

5-2013

## **VizLab: The Design and Implementation of An Immersive Virtual Environment System Using Game Engine Technology and Open Source Software**

Moises D. Carrillo  
*University of Texas-Pan American*

Follow this and additional works at: [https://scholarworks.utrgv.edu/leg\\_etd](https://scholarworks.utrgv.edu/leg_etd)



Part of the [Computer Sciences Commons](#)

---

### **Recommended Citation**

Carrillo, Moises D., "VizLab: The Design and Implementation of An Immersive Virtual Environment System Using Game Engine Technology and Open Source Software" (2013). *Theses and Dissertations - UTB/UTPA*. 726.

[https://scholarworks.utrgv.edu/leg\\_etd/726](https://scholarworks.utrgv.edu/leg_etd/726)

This Thesis is brought to you for free and open access by ScholarWorks @ UTRGV. It has been accepted for inclusion in Theses and Dissertations - UTB/UTPA by an authorized administrator of ScholarWorks @ UTRGV. For more information, please contact [justin.white@utrgv.edu](mailto:justin.white@utrgv.edu), [william.flores01@utrgv.edu](mailto:william.flores01@utrgv.edu).

VIZLAB: THE DESIGN AND IMPLEMENTATION OF AN IMMERSIVE VIRTUAL  
ENVIRONMENT SYSTEM USING GAME ENGINE TECHNOLOGY AND OPEN SOURCE  
SOFTWARE

A Thesis

by

MOISES D. CARRILLO

Submitted to the Graduate School of the  
University of Texas-Pan American  
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2013

Major Subject: Computer Science



VIZLAB: THE DESIGN AND IMPLEMENTATION OF AN IMMERSIVE VIRTUAL  
ENVIRONMENT SYSTEM USING GAME ENGINE TECHNOLOGY AND OPEN SOURCE  
SOFTWARE

A Thesis  
by  
MOISES D. CARRILLO

COMMITTEE MEMBERS

Dr. Richard H. Fowler  
Chair of Committee

Dr. Zhixiang Chen  
Committee Member

Dr. Wendy Lawrence-Fowler  
Committee Member

Dr. Emmett Tomai  
Committee Member

May 2013





Copyright 2013 Moises D. Carrillo

All Rights Reserved



## ABSTRACT

Carrillo, Moises D., VizLab: The Design and Implementation of an Immersive Virtual Environment System using Game Engine Technology and Open Source Software. Master of Science (MS), May, 2013 98pp., 30 figures, references, 44 titles.

Virtual Reality (VR) is a term used to describe computer-simulated environments that can immerse users in a real or unreal world. Immersive systems are an essential component when experiencing virtual environments. Developing VR applications is time-consuming, and developers use many resources in creating VR applications. The separate components require integration, and the challenges in using public domain open source software present complex software development. The VizLab Virtual Reality System was created to meet these challenges and provide an integrated suite of tools for VR system development.

VizLab supports the development of VR applications by using game engine and CAVE system technology. The system consists of software modules that provide rendering, texturing, collision, physics, window/viewport management, cluster synchronization, input management, multi-processing, stereoscopic 3D, and networking. VizLab combines the main functional aspects of a game engine and CAVE system for an improved approach to developing VR applications, virtual environments, and immersive environments.



## ACKNOWLEDGMENTS

I would like to thank my parents, Maria E. Carrillo and Juan M. Carrillo, for all their support throughout my undergraduate and graduate years. They inspired me throughout my life and helped me accomplish my goals.

I would like to express my gratitude to my committee chair, Dr. Richard H. Fowler, for supporting me throughout my graduate years. His never-ending support and guidance with research and development in the visualization lab made this thesis possible.

I would like to thank my peers that helped and supported me throughout my career at The University of Texas Pan-American. I would like to thank my girlfriend, Karla Lopez Bray, for her support and advice when reading my thesis. I thank David Chavez for aiding in appendices validation processes. I thank Raymundo Rivera for assisting with the integration of Bullet Physics into VizLab. Finally, I would like to thank the Department of Defense, Army Research Office, W911NF-11-1-0126 for funding this research.



## TABLE OF CONTENTS

|                                  | Page |
|----------------------------------|------|
| ABSTRACT.....                    | iii  |
| ACKNOWLEDGMENTS .....            | iv   |
| TABLE OF CONTENTS.....           | v    |
| LIST OF FIGURES .....            | viii |
| CHAPTER I. INTRODUCTION.....     | 1    |
| Motivation.....                  | 2    |
| Goals .....                      | 3    |
| Organization of the Thesis ..... | 3    |
| CHAPTER II. BACKGROUND .....     | 5    |
| Hardware.....                    | 6    |
| Display Hardware .....           | 6    |
| Tracking Hardware .....          | 7    |
| CAVE Systems .....               | 8    |
| CAVELib .....                    | 10   |
| VRJuggler .....                  | 10   |
| CaveUT.....                      | 10   |



|   |    |
|---|----|
| Game Engines .....                              | 11 |
| Textures, Lighting and Rendering .....          | 11 |
| Physics and Effects .....                       | 14 |
| Input, Output and Networking .....              | 15 |
| Game Editor .....                               | 16 |
| Game Engines .....                              | 17 |
| Modern Open Source Tools .....                  | 18 |
| OpenSceneGraph and OpenSG .....                 | 19 |
| Bullet Physics Engine .....                     | 20 |
| OsgBullet .....                                 | 20 |
| OsgWorks .....                                  | 21 |
| OsgEdit .....                                   | 21 |
| Virtual Reality Peripheral Network (VRPN) ..... | 22 |
| CHAPTER III. VIZLAB VR SYSTEM .....             | 23 |
| VR Juggler .....                                | 24 |
| Modules .....                                   | 24 |
| Installation and Configuration .....            | 26 |
| Applications .....                              | 34 |
| Rendering and Physics .....                     | 37 |
| GUI Capabilities .....                          | 37 |
| Scene Graph Integration .....                   | 37 |
| Physics and Collision Integration .....         | 39 |
| Composing Scenes .....                          | 40 |

|                               |    |
|-------------------------------|----|
| Device Management .....       | 41 |
| Device Integration.....       | 41 |
| VRPN Integration .....        | 43 |
| Motion and Gestures.....      | 44 |
| Microsoft Kinect.....         | 44 |
| Nintendo WiiMote .....        | 45 |
| CHAPTER IV. CONCLUSIONS ..... | 47 |
| Conclusion .....              | 47 |
| Future Work.....              | 48 |
| REFERENCES .....              | 51 |
| APPENDIX A.....               | 55 |
| APPENDIX B.....               | 59 |
| APPENDIX C.....               | 76 |
| APPENDIX D.....               | 93 |
| BIOGRAPHICAL SKETCH .....     | 98 |



## LIST OF FIGURES

|   | Page |
|---|------|
| FIGURE 1: CAVE SYSTEM.....                          | 9    |
| FIGURE 2: LEVEL OF DETAIL.....                      | 12   |
| FIGURE 3: STATIC LIGHTING.....                      | 13   |
| FIGURE 4: EFFECTS ENGINE.....                       | 14   |
| FIGURE 5: UDK GAME EDITOR.....                      | 17   |
| FIGURE 6: VIZLAB SOFTWARE ARCHITECTURE.....         | 24   |
| FIGURE 7: VRJUGGLER MODULES.....                    | 24   |
| FIGURE 8. VRJUGGLER DEPENDENCIES.....               | 28   |
| FIGURE 9: WINDOWS SYSTEM PROPERTIES.....            | 29   |
| FIGURE 10: ENVIRONMENT VARIABLES.....               | 29   |
| FIGURE 11: CMAKE GUI.....                           | 29   |
| FIGURE 12: VIZLAB WINDOWS INSTALLATION.....         | 30   |
| FIGURE 13: VIZLAB WINDOWS INSTALLATION PROCESS..... | 30   |
| FIGURE 14: UBUNTU PACKAGES.....                     | 31   |
| FIGURE 15. VIZLAB SYSTEM LAYOUT.....                | 33   |
| FIGURE 16: VRJCONFIG.....                           | 34   |
| FIGURE 17: VRJCONFIG PANEL.....                     | 34   |
| FIGURE 18: OPENGL APPLICATION.....                  | 35   |
| FIGURE 19: OPENSCENEGAPH APPLICATION.....           | 36   |

|  |    |
|--|----|
| FIGURE 20: OPENSCENEGAPH AND BULLET APPLICATION.....           | 36 |
| FIGURE 21: OPENSCENEGAPH APPLICATION RUNNING IN VRJUGGLER..... | 38 |
| FIGURE 22: OPENSCENEGAPH WITH BULLET PHYSICS ENGINE.....       | 40 |
| FIGURE 23: OSGEDIT SAMPLE SCENE.....                           | 41 |
| FIGURE 24: VRJCONFIG DEVICE LAYOUT.....                        | 43 |
| FIGURE 25: VRJCONFIG DEVICE PROXIES.....                       | 43 |
| FIGURE 26: VRJCONFIG VRPN SERVER SETUP.....                    | 44 |
| FIGURE 27: FFAST GUI.....                                      | 45 |
| FIGURE 28: CWIID INTERFACE.....                                | 46 |
| FIGURE 29: VRJCONFIG AND OSGEDIT.....                          | 49 |
| FIGURE 30: OPENSCENEGAPH & BULLET.....                         | 50 |

## CHAPTER I

### INTRODUCTION

Virtual reality (VR) is a term used to describe computer-simulated environments of a real or unreal world. Physically immersive displays and interaction techniques are often employed to enhance users' experiences in virtual environments. VR systems are often intricate systems built using complex software and hardware systems and provide the experience of telepresence<sup>[2]</sup>. These systems are widely used in visualization, graphics, immersion, and presence research. Among the system elements that must be integrated are display system, tracking system, data management, networking, and scene management.

This thesis focuses on the utilization of use of open source software to create an immersive system development called the VizLab Virtual Reality System. The system supports the development of virtual reality applications and immersive games by using a combination of game engine methodology and CAVE system technology. This provides the foundation modules for virtual reality system that can be configured and developed primarily through a graphic user interface (GUI). The VizLab system consists of software modules that provide rendering, texturing, collision, physics, window/viewport management, cluster synchronization, cluster data sharing, input management, multi-processing, stereoscopic 3D, and networking. This thesis will demonstrate how the VizLab Virtual Reality System combines the main functional aspects of a

game engine and CAVE system to serve as a basis for more efficient development VR applications, as well as virtual environments (VEs), and immersive environments (IEs).

### **Motivation**

My interest in computer science lies in human computer interaction and gaming. I have always been interested in computer graphics, VR, VEs, and measuring presence and immersion. However, the Department of Computer Science at The University of Texas – Pan American (UTPA) had yet to develop a CAVE system. During my undergraduate years I was part of a group of students that were highly interested in games, graphics, and artificial intelligence. We combined our passion for gaming technologies to create small games in hopes of having game development and visualization courses started at UTPA.

As I entered the Master's program at UTPA, Dr. Fowler introduced me to immersive and interactive technologies. I began research in this field and became devoted to learning more about the concepts of presence and immersion within virtual reality literature. At this time Dr. Butler and Dr. Fowler received funding from the Department of Defense that allowed them to build a relatively low cost CAVE system at UTPA. This provided an environment for visualization and computer graphics applications. Given the system's budget, a commercial CAVE system was not a viable option, due to the extremely expensive hardware and software required.

The goal was set, and our challenge was to create a relatively low cost (\$120k) virtual reality system. Our approach was a practical one: use open source software, mid-level professional computer graphics hardware, robust commodity workstations, and recently available stereoscopic projection systems targeting the home theater market.

## **Goals**

The goals of the project center on design and implementation of an immersive system that uses open source tools as the foundation for the efficient development and maintenance of VR, VE, and information visualization (IV) applications. The use of open source software allows the cost of the system to be lower and ensures that it can be maintained by future students interested in these research fields. As with game engine technologies, the system should provide foundation elements such as physics simulation, model rendering in a variety of formats, collision detection, and scene graph implementations.

A fundamental goal is that system be reconfigurable and extensible for both hardware and software components. Because the system will be used for future research in VR, VE and IV applications, it is essential to have a configuration that will provide dependable performance and information. Having realistic simulation involves having dependable hardware and software features. For many application the more realistic the simulation is, the greater the immersion the user will experience<sup>[5]</sup>. In addition, the system should remain stable when adding new hardware or software, since computing and graphics hardware equipment capabilities continually increasing. Finally, the system should provide essential documentation of installation, configuration, and basic standard applications.

## **Organization of the Thesis**

The following sections describe the background, design, implementation and material used to create VizLab. First, a brief introduction describing the background of immersive systems will cover hardware and software used in deployment of such systems. Next, game engine technology is described and an overview of game engine modules followed by open



source software and its incorporation into immersive systems will be covered. Then, VizLab software suite and its components will be described and explained. Finally, the conclusion will provide an overview of the system and future work that could enhance development of VR applications.

## CHAPTER II

### BACKGROUND

The Ultimate Display by Ivan Sutherland is often considered the starting point of immersive systems<sup>[37]</sup>. It consisted of a head mounted display constructed from two cathode ray tubes and a mechanical arm with early head tracking capabilities<sup>[36]</sup>. The system was heavy, and the mechanics required it to be suspended from the ceiling. The graphics consisted of simple wire frame models shapes and environments. Sutherland was pioneering and engineering a visualization system, and in his work he describes the capabilities of a computer system modeling a real world environment. He had the insight that computers were not just capable of arithmetic and calculations, but also capable of simulating real world environments through the use of virtual environments. Although the computing power and display technology at the time were not powerful, he was able to accomplish his mission of portraying a virtual world using wire frame objects and shapes with simulated physics and forces. Immersive systems and gaming software emerged in the years following this landmark work.

Immersive systems such as Cave Automatic Virtual Environment (CAVE) systems, head mounted display systems, and simulators are complex systems with software and hardware modules working together to create an immersive experience<sup>[10][11]</sup>. Such systems can be traced to the beginnings of computer graphics and are rooted in developing scientific visualizations and scientific research for IV, VE, IE, and VR fields. Most immersive systems provide robust

execution through separate modules to help the development of immersive applications. This approach can also be seen in game engine technologies that have the same modular approach <sup>[35]</sup>. As games have become more popular, gaming companies evolved such as Epic, Bioware, Bungie, Crytek and Blizzard. These companies have created the gaming technologies and game engines that we know today.

## **Hardware**

### **Display Hardware**

Head mounted displays (HMDs) are a type of display device that has gone through a series of changes since first used by Sutherland. Significant advances in HMDs have occurred in resolution, color, brightness and design. The design has evolved from cathode ray tubes (CRT) to LCD and OLED displays. However, the design still has the fundamental field of view constraint that typically limits the user to a 25-45 degree viewing angle <sup>[36]</sup>.

Projection based displays provide many options and can be configured flexibly when building a VR system. One can use multiple projectors to enhance visibility and provide multiple projections of a scene or simulation. Using panoramic projectors to display a virtual environment enhances visibility and field of view for the user. The use of multiple projectors can simulate the panoramic view by integrating display pipelines and merging the displayed projections together to provide a panoramic view <sup>[11][19]</sup>.

The use of workbench projector can also be used in a VR system. A workbench projector uses a rear projection screen flat like a table and the projector uses an offset length close to the length of a human body. The user is tracked and views an image on the screen. These types of

setups mimic the setup of a drafting table and can be used with multiple sensors and equipment that allow the user to interact with the virtual environment <sup>[42]</sup> <sup>[44]</sup>.

The use of desktop can also be used to create a VR system. With today's graphics cards and technology, it is manageable to connect multiple computer displays to a single computer and have them set up by the operating system. No special software requirements are needed besides the built in software and drivers that come with the graphic cards. Although this is one of the cheapest solutions for creating a wide field of view such as used in immersive environments, it has many disadvantages. Computer monitor displays do not fully surround or conceal the real world. The objects are rendered to the display size and therefore do not fully represent the object in real size, preventing the user from becoming fully immersed in the virtual world <sup>[44]</sup>.

Limitations of desktop size can be addressed in a limited manner through the use larger flat panel displays, such as used in commodity televisions.

### **Tracking Hardware**

Tracking hardware can be used to provide an interface that allows the user to manipulate objects and for the system to track the user within the virtual environment. There are different methods to track user's actions and show them through software into the virtual environment. One method is by using magnetic sensors to track the user is by using two magnetic sensors to track a user within a limited range. The user has stay within a perimeter in order for the sensors to track the user. Furthermore, the sensors might receive interference from other electronic devices, since most electronic devices create magnetic fields, it may affect the sensors depending on the magnetic sensitivity <sup>[44]</sup>.

The use of cameras for tracking has become an effective method for tracking. Along with the motion detection software to track the individual, having more cameras provides better accuracy of the tracking system because more angles can be covered and detected. Accuracy also depends on the camera resolution; having a higher resolution camera will increase the system's mapping accuracy in a VE <sup>[33]</sup>.

A data glove is a device that allows the user to manipulate virtual objects or graphical data within the VR system. Data gloves can motion sensors and accelerometers embedded within, the user places the gloves on his or her hand and can directly manipulate the objects. The software then maps the motion and direction of the glove and portrays those coordinates to a VE <sup>[33]</sup>.

The VR community has adopted gaming peripherals and ported them into VR systems. For example, using a Nintendo Wii wireless controller allows a measurement of actions using the controller's accelerometer and gyroscopic sensors. Gaming devices can provide an inexpensive solution to high-end motion tracking devices <sup>[16][42]</sup>. The use of Microsoft Kinect also provides an inexpensive alternative to expensive camera tracking system and allows for full skeleton tracking <sup>[16]</sup>. Also, much open source software is available allowing the programming needed to adapt the device to a VR system.

### **CAVE Systems**

A CAVE is an immersive system that consists of multiple projectors with each projector displaying a part of a virtual environment to surround the user with the visual elements of a simulation. Its origins are in the use of visualization and scientific research. CAVE designs vary, but usually exhibit a variation of the original wall centric design <sup>[10][11]</sup>. A CAVE can vary from a

V shape design, two walls at a 90 degree angle, to a fully enclosed CAVE system surrounding a user, a cube like structure, as shown in Figure 1. Although the system may be front projected or rear projected system, but most setups use a rear projection design.

CAVE systems are run by different software systems operating in unison to provide an integrated system. Most CAVE software platforms consist of a multi-display system, networking system, cluster management system, and input/output system.

An advantage to using a CAVE as a VR system is that the system provides a wide field view to the user or multiple users. This allows the user the freedom to walk around within a parameter, making it a conventional way to engage a user within a VR application. The main disadvantage of CAVE system is that it is composed of various systems; the rendering of the scene is done by multiple computers, one computer rendering one part of the scene. The computers have to work together to bring the scene together and this brings forth other problems that need to be handled by either software or hardware systems<sup>[19]</sup>.

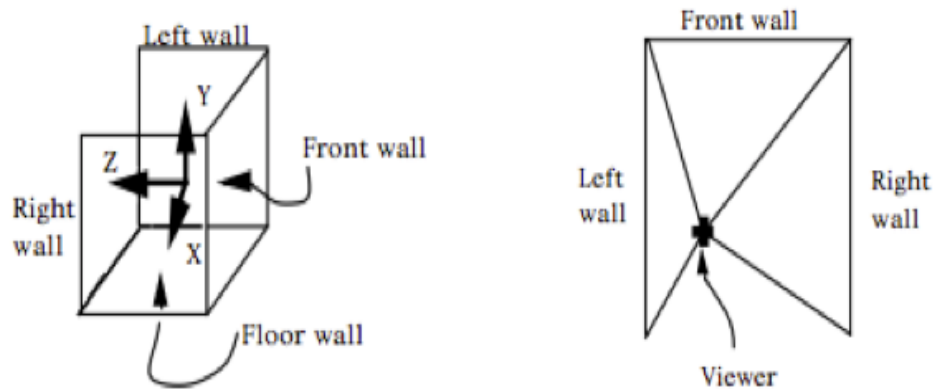


FIGURE 1: CAVE SYSTEM<sup>[11]</sup>.

## **CAVELib**

CAVELib is the original software platform used in the first CAVE system and was designed to run on specific hardware. The system provides essential elements needed to create a virtual reality system and provides support for multi-display, cluster design architecture, and interaction device integration. At the time, the system provided an open source alternative to expensive commercialized simulators making it a widely used platform. The system was later upgraded by the research community to support a variety of platforms and architectures and commercialized by Mechdyne, making it a licensed software platform <sup>[11]</sup>.

## **VRJuggler**

VRJuggler is an open source CAVE platform developed by Iowa State University that provides a package that combines different APIs to simplify the VR application development process <sup>[13]</sup>. VR Juggler is built upon open source software libraries and is platform independent. It provides the developer with the freedom to write an application that will work with many different display and input devices without altering code or recompiling the application. It provides a GUI for window and viewport management. It uses OpenGL standard for low level application, and integrates OpenSceneGraph and OpenSG for rendering complex scenes and virtual environments <sup>[1][40]</sup>.

## **CaveUT**

CaveUT is an open source extension of the Unreal Tournament 2004 game engine. Developed by PublicVR, CaveUT takes advantage of gaming technologies to create a CAVE environment. It uses Unreal Tournament's game engine to design virtual environments and scenes. Within the game engine there is a spectator function that CaveUT utilizes to mimic real

life viewpoints and create virtual viewpoints around the player's head. Each viewpoint is rendered separately and gives the illusion of a 3D environment when projected onto separate walls <sup>[19] [23]</sup>.

## **Game Engines**

Formerly, video games used integrated purpose based rendering and display software as opposed to separate interchangeable modules used in today's video games. In 1993, the video game Doom introduced a new game-programming standard; its design architecture was based on the "game engine" concept. The game engine has become the foundation software module that can be used to create new versions of the original game or even all-new games <sup>[23]</sup>. Today's game engines consist of a variety of different extensible modules that handle textures, lighting, rendering, physics, effects, input, output, networking and game editor. These are discussed in the following sections.

### **Textures, Lighting and Rendering**

The procedure in which polygon objects are given detail by the application of images to their surfaces is called texturing. Textures can provide the illusion of realistic objects, making the scenes seem more real. However, a disadvantage in adding realistic features to a scene is that the textures will occupy more memory in the system. Game engines and other computer graphics approaches provide ways to minimize and reduce texture memory space in order to provide better performance while playing a game. An example of optimization is the use of MIP mapping textures within a scene. This allows the minimization of texture file size. The theory behind it is that as the camera view moves away from an object, the object's texture resolution will be reduced and therefore increase performance. With the MIP mapping technique textures are



rescaled and saved at different resolutions ahead of time in order to avoid texture distortion as the resolution is decreased when the distance between the object and the camera is increased<sup>[24][26]</sup>.

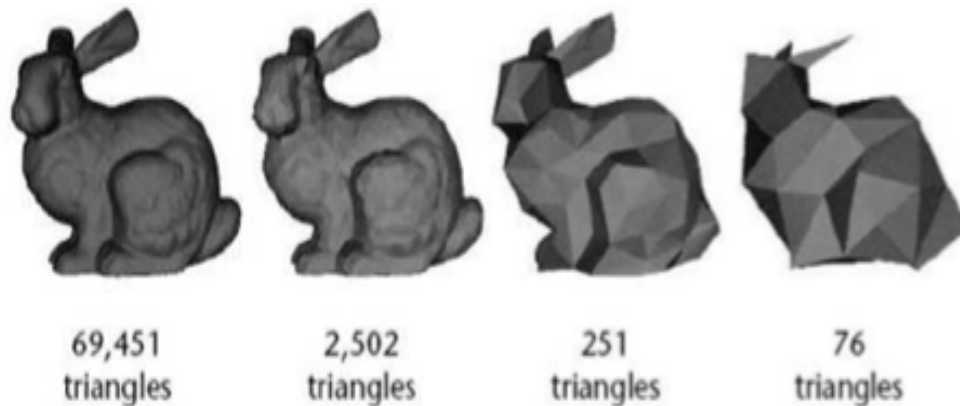


FIGURE 2: LEVEL OF DETAIL <sup>[14]</sup>.

This is analogous to the level of detail (LOD) approach used in some game engines <sup>[15]</sup>. LOD systems use algorithms to scale the geometric detail of object models based on the camera distance within the scene. They support the ability to alter the polygons used to render a model within the camera view. In advanced LOD systems, different render models are used depending on the distance between the object and the camera view or dynamically decrease the polygon count in real time.

There are two different forms of lighting within a scene, static lighting and dynamic lighting. Static lighting, like its name states, is lighting that does not move within the object. This is best for rendering a scene where objects are not in motion, hence, allowing the lighting effects to be pre-calculated in order to increase efficiency. This technique is known as “light maps” and is widely used in game engines <sup>[3][24]</sup>. Light maps are static lights calculated within a scene; the resulting shadows, brightness, directional static lights and ambient lights are then overlaid on a

scene's pre-existing textures. It is then blended with the textures to create the final lighting effect of the scene.

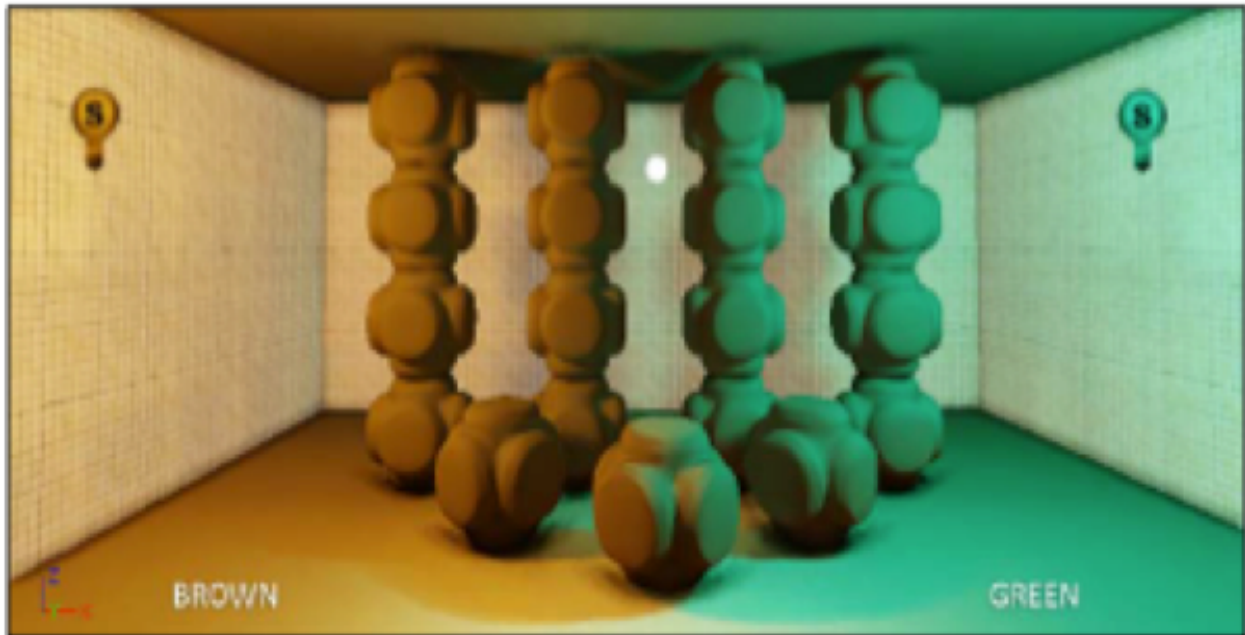


FIGURE 3: STATIC LIGHTING <sup>[23]</sup>.

Dynamic lighting, on the other hand, is lighting that moves or changes within the scene. It cannot be pre-calculated because it will change according to the actions in the scene. Dynamic lights are calculated frame by frame in real time, though they are often stored using the same light map algorithms but constantly replaced as needed <sup>[24] [27] [28]</sup>.

