Porthole: A Decoupled HTML5 Interface Generator For Virtual Environments

BY

DANIELE DONGHI
Laurea, Politecnico di Milano, Milan, Italy, 2010
Laurea Specialistica, Politecnico di Milano, Milan, Italy, 2012

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:

Jason Leigh, Chair and Advisor
Andrew Johnson
Pier Luca Lanzi, Politecnico di Milano

to my parents . . .

ACKNOWLEDGMENTS

I would like to thank Alessandro Febretti of the Electronic Visualization Laboratory for his support and guidance: without him, this research would not have been possible. I would also like to thank Franca Garzotto for her feedback that helped me go in the right direction.

Besides, I would like to thank my friends and classmates that have shared this experience with me: Giovanni, Andrea, Matteo, Mattia and Stefano. Thanks to Bianca for helping me testing this work with her iPad.

Lastly, and most importantly, I would like to thank my family and, in particular, my mom, who has always supported me these years.

DD

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

LIST OF FIGURES (Continued)

LIST OF ABBREVIATIONS

CAVE CAVE Automatic Virtual Environment. 7, 12, 13, 21

CAVE2™ CAVE Automatic Virtual Environment 2. vii, xii, 2, 6, 7, 9–12, 29, 86, 87, 90, 104–106

CAVERN CAVE Research Network. 12

CSS Cascading Style Sheet. xii, 23, 47, 49, 104

CVE Collaborative Virtual Environment. xii, 2, 3, 5, 6, 9–12, 21, 105

DOM Document Object Model. 31, 34, 41–43, 52, 68

EVL Electronic Visualization Laboratory. 7, 9, 90

FPS Frames Per Second. 82, 83, 100, 102

GPU Graphics Processing Unit. 30, 48

GUI Graphical User Interface. xii, 3–5, 14–16, 21, 24, 25, 27, 44, 50, 52, 54, 55, 65–68, 70, 73, 74, 83, 104, 105

HCI Human-Computer Interaction. xii, 1, 3, 6, 17, 24, 105, 106

HTML HyperText Markup Language. 18, 19, 25, 31, 33, 45, 52, 67, 70, 76, 78, 80, 81, 104, 105

HTML5 HyperText Markup Language version 5. 4, 22, 23, 25, 29, 31–33, 36, 37, 40, 47, 55, 61, 86, 92, 104

HTTP Hypertext Transfer Protocol. 25, 29, 37–39, 45, 55–59, 68, 80

IQR InterQuartile Range. 84, 85, 90, 93–95, 97–99, 101

JPEG Joint Photographic Experts Group. 60, 61, 83, 86, 92, 95, 96, 98

LIST OF ABBREVIATIONS (Continued)

JSON JavaScript Object Notation. 40, 60–63, 65, 70, 73

OpenGL Open Graphics Library. 30, 86, 92

OS Operating System. xii, 4, 17, 21, 40, 105

OSG OpenSceneGraph. 30, 86, 92

PDA Personal Digital Assistant. 19

RTT Round Trip Time. 83, 86, 87, 89–91, 104

SD Standard Deviation. 84, 90, 93–95, 97–99, 101

SP Smartphone. 85, 86, 93–95, 97–99, 101

TL Tablet. 85, 86, 93–95, 97–99, 101

UIC University of Illinois at Chicago. 7, 9, 86, 89

UML Unified Modeling Language. 27

URI Uniform Resource Identifier. 58, 61

URL Uniform Resource Locator. 69, 80

VE Virtual Environment. vii, xii, 1–25, 27, 29–31, 33, 34, 39, 40, 44, 45, 50, 53, 65, 68, 75, 76, 81, 83, 104–106

VR Virtual Reality. 2, 9, 12, 20, 21, 24, 53

VTK Visualization Toolkit. 30

XID XML Interfaces Description. xii, 5, 45, 47, 49, 50, 52–54, 65, 66, 75, 76, 78, 86, 104, 111

XML Extensible Markup Language. 23, 45, 50, 52, 53, 55, 77

SUMMARY

This thesis presents Porthole, a framework that helps applications developers to create user interfaces to Virtual Environment (VE). The goal is to enable interaction with VE systems using smartphones, tablets, laptops or desktop computers.

With Porthole, messaging and streaming management is transparent to VE applications developers: any device that has a HTML5 enabled browser can connect to a virtual environment, receive a Graphical User Interface (GUI) tailored on its specifications, and manage application cameras and parameters.

The developers can enable Porthole service just by providing a XML Interfaces Description (XID) (and, optionally, a Cascading Style Sheet (CSS) file for changing client interface style).

The end users of virtual environments, such as CAVE2™, can, then, use handheld devices to interact with VE applications. Moreover, end users can employ the same devices for both local and remote interaction, without the need of ad-hoc client applications.

Porthole addresses earlier work limitations by proposing a novel Human-Computer Interaction (HCI) model that exploits browsers as a mean of interaction with VE systems. State of art works, instead, present application or Operating System (OS) specific solutions. Finally, Porthole suggests a new concept of low-cost remote collaboration, with respect to Collaborative Virtual Environments (CVEs) model proposed in earlier work in this research area.

CHAPTER 1

INTRODUCTION

A Virtual Environment (VE) is a human-computer interface, in which the computer creates a sensory-immersing environment that interactively responds to and is controlled by the behavior of the user.

Different terms have been applied to such a system. Some, like the oxymoronic "artificial reality" and "virtual reality", suggest much higher performance than current technology can generally provide. Others, like "cyberspace" are puzzling neologisms. The term "virtual environment" seems preferable because is linguistically conservative, relating to well-established terms like virtual image.

In fact, virtual environments can be defined as interactive systems that provide both two and three dimensional interaction techniques, such that user feels immerse inside an environment that surrounds him, providing much more informations than a classic, single display computer.

A VE potentially provides a new communication medium for Human-Computer Interaction (HCI). VEs have broad applications potential, particularly in the following fields (a detailed description of them can be found in *The Encyclopedia of Virtual Environments* [8]):

- education, for training that would otherwise be too expensive or too dangerous;
- architecture, simulating a "walkthrough" of a virtual space before the expensive construction on the physical structure begins;
- science, giving scientists new ways of data visualization (for example, mathematicians can see three-dimensional equations, biologists can build 3D models of genes, and so on);

- medicine, allowing medical professionals to view large amounts of information by navigating through 3D models (for example, radiation planning can be aided by adjusting virtual laser beams on a virtual body and seeing how well they will converge on a tumor);
- robotics, allowing operators of robotic systems to help control a telerobot when performing difficult tasks;
- military, for flight simulation and training, or for telepresence during military missions;
- art, for creating interactive and surrealistic art forms or as an instrument that takes the user on a guided tour of existing conventional art;
- design, for evaluating virtual designs and virtual prototyping, so that many ideas and possibilities can be tried in a short period of time, without physically building each prototype;
- entertainment, such as video games;
- sports and fitness, creating realistic simulations and enhancing the experience of indoor exercise.

CAVE Automatic Virtual Environment 2 (CAVE2$^{\text{TM}}$)[3] system is currently one of the best Virtual Reality (VR) immersive device available for rendering virtual environments. Unfortunately, such kind of hardware is extremely expensive and complex, thus limited in its spread[9]. In fact, due to the high cost of professional solutions and their complexity, the proliferation of such kind of environments is limited to institutes or organizations able to pay and manage such structures.

In all the fields described above, multiple users collaboration may enhance the usefulness of these expensive systems. This is the reason behind the introduction of Collaborative Virtual Environments (CVEs), where collaboration is achieved by sharing virtual worlds among participants, each one acting inside a VE, linked by a computer network.

1.1 Challenges

In a virtual environment you can use your eyes, ears, and hands much as you do in the real world: move your head to set your viewpoint, listen to sounds that have direction, reach out your hands to grab and manipulate virtual objects. VE technologies (such as wand controllers, gloves and body tracking) provide a better understanding of three-dimensional shapes and spaces through perceptual phenomena such as head-motion parallax, the kinetic depth effect, and stereopsis. Precise interaction, however, is difficult in a virtual world.

Virtual environments suffer from a lack of haptic feedback (which helps us to control our interaction in the real world) and current alphanumeric input techniques for the virtual world (which we use for precise interaction in the computer world, such as mouse and keyboard) are ineffective, because they are linked to a laptop or a desktop computer, that are difficult to move around in a virtual environment.

Thus, "simple" tasks like changing visualization colors or brightness inside a VE system are difficult to achieve. This challenge led to the introduction of two dimensional menus inside three dimensional world. But, using such a Graphical User Interface (GUI) in a VE system brings a set of problems, such as GUI elements pointing issues or occlusion of parts of the scene. Details on how earlier work tried to solve these problems are described in chapter 2.

Another challenge is represented by multi-user and collaborative HCI techniques in VE systems, that pose a set of problems related to network usage, costs, remote access, and so on. To-date collaborative and remote interactions models have been based on CVE systems (as analyzed in chapter 2), but challenges about creation of both low-cost and remote interactions with VE systems have not been tackled yet.

A further challenge is the creation of a universally accessible way of interacting with VE systems, not using an ad-hoc client application, nor other expensive VE systems, which, as

described before, are available only to a small set of institutes and organizations. For example, while exploring a brain model for blood flow analysis inside a VE, it is currently difficult to involve a remote medical professional, who may contribute in such analysis, if he does not have access to a VE system.

1.2 Porthole goals

This thesis presents Porthole, a framework that helps applications developers to create user interfaces to Virtual Environment (VE). The first goal of Porthole is to overcome limitations of traditional VE interaction models. Tracking devices, such as wand controllers or gloves, and pointing techniques have numerous degrees of freedom that render them unwieldy, failing to recognize gestures or accidentally triggering others. Thus, Porthole goal is to create a framework for ease the generation of 2D tactile interfaces on handheld devices. The goal is not to create an ad-hoc client application, but make VE applications developers able to create and customize their own interface, making clients connections and interactions events management transparent to them.

The second goal of Porthole is to make these 2D interfaces universally accessible: by generating HyperText Markup Language version 5 (HTML5) interfaces, devices can connect and interact with VE systems without the need of installing Operating System (OS) specific applications. Thus, multiple devices with HTML5 browsers can connect, both locally and remotely, and receive an interface where users can see virtual world camera image streams and change VE applications specific parameters.

The third goal is to generate tailored interfaces: each client, once connected, should receive an interface which is function of its width, height and orientation, but also of VE application requirements. Thus, Porthole make developers able to differentiate GUI elements and their disposition by defining a set of different interfaces. The reason is that a device with a big screen,

such as a tablet or a laptop, is able to display much more information than the ones with small screen, such as smartphones.

The last goal of Porthole is to create a new model for remote universal collaboration. It wants to propose a different way of remote participation in VE sessions, with respect to state of art CVEs models, that cannot overcome the problems described above. The goal is to be able to share VE images with remote professionals and researchers, and make them able to interact with VE system.

1.3 Structure of the thesis

This chapter has provided an introduction to virtual environment, current research challenges and Porthole goals. The rest of the thesis is organized as follows: chapter 2 provides details about earlier works in areas related to this thesis; then, chapter 3 describes Porthole requirements, the technologies involved in its implementation, and a high level overview of the system; chapter 4 describes Porthole implementation, analyzing the architecture, the XML Interfaces Description (XID) file structure, and, finally, Porthole Server, Client, GUI and Events objects; then, chapter 5 provides the design of experiments and an analysis of results obtained; finally, chapter 6 presents conclusions and future works.

CHAPTER 2

STATE OF THE ART

The state of the art in the research area related to Virtual Environments is very rich and shows that it is the focus of great interest among researchers. In fact, it has been shown that they can be an effective way to visualize, interact and manipulate scientific data [10]. Moreover, the research is now moving towards new kind of HCI models that may involve multiple users, both local and remote, in order to best exploit this expensive multi-displays, often cluster-based, systems that can help solve complex problems involving very large and complex datasets.

In addition, this work involves interactions between handheld devices and large displays systems, that in this case are represented by a Virtual Environment (VE), in particular the CAVE2™ at University of Illinois at Chicago, avoiding public displays research area. The HCI models proposed in the state of art works largely vary, mainly depending on the domain involved, and are often domain specific solutions.

This chapter, thus, explores the research areas involved and is divided as follows. In section 2.1, the VE related works are presented, and the CAVE2™ system is analyzed, because it is enhanced with this work and it is used for testing; in section 2.2 the attention is on the CVEs, that are compared to the solution proposed; finally, in section 2.3, the state of the art models of interactions between handheld devices and VEs are presented.

2.1 Virtual Environment (VE)

VEs offer a new HCI paradigm in which users are no longer simply external observers of images on a computer screen but are active participants within a three dimensional (3D) virtual world generated by a computer (often a cluster of computers, in order to be able to manipulate

very large and complex datasets). Proposed application domains in this field include design, visualization, education, and both training and clinical uses in medicine.

However, there are still very few immersive VE applications in common use outside the research laboratory, despite the rapid growing in the technology of displays, graphics processors, and tracking systems, and in the realism and speed of computer graphics. Such applications include architectural walkthrough [11] (and other passive visualizations), phobia therapy [12], and entertainment.

A general description and analysis of VEs can be found in [10]. In figure 1, the first CAVE Automatic Virtual Environment (CAVE) system developed at Electronic Visualization Laboratory (EVL) at University of Illinois at Chicago (UIC) is shown. This system has represented the first of a series of VE projects that now brought to the CAVE2™ [3].

Many VE interaction tasks fall into four categories: viewpoint motion control, selection, manipulation and system control. In particular, the viewpoint motion control refers to a task in which the user changes the position and the orientation of his point of view within the environment. Selection is a task which involves the picking of one or more virtual objects in order to make a modification of a particular object inside the environment. Manipulation, instead, refers to a task regarding the changing of the position and the orientation of objects. Selection and manipulation tasks are often paired together, although selection may be used for specific purposes, such as identify the object whose color is to be changed. The fourth interaction task, system control, includes other commands that the user gives to accomplish work within the application, such as changing the parameters of the visualization, adding users, and related cameras, inside the VE, saving the current location, loading a new model, etc.

These universal interaction tasks may have different interaction techniques, mostly depending on the applications domains. For example, one could accomplish a selection technique in a very

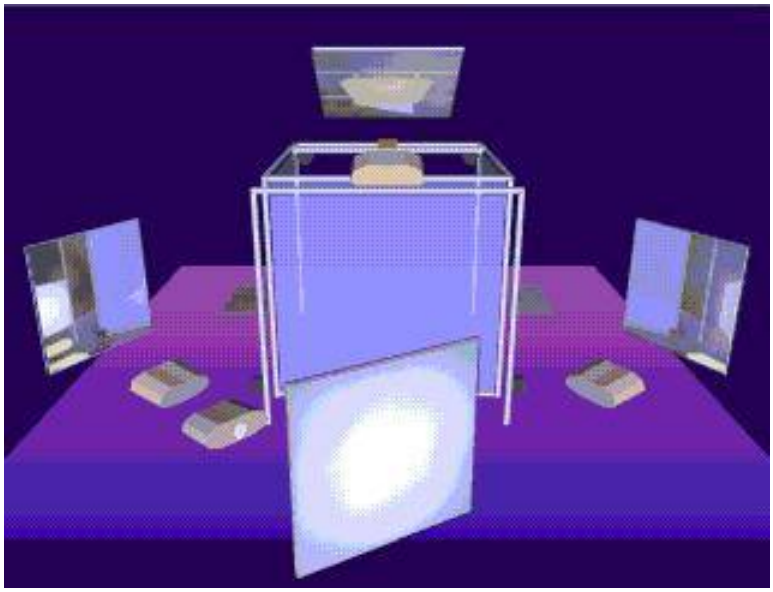

Figure 1: The CAVE: a surround-screen, surround-sound, projection-based Virtual Environment (VE) system [1]

indirect way, by choosing an entry from a list of selectable objects. Alternately, one could use a direct technique, where the user moves his tracked hand or uses some kind of gloves, so that he simulates the touching of the virtual object to be selected. Each of these interaction technique has advantages and disadvantages, and the choice of a certain technique may depend on many parameters, such as the available hardware, the experience of the user, or the precision required by the task. The main problem in this research area is that most of the state of art interaction techniques for immersive VEs have been designed and developed in an ad-hoc fashion, often because a new application had unusual requirements or constraints that forced the development of a new technique.

Currently, the main research question in this area is what may be the best technical solution for easy and natural 3D interactions within VEs. The general trend is focusing on immersion [13], but to obtain high quality immersion you need stereo vision for the visualization, linked to a 3D tracking device in order to track the position of devices and of user head. Indeed, such technical solutions allow the images to be generated such that virtual tools can be co-located with parts of the users body or with the real devices he is using. In this way, a user feels like his arms, hands, or devices were really embedded within the VEs. In this context, the interaction metaphors that are usually used, such as virtual hands [14], virtual rays [15] or virtual 3D cursors [16], are interesting also for CVEs because they provide a natural 3D representation that is perceptible for other users of the CVE. Nevertheless, Bowman et al. in [17] states that 2D metaphors and input devices have also to be considered for 3D interactions and are always considered user friendly, because they are sometimes easier to use than 3D input devices and tools.

CAVE2™ represents the state of the art in VE systems and it is the address of this work, that has the goal of proposing a new model of interactions with such a system. It is, in fact, a VE system in constant developing at the EVL at UIC.

CAVE2™ [3] has a diameter of about 24 feet and is 8 feet tall, consisting of 72 near-seamless passive stereo off-axis-optimized 3D LCD panels, a 36-node high-performance computer cluster, a 20-speaker surround audio system, a 10-camera optical tracking system and a 100-Gigabit/second connection to the outside world. The system is shown in figure 2.

CAVE2™ is a large-scale VE: it is a hybrid system that combines the benefits of both scalable-resolution display walls and VR systems to create a seamless 2D/3D environment that supports both information-rich analysis as well as VR simulation exploration at a resolution matching human visual acuity. The system covers 320 degrees around the user, displaying information at 37 Megapixels in 3D or 74 Megapixels in 2D.

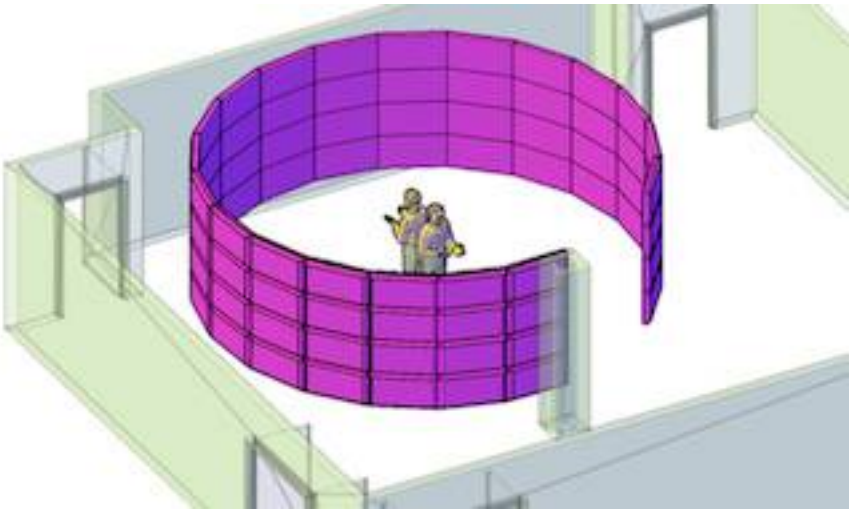

Figure 2: Design of CAVE2™ prototype. Image provided by J. Talandis, EVL [2]

The result is a system that enables users to study a wide range of new problems at the intersection of human-computer interaction, virtual reality, computer graphics, high-performance computing, high-speed networking, and computer-supported cooperative work. It may immerse people into worlds too large, too small, too dangerous, too remote, or too complex to be viewed otherwise. An example of CAVE2™ visualization is shown in figure 3.

CAVE2™ is programmable using a number of application programming interfaces. While some are virtual-reality only, it is SAGE (Scalable Adaptive Graphics Environment) [18] that enables CAVE2™'s huge wall to be partitioned into "windows", enabling one or many 2D and 3D windows of information to simultaneously be displayed.

2.2 Collaborative Virtual Environment (CVE)

One of the most challenging area of research, related to VE systems, is CVE model, because it adds new challenges regarding human-factors, networking, graphics capabilities and databases [19].

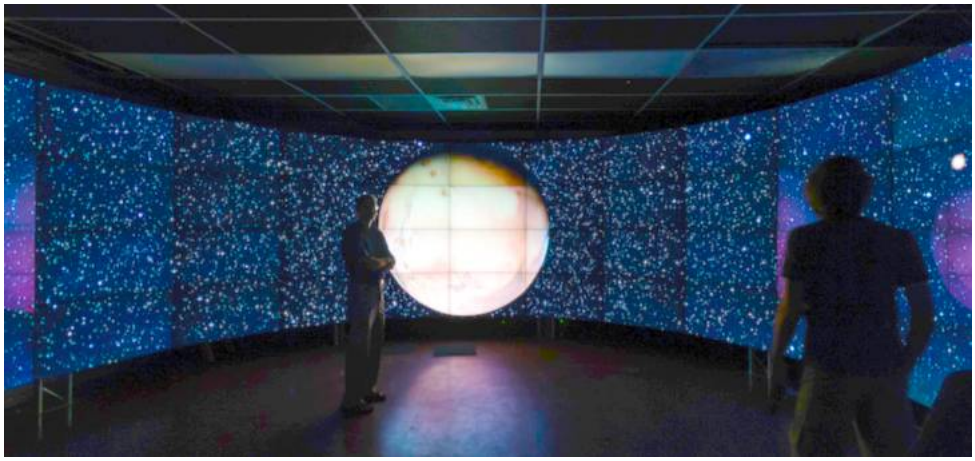

Figure 3: Individuals in the circular CAVE2™ navigate 3D Mars. Image provided by Luc Renambot, UIC/EVL (data NASA and European Space Agency) [3]

For example, while traditional VE systems do not have particular problems with networking bandwidth consumptions, in CVE the network is greatly used for connecting remote users. Moreover, in traditional VE systems the focus has always been the development of natural interfaces for manipulating virtual objects and traversing virtual landscapes, in CVEs a big challenge is represented by collaborative manipulation, that requires the consideration of how participants should interact with each other in a shared space, in addition to how co-manipulation of objects should be done.

