle mesh in optimal form using vertex, normal and texture coordinates for each vertex. A "VertexLst" encapsulates a group of "Vertex" Objects as shown in Figure 3.3. Material and light properties include ambient, diffuse, and specular attributes to implement basic Blinn-

Phong lighting Model. The position of the light is also cached along with the reflective properties.

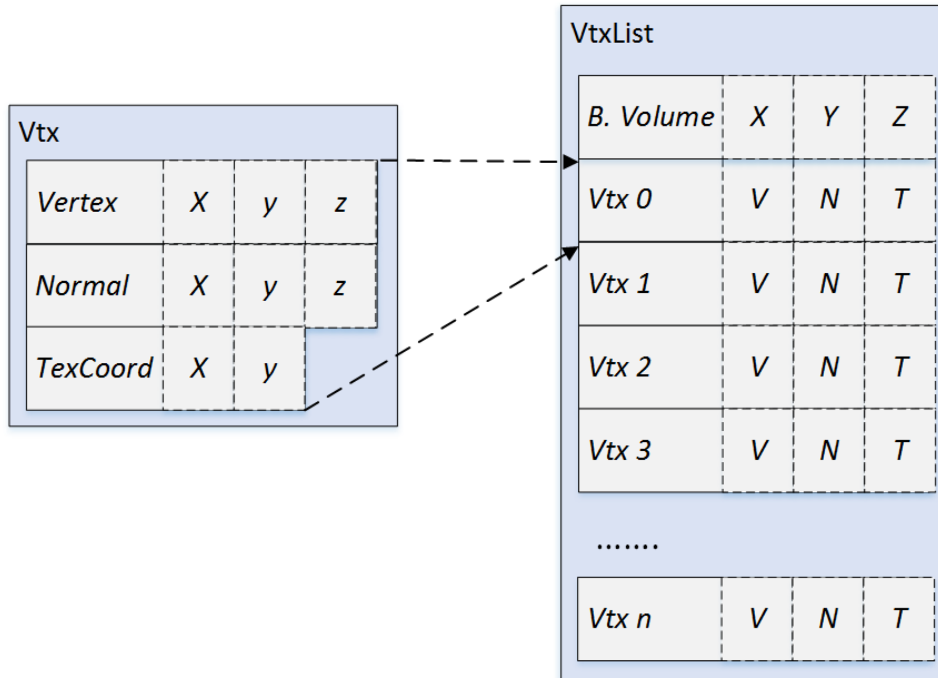

Figure 3.3: Vertex List Definition

The removal and update of these render variables is also grouped under Cache instructions. The remove instructions clear these objects from the memory after rendering.

3.2.3 Render Instructions

The render instructions are responsible for enabling the actual rendering process by activating/deactivating the light, binding the material, and rendering the vertex lists.

Cache and Render instructions cannot be mixed and are delineated by Control commands. This allows for the render hardware to better manage cache updates during a render cycle. The applications running ObjGL would cache geometry, textures, and lights in ad-

vance of rendering a frame. The cache cannot be updated during a render cycle. If there are defined cache and render states, the host can cache the geometry definition and as long as only asset transforms are updated, the renderer can ignore transform updates while rendering the current scene. This allows the host app to update independently of the updates to the connected client displays, effectively allowing displays to update separately while rendering the same content. As it can be seen from Figure 3.4, ObjGL instructions are ordered in such a way that pre-caching of materials, geometry, and lights happen outside of the control instructions.

Cache Material
Cache Vertex List
Cache Light
Cache Viewpoint
Cache View Volume
- **Clear**
  - **Activate Light**
  - **Bind Material**
  - **Render Vertex List**
- **Finish**

Cache Material2
Cache Vertex List2
Update Viewpoint
- **Clear**
  - **Activate Light**
  - **Bind Material**
  - **Render Vertex List**
  - **Deactivate Light**
  - **Bind Material2**
  - **Render Vertex List2**
- **Finish**

Remove Material
Remove Vertex List
Remove Light
Remove Viewpoint
Remove View Volume

Figure 3.4: Example ObjGL Instruction Set Ordering

3.2.4 ObjGL Shapes

Using the ObjGL generic classes such as Vertex and VertexLst, and cache instructions, a display-agnostic specification of geometry information including vertices, normal and texture co-ordinates are created and stored in the class members. This display agnostic specification abstracts the low-level display/version specific implementation and lets the user write applications without the need to worry about portability or future deprecation possibilities. Several shapes such as gear, pyramid, prism, cube, and torus and so on were created using simple linear algebra and trigonometry as shown in Figure 3.5.