Rendering is the last process and consists of generating the display that will be viewed on the screen. During this process, the lighting, shading and shadows, scene geometry, camera viewpoints, and other scene details are used to create an image. The scene is rendered in real time which allows the user to interact with the virtual world in real time, as well. There are some constraints while rendering a virtual environment. Most of the issues are due to processing power and memory. Because players expect more realistic scenes as computational technology

advances each year, the renderer can be considered the most updated module within a game engine<sup>[24][27] [28]</sup>.

## Physics and Effects

When interacting with a virtual world, the user expects to be able to interact with the virtual scene as they would with the real environment. A physics engine provides a general physics simulation and supplies extensibility to provide game designers facilities to create custom physics rules within a scene<sup>[3] [6] [24]</sup>. One element limiting the physics engine's is the precision of the "collision box", which bounds individual objects in order to detect collision, or overlap, as well as define the forces acting upon the objects, and determining the positional data of those objects. Limitations occur and are commonly known as glitches in a game, often due to not only the collision box not exactly matching the object, but also rounding errors within calculations.



FIGURE 4: EFFECTS ENGINE<sup>[6]</sup>.

Game engines also have what is known as an “effect engine”, produced by the particle effect generator found within the game engine<sup>[3][6]</sup>. Particles are very small objects grouped together to create effects such as fire, fog and smoke. Most particle effects are rendered in real time, affecting performance of the renderer and physics engine. Each particle is bounded by the physical interaction between each other and the virtual environment, making it a computationally intensive, as well as requiring a large amount of memory. However, game engines provide essential optimized particle effects and extensible tools for designers to create their own custom effects.

### **Input, Output and Networking**

Game engines provide essential input and output device interfaces, along with networking capabilities. Game engine designers often work with console manufactures to incorporate their control devices into the game engine, providing the crucial necessities for cross platform development that game developers need to create a game for a greater audience. Game engines support keyboard commands, game controllers, mouse, Wii remotes and new touchpad controls; in addition, they support audio and video, which make game engines a favorable game development platform<sup>[3][19][24][35][38][39]</sup>.

Display device standards are already set by the industry (Televisions, Monitors, etc), alongside with audio industry standards such as 5.1 and 7.1 Dolby Digital; game engines meet such standards to meet the consumer demands in the industry<sup>[19][25][35][39]</sup>. Yet, some game engines have built in APIs for multi-channel sound integration in the games, providing the developer the necessary tools for an immersive sound experience.

Network support is a huge concern when building a game. Multiplayer platforms drive the game industry in different genres of games, making them become a very popular activity<sup>[35]</sup>  
<sup>[21] [23] [38] [39]</sup>. The game engine sends information across the network using industry standards; relaying information about location, actions and scores. Gaming engines mostly follow the server/client model, where the host will provide most of the management concerning scores and detecting game play abnormalities.

### **Game Editor**

Game engines provide a useful tool called a game editor which shields game designers from low level programming and middle level programming when creating and designing a game. Game editors use a GUI approach in game design; this allows them to import and create objects and models, add lighting effects, textures, physics, particle effects, animation, scripts, music and grants them access to other tools used in game design. It can import numerous formats concerning video, audio and models. Game editors have become highly sophisticated in game development; they now have built in capabilities for editing, executing, and debugging a game without programming<sup>[24] [38] [21] [39]</sup>. A resource such as a game editor speeds up designing virtual environment and game environments.

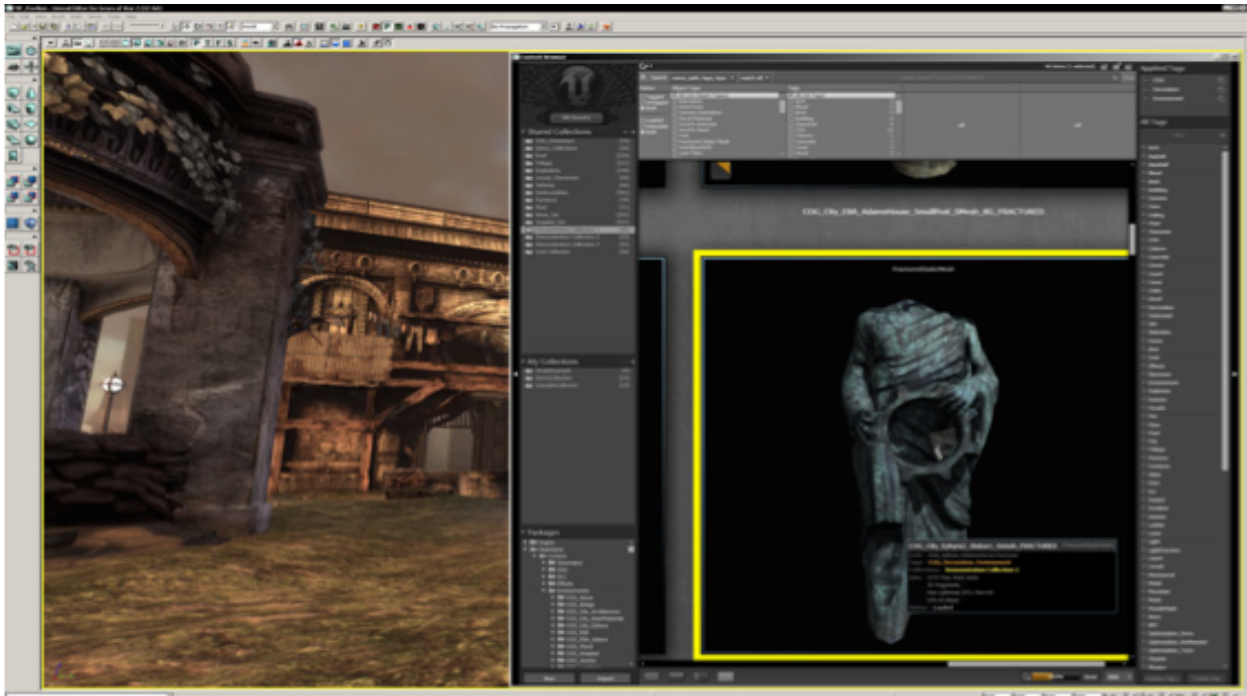


FIGURE 5: UDK GAME EDITOR<sup>[23][37]</sup>

## Game Engines

Today's game engines provide the essential framework of tools aiding in the development of games. There is a variety of different game engines that provide different development resources and approaches in game design.

Unreal Development Kit (UDK) is a game designer toolkit that is built on Epic's Unreal Engine; it provides game designers with a highly developed and extensible libraries aiding in game development. It is a full software package, which allows the user to fully create, compile, debug and play a game created with this software. It supports all the major platforms such as Playstation, Xbox, Wii U, Windows, iOS and OS X. Although Unreal's SDK package it is available as free version software, a license is needed for commercialized games <sup>[24][38][35]</sup>.

Another commercial game engine is Crytek's Cry Engine. Like Epic, Crytek's engine provides a free SDK called Sandbox Editor, which is a game editor and level creator. It uses a different approach in game editor as opposed to UDK game editor. UDK's game editor uses a "subtractive" game design while CryEngine uses an "additive" game design. Additive game design is when objects are created by adding more polygons to a primitive object, meanwhile subtractive design takes away from primitive models until the desired object is created. It supports most platforms with the exception of iOS and OS X<sup>[21]</sup>.

The Unity Engine provides a cross platform for web centric game development. Unlike Epic's UDK and Crytek's Sandbox SDK, Unity's primary focus is mobile and web games. It provides the same principles of game engine design, but uses both open source and proprietary formats for multi-platform development. Since it is web focused, most of the gaming system platforms are supported (Xbox, Wii U, Playstation), but requires effort on developer's part because of its web centric design. Although Unity provides both free and commercial licenses, the developers don't need a commercial license to commercialize their games<sup>[39]</sup>.

### **Modern Open Source Tools**

In recent years, the game industry has produced several software systems capable of providing realistic sceneries and environments with an extensive toolset, minimizing time and effort in creating visually attractive virtual environments. Alongside gaming engines, graphical toolkits such as OpenSceneGraph and OpenSG have pioneered the way for the creation of IV, VE, and VR applications. They have also influenced early research and development of VR systems, which modern VR systems have adapted their technology such as peripheral devices, clustering, networking, window management, and scene development into modern immersive

system infrastructure. This has driven the development of open source tools from the open source community to provide a viable option in game and scene development. In correlation with the gaming industry and graphical toolkits, hardware systems have dramatically increased in their capabilities and decreased cost in recent years, driving graphical software to a new era of concepts and integration.

With the introduction of computer games, people had access to games resource files, which allowed a person to see the essential components of the game. Having access to the components meant that people could develop their own games; this is known through the term “Mod”, meaning modification of a game<sup>[3]</sup>. Mods gave way to the open source community and allowed the creation of new tools and software based on these game modifications. With time the open source community developed great resources aiding in rendering, texturing, window management, display management, physics, networking, device management and scene management based on gaming technology.

### **OpenSceneGraph and OpenSG**

OpenSceneGraph is a highly developed toolkit for rendering 3D graphics. It uses a hierarchy graph of nodes describing graphical layout and objects that compose a scene<sup>[27]</sup>. This provides a low level API which utilizes low level graphic primitives to create highly extensible features. It provides basic features such as texture mapping, shaders, lighting, rendering, camera, memory management, and modeling. OpenSceneGraph also supports advanced features such as particle effects, animations and multi-thread support. It provides foundational elements for scene development and is a highly extensible toolkit aiding in the creation of new platform independent tools. Similar to OpenSceneGraph, OpenSG is an open source scene graph system for rendering



3D graphics. It uses some of the same foundations and provides a scene graph development alternative to OpenSceneGraph <sup>[28]</sup>. Yet, OpenSG provides fewer supports on model formats, clustering and advance rendering techniques.

Both platforms are fully capable of creating robust clustered applications, but require additional time in developing such applications. The low level API, clustered data management and viewport handling capabilities of both toolkits need to be further developed for VR applications.

### **Bullet Physics Engine**

Bullet is an open source cross platform physics engine equipped with a variety of gaming and visual effect applications. This physics engine has been widely used by entertainment and gaming industry because it provides customizable API handling animations, 3D collision detection, rigid body dynamics (static objects), and soft body dynamics (deformable objects) that can be used to model cloth, ropes or deformable volumes. The engine implements Newton's laws of motions and uses a simple friction model to simulate friction. Objects are prevented from colliding with each other by automatically introducing constraints (creating a bounding box around an object). These elements are used to simulate object interaction, allowing a scene to provide a realistic feel to the environment <sup>[6]</sup>.

### **OsgBullet**

OsgBullet provides libraries that extend OpenSceneGraph and Bullet. It uses Bullet libraries for collision detection and physics simulations and OpenSceneGraph libraries for model rendering and data management. OsgBullet provides concrete manipulation functions that allow the programmer to import a model into OpenSceneGraph and automatically generate the Bullet

collision box based on the model's matrices. It takes advantage of multi-threading operations for interleaved and serial physics which handles the object's mass and scaling operations. In addition, it provides custom debugging tools regarding Bullet collision shapes and intersection points between objects. OsgBullet is a crucial component for incorporating Bullet physics engine in OpenSceneGraph applications <sup>[29]</sup>.

### **OsgWorks**

OsgWorks is a package of applications and libraries used for extending OpenSceneGraph capabilities when working with model matrix transformations. Its principle function is to extend functionality of scene matrix and geometry manipulation. Its geometry modifier infrastructures can reduce an object's geometric data set or add new data sets to objects. It provides shape manipulation functions for polygons, which is a useful tool when rendering objects. OsgWorks is a rudimentary component for a system using OsgBullet toolkit and Bullet physics engine<sup>[31]</sup>.

### **OsgEdit**

OsgEdit is an open source toolkit known as a scene composer which provides the necessary tools to edit model positions in 3D space. It contributes essential actions that are expected of game editor and isolates the user from the hassles of programming a scene in source code because most of the functionality is provided in the form of a menu and toolbars. Object manipulation is enabled in 3D space with standard attributes features like rotation, scaling, translation, and lighting. OsgEdit provides extensibility of customizing your own features, allowing you to directly customize objects and nodes. Most of its functionality is XML based, providing an excellent foundation for extensibility <sup>[30]</sup>.

## **Virtual Reality Peripheral Network (VRPN)**

VRPN is a system of libraries that provide a network transparent interface for device integration in cross platform VR applications <sup>[34]</sup>. Through the use of TCP/IP protocols, it establishes a server/client model to establish device connections making VRPN a reliable platform against the use of conventional hardware drivers that are tightly coupled to an operating system. VRPN allows for multiple peripherals like trackers, button devices, analog inputs, sounds and wands to be used by an immersive system. There are pre-supported peripherals that are widely used within the VR research community, making VRPN a must-have device manager<sup>[41]</sup>.

## CHAPTER III

### VIZLAB VR SYSTEM

There are different things to consider when deciding to build a VR system; for example, what are the goals of the VR system? What display technologies are best to use? How will device integration be handled? What will aid scene development? These are just some of many questions that need to be addressed when developing an immersive system. The development of VR applications can be a burden because most of the documentation for open source software's can be sparse and outdated at times. This de-motivates developers, making them search for other alternatives. However, the use of open source software can help developers create robust custom software.

VizLab VR System is built upon combining open source software to aid in the development of VR applications. It is composed of VR Juggler handling display management, system “clusterization”, data management, input management and output management<sup>[1][40]</sup>. OpenSceneGraph and OpenSG provide the system with model rendering and scene integration, while using OsgEdit for scene development as a primitive game editor. Scene physics and object collisions are managed through a combination of OsgWorks, OsgBullet and Bullet layered above OpenScenGraph. VRPN is used for extending device integration apart from VRJuggler own

device management. A system layout can be seen in Figure 6 and the following sections describe the VizLab System.

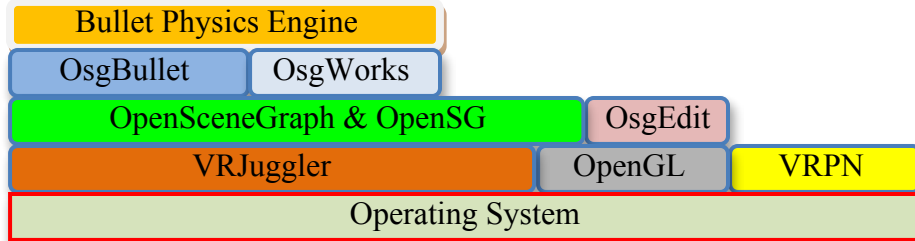


FIGURE 6. VIZLAB SOFTWARE ARCHITECTURE

## VR Juggler

### Modules

VRJuggler modules are the core of VRJuggler and are what helps VRJuggler build a robust VR system<sup>[1]</sup>. These modules are:

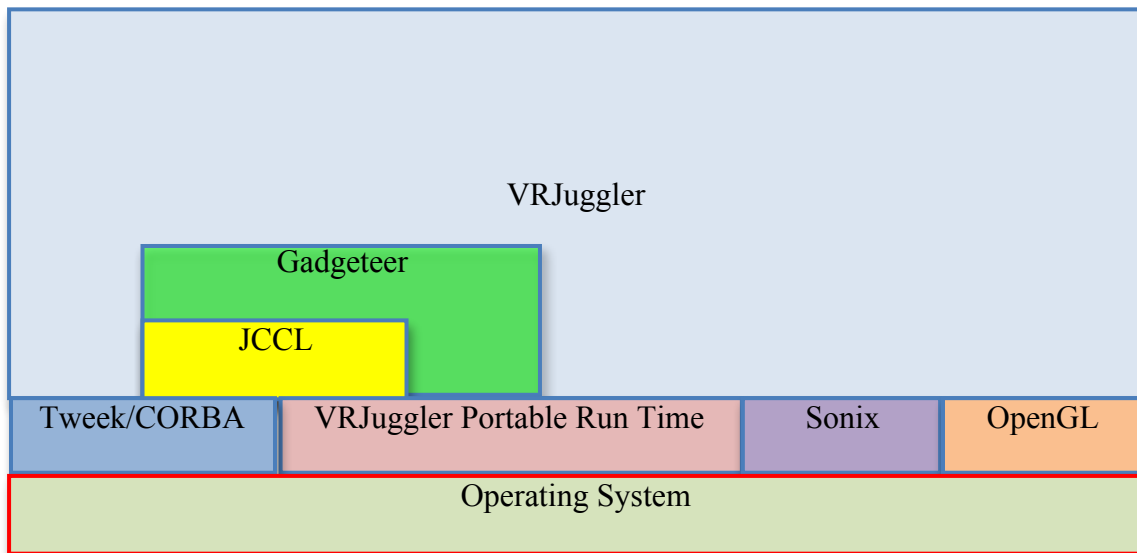


FIGURE 7. VRJUGGLER MODULES

- Gadgeteer - device management system used to handle the configuration, control, acquisition, and representation of data from VR devices.

- Juggler Configuration Control Library (JCCL) - XML-based configuration system with multivariate types used internally by VR Juggler for all system configurations.
- Sonix - high-level simple abstraction for audio hardware or audio APIs that provides an interface useful to many simple VR and entertainment applications such as trigger, 3D position, etc.
- VRJuggler Portable Runtime (VPR)- provides platform-independent abstractions for threads, sockets (TCP/UDP), and serial I/O primitives.

Gadeteer manages the hardware devices within VRJuggler. It contains a Remote Input Manager that allows sharing of devices between computers and provides device input for use with VR Juggler applications. Gadeteer hides input device hardware from programmers during the implementation of immersive software in order to shield them from low-level device drivers. With Gadeteer, applications can be easily migrated between different hardware configurations, allowing the programmer to have no required knowledge on device's vendors, models, drivers, or hardware specific protocols. Gadeteer categorizes input devices based on abstract input types such as analog, command, digital, glove, gesture, position, simulator and string. In this categorization, devices from different vendors may return data mapping to the same abstract form. A single piece of hardware may even map to multiple input types, and more device types can be added as new hardware becomes available. Programmers can write their code in terms of abstract input types, so as long as a device is available that provides the needed input, the application can function<sup>[1][40]</sup>.

JCCL provides configuration and performance monitoring tools for applications, allowing easy access to configuration information stored in XML-based files for the applications built with JCCL. It provides a Java-based GUI called VJ Control for editing these configuration

files and then incorporates additional tools to monitor the performance of an application. The JCCL GUI also provides facilities to edit and control the state of running applications<sup>[40]</sup>.

Sonix is a sound toolkit that is kept very simple in order to get sound running as fast as possible. Sonix is reconfigurable, allowing audio APIs to be safely swapped out at runtime without the dependent systems noticing. Systems using Sonix API layer can expect their sound application to be completely portable<sup>[40]</sup>.

VPR provides a cross-platform object-oriented abstraction layer to common operating system features. It is the key to the portability of Gadgeteer, Tweek (see GUI Capabilities), VR Juggler, and other middleware included with VRJuggler. Software written on top of VPR can be compiled on IRIX, Linux, Windows, FreeBSD, and Solaris, usually without modification. Internally, VPR wraps platform-specific APIs such as BSD sockets, POSIX threads, Win32 threads, and Win32 overlapped I/O. Depending upon how it is compiled it may also wrap the Netscape Portable Runtime, another cross-platform OS abstraction layer written in C<sup>[40]</sup>. In summary, VPR is a collection of utility classes.

## **Installation and Configuration**

Like most open source software, the source code for compilation is provided. Installation of VRJuggler can be challenging since it is dependent on different open source software. The documentation does not clearly state what versions of the software are needed, which makes it difficult when trying to acquire the correct version for building VRJuggler. Some of the required dependencies for building VRJuggler are:

- Boost Library - consists of a C++ library that provides many powerful utility classes and libraries<sup>[4]</sup>
- Generic Math Template Library GMTL - a generic math library that makes use of C++ templates and STL paradigms<sup>[18]</sup>.
- CPP Document Object Model (CppDOM)- a lightweight, easy-to-use XML parser written in C++ <sup>[8]</sup>.
- Java Developer Kit (or JDK) - used to compile all the Java code used in the Juggler Project. Without it, none of the Java code in VR Juggler can be built<sup>[20]</sup>.
- Java 3D – Java extension used for displaying three dimensional graphics. Version 1.5.2 of Java 3D was used for this particular project <sup>[20]</sup>.
- omniORB - a C++ implementation of CORBA 2.3, required for the Tweek C++ API. The omniORB version required for this project is available as a pre-built snapshot<sup>[32]</sup>.
- CppUnit - unit testing framework for C++. The Juggler C++ test suites make use of extensions to CppUnit<sup>[9]</sup>.
- Virtual-Reality Peripheral Network (VRPN) - a set of classes within a library and a set of servers that are designed to implement a network-transparent interface between application programs. VRPN is needed for Device Interface communication between Devices and VR Juggler<sup>[41]</sup>.
- Open Audio Library (OpenAL) and OpenAL Utility Toolkit - cross-platform 3D audio API appropriate for use with gaming applications and many other types of audio applications. OpenAL is required to add audio into VR Juggler Applications<sup>[25]</sup>.



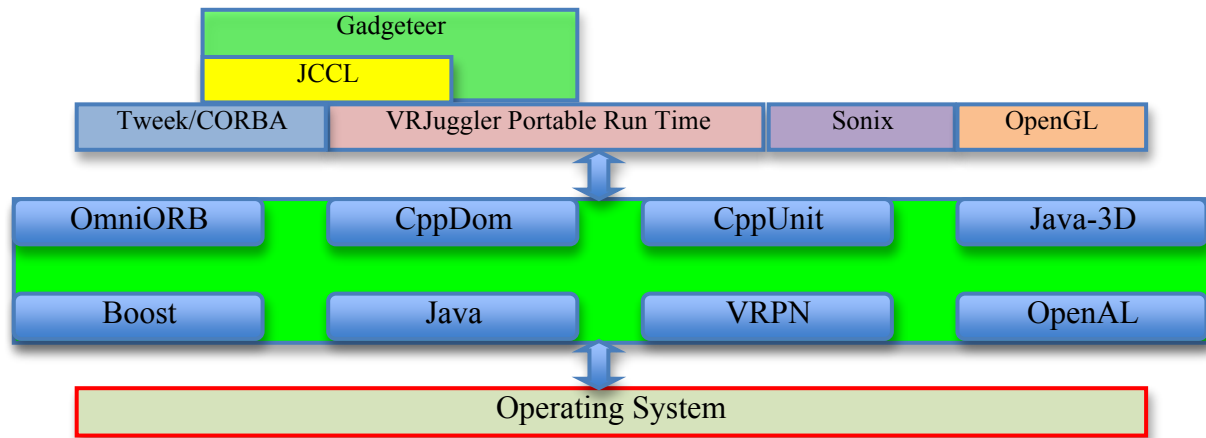


FIGURE 8. VRJUGGLER DEPENDENCIES

These dependencies often require the use of environment variables to be set to specific predefined parameters in order for them to work, as seen on Figure 9. Environment variables help the application's processes run, find, or execute required resources. This is often the case with open source software, since it is not a package installation, which would take care of this nuisance and makes working with open source software a considerable problem. The compilation of the dependencies can take a considerable amount of time because everything needs to be tracked by environment variables, as seen in Figure 10. Dependencies are often based on others, so having to deal with different versions of the software and different environment variables can be troublesome. When updating the software, the updated version will update, delete, or add new functionality to the software, making it difficult for compilation. This is often the case in layered multi-dependent software, as functions are updated or deleted, software layers are may become dependent on such functionality that would make compilation impossible.

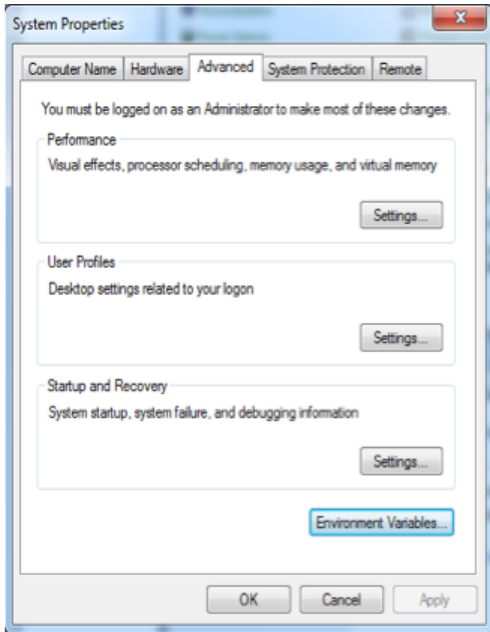


FIGURE 9: WINDOWS SYSTEM PROPERTIES

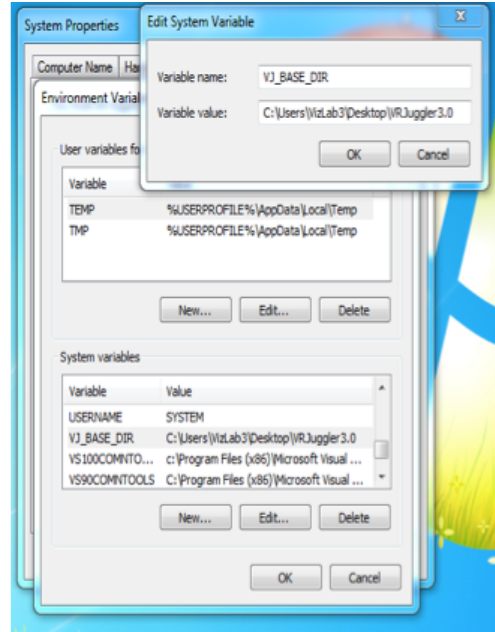


FIGURE 10: ENVIRONMENT VARIABLES

Compilation will require compiling packages that are platform dependent. Since most dependencies are platform independent, they use a cross compiler package called Cmake to build software to specific platforms<sup>[7]</sup>. Figure 11 shows that Cmake is script-based with a nice interface allowing to set up compiling flags, libraries, specific environment variables, installation folder, and compiler. Most dependencies are compiled with Cmake; however, if a Cmake script is not supplied, the dependency will mostly likely have a script for installation.

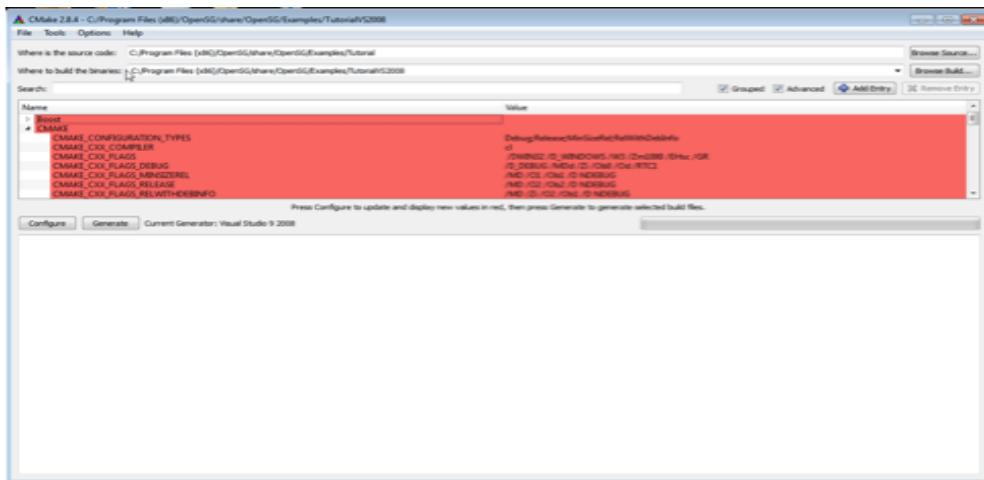


FIGURE 11: CMAKE GUI<sup>[7]</sup>

All dependencies have to be precompiled because VRJuggler only needs the binary files of all the dependencies. After using Cmake or installation scripts, the software is compiled using platform specific compilation tools. For Windows systems it is preferred to use Visual Studio for compilation, while for Linux systems g++ and make are used. In the case of Windows, Cmake scripts already create a Visual Studio project depending on which version is selected by the user in the script. Figure 11 illustrates how this project will create the binaries for software. Based on the architecture selected on the Cmake script, it will allow you to compile the build; it is important to note that not all builds compile. It is in this section where it is most likely to fail if a different version of the dependency software is needed. Because VizLab is packaged software, it removes such difficulties. All dependencies are precompiled in this package, allowing installation by just clicking a button. Figure 12 and figure 13, show what the GUI for installation looks like.