These new challenges, that arises when CVEs are considered, make the rapid construction of complex CVEs difficult and, in most of the cases, expensive. In the current state of the art, these problems have resulted in domain specific solutions, that have been specifically designed to solve a small range of problems [20–25]. This work tries to overcome these problems, giving a general framework, instead of proposing an ad-hoc solution or using a particular application,

and to propose a new low-cost model for remote collaboration for a given VE system, such as CAVE2™ , exploiting clients connections through common browsers.

One of the main use of CVE is collaborative design [26]. Both in the research and in industrial contexts, small groups of users, either synchronously or asynchronously, try to solve a common problem, hoping that their combined experiences and expertise will contribute new perspectives and solutions. Compared to 2.5D CAD packages, the interfaces for 3D modeling in VEs are still relatively imprecise, thus most of collaborative tasks in CVE are focused on design evaluations, redesign or brainstorming for new design possibilities [27–29].

In most CVEs to-date, the default assumption has been to display the collaborative world with one or multiple perspective to all its participant VE systems [19,26]. These works are however based on a strong assumption: the collaboration model is always based on a connection between two or more VEs. The model that can explain this assumption is represented by CAVE Research Network (CAVERN) [19], that is a solution for supporting collaborative participation of industrial and research institutions equipped with CAVEs and high-performance computing resources, interconnected by high-speed networks, for the purpose of supporting: collaborative engineering and design; education and training; and scientific visualization and computational steering, in VR. CAVERN system, for example, uses the Information Request Broker (IRB) for managing remote database connections. IRB is an autonomous repository of persistent data driven by a database, and accessible by a variety of networking interfaces. In figure 4, CAVERN system overview is shown.

The cited works have shown that only research or industrial institutions equipped with these expensive VE systems could collaborate on ad-hoc applications that permit long distance communications or shared databases. What Porthole wants to achieve is, instead, a more accessible and general way of collaboration, with a framework that enables any device with Internet connection to be able to interact both locally and remotely with a scientific visualization,

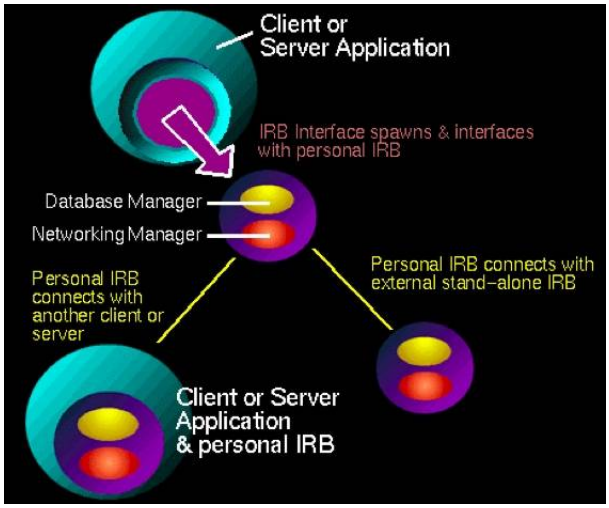

Figure 4: CAVERN (CAVE Research Network) overview. Clients/Servers use the IRB interface to spawn personal IRBs with which to communicate with other clients/servers or standalone IRBs[2]

exploiting a centralized cluster for computations. Moreover, Porthole aims to create an easy to use framework for interface creations, available for any CAVE applications developer who wants to enable this kind of interactions, not focusing on an ad-hoc solution as the previous works in these area.

The collaborative model is different from the ones proposed in common state of art researches, because this solution looses the immersive properties of VE systems on connected clients. However, the goal is to easily show something interesting found during exploration of a visualization to a remote user, who can easily look at what the user inside the VE sees, while potentially being in any part of the world, and without the need of installing any ad-hoc application, using the VE system "as a service".

2.3 Handheld devices and virtual environments interaction

Virtual Environments (VEs) provide unique perspectives of complex and large datasets, that may not be explored with standard desktop hardware. The typical interaction models are based on tracked devices and tools, which allow users to use natural gestures in order to interact with these systems. The state of art research has shown that the practicality of these interactions diminish as the set of user controllable parameters and options increases [5]. In this section, the motivations and earlier works related to decoupled two dimensional interfaces for VE systems are presented.

Two dimensional GUI may be advantageous for both familiarity and organizational reasons [5]. In VEs, however, their use can present difficulties in term of usability when these interfaces give no tactile feedback and float freely in the virtual world, compared, for example, to a click or a touch on a button in a typical application GUI. A reason is that a two dimensional GUI inside the three dimensional world requires further interactions to adjust it to a reasonable orientation, in order to facilitate visibility, and reach these GUI elements. The ubiquity of to-date touchscreen devices suggests a potentially low cost solution to these problems, by providing a tactile handheld responsive and user-friendly interface.

VEs present the opportunity for users to interact with data in ways that standard desktop computers cannot provide, such as touch inputs. Wand devices such as the Nintendo Wii Remote [30] in figure 5 and tracked gloves [31] make the user able to interact with VEs by either pointing or gesturing. However, earlier work has shown that these kinds of input modalities are ill suited for some tasks [32]. For example, pointing in a three dimensional space can be imprecise, and glove degrees of freedom render them unwieldy, because they may fail to recognize gesture or may accidentally trigger unwanted inputs. In fact, research has proven that the common two dimensional GUI menu proves effective [10].

Figure 5: The Nintendo Wii Remote

Two dimensional menu presented in a three dimensional space has its own disadvantages, as seen before. In particular, a menu that floats in the VE virtual world may be lost by the user when he changes position or orientation inside the VE. Moreover, it may become unreadable if it is fixed in a predefined position inside the world, or difficult to manipulate if the user is physically in a bad angle. These issues can be solved by always rendering the menu in the user's field of view, showing it on the screen at a fixed position, but this solution results in the menu occluding part of the scene. Research works have tried to reduce this problem by minimizing the menu when not needed[10], but the problem is always present. Porthole, and in general a two dimensional interface on an handled device, can completely overcome this occlusion, by presenting applications controls on a decoupled interface.

In figure 6, a VE application presents a 2D menu inside a 3D world: this results in occlusion of part of the scene. Moreover, the user needs to point a GUI item in order to change a parameter.

With Porthole, instead, the same 2D interface can be presented on a handheld device, avoiding the occlusion effect.

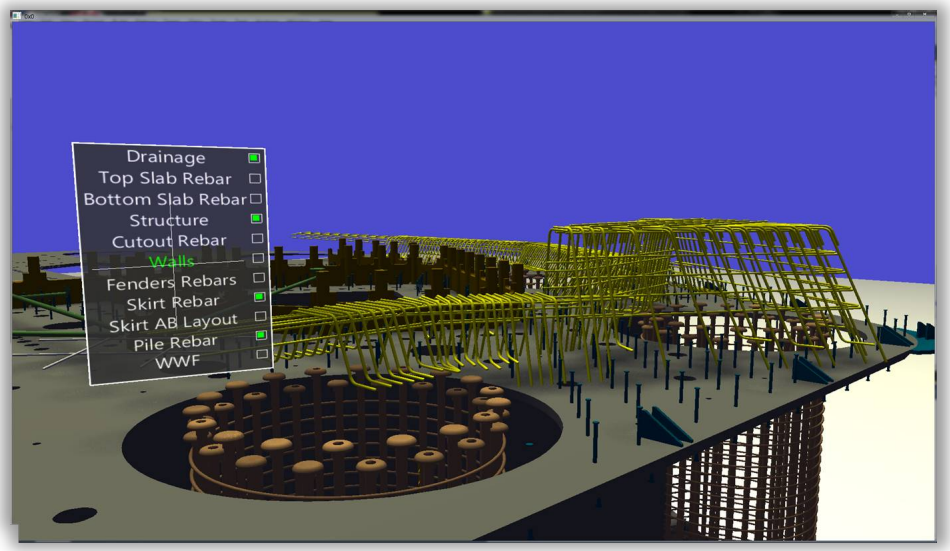

Figure 6: A VE application that uses a 2D menu inside a 3D world.

The two dimensional menus have another problem: the interaction with them needs either pointing or touching, both of which can be imprecise when the additional three dimension is involved. For example, the problem can arise when the user has to guess at how far he has to be in order to interact with the GUI, or to guess what gesture may trigger the interaction with the menu (a problem that often arises with tracked gloves). The possibility of having an interface

on a handheld touchscreen device solves also these issues, since the touchable surface provides tactile feedbacks.

Moreover, traditional interaction methods based on combinations of computer mice and keyboards often do not scale to the number of users or the size of the display [33]. Thus, these traditional HCI models are ill suited for interactions with VEs.

Moreover, recent studies on the WILD room [32], an environment used for exploring multi-surface interaction that includes an ultra-high resolution wall display, a multitouch table, a motion tracking system and various mobile devices, show that distant pointing with standard laser pointing [34] and with gyroscopic mouses have many limits on creation of an efficient and intuitive pointing technique for the wall. Thus, the conclusion is that touch interfaces on handheld devices can overcome these limitations.

The research area of handheld devices interfaces for VE interactions has shown little progress on this field, mainly due to the big challenge of having a global interface that would be usable from every device, avoiding ad-hoc solutions such as OS specific applications. In particular, two works has explored this area of research [4,5], but have some limitations that Porthole overcomes.

The work of Bayon et al. [4] introduced the concept of decoupled interfaces for VE systems, a concept also present in Porthole. In their work, the authors describe how decoupled interactions improve some of the functionalities in interactive VEs, by exporting aspects of three dimensional manipulation tasks into the two dimensional interaction domain with three main objectives:

- to provide an easier mechanism to trigger interaction and access functionality embedded within the VE;
- to support multimodal and multi-device forms of interaction to perform the same actions;
- to allow more than one user to participate in the interaction while using the VE as non-immersive user.

In Bayon et al.[4], the VE consisted of a vehicle model that provided a number of interaction opportunities. The VE allows users to change virtual world parameters and options from an HyperText Markup Language (HTML) tree-structure menu, accessible from handheld devices. In figure 7 a user is interacting with the prototype VE using a wifi-enabled handheld device. However, the main problems of this solution are:

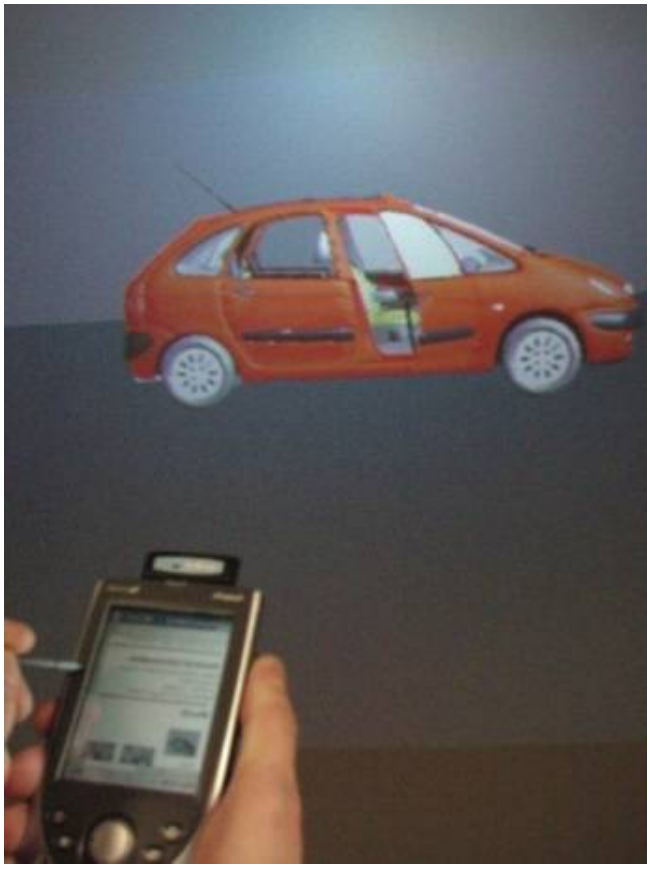

Figure 7: Decoupled interaction using a handheld device[4]

- the interface is application specific, because it is constrained to a particular HTML structure, because the solution proposed uses a list of pre-generated images, with a couple of links each, in order to enable some interactions;
- there is no VE to Personal Digital Assistant (PDA) streaming of images;
- there is no direct manipulation of VE user's position nor orientation inside the virtual world.

These problems are solved in Porthole, while maintaining the proposed general idea of using a HTML-based decoupled touchable interface on a handheld device.

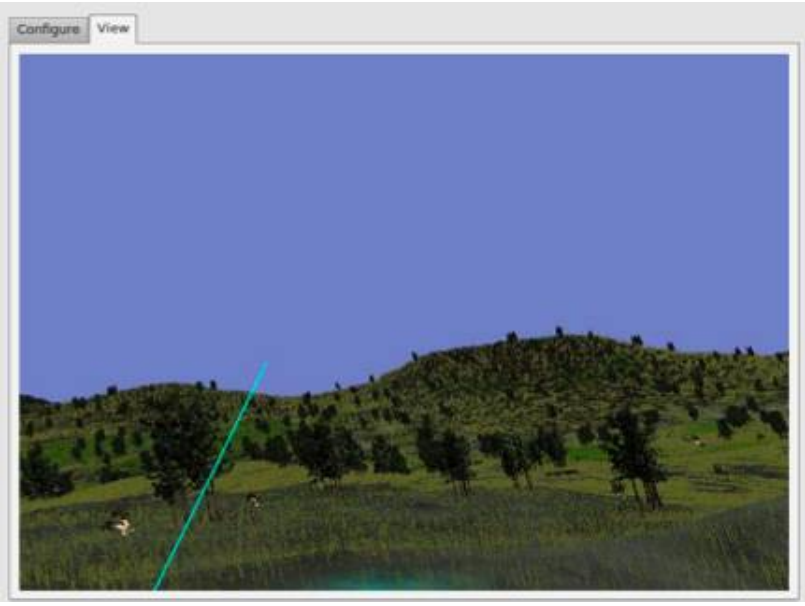

Figure 8: The Scrybe tablet Qt application [5]

Figure 9: The Scrybe Android smartphone application [5]

In Hoang et al. [5], the authors describe Scrybe, a VR toolkit that allows a set of smartphone and tablet interfaces to be used for VR applications. In particular, two domain specific solutions are used, one is a Qt application for tablet interactions, shown in figure 8, while the second is an Android application, shown in figure 9. However, also this solution presents some problems:

- it is a domain specific solution, working only on supported tablets and Android smartphones;
- it needs the installation of a specific application on the handheld devices;
- there is no direct manipulation of VE user's position nor orientation inside the virtual world from the devices.

These problems are solved in Porthole, while maintaining the idea of streaming a portion of the VR world and generating GUI events using handheld devices.

2.4 Final remarks

In this chapter, the research areas involved with this work have been presented and discussed. As described, many problems have not yet been solved and Porthole tries to make a step further in these fields.

Interactions between one or multiple users and large screen display environments have always presented a set of limitations that need to be addressed in order to create a usable and effective interaction model. First of all, Porthole establishes a new interaction model for managing VE virtual world, compared to the earlier works, because:

- handheld touchable interfaces overcome some limitations of interacting with two dimensional GUI in three dimensional virtual world for options and parameters manipulation;
- they avoid problems related to tracked devices or pointing techniques that make users not able to move freely inside the VE system, because they may trigger unwanted interactions or make pointing difficult from some positions;
- the decoupled interface enables both local and remote interactions with the VE, proposing a new model of collaboration with people that do not have access of classic CVE systems such as a CAVE;
- using the browser and standard sockets protocols, instead of OS specific applications and custom sockets, makes a VE system potentially universally accessible;
- implementing a framework for VE applications developers, instead of proposing an ad-hoc VE application, makes Porthole usable for many different types of applications.

In the next chapters, the proposed approach and implementation are presented, providing justifications for the assertions about Porthole made in this chapter.

CHAPTER 3

PROPOSED APPROACH

This chapter focuses on the high level design of Porthole, in order to present the general idea behind its implementation, the context in which it has been implemented and the technologies chosen to support the implementation phase.

The proposed approach is analyzed starting with requirements, described in section 3.1, ranging from general concepts to a concrete use case scenario. Then, the technologies involved are described in section 3.3. Finally, the design phase ends with a system overview, in order to present the solution that has been implemented.

3.1 Requirements

In this part of the chapter, the Porthole requirements are described. First of all, in section 3.1.1 an high level description of Porthole behavior is analyzed using the five Ws method, which is useful for conveying the idea of the work requirements to be, then, implemented. In section 3.1.2, instead, a concrete use case scenario is presented, which is helpful for getting the idea of users interactions with a VE application, using the Porthole framework.

3.1.1 The five Ws

In this section, the five Ws method is used in order to best convey the general idea behind this work and to describe its main characteristics.

What: What is Porthole?

Porthole is an framework for decoupled HTML5 interface generation for Virtual Environments (VEs). Basically, it generates an HTML5 page that maintains a full-duplex connection with a VE using HTML5 WebSockets, in order to make the user able to manage

application parameters and options using HTML5 inputs such as buttons, sliders, radio buttons, etc, and to get a streaming of images from the application, with which the user can interact in order to move inside the three dimensional virtual world. The Porthole camera could be in sync with the VE camera, so that every interaction is reflected on the VE screen, or not, so that it could be used for free roaming inside the world. The interface supports mouse, keyboard and touch inputs and could be customized by the VE application developer, in order to create a domain specific interface. Moreover, the framework could be used to specify different interfaces based on device specifications, tailoring them to fit application needs.

Who: Who should use Porthole?

Porthole should be used, during development phase, by VE applications developers in order to easily expose a decoupled interface accessible by HTML5 enabled browsers. All the clients management is transparent to the developers, that have to simply write an Extensible Markup Language (XML) file and, optionally, a Cascading Style Sheet (CSS) file in order to have their custom interfaces. The end users, then, can connect to the VE system using any device that has an HTML5 enabled browser, and interact locally or remotely with the VE system.

Why: Why Porthole is needed?

The main reason for using Porthole is because a VE applications developer may want to manage application parameters and application options, which is difficult, if not impossible, with standard VE interactions models, such as direct VE screens touches, laser pointing, tracked devices or tools, and console joysticks. Another usage of Porthole is for remote collaboration, when the VE user may want to share something interesting, found during exploration, with a remote user (for example an expert of that application domain) in order to get his opinion on it.

Where: Where to find Porthole?

Porthole can be found inside omegalib[35], a library used for the creation of Virtual Reality (VR) applications, which are used in VE systems.

When: When Porthole is used?

As already pointed out, Porthole is necessary when typical interactions models are not enough, for example when a VE application needs to generate a GUI that may be used to change application parameters, such as objects colors, virtual world brightness, visualization modalities, etc.

3.1.2 Use case scenario

A use case scenario is described in this section, in order to explain the HCI model that Porthole implementation follows. The system physical components are, basically: VE cluster and displays, Internet network and client devices. In particular, Internet network is necessary as the mean of communications, in order to be able to support both local and remote multi-user interactions. This is the highest level of abstraction, and the starting point for defining the system more in details.

The communication among cluster nodes and between cluster nodes and displays is already done by omegalib[35], which is described in section 3.3.1. The goal is to create a one-to-many communication between a server, inside the VE system, and one or more client devices.

The desired functionality that Porthole wants to achieve is that the server, not the input device itself alone, is responsible for setting up the layout of the interface. Thus, the system must know the devices characteristics and take a decision, based on the application domain and the received specifications, and communicate back to the devices how the interface should be composed. Thus, the server receives the communications on real-time from the devices, that

connect through their browsers (and not with an ad-hoc installed application), and decides what kind of interface to serve to each of them.