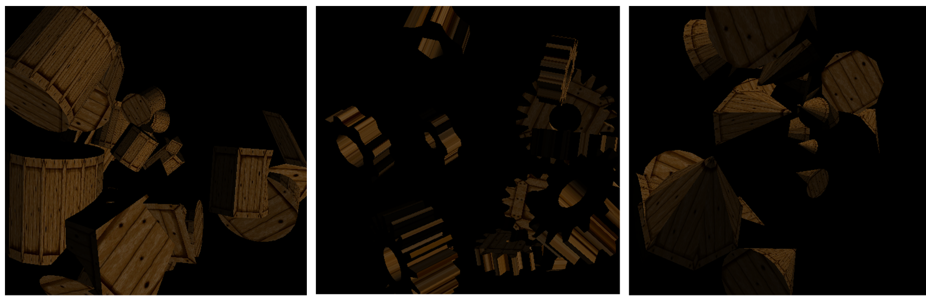

Figure 3.5: ObjGL Shapes - Prisms, Gears, and Pyramids

3.2.5 ObjGL High-level Rendering Acceleration Constructs

ObjGL is being developed with several constructs and constraints to simplify the geometry definition and accelerate render rates for multi-view displays. Bounding volumes, foreground and background segmentation, dynamic/geometric material level of detail, data phasing, stream protocol, critical object and representation of regular meshes are some of the proposed speed-up mechanisms to help efficient multi-view rendering. While most of these acceleration techniques can apply for all kinds of displays and rendering, some of

these are exclusive or work very well for light-field displays, typically because a scene is rendered top-down or for god's eye view(seen from above looking down) perspective.

3.2.5.1 Bounding Volumes and Frustum Culling

Bounding Volume is a closed volume in the shape of a cube or sphere computed for each or group of geometric entities. The minimum and maximum extents of the geometry is computed in the model space from the list of vertex coordinates. It is essential to cull the potentially unnecessary geometry for a given view. Frustum culling is the process of determining whether a chosen geometric object or bounding volume will appear in the viewport. With the view and projection matrices, a viewing frustum with near and far Z, top, bottom, left and right bounds is determined and tested against each geometric object or bounding volume. If the object or the bounding volume fails the frustum culling test, then the set of vertices from the object is not sent to the GPU for rendering. Frustum culling can significantly reduce the rendering time when there are multiple objects with a large number of vertices. This technique can also slow down frame rates especially when there are not a lot of objects in the scene. The operations involved in calculating the frustum for relatively smaller scenes can out-run the benefit gained by skipping few objects from rendering. Nevertheless, this technique can be used in both first-person and multi-view rendering

3.2.5.2 Critical Objects

In light field rendering, some of the objects in the scene may be less significant from a god's eye perspective. For example, the wheels on a vehicle may not be as critical as the body of the vehicle from god's eye perspective and hence can be ignored to reduce the complexity of the rendered scene. Based on the available bandwidth and rendering power, the displays can choose the critical objects that are to be rendered. This is done with the help of assigning priorities or criticality to the primitives, and then the displays choose

or leave the low priority primitives when the rendering power is relatively low for the content being rendered. This technique may not work well for virtual reality displays or any other first-person view displays, but stereo-rendering or single view-point rendering are less likely to be affected much without criticality, compared to light-field rendering which generates multiple views.

3.2.5.3 Foreground and Background Segmentation

In applications such as battle space or sports visualization, the background in the scene may not change as frequently as the objects in the foreground. This gives an opportunity to segment the geometry into foreground and background assets and the background can be rendered only when the background objects are changed or the view-volume definition is changed. This segmentation of objects can be significantly beneficial in multi-view rendering when the view volume remains static.

3.3 Implementation

3.3.1 Virtual Reality Rendering

For this research, OpenVR API is used to connect and talk to HTC Vive and Oculus Rift drivers. A SteamVR room setup must be run to set the base stations and the HMD must be calibrated for the room for proper tracking. In virtual reality, two frames are rendered, one for each eye, with two virtual cameras separated by the distance between two eyes known as the inter-pupillary distance. With head tracking using sensors and base stations, the position and orientation of these virtual cameras are determined inside the virtual environment. From these two viewpoints, the appropriate view and projection matrices are generated and the rest of the graphics transformation pipeline tasks are carried out sequentially.