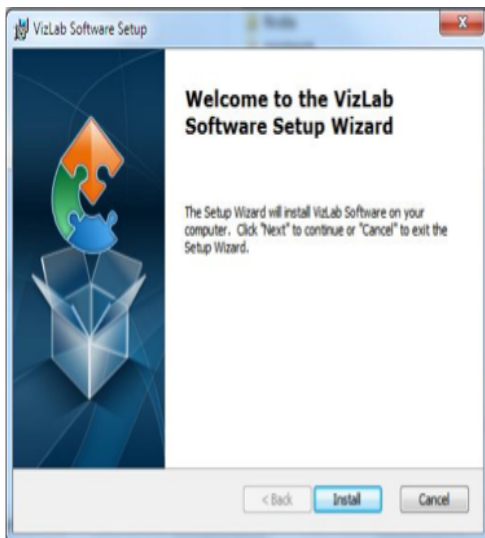


FIGURE 12: VIZLAB WINDOWS INSTALLATION

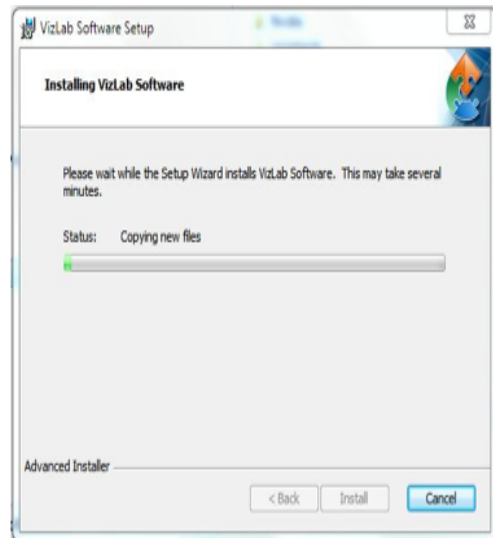


FIGURE 13: VIZLAB WINDOWS INSTALLATION PROCESS

This installation wizard installs VRJuggler and all its dependencies. Therefore, the user will not need to set up environment variables or destination paths because it is done through the

wizard, minimizing the installation process of an immersive system. There are two installation versions that come with VizLab, a Microsoft installation package seen in Figure 12 and Figure 13, and in a Linux Ubuntu Debian packages shown in Figure 14. The Debian packages have to be installed in a certain order; information on Windows platform installation is provided in Appendix A and Appendix B for Ubuntu installation. It is important to note that these installations minimize the effort and time from many hours to just a few minutes. The VizLab used is a x64 build because the first release was x86 build and had performance issues. This was not a wise decision since we are using mid to high end workstations with high end graphics cards. The main issue was memory usage; each workstation has 24GB of RAM but was not fully utilized. When running intensive demo scenes, we saw a drop in performance; when using the x86 build architecture, we could only access 4GB of space. Yet, you have to account that a compiler may reduce memory allocation, so moving onto x64 build increases memory allocation space.

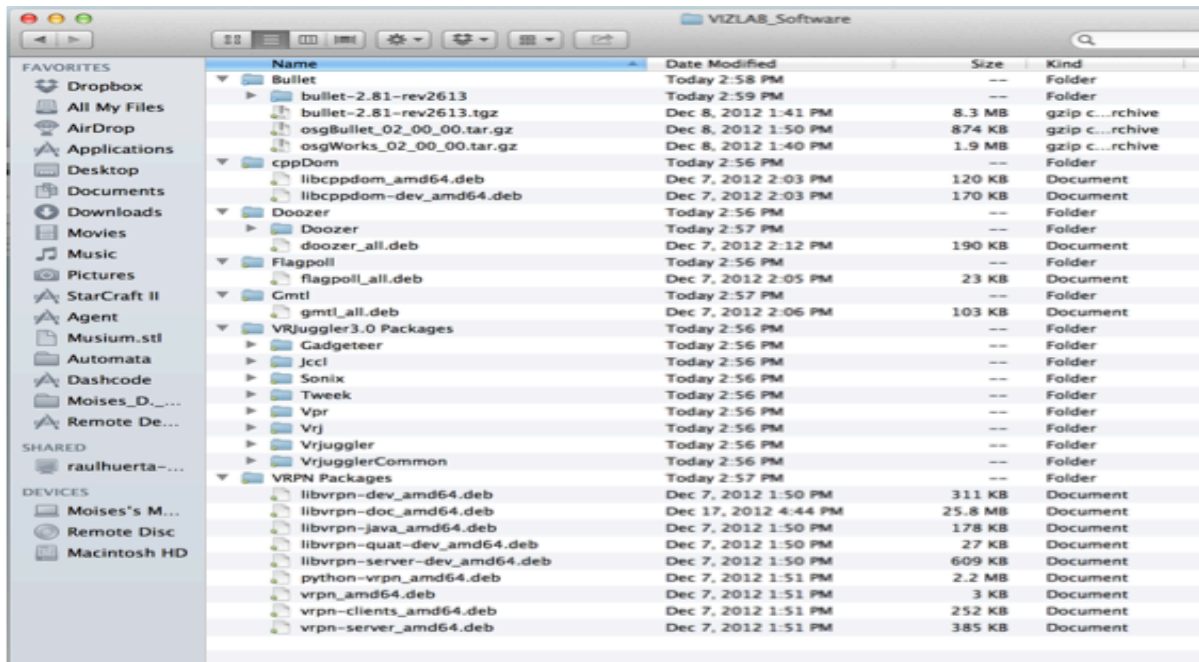


FIGURE 14: UBUNTU PACKAGES

After the VizLab is installed, the next step is to configure the immersive system. To do this it is necessary to determine the hardware that will be used in the system. For example, how many computers will form the immersive system and which of those computers will control input and output peripherals? VizLab is a multi-configurable system that ranges from a standard one display system to a multi-display system when needed. It can run from one computer or multiple computers that form a system known as a cluster. An advantage of running a clusterized system is that as the complexity of scenes increases, a cluster will better maintain performance than a single computer. Having a multi-display system run on a single computer with one graphic card can be done, but graphics performance will be significantly impacted for complex VEs.

The VizLab systems used for this project consists of five computers configured as a cluster. In this system there are four slave nodes and one master node. The slave nodes in VizLab handle the display of the VEs, while the last slave node handles the tracking software. The master node keeps track of all slave nodes and syncs the system together and is in charge of distributing input device data to the other nodes. It also keeps track of each viewport displayed by each node and syncs the movement in the VE so that the movement through a VE can be smooth throughout the displays. A viewport is the term used to describe the viewing region of a scene in which each display slave node runs its own viewport.

The network for the cluster consists of a regular Netgear Router and a Linksys ethernet switch. Although the whole network could be run with a switch leaving the master node handle all communication, a router minimizes the effort in data routing, port forwarding, and resource allocation. Today's routers have built in tools for performance factors; in the event that a node should fail, all IP settings are stored in the router, allowing for easier management should the node be replaced. VizLab's system layout can be seen in figure 15.

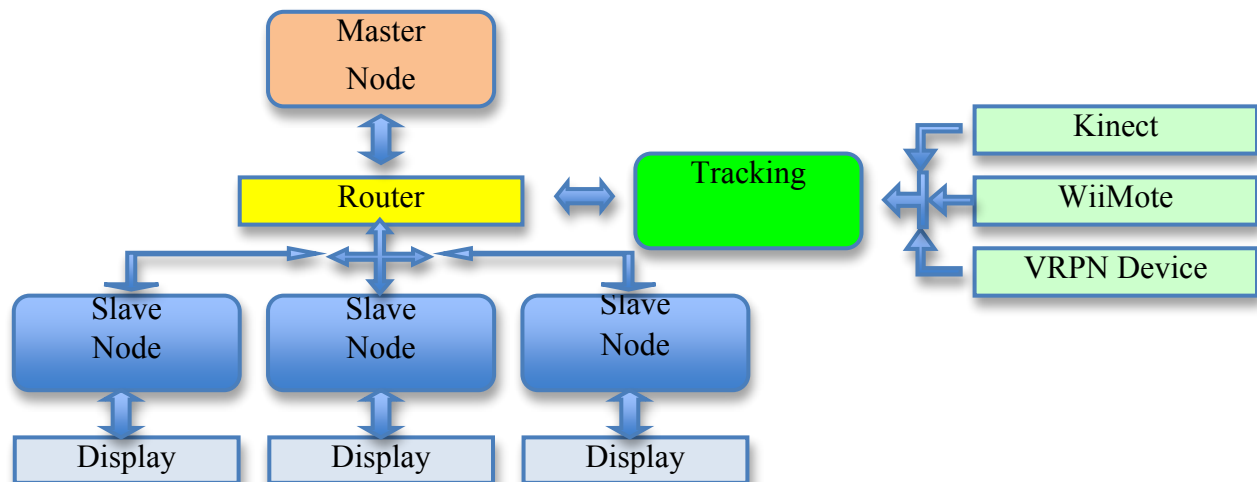


FIGURE 15. VIZLAB SYSTEM LAYOUT

Configuring the system is straightforward and uses VRJuggler's the VRJConfig system as seen in Figure 16. VRJConfig system acts as middleware for configuring an immersive system by providing detail about Cluster Nodes, Input and Output devices, Viewports, and Users that make the system, as well as a method to relatively easily set parameters. VRJConfig has a GUI, making the system easier to work with when configuring an immersive system. Although VizLab comes with predefined configurations, the user can create other configurations. Figure 17 shows a cluster configuration typical of VizLab's configurations. As can be seen, it shows a five node cluster and the user to add and delete nodes with the click of a button. Selecting a node would give the user a detailed view for configuring the node on the right hand side, as shown of Figure 16. Each node can render different viewports. The user can set up all viewports' attributes, including 3D rendering, positional location within the scene, viewport size, trackers tied to the viewport, as well a several other attributes associated with each node. There are different devices configured in the system that can be be added and configured through this editor. Adding devices will be covered later , and more detail is provided in Appendix C. Note that the configuration uses proxies, middleware between device input, to handle lower levels of communication

between devices and data manipulation within an application. A proxy can contain any type of device, analog, digital, positional, etc., and will have naming conventions easier to work with within a programming environment. Setting up multiple users within the system allows different tracking purposes and keeps information regarding each user. In summary, VRJConfig serves to fully configure the system, and Appendix C contains further information regarding system configuration.

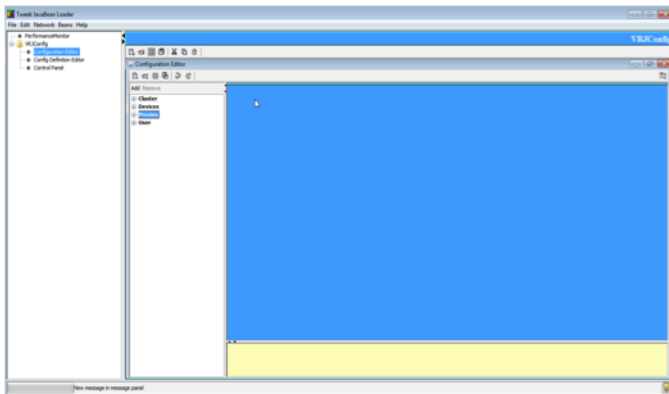


FIGURE 16: VRJCONFIG

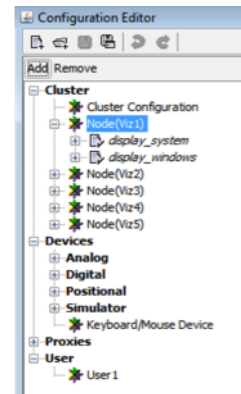


FIGURE 17: VRJCONFIG PANEL.

## Applications

VizLab has some sample applications that are provided as VRJuggler sample applications. Though they are inherited from VRJuggler, the applications in VizLab are modified versions of the original applications. The main sample applications are OpenGL applications, OSG applications, device applications and sound Applications. All are sample applications that help developers become acquainted with VR application development. Typically, the principle elements of the sample applications are combined to create fully immersive VR applications.

OpenGL applications share the GLUT programming paradigm, which has drawing functions, initialization functions, frame functions, and update functions<sup>[17][26]</sup>. This object-oriented structure is the base of VRJuggler's programming paradigm. The main application runs

the VRJuggler kernel, which controls the application execution. The kernel instance takes in the configuration of the system and then calls the main execute function. This execute function will call the application's initialization, drawing, and update functions. By using this method, the developer is removed from VRJuggler's execution engine, and can focus on developing immersive applications. Figure 18, illustrates MPApp application running in VRJuggler. Notice that there is a ball and a stick within the display window, this is the simulated head and wand position rendered by the application. By using the simple standard configuration, the user can navigate using a keyboard and mouse. The programmer can incorporate full tracking to suit the needs of a specific application. Application developed using OpenGL or GLUT can be easily incorporated into VizLab.

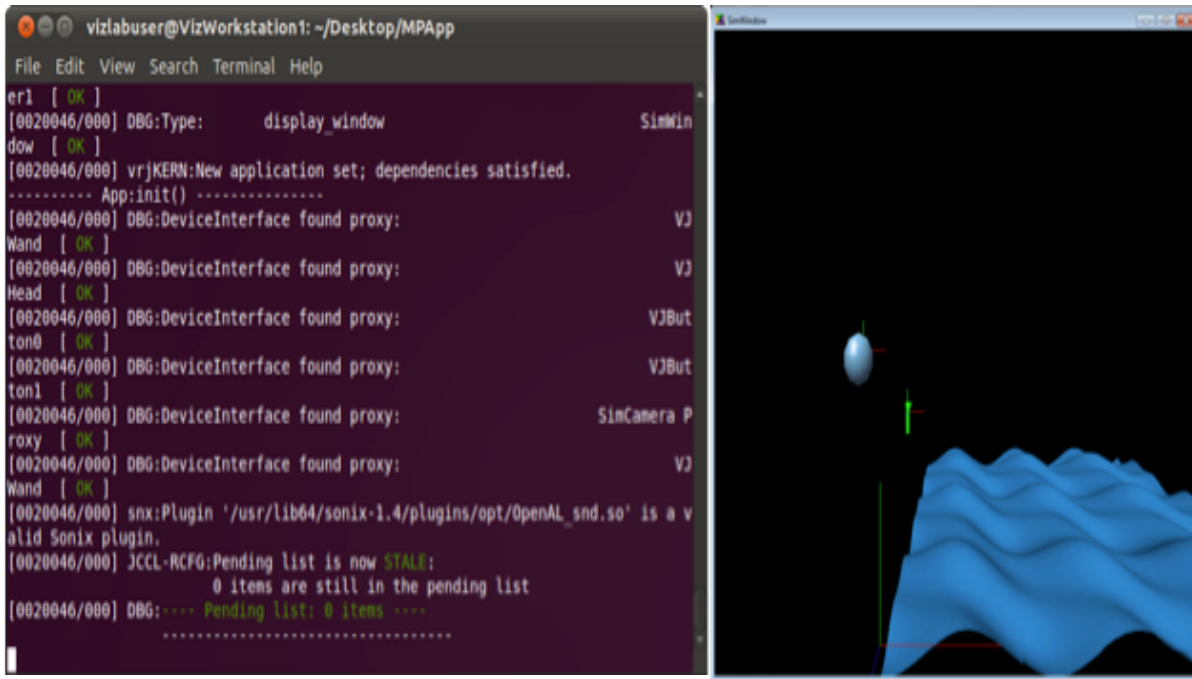


FIGURE 18: OPENGL APPLICATION



OpenGL applications are not that much different from OSG applications, as the same programming paradigm is used to render OpenSceneGraph and OpenSG applications. There are some added functions besides initialization, drawing, and update functions. The added functions are for scene initialization, scene scale conversions, and scene navigation. In Figure 15, the head and wand are simulated, as shown in the previous OpenGL application example. The scene can be imported from various different models or a single model. A variety of major file formats are supported, including .obj, .3ds, .dae, .wrl, .flr, and native OSG binary formats<sup>[27]</sup>. Figure 19 is an imported scene from the city of Boston running on VizLab.

The last sample application is the OpenGL and OSG applications with Bullet physics engine. Since a similar design structure is used in building both applications, Vizlab has a sample application that covers both types of applications in one application. The integration of the Bullet physics engine provides the developer a fully developed physics engine to assist in the building of VE and VR applications. The Bullet integration is shown in Figure 20. This sample application demonstrates how to add objects of different shapes that interact with all other objects in the VE.



FIGURE 19: OPENSCEENEGRAPH APPLICATION

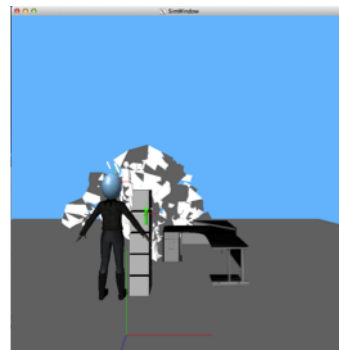


FIGURE 20: OPENSCEENEGRAPH AND BULLET

## Rendering and Physics

### GUI Capabilities

VizLab can help developers build VR, VE, and information visualization (IV) applications. It is extensible and allows for “clusterization” of VR applications, limited only by the hardware controlled by the applications. It is also capable of creating OpenGL and Scene Graph applications, as well as interactive VEs incorporating Bullet as a physics engine. Another aspect of VRJuggler’s integration in VizLab is the use of Tweak to provide VR user interfaces (UI) <sup>[1][40]</sup>. The UI toolkit acts as middleware and addresses input hardware mapping for user driven actions or interaction in a VR application. When using this tool, developers can create a GUI that provides the capabilities users need for interacting with the VE. For example, a VR application can track the user’s position and orientation with Tweak providing the tools necessary for integrating GUIs in a VR application. Another use would be for an IV application to display data at certain times or to keep track of data and display it to the user. Tweak can also act as a spectator to the application, only extracting information from it, providing a tool that does not affect performance of a VR application.

### Scene Graph Integration

Integrating Scene Graphs in a VizLab application is made possible by VRJuggler’s integration of OSG functions. VRJuggler allows for both OpenSceneGraph and OpenSG packages to work in conjunction with VRJuggler’s architecture. Note that OpenSceneGraph and OpenSG are not fully integrated into VRJuggler and must be installed before VRJuggler in order to create the bindings for a VRJuggler application. VRJuggler has built in wrappers to work with OSG based applications which shields the developer from directly manipulating the VRJuggler

kernel for rendering scene graph applications<sup>[40]</sup>. This wrapper invokes running a scene graph application, and in this way VRJuggler's kernel API will call OpenSceneGraph directly from its libraries. The wrapper's main functions are `initScene()` and `getScene()`. The `initScene()` function initializes and notifies the kernel to invoke OSG libraries. The scene is passed by a parameter when `getScene()` is invoked, loading the scene into memory for rendering. After the scene is loaded, the kernel loops until an update occurs, which will trigger the built in update function of the wrapper. Figure 21 shows the traversal information (left) when traversing a scene together with the scene (right). The figure also shows the position of the simulated head and wand, however the simulated head and wand are simulated devices that are kernel based, so the scene layer is later added to the rendering module running in the kernel. This will be explained later when Bullet physics engine is working with scene graph applications.

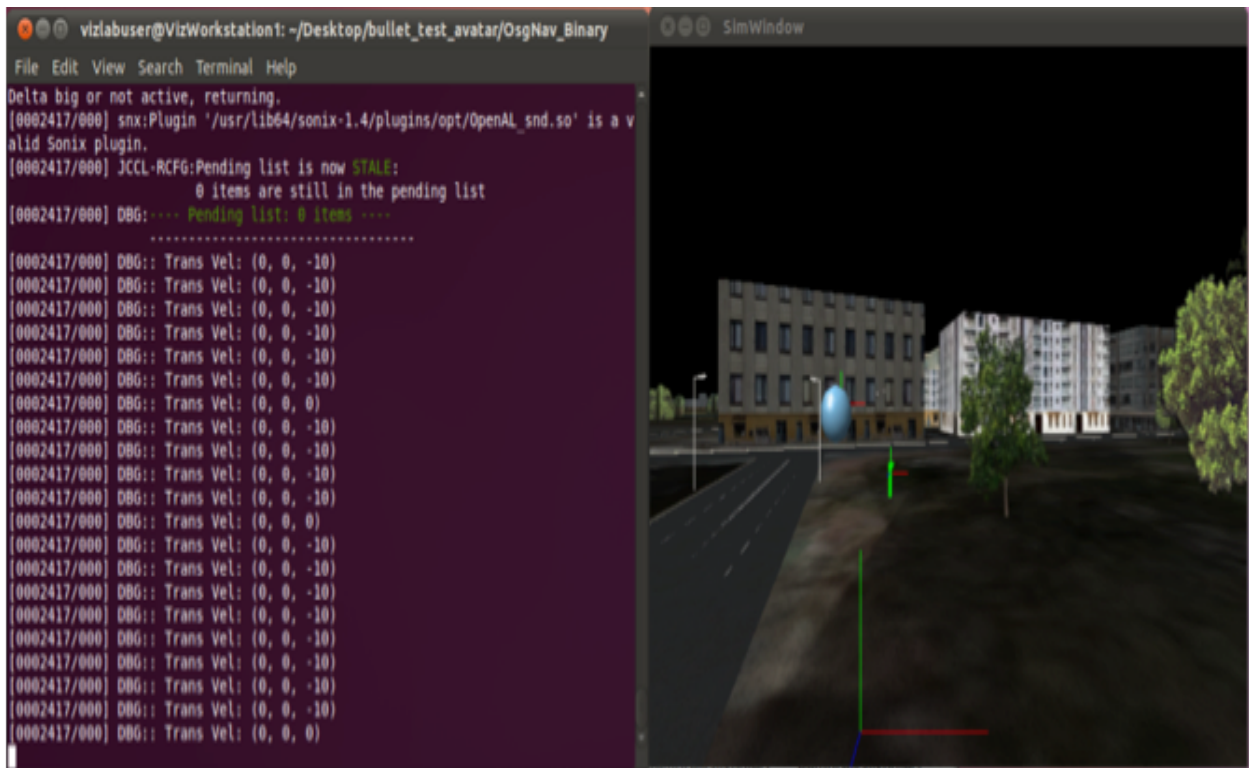


FIGURE 21: OPENSCEINGRAPH APPLICATION RUNNING IN VRJUGGLER.

## Physics and Collision Integration

Typically, when creating immersive games or virtual environments, the physical interaction with virtual objects involves multiple processes and is time consuming to implement. Using the Bullet physics engine minimizes development time for VR, VE, and IV applications. In order to work with Bullet and scene graph toolkits, middleware is used to handle communication between the two libraries. OsgWorks and OsgBullet provide that communication layer needed for OSG objects model information (the objects' matrices) to be passed to Bullet for collision detection. Bullet takes the objects information and creates a 3D bounding box for the object. The bounding box is what distinguishes one object from another. The other properties of the object are its mass and whether the object is moveable or not. These are to Bullet as parameters when the Bullet engine is invoked. It is important to note that Bullet is an independent process from the VRJuggler kernel and is therefore not integrated with VRJuggler. The isolation between the two has good and bad aspects for development and performance. First, there is no performance penalty in that the VRJuggler kernel is not dependent on Bullet when rendering. This means that the kernel can be rendering the scene while Bullet is calculating the physics elements and performing the model transformations for the nodes of the scene graph. When the OSG wrapper calls the update function, the scene graph is already updated by Bullet, taking heavy calculations out of the kernel. If this were not the case, the kernel would be slower, since physics calculations are often computationally intensive. A disadvantage to this isolation is that it is not as easily developed. Most of the time will be spent transferring scene graph objects to the Bullet engine. Also, since virtual devices are rendered apart from the scene graph, as noted in previous section, the binding of virtual devices may result in imperfect rendering when next to

scene graph objects. For example, in Figure 22 the avatar is going through the object and not colliding with it, even though the collision box is evident around the avatar.

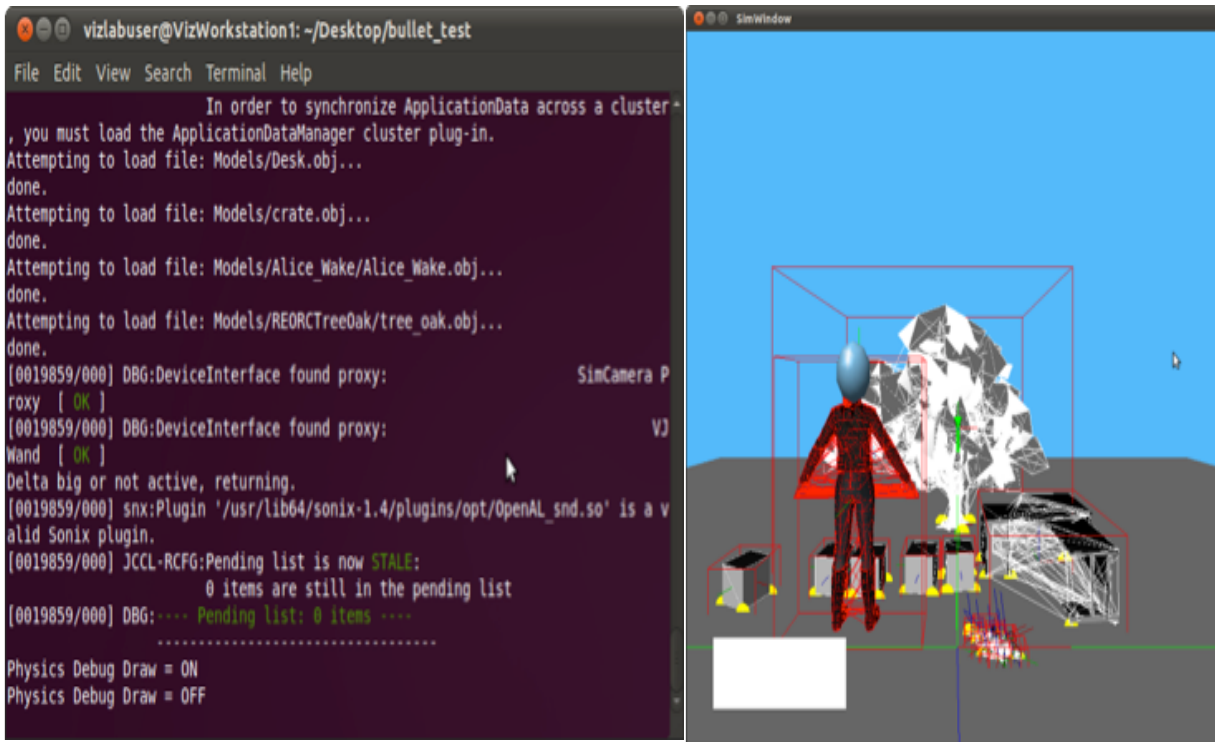


FIGURE 22: OPENSCEINGRAPH WITH BULLET PHYSICS ENGINE

## Composing Scenes

Making scenes within a coding environment is very time consuming and not an efficient means to create non-trivial scenes. OsgEdit provides an alternative for scene development as a game editor that composes scenes<sup>[30]</sup>. Because it is open source it can be integrated with VRJconfig since both handle information in XML format. Figure 23 shows the basic layout of a scene project in OsgEdit. On the top of the screen displayed in the figure is the menu that provides an interface for basic manipulation of objects in the scene, as well as creating basic shapes like spheres, squares, cones, and cylinders. Within the application, there are window layers that show the objects in 3D space and a scene graph node tree showing the components of

the scene graph. It is necessary to create a node within the scene graph to add an object into a scene. A node holds the object (information) and is used to identify the object for scene rendering. Ultimately, a scene is composed of various nodes such as LOD nodes, light nodes, transformation nodes, object nodes, surface nodes, etc. Within the component layer of OsgEdit, facilities are provided to edit the properties of the node or nodes selected within the scene.