With this in mind, a typical use case scenario is described, which involves a client that connects to the VE using the browser, receives a GUI (in which, in this case, both camera streaming and other interaction elements are defined by the application developer) and interacts with the VE system:

1. The VE application starts
2. The VE application adds Porthole as an omegalib service
3. Porthole server waits for clients
4. A client connects, thus an Hypertext Transfer Protocol (HTTP) Request is received by the server
5. Porthole server sends the HTML page back to the user
6. HTML page contains JavaScript code that sends an HTTP Upgrade request, opening a full-duplex bidirectional connection, using HTML5 WebSockets
7. Porthole server switches client protocol from HTTP to HTML5 WebSockets protocol
8. Once the socket is opened, client sends it's description to Porthole server, through the socket opened
9. Porthole server sends back a GUI interface
10. Porthole server starts streaming VE images to the client (because, in this case, inside the GUI created there is a camera canvas defined)
11. User interacts with the GUI and a Porthole event is sent to Porthole server (for example, a camera movement event)
12. Porthole server applies changes accordingly to the received event (for example, the camera is moved)

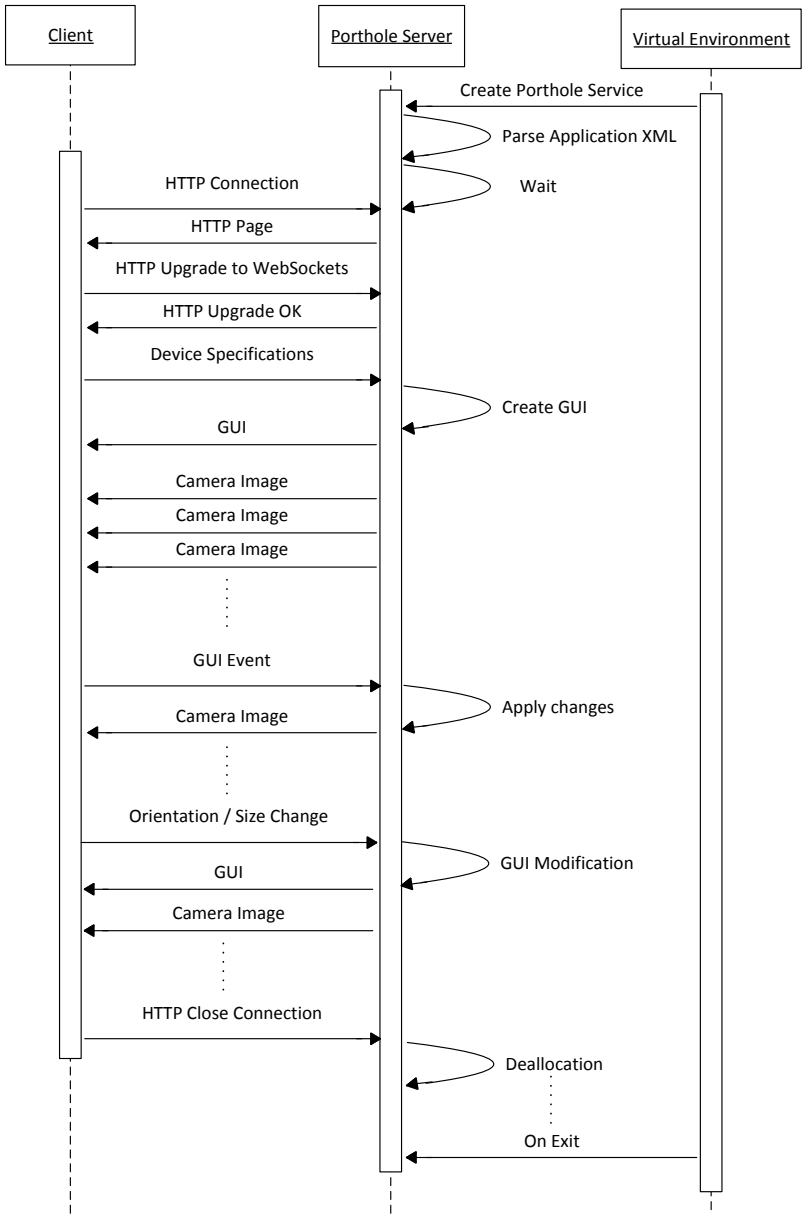

Figure 10: Porthole use case scenario: UML sequence diagram

13. Images streaming reflects the change
14. After a while, client may change its specifications (for example, device orientation changes, or the browser windows is resized)
15. Porthole client code intercepts these changes, and sends the new specifications to Porthole server.
16. Porthole server creates the new interface, while maintaining the current camera state, and sends back the new GUI
17. Client receives the new GUI
18. At a certain point, client disconnects and the server-side resources associated are released
19. When the VE application receives a termination event, the server stops and releases memory allocated
20. VE application ends

The Unified Modeling Language (UML) sequence diagram associated with these steps are shown in figure 10. Notice that while Porthole server manages this client, obviously it can serve other clients as well, which in turn makes this interaction model multi-user.

The implementation of these steps are described in Chapter 4.

3.2 System overview

This section provides an overview of the system, which makes clear how the system supports the use case scenario described in section 3.1.2.

The global system overview is shown in figure 11. As depicted, Porthole is designed to be inside omegalib as an additional service, that can be added by developers as any other service like Wiimote, Kinect, Wand, and so on. Porthole service, once requested, creates a server in a

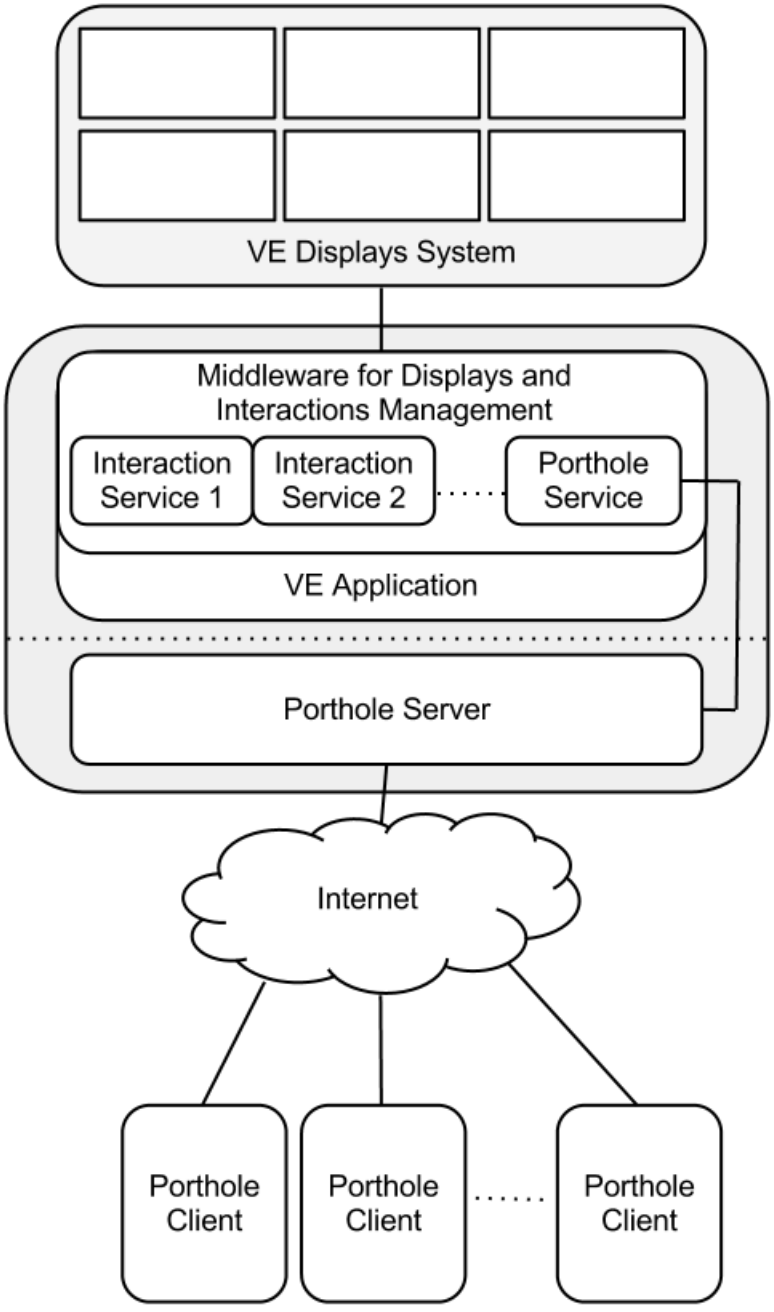

Figure 11: System overview

separate thread in order to be able to serve client devices that want to connect to the VE system using their browser. The behavior of this server has been already described in section 3.1.2.

The proposed architecture brought to the choice of technologies described in section 3.3. In particular, the service and the server are written in C/C++. The server, in particular, is designed to support both HTTP standard protocol and WebSocket protocol. Clients receive an HTML5 page; the WebSocket is managed using JavaScript; the <canvas> element is used to draw the images received; *Hammer.js* is used to attach callbacks to touch screen events; jQuery is mainly used to manage browser events and the style of the page is created with jQuery Mobile.

3.3 Enabling technologies

Porthole uses a set of technologies for its implementation. These are described in this section, so that it will be easier to follow the implementation phase in the next chapter.

This section is divided as follow: omegalib library, in which Porthole is implemented as an additional service, is described in section 3.3.1; HTML5 is presented in section 3.3.2, in order to present all the features introduced with this technology; then, the HTML5 features used inside this thesis work are analyzed in details in sections 3.3.3, 3.3.4 and 3.3.5, presenting respectively the canvas element, the WebSockets API and the Touch Events API; finally jQuery and jQuery Mobile frameworks are described in section 3.3.6.

3.3.1 Omegalib

The official omegalib webpage [35] defines it as "a middleware designed to ease the development of applications on virtual reality and immersive systems." In particular, it is currently used at the University of Illinois at Chicago for CAVE2™ management, which is the main target of Porthole. That's the reason why this section briefly described it, in order to give the reader an overview of this library.

Omegalib is a library with many functionalities, which can be used in order to develop applications based on different technologies and to make them scale where multiple Graphics Processing Unit (GPU) units, displays or cluster nodes are available, such as inside cluster-driven VEs. The supported types of applications, whose descriptions are taken from their official webpages, include:

Open Graphics Library (OpenGL) [36]

a cross-language, multi-platform API for rendering 2D and 3D computer graphics;

OpenSceneGraph (OSG) [37]

an open source 3D graphics application programming interface, used by application developers in fields such as visual simulation, computer games, virtual reality, scientific visualization and modeling;

Cyclops [38]

a utility library that sits on top of omegalib and OpenSceneGraph, and it is designed to simplify development of applications that need to draw complex and visually pleasant scenes, without having to deal with the low-level details of OSG;

Visualization Toolkit (VTK) [39]

an open-source, freely available software system for 3D computer graphics, image processing and visualization.

Omegalib makes applications scale with the number of available displays, so that it may be used with either single display and multi displays systems. It also supports hybrid systems, managing both 2D and 3D contents visualization on the same single or multiple displays system.

One of the main advantage in using omegalib is that it supports a wide range of input peripherals, such as controllers, motion capture systems, and so on, but the lack of support for handheld devices is important when VE users may want to change some parameters. Thus, Porthole address this limitation.

Omegalib is written in C/C++, but supports Python scripting, in order to ease the development of VE applications. This feature is important and is used for direct Python code injection described in chapter 4, during the implementation phase.

3.3.2 HTML5

In this section, we discuss the main characteristics of HTML5 and the reasons why it has been necessary to use some of its features for developing the Porthole browser front-end. In fact, the previous versions of HTML do not support: a real time control of images drawing, because the <img> tag does not provide a way to control when the image source must be drawn; multitouch events; full duplex bidirectional connections between client and server.

HTML5 is a markup language used for defining World Wide Web contents structure and presentation. This fifth revision of HTML is still under development, as of Fall 2012. HTML5 has been created not only for enhancing HTML capabilities, but also to include under a unique standard three currently available specifications: HTML 4.01, XHTML 1 and Document Object Model (DOM) Level 2 HTML. Since HTML5 is not a standalone software nor a browser add-on, the browsers vendors must natively support it in order to make the new functionalities universally accessible.

Thus, HTML5 can be considered as a set of new functionalities available for web applications development, built on top of HTML existing standard. The main improvement is a much better support for multimedia and server communication, comprising a set of small additions to the existing web standards. Currently, each browser vendor decides which new features to

implement, eventually implementing all the new standard concepts, such that every browsers will be HTML5 compliant.

The up-to-date HTML5 specifications can be found online, and are still under active development:

- W3C HTML5 specifications [40]
- WHATWG HTML5 living standard [41]

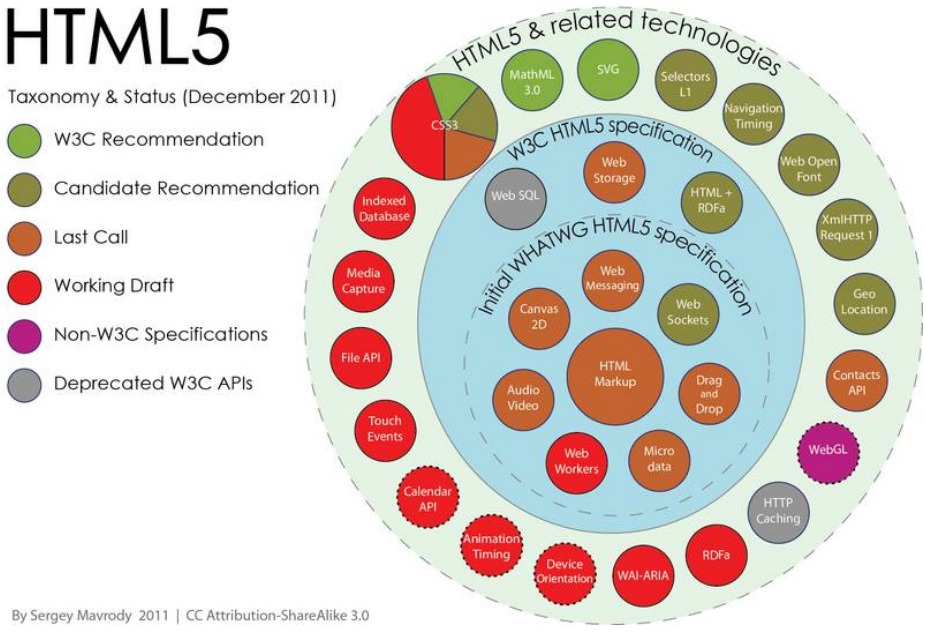

Figure 12: HTML5 related APIs overview. Image from HTML5 and CSS3 Quick Reference [6]

As a general overview of the HTML5 draft, the proposed standard adds many new syntactic features,. The most relevant tags introduced may be <video>, <audio> and <canvas> elements and tags for managing Scalable Vector Graphics (SVG) contents and MathML mathematical formulas. These features are designed to make it easy to include and handle multimedia and graphical contents on the web without having to resort to proprietary plugins and APIs. Other new elements, such as <section>, <article>, <header> and <nav>, are designed to enrich the semantic content of documents. New attributes have been introduced for the same purpose, while some elements and attributes have been removed. Some elements, such as <a>, <cite> and <menu> have been changed, redefined or standardized. In addition to these tags, the standard introduces some new Application Programming Interfaces (APIs), that can be accessed using JavaScript, such as the WebSocket API that has been extensively used in Porthole development, and is discussed in section 3.3.4. HTML5 is then a potential candidate for cross-platform development, better supporting mobile development than its predecessors. In figure 12, an overview of HTML5 new features is shown.

Porthole couldn't have been implemented using browsers as front-end without HTML5 functionalities. In particular, the <canvas> element, the WebSockets API and the Touch Events API have been used for VE images streaming and manipulation, and for passing bidirectional event messages between the client and the server. These couldn't have been achieved real-time using other pre-HTML5 standards. The following sections analyze these three features used in this work.

3.3.3 Canvas element

HTML5 introduces the <canvas> element: a resolution-dependent bitmap canvas that can be used for rendering graphs, game graphics, or other visual images on the fly. Basically, a canvas element is a rectangle inside the body of the HTML page that can be used by JavaScript in order to draw anything on it.

A canvas element has no content and no border of its own. The Porthole markup for this element looks like this:

```
<canvas id="camera-canvas" width="300" height="300"></canvas>
```

This element is dynamically created by the Porthole server during the interface creation process, so that the correct width and height values are set, based on both application interfaces definition and the connected device specifications. This step is important because using a pre-defined width and height would result in distorted client-side images and too big or too small server-side allocations for VE camera images.

The canvas element can be accessed just like any other DOM element. In Porthole, for example, the canvas element is simply retrieved with:

```
var camera = document.getElementById("camera-canvas");
```

Now that the canvas object is defined, in order to be able to draw on it, the drawing context must be retrieved. Every canvas has a drawing context, which is where all drawing events happen. Just call its *getContext()* method to obtain the context. You must pass the string "2d" to the *getContext()* method.

Once the drawing context is retrieved, the context properties and methods can be used, and the changes are reflected inside the canvas element. In fact, the canvas is a two-dimensional grid. The coordinate (0, 0) is at the upper-left corner of the canvas. Along the x-axis, values increase towards the right edge of the canvas. Along the y-axis, values increase towards the bottom edge of the canvas.

The canvas drawing context defines a *drawImage()* method for drawing an image on a canvas. The method can take three, five, or nine arguments:

drawImage(image, dx, dy)

It takes an image and draws it on the canvas. The given coordinates (dx, dy) will be the

upper-left corner of the image. Coordinates (0, 0) would draw the image at the upper-left corner of the canvas.

drawImage(image, dx, dy, dw, dh)

It takes an image, scales it to a width of dw and a height of dh, and draws it on the canvas at coordinates (dx, dy).

drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)

It takes an image, clips it to the rectangle (sx, sy, sw, sh), scales it to dimensions (dw, dh), and draws it on the canvas at coordinates (dx, dy).

Figure 13: Browsers support for HTML5 Canvas element [7]

Now that it is clear what the new HTML5 canvas element is, the images streaming from the server to a client would be easier to understand. In chapter 4, the implementation of the streaming phase, achieved also using the canvas element, is discussed in details. Finally, notice that the canvas element doesn't have any particular compatibility problem with to-date mobile and desktop browsers, as shown in figure 13.

3.3.4 WebSocket API

In this section, the new WebSocket API introduced with HTML5 is described, because it represents one of the main concept behind Porthole development. In particular, this API enables web pages to use a new protocol for a full-duplex communication with a remote server. The main advantage is that it provides a great reduction in unnecessary client-server traffic and latency, compared to the pre-HTML5 polling and long-polling techniques used to simulate a full-duplex channel with remote hosts.

WebSocket API was originally part of the HTML5 specification, but since it represents a big and complex change in web applications development, it has been moved to a separate standards document [42], in order to keep it clean and focused. The same solution has been adopted for other important pieces of HTML5 initial specification like LocalStorage and Geolocation, which are not used in this work and, thus, are not described. There are different positions regarding the inclusion of WebSockets inside the HTML5 core: for sake of simplicity, they are assumed to be somehow part of HTML5 family.

One of the big advantages of HTML5 WebSocket API is that it account for network hazards, such as proxies and firewalls, in order to make streaming possible over any connection, plus the ability for upstream and downstream communications support over a single connection. In fact, WebSocket has the ability to traverse firewall and proxies, which nowadays represents a problem for many web applications. Some workarounds have been employed by pre-HTML5 applications, such as the Comet-style ones, that use long-polling as a rudimentary line of defense against

firewalls and proxies. This technique is working, but is not a good solution for applications that need to have a very small latency, such as less than 500 milliseconds, or high throughput requirements. Adobe Flash has also been used in order to provide socket support, but has several problems with proxy and firewall traversals, that WebSocket do not have. Moreover, Adobe Flash is not well supported by some browsers like iOS Safari Mobile, which represents a problem for Porthole development. This is another reason for choosing WebSocket for this work.

Regarding proxy traversals, WebSocket detects the presence of a proxy server in the client-server path and automatically set up a tunnel, used to pass through the proxy. This tunnel is created by issuing an HTTP CONNECT statement to the proxy server, which sends a request to the actual server for opening a TCP/IP connection to a specific host and port. Once the tunnel is created, the communication can proceed without problems through the proxy server.

The main steps for establishing an HTML5 WebSocket bidirectional connection are shown in figure 14. The protocol is designed in order to work inside the existing web infrastructure: one of its principle is that a WebSocket connection starts as any HTTP connection, guaranteeing backward compatibility with the existing technologies. Then, the connection with the server is opened by using the WebSocket handshake.