These two images are rendered onto the framebuffer memory in the GPUs and then submitted to the VRCompositor, which then relays the image to the HMDs. The compos-

itor takes care of some of the post processing work that needs to be done on the image before it is projected in the display. This includes applying a barrel distortion to account for the pincushion distortion caused by the near-eye lenses that affects the corners of the image. The compositor also does color manipulation operations like gamma correction at the command of the user.

3.3.2 OpenVR

Open Virtual Reality or OpenVR is an application programming interface developed by Valve to support Virtual Reality Rendering to multiple vendors' VR displays. The main advantage of using OpenVR is that the API provides a way to interact with different virtual reality headsets without relying on hardware specific software development kits [15]. Currently OpenVR supports stereo rendering for the HTC Vive and Oculus Rift Figure 3.6.

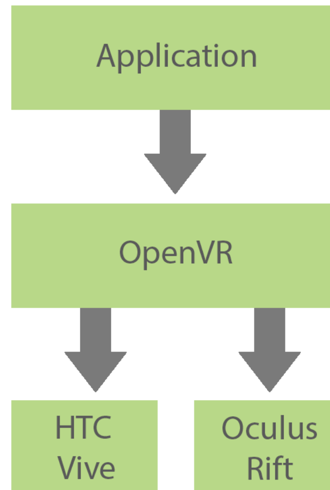

Figure 3.6: OpenVR

The OpenVR API is implemented as a set of interface classes that contains pure virtual functions. These pure virtual functions act as interfaces for the hardware specific SDK and abstracts them directly from the user or the application. They return the matching and appropriate header calls for specific actions as directed by the programmer. OpenVR has interface classes that can interact with all VR accessories including the headsets and touch controllers. This gives a great advantage for the application programmers to develop once and play for all the VR devices. OpenVR requires Valve's SteamVR client running at the back to support HTC Vive and in addition requires Oculus runtime running at the background to support Oculus Rifts. The OpenVR API is primarily broken down into six interfaces. They are

1. IVRSystem
2. IVRChaperone
3. IVRCompositor
4. IVROverlay
5. IVRRenderMod
6. IVRScreenShots

IVRSystem is the main interface that provides access to display configuration information, distortion, tracking data and controller access. IVRCompositor interface is responsible for displaying images to the user, taking care of distortion and synchronization issues to provide the user a solid virtual reality experience. For this research, IVRSystem and IVRCompositor interfaces will only be used and how their functionalities are used in the scope of this research are discussed in the upcoming sections.

3.3.3 ObjGL Host and Render Clients

The ObjGL host application defines a three-dimensional scene within a view volume. In the host application, the defined cache and render instructions that are enabled for forwarding/broadcasting to clients, are queued as ZeroMQ messages. The essential communication between the host and the client is established by the ZeroMQ library. ZeroMQ is a message oriented middleware library used in distributed or concurrent applications. ZeroMQ is used to relay drawing and rendering commands from the ObjGL host application to each display. The ObjGL host packs geometry, texture, light and material data into a ZMQ message stack. This stack is serviced by a separate thread managed within and behind the ObjGL interface. On the client side, the messages are parsed to retrieve the data and then each of the individual clients render them appropriately.