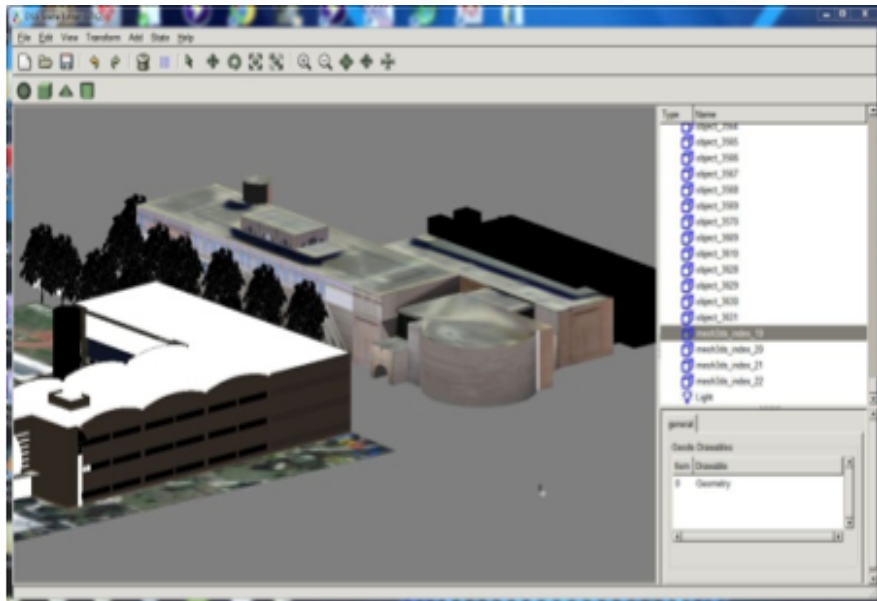


FIGURE 23: OSGEDIT SAMPLE SCENE<sup>[30]</sup>

## Device Management

### Device Integration

Devices are integrated using VRJConfig<sup>[1][40]</sup>. VRjuggler supports multiple devices including datagloves, mice, keyboards, joysticks, and trackballs. Figure 24 shows that the described devices are subcategories of the several device categories seen in Figure 17. These categories include the following. Analog devices stream data in a continuous range from a set of defined minimum and maximum values. These input values range from 0.0 to 1.0 when using analog devices with VRJuggler. Digital devices send information as on and off, or 1s and 0s and

usually correspond to button devices. Positional devices are multi-input devices usually employed for system tracking. Keyboard and mouse devices are bindings to keyboard and mouse drivers. Digital Glove devices allow for the distinction of finger input, and Gesture Glove devices allow for hand gestures based on knuckle angles. More in-depth information about configuration is available in Appendix C.

After configuring devices, the next step is creating the communication proxies. Proxies are used to relay device information to specific predefined names. Device proxies introduce a layer of abstraction and indirection needed to ensure that VR applications do not become tightly coupled with specific devices. Instead, the proxy acts as a pointer to a single data source from the actual device. This is also important when working with multi-input devices, as it distinguishes inputs from one another. For example, a keyboard device consists of multiple keys, and since not all keys might be used in an application, a Button Proxy that can take the value of a specific key. Figure 25 shows that there are more proxies than there are devices shown in Figure 24, and there are multiple proxies that use the same device. Here, buttons 0 through 5 are proxies that point to keyboard device and are configured to refer to specific keys. Another element needed for the use of specific inputs is a pointer pointing to the specific proxy. For example, `VJButton0` is pointing to `Button0Proxy`, and this `VJButton` will act as the communicator inside of the application. On the programming side, the developer only needs to get information from `VJButton0`. This allows for devices to be switched from a keyboard to another button device, such as a Wii controller, not needing to change the application, just the configuration. For further information about integrating devices into a VizLab application, please see Appendix C.

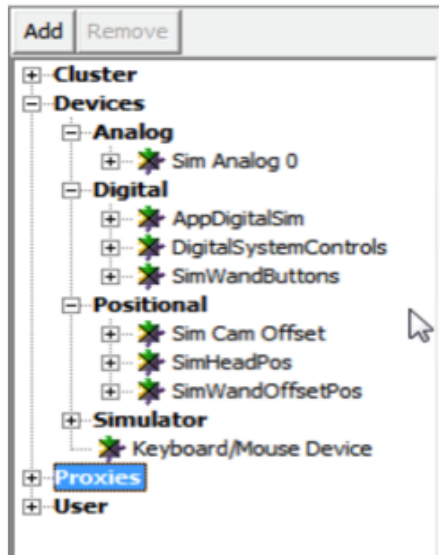


FIGURE 24: VRJCONFIG DEVICE LAYOUT

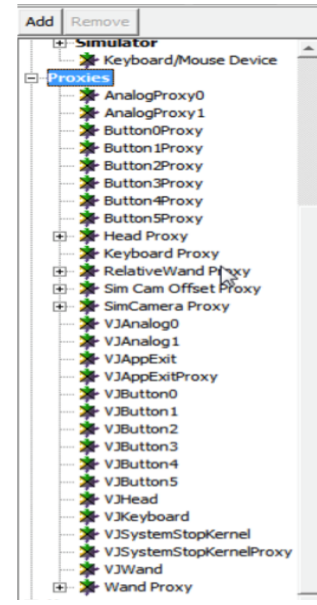


FIGURE 25: VRJCONFIG DEVICE PROXIES

## VRPN Integration

Integrating a VRPN device is accomplished through VRJConfig by adding a VRPN device<sup>[34] [40]</sup>. With VRPN devices, the user needs to configure the VRPN servers that are going to be used with a given VR application. Establishing VRPN servers is done by entering the computer's IP address in the corresponding field, as seen in Figure 22. There are three types of servers that could be specified when setting up a VRPN device. A tracking server is a VRPN server that sends information for tracking information and usually consists of multiple sensors. The sensors consist of positional data and can be send with XYZ coordinates, quaternions or a combination of the two. A VRPN Button Server is a server that sends digital information, meaning 0 or 1. A VRPN Analog Server sends a constant stream of information and usually consists of a mouse device or joystick device. In Figure 26 we can see that the developer must specify the specific VRPN servers for the VR application with the sensor trackers, buttons, and analog devices streamed with each server. It is also possible to specify the range of the analog



stream. With this configuration information, VRJuggler invokes the requested VRPN clients for receiving data streams. Again, more detail is provided in Appendix C.

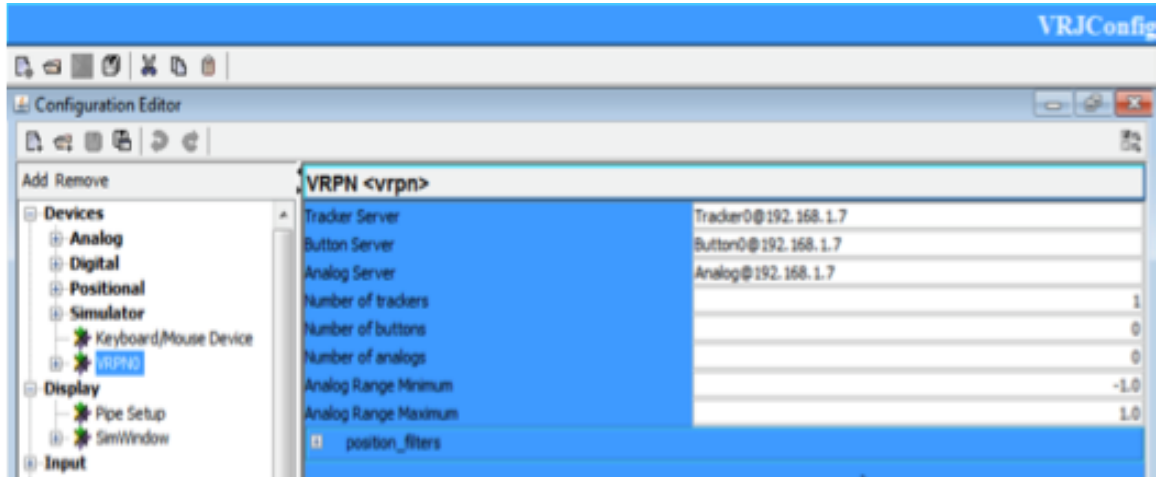


FIGURE 26: VRJCONFIG VRPN SERVER SETUP

## Motion and Gestures

### Microsoft Kinect

When introduced, Microsoft's Kinect gaming peripheral showed promise as a low cost alternative for full body tracking. This inspired the open source community to build toolkits for its use, among which is the Flexible Action and Articulated Skeleton Toolkit (FAAST). FAAST is a toolkit that provides enhancement for the motion, gesture, and body tracking capabilities for VR applications<sup>[34]</sup>. It incorporates a VRPN server to stream up to four user skeletons by reading skeletal joints as sensors using a VRPN client. The toolkit also emulates keyboard input triggered by body posture and specific gestures, allowing the programmer to add custom body-based controls to VR applications<sup>[16][41]</sup>. Figure 27 displays FAAST GUI using Microsoft's Kinect for full skeleton tracking. The FAAST GUI provides an excellent means for creating gestures, configuring gestures and motions, and handling Kinect sensor configurations. Gestures are a sequence of movements based on joint positions. These sequences trigger the actions to be

executed. Configuring gestures is accomplished by arranging specific sequences that are then mapped to either buttons or events. It is possible for a gesture to consist of a series of events, making complex gestures possible. FFAST can run all three VRPN servers (see VRPN section), which allows all gestures and movements to be stream to corresponding VRPN servers. Detail is provided in Appendix C.

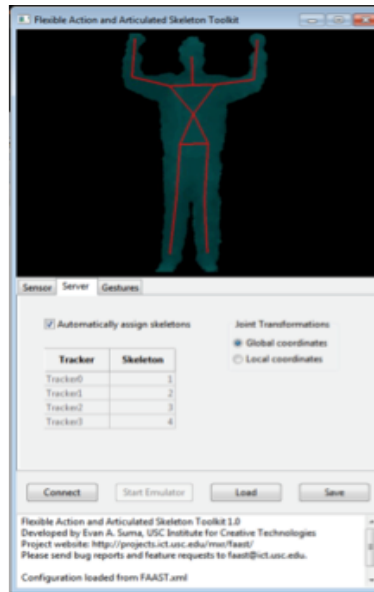


FIGURE 27: FFAST GUI<sup>[7]</sup>

## Nintendo WiiMote

Nintendo's Wii and its gaming peripheral the WiiMote also has open source toolkits available. The WiiMote can be used as a head-tracking device with the use of infrared LEDs or a wand. Figure 28 shows a toolkit called CWiid, which allows for the configuration of button and the motion sensors to be mapped to keyboard keys and positional software<sup>[12][13]</sup>. This toolkit handles the communication and decrypts WiiMote's peripheral codes from BlueTooth connectivity. First, WiiMote is paired with Cswiid's corresponding software. Then, it is possible to configure its controls with the WMgui interface software provided by CWiid, as detailed in Appendix C.

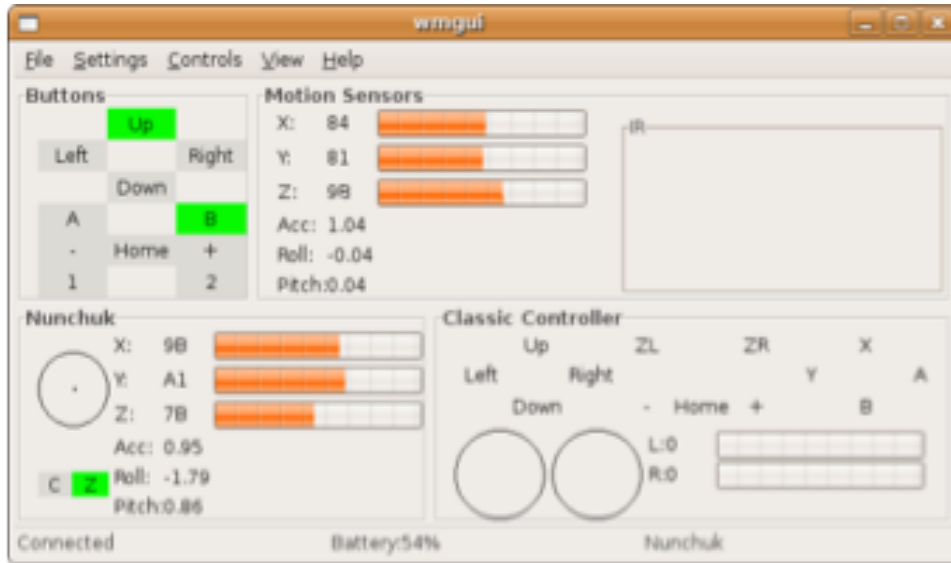


FIGURE 28: CWIID INTERFACE<sup>[12]</sup>

## CHAPTER IV

### CONCLUSIONS

#### **Conclusion**

The popularity of video games has driven development of more sophisticated software technology that supports higher programmer productivity to create visually attractive games capable of displaying realistic scenery. Current game engines support cutting edge graphics and sophisticated audio, physics, and network capabilities. Game engines are a significant contributing factor in achieving realistic and visually attractive sceneries. However, commercial game engines can have large licensing and software costs. This has led independent developers and the open source community to develop alternative software toolkits to create realistic scenery and support game development. At the same time, commercialization and licensing of once open source VR toolkits has resulted in powerful software companies taking over the market, turning VR systems into expensive packages using proprietary software. Such market free market sources have driven many academic researchers and independent developer to open source alternatives for VR development. As with all open source toolkits, public domain game engines and VR development software are sometimes not well documented, toolkits have to be custom tailored to specific applications, and development is a tedious process of combining open source resources to fit the application.

The VizLab system is a packaged customized set of public domain toolkits focusing on immersive system development that provides an open source solution to support the development

of VR applications. VizLab removes many of the challenges of integrating and configuring open source toolkits by providing a packaged suite for VR, VE, and IV applications. VizLab uses game engine tools, such as a game editor and physics module, integrated with open source VR software to create the foundation of a relatively easy to configure system.. Since it is built on open source tools, it provides developers the abilities capabilities to extend the capabilities of VizLab. By integrating game engine methodology, it provides essential tools for creating VR, VR, and IV applications. Having a graphical package integrated in VizLab allows creation of realistic scenes and applications. The physics engine allows for seamless interaction between virtual objects and virtual interaction within any application. Without these tools and their integration in VizLab, creating open source applications with all of these capabilities can be exceptionally time consuming. Having cross-platform capabilities make developers feel at home because they can use their preferred IDE development kits to develop VR applications.

### **Future Work**

VizLab capabilities can be further refined. By combining several modules to work in conjunction, configuring the system and scene development can be dramatically reduced. OsgEdit's API can be extended by adding new XML based middleware and VRJConfig is based on XML schema. Combining the two modules would require building a new extension element for OsgEdit that would emulate the VRJConfig with a window layer on OsgEdit. Figure 29 shows VRJConfig and OsgEdit GUI design. Combining the two would allow programmers to configure the system as they view the scene and environment. This would apply when creating multiple display configurations since it would allow the programmer to see calculations for each viewport in 3D space. In developing with VizLab there have been times when even a minor error in viewport calculation, especially in a clustered system, can cause a particular viewport to

malfunction and display odd camera angles. Combining these two modules would reduce such problems and maximize efficiency when designing and configuring an application.

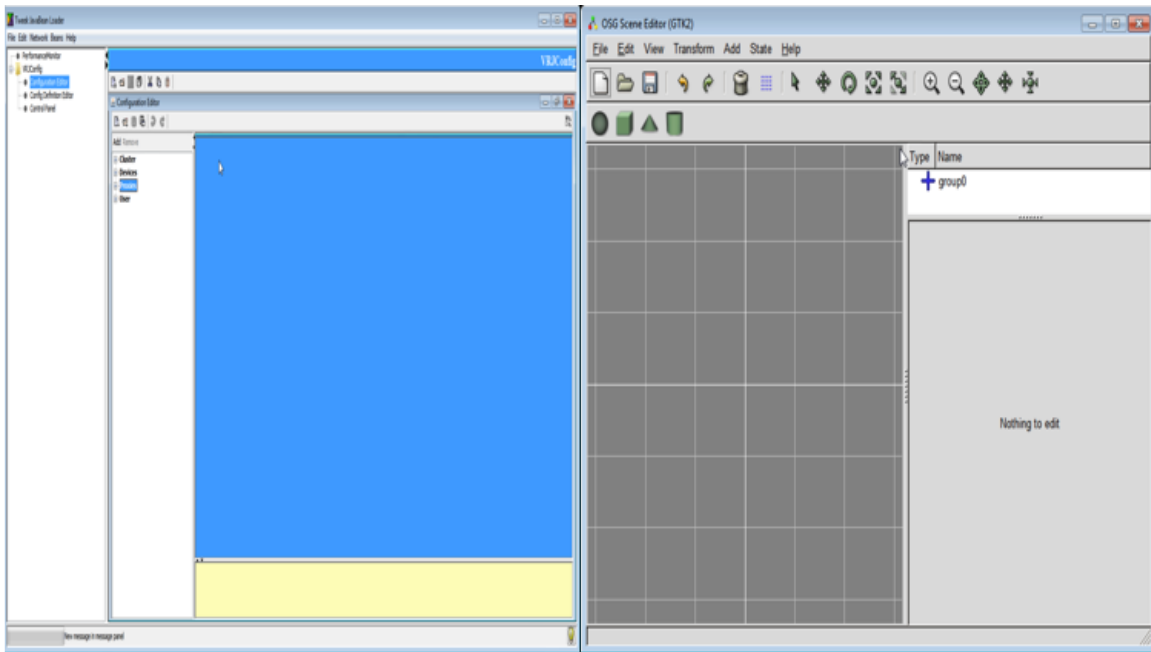


FIGURE 29: VRJCONFIG AND OSGEDIT

Were OsgEdit capable of configuring an application, it would be useful to extend the module for creating a higher layer of code for predetermined devices in the configuration and tying them to specific scene objects. This would allow selecting a particular object in the scene and tying it to a specific configuration element. For example, as you can see from Figure 26?, there is an avatar simulating a user. The avatar consists of multiple objects: head, arms, legs, hands, and feet. Within the configuration it is necessary to specify multiple instances for such objects such as VJHead, VJArms, VJLegs, and VJFeet. In this example these objects are only tied into parameters passed by a Kinect tracking device. Within the application the developer has to connect each object configured in VRJConfig with the corresponding object in the scene graph. This process is very tedious and can be minimized by extending OsgEdit. It can be further improved by having OsgEdit provide a base code for the configuration, instead of editing VizLab's applications separately. These enhancements can be implemented. However, due to the

time constrictions of the project, it was not able to implement them. Nonetheless, all modules could be extended, making VizLab a starting point that eliminates much of the effort required for advanced VR, VE, and IV systems.

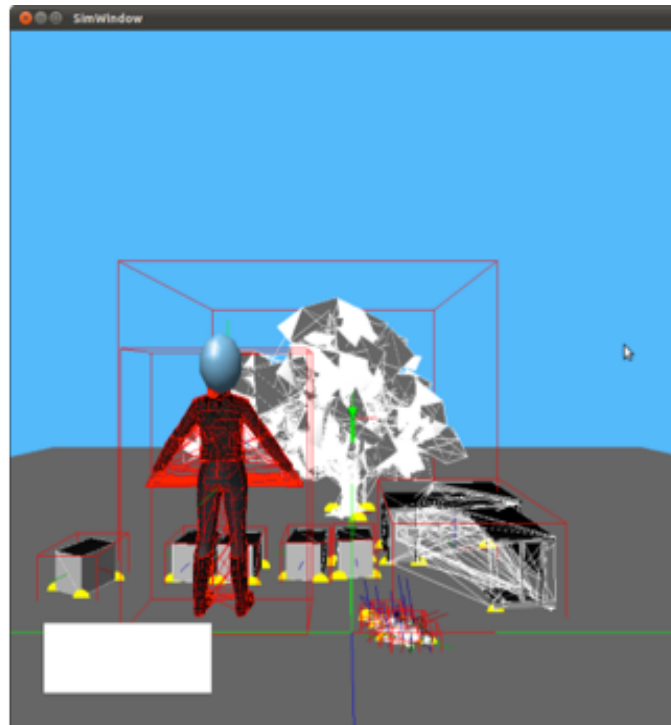


FIGURE 30: OPENSCEENEGRAPH & BULLET

## REFERENCES

1. A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker and C. Cruz-Neira, “VR Juggler: A virtual platform for virtual reality application development;” *Proc. IEEE Conf. Virtual Reality*, pp. 89–96, 2001.
2. Biocca, F., & Delaney, B. (1995). Immersive Virtual Reality Technology. In F. Biocca & M. R. Levy (Eds.), *Communication in the Age of Virtual Reality*. (pp. 57-124). Hillsdale, NJ: Lawrence Erlbaum Associates.
3. Bogacs, Hannes. *Game Mods: a survey of modifications, appropriation and videogame art*. Vienna University of Technology, 2008.
4. Boost C++ Libraries, <http://www.boost.org>, 2013.
5. B. Insko. *Measuring Presence: Subjective, Behavioral and Physiological Methods*. *Being There: Concepts, effects and measurement of user presence in synthetic environments* G. Riva, F. Davide, W.A IJsselsteijn (Eds.) Ios Press, 2003
6. Bullet: Game Physics Simulation, <http://bulletphysics.org/wordpress>, 2013.
7. CMake, <http://www.cmake.org>, 2013.
8. CppDom, <http://sourceforge.net/projects/xml-cppdom>, 2013.
9. CppUnit, <http://sourceforge.net/projects/cppunit/>, 2013.
10. Cruz-Neira C., Sandin D., DeFanti T., Kenyon R. and Hart J., “The CAVE: Audio visual experience automatic virtual Environment;” *Communications of the ACM*, vol. 35, no. 6, pp. 64-72, 1992.



11. Cruz-Neira C., Sandin D. and DeFanti T., “Surround-screen projection-based virtual reality: The design and implementation of the CAVE,” *SIGGRAPH '93 Proc. 20th Ann. Conf. on Computer Graphics and Interactive Techniques*, pp. 135-142, 1993.
12. CWiid: WiiMote toolkit, <http://abstrakraft.org/cwiid>, 2013.
13. CWiid: WiiMote toolkit Ubuntu Documentation, <https://help.ubuntu.com/community/CWiiD>, 2013.
14. D. Pape, C. Cruz-Neira and M. Czernuszenko, “CAVE user’s guide,” Electronic Visualization Laboratory, University of Illinois at Chicago, 1997.
15. Erikson, Carl. Hierarchical Levels of Detail to Accelerate the Rendering of Large Static and Dynamic Polygonal Environments.
16. E. Suma, B. Lange, A. Rizzo, D. Krum, and M. Bolas, “FAAST: The Flexible Action and Articulated Skeleton Toolkit,” *Proceedings of IEEE Virtual Reality*, pp. 247-248, 2011.
17. OpenGL Utility Toolkit (GLUT), <http://www.opengl.org/resources/libraries/glut>, 2013
18. GML Libraries, <http://ggt.sourceforge.net/gmlReferenceGuide-0.6.1-html/main.html>, 2013.
19. Jacobson J. and Lewis L., “Game engine virtual reality with CaveUT,” *IEEE Computer*, vol. 38, no. 4, pp. 79-82, 2005.
20. Java Development Kit (JDK), <http://www.oracle.com/us/technologies/java/overview/index.html>, 2013.
21. Juarez A., Schonenberg W. and Bartneck C., “Implementing a low-cost CAVE system using the CryEngine2”, *Entertainment Computing*, vol. 1, pp. 157–164, 2010.
22. Luebke, D., Watson, B., Cohen J., Reddy M., Varshney A., *Level of Detail for 3D Graphics*, Elsevier Science Inc., New York, NY, 2002

23. Lewis M., and Jacobson J., "Game Engines in Scientific Research" (Special Issue: Game Engines in Scientific Research), *Communications of the ACM* 45, No. 1 (January 2002): 27.
24. Mooney, T. *Unreal Development Kit Game Design Cookbook*. Packt, 2012.
25. OpenAL, <http://connect.creativelabs.com/openal/default.aspx>, 2013.
26. OpenGL, <http://www.opengl.org>, 2013.
27. OpenSceneGraph, <http://www.openscenegraph.org>, 2013.
28. OpenSG, <http://www.opensg.org>, 2013.
29. OSGBullet, <http://code.google.com/p/osgbullet>, 2013.
30. OSGEEdit: OpenScenegraph Scene Editor, <http://osgedit.sourceforge.net>, 2013.
31. OSGWorks, <http://code.google.com/p/osgworks>, 2013.
32. OmniORB, <http://omniorb.sourceforge.net>, 2013.
33. Preddy, S.M., Nance, R.E., "Key requirements for CAVE simulations," Simulation Conference, 2002. Proceedings of the Winter , vol.1, no., pp. 127- 135 vol.1, 8-11 Dec. 2002
34. Taylor, R., Hudson, T., Seeger, A., Weber, H., Juliano, J., Helser, A., "VRPN: a device-independent, network-transparent VR peripheral system." Proc. ACM Symposium on Virtual Reality Software and Technology 2001:55-61.
35. Shiratuddin, M. F., & Thabet, W. (2002). Virtual Office Walkthrough using a 3D Game Engine. *International Journal of Design Computing*, pp. 1 - 25.
36. Sutherland, I. E. "A Head Mounted Three Dimensional Display", Proceedings of the Fall Joint Computer Conference (1968).

37. Sutherland, I.E. The Ultimate Display. Proc. IFIP 65, 2, pp. 506-508,582-583.
38. UDK – Unreal Development Kit – Epic Games, <http://udk.com/>, 2013.
39. Unity Game Engine, <http://unity3d.com>, 2013
40. vrjuggler, <http://code.google.com/p/vrjuggler/>, 2013.
41. VRPN, <http://www.cs.unc.edu/Research/vrpn/>, 2013.
42. W. Krueger and B. Froehlich, "The Responsive Workbench: A Virtual Work Environment," *IEEE Computer Graphics and Applications*, Vol. 14, No. 3, May 1994, pp. 12-15.
43. Wingrave, C. A. et al. (2010). The Wiimote and Beyond: Spatially Convenient Devices for 3D User Interfaces , *IEEE Computer Graphics and Applications*, March/April, 24-38.
44. Witmer, B. G. & Singer, M. J. (1998). “Measuring Presence in Virtual Environments: A Presence Questionnaire”, *Presence*, Vol. 7, No. 3, June 1998, 225–240

## APPENDIX A

## APPENDIX A

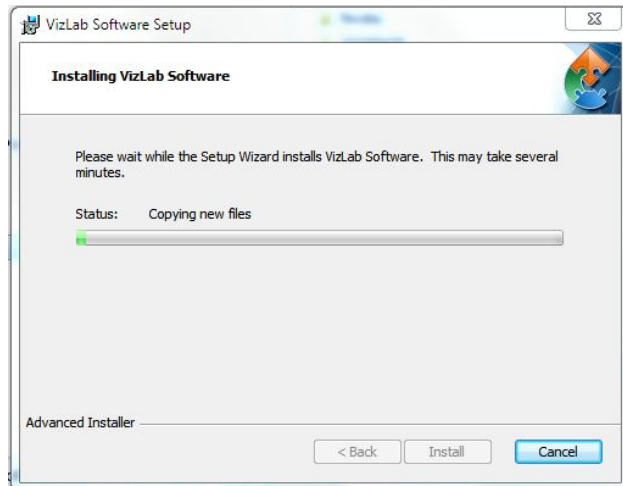
### VIZLAB WINDOWS INSTALLATION

The VizLab Windows installation is done by a MSI Installer package. This installation provides all the binary dependencies for VRJuggler and OpenSceneGraph. It also sets up all the Environment Variables for all the software to work. The package contains all the dependencies and executables. This guide will make a 24-hour job and cuts it into a 10-minute process.