In the following listings, an handshake example is given. The browser sends an upgrade request to the server, with the indication of switching protocols from HTTP to WebSocket. The server, once this type of message is received, if it supports WebSockets, it agrees with the protocol switching, using another upgrade header (that is a an header with the line *Connection: Upgrade*)

```
GET / HTTP/1.1
Connection: Upgrade
Upgrade: websocket
Host: lyra.evl.uic.edu:4080
```

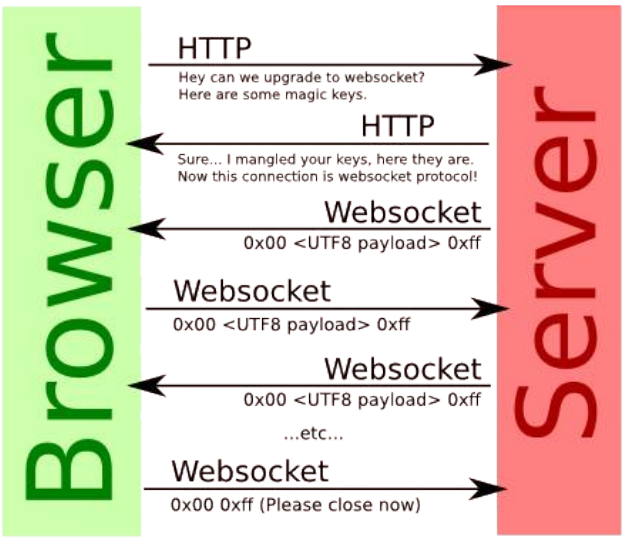

Figure 14: An HTML5 WebSocket connection. It starts with an HTTP connection and then opens a full-duplex channel with the server

```
Sec-WebSocket-Protocol: porthole_websocket
Sec-WebSocket-Key: d80aHIoy0XeH1MXIVSMkIg==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-webkit-deflate-frame
```

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: WebSocket
Sec-WebSocket-Accept: rLHCkw/SKsO9GAH/ZSFhBATDKrU=
Sec-WebSocket-Protocol: porthole_websocket
```

At this point the standard HTTP connection is closed and is replaced by the WebSocket connection over the same underlying TCP/IP connection. The WebSocket connection uses the

same ports as HTTP (80) by default. In Porthole, each application could specify its own port number, so that each one can have its own private WebSockets enabled server that provides the VE interface.

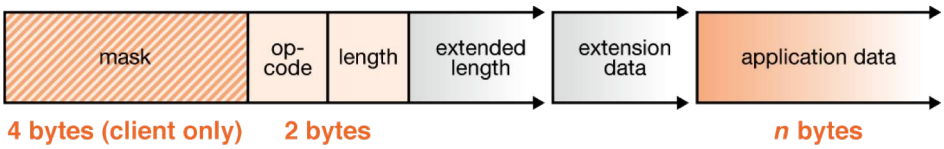

Figure 15: The WebSocket frame structure

Once established, WebSocket data frames can be sent back and forth between the client and the server in full-duplex mode. Both text and binary frames can be sent in either direction at the same time. The data is minimally framed with just two bytes. In the case of text frames, each frame starts with a 0x00 byte, ends with a 0xFF byte, and contains UTF-8 data in between. WebSocket text frames use a terminator, while binary frames use a length prefix. The frame structure is shown in figure 15.

In order to use WebSockets, Porthole uses an external library, called *libwebsockets* [43], which basically allows access to the WebSockets API inside a C/C++ environment. Thus, this library has been added to omegalib for using inside Porthole server, and it has been modified in order to fix some bugs and better support all the different WebSocket versions of the specification. In fact, the main bug of this library is that it doesn't fully support iOS devices and Opera browsers, because the buffer used does not take into account the fact that these devices do not end strings sent using WebSocket with the string terminator character. This problem caused

Porthole JavaScript Object Notation (JSON) parser to not working properly. Once fixed, iOS devices and Opera browsers have become fully supported by Porthole.

In conclusion, HTML5 WebSocket protocol provides an enormous step forward in the implementation of real-time web application and has been successfully used in Porthole in order to stream camera frames and interface events. In particular, Porthole enables VE applications developers to support smartphones, tablets and laptop interactions without implementing an ad-hoc WebSocket server, nor a specific client-side application in order to interact with the VE , resulting in a transparent and easy to use framework inside omegalib.

3.3.5 Touch Events API

HTML5 introduces also the concept of multi-touch events, which can be used by JavaScript in order to support touchable screens. Before HTML5, the browsers did not recognize manual gestures, mainly because there was no specification about how to expose such events. Basically, web browsers on touchscreen devices reacted to touch interactions as if they were simple mouse events, limiting possibilities to simple page scrolling and clicks.

The HTML5 Touch Events API specify how the browsers should expose touch events to JavaScript, in order to identify them, recording their location, movement, and so on. Touch events can be used by web developers to create dynamic, multi-touch experiences that will work on most web browsers. Browser independence leads to device independence: that's way this API is useful for Porthole in order to be device independent. Without HTML5 Touch Events, gestures like pinch to zoom and two fingers rotation couldn't be possible, simply because before HTML5, no web API was able to get multi-touch events.

Touch Events API is still a working draft and thus modern browsers have different levels of implementation of this standard. Regarding Porthole development, what matter is mobile browsers support that is already available for all the main mobile OS . The main web browsers

Figure 16: Browsers support for Touch Events API[7]

support for Touch Events API is shown in figure 16. There is still a lack for multi-touch events support in the desktop versions of most browsers, but Porthole is not intended for laptop with touchable screens, and they can use mouse and keyboard to interact with Porthole interfaces.

In order to unify the different implementation of the Touch Screen API that are currently available, that are basically different names for accessing the same objects, Porthole relies on a small JavaScript library, *Hammer.js*[44], which allows Porthole to add support for tap, drag and transform gestures.

Using *Hammer.js* is pretty simple, while the documentation at the moment seems pretty limited. Essentially, you instantiate an Hammer object linked to a DOM object and add a callback function for whatever gesture events you want to manage.

The Hammer object need to be created with a set of callback functions, one for each type of desired gesture, which can access the following elements:

originalEvent

The original DOM event.

position

An object with the x and y position of the gesture (e.g. the position of a tap and the center position of a transform).

touches

An array of touches, containing an object with the x and the y position for every finger.

scale

The distance between two fingers since the start of an event as a multiplier of the initial distance. The initial value is 1.0. If less than 1.0 the gesture is pinch close to zoom out. If greater than 1.0 the gesture is pinch open to zoom in. Defined only for Transform.

rotation

A delta rotation since the start of an event in degrees where clockwise is positive and counter-clockwise is negative. The initial value is 0.0. Defined only for Transform.

angle

The angle of the drag movement, where right is 0 degrees, left is -180 degrees, up is -90 degrees and down is 90 degrees. This picture makes this approach somewhat clearer. Defined only for Transform.

direction

Based on the angle, we return a simplified direction, which can be either up, right, down or left. Defined only for Drag and Swipe.

distance

The distance of the drag in pixels. Defined only for Drag and Swipe.

distanceX

The distance on the X axis of the drag in pixels. Defined only for Drag and Swipe.

distanceY

The distance on the Y axis of the drag in pixels. Defined only for Drag and Swipe.

3.3.6 jQuery and jQuery Mobile

jQuery is a JavaScript framework that extends JavaScript capabilities. jQuery can be seen as an abstraction layer, since it provides some functionalities that could be otherwise achieved with many lines of JavaScript. It's important to note that jQuery does not replace JavaScript, and while it does offer some syntactical shortcuts, the code you write when you use jQuery is still JavaScript code.

In Porthole, jQuery has been used for DOM elements manipulations and for browser events handling, such as for controlling and triggering touch inputs events. For example, it has been used for managing inline JavaScript mousedown event handlers, that need to be triggered also when the user start touching the screen, otherwise triggered when the user stop touching the screen (an odd behavior of iOS devices). jQuery is used in Chapter 4 during implementation description, where all the benefits of this framework can be seen.

jQuery Mobile is a user interface system built on top of jQuery framework. It basically consists of GUI elements and programming constructs that provide consistent functionalities for both mobile and desktop browsers. Its design is based on the concepts of progressive enhancement and graceful degradation, meaning that it is used for giving new functionalities on supported platforms and degrading to standard solutions when the browser does not support them, avoiding JavaScript errors.

Porthole relies on jQuery Mobile for user interface graphics, such as buttons style, so that they are already optimized to work for both mobile and desktop browsers. However, a VE applications developer could override this behavior, by defining his own style, as described later in chapter 4 during the implementation description.

The purpose of this section has been to briefly describe two technologies used by Porthole for defining client-side interfaces, jQuery and jQuery Mobile. The former is used for enhancing JavaScript functionalities, while the latter is used for giving a basic application interface style, consistent with both mobile and desktop browsers.

In Chapter 4, the details of Porthole implementation are described.

CHAPTER 4

PROPOSED IMPLEMENTATION

This chapter explores Porthole implementation, which follows design guidelines and technologies described in chapter 3. Porthole has been mostly implemented using Visual Studio 2010 and all the code written can be found inside the omegalib repository [35].

The code listings are presented in original language version when short and readable; otherwise pseudo-code is preferred, in order to give the reader a better overview of the algorithms implemented.

The structure of this chapter is divided into six sections. First of all, Porthole architecture is presented in section 4.1, in order to have a refined and more precise idea of how Porthole is implemented inside the cluster-driven system running omegalib applications. In section 4.2, then, XID structure is described, focusing on how this file should be written by VE applications developers, using a formal XML Schema definition, in order to explain each part of it. After that, in section 4.3, Pothole Server implementation is described, presenting how HTTP and WebSocket connections are initialized and managed. Then, section 4.4 describes how XID elements and interfaces are used to create the final HTML interface that is sent to the client. The client-side implementation is then described in section 4.5, which analyzes interactions management, messages handling and canvas image refreshing. Section 4.6 is focused on describing how events defined inside XID by developers can be triggered by client browser in order to execute server-side code, utilizing C++ functions binding and Python code injection. Finally, section **??** presents Porthole running inside CAVE2™, describing two scenarios: a single user interaction and a two users collaboration.

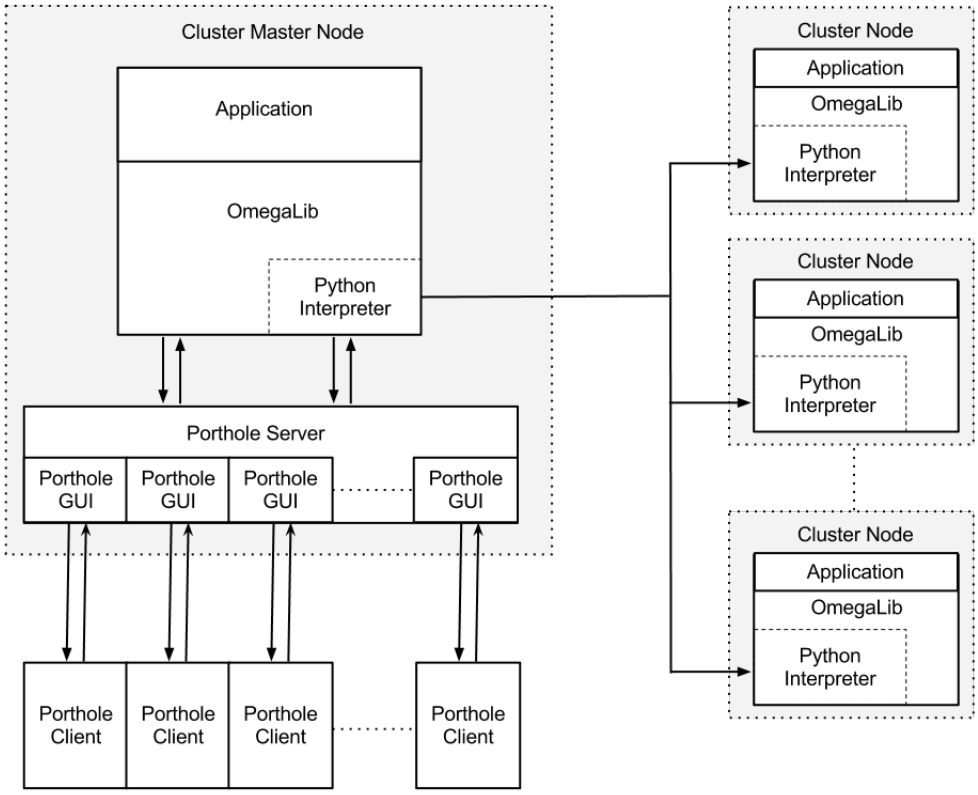

Figure 17: System architecture

4.1 System architecture

Starting from the design model described in section 3.2, let's focus on a detailed view of how the implementation has been done, taking in consideration the fact that the system can be cluster-driven, instead of running on a single machine. The system architecture is shown in figure 17.

In order to start Porthole Service, and as a consequence Porthole Server, the application should include few lines of code. These are different whether the application is written in C++ or in Python. In the first case, an example of C++ code used to start the service is:

```
// Initialize Porthole Service
PortholeService* service = PortholeService::createAndInitialize(port_number,
    path_to_xml, path_to_css);

if(service != NULL)
{
   // C++ Functions Bind
   PortholeFunctionsBinder* binder = service->getFunctionsBinder();
   binder->addFunction("up(event)", &up);
   binder->addFunction("down(event)", &down);
   binder->addFunction("left(event)", &left);
   binder->addFunction("right(event)", &right);
   binder->addFunction("zoom(event)", &zoom);
}
```

Otherwise, if the application is written in Python, the function to be called is:

```
PortholeService.createAndInitialize(port_number, path_to_xml, path_to_css)
```

In both cases, the service itself could be started with a similar function, which is *createAndInitialize*, that takes as parameters: the port number, used for starting the server; the path

to a XID file, which is described in section 4.2; and the path to a CSS file, linked by the final HTML5 file, that can be used by developers to tune interface style, if jQuery Mobile style is not sufficient or needs any fix.

Both C++ and Python *createAndInitialize* functions refer to the same implementation, which follows this pseudo-code:

```
procedure createAndInitialize( port, xml_path, css_path )
begin
   if this node is the master node then
      get omegaLib Services Manager
      add Porthole Service
      transform xml_path from relative to absolute
      transform css_path from relative to absolute
      start Porthole Service with port, xml_path, css_path
      return Porthole Service object
   else
      return null
end
```

The function, basically, first checks if it is being executed on the cluster master node. If this condition is true, *PortholeService* is created and added to omegalib set of registered application services. The system, in fact, is cluster-driven and is composed by n nodes, as shown in figure 17. Once an application is started on the master node, omegalib replicates the application on the other $(n-1)$ nodes. All the nodes run in parallel, splitting the GPU workload, since each node manages only a part of the multi-displays system.

Porthole Service attempts to start a server on every node, but since network connection is available only on the master node of the cluster, and there's no need of having multiple server instances, a check is needed in order to start the server only on that node. This is done inside

the *createAndInitialize* function, that initializes and starts Porthole Server on a different thread on the master node and, once the server is started correctly, it returns.

If the condition about being the cluster master node returns true, the relative paths given are transformed into absolute paths (remember that omegalib runs on both Unix and Windows machines, thus different addressing mechanisms need to be unified). Finally, the service starts, which in turn set the XID and CSS files paths and creates and starts Porthole Server in a separate thread. Porthole Server implementation is described in details in section 4.3.

The main difference between the two ways of starting the service is that if the application is written in C++, the developer may set the PortholeFunctionBinder object. This object is used by Porthole to match what function to be called once a client triggers a JavaScript function, described in details in section 4.6. It basically contains two HashMaps:

```
std::map<std::string, memberFunction> cppFunctionsMap;
std::map<std::string, string> pythonFunctionsMap;
```

These maps are used to redirect the event received to a defined function. The *cppFunctionsMap* contains the functions added with the previously seen function:

```
void addFunction(std::string, memberFunction);
```

It basically fills the map with the C++ functions defined by the user.

The *memberFunction* object has the following type definition, representing a function pointer:

```
typedef void(*memberFunction)(PortholeEvent&);
```

The second map object, called *pythonFunctionsMap*, instead, is filled by Porthole parser from the XID file. This difference is due to the fact that Python is a scripting language, thus Python code could be parsed, added to the map, and, when necessary, passed to the Python interpreter at runtime. C++ instead is a compiled language and, moreover, does not support reflection, thus an explicit mapping must be done in the code. More details on the mapping

between JavaScript interactions events and the corresponding server-side function can be found in section 4.6.

Once Porthole server is running and a client connection is established, a *PortholeGUI* object is allocated. This object is responsible for managing the interface, the omegalib camera associated with that user, and it contains other user related variables. In order to understand how this object works and how the interface is created, we introduce in section 4.2 the concept of XID, written by the VE application developers in order to specify all the interfaces tailored on client devices specifications.

4.2 XML Interfaces Description

Before diving into server implementation, let's introduce and describe the concept of XML Interfaces Description (XID): a XML file that makes developers able to describe clients interfaces based on their screen size and orientation.

The XID file is necessary to give an omegalib developer the ability of specifying the set of interfaces elements, their disposition and the interactions to be enabled in each type of client connected to the VE system. Moreover, developers could directly bind the touchable GUI elements events and the application functionalities in this XML document.

More in details, this XML file has a root element called <*xid*>, that can be roughly subdivided into two parts:

<elements>

which contains a list of elements to be used to create the interfaces for all the clients

<interfaces>

which contains a set of interfaces for each type of device the developer wants to support

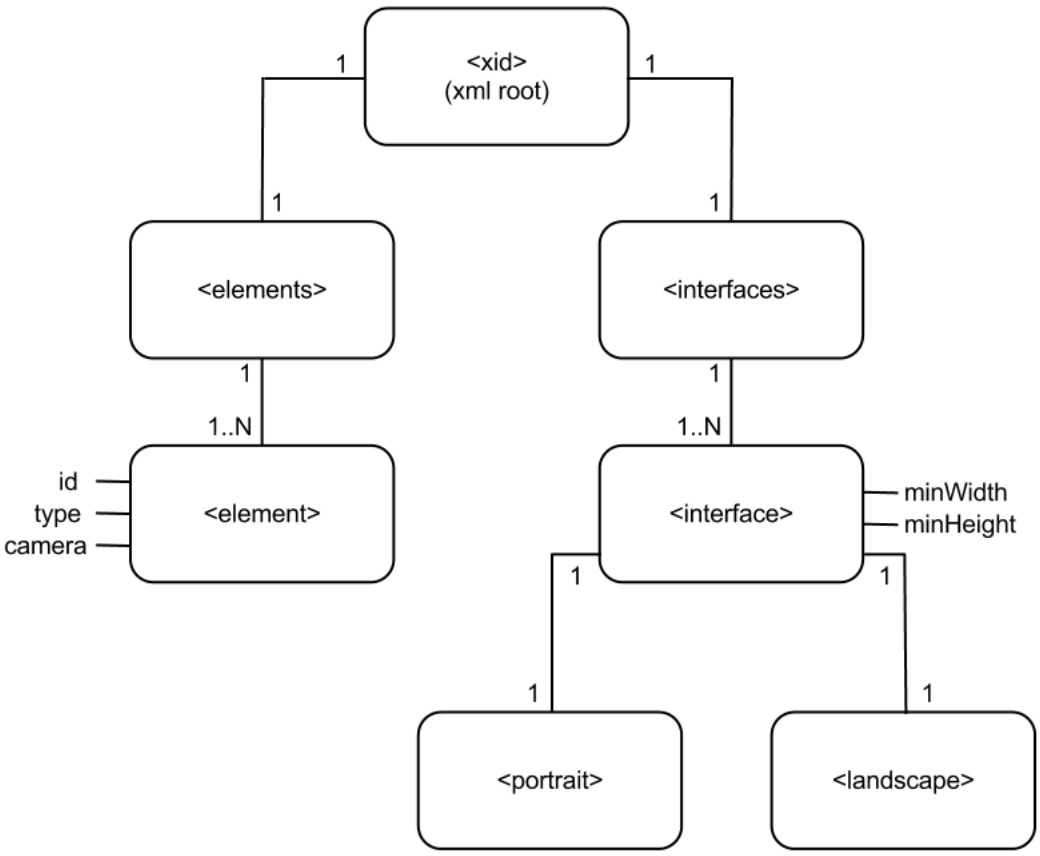

Figure 18: XML Interfaces Description (XID) structure

The general overview of this XML structure is shown in figure 18. The more detailed XML Schema that can be used to validate a XID file can be found in Appendix A.

The <elements> XML element contains the number and type of user interface parts, each one defined with a <element> XML element. Each one represents a main block for the construction of client GUI. In particular, the <elements> part contains how many global interface blocks as needed, and each <element> is divided into two types.

A <element> must be created inside <elements> with a set of attributes specified by the XML Schema as:

```
<xsd:attribute name="id" type="xsd:string" use="required"/>
<xsd:attribute name="type" type="xsd:string" />
<xsd:attribute name="camera" type="xsd:string" />
```

Thus, the *id* attribute is required and is used to identify a GUI element inside a <interface> content. The *type* attribute is used inside the <elements> part of the XID in order to specify its type, which can be:

HTML

This type of element can include any HTML piece of code. JQuery Mobile support is available in the client, thus one could for example specify buttons, sliders, radio buttons, and so on, simply using JQuery Mobile classes. The jQuery Mobile style will apply to all HTML DOM elements. Thus, in order to disable jQuery Mobile style on a particular HTML tag (and all its children), the developer can use the attribute *data-role=”none”*;

Camera

a Camera element is converted into a proper Canvas element used to display an images stream coming from a virtual world camera.

The last attribute, *camera*, is defined only for elements whose type attribute is equal to "Camera" and, currently, can assume two values:

Default

the omegalib camera associated with this element is the default one inside the VE; thus, any interactions in the VE is reflected in the interface canvas, and vice versa;

Custom

the omegalib camera can freely move inside the VR world, based on interface interactions.

The second part of XID document describes all the interfaces. Each one is defined as a composition of the previously defined <element> and, as could be seen in the complete XID XML Schema in Appendix A, must follow this specification:

```
<xsd:complexType name="interfaceType">
  <xsd:all>
    <xsd:element name="portrait" type="portraitType" minOccurs="1" />
    <xsd:element name="landscape" type="landscapeType" minOccurs="1" />
  </xsd:all>
  <xsd:attribute name="minWidth" type="xsd:int" use="required" />
  <xsd:attribute name="minHeight" type="xsd:int" use="required" />
  <xsd:attribute name="id" type="xsd:string" />
</xsd:complexType>
```

Thus, it must contain one and only one <portrait> element and one and only one <landscape> element. Moreover, two attributes are required:

minWidth

its value represents the minimum width, in pixels, for this interface to be matched and used as client interface;

minHeight

its value represents the minimum height, in pixels, for this interface to be matched and used as client interface.

Each XID interface element must contain, as already seen, two elements (<portrait> and <landscape>) that follow these specification:

```
<xsd:complexType name="portraitType">
  <xsd:sequence maxOccurs="unbounded" >
    <xsd:element name="element" type="elementType" />
  </xsd:sequence>
  <xsd:attribute name="layout" type="xsd:string" use="required" />
</xsd:complexType>

<xsd:complexType name="landscapeType">
  <xsd:sequence maxOccurs="unbounded" >
    <xsd:element name="element" type="elementType" />
  </xsd:sequence>
  <xsd:attribute name="layout" type="xsd:string" use="required" />
</xsd:complexType>
```

These elements required one attribute, *layout*, which can assume two values:

horizontal

the set of elements are distributed from left to right in the client GUI;

vertical

the set of elements are distributed from top to bottom in the client GUI.

Finally, each <element> inside <portrait> and <landscape> should have two attributes, that follow the following XML Schema:

```
<xsd:attribute name="width" type="xsd:string" />
<xsd:attribute name="height" type="xsd:string" />
```

They contain respectively width and height values, in percentage, to be applied to a specific element inside that interface.

We now discuss the process of assigning an interface to a client. When a client connects through Porthole, after WebSocket handshake, it sends its width, height and orientation. When orientation is not available or reliable (such as in some desktop browsers where only portrait is defined), the width and height are compared: if height is greater or equal to width, portrait mode is defined, otherwise, landscape mode is defined.

Giving the set of interfaces described as a pair of <minWidth,minHeight> Porthole assigns the nearest among the ones that satisfy these two conditions. Once an interface is selected, the orientation condition is checked, and, then, the final GUI, to be sent to the client, is created.

4.3 Porthole Server

Porthole Server is created by Porthole Service module and it's running on a separate thread. In this section the focus is on describing how the connections are managed, based on the two protocols supported: standard HTTP and HTML5 WebSocket.

In this section the focus is on the most relevant parts of the implementation flow, selected in order to explain the most important part of the code. The full code can be found at the official omegalib repository [35].

First of all, let's examine Porthole Server thread main flow, which can be represented by this pseudo-code:

```
procedure thread_code ( port number , protocols )
begin
   create server context
   set server port number
   set server protocols
   n := 0

   while n >= 0
      server loop ( loop interval in milliseconds)
      if server is closing then
         n := 0

   destroy server context
end
```

The server context is created based mainly on two parameters: the port number (which has been defined by application developer) and a set of protocols. If the context is created successfully, the server is now set up and in the main loop all the clients are managed. When the server receives a termination signal, n is negative, the loop ends and the context is destroyed.

The *protocols* object represents the set of supported protocols, i.e. HTTP and WebSocket, and it is defined as follows:

```
struct libwebsocket_protocols protocols[] = {
   { "http-only", ServerThread::callback_http, 0 },
   { "porthole_websocket", ServerThread::callback_websocket, sizeof(struct
       per_session_data) },
   { NULL, NULL, 0 }
};
```

Each protocol supported is defined by three variables:

- protocol name;
- protocol callback function, which manages each user connecting with that protocol;
- size of memory, which is allocated for each user connecting with that protocol.

While HTTP protocol does not have any persistent data across client HTTP requests, WebSocket protocol needs to allocate memory for each client session. This data is represented by the following struct:

```
struct per_session_data {
   PortholeGUI* guiManager;
   unsigned long long timestamp;
};
```

Session data consists of:

- PortholeGUI object, which is responsible for session data management, such as allocating Porthole camera object;
- timestamp of the last time server streamed session camera image, in order to manage streaming frequency.

Let's now analyze the two callback functions for managing HTTP and WebSocket connections. They have the same set of parameters, that are:

- server context;
- protocol manager instance;
- callback reason;
- user session data;
- input message content;

- input message length.

With these objects, all the bidirectional messages can be managed within a single callback function. The action triggered by a callback function is done based, first of all, on the reason value. We skip most of the side reasons management and focus on the most relevant, which are triggered when:

- a successfully handshake has been accomplished with a new client, defined by *LWS_ CALLBACK_ESTABLISHED*;
- the server output channel is free and could be locked for writes, defined by *LWS_ CALLBACK_SERVER_WRITEABLE*;
- a message is received from a client, defined by *LWS_ CALLBACK_SERVER_RECEIVE*;
- a connection is closed, defined by *LWS_ CALLBACK_CLOSED*.

Starting from HTTP callback function, the first and last events are not used, because there is no need to allocate and deallocate any omegalib data with HTTP sessions. The second is not used because the server does not initiate any HTTP connection by itself, but only responds to HTTP requests.

The *LWS_CALLBACK_SERVER_RECEIVE* type of HTTP callback is implemented as follows: it parses the requested Uniform Resource Identifier (URI), checks for availability of that URI, and, if the condition is fulfilled, the file is sent to the client. Moreover, a particular case is represented by the request to the *porthole_functions_binder.js* resource, in which case the server dynamically creates the response, which is not related by a particular file. The content is used for mapping JavaScript functions to server-side application functions: this topic is covered in details in section 4.6.

WebSocket callback function implements all the four callback reasons previously listed. Let's start from *LWS_CALLBACK_ESTABLISHED*: when the client upgrade the connection from

HTTP to WebSocket protocol, a new PortholeGUI object associated with the client is created, waiting, basically, for the device specification to be set, and the streaming timeout is set to 0. When the client disconnects, *LWS_CALLBACK_CLOSED* part of WebSocket callback function is executed, which deallocate these session specific objects.

WebSocket callback function with reason *LWS_CALLBACK _SERVER_WRITEABLE* is called when this session can access the output channel because it currently has a lock on it. It is used to write at about 50Hz images taken from each session camera and implements the following pseudo-code:

```
Case LWS_CALLBACK_SERVER_WRITEABLE
begin
   if camera is ready to stream then
      release channel lock
      return code: ok

   if ( current timestamp - last stream timestamp ) < 20 milliseconds then
      release channel lock
      return code: ok

   get session camera from omegalib
   lock camera data
   encode camera data into JPEG image
   unlock camera data
   encode image into base64 string
   create JSON event of type image stream
   allocate output buffer
   write JSON event to output buffer
   deallocate the buffer
   release channel lock
```

```
   if message is sent correctly then
      save current timestamp into session data
   else
      return code: not_sent

   return code: ok
end
```

Let's describe step by step these pseudo-code lines:

1. If this session has no camera associated (i.e. the interface for this client does not contain a camera), release the channel lock and return;
2. If the difference between the last stream timestamp and the current timestamp is less than 20 milliseconds, release the channel lock and return;
3. Get the camera object associated with this session;
4. Lock the bytes array of the session camera, which has been previously set with the width and height of the final image in order to avoid extra size;
5. Encode bytes array in Joint Photographic Experts Group (JPEG) format and obtain the image data;
6. Unlock the bytes array of the session camera;
7. Encode image object into a base64 string, so that it could be sent using WebSocket protocol;
8. A JSON message to be sent is created, which contains the camera id and the base64 encoded image;
9. Allocate the correct buffer to match WebSocket frame structure;

10. Write the message using the buffer created;
11. Deallocate the buffer;
12. Release channel lock so that it can be used for other clients;
13. If the message has been correctly sent, save current timestamp in session data and return successfully;
14. Otherwise return a "not sent" code.

Camera frames must be encoded in order to save output channel bandwidth, and JPEG encoding algorithms has been implemented inside Porthole. The JPEG encoding process has been built on top of FreeImage[45].

Once camera encoded image has been obtained as an array of bytes, base64 encoding is needed. This step is necessary for two reasons: the first reason is that image must be serializable as a string in order to be appended to a JSON event message (see section 4.5 for more details); the second reason is that <canvas> drawing function is faster when the image is set through a base64 image URI, instead of manually decoding and drawing the image data. In fact, HTML5 *drawImage* function is the browsers vendors suggested function to draw an image into the canvas, due to browsers optimization, and takes as parameter an *Image* object, whose source is set through a base64 image URI (many image encoding algorithms are supported, such as JPEG).

JSON event message sent to client using the WebSocket protocol has the following structure (where *base64Image* and *sessionCameraId* represent, respectively, the encoded image string and the session camera identifier, which is an integer casted to a string):

```
{
   "event" : "stream",
   "image" : base64Image
}
```

After escaping the double quotes, the message is ready to be sent. A buffer is allocated and initialized in order to match the WebSocket API message structure. Finally, the message is sent and, based on the returned value of this sub-procedure, the server knows if it has been successful or not.

Regarding the last WebSocket callback reason, *LWS_CALLBACK_SERVER_RECEIVE*, the pseudo-code for handling a message is:

```
case LWS_CALLBACK_SERVER_RECEIVE
begin
   if array of bytes received is a valid JSON object
      declare and initialize PortholeMessage
      call parse_JSON_message ( array of bytes )
      call handle_PortholeMessage ( PortholeMessage )
      return code: ok
   else
      return code: not_valid
end
```

First of all, a check must be done on the received array of bytes, which must represent a string formatted as a JSON object. If the check returns successfully, the PortholeMessage is created, filled and handled. A PortholeMessage object is a struct which contains all the fields that can be used later in the handling process, and it is declared and initialized on the stack, and passed by reference, in order to avoid memory fragmentation caused by continuously allocating and deallocating objects on the heap. The initialization is necessary because a message could contain some fields that other ones do not (in order to save bandwidth, message has only the needed fields).

The array of bytes received must be a JSON message and is parsed in order to fill the PortholeMessage object. The pseudo-code of this procedure is:

```
procedure parse_JSON_message ( input )
begin
   if input is array:
      for each element in array
         call parse_JSON_message ( element )
   else:
      assign PortholeMessage field with value
end
```

This procedure is, in fact, recursive, because, if the input is a pair of variable name and value, it is assigned to the PortholeMessage object; otherwise, we have an array of object, which could be a set of either pair objects (field and value) or JSON objects, and the procedure is recursively called for each one. JSON objects would eventually be split again, because recognized as an array itself, and the procedure would call itself again, and so on. We don't have problems about checking if an element does not follow JSON standards, because this check has already been done before starting the main procedure.

A PortholeMessage is handled by the following pseudo-code:

```
procedure handle_PortholeMessage ( PortholeMessage )
begin
   Switch PortholeMessage event:

   Case drag:
      get session camera from client related PortholeGUI
      transform yaw angle from degrees to radiants
      transform pitch angle from degrees to radiants
      create rotation Quaternion from yaw and pitch angles
      get current orientation as a Quaternion
      multiply current Quaternion by rotation Quaternion
```

```
      set the new rotation

   Case transform:
      if rotation
         get session camera from client related PortholeGUI
         transform roll angle from degrees to radiants
         create rotation Quaternion from roll angle
         get current orientation as a Quaternion
         multiply current Quaternion by rotation Quaternion
         set the new rotation
      else if zoom
         get session camera from client related PortholeGUI
         get camera position
         change camera position along z-axis
         set new camera position

   Case device_specification:
      set device specifications
      call create_GUI ( first_time )
      create JSON event of type GUI definition
      allocate output buffer
      write JSON event to output buffer
      deallocate the buffer
      set writable output

   Case javascript_event:
      get function key
      if C++ functions HashMap contains the key
         get C++ function by key
```

```
         call cpp function
      else if python codes HashMap contains the key
         get Python code by key
         send code to Python interpreter
end
```

Let's discuss each case, starting from dragging event. When a user touches with one finger the canvas on a web browser, a dragging event is sent, which contains yaw and pitch angles of the rotation (details on how to get those values can be found in section 4.5). Then, the angles need to be transformed into radiants and rotation Quaternion matrix is created starting from these values. The current orientation Quaternion is obtained and multiplied by rotation Quaternion. Finally, the result is set as the new orientation of the session camera. Of course, if the camera is the default camera, the result of this change is visible inside the VE.

Transformation, that is a two fingers interaction, can refer to either a rotation or a zooming event. The former is similar to a dragging event, but involves the roll angles, while the latter is related to changing the z-axis value. In fact, in zooming event, the session camera position is obtained, changed and set with the new value.

A complete different handling is done for a device specification event, whose message contains device width, height and orientation. Once received, the GUI is created based on these values. More details can be found in section 4.4. Once the GUI is created, a JSON object representing a GUI definition event is created and sent, as already seen, using WebSocket protocol. Finally, it starts watching for availability of channel for streaming purpose.

The last case involves the JavaScript events mapping, with which the events specified inside the XID file are executed server-side once they are fired client-side. More details on how this has been accomplished can be found in section 4.6.

4.4 Porthole GUI

In this section, the focus is on creation and modification of the GUI that is sent to the client. All the instances of PortholeGUI, one for each client, share the same data structure representing the XID file parsed. The XID interface layout used for a client is based, as already seen, on the closest successful matching with *minWidth*, *minHeight* and orientation values.

The interface creation process depends on a boolean variable, which is used to distinguish if the device specification for a client has been received for the first time or not (for example, after changing device orientation). This is useful because a new camera object is not allocated each time a device changes its specification, in which case only a modification of camera properties is needed. The pseudo-code of this process is the following:

```
procedure create_GUI ( first_time )
begin
   declare and initialize GUI string
   GUI += <table at 100% width and 100% height >

   if layout is horizontal
      GUI += <tr at 100% width and 100% height >

   for each element in disposition:
      parse attributes and save element_width and element_height values
      Switch element type:
      Case HTML:
         if layout is horizontal
            GUI += <td at element_width and element_height > element content <\td>
         else if layout is vertical
            GUI += <tr at 100% width and 100% height >
                <td at element_width and element_height > element content <\td>
```

```
                <\tr>
      Case Canvas:
         if first_time
            create session camera (element_width and element_height)
         else
            modify session camera (element_width and element_height)
         if layout is horizontal
            GUI += <td at element_width and element_height >
               <canvas id="camera-canvas" at element_width and element_height
                   /><\td>
         else if layout is vertical
            GUI += <tr at 100% width and 100% height >
                <td element_width and element_height >
                <canvas id="camera-canvas" at element_width and element_height />
                <\td><\tr>
   if layout is horizontal
      GUI += <\tr>

   GUI += </table>
end
```

The focus is to: obtain a HTML structure that would hold GUI components, send this string obtained using WebSocket and append it to the body of the browser HTML page. In order to obtain the HTML structure, the algorithm starts with opening a table tag. Then, if layout is horizontal, we dispose the elements from left to right, as table datas (td) inside a single table row (tr), while if layout is vertical, we dispose the elements from top to bottom, as one single table data (td) for each table row element (tr). The algorithm distinguishes between HTML and Canvas elements: in the first case, the content is just used ”as is”; in the second case, a <canvas> element with a specific id is created and the session camera related to it is initialized

(if *first_time* is true) or modified (otherwise). The id attribute is used inside client-side code in order to get the DOM element of the canvas.

4.5 Porthole Client

In this section, the focus is on the client-side code implementation. The main part of the code is inside an *index.html* file, which is responsible for managing connections, GUI creation and interactions.

When a client connects to the VE HTTP address, which refers to VE master node machine, HTTP callback described in section 4.3 is used to send *index.html* to the client. The WebSocket connection creation and management follow this pseudo-code:

```
declare socket


procedure create_WebSocket_connection
begin
   get current url
   initialize socket
end


procedure socket.onopen
begin
   call send_specification
end


procedure socket.onmessage (message)
begin
   declare and initialize a JSON object
   JSON object := convert message.data into a JSON object
   if event is a GUI definition then
```

```
      set body inner HTML
      set page height as current viewport height
      apply jQuery Mobile style
      search for DOM element with id camera-canvas
      if canvas found
         get camera-canvas context
         attach Hammer object to canvas element
      start looping at 50 Hz
   else if event is a camera image then
      set current canvas image as received image
end

procedure socket.onclose
begin
   if looping is true then
      end looping
end

procedure send_specification
begin
   get viewport width and height
   get device orientation
   create JSON event with width, height, orientation
   send JSON event
end
```

The WebSocket object is created by inspecting the current Uniform Resource Locator (URL), replacing the address protocol from *http://* to *ws://*. Once the WebSocket connection is established, the *socket.onopen* callback function is called, which, in turn, calls *send_specification*.

This function creates a JSON object representing device specification and sends it using the opened socket.

Once a message from the server is received, the *socket.onmessage* callback function is called: string received is parsed as a JSON object and the event type field is evaluated. If the event is a GUI specification, the inner HTML page body is set as specified in the received message. Then, the body height is set equal to the current viewport height, in order to avoid jQuery Mobile auto-resizing, and jQuery Mobile style is applied. After that, the body is inspected for any canvas element whose id is *camera-canvas*; if such element is found, the canvas element context is retrieved to be used inside the main loop (which is described below), and *Hammer.js* is used to initialize canvas mouse and touch events. Finally, main loop is started using JavaScript *setInterval* function. If the event received is a camera image, the current canvas source is changed with the new base64 encoded image received.

Before describing the main loop, let's analyze the Hammer callback functions, whose pseudo-code are:

```
procedure hammer.ondragstart (event)
begin
   dragging := true
   drag_start_x := event x value
   drag_start_y := event y value
end

procedure hammer.ondrag (event)
begin
   drag_end_x := event x value
   drag_end_y := event y value
end
```

```
procedure hammer.ondragend (event)
begin
   dragging := false
end

procedure hammer.ontransformstart(event)
begin
   transforming := true
   delta_start_rotation := 0
   scaling_factor := 1
end

procedure hammer.ontransform(event)
begin
   scaling_factor *= event scale factor
   delta_rotation += ( event rotation value - delta_start_rotation)
   delta_start_rotation = event rotation value
end

procedure hammer.ontransformend(event)
begin
   transforming := false
end
```

These callback functions are used to set fundamental variables that are used in the main loop function in order to send interaction events to the server. Once dragging starts, a flag is set to true. This signals a dragging event to be sent, and touch (or mouse) x and y values are stored; while dragging, the final x and y values are saved; once the dragging ends the flag is set to false. Once transformation starts (i.e. two fingers touch the canvas), the transformation flag

is set to true, in order to start sending transformation events to the server in the main loop, rotation delta is set to zero and scaling factor is set to one (i.e. no scaling). While transforming, the scaling factor is multiplied by the new value: if new value is between zero and one, the fingers distance is decreased; if it is greater than one, the distance is increased. Delta rotation, instead, is summed to the difference between new delta value (which is positive if the movement is clock-wise and negative if it is counter clock-wise) and its old value. Finally, the old delta value is updated with current value. Once the transformation ends, transforming flag is set to false.

The main loop implements the following pseudo-code:

```
procedure loop
begin
   if looping is true then

      if dragging is true and caves exists then
         if (drag_end_x - drag_start_x) is not 0 or (drag_end_y - drag_start_y) is
             not 0 then
            yaw := ( (drag_end_x - drag_start_x) / canvas width ) * 90
            pitch := ( (drag_end_y - drag_start_y) / canvas height ) * 90
            create JSON event object
            send JSON event to server
            drag_end_x := drag_start_x
            drag_end_y := drag_start_y

      else if transforming is true and caves exists then
         if scaling_factor is not 1 or delta_rotation is not 0 then
            round scale between 0.1 and 10
            create JSON event object
            send JSON event to server
```

```
            scaling_factor := 1
            delta_rotation := 0

      if canvas exists then
         draw image inside canvas

      if ( mouse button pressed or screen touched ) && send_continuous is true
         get event from target
         create JSON event object
         send JSON event to server
end
```

This procedure is called at 50Hz and is done if looping flag has been set to true. It can be divided into four main sections, which are:

- if the user is dragging, the canvas exists and the delta values are not both equal to zero, then yaw and pitch values are obtained by scaling the number of pixels "dragged" by canvas dimensions and a 90 degrees angle. Once the yaw and pitch angles are obtained, they are enveloped into a JSON event object and sent to the server;
- if the user is touching the screen with two fingers, the canvas exists and rotation and scaling are not at their default values, then the scaling factor is rounded between 0.1 and 10, so that it cannot be less or equal to zero nor it is too much high, i.e. greater or equal to 10. Then, these values are enveloped into a JSON event object and sent to the server;
- if GUI contains a canvas, the image received by the server is drawn inside the canvas with JavaScript *drawImage* function, described in section 3.3.3.
- if mouse is pressed or the screen is touched, and *send_continuous* is set to true (see section 4.6 for more details), then, the target element state is obtained, serialized and sent to the server as a JSON event.

Another important piece of code used to manage client GUI is about browser window resizing events and orientation change events. These events both trigger the same timeout function, whose pseudo-code is:

```
procedure send_specification_timeout
begin
   if not waiting
      waiting := true
      wait 1 seconds
      call send_specification
      waiting := false
end
```

This procedure avoids sending multiple times the GUI recreation event for each small change while resizing the windows. It enqueues one request and rejects any other call to that function before one second (this value has been chosen empirically and may change in future work). Once device specification event has been sent to the server, the function can be called again from JavaScript resizing and orientation changing handlers. Finally, notice that "waiting" is not blocking, since JavaScript *setTimeout* function has been used.

The last piece of code that is important to describe is related to the fourth *if* condition, which is based on two control values: the first is a signal that a mouse button is pressed or screen is touched; the second is a signal that the GUI element targeted by this action wants to stream its value continuously instead of one time only. Both are implemented by the following jQuery code, which may explain the algorithm better than a pseudo-code:

```
if (isTouchable) {
   $('body').bind('touchstart', function (event) {
      mouseDown = 1;
      $(event.target).trigger('mousedown');
   });
```

```
    $('body').bind('touchend', function () {
        mouseDown = 0;
        send_continuous = false;
    });
}
else {
    $('body').bind('mousedown', function (event) {
        mouseDown = 1;
    });
    $('body').bind('mouseup', function () {
        mouseDown = 0;
        send_continuous = false;
    });
}
```

jQuery is needed because it can programmatically bind and trigger user events. In particular, this code unifies the behaviors of *mousedown* and *touchstart* events: on iOS devices, *mousedown* is triggered only *after* the finger is up[46], and inline *touchstart* events defined in XID file are not triggered. Thus, we need to manually bind *touchstart* events and link them to XID inline *mousedown* events, such that VE applications developers can simply define *mousedown* events and use them with both touchscreen devices and desktop or laptop computers.

Moreover, this jQuery code manages *mouseDown* variable, which is used inside the main loop shown above, and *send_continuous* variable, which is set to false in order to stop any value streaming to the server, because interaction ends.

4.6 Porthole Events

In this section, the focus is on creating, binding and triggering XID defined events. This section has been put as the last one because it involves XID parsing, server callbacks and client-side code.

First of all, events can be used inside any XID *element* whose type is HTML . The events allowed are all the JavaScript ones, which are the followings, defined inside a static array of strings used in XID parsing:

```
// All HTML compatible events that could be found parsing xid file
static const string events[eventsNumber] = {
   "onload", /* Script to be run when a document load */
   "onunload", /* Script to be run when a document unload */
   "onblur", /* Script to be run when an element loses focus */
   "onchange", /* Script to be run when an element changes */
   "onfocus", /* Script to be run when an element gets focus */
   "onreset", /* Script to be run when a form is reset */
   "onselect", /* Script to be run when a document load */
   "onsubmit", /* Script to be run when a form is submitted */
   "onabort", /* Script to be run when loading of an image is interrupted */
   "onkeydown", /* Script to be run when a key is pressed */
   "onkeypress", /* Script to be run when a key is pressed and released */
   "onkeyup", /* Script to be run when a key is released */
   "onclick", /* Script to be run on a mouse click */
   "ondblclick", /* Script to be run on a mouse double-click */
   "onmousedown", /* Script to be run when mouse button is pressed */
   "onmousemove", /* Script to be run when mouse pointer moves */
   "onmouseout", /* Script to be run when mouse pointer moves out of an element */
   "onmouseover", /* Script to be run when mouse pointer moves over an element */
   "onmouseup", /* Script to be run when mouse button is released */
};
```

The XID file is, then, parsed, looking for any of these entries. The goal, in this case, is to store inline events handlers. This procedure is done once Porthole Service is called by the VE application, and the pseudo-code implemented is:

```
procedure search_xid_node ( node )
begin
   if node is null then
      return
   attributes := node attributes

   for each attribute in attributes
   name := attribute name
   value := attribute value
   if name is a JavaScript event then
      if value is a C++ function then
         add to cppFunctionsMap <name, c++ function address>
      else
         add to pythonFunctionsMap <"python_function" + (N++) + "(event)", value >

   for each child of node
      call search_xid_node ( child )
end

procedure start_parser
begin
   parser_root := get xid elements node
   call search_xid_node ( parser_root )
end
```

The second procedure, called by Porthole Service, starts parsing the <elements> XML node and searches for JavaScript events. The first procedure is used recursively for this purpose: for each node, it parses all the defined attributes, searching for a matching with a JavaScript event. Once found, if the value has been defined as a key of the PortholeFunctionsBinder object (i.e. a bind between a string and a C++ function address, described in section 4.1), that pair is added

to the *cppFunctionsMap* HashMap; otherwise, the code snippet is considered as a Python script, and a new function name is created (which is used eventually as a JavaScript function name). This name is used as a key for *pythonFunctionsMap* HashMap, whose values are the real Python code snippets.

For example, let's say a developer needs to specify a button with a jQuery Mobile "up" arrow icon, that changes the camera position upward along y-axis and streams its value continuously while pressed or touched. Using Python code injection, the following code can be put inside a XID HTML element:

```
<input type="button" data-role="button" onmousedown="if(isMaster()):
    camera=getCameraById(%id%); position = camera.getPosition(); position[1] +=
    0.025; camera.setPosition(position);" data-iconpos="notext" data-icon="arrow-u"
    data-continuous="true" class="ui-btn ui-btn-icon-notext"/>
```

Using C++ code binding, instead, the same can be accomplished by:

```
<input type="button" data-role="button" onmousedown="up(event)" data-iconpos="notext"
    data-icon="arrow-u" data-continuous="true" class="ui-btn ui-btn-icon-notext"/>
```

In this case the developer have to add, inside the application, the function associated with *up(event)* function and add this binding to the PortholeFunctionsBinder object. The C++ function would be like:

```
  void up(PortholeEvent &event){
     if(SystemManager::instance()->isMaster()){
        Vector3f myPosition = event.sessionCamera->camera->getPosition();
        myPosition[1] += 0.025f;
        event.sessionCamera->camera->setPosition(myPosition);
     }
  }
```

The result is the same: the master node changes the position of the camera along the y-axis. Eventually, omegalib updates cluster nodes state accordingly, as shown in section 4.1.

The attribute *data-continuous* can be true or false, depending if the developer wants the object to execute the action continuously or one time only.

Regarding event variables, in case of Python code injection, the tokens that can be used are:

%id%

the session camera id;

%value%

the field value, such as the state of an input field;

%key%

the keyboard char which triggered the event;

%btn%

the number of the mouse button which triggered the event;

%event%

the name of the triggered JavaScript event.

The PortholeEvent object used for C++ functions binding, instead, is defined as follows:

```
typedef struct PortholeEvent{
   std::string event;
   int mouseButton;
   char key;
   std::string value;
```

```
   PortholeCamera* sessionCamera;
}PortholeEvent;
```

The event objects that can be obtained are, thus, basically the same.

Let's now describe how the client can dynamically obtain the client-side JavaScript functions, which can trigger server-side code execution once fired. When the client receives the HTML page, it contains an external script file reference, which is obtained by sending the request to the URL specified, that is *./porthole_functions_binder.js*. This file does not exists, but is dynamically created upon request. Its creation is managed inside the HTTP callback function, described in section 4.3, and implements the following pseudo-code:

```
Case LWS_CALLBACK_HTTP:
begin
   if requested file is ./porthole_functions_binder.js then
      declare and initialize content_to_send with HTTP header

      for each key in cppFuncionsMap and pythonFunctionsMap
         content_to_send += " function " + key +
               " { JSONToSend = {
               event_type : input,
               button : event.button,
               char : getChar(event),
               value : event.target.value,
               function : " + key + " };
               send_continuous = event.target.getAttribute(\"data-continuous\");
               socket.send(JSON.stringify(JSONToSend));
               };"

      update content_to_send header with content length
```

```
      send content_to_send to client
   else
   manage other requests ...
end
```

For each event to be triggered server-side, a JavaScript function is sent to the client. Thus, JavaScript now "understands" HTML inline functions, such as *up(event)* or *python_function0(event)* that would have been created for the example shown above. Once triggered, such function creates the event to be sent, serializes it and sends it using the opened WebSocket connection.

Finally, if VE developer defines and sets *data-continuous* attribute to true for one HTML element, the control variable *send_continuous* is set to true. This variable is used in client main loop for streaming that element events, and eventually set to false once the user stops touching with the finger or pressing with the mouse, as described in section 4.5. In such case, the binded C++ function or the injected Python code snippet is executed at about 50Hz, the same frequency of client main loop, in order to ensure consistency with canvas context update frequency.

4.7 Porthole in action

In this section, the implementation results are presented. In particular, two scenarios are analyzed: the first one is about one person interacting with a tablet inside the Virtual Environment; the second one is about collaboration between two users inside it. The VE used is CAVE2™.

The first scenario is depicted in figures 19 and 20. The user is interacting with CAVE2™ using a tablet.

The user receives a real-time streaming and can control a camera inside the virtual world directly from his tablet. Moreover, the user can control the visualization options with a set of

buttons in order to enable and disable each model. The result is that the user can feel less lost inside the 3D world because he has an overview of it inside his tablet.

Figure 19: User interaction inside CAVE2™ running Molecule application.

The second scenario is depicted in figure 21. Two users collaborates inside CAVE2™: one of them is navigating inside the virtual world using a 3D controller, while the other is receiving a streaming from a decoupled camera and manipulating the application options directly from a browser. The two users can collaborate using vocal communication in order to better explore the visualization.

Figure 20: User interaction inside CAVE2™ running Brain 2 application.

Figure 21: Collaboration inside CAVE2™ running Baybridge application.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we provide experimental results about Porthole. The goal is to perform local and remote experiments to evaluate *system performance*, in order to point out how much time and resources are necessary for Porthole. Thus, the experiments focused on three functionalities:

- user interactions, measuring the delay between a user interaction and the visible result of that interaction;
- session camera images streaming, measuring the time spent on each server-side operation related to the creation of a camera image message;
- omegalib Frames Per Second (FPS), comparing its value before and after a client is connected.

These experiments are considered necessary for testing the performance of the system, in order to understand what are the bottlenecks of it, leaving more detailed multi-user testing with human resources for future works. In fact, the proposed evaluation is considered about performances is considered as a prerequisite for usability.

This chapter is divided as follows: section 5.1 presents the design of the experiments, describing environment, devices and applications used; then, in section 5.2, an analysis of connection routing is provided, in order to understand how remote connection has been performed and how this can impact on experiments; finally, in section 5.3 the experiments done are described and the relative experimental statistical results are provided and evaluated.

5.1 Design of experiments

The first focus of the experiments is to understand how the stream of camera images, to be drawn inside the canvas element, impacts on performance, because this feature is necessary in order to use Porthole as a decoupled remote VE interface. Thus, we have measured the time spent on the creation of a camera image and the final size of that type of message. This has been done by using timestamps before and after each one of the main parts of the server-side algorithm, that are: JPEG encoding, base64 encoding and WebSocket buffer writing. Then, the final stream message size has been evaluated. In particular, each experiment involves the use of *ifdef* C++ code blocks, in order to evaluate each step without impact on the others.

Another goal of the experiments is to measure the time between a GUI interaction, that modifies the application, and the receiving of the actual changed image. This is considered as Porthole Round Trip Time (RTT). In order to have this value, a timestamp is taken as soon as the user interacts with a GUI button; then, when the interaction message is received and managed by the server, the next image sent to the client has a flag set, which is used to take the second timestamp. Finally, the difference between the two timestamps, that is our RTT, is printed to the browser console. As before, this experiment has been done by using *ifdef* C++ code blocks, so that these C++ blocks are compiled and run only during this experiment, without any impact on normal execution neither on other experiments.

Finally, a more qualitative analysis about virtual environment FPS has been done, in order to have an overview of how a client connection impacts on VE system performance. For this experiment, the omegalib performance monitor has been used, which calculates the average FPS at runtime.

Regarding experiments about times and dimensions, we have run each test for about one minute each, interacting with the application as similar as possible. The data collected are analyzed using box plots, created with Microsoft Excel. This kind of statistical visualization

is particular useful in our case, because the same data, such as a timestamps difference, is collected for each iteration. Moreover, each value can be different from the previous because of a scene change or a different network connection delay. Thus, only mean or median value can be insufficient to correctly study the sampled data.

In order to evaluate the data collected and to plot the relative box plot, the following statistics are calculated:

Mean

The sample mean value;

SD

The Standard Deviation (SD) value;

Min

The sample minimum value;

Q_1

The first quartile, or lower quartile, equals to the 25th percentile, that splits lowest 25% of data;

Median

The median, or second quartile, equals to the 50th percentile, that cuts data set in half;

Q_3

The third quartile, or upper quartile, equals to the 75th percentile, that splits highest 25% of data;

Max

The sample maximum value;

IQR

The InterQuartile Range (IQR), that is the difference between upper and lower quartiles;

Lower Whisker

The maximum value between the sample minimum value and Q_3 plus 1.5 times the interquartile range: $MAX\{Min, Q_3 + 1.5 * IQR\}$;

Upper Whisker

The minimum value between the sample maximum value and Q_1 minus 1.5 times the interquartile range: $MIN\{Max, Q_1 - 1.5 * IQR\}$;

When time is considered, the unit of measurement is millisecond (ms), while when message dimension is considered, it's kilobyte (kb).

The statistical results mainly depend on two factors:

- image dimensions;
- scene complexity.

Thus, we have used a set of five different dimensions and two different applications.

The devices used for evaluating the impact of different dimensions are:

Smartphone (SP)

Screen: 480x800 pixels;

Tablet (TL)

Screen: 800x1280 pixels;

Laptop HD

Screen: 1920x1080 pixels.

In fact, we have used a remote Android smartphone (HTC HD2) using Firefox, two general laptops with full HD resolution using Chrome (one local and one remote, in order to test the

different RTTs), and a local Android tablet (Nexus 7) using Chrome. Thus, we ensures a good mix of both local and remote interactions.

Regarding scene complexity, which affects JPEG encoded image size, two different applications have been chosen:

Porthello

An OpenGL simple application that displays a 3D cube with colored faces, put inside a plain gray background;

Molecule

An OSG complex application that displays a group of molecules interacting inside a 3D world.

The virtual environment used has been the CAVE2™, at the UIC.

The XID file used to generate the decoupled HTML5 interfaces is provided in appendix B, with images of generated client interfaces. In particular, each different interface contains a canvas element used for displaying the images received. The server-side camera image dimensions thus reflect these canvas elements dimensions. With respect to each device and each orientation available, they set the following number of pixels:

Smartphone (SP) Portrait

Canvas: 320x412 pixels; Number of pixels: *131,840*

Smartphone (SP) Landscape

Canvas: 532x252 pixels; Number of pixels: *134,064*

Tablet (TL) Portrait

Canvas: 600x712 pixels; Number of pixels: *427,200*

Tablet (TL) Landscape

Canvas: 960x440 pixels; Number of pixels: *422,400*

Laptop HD

Canvas: 1728x968 pixels; Number of pixels: *1,672,704*

Thus, during results evaluation in section 5.3, they are put inside tables and plots in increasing order of number of pixels to look for significant performance informations related to this value.

Finally, in the next section, a traceroute analysis is provided, in order to have more details on remote connection routing. This, in fact, affects the RTT analysis presented, later, in section 5.3.1.

5.2 Traceroute analysis

In this section we provide remote connection routing analysis. In fact, the number of hops and the quality of connection, from remote client to CAVE2™, affect Porthole RTT values evaluation.

Figure 22: Traceroute visualization

Figure 23: Traceroute hops delays (in milliseconds)

For the experiments, we have established a remote connection from Milan area to UIC network. VisualRoute[47], a freeware visualization tool for traceroutes, has been used to have more details about remote connection. A map of this connection is shown in figure 22. Figure 23, instead, shows time, in milliseconds, necessary for reaching each route hop.

TABLE I: TRACEROUTE STATISTICS

Destination Host	lyra.evl.uic.edu
Number of hops	21
RTT Mean	121.2 ms
RTT Max	398 ms
Packet Loss	1.5%

Finally, in table I some statistical values are presented. In particular, we can see RTT differences between the mean value and maximum value, mainly because of Chicago-to-Washington hop delay, as shown in figure 23.

5.3 Experiments and results evaluation

In this section, the experiments and the related data collected are evaluated, in oder to look for significant informations related, mainly, to camera dimensions (i.e. number of pixels) or scene complexity, as described in section 5.1.

5.3.1 Round Trip Time (RTT)

The first evaluation is about Round Trip Time (RTT). When the user clicks one of the interface button, a timestamp is saved. Then, the event message is crafted and sent to the server

using the opened WebSocket. Once it is received, the corresponding Python codes snippet or C++ function is executed and, once finished, a flag is set to *true*. The first image message to be sent to the client contains this flag set, so that client, once received, saves another timestamp. Finally, RTT is obtained as the difference between the second timestamp and the first timestamp saved, and the value is printed in Chrome browser console.

TABLE II: LOCAL AND REMOTE ROUND TRIP TIMES (IN MILLISECONDS)

	Local	Remote
Mean	78.1	356.15
SD	44.91	216.95
Min	9	132
Lower Whisker	9	132
Q_1	45	189
Median	64	261
Q_3	129	610
Upper Whisker	159	761
Max	159	761
IQR	84	421

This experiment evaluates the different delays between local and remote client, representing the time necessary to receive a feedback from an interaction. The experiment has been done by using two laptops with the same screen size (and, thus, canvas size): one locally connected to CAVE2^{TM} through the EVL internal Wi-Fi network; one, instead, remotely connected from Milan area.

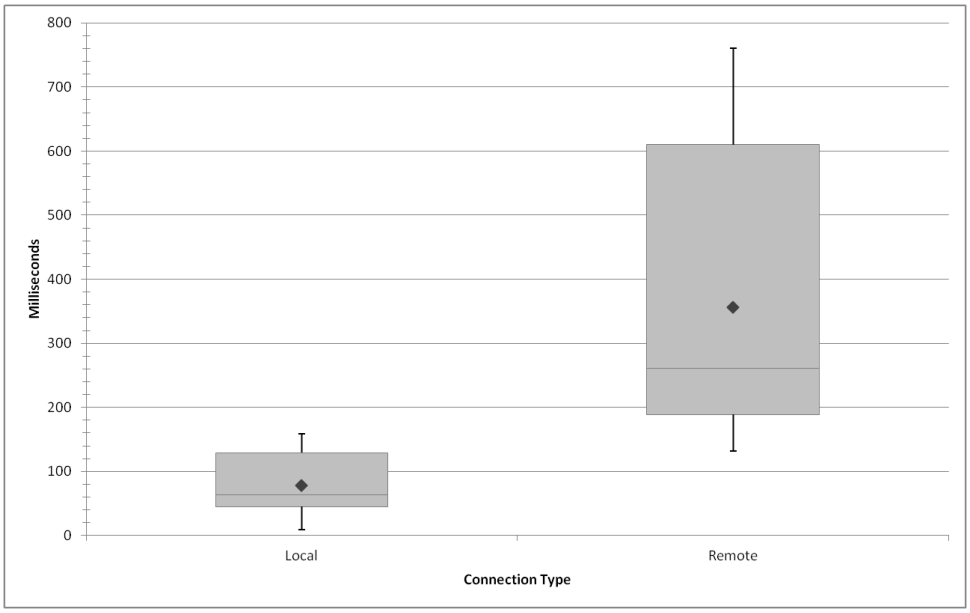

Figure 24: Local and remote round trip times (in milliseconds)

The statistical results are shown in table II (where values are in milliseconds) and plotted in figure 24. The result is clear: the network delays impact on remote interactions. In fact, the standard deviation is much higher in remote communications: this is related to the traceroute analysis done in section 5.2, where we have shown that the average delay is about 120 ms, but can be up to 400 ms.

Local RTT mean value is about 78 ms, which is an acceptable delay, while remote RTT mean value is about 356 ms, which is, instead, noticeable in real time applications, requiring a small interval between an action and another, but we still find the application usable.

5.3.2 Camera image message size

In this section we evaluate the experimental results about camera image message size. Porthole Server, once it is time to send a new image (because 20 ms interval has finished), it encodes camera image into JPEG format, then encodes it into a base64 image and creates a message of the following type (as seen in section 4.3):