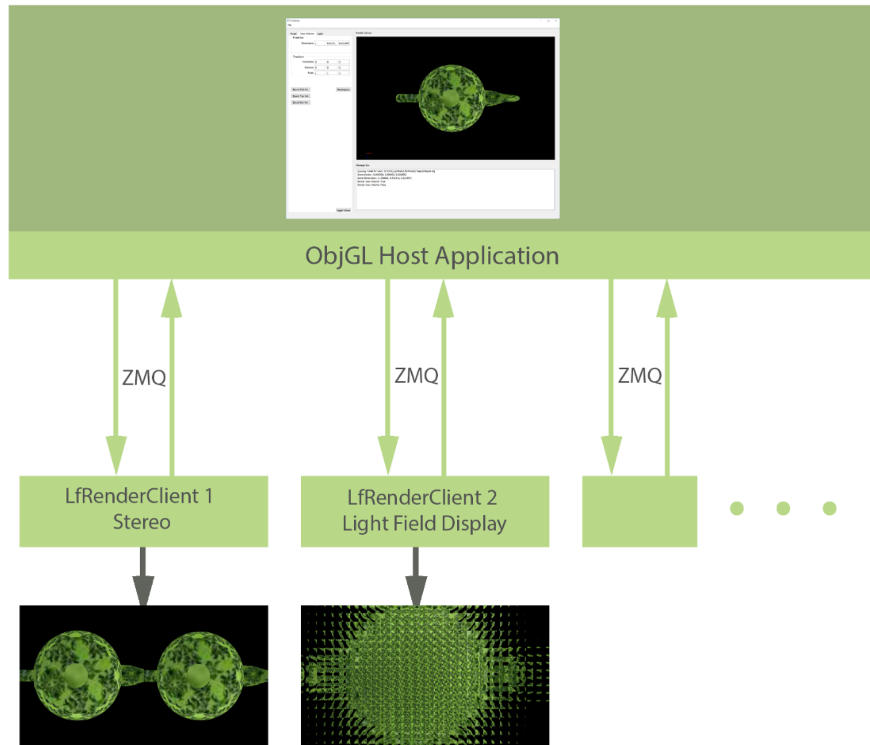

Figure 3.7: ObjGL Client-Server Render flowchart

Render client interfaces are derived from a common parent interface "Renderer." Render clients define the necessary render states and sets the appropriate viewing parameters. Renderer receives a list of cache/render commands from the host application, parses and queues the instructions and then the clients execute them in a display specific manner for the connected single or multi-view display. In Figure 3.7, we see a list of render clients implemented by FoVI^{3D} for multi-view simulations and different multi-view displays including stereo head-mounted displays, double-frustum light field displays and oblique slice displays.

3.3.4 ObjGL Thread Model

In ObjGL, the client-side rendering operation is comprised of two major functionalities, the communication control, and the render control, which are served by three threads as shown in the Figure 3.8. Communication control is serviced by two threads, the subscriber thread and the requester thread. The render control task is managed by the main thread, which talks to the communication control to receive commands and manages the rendering process. The subscriber thread receives broadcast messages from the ObjGL host and feeds it to communication control, which then parses the messages into cache instructions(cache/state commands) and render instructions(bind/draw commands). The main thread manages the display frame render loop by updating the cache/state, renders all the views as defined by the client and swaps the buffer for continuous rendering. When the client cache update is out of sync with the host, an out-of-band cache update request is triggered. The requestor thread receives the out-of-band cache updates from the ObjGL host and queues the local geometry cache update.

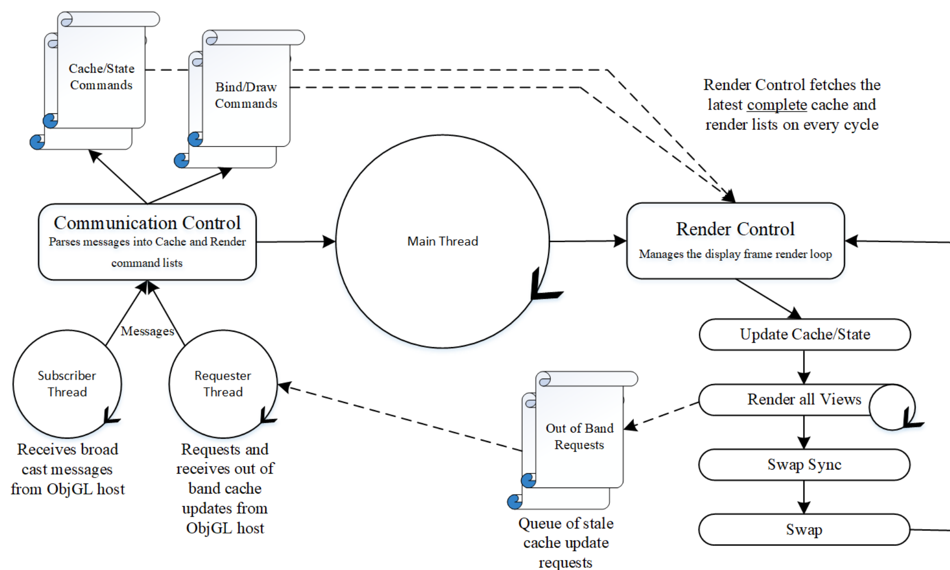

Figure 3.8: ObjGL Client Render Model

3.3.5 ObjGL Forwarding

The host application running on ObjGL should enable forwarding draw and render commands so that the clients can connect, receive and render the scene. This is done using the enable forwarding option in ObjGL.

```
LfD::OGL::Interface ObjGL;
ObjGL.enable(LfD::OGL::Interface::Forwarding);
..
ObjGL.disable(LfD::OGL::Interface::Forwarding);
```

The render commands between enable() and disable() are forwarded to the clients for rendering. The enable() and disable() functions set the "Forwarding" flag to ON and OFF respectively.