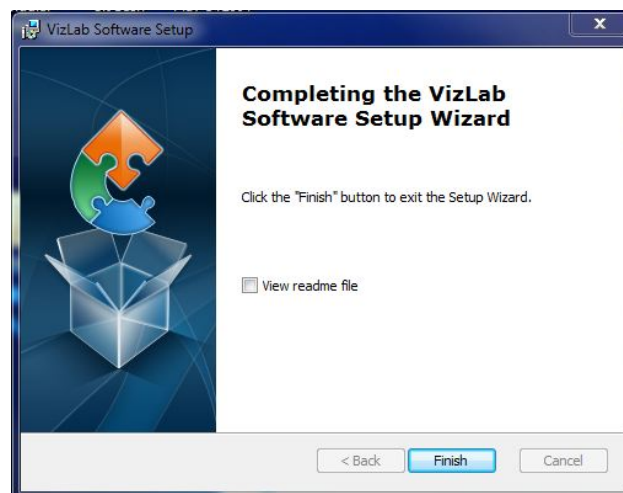
1. Double click on the VizLab\_Software installation package. The VizLab Software Setup Window will pop up.



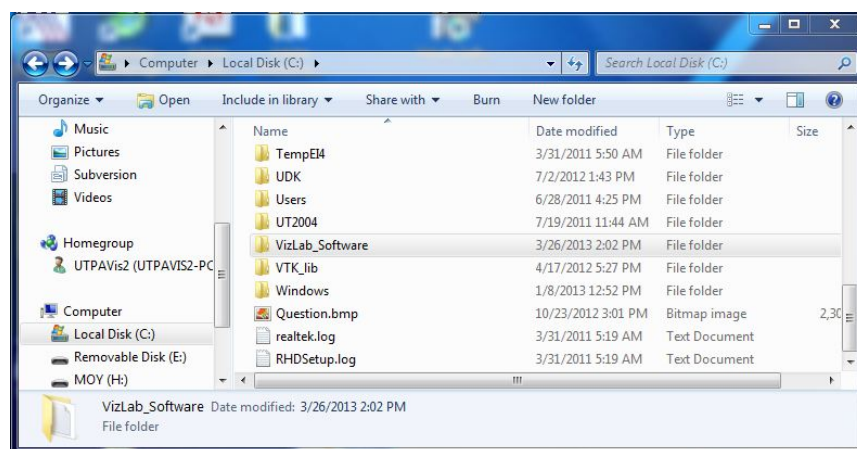
2. Click Install for installation.
3. The installation process will proceed. This can take up to 10 minutes depending on your hardware.



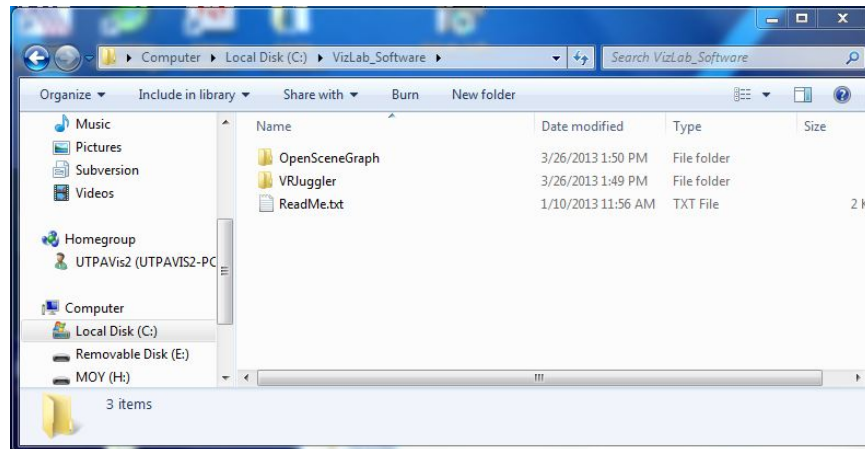
4. Installation finishes and will ask you if you want to view the read me file.



5. There should be a VizLab\_Software folder installed on the C:/ drive.

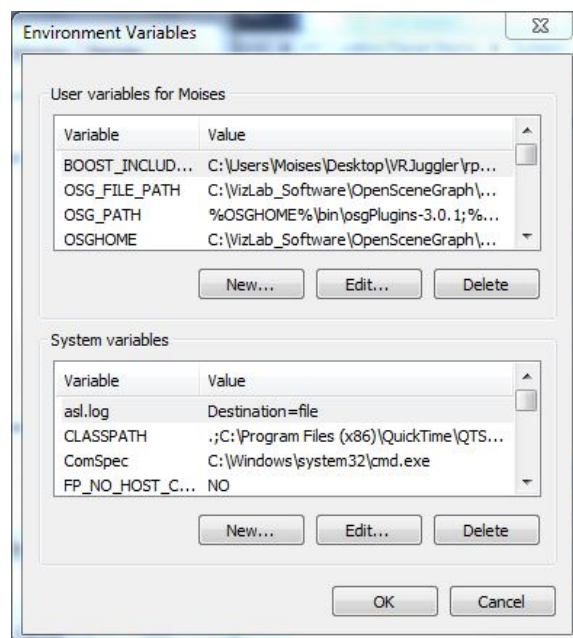


6. Within the VizLab\_Software Folder there should be VRJuggler folder, OpenSceneGraph folder and ReadMe.txt file.



7. Double check the Environment Variables are pointing to the VizLab\_Software Folder.

Go To: Start -> Control Panel -> System -> Environment Variables



9. If they are pointing to their respective parameters then the installation was done correctly.

This installation package includes everything to get started developing VizLab applications.

## APPENDIX B



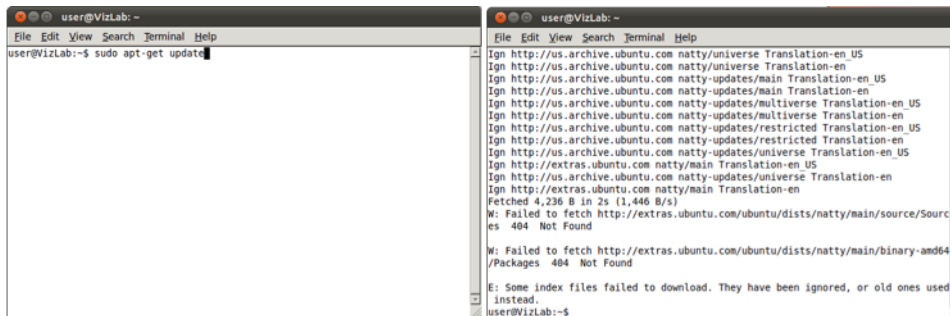
## APPENDIX B

### VIZLAB LINUX INSTALLATION

The following Appendix is the installation of VizLab on UBUNTU Natty 11.04. This installation is in the form of Ubuntu packages, which makes for an easier installation process. There are a couple dependencies that come directly packaged from Ubuntu repositories, so internet connection is required. The installation process should take around 30 to 45 minutes; it varies depending on Internet connection and processor speed. The installation guides you through a series of terminal commands and debian packages to install VizLab software. All debian packages have to be double clicked and then installed through Ubuntu's Software Center. The following steps are:

First Copy the folder VIZLAB\_Software folder into the Desktop and open up the Terminal Console. The terminal console can be reached by going into Applications menu, Accessories sub-menu, and then Terminal. Upgrade the packages:

```
sudo apt-get update
```



```
user@VizLab:~$ sudo apt-get update
Ign http://us.archive.ubuntu.com natty/universe Translation-en_US
Ign http://us.archive.ubuntu.com natty/universe Translation-en
Ign http://us.archive.ubuntu.com natty-updates/main Translation-en_US
Ign http://us.archive.ubuntu.com natty-updates/main Translation-en
Ign http://us.archive.ubuntu.com natty-updates/multiverse Translation-en_US
Ign http://us.archive.ubuntu.com natty-updates/multiverse Translation-en
Ign http://us.archive.ubuntu.com natty-updates/restricted Translation-en_US
Ign http://us.archive.ubuntu.com natty-updates/restricted Translation-en
Ign http://us.archive.ubuntu.com natty-updates/universe Translation-en_US
Ign http://us.archive.ubuntu.com natty-updates/universe Translation-en
Ign http://extras.ubuntu.com natty/main Translation-en_US
Ign http://us.archive.ubuntu.com natty-updates/universe Translation-en
Ign http://extras.ubuntu.com natty/main Translation-en
Fetched 4,236 B in 2s (1,446 B/s)
W: Failed to fetch http://extras.ubuntu.com/ubuntu/dists/natty/main/source/Sources
 404 Not Found
W: Failed to fetch http://extras.ubuntu.com/ubuntu/dists/natty/main/binary-amd64
/Packages 404 Not Found
E: Some index files failed to download. They have been ignored, or old ones used
instead.
user@VizLab:~$
```

After getting the update, the next step is installing VRJuggler Dependencies. Install uuid packages from Ubuntu apt repository

```
Command: sudo apt-get install uuid
```

```
Command: sudo apt-get install uuid-dev
```

```

user@VizLab:~$ sudo apt-get install uid
The following NEW packages will be installed:
 libossp-uid16 uid
0 upgraded, 2 newly installed, 0 to remove and 320 not upgraded.
Need to get 83.4 kB of archives.
After this operation, 254 kB of additional disk space will be used.
Do you want to continue [Y/n]? y
Get:1 http://us.archive.ubuntu.com/ubuntu/ natty/main libossp-uid16 amd64 1.6.2-1ubuntu1 [61.8 kB]
Get:2 http://us.archive.ubuntu.com/ubuntu/ natty/universe uid amd64 1.6.2-1ubuntu1 [21.6 kB]
Fetched 83.4 kB in 0s (135 kB/s)
Selecting previously deselected package libossp-uid16.
(Reading database ... 129621 files and directories currently installed.)
Unpacking libossp-uid16 (from .../libossp-uid16_1.6.2-1ubuntu1_amd64.deb) ...
Selecting previously deselected package uid.
Unpacking uid (from .../uid_1.6.2-1ubuntu1_amd64.deb) ...
Processing triggers for man-db ...
Setting up libossp-uid16 (1.6.2-1ubuntu1) ...
Setting up uid (1.6.2-1ubuntu1) ...
Processing triggers for libc-bin ...
ldconfig deferred processing now taking place
user@VizLab:~$

user@VizLab:~$ sudo apt-get install uid-dev
user@VizLab:~$ sudo apt-get install uid-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
 uid-dev
0 upgraded, 1 newly installed, 0 to remove and 320 not upgraded.
Need to get 28.3 kB of archives.
After this operation, 201 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu/ natty/main uid-dev amd64 2.17.2-9.1ubuntu4 [28.3 kB]
Fetched 28.3 kB in 0s (95.0 kB/s)
Selecting previously deselected package uid-dev.
(Reading database ... 129635 files and directories currently installed.)
Unpacking uid-dev (from .../uid-dev_2.17.2-9.1ubuntu4_amd64.deb) ...
Setting up uid-dev (2.17.2-9.1ubuntu4) ...
user@VizLab:~$

```

Install cmake packages from Ubuntu apt repository

Command: `sudo apt-get install cmake`

```

user@VizLab:~$ sudo apt-get install cmake
(Reading database ... 129656 files and directories currently installed.)
Unpacking libcurl3 (from .../libcurl3_7.21.3-1ubuntu1.5_amd64.deb) ...
Selecting previously deselected package libxmlrpc-core-c3-0.
Unpacking libxmlrpc-core-c3-0 (from .../libxmlrpc-core-c3-0_1.16.32-0ubuntu3.1_amd64.deb) ...
Selecting previously deselected package emacsen-common.
Unpacking emacsen-common (from .../emacsen-common_1.4.19ubuntu2_all.deb) ...
Selecting previously deselected package cmake-data.
Unpacking cmake-data (from .../cmake-data_2.8.3-3ubuntu7_all.deb) ...
Selecting previously deselected package cmake.
Unpacking cmake (from .../cmake_2.8.3-3ubuntu7_amd64.deb) ...
Processing triggers for man-db ...
Setting up libcurl3 (7.21.3-1ubuntu1.5) ...
Setting up libxmlrpc-core-c3-0 (1.16.32-0ubuntu3.1) ...
Setting up emacsen-common (1.4.19ubuntu2) ...
emacsen-common: Handling install of emacsen flavor emacs
Setting up cmake-data (2.8.3-3ubuntu7) ...
emacsen-common: Handling install of emacsen flavor emacs
Setting up cmake (2.8.3-3ubuntu7) ...
Processing triggers for libc-bin ...
ldconfig deferred processing now taking place
user@VizLab:~$

```

Install Autoconf packages from Ubuntu apt repository

Command: `sudo apt-get install autoconf`

```
user@VizLab:~$ sudo apt-get install autoconf
Fetched 1,473 kB in 1s (839 kB/s)
Selecting previously deselected package m4.
(Reading database ... 130256 files and directories currently installed.)
Unpacking m4 (from .../archives/m4_1.4.14-3 amd64.deb) ...
Selecting previously deselected package autoconf.
Unpacking autoconf (from .../autoconf_2.67-2ubuntu1_all.deb) ...
Selecting previously deselected package autotools-dev.
Unpacking autotools-dev (from .../autotools-dev_2010122.1_all.deb) ...
Selecting previously deselected package automake.
Unpacking automake (from .../automake_1:3.1.1-1-ubuntu1_all.deb) ...
Processing triggers for install-info ...
Processing triggers for man-db ...
Processing triggers for doc-base ...
Processing 34 changed 1 added doc-base file(s)...
Registering documents with scrollkeeper...
Setting up m4 (1:4.14-3) ...
Setting up autoconf (2.67-2ubuntu1) ...
Setting up autotools-dev (2010122.1) ...
Setting up automake (1:1.11.1-ubuntu1) ...
update-alternatives: using /usr/bin/automake-1.11 to provide /usr/bin/automake (
automake) in auto mode.
user@VizLab:~$
```

Install JDK(Java Libraries) from Ubuntu apt repository:

Command: `sudo apt-get install default-jdk`

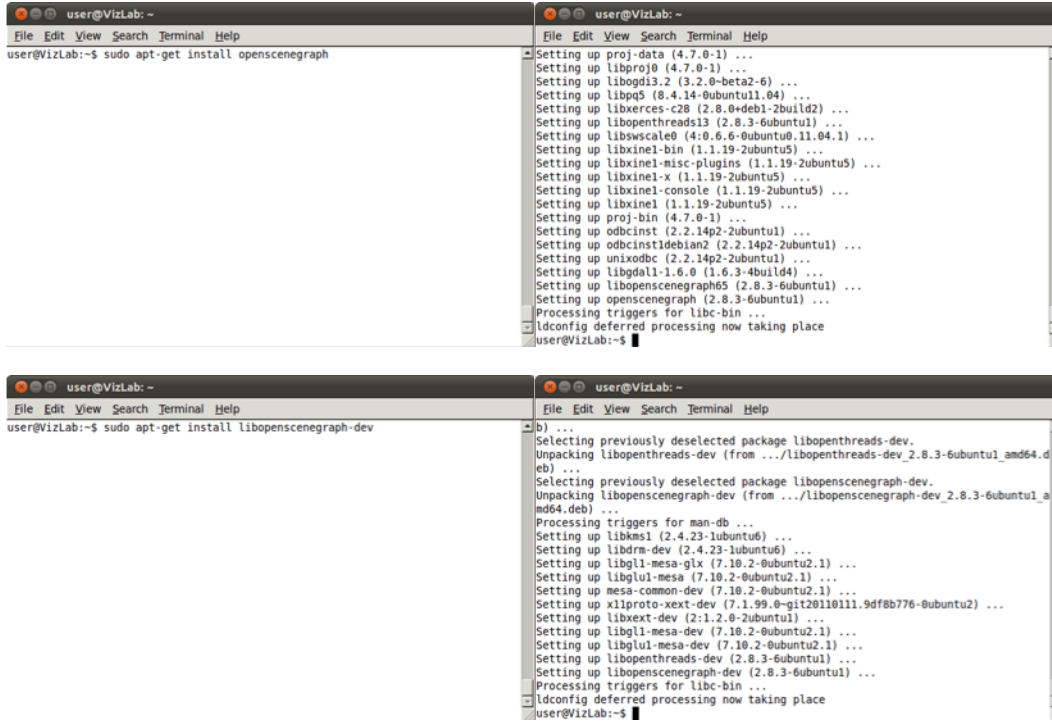
```
user@VizLab:~$ sudo apt-get install default-jdk
/bin/wsgen (wsgen) in auto mode.
update-alternatives: using /usr/lib/jvm/java-6-openjdk/bin/wsgen to provide /
usr/bin/wsgen (wsgen) in auto mode.
update-alternatives: using /usr/lib/jvm/java-6-openjdk/bin/xjc to provide /usr/b
in/xjc (xjc) in auto mode.
Setting up icedtea-6-jre-cacao (6b24-1.11.5-0ubuntu1-11.04.1) ...
Setting up icedtea-6-jre-javm (6b24-1.11.5-0ubuntu1-11.04.1) ...
Setting up icedtea-netx (1:2-2ubuntu0.11.04.3) ...
update-alternatives: using /usr/lib/jvm/java-6-openjdk/jre/bin/javaws to provide
/usr/bin/javaws (javaws) in auto mode.
update-alternatives: using /usr/lib/jvm/java-6-openjdk/jre/bin/itweb-settings to
provide /usr/bin/itweb-settings (itweb-settings) in auto mode.
Setting up default-jre-headless (1:1.6-40ubuntu1) ...
Setting up ca-certificates-java (20100412ubuntu0.11.04.1) ...
creating /etc/ssl/certs/java/cacerts...
added certificate mozilla/DigiNotar_Root_CA.crt
done.
Setting up default-jre (1:1.6-40ubuntu1) ...
Setting up default-jdk (1:1.6-40ubuntu1) ...
Processing triggers for libc-bin ...
ldconfig deferred processing now taking place
user@VizLab:~$
```

Install Boost Libraries from the Ubuntu repository:

Command: `sudo apt-get install libboost-all-dev`

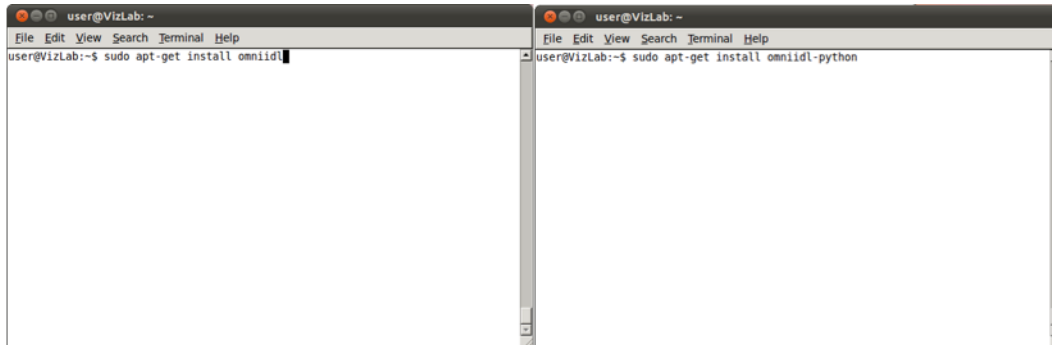
```
user@VizLab:~$ sudo apt-get install libboost-all-dev
Setting up python-dev (2.7.1-0ubuntu5.1) ...
Setting up libboost-python1.42-dev (1.42.0-4ubuntu2) ...
Setting up libboost-python-dev (1.42.0-1ubuntu1) ...
Setting up libboost-regex-dev (1.42.0-1ubuntu1) ...
Setting up libboost-serialization-dev (1.42.0-1ubuntu1) ...
Setting up libboost-signals1.42.0 (1.42.0-4ubuntu2) ...
Setting up libboost-signals1.42-dev (1.42.0-4ubuntu2) ...
Setting up libboost-signals-dev (1.42.0-1ubuntu1) ...
Setting up libboost-system-dev (1.42.0-1ubuntu1) ...
Setting up libboost-test-dev (1.42.0-1ubuntu1) ...
Setting up libboost-thread1.42.0 (1.42.0-4ubuntu2) ...
Setting up libboost-thread1.42-dev (1.42.0-4ubuntu2) ...
Setting up libboost-thread-dev (1.42.0-1ubuntu1) ...
Setting up libboost-wave1.42.0 (1.42.0-4ubuntu2) ...
Setting up libboost-wave1.42-dev (1.42.0-4ubuntu2) ...
Setting up libboost-wave-dev (1.42.0-1ubuntu1) ...
Setting up libboost-all-dev (1.42.0-1ubuntu1) ...
Setting up libboost-graph1.42.0 (1.42.0-4ubuntu2) ...
Processing triggers for libc-bin ...
ldconfig deferred processing now taking place
Processing triggers for python-support ...
user@VizLab:~$
```

Install OpenSceneGraph from the Ubuntu repository:  
Command: `sudo apt-get install openscenegraph`  
Command: `sudo apt-get install libopenscenegraph-dev`



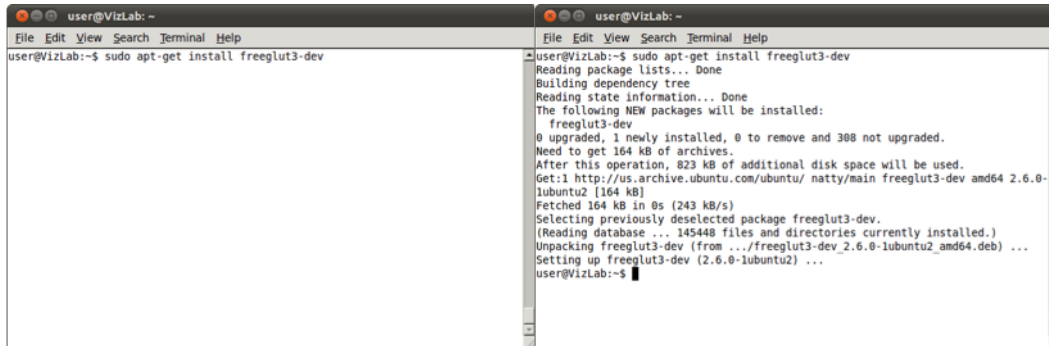
Install Omniidl from the Ubuntu repository:

Command: `sudo apt-get install omniidl`  
Command: `sudo apt-get install omniidl-python`



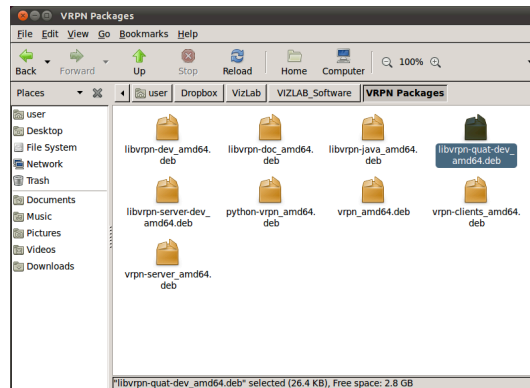
Install Glut libraries from the Ubuntu repository:

Command: `sudo apt-get install freeglut3-dev`



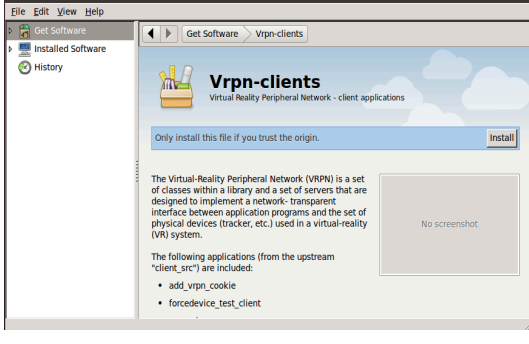
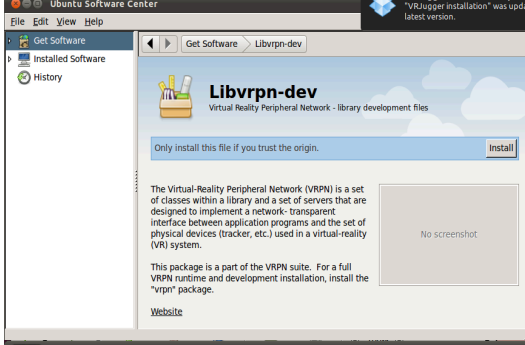
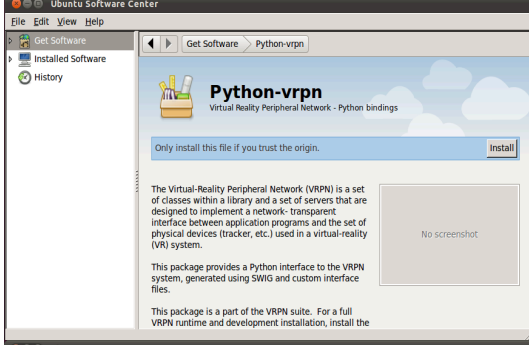
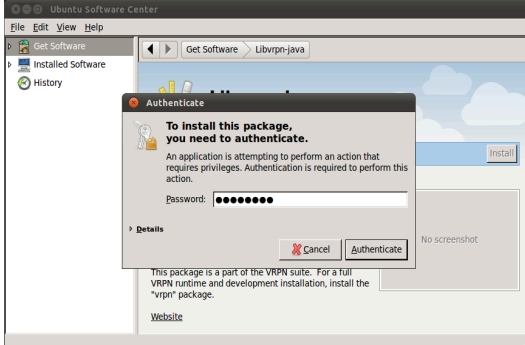
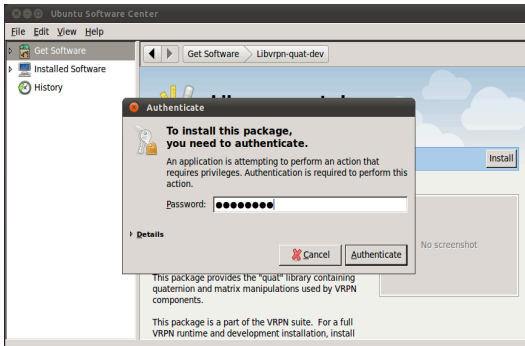
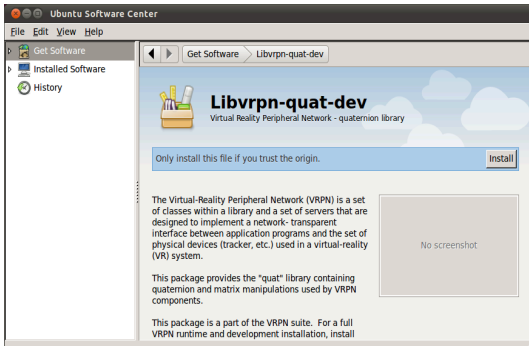
```
user@VizLab:~$ sudo apt-get install freeglut3-dev
user@VizLab:~$ sudo apt-get install freeglut3-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  freeglut3-dev
0 upgraded, 1 newly installed, 0 to remove and 308 not upgraded.
Need to get 164 kB of archives.
After this operation, 823 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu/natty/main freeglut3-dev amd64 2.6.0-1ubuntu2 [164 kB]
Fetched 164 kB in 0s (243 kB/s)
Selecting previously deselected package freeglut3-dev.
(Reading database ... 145448 files and directories currently installed.)
Unpacking freeglut3-dev (from .../freeglut3-dev_2.6.0-1ubuntu2_amd64.deb) ...
Setting up freeglut3-dev (2.6.0-1ubuntu2) ...
user@VizLab:~$
```

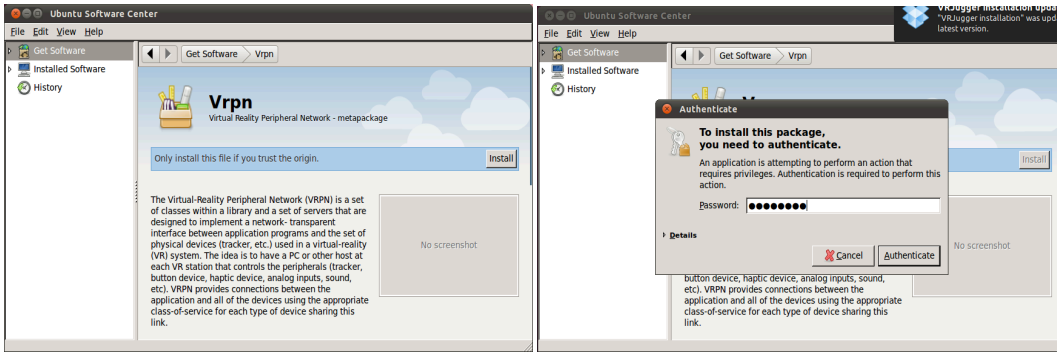
Install VRPN Packages :



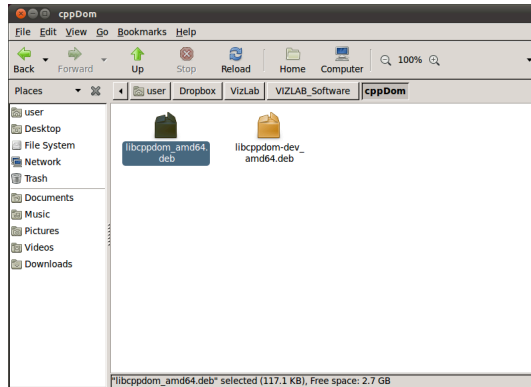
This is in the VIZLAB\_Software folder that you copied into the Desktop. In the VIZLAB\_Software folder there is a sub-folder called VRPN Packages. There are a number of packages in the VRPN folder. The Packages have to be installed in this particular order.