```
{
   "event" : "stream",
   "image" : base64Image
}
```

The data have been collected by writing to a file the dimension of each message sent to a client interacting with the generated HTML5 interface. Five types of clients, based on different dimensions and orientation, and two different applications, Porthello and Molecule, are used, as described in section 5.1. The statistical results for Porthello are shown in table III, the ones for Molecule are shown in table IV (units of measurement is bytes). The corresponding box plots are shown in figure 25 and 26.

The results point out that the complexity of the scene and the technology used for rendering (OpenGL vs OSG) greatly affect the final base64 encoded image size and the corresponding message size. The increase ranges from about 162% for Laptop HD to at most about 674% for Tablet Portrait. Thus, the message size for a "complex" application can be 8 times the size for a "simple" one. More studies need to be done to understand if it is possible to reduce this big difference.

5.3.3 JPEG encoding

In this section, the focus is on evaluating the encoding time necessary to create the bytes array that represents a JPEG image from the bytes array that contains the rendered camera

TABLE III: CAMERA IMAGE MESSAGE SIZE FOR PORTHELLO (IN KILOBYTES)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	5194	6250.25	14352.63	12630.75	42787.39
SD	435.07	742.86	1419.93	1006.8	2771.15
Min	3666	3794	9850	10010	36026
Lower Whisker	3829	5416	10876	10010	36026
Q_1	4840	6214	13522	11560	41550
Median	5370	6494	15054	12862	42026
Q_3	5514	6746	15286	13486	45342
Upper Whisker	6022	7286	15994	14290	47686
Max	6022	7286	15994	14290	47686
IQR	674	532	1764	1926	3792

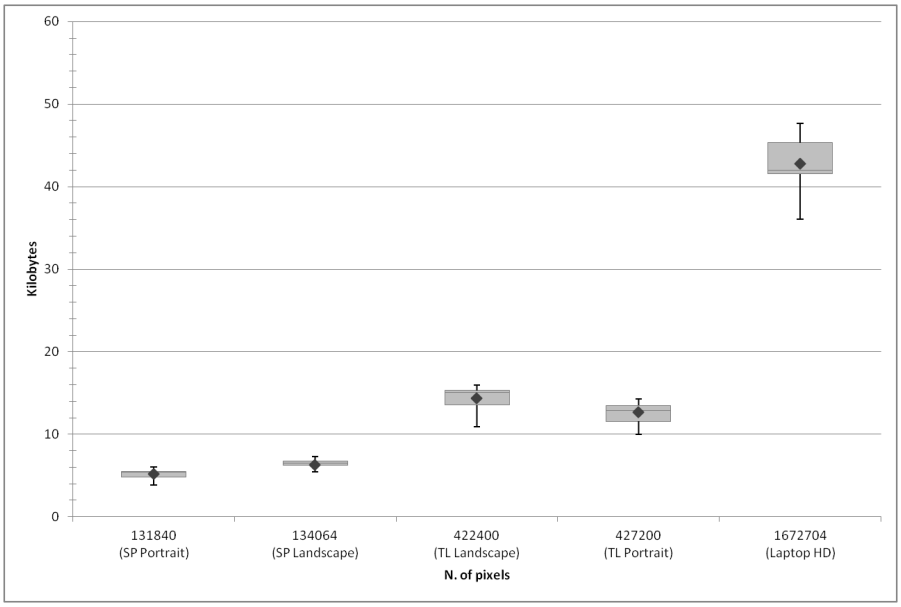

Figure 25: Camera image message size for Porthello application (in kilobytes)

TABLE IV: CAMERA IMAGE MESSAGE SIZE FOR MOLECULE (IN KILOBYTES)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	33729.46	38649.83	101345.23	97756.23	330560.49
SD	2250.73	11082.14	31810.20	8274.87	112223.05
Min	5374	5862	15546	11590	14218
Lower Whisker	26230	5862	15546	76954	14218
Q_1	31978	21882	73606	92830	202110
Median	34450	44118	105438	100414	375774
Q_3	35810	46678	135442	103414	435068
Upper Whisker	36586	51430	142522	106758	442478
Max	36586	51430	142522	106758	442478
IQR	3832	24796	61836	10584	232958

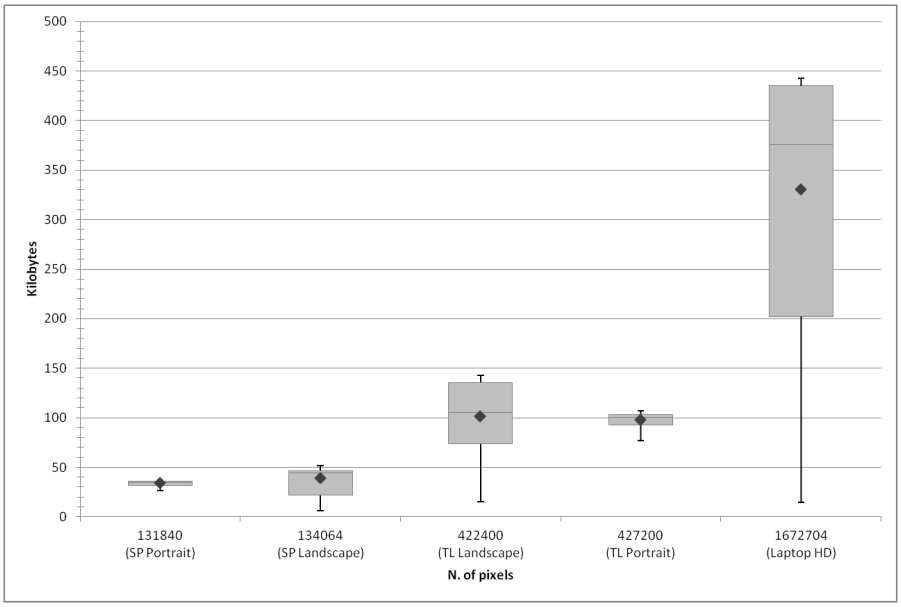

Figure 26: Camera image message size for Molecule application (in kilobytes)

view. The experiment has been done by inserting a timestamp after the session camera pointer is obtained; then, the camera data is locked, to avoid concurrency access between Porthole Server and omegalib renderer, encoded into JPEG format, unlocked and, finally, the second timestamp is taken. The difference between the two timestamps is, then, written to a file.

The statistical results for Porthello JPEG encoding time are shown in table V, the ones for Molecule are shown in table VI (units of measurement is milliseconds). The corresponding box plots are shown in figure 27 and 28.

The results point out that the number of pixels is what most influence the JPEG encoding time. In fact, Laptop HD image encoding needs about 4 times SP Portrait encoding. However, there is also a difference between Porthello and Molecule JPEG encoding times: the different dimensions pointed out in section 5.3.2 are linked with these values. Molecule encoding mean time is from about 1.1 to about 2 times higher than Porthello encoding mean time.

TABLE V: TIME SPENT ON JPEG ENCODING IN PORTHELLO (IN MILLISECONDS)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	11.11	11.25	13.93	14.19	47.67
SD	6.24	6.07	5.41	5.82	1.12
Min	2	2	6	6	36
Lower Whisker	2	2	6	6	45.5
Q_1	7	7	10	10	47
Median	11	11	13	13	47
Q_3	15	15	17	17	48
Upper Whisker	27	27	27.5	27.5	49.5
Max	85	74	57	57	51
IQR	8	8	7	7	1

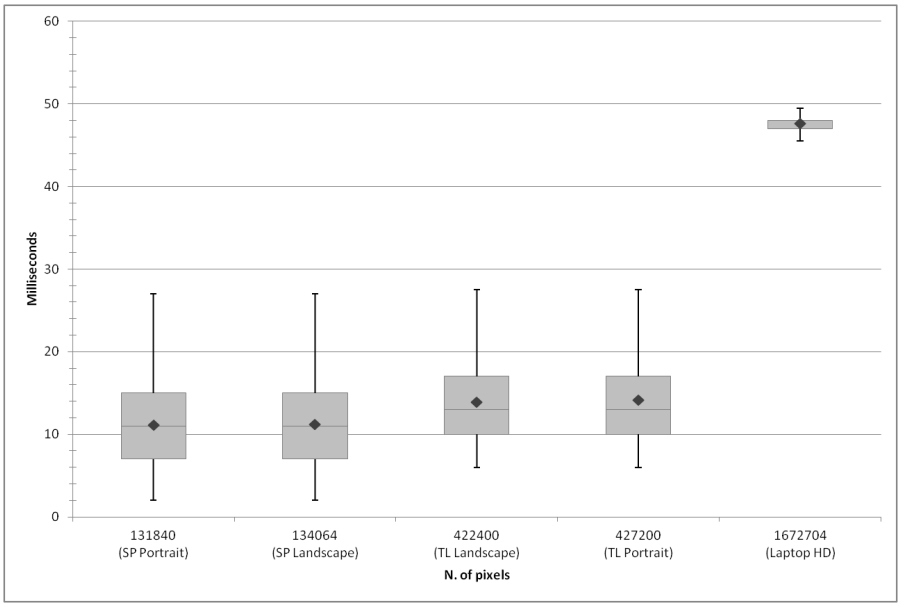

Figure 27: Time spent on JPEG encoding in Porthello (in milliseconds)

5.3.4 Base64 encoding

In this section, statistical data about the time spent on base64 encoding are provided. In fact, after getting the JPEG encoded image as an array of bytes, Porthole Server creates a base64 encoded string that represents the image. This encoding time is obtained by the difference between a timestamp before and one after the encoding process.

The statistical results for Porthello JPEG encoding time are shown in table VII, the ones for Molecule are shown in table VIII (units of measurement is milliseconds). The corresponding box plots are shown in figure 29 and 30.

The statistical results point out that base64 encoding time is often less than 1 ms. Thus, this procedure is fast with respect to the other parts of streaming process, taking at most 4

TABLE VI: TIME SPENT ON JPEG ENCODING IN MOLECULE (IN MILLISECONDS)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	12.66	12.77	29	29.74	68.94
SD	4.97	5.24	8.28	8.1	16.98
Min	2	2	9	8	21
Lower Whisker	2	2	10.5	10.5	25
Q_1	8	8	24	24	59.5
Median	13	13	28	31	67
Q_3	17	17	33	33	82.5
Upper Whisker	29	30.5	46.5	46.5	106
Max	29	41	65	50	106
IQR	9	9	9	9	23

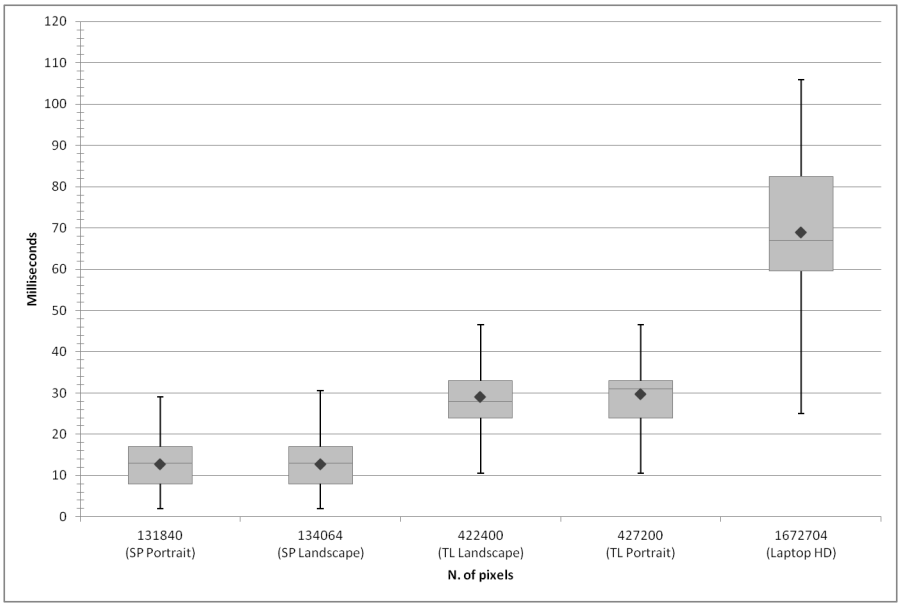

Figure 28: Time spent on JPEG encoding in Molecule (in milliseconds)

ms with Laptop HD and Molecule application. Base64 encoding depends only on bytes array length, so it depends on JPEG encoded image size. In fact, the mean value of the time spent on base64 encoding in Molecule can take up to about 11 times the one in Porthello, in case of SP Landscape client.

TABLE VII: TIME SPENT ON BASE64 ENCODING IN PORTHELLO (IN MILLISECONDS)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	0.03	0.02	0.06	0.05	0.21
SD	0.16	0.14	0.23	0.22	0.41
Min	0	0	0	0	0
Lower Whisker	0	0	0	0	0
Q_1	0	0	0	0	0
Median	0	0	0	0	0
Q_3	0	0	0	0	0
Upper Whisker	0	0	0	0	0
Max	1	1	1	1	1
IQR	0	0	0	0	0

5.3.5 WebSocket buffer writing

In this section, the experimental results about the time spent on writing the WebSocket buffer, in order to send the stream message that contains a base64 encoded image, are provided and evaluated. In fact, once such a message is created, Porthole Server writes it on the WebSocket output buffer, which will be eventually send through the network. Before and after that operation, two timestamps are taken in order to evaluate the time spent on it.

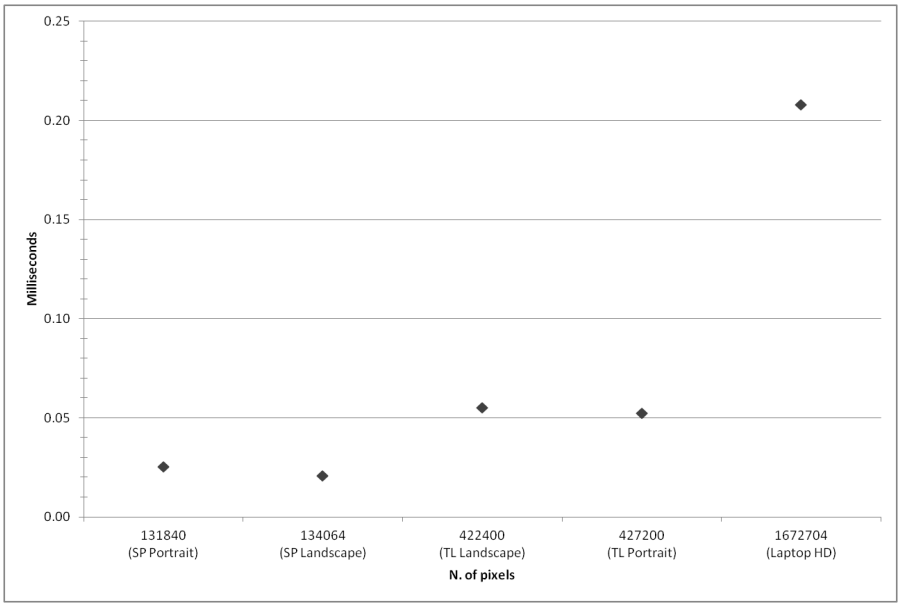

Figure 29: Time spent on base64 encoding in Porthello (in milliseconds)

TABLE VIII: TIME SPENT ON BASE64 ENCODING IN MOLECULE (IN MILLISECONDS)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	0.19	0.22	0.58	0.5	1.76
SD	0.39	0.42	0.5	0.5	0.69
Min	0	0	0	0	0
Lower Whisker	0	0	0	0	0
Q_1	0	0	0	0	1
Median	0	0	1	0	2
Q_3	0	0	1	1	2
Upper Whisker	0	0	2	1	3.5
Max	1	1	2	1	4
IQR	0	0	1	1	1

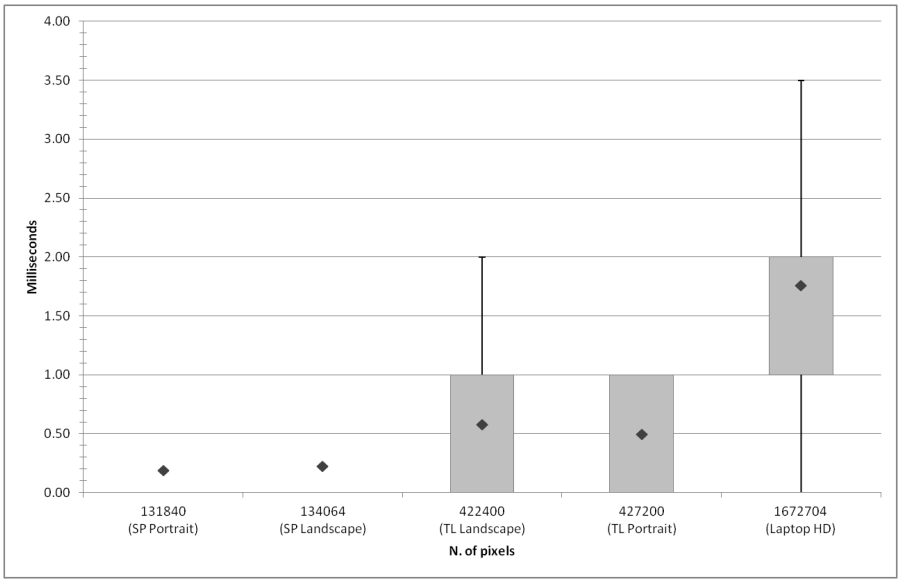

Figure 30: Time spent on base64 encoding in Molecule (in milliseconds)

The statistical data about the time spent on writing the socket buffer for Porthello application are shown in table IX, the ones for Molecule are shown in table X (units of measurement is milliseconds). The corresponding box plots are shown in figure 31 and 32.

Experimental results reveal that the time spent on writing the WebSocket buffer can be considered negligible for all the devices but the Laptop HD in case of Molecule application. More studies need to be done in future work in order to understand why there is such a difference when the dimension of the message increases like in Molecule.

5.3.6 Frames Per Second (FPS)

The last experiment is more qualitative than the others, because it is about a measure, FPS, that is more constant in time than the previous ones. Thus, only average FPS is considered.

TABLE IX: TIME SPENT ON WRITING WEBSOCKET BUFFER IN PORTHELLO (IN MILLISECONDS)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	0.02	0.02	0.02	0.03	0.06
SD	0.13	0.13	0.14	0.16	0.24
Min	0	0	0	0	0
Lower Whisker	0	0	0	0	0
Q_1	0	0	0	0	0
Median	0	0	0	0	0
Q_3	0	0	0	0	0
Upper Whisker	0	0	0	0	0
Max	1	1	1	1	1
IQR	0	0	0	0	0

TABLE X: TIME SPENT ON WRITING WEBSOCKET BUFFER IN MOLECULE (IN MILLISECONDS)

	SP Portrait	SP Landscape	TL Landscape	TL Portrait	Laptop HD
Mean	0.04	0.06	0.15	0.13	21.49
SD	0.19	0.25	0.36	0.47	18.8
Min	0	0	0	0	0
Lower Whisker	0	0	0	0	0
Q_1	0	0	0	0	12
Median	0	0	0	0	19
Q_3	0	0	0	0	28
Upper Whisker	0	0	0	0	52
Max	1	3	1	11	208
IQR	0	0	0	0	16

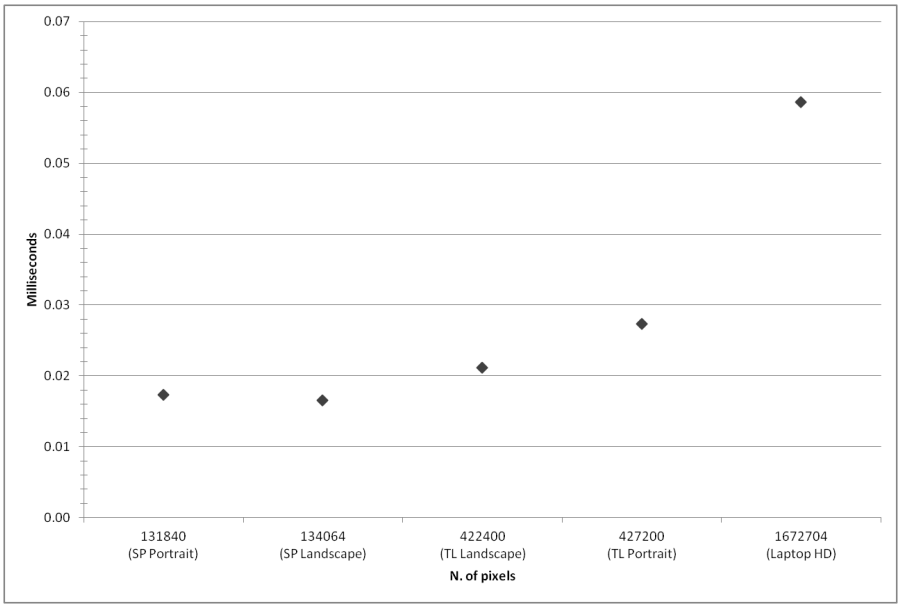

Figure 31: Time spent on writing WebSocket buffer in Porthello (in milliseconds)

The goal of this experiment is to understand how this value decrease with respect to the type of client connected.

In particular, Porthello application has been used, whose average FPS without any client connected is about 50 fps. The data collected are provided in table XI. Both local and remote Laptop HD clients are present, because for this experiment the result is surprisingly different: more studies need to be done about this in future works.

The results obtained point out that application FPS decreases with respect to session camera dimensions, allocated for the client connected. In fact, omegalib needs to render the scene inside an additional camera, spending time proportionally to the size of its dimensions.

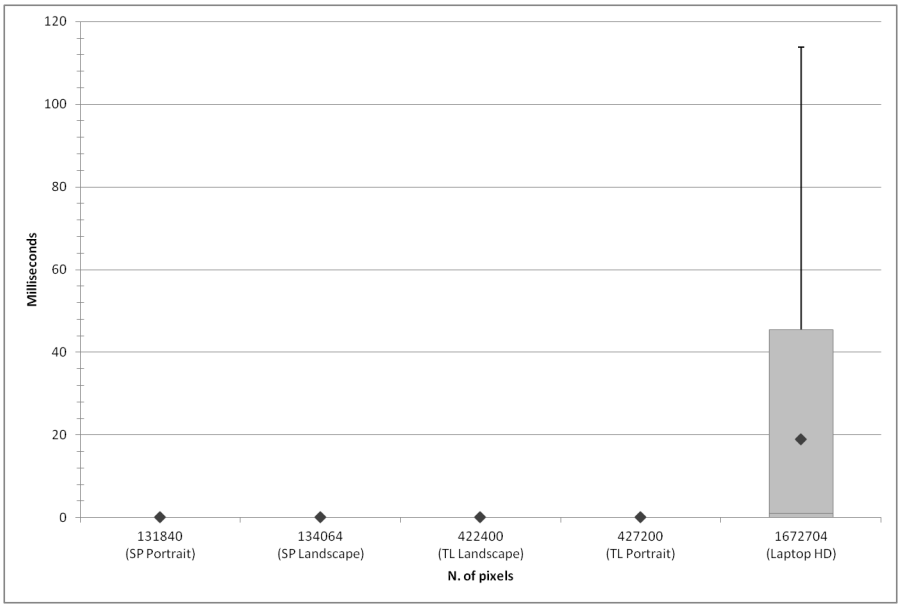

Figure 32: Time spent on writing WebSocket buffer in Molecule (in milliseconds)

TABLE XI: FRAMES PER SECOND (FPS)

Client	FPS
None	50
SP Portrait	48
SP Landscape	42
TL Landscape	40
TL Portrait	38
Laptop HD Remote	20
Laptop HD Local	16

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis presented Porthole, a framework that helps Virtual Environment (VE) applications developers to generate decoupled HTML5 interfaces, and that eases the interaction between a VE system, such as CAVE2™, and smartphones, tablets, laptops or desktop computers, without the need of ad-hoc client applications. CAVE2™ developers now can use handheld devices to interact with their VE applications, just by providing a XML Interfaces Description (XID) file (and, optionally, a CSS for changing GUI style).

The proposed XID file is used by developers in order to specify a set of GUI elements, that are either HTML structures or camera views (used to display in the client either the default VE camera or a custom one), and a set of different interfaces supported, each one composed of a set of GUI elements previously defined.

With Porthole, any device that has a HTML5 enabled browser can now connect to a VE system such as CAVE2™ and receive a tailored interface. The received GUI can display a camera view streaming and makes user able to manage application parameters by interacting with it. Connection, messaging and streaming managements are, hence, transparent to VE developers.

Experimental results have shown that the remote interaction is possible with an acceptable amount of delay, mainly caused by distance, and that measured RTTs of Porthole events ensure that the result of an interaction is always visible before 1 sec. Experiments have also shown that future work is required to address the differences between simple and complex scene streaming.

6.1 Contributions

Earlier work on interaction models between handled devices and large display environments have most focused on using ad-hoc client applications. While this technique may ensure a most performant solution, it is impossible to reuse the same client for each application that developers want to deploy on VE systems such as the targeted CAVE2™.

A relevant problem is that most of earlier work proposed OS dependent solutions, supported only by specific tablets or smartphones, needing the installation of a specific application on a specific device. Moreover, there is no direct manipulation of camera position or orientation from a device.

Porthole addresses these limitations by proposing a novel HCI model that exploits browsers as a mean of interaction with VE systems. The only earlier work that uses such an approach was constrained to a particular HTML structure, usable only in few application domains, and does not provide a streaming of camera images on the client browsers.

Porthole uses Python code injection and C++ functions binding in order to link JavaScript events to VE application changes, making developers able to bind JavaScript events to a server-side execution of a Python code snippet or to a server-side C++ function call. Thus, the proposed approach is not application domain specific, but each developer can use this framework to tailor handheld devices GUI look and behavior for his application.

Finally, Porthole suggests a new concept of remote collaboration, with respect to the proposed Collaborative Virtual Environments (CVEs) model. In fact, earlier work on CVEs shows that such solutions can be used only by institutions or companies that can afford to buy expensive VE systems. Porthole addresses this limitation by making remote collaboration low-cost and by avoiding building any other physical system. With Porthole, remote professionals are now able to see a CAVE2™ scene and interact with it by using ordinary devices.

6.2 Future work

The proposed HCI model proved to be vary useful for both local and remote interactions. Future work can follow different directions. First of all, a general performance improvement needs to be achieved, in order to be able to scale better and support more users, each one with lower interaction and streaming delay. In particular, the idea is to be able to stream a camera view only when something changes in the visualization, not at 50 Hz as it currently is. But, in order to achieve it, omegalib library should be able to detect any application change and inform Porthole Server about it. This would lead to a great performance improvement.

Another future work regards the use of the streaming channel not only for displaying a camera view representing the virtual world scene, but also a completely decoupled 2D visualization, that may show detailed information about the current VE application. Future study will focus on mixing VE 3D applications and handheld devices 2D interfaces inside the same psychical environment.

Regarding remote collaboration, a future work is to enable audio input from devices microphones. In fact, they are currently not usable because of browsers limitations on smartphones and tablets (for security and privacy reasons). Once the browsers will be able to support such a feature, remote collaboration would greatly benefit from user audio input.

Finally, Porthole may lead the way to a new concept of web service, where devices that do not heave enough computational power to visualize a huge dataset, may use cluster-based systems such as CAVE2$^{\text{TM}}$ in order to upload the dataset, visualize it and interact with it on demand, using the cluster as a service.

APPENDICES

Appendix A

XML SCHEMA

This appendix shows the XML Schema for XML Interfaces Description (XID) validation.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema version="1.0"
    attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="xid" type="xidType" />

  <xsd:complexType name="xidType">
    <xsd:sequence>
      <xsd:element name="elements" type="elementsType" />
      <xsd:element name="interfaces" type="interfacesType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="elementsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="element" type="elementType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="elementType">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded" >
       <xsd:any namespace="##any" processContents="skip" />
    </xsd:sequence>
```

Appendix A (Continued)

```
  <xsd:attribute name="id" type="xsd:string" use="required"/>
  <xsd:attribute name="type" type="xsd:string" />
  <xsd:attribute name="camera" type="xsd:string" />
  <xsd:attribute name="width" type="xsd:string" />
  <xsd:attribute name="height" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="interfacesType">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:element name="interface" type="interfaceType" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="interfaceType">
  <xsd:all>
    <xsd:element name="portrait" type="portraitType" minOccurs="1" />
    <xsd:element name="landscape" type="landscapeType" minOccurs="1" />
  </xsd:all>
  <xsd:attribute name="minWidth" type="xsd:int" use="required" />
  <xsd:attribute name="minHeight" type="xsd:int" use="required" />
  <xsd:attribute name="id" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="portraitType">
  <xsd:sequence maxOccurs="unbounded" >
    <xsd:element name="element" type="elementType" />
  </xsd:sequence>
  <xsd:attribute name="layout" type="xsd:string" use="required" />
</xsd:complexType>
```

Appendix A (Continued)

```
  <xsd:complexType name="landscapeType">
    <xsd:sequence maxOccurs="unbounded" >
      <xsd:element name="element" type="elementType" />
    </xsd:sequence>
    <xsd:attribute name="layout" type="xsd:string" use="required" />
  </xsd:complexType>

</xsd:schema>
```

Appendix B

INTERFACE GENERATION EXAMPLE

This appendix provides a complete and valid example of a XML Interfaces Description (XID) file. Then, the interfaces generated for a laptop with HD resolution display, an android smartphone and an android tablet are shown in figure 33, 34, 35, 36, 37, 38 and 39.