The main objective of the heterogenous display system is for the displays to be able to connect to an ObjGL host and render content simultaneously either from a local desktop or from any remote location through server. To do this, ObjGL introduces render clients known as Executors.

3.3.6 RenderStereo

A pointer instance of base class "Renderer" will be created and dynamically allocated to the calling client based on the information from a user defined configuration(.config) file. The .config file is a JSON file that has display particular details, server address and rendering information such as type of display, render resolution and so on(Figure 3.9 RenderStereo JSON file). With this information, the appropriate render instance for the calling client is created using dynamic polymorphism.

```json
{
  "ComMan": {
    "HostAddr": "127.0.0.1",
    "HostPort": "5557"
  },

  "RenCon": {
    "Renderer": "Stereo",
    "WindowDim":  [1280, 720],
    "FrameBufferDim":  [1280, 720]
  }
}
```

Figure 3.9: RenderStereo JSON file

RenderStereo is implemented as a derived class of "Renderer." When RenderStereo client is run, an instance of "RenderStereo" is created and assigned to a base class pointer of "Renderer." Init() and Render() are the two main routines under RenderStereo that take care of the initialization of OpenVR session elements, rendering and submitting the rendered frames to the VR Memory.

3.3.6.1 Init()

A virtual function init() initiates an OpenVR Session of type vr::IVRSystem. IVRSystem is responsible for connecting and maintaining the HMD session, Contollers and the Base Station. The recommended rendering target size for each eye is retrieved using the GetRecommendedTargetSize() method. HTC Vive has a default recommended size resolution of 1512x1680, but it can be modified using a super sampling factor from the Steam VR Control Panel Settings. Under the init() function, a compositor object of type vr::IVRCompositor is also created. Compositor is responsible for the relay of rendered

framebuffers to the VR display. The following are the main operations that are done under the init() routine.

```
vr::IVRSystem* hmd = vr::VR_Init(&eError, vr::VRApplication_Scene);
hmd->GetRecommendedRenderTargetSize(&hmdWidth, &hmdHeight);
vr::IVRCompositor* compositor = vr::VRCompositor();

_fbObj[0].create(glm::ivec2(w, h), GL_RGB);
_fbObj[1].create(glm::ivec2(w, h), GL_RGB);
```

After initializing IVRSystem and IVRCompositor objects, two framebuffer memory regions are created. The framebuffer object is created and texture and depth buffers are attached to the framebuffer. The color attachment is created, allocated memory and bound as the target for rendering inside a custom ObjGL framebuffer class as shown in the following snippet.

```
glGenTextures(1,&_cbo);

glBindTexture(GL_TEXTURE_2D, _cbo);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, dim.x, dim.y, 0,
    GL_RGBA, GL_UNSIGNED_BYTE, nullptr);

glBindFramebuffer(GL_FRAMEBUFFER, _fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
    GL_TEXTURE_2D, _cbo, 0);
```

When the RenderStereo client is executed, the init() function will be called and all the above mentioned initializations take place. A rendering window and a console window will be opened. The rendering window will be a blank screen as no rendering operation has started and the console window shows the initialization status messages.

3.3.6.2 Render()

The Render function is split into three main operations: RenderSetup, RenderStereoViews and RenderTearDown. RenderSetup() lays down the basic groundwork setting of enabling required OpenGL states. RenderStereoViews() routine is where the major action takes place. It receives the tracking pose of the head mounted display and converts it into absolute tracking head matrix, retrieves the left and right eye matrices and also the left and right projection matrices.

```
vr::VRCompositor()->WaitGetPoses(trackedDevicePose,
    vr::k_unMaxTrackedDeviceCount, nullptr, 0);
vr::HmdMatrix34_t head = trackedDevicePose
    [vr::k_unTrackedDeviceIndex_Hmd].mDeviceToAbsoluteTracking;
vr::HmdMatrix34_t& mLEye  = hmd->
    GetEyeToHeadTransform(vr::Eye_Left);
vr::HmdMatrix34_t& mREye  = hmd->
    GetEyeToHeadTransform(vr::Eye_Right);
vr::HmdMatrix44_t& mLProj = hmd->
    GetProjectionMatrix(vr::Eye_Left, nearPlaneZ, farPlaneZ);
vr::HmdMatrix44_t& mRProj = hmd->
    GetProjectionMatrix(vr::Eye_Right, nearPlaneZ, farPlaneZ);
```