- Double Click: libvrpn-quat-dev\_amd64.deb
- Double Click: libvrpn-java\_amd64.deb
- Double Click: python-vrpn\_amd64.deb
- Double Click: libvrpn-dev\_amd64.deb
- Double Click: vrpn-server\_amd64.deb
- Double Click: libvrpn-server\_amd64.deb
- Double Click: vrpn-clients\_amd64.deb
- Double Click: libvrpn-doc\_amd64.deb
- Double Click: vrpn\_amd64.deb





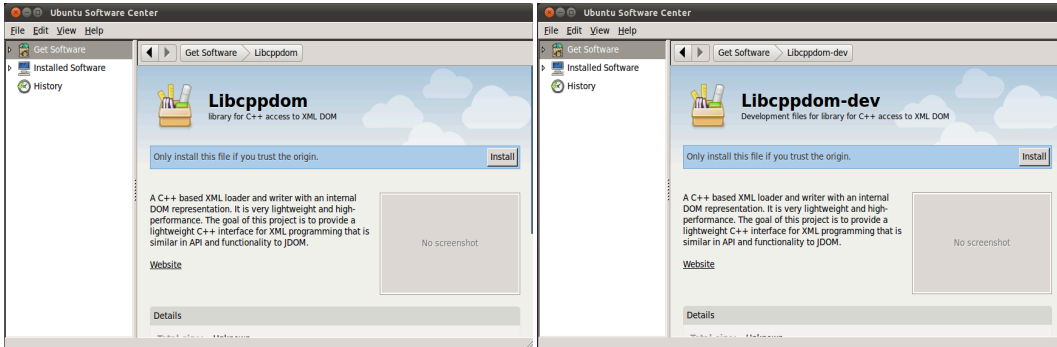
## Install CppDom Packages:



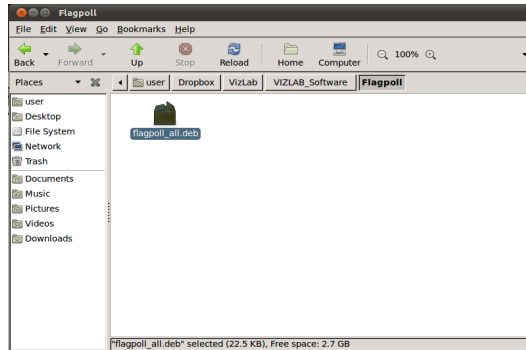
This is in the VIZLAB\_Software Folder in a folder called cppDom.  
The Packages have to be installed in this particular order.

Double Click: libcppdom\_amd64.deb

Double Click: libcppdom-dev\_amd64.deb

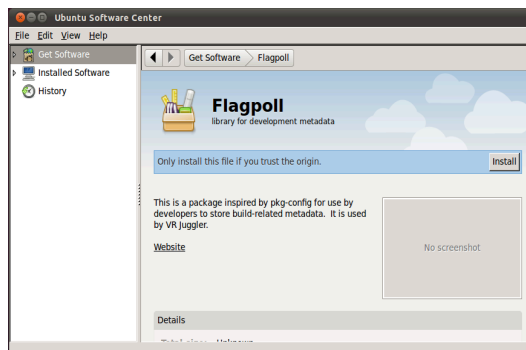


## Install Flagpoll Package:

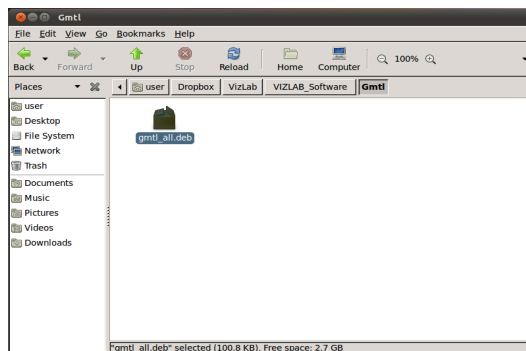


This is in the VIZLAB\_Software Folder in a folder called flagpoll.

Double Click: flagpoll\_all.deb



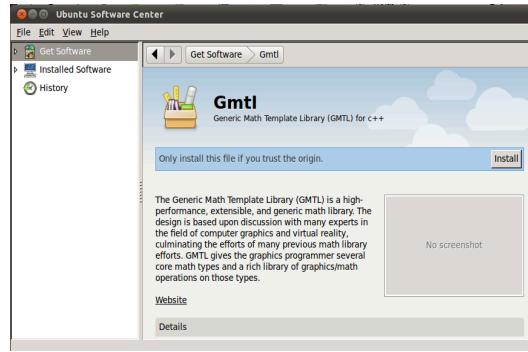
## Install Gmtl Package:



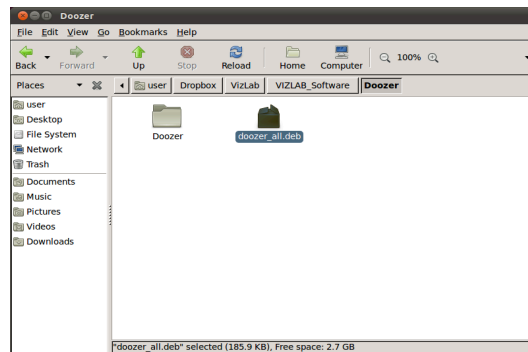
This is in the VIZLAB\_Software Folder in a folder called Gmtl.

Double Click: gmtl\_all.deb



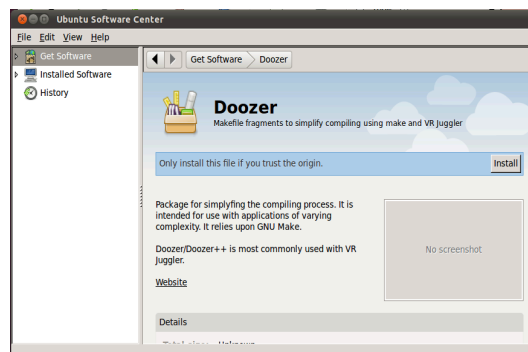


Install Doozer Package:



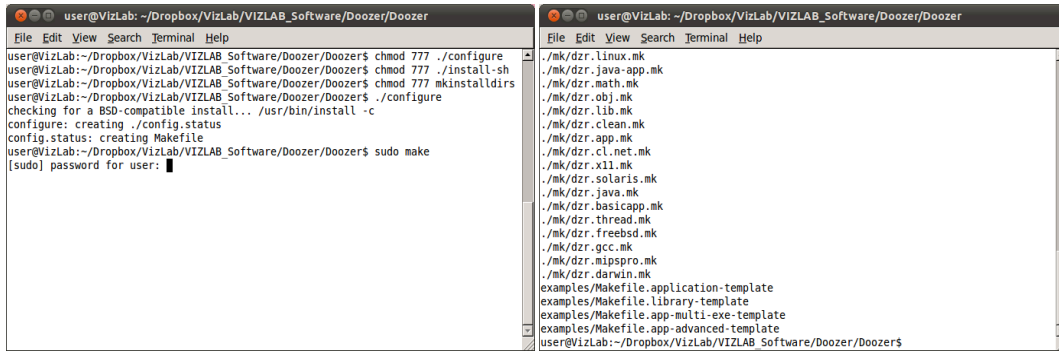
This is in the VIZLAB\_Software Folder in a folder called Doozer.

Double Click: doozer\_all.deb



Through the Command line go into the Doozer/Doozer folder.

```
Command: chmod 777 ./configure  
Command: chmod 777 ./install-sh  
Command: chmod 777 mkinstaldirs  
Command: ./configure  
Command: sudo make
```



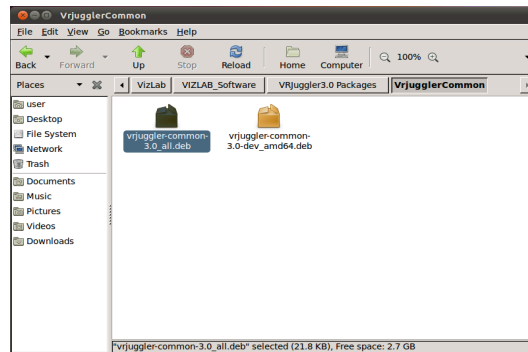
## Install VrJuggler Packages:

This is in the VIZLAB\_Software Folder in a folder called VRJuggler3.0 Packages. This folder is composed of Gadgeteer, Jccl, Sonix, Tweek, Vpr, Vrj, Vrjuggler folders.

The installations of these packages have to be installed in a particular order in order for VRJuggler to be installed. It is essential that you have already installed the previous Dependencies, if not, these packages cannot be installed due to the dependences to the previous packages.

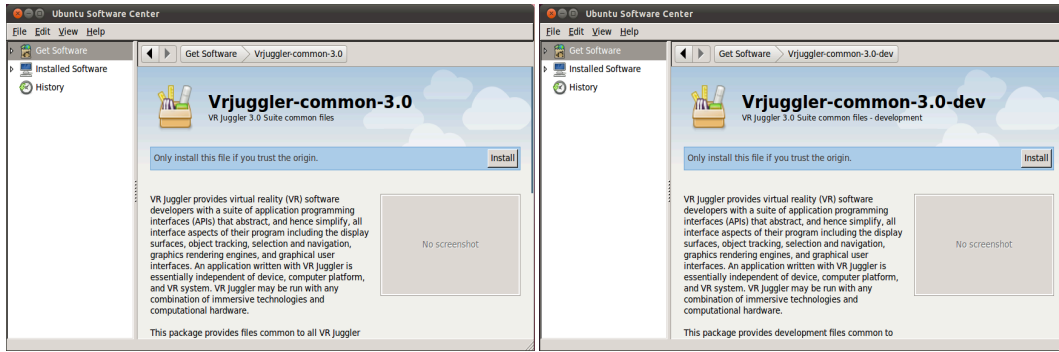
The order is following:

## Open VrJugglerCommons Folder:

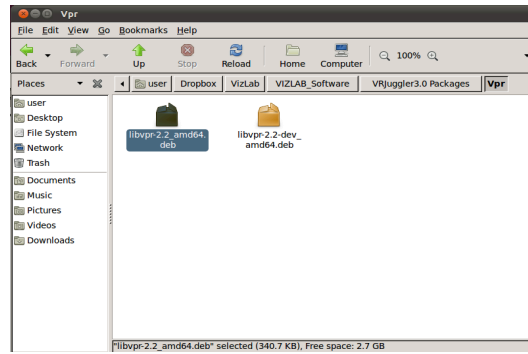


Double Click: vrjuggler-common-3.0\_all.deb

Double Click: vrjuggler-common-3.0-dev\_amd64.deb

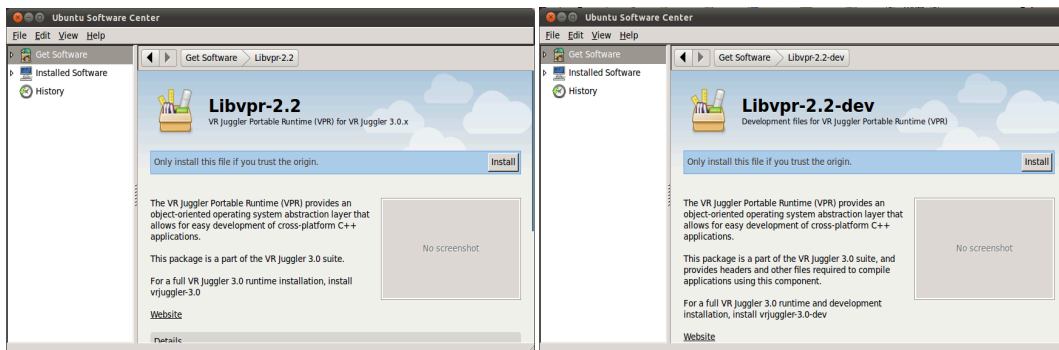


Open Vpr Folder:

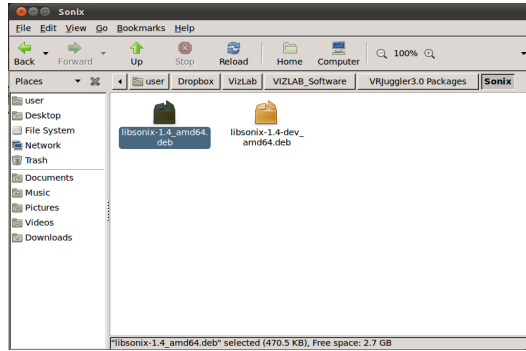


Double Click: libvpr-2.2-amd64.deb

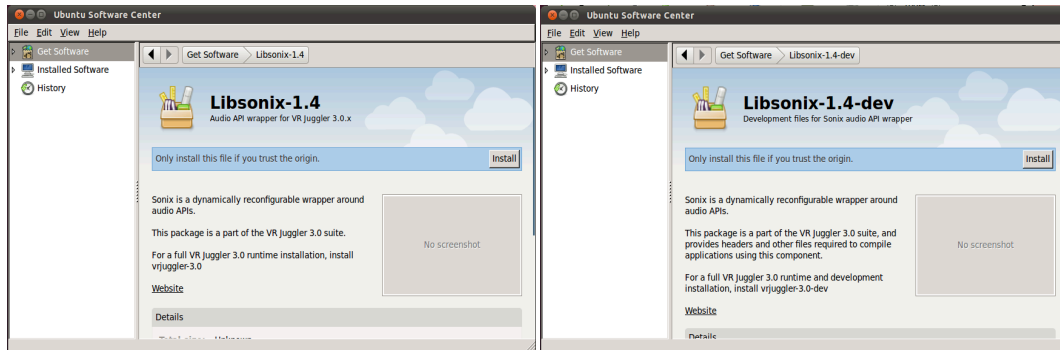
Double Click: libvpr-2.2-dev-amd64.deb



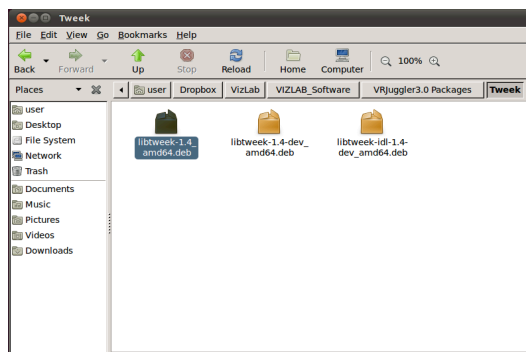
Open Sonix Folder:



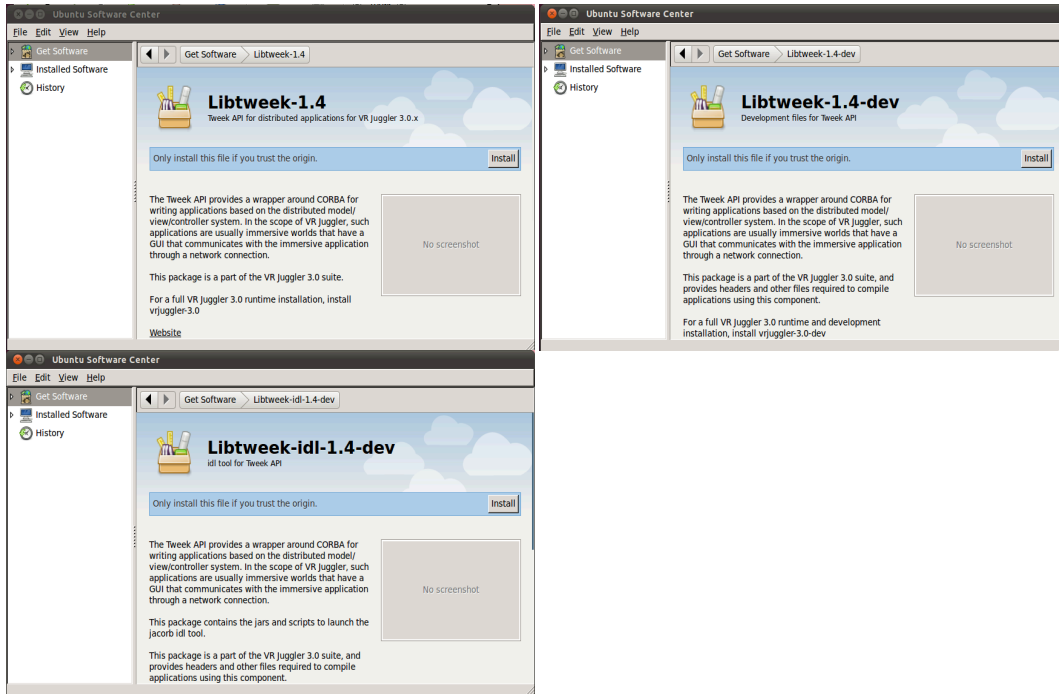
Double Click: libsonix-1.4\_amd64.deb  
Double Click: libsonix-1.4-dev\_amd64.deb



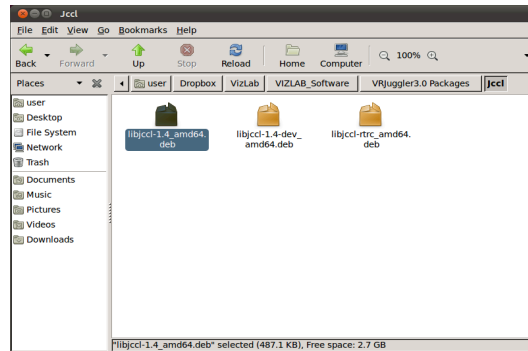
Open Tweek Folder:



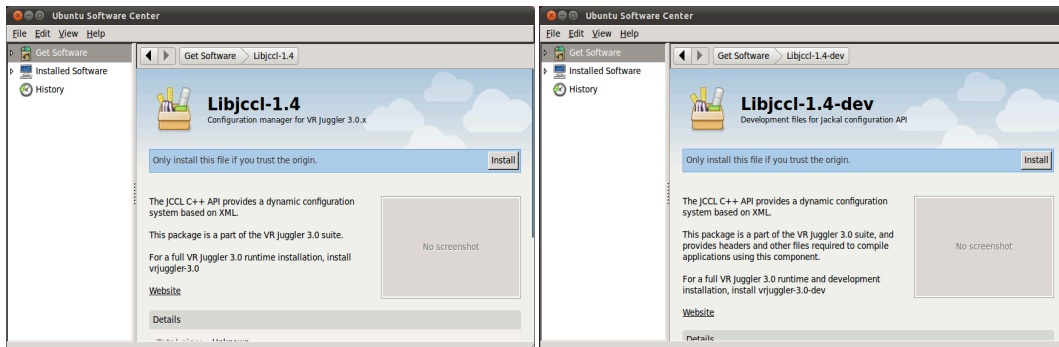
Double Click: libtweek-1.4\_amd64.deb  
Double Click: libtweek-1.4-dev\_amd64.deb  
Double Click: libtweek-idl-1.4-dev\_amd64.deb

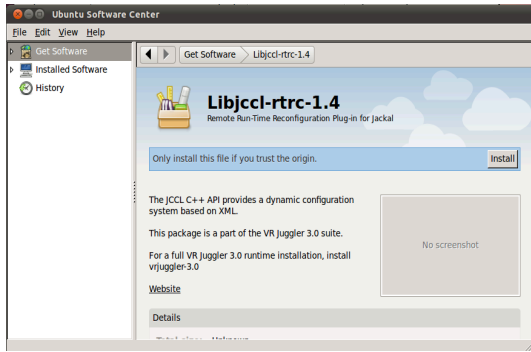


Open Jccl Folder:

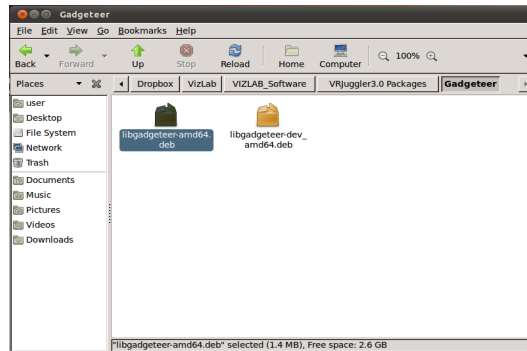


Double Click: libjccl-1.4\_amd64.deb  
 Double Click: libjccl-1.4-dev\_amd64.deb  
 Double Click: libjccl-rtrc\_amd64.deb

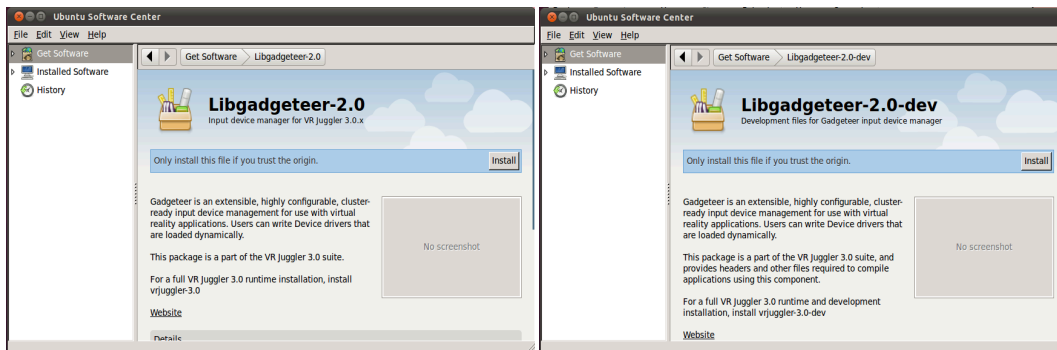




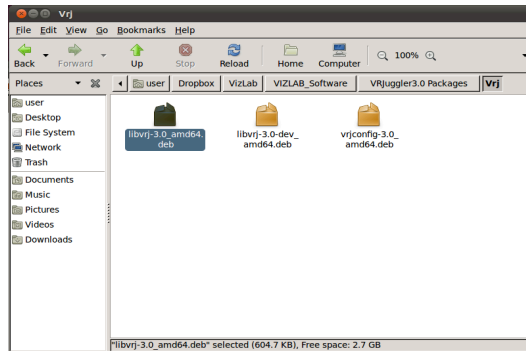
Open Gadgeteer Folder:



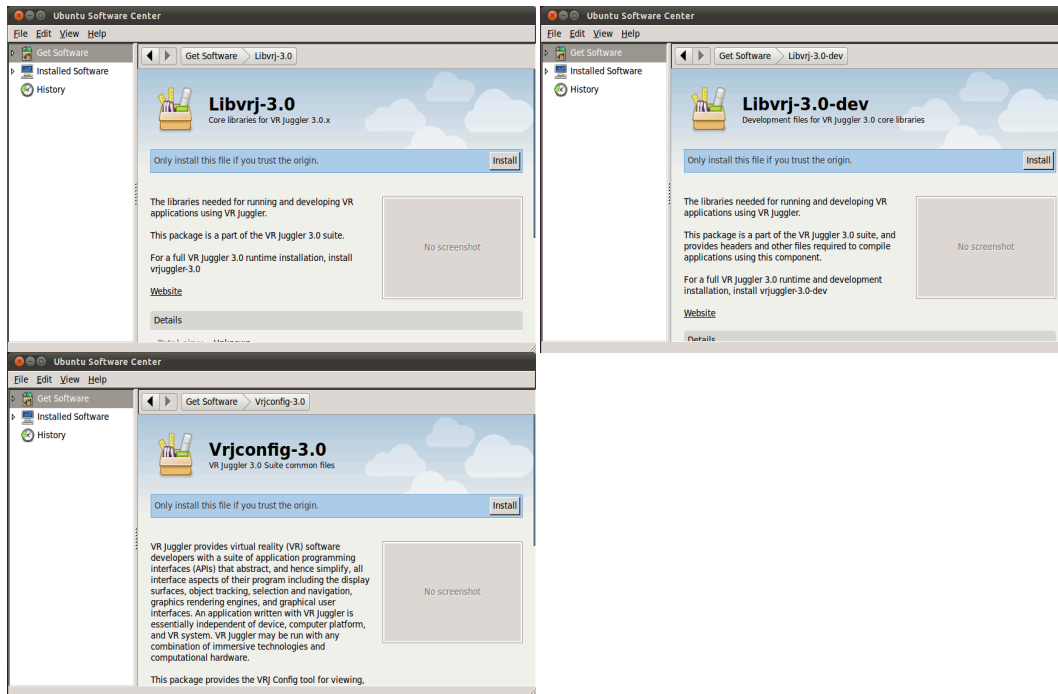
Double Click: libgadgeteer-amd64.deb  
Double Click: libgadgeteer-dev\_ amd64.deb



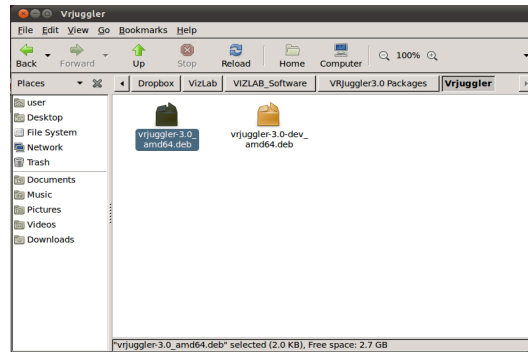
Open Vrj Folder:



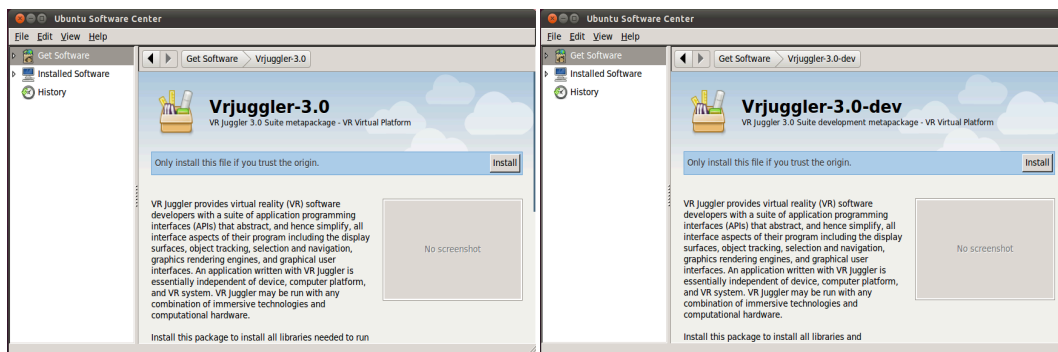
Double Click: libvrj-3.0\_amd64.deb  
Double Click: libvrj-3.0-dev\_amd64.deb  
Double Click: vrjconfig-3.0\_amd64.deb



Open Vrjuggler Folder:



Double Click: vrjuggler-3.0\_amd64.deb  
Double Click: vrjuggler-3.0-dev\_amd64.deb



VizLab Software should be installed after installing VRJuggler. Now you should be able to run VizLab Applications. All environment variables are set into the system and the application folders are installed into the /usr/bin, /usr/include, and /usr/share folders.