The matrices received from OpenVR get-matrix calls are of the inbuilt OpenVR HmdMatrixXX_t types, which then need to be converted into GLM format that are readable by ObjGL for further processing and setting the appropriate view and projection matrix stack. Now that these matrices are set, the appropriate render framebuffer is activated. This is

done using the following operation and happens inside a loop that runs once for each eye. The first framebuffer will be activated for the left eye and the processing happens with left view and projection matrices.

```
_fbObj[i].bind();

glm::mat4 cameraToWorldMatrix = headToWorldMatrix * _mEye[i];
glViewport(0, 0, w, h);
c.setView((cameraToWorldMatrix));
c.setPerspective(_mProj[i]);
```

Now that the appropriate framebuffer is bound and the corresponding matrix stack put in place, rendering commands for the first eye are ready to be processed. The pixels values are drawn into the left eye framebuffer and they are submitted to the VR::Compositor.

```
const vr::Texture_t tex = { reinterpret_cast<void*>
    (intptr_t(_fbObj[i]._cbo)),
    vr::TextureType_OpenGL, vr::ColorSpace_Gamma };
vr::VRCompositor()->Submit(vr::EVREye(i), &tex);
```

The above set of operations are repeated for the right matrix by binding the right eye framebuffer and the corresponding matrix stack. The written framebuffer texture for both the eyes are submitted to the HMD using the Submit() routine of VRCompositor. And finally, to display the two rendered frames to the 2D display for monitoring, a framebuffer blit operation is carried out. Blitting copies the contents of the user defined stereo framebuffers into the default draw framebuffer. The viewport is split into half to display the left and right eye images. This avoids a re-render of all the geometry just to be displayed in the 2D screen.

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0, 0, WIDTH, HEIGHT);
glBindFramebuffer(GL_READ_FRAMEBUFFER, _fbObj[0]._fbo);
```

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, GL_NONE);
glBlitFramebuffer(0, 0, w, h, 0, 0, WIDTH, HEIGHT,
    GL_COLOR_BUFFER_BIT, GL_LINEAR);
glBindFramebuffer(GL_READ_FRAMEBUFFER, GL_NONE);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, GL_NONE);

glBindFramebuffer(GL_READ_FRAMEBUFFER, _fbObj[1]._fbo);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, GL_NONE);
glBlitFramebuffer(0, 0, w, h, WIDTH, 0, 2 * WIDTH, HEIGHT,
    GL_COLOR_BUFFER_BIT, GL_LINEAR);
glBindFramebuffer(GL_READ_FRAMEBUFFER, GL_NONE);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, GL_NONE);
```

3.3.7 Heterogeneous Display Rendering in Action

Similarly for other displays, the Renderer client has to be written with init() and render() virtual functions with appropriate initializations and render support as applicable for the displays. Multiple clients can be run simultaneously waiting for render commands from the ObjGL host application. Now when the ObjGL host is executed, the commands for which the "ObjGL::Fowarding" flag is enabled will be sent to the clients for rendering. With the appropriate draw and render constructs under Render() virtual function in place for each of the connected clients, the rendering takes place, resulting in a functional heterogeneous display system. In the Figure 3.10, we see heterogeneous display rendering of a ObjGL host application doing mock battle field simulation in action with a RenderStereo client rendering stereo image for VR and a RenderRadiance client rendering radiance image for Light-field Displays. In the Figure 3.11, another ObjGL host application "TestObjGL" that renders ObjGL shapes rendering light-field radiance and stereo images.