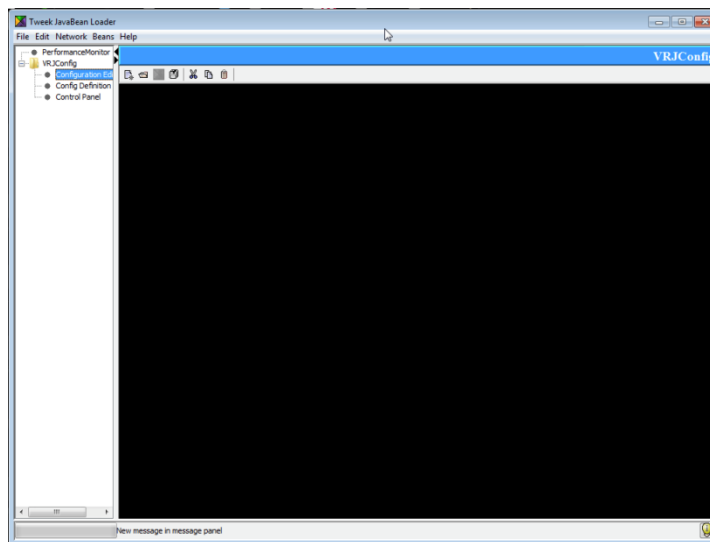


## APPENDIX C

## APPENDIX C

### VIZLAB CONFIGURATION WITH VRJCONFIG

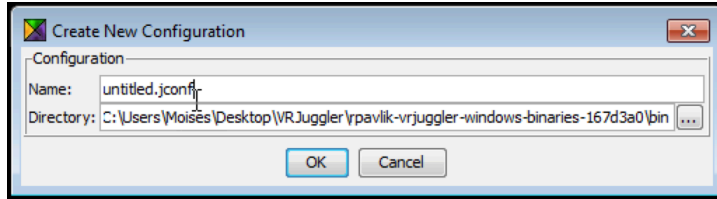
This Appendix provides details for configuring VizLab system with VRJuggler's VRJConfig. VRJConfig can be found in Windows VizLab\_Software -> VRJuggler -> Bin Folder or Linux Platform in /usr -> bin Folder . In the folder you will see a file called VRJConfig.bat(Windows) or VRJConfig.sh(Linux), this file is a batch executable or shell script file, double clicking should start execution and should display VRJConfig GUI on the screen.



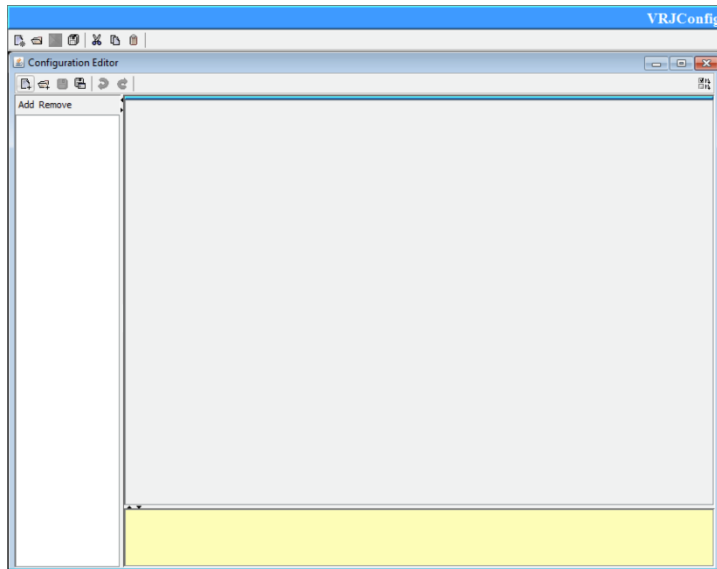
VRJConfig starts with an empty project. You can create a new configuration, open a configuration file, save configuration, cut, copy and paste within the menu bar on top.



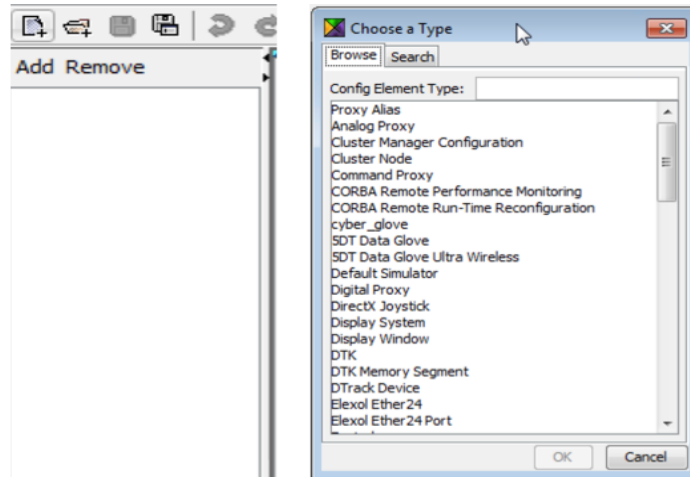
Creating a new Configuration will prompt you for the name and the path for the file to be saved. Configurations are XML files and saved with a .jconf extension.



After entering the name and the path of the file to be saved, the configuration editor should appear within the right panel of VRJConfig window.



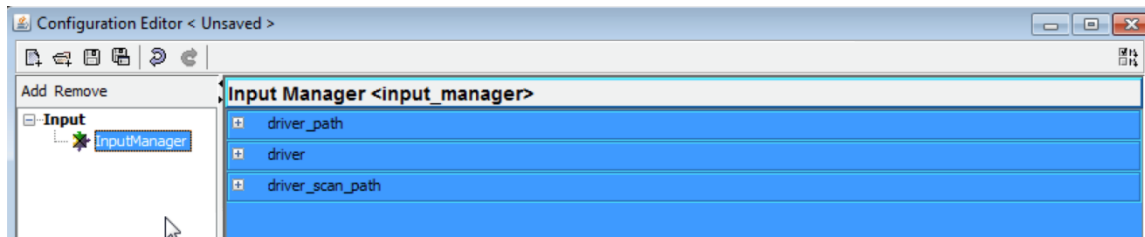
A Configuration is a collection of configuration elements that provide a complete set of parameters needed to execute a VR application. On the left hand side, you should see the Configuration Element Navigator Panel; within the Element Navigation Panel you can add and remove configuration elements. Configuration elements hold properties of the elements and their corresponding values.



The main elements discussed in this appendix are Input Manager, Devices, Device Proxies, Displays, Cluster, and User elements. These elements are enough to get you started creating a configuration file and are the basic elements for a VR application.

## Input Manager

The Input Manager configuration element has three properties: a driver search path, a driver DLL name, and a directory to scan for drivers to load. All three properties may have zero or more values, but at least one must have a value in order for the configuration element to be useful.



- The driver search path provides the Input Manager with a list of directories where driver DLLs may be found.
- The driver DLL name property is used to name specific device drivers to load.
- The driver scan path provides a mechanism for loading all the device drivers found in the named directory or directories.

The Drivers are precompiled drivers provided by gadgeteer for the inclusion of old devices. These Drivers can be found in:

VizLab\_Software-> VRJuggler -> lib -> gadgeteer -> drivers (Windows)

/usr -> lib -> gadgeteer -> drivers (Linux)

It is important to note that a non-standard device (mouse, keyboard, etc) needs a .dll/.so or a VRPN server in order to work with VR applications.

## Devices

Gadgeteer supports two categories of devices physical devices and simulator devices. The term physical device is arguable because simulator devices get their input from a keyboard and mouse, and a keyboard and mouse are certainly physical input devices. The distinction is that simulator devices mimic the behavior of physical devices such as trackers using input from a keyboard and/or a mouse. The configuration of simulator devices differs from that of physical devices when it comes to VRJConfig.

A key aspect that is common to all input devices handled by the Input Manager is that they define input sources of one or more input categories. The input categories currently supported by Gadgeteer are the following:

- Analog: Data in a continuous range with well-defined minimum and maximum values and receive the data as normalized values in the range 0.0 to 1.0.
- Command: Discrete command input, often in the form of recognized spoken commands or pre-defined gestures.

- Digital: on/off input, usually corresponding to simple button presses and releases.
- Digital glove: Distinct combinations of fingers.
- Gesture glove: Recognizable hand gestures based on knuckle angles.
- Keyboard/mouse input handler: Source of keyboard and mouse events from the native window system.
- Positional: Multi-degree tracker data that can track up to six axis.
- String: Arbitrary sequences of characters, usually corresponding to spoken words or phrases.

A device, physical or simulator, will fall into one or more of those categories.

Physical devices are those that provide what is traditionally considered immersive input. In terms of the Gadgeteer Input Manager, physical devices require a device driver plug-in. The driver implements in software the communication protocol with the hardware so that data can be read from the device and interpreted before being passed on to the VR application. The drivers handle configuration elements for device and if the Input Manager does not load a required device driver, the configuration element cannot be processed. It is very important to remember to configure the Input Manager as part of configuring a physical device.

The configuration of simulator devices is much more involved than that required for physical devices. All simulator devices read data from a traditional desktop keyboard and mouse and translate the data into information that mimics the behavior of the various device types supported by the Input Manager. To configure a simulator device, there are three pieces that must be configured: a keyboard/mouse input handler, an input window, and the actual simulator device.

The following are the possible Simulated Device Types and their properties:

#### Simulated Analog Devices

- keyboard/mouse proxy- a pointer to another configuration element of type keyboard\_mouse\_proxy.
- increment key- property defines a list of key pairs that are used to identify the key presses for incrementing the value of the analog input sources.
- decrement key- property is the complement to the “increment key” property.
- Delta- The “delta” property defines the change in the analog value per key press.
- range minimum - property sets the minimum possible value for the device as a floating-point value. The default is 0.0.
- range maximum - property sets the maximum possible value for the device as a floating-point value. The default is 255.0.
- initial value - property sets the starting value for the device as a floating-point value. This must be in the range [min, max].

### Simulated Digital Devices

- keyboard/mouse proxy - property specifies a pointer to another configuration element of type keyboard\_mouse\_proxy.
- digital button key - property defines a list of key pairs that provide digital input sources.

### Simulated Digital Glove Devices

- keyboard/mouse proxy - property specifies a pointer to another configuration element of type keyboard\_mouse\_proxy.
- left glove position - property sets a pointer to a position proxy that provides tracker information for the left hand.
- right glove position - property sets a pointer to a position proxy that provides tracker information for the right hand.

### Simulated Positional Devices

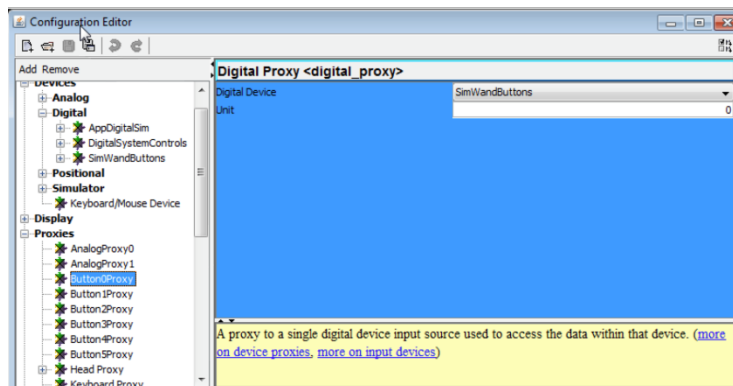
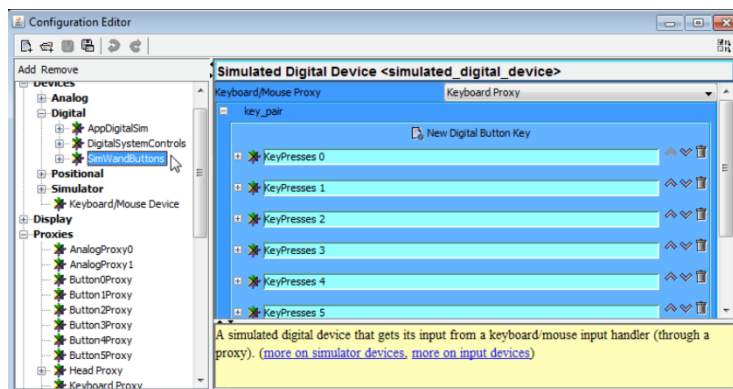
- keyboard/mouse proxy - property specifies a pointer to another configuration element of type keyboard\_mouse\_proxy.
- key pairs - property has twelve values that define the key pairs that are translated into changes in the 4×4 transformation matrix.
- initial position- property has three values that specify the initial position in three-dimensional space of this positional device.
- initial rotation- property has three values that define the rotation about the X, Y, and Z axes for this device.
- translation delta - property sets the change in translation per key press received from the keyboard/mouse input handler.
- rotation delta - property sets the change in rotation per key press.
- translation coordinate system - property has two possible values: Local or Global. The value chosen indicates whether translations occur in the local coordinate frame or in the global coordinate frame.
- rotation coordinate system - property has two possible values: Local or Global. The value chosen indicates whether rotations occur in the local coordinate frame or in the global coordinate frame.
- position filters - property defines zero or more position filters to apply to data received from this device.



## Device Proxies

Device proxies introduce the layer of abstraction and indirection needed to ensure that VR applications do not become tightly coupled with specific devices. The proxy acts as a pointer to a single data source from the actual device. Device proxies are tied in to the input devices described earlier. This makes them have corresponding proxy type for each device.

| Device         |    | Corresponding Proxy  |
|----------------|----|----------------------|
| Analog         | -> | Analog Proxy         |
| Command        | -> | Command Proxy        |
| Digital Glove  | -> | Digital Glove Proxy  |
| Gesture Glove  | -> | Gesture Glove Proxy  |
| Keyboard/Mouse | -> | Keyboard/Mouse Proxy |
| Positional     | -> | Positional Proxy     |
| String         | -> | String Proxy         |



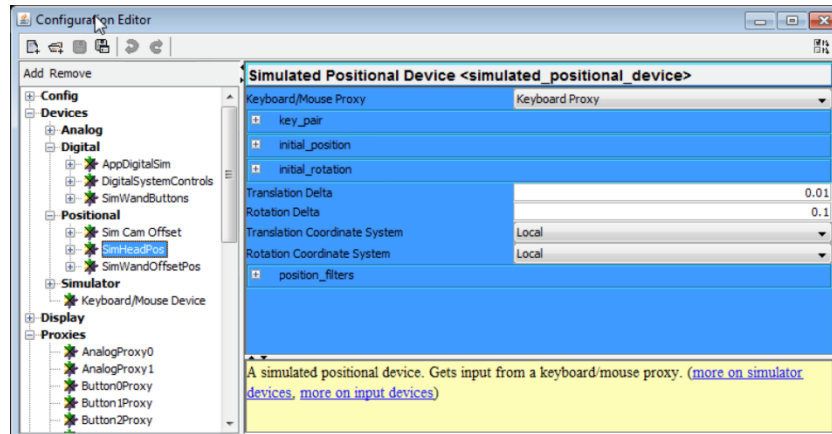
It can be seen from the top figure that Button0Proxy is selected. Within the property fields there is a Digital Device property and it is pointing to SimWandButtons, which is a device within our Devices. The Unit property specifies the value of the button when this button is pressed. This shows the main idea of configuring a Device Proxy to a particular Device Element.

## Proxy Aliases

Proxy aliases provide more abstract identifiers for use at the level of VR applications. In a VR application, there will invariably be at least one device interface object, and that object must be initialized with a symbolic identifier used by the Input Manager to connect the device

interface with a proxy. While the specific proxy name can be used, aliases offer some additional flexibility because a single proxy may have many aliases.

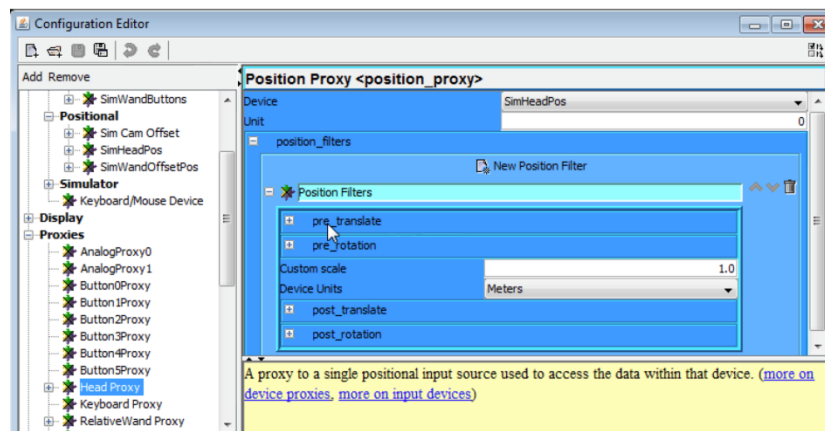
For example, we had already created a proxy for a button in SimWandButtons device called Button0Proxy. I have created a VJButton0 proxy alias. A proxy alias only has one property field called Proxy Pointer. This property will point to a specific Proxy, in this particular case Button0Proxy. It is important to not that when working with application code, you will only have to reference the proxy alias to retrieve the value of a specific device element.



### Position Filters

When dealing with data from a tracker, the positional references of the tracker or its sensors may not be what you might expect. Most of the time, tracking data needs to be transformed into common coordinates.

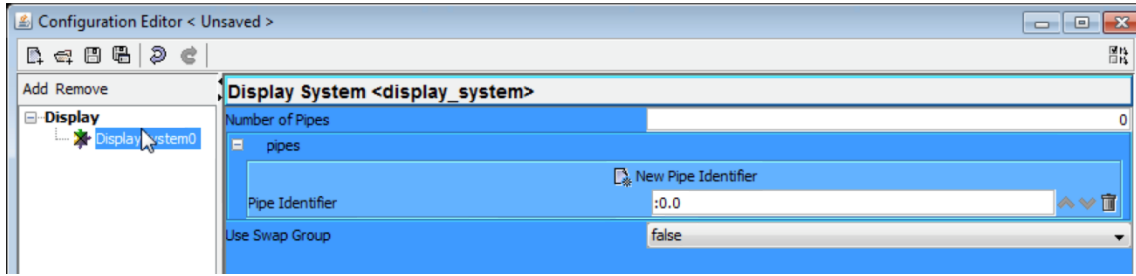
We can use the figure up above to describe this process. This application has a Positional Device that gets the head position. SimHeadPos is the device that retrieves head positional data. The properties can be set to the desired positions.



Next, a head proxy is created called HeadProxy. HeadProxy points to SimHeadPos. Within the HeadProxy we can add a position filter that will convert device coordinate data to our defined coordinate system. It is important to note that it is better to make positional changes within a Proxy than in the Device. This is because a same device can be used with multiple VR applications at the same time.



## Displays



### Display System

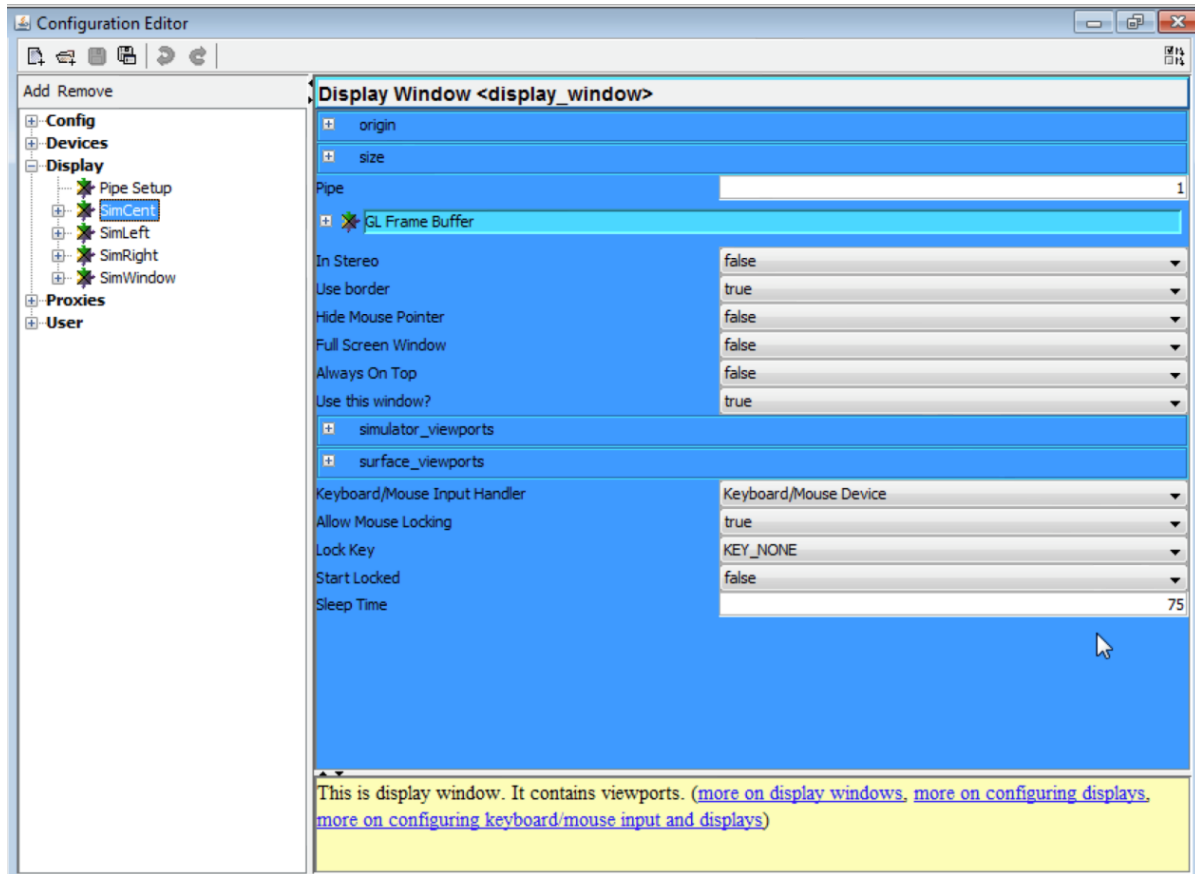
When configuring displays, the Display Manager is the first piece that must be configured. The Display Manager is configured by the Display System element. Except in the case of a cluster configuration, there must always be one and only one “Display System” configuration element in a configuration. The Display System configuration element has only two properties: “Number of Pipes” and “X11 displays (Pipe Identifier).” These are explained below.

- *Number of Pipes* – property tells the Display Manager how many *physical* graphics pipes the host machine has.
- *Pipe/Pipe Identifier* – property that defines a list of available pipe identifiers available for opening windows of any type.
- *Swap Group* - property that defines if the display needs to be sync with other displays. (Note: This is if your running more than one display with one computer or a System cluster.)

There must always be one Display System configuration element in a non-cluster configuration with an accurate value set for the number of available graphics pipes. The Display Manager configuration impacts the number of rendering threads created. For a multi-pipe configuration, there will be one rendering thread created per pipe. If more than one display window is opened on a single pipe, all the windows in that pipe will render in the same thread.

### Display Windows

Display windows are used for rendering graphics. Display windows are capable of displaying OpenGL graphics, and make use of the platform-specific (depending if Windows or Linux) API for opening and managing OpenGL-aware windows. The details are hidden behind the abstract concept of a display window, therefore making the configuration process the same through all platforms. The display window is made out of several-nested configuration elements, making it the most configurable element when configuring a system.

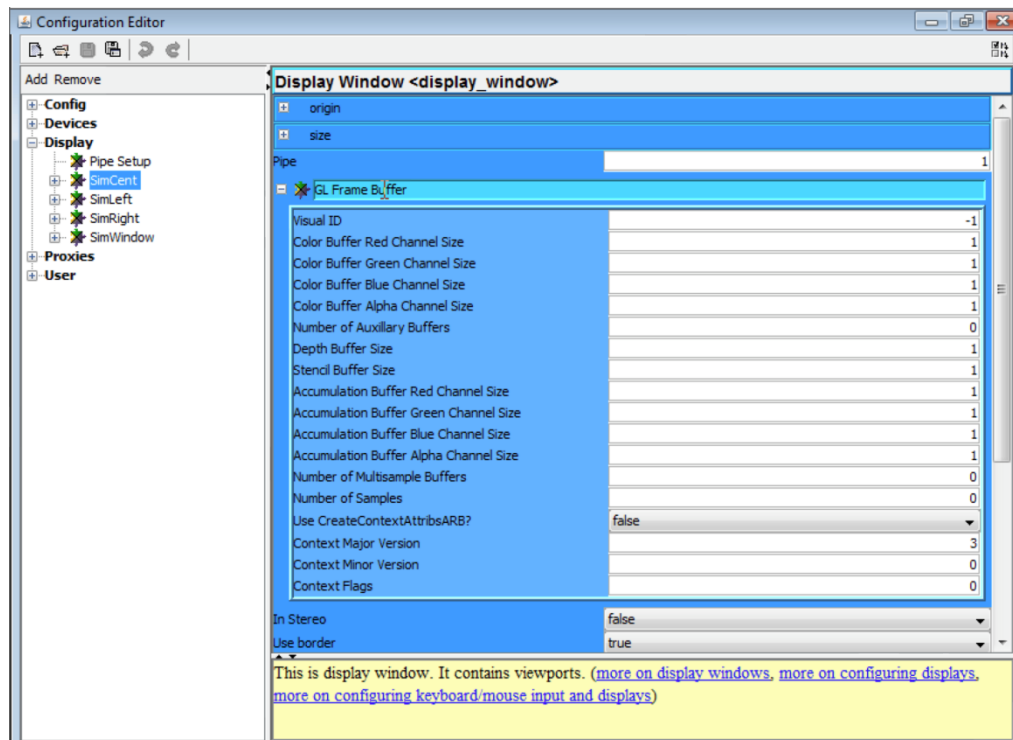


- *Origin* - property defines the (X,Y) coordinate for the window origin(Top Left).
- *Size* - property defines the width and the height of the window.
- *Pipe* - property is an index into the zero-based list of graphics pipes (rendering threads) defined in the Display System configuration element.
- *Use border* – property that will display window border if set to true.
- *Hide Mouse Pointer* – property that will hide the mouse displaying on window.
- *Stereo* - property indicates where the display window should be opened with active stereo rendering capabilities enabled.
- *Full Screen Window* - property indicates whether the window should be opened full screen. If it is set to true, the origin and size properties are ignored and the window is opened to the maximum resolution of the display.
- *Always on Top* - property indicates whether the window is to be positioned above all other windows on the desktop. If it is set to true, no other windows will be able to be opened above the display window. (Note: There is an issue when using NVidia framelock it will cause a deadlock or block for long periods of time. It's recommended that this property be set to false when used with NVidia framelock.
- *Use this Window*- property defines whether the display window should be used or not.
- *Keyboard/Mouse Input Handler* - property points to another configuration element of type `keyboard_mouse_device`.
- *Allow Mouse Locking* - property that will mouse pointer into the center of the window. (Note: This is done by pressing a key that is configured by “Lock Key”.)
- *Lock Key* - property defines a key that, when pressed, causes the mouse pointer to be locked to the center of the window.

- *Start Locked* - property indicates where the mouse pointer should be locked to the center of the window as soon as the window opens.
- *Sleep Time* - property sets a sleep time (milliseconds) for the input window thread. (Note: To prevent the display window from starving other event threads.)

### OpenGL Frame Buffer

Each display window has a unique OpenGL frame buffer configuration. The nested configuration element has a direct correspondence to the way that the window API works, so familiarity with WGL, GLX, or AGL is helpful when understanding what values to use for the frame buffer settings. Users must understand what is meant by *color depth* in order to configure the frame buffer correctly, those who do not should leave all values to the default settings.



- *Visual ID* - property, a specific GLX or WGL visual ID can be chosen for the display. (Note: glxinfo for Linux and wglinfo for Windows)
- *Red Channel Size* - property specifies the minimum number of bits per pixel to use for the red channel.
- *Green Channel Size* - property specifies the minimum number of bits per pixel to use for the green channel.
- *Blue Channel Size* - property specifies the minimum number of bits per pixel to use for the blue channel.
- *Alpha Channel Size* - property specifies the minimum number of bits per pixel to use for the alpha channel.
- *Number of Auxiliary Buffers* - property specifies the minimum number of bits to use for the Auxiliary buffer. (Note: It's typically used for accumulating a series of images into a final, composite image.)

- *Depth Buffer Size* - property specifies the minimum number of bits to use for the depth buffer.
- *Stencil Buffer Size* - property specifies the minimum number of bits to use for the stencil buffer. (Note: stencil buffer is to restrict drawing to certain portions of the screen.)
- *Acc. Buffer Red Channel Size* - property specifies the minimum number of bits per pixel to use for the Acc. red buffer channel.
- *Acc. Buffer Green Channel Size* - property specifies the minimum number of bits per pixel to use for the Acc. green buffer channel.
- *Acc. Buffer Blue Channel Size* - property specifies the minimum number of bits per pixel to use for the Acc. blue buffer channel.
- *Acc. Buffer Alpha Channel Size* - property specifies the minimum number of bits per pixel to use for the Acc. alpha buffer channel.
- *Number of Multisample Buffers* - property that enables the number of multisample buffers can be configured and used.
- *Number of Samples* - property that enables the number samples per buffer can be used.
- *Use create context attribs* - property specifies whether to use OpenGL 3.0 functions such as `glCreateContextAttribsARB()` or `glXCreateContextAttribsARB()` when creating the OpenGL context. (Note: Setting this property to true, the user must have OpenGL 3.0 or newer)
- *Context Major Version* - property can be used to set the major version for the OpenGL context.
- *Context Minor Version* - property can be used to set the minor version for the OpenGL context.
- *Context Flags* - property can be used to flags for the creation of the OpenGL context.

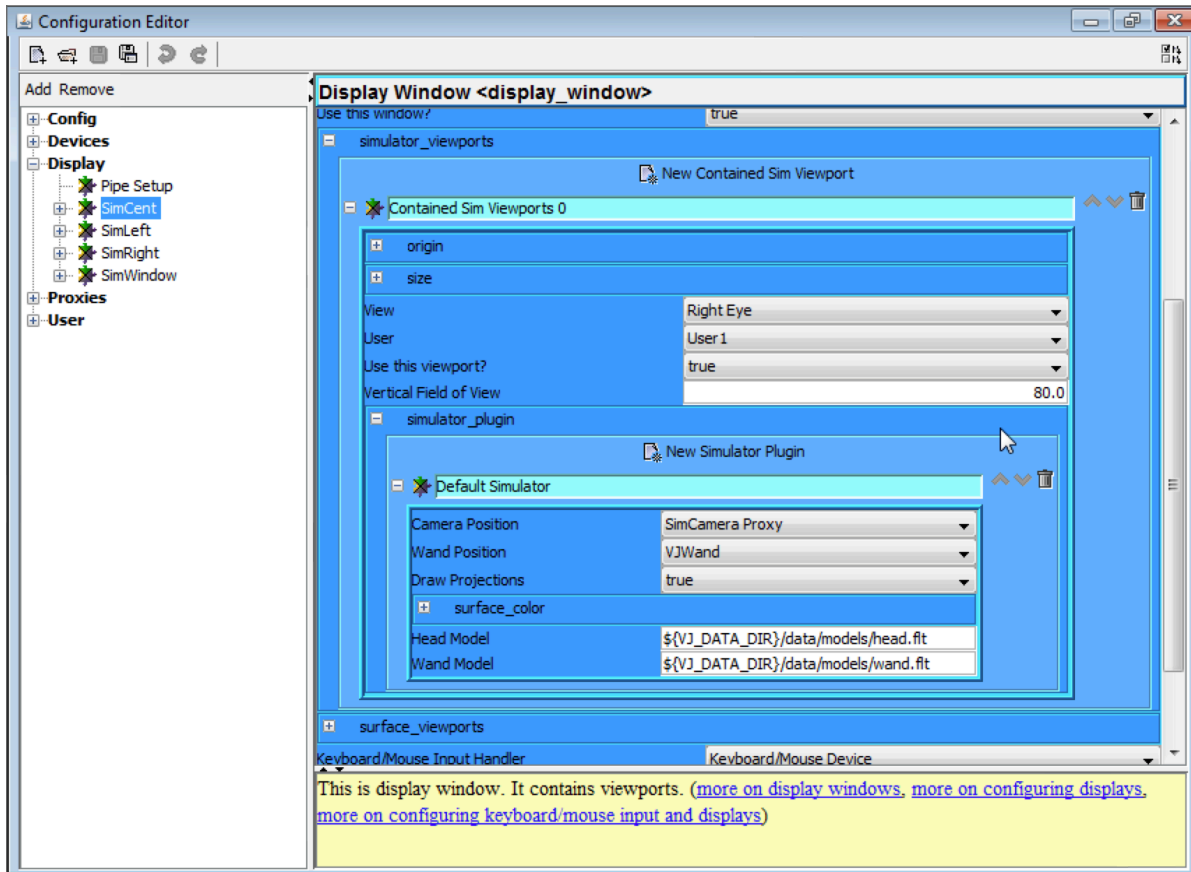
## Viewports

Viewport can be perceived as a window into the graphical display or scene. There are two types of viewports when configuring a system; a Simulator Viewport and Surface Viewport. Both type of viewports have common configuration properties, these properties are as followed:

- *Origin* - property defines the (X,Y) coordinate for the viewport origin(Top Left). (Note: This is within the window frame)
- *Size* - property defines the width and the height of the viewport. (Note: this is from value 0 - 1. It is the percentage of how much the viewport will take out of the window space.)
- *View* - property defines which eye will be rendered by the Draw Manager's rendering thread for the display window. Possible elements are Left Eye, Right Eye, and Stereo (if stereo option is chosen). (Note: The default setting is to render on the left eye.)
- *User* - property points to a configuration element defining a single User.

### *Simulator Viewports*

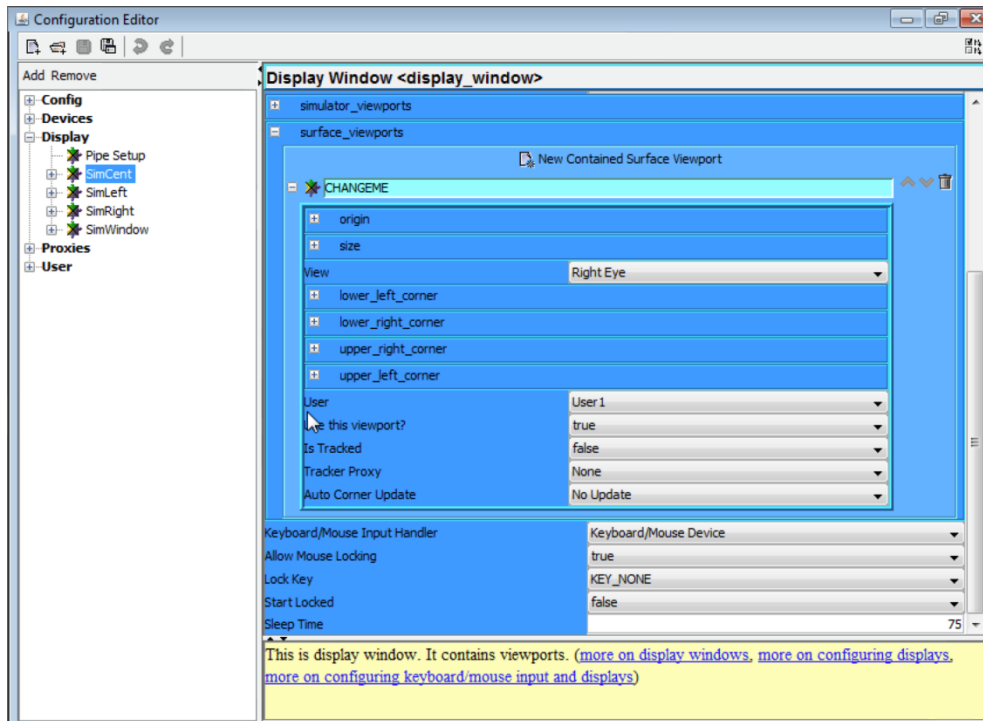
Simulator viewports are used for rendering a simulator interface. The default simulator interface has three components: a head, a wand, and a detached, movable camera.



- *Vertical Field of View* - property accepts real-numbered (float-point) values representing the degrees of the viewport's field of view. (Note: Default is 80.0 degrees)
- *Simulator Plugin* - property that configures OpenGL Drawing Manager and consist of multiple elements.
  - ⇒ *Camera Position* - property points at a position device proxy config element that can move the detached camera around in the scene.
  - ⇒ *Wand Position* - property points at a position device proxy config element that can move the wand around in the scene.
  - ⇒ *Draw Projections* - property that indicates whether projections for surface viewports should be rendered within the simulator interface.
  - ⇒ *Surface Color* - property has three real-numbered values that provide the red, green, and blue settings for the color of the rendered projection surfaces.
  - ⇒ *Head Model* - property that names the model to be loaded by the simulator plug-in. (Note: The default model is in (VRJuggler main folder then /share/vrjuggler/data/models/head.flt)
  - ⇒ *Wand Model* - property that names the model to be loaded by the simulator plugin. (Note: The default model is in (VRJuggler main folder then /share/vrjuggler/data/models/wand.flt)

## Surface Viewports

These are used for projection surfaces such as the walls for a CAVE, the surface of an immersive desk, or the eyes of an HMD. In these viewports, the user's eyes and the camera are permanently attached. The rendered view is for the perspective of the tracked user. The manner by which the view is calculated depends on one key factor; whether the surface is at a fixed point or if it can move. Those surfaces that are in fixed positions are typically CAVE walls or immersive desk and those that can move have a tracker associated with them in some way.



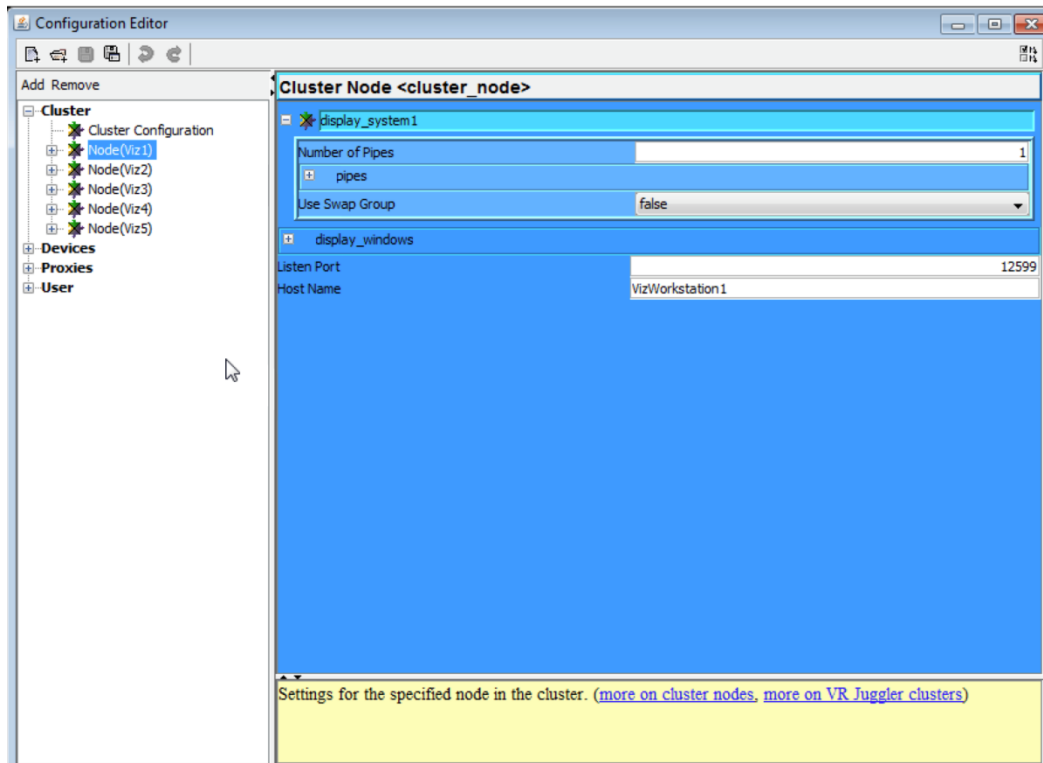
- *Corners* - property that positions the four corners of the projection surface. The surface must be rectangular in shape, but it may be at any orientation. The settings for the corners represent the actual position of the physical surface's corners, and the units for the corners are always entered in meters. The origin point for the corners takes on a different meaning depending on whether the surface is fixed or movable, however. For a fixed-position projection surface, the origin will be the origin set by the tracker configuration. If there is no tracker, then choose an origin and set the corners relative to it.
- *Is tracked* - property indicates whether the surface is fixed in place (false) or movable (true).
- *tracker proxy* - property to set the position device proxy for the tracker sensor that will provide the viewing information. (Note: "Is Tracked" should be true.)

## Cluster Configuration

Cluster configurations have two required pieces; one or more cluster nodes and a single Cluster Manager.

## Cluster Nodes

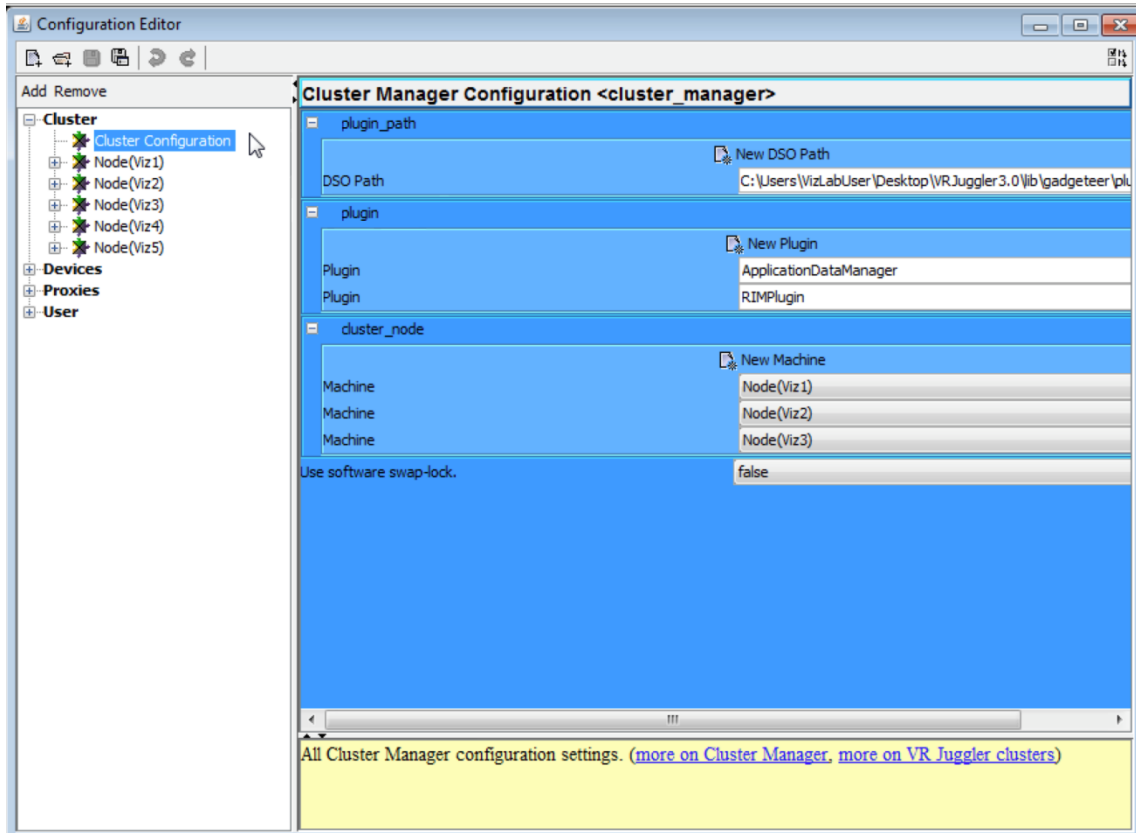
A cluster configuration makes use of a cluster node configuration element. A cluster node captures aspects of the hardware and software that are unique to each node (A computer in a cluster configuration) in the cluster, and only the node identified in the configuration element will handle the information intended for it. All the cluster nodes in the cluster make up the cluster network that is handled by the Cluster Manager.



- *Display system* - property sets up the graphics pipe information, just as was presented in *Display System*. There must be only one display system configuration per node, even if display windows will not be opened.
- *Display windows* - property that lists 0 or more display windows to be opened on the local node. This is where each rendering node in the cluster defines its window(s). The windows themselves are configured exactly as described in *Display Windows*.
- *Listen port* - property that identifies the port on which the cluster node will listen for incoming connections from the other nodes in the cluster. (Note: On most operating systems, port numbers below 1024 are not allowed for user-level applications, it is best to pick a port in the range 10000–65536).
- *Host Name* - property identifies the node of the cluster. The value must be either a valid host name of the node.

## Cluster Manager

Configuring the Cluster Manager is very straightforward. First, any cluster plug-ins that will be needed for the cluster configuration should be identified. Then, the list of cluster nodes must be built up using pointers to the cluster node configuration elements.



- *Plug-in path* - property that provides path information for loading cluster plugins. (Note: The path for Windows is VizLab\_Software-> VRJuggler -> lib ->gadgeteer -> plugins or Linux is /usr -> /lib -> /gadgeteer -> /plugins folder.)
- *Plug-in* - property can have zero or more values with each value providing the abstract name of a cluster plug-in to load. (Note: depends on the drivers on the folder from plug-in path)
- *Cluster node* - property lists all the nodes of the cluster, regardless of whether they are rendering nodes, input nodes, or both.
- *Use Software swap lock* - property indicates whether a software-level swap lock based on TCP/IP should be used. If false, it is expected that some other software- or hardware-level swap lock solution is in place. Otherwise, there will be no swap locking, and visual “tearing” will likely be seen between screens.



## *Cluster Plug-Ins*

### Remote Input Manager(RIM):

Remote Input Manager is required for sharing data from input devices among all the cluster nodes. Without this plug-in, input device data is only available on the node where the device is physically connected. If the applications are expecting to get input from devices, then this plug-in must be loaded. This plug-in does not have a configuration element, so including it with the list of plug-ins for the Cluster Manager to load is sufficient to enable its use.

### Application Data Manager:

Application Data Manager is required when applications have custom data structures that will be shared among the cluster nodes. The Application Data Manager itself requires no configuration, but applications making use of the Application Data Manager features may require additional configuration information (GUID and hostname).

globally unique identifier (GUID) - Every data type that will be shared by the clustering subsystem must be registered at run time. Globally unique identifiers are 128-bit values that are guaranteed to be statistically unique. They provide a means for giving every user-defined data structure a unique identifier.

responsible host(hostname) - The “data-local” cluster node is identified using this property. This is a string value type that will be used to match against the host name of each cluster node. When the value of this property matches the host name of a cluster node, that cluster node will be the data-local node for the shared data structures.

## Users

In the configuration, users are configurable and this includes the separation between the user's eyes, which is also called the interocular distance. A configuration element for a user names the position device proxy for the user's head as well. At least one user must be defined per configuration because the viewports contained within a display window must refer to the user.

For more information in configuring the system please look at VRJuggler's Documentation at [vrjuggler.org/documentation.php](http://vrjuggler.org/documentation.php).

## APPENDIX D

## APPENDIX D

### VIZLAB APPLICATIONS

This Appendix describes VizLab's Applications, which are extensions of VRJuggler's sample applications. There are three applications, which are going to be covered:

MPapp – OpenGL application

OsgNav – OpenSceneGraph and OpenSG applications

OsgNav\_Bullet – Scene Graph applications with Bullet Physics Engine

These applications have something in common, which is the main.cpp file. This file consists of calling VRJuggler kernel, it follows more or less the GLUT programming paradigm. The main program just runs a VRJuggler kernel instance; the kernel takes care of input handling, rendering, clustering, networking, etc.

#### **MPapp**

MPapp is an OpenGL application made to run with VRJuggler Kernel Engine. It consists of the following files:

- Main.cpp – The main program which runs in VRJuggler Kernel.
- Mesh.h – Header file used for a mesh implementation.
- MPApp.h – Header file that declares OpenGL functions and integrates them with VRJuggler.
- MPApp.cpp – Implementation file for MPApp.h functions.

Within the MPApp.h file there are the following functions:

- Init()
- apiInit()
- bufferPreDraw()
- preFrame()
- intraFrame()
- postFrame()
- contextInit()
- draw()

The Init() function serves as the initialization function for Devices and Interfaces, this is where you would link the devices to the proxy aliases that were declared in the configuration process and also serves as initialization for non-OpenGL functionality.

The `apiInit()` function serves as initialization function for any API that you may have added to the application; you can also use it to initialize any non-OpenGL functionality. This function can be used to initialize any Tweak interface that you may have developed for an OpenGL application, as it is called before the drawing manager.

The `bufferPreDraw()` function is designed for updating any functionality when working with stereoscopic stereo; it is called before each buffer (left and right).

The `preFrame()` function is designed for updating any functionality within your OpenGL application before the frame is called to the drawing manager. This is where you would want to update any camera, user positions, matrix transformations, devices, etc.

The `intraFrame()` function is designed for updating any functionality while the frame is in the drawing manager and it is rendering. This can be used for updating information or can trigger any Tweak applications. It is mostly used if your application runs multiple frames.

The `postFrame()` function is designed for updating any functionality when the frame has finished rendering, but before any kernel updates are applied (Device and Cluster functionality).

The `initGLState()` function is designed for initializing OpenGL state information, this includes lighting, shading, ambient lighting, etc.

The `draw()` function is designed for the drawing manager. This is the function used to draw the scene within the drawing manager. It can be called once or more times per frame according to your specifications.

Implementation occurs within the `MPApp.cpp`. This is where you would implement the functionality of the functions declared on the `MPApp.h` file. A GLUT imported to `MPApp` example application can be reached in: <http://dchavez.net/?p=703>

## **OsgNav**

OsgNav is a Scene Graph based application made to run with VRJuggler Kernel Engine. The project consists of six files.

- `Main.cpp` – The main program which runs in VRJuggler's Kernel.
- `Nav.h` – Header file that provides Navigation properties.
- `OsgNav.h` – Header that declares `OsgNav.h` functions.
- `OsgNav.cpp` – Implementation of the `OsgNav.h` Navigation functions.
- `RemoteNavSubject.h` – Header if running a Cluster Configuration. It declares functions needed for sharing navigation information across a cluster using RIM plug-in.
- `RemoteNavSubject.cpp` – Implementation of Cluster functions declared in `RemoteNavSubject.h`.

It is important to note that the make file will compile `RemoteNavSubject.h` and `RemoteNavSubject.cpp`, even if you're not running a cluster configuration.

Navigation in OsgNav is handled by Nav.h and there are 4 main function that you would need to get familiar in working with Nav.h and are as follows:

- setWalkMode – sets if the user will be bound to the bottom of the scene or if he can fly around the scene.
- setVelocity – sets how fast the user will travel within in the scene.
- seRotationVelocity – sets how fast the user will rotate within the scene (angular velocity).
- setCurPosition- will set the starting position of the user when the scene initializes.

The application itself is within OsgNav.h and OsgNav.cpp. As MPApp, it uses GLUT paradigm for programming the application. The following are the functions within the application:

- initScene – This function initializes the scene; where lighting, ambient lighting, and other properties are initiated.
- myInit - This function is for any API other than OSG initialization.
- initTweak - is for any java interfaces that uses Tweak
- getScene – initiates the root of the Scene Graph.
- bufferPreDraw – This function provides updates for each eye if your rendering in 3D.
- latePreFrame – This function is used to update any information before the scene gets drawn.
- intraFrame – This function is used to do anything while the scene is drawn but is still in the pipeline.
- postFrame – This function is used to update any information when the scene is drawn.
- setModelFilename – gets a model from the string path.

More information about OsgNav application can be reached in: <http://dchavez.net/?p=739>

### **OsgNav\_Bullet**

OsgNav\_Bullet is a Scene Graph based application with Bullet physics engine made to run with VRJuggler Kernel Engine. The project is derived from OsgNav and uses the same files but with added functionality for Bullet integration. It uses OsgWorks and OsgBullet libraries to join Scene Graphs with Bullet.

- Main.cpp – The main program, which runs in VRJuggler’s Kernel.
- Nav.h – Header file that provides Navigation properties and declaration of Bullet physics functions.
- OsgNav.h – Header that declares Navigation functions and where bullet instances are joined with OsgWorks and OsgBullet.
- OsgNav.cpp – Implementation of the OsgNav.h Navigation functions and Bullet interactions.

- RemoteNavSubject.h – Header if running a Cluster Configuration. It declares functions needed for sharing navigation information across a cluster using RIM plug-in.
- RemoteNavSubject.cpp – Implementation of Cluster functions declared in RemoteNavSubject.h.
- Ragdoll.h – Declare the functionality for a ragdoll avatar
- Ragdoll.cpp – Implementation for the Ragdoll avatar

OsgNav and OsgNav\_Bullet have the same foundations, but differ in that OsgNav\_Bullet includes Bullet physics engine. In order for them to work together OsgWorks and OsgBullet are needed. OsgWorks and OsgBullet provide the communication layer between OpenSceneGraph and Bullet.

- initScene – This function initializes the scene; where lighting, ambient lighting, and other properties are initiated.
- initPhysics – Initializes Bullet, OsgWorks, OsgBullet libraries to use in the application
- myInit - This function is for any API other than OSG initialization.
- initTweak - is for any java interfaces that uses Tweak
- getScene – initiates the root of the Scene Graph.
- bufferPreDraw – This function provides updates for each eye if your rendering in 3D.
- latePreFrame – This function is used to update any information before the scene gets drawn.
- intraFrame – This function is used to do anything while the scene is drawn but is still in the pipeline.
- postFrame – This function is used to update any information when the scene is drawn.
- setModelFilename – gets a model from the string path.

More information about OsgNav application can be reached in: <http://dchavez.net/?p=749>

## BIOGRAPHICAL SKETCH

Moises D. Carrillo, raised in Mission, Tx, graduated in Spring 2011 from University of Texas – Pan American with a degree in Computer Science. He entered the Masters program at University of Texas – Pan American in Fall 2011. During his time as a graduate student, he worked as a Teacher Assistant and Research Assistant for the Department of Computer Science. After he graduates in May 2013, Moises will move to Austin, Tx, where he will work for IBM as a Software Engineer. You may contact Moises D. Carrillo at [carrillo\\_moises@hotmail.com](mailto:carrillo_moises@hotmail.com)