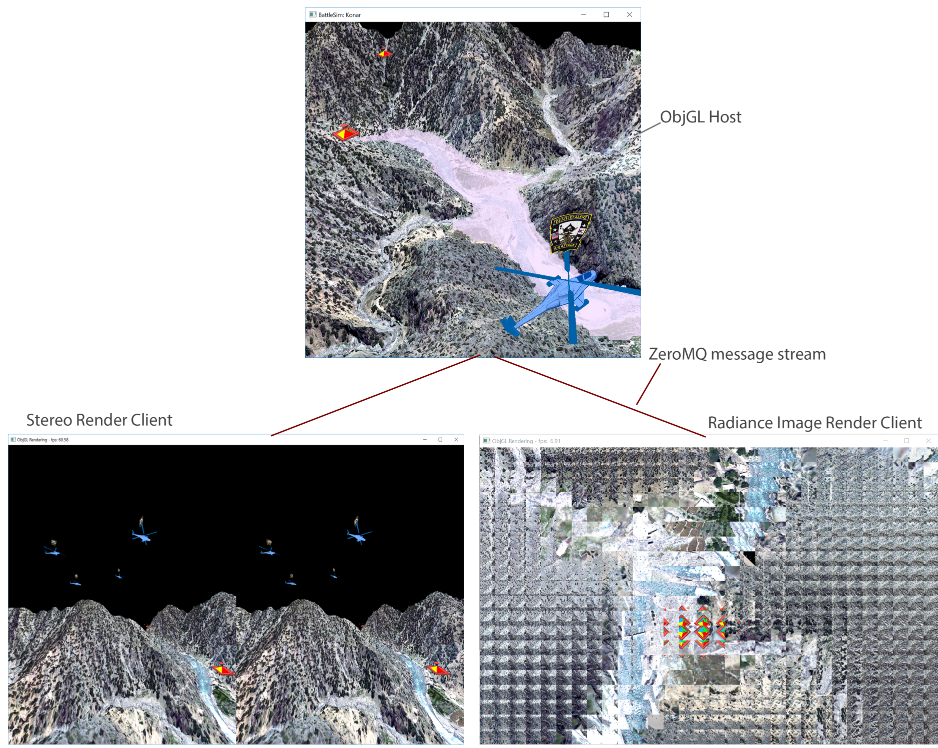

Figure 3.10: Heterogeneous Display Rendering in action - Battle Field Simulation

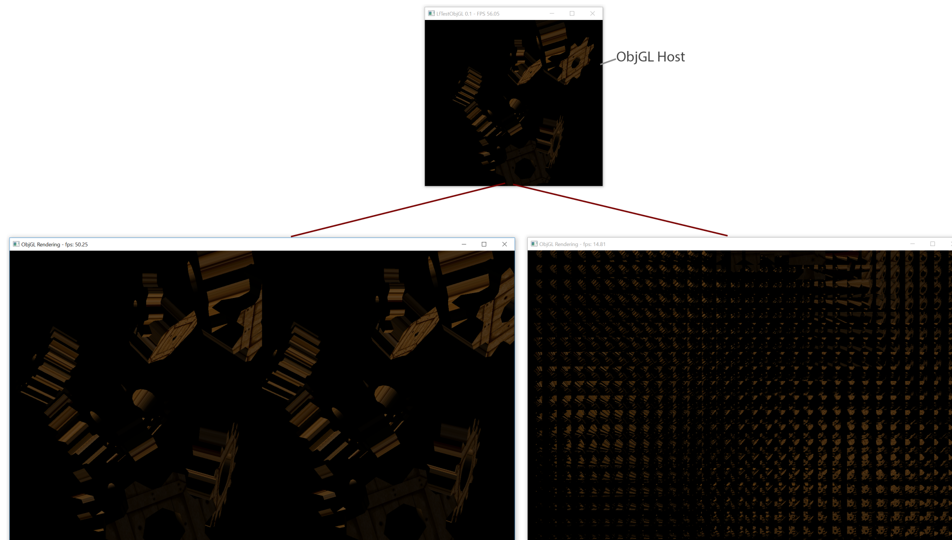

Figure 3.11: Heterogeneous Display Rendering in action - ObjGL Custom Shape Rendering

3.3.8 Test Results

For this thesis, version agnostic ObjGL shape geometry such as prism, pyramid and Torus geometry were created with vertices, normal and texture coordinates using linear algebra and trigonometry in the ObjGL host application "TestObjGL", tested successfully to run with OpenGL versions fixed function 2.0, fixed function 3.0 and shader based 3.3. Virtual reality stereo rendering was tested with both Oculus Rift and HTC Vive with ObjGL clients running in the same host machine and across a network. Oculus DK2 was connected using the sensor mounted in front of the user. HTC Vive was installed with the two lighthouse base stations for tracking the headset within the sensor area. Stereo rendering was tested with all the three ObjGL host applications TestObjGL, ModelViewer and BattleSim successfully.

The heterogeneous display system was also tested across a network with clients connecting to ObjGL hosts through LAN using Wi-Fi and ethernet cables. The ZeroMQ

messages were received as packets across the network and rendered by the client for each display. When tested across networks, there have been sporadic delays or render freeze on either light-field display or VR display or both. Consistency in rendering across the network is being studied and needs improvement for a more robust heterogeneous display system.

4. SUMMARY AND CONCLUSIONS

This heterogeneous display environment enables users to better understand the projected 3D data by viewing them in true 3D displays. It helps the user make well-informed decisions, a crucial factor in disciplines including medical and military visualization. It also aids in reducing the cognitive load of the viewer and augments collaboration between the stakeholders. Adding stereo rendering further aids this effort and collaborators to obtain a first person view of any terrain or virtual environment.

ObjGL also plays an instrumental role in overcoming the fragmentation problem with proprietary and device-specific solutions. When properly scaled as an Open Standard, ObjGL has the potential to disrupt and change how 3D data is stored, transmitted, rendered and visualized in different displays.

4.1 Future Work

While ObjGL is a first step in making a display agnostic rendering system, there is a broad scope for improvements and enhancements. There are different acceleration mechanisms such as foreground/background segmentation, level of detail, and others which can be used to further speed up render rates. With an increased render rate realized through acceleration schemes, interactivity can be achieved in multi-view displays which can open up further opportunities and possibilities for using these displays in gaming, entertainment, and other purposes. In first person stereo rendering, interactivity with touch controllers and navigation inside the virtual environment will be added.

To evolve ObjGL into an Open Standard for multi-display visualization, ObjGL is to be made Open Source to attract multiple collaborators contributing to content development, resulting in incremental architectural sophistication. Other multi-view rendering displays can also be brought aboard with ObjGL to expand the horizons of this heterogeneous

display environment.

One proposed future FoVI^{3D} endeavor is to develop a Multi-view Processing Unit (MvPU), which will be a dedicated multi-view processor designed specifically for light-field displays and multi depth-plane VR displays. MvPU will replace the conventional GPUs or cluster of GPUs used in light-field rendering and thus the bottleneck problem with multi-view point rendering can be overcome. ObjGL will be matured as a software interface to transfer graphics rendering information to MvPU without the need for OpenGL.

REFERENCES

[1] W. Lages, C. Cordeiro, and D. Guedes, "A parallel multi-view rendering architecture," *International Journal of Computer Graphics*, pp. 947–958, 2009. http://ieeexplore.ieee.org/document/4654169.

[2] J. J. Martin, M. Holzbach, J. Riegler, C. K. Tam, and A. Smith, "Evaluation of holographic technology in close air support mission planning and execution," 2008.

[3] E. A.-L. Lee, K. W. Wong, and C. C. Fung, "Educational values of virtual reality: The case of spatial ability," *International Journal of Social, Behavioral, Educational, Economic, Business and Industrial Engineering*, vol. 6, 2009. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.308.8755.

[4] T. Shibata, Y. Ishihara, and K. Satou, "Usefulness of stereoscopic 3d images in elementary school classes," *SID Symposium Digest of Technical Papers*, vol. 46, 2015. http://onlinelibrary.wiley.com/doi/10.1002/sdtp.10312/abstract.

[5] M. Agus, F. Bettio, E. Gobbetti, and G. Pintore, "Medical Visualization with New Generation Spatial 3D Displays," in *Eurographics Italian Chapter Conference* (R. D. Amicis and G. Conti, eds.), The Eurographics Association, 2007. 10.2312/LocalChapterEvents.ItalChap.ItalianChapConf2007.161-166.

[6] P. T. Kovacs, A. Boev, R. Bregovic, and A. Gotchev, "Quality measurements of 3d light-field displays," *Proceedings of the Eight International Workshop on Video Processing and Quality Metrics for Consumer*, pp. 1–6, 2014.

[7] M. Lucente, "Diffraction-specific fringe computation for electro-holography," 1995. https://dl.acm.org/citation.cfm?id=222938.

[8] J. Geng, "Three-dimensional display technologies," *Advances in Optics and Photonics*, vol. 5, pp. 456–535, 2013. https://doi.org/10.1364/AOP.5.000456.

[9] D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 3.0 and 3.1*. Addison Wesley Professional, 2009.

[10] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison Wesley Professional, 2013.

[11] Khronos, "Framebuffers." Web. https://www.khronos.org/opengl/wiki/Framebuffer.

[12] N. A. Dodgson, J. R. Moore, and S. R. Lang, "Multi-view autostereoscopic 3d display," *International Broadcasting Convention*, pp. 497–502, 1999. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7623.

[13] F. D. Sorbier, V. Nozick, and H. Saito, "Multi-view rendering using gpu for 3-d displays," *GSTF Journal on Computing*, 2010. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.662.1059.

[14] T. Burnett, "Light-field display architecture and the challenge of synthetic light-field radiance image rendering," *Society for Information Display*, pp. 899–902, 2017.

[15] "Openvr api documentation." Web, 2015. github.com/ValveSoftware/openvr/wiki/API-Documentation.