Moreover, the following XID has been used during the experiments described in chapter 4, since it defines the camera element for each interface, needed for streaming analysis.

```
<xid>

 <elements>

   <element id="move_ctrl" type="HTML">
     <div class="div-aligned">
       <fieldset data-type="horizontal" class="">
         <input type="button" data-role="button" onmousedown="if(isMaster()):
             camera=getCameraById(%id%); position = camera.getPosition(); position[1]
             += 0.025; camera.setPosition(position);" data-shadow="false"
             data-inline="true" data-corner="false" data-iconpos="notext"
             data-icon="arrow-u" data-continuous="true" class="ui-btn
             ui-btn-icon-notext"/>
         <input type="button" data-role="button" onmousedown="if(isMaster()):
             camera=getCameraById(%id%); position=camera.getPosition(); position[1]
             -= 0.025; camera.setPosition(position);" data-shadow="false"
             data-inline="true" data-corner="false" data-iconpos="notext"
```

Appendix B (Continued)

```
            data-icon="arrow-d" data-continuous="true" class="ui-btn
            ui-btn-icon-notext"/>
        <input type="button" data-role="button" onmousedown="if(isMaster()):
            camera=getCameraById(%id%); position=camera.getPosition(); position[0]
            -= 0.025; camera.setPosition(position);" data-shadow="false"
            data-inline="true" data-corner="false" data-iconpos="notext"
            data-icon="arrow-l" data-continuous="true" class="ui-btn
            ui-btn-icon-notext"/>
        <input type="button" data-role="button" onmousedown="if(isMaster()):
            camera=getCameraById(%id%); position=camera.getPosition(); position[0]
            += 0.025; camera.setPosition(position);" data-shadow="false"
            data-inline="true" data-corner="false" data-iconpos="notext"
            data-icon="arrow-r" data-continuous="true" class="ui-btn
            ui-btn-icon-notext"/>
      </fieldset>
    </div>
</element>

<element id="camera" type="camera_stream" camera="custom"/>

<element id="zoom_slider" type="HTML">
    <div>
        <input type="range" name="slider" id="slider" value="0" min="-800" max="800"
            data-highlight="true" onchange="if(isMaster()):
            camera=getCameraById(%id%); position=camera.getPosition(); position[2] =
            float(%value%)/100; camera.setPosition(position);"/>
    </div>
</element>
```

Appendix B (Continued)

```
  <element id="zoom_ctrl" type="HTML">
    <div class="div-aligned">
      <fieldset data-type="horizontal" class="">
        <input type="button" data-role="button" onmousedown="if(isMaster()):
            camera=getCameraById(%id%); position=camera.getPosition(); position[2]
            -= 0.025; camera.setPosition(position);" data-shadow="false"
            data-inline="true" data-corner="false" data-iconpos="notext"
            data-icon="plus" data-continuous="true" class="ui-btn
            ui-btn-icon-notext"/>
        <input type="button" data-role="button" onmousedown="if(isMaster()):
            camera=getCameraById(%id%); position=camera.getPosition(); position[2]
            += 0.025; camera.setPosition(position);" data-shadow="false"
            data-inline="true" data-corner="false" data-iconpos="notext"
            data-icon="minus" data-continuous="true" class="ui-btn
            ui-btn-icon-notext"/>
      </fieldset>
    </div>
  </element>

</elements>

<interfaces>

  <interface minWidth="0" minHeight="0" id="small">
    <portrait layout="vertical">
      <element width="100%" height="10%" id="zoom_ctrl"/>
      <element width="100%" height="90%" id="camera"/>
    </portrait>
    <landscape layout="horizontal">
```

Appendix B (Continued)

```
    <element width="100%" height="100%" id="camera"/>
  </landscape>
</interface>

<interface minWidth="300" minHeight="300" id="medium">
  <portrait layout="vertical">
    <element width="100%" height="10%" id="zoom_ctrl"/>
    <element width="100%" height="90%" id="camera"/>
  </portrait>
  <landscape layout="horizontal">
    <element width="100%" height="100%" id="camera"/>
  </landscape>
</interface>

<interface minWidth="600" minHeight="600" id="large">
  <portrait layout="vertical">
    <element width="100%" height="5%" id="move_ctrl"/>
    <element width="100%" height="5%" id="zoom_ctrl"/>
    <element width="100%" height="90%" id="camera"/>
  </portrait>
  <landscape layout="horizontal">
    <element width="100%" height="100%" id="camera"/>
  </landscape>
</interface>

    <interface minWidth="900" minHeight="900" id="xlarge">
  <portrait layout="vertical">
    <element width="100%" height="5%" id="move_ctrl"/>
    <element width="100%" height="5%" id="zoom_ctrl"/>
```

Appendix B (Continued)

```
      <element width="100%" height="90%" id="camera"/>
    </portrait>
    <landscape layout="horizontal">
      <element width="90%" height="100%" id="camera"/>
      <element width="10%" height="100%" id="move_ctrl"/>
    </landscape>
  </interface>

 </interfaces>

</xid>
```

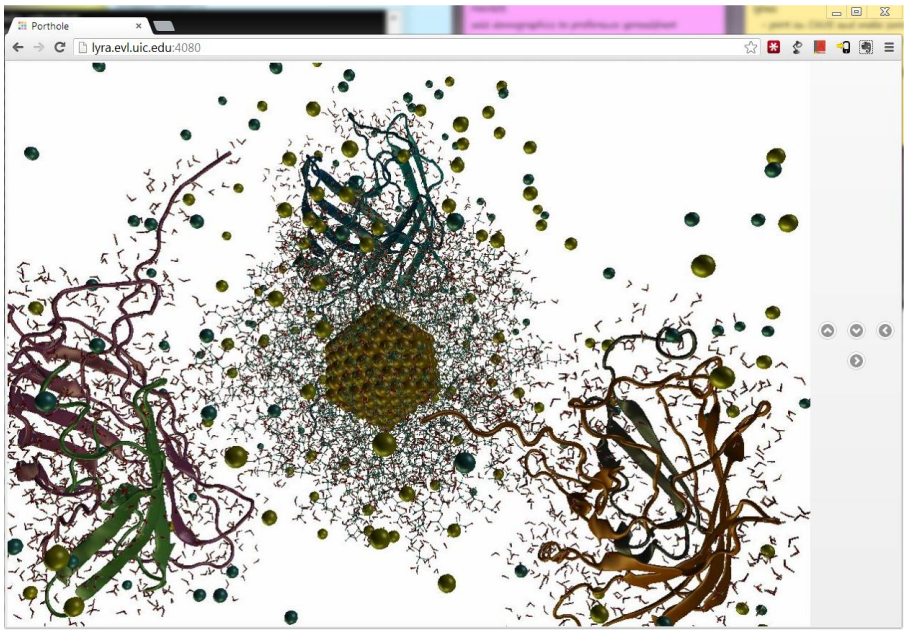

Figure 33: Molecule application (white background version) interface generated for a laptop with HD resolution display

Appendix B (Continued)

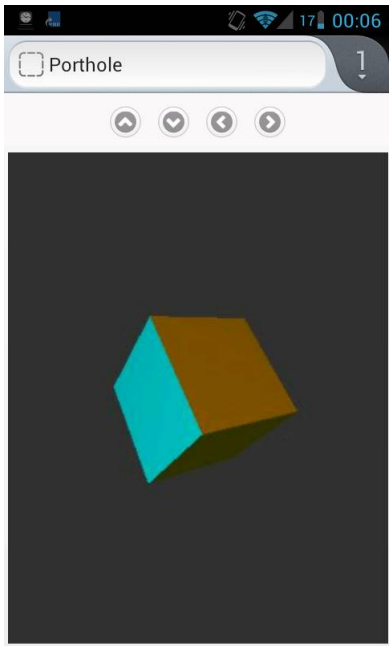

Figure 34: Porthello application interface generated for a smartphone in portrait orientation

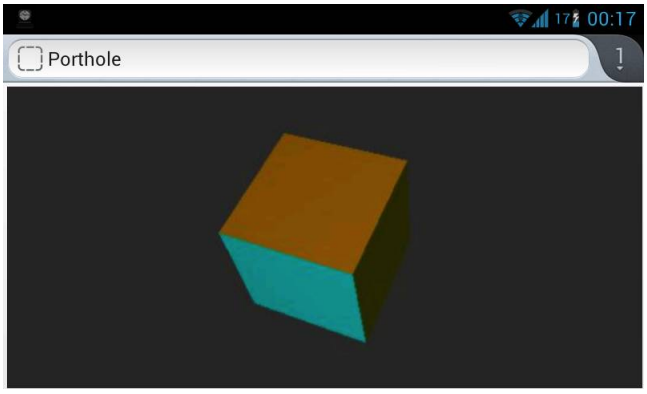

Figure 35: Porthello application interface generated for a smartphone in landscape orientation

Appendix B (Continued)

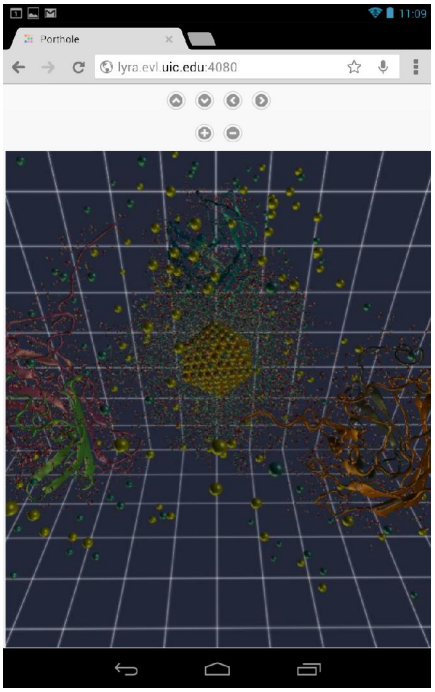

Figure 36: Molecule application (blue background version) interface generated for a tablet in portrait orientation

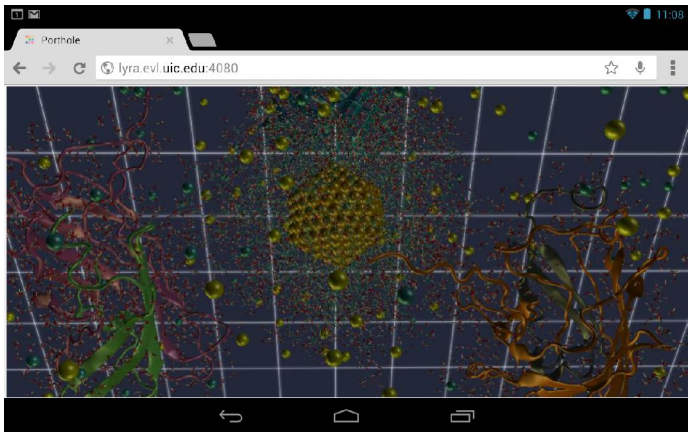

Figure 37: Molecule application (blue background version) interface generated for a tablet in landscape orientation

Appendix B (Continued)

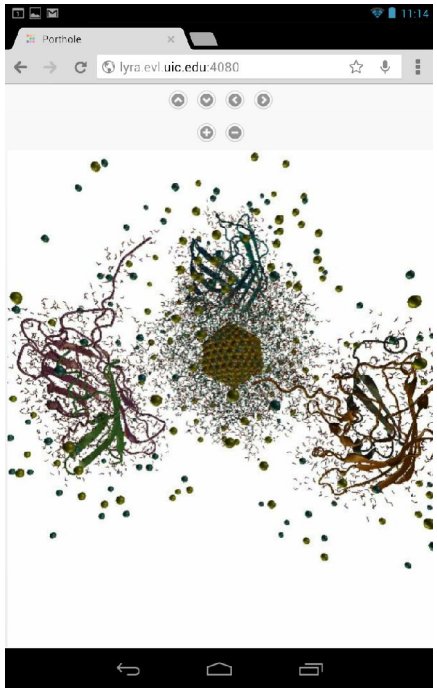

Figure 38: Molecule application (white background version) interface generated for a tablet in portrait orientation

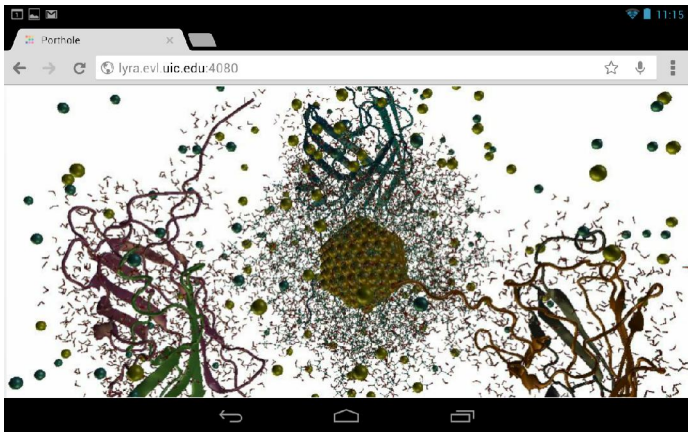

Figure 39: Molecule application (white background version) interface generated for a tablet in landscape orientation

CITED LITERATURE

1. The CAVE Virtual Reality System. http://www.evl.uic.edu/pape/CAVE/.

2. The Next-Generation CAVE Virtual Environment (NG-CAVE). http://www.evl.uic.edu/core.php?mod=4&type=1&indi=421.

3. CAVE2: Next-Generation Virtual-Reality and Visualization Hybrid Environment for Immersive Simulation and Information Analysis. http://www.evl.uic.edu/core.php?mod=4&type=1&indi=424.

4. Stedmon, A., Bayon, V., Griffiths, G.: Expanding interaction potentials within virtual environments: Investigating the usability of speech and manual input modes for decoupled interaction. *Adv. Human-Computer Interaction*, vol. 2011, 2011.

5. Hoang, R.V., Hegie, J., Jr, F.C.H.: Scrybe: A tablet interface for virtual environments. In F.C.H. Jr., F. Hu, eds., *Proceedings of the ISCA 23rd International Conference on Computer Applications in Industry and Engineering, CAINE 2010, November 8-10 2010, Imperial Palace Hotel, Las Vegas, Nevada, USA*, pp. 105–110. ISCA, 2010.

6. Mavrody, S.: *Sergey's Html5 & Css3 Quick Reference*. Belisso, 2010.

7. Can I Use: Compatibility tables for support of HTML5, CSS3, SVG and more in desktop and mobile browsers. http://caniuse.com/.

8. The Encyclopedia of Virtual Environments. http://www.hitl.washington.edu/scivw/EVE/.

9. Peternier, A., Cardin, S., Vexo, F., Thalmann, D.: Practical design and implementation of a cave system - high quality four walls cave howto. In J. Braz, P.P. Vázquez, J.M. Pereira, eds., *GRAPP (AS/IE)*, pp. 129–136. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, 2007.

10. Bowman, D.A., Hodges, L.F.: User interface constraints for immersive virtual environment applications. Tech. Rep. 95-26, Graphics, Visualization and Usability Center, Georgia Institute of Technology, USA, 1995.

11. Brooks Jr., F., et al.: Final technical report: Walkthrough project, 1992.

12. Hodges, L., Kooper, R., Meyer, T., Rothbaum, B., Opdyke, D., Graaf, J., Willford, J., North, M.: Virtual environments for treating the fear of heights. *IEEE Computer*, vol. 28:27–34, 1995.

13. Duval, T., Fleury, C.: An asymmetric 2D pointer / 3D ray for 3D interaction within collaborative virtual environments, 2009.

14. Poupyrev, I., Billinghurst, M., Weghorst, S., Ichikawa, T.: The go-go interaction technique: Non-linear mapping for direct manipulation in VR. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Papers: Virtual Reality (TechNote), pp. 79–80, 1996.

15. Bowman, D.A., Hodges, L.F.: An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. pp. 35–38, 1997.

16. Zhai, S., Buxton, W., Milgram, P.: The "silk cursor": Investigating transparency for 3D target acquisition. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, vol. 1 of *Interacting in 3-D*, pp. 459–464, 1994.

17. Bowman, D.A., Coquillart, S., Froehlich, B., Hirose, M., Kitamura, Y., Kiyokawa, K., Stürzlinger, W.: 3D user interfaces: New directions and perspectives. *IEEE Computer Graphics and Applications*, vol. 28:20–36, 2008.

18. Sage: Scalable adaptive graphics environment. http://sagecommons.org/jupgrade/.

19. Leigh, J., Johnson, A.E., DeFanti, T.A.: Issues in the design of a flexible distributed architecture for supporting persistence and interoperability in collaborative virtual environments. In *SC*, p. 21. IEEE, 1997.

20. Macedonia, M.R., Zyda, M.: A taxonomy for networked virtual environments. *IEEE MultiMedia*, vol. 4:48–56, 1997.

21. Barrus, J., Waters, R., Anderson, D.: Locales and beacons: Precise and efficient support for large multi-user virtual environments. *Proceedings of VRAIS'96, Santa Clara CA*, pp. 204–213, 1996.

22. Carlsson, C., Hagsand, O.: DIVE - a multi user virtual reality system. In *VR*, pp. 394–400, 1993.

23. Shaw, C., Green, M.: The MR toolkit peers package and experiment. *Proceedings of VRAIS'93*, pp. 463–469, 1993.

24. Hariri, S., Kim, D., Kim, Y., Ra, I.: Virtual distributed computing environment.

25. Wang, Q., Green, M., Shaw, C.: EM - an environment manager for building networked virtual environments. *Proceedings of VRAIS'95*, 1995.

26. Park, K.S., St, S.M., Kapoor, A., Leigh, J.: Lessons learned from employing multiple perspectives in a collaborative virtual environment for visualizing scientific data, 2000.

27. Leigh, J., Johnson, A.E.: Supporting transcontinental collaborative work in persistent virtual environments. *IEEE Computer Graphics and Applications*, vol. 16:47–51, 1996. ISSN 0272-1716.

28. Leigh, J., Johnson, A.E., Vasilakis, C.A., Defanti, T.A., Wurman, R.S.: Multi-perspective collaborative design in persistent networked virtual environments, 1996.

29. Leigh, J., Johnson, A.E.: CALVIN: An immersimedia design environment utilizing heterogeneous perspectives. In *ICMCS*, pp. 20–23, 1996.

30. Wingrave, C.A., Williamson, B., Varcholik, P., Rose, J., Miller, A., Charbonneau, E., Bott, J.N., Jr, J.J.L.: The wiimote and beyond: Spatially convenient devices for 3D user interfaces. *IEEE Computer Graphics and Applications*, vol. 30:71–85, 2010.

31. Kavakli, M., Taylor, M., Trapeznikov, A.: Designing in virtual reality (desIRe): a gesture-based interface. In K.K.W. Wong, L.C.C. Fung, P. Cole, eds., *Proceedings of the Second International Conference on Digital Interactive Media in Entertainment and Arts, DIMEA 2007, 19-21 September 2007, Perth, Western Australia*, vol. 274 of *ACM International Conference Proceeding Series*, pp. 131–136. ACM, 2007.

32. Beaudouin-Lafon, M.: Lessons learned from the WILD room, a multisurface interactive environment. In M. Riveill, ed., *23th French speaking Conference on Human-Computer Interaction, IHM'11, Sophia Antipolis, France - October 24 - 27, 2011*, p. 18. ACM, 2011.

33. Bauer, J., Thelen, S., Ebert, A.: Using smart phones for large-display interaction. In *User Science and Engineering (i-USEr), 2011 International Conference on*, pp. 42 –47, 2011.

34. Brown, M.S., Wong, W.K.H.: Laser pointer interaction for camera-registered multiprojector displays. In *ICIP (1)*, pp. 913–916, 2003.

35. Omegalib: A next generation framework for virtual reality and cluster-driven display systems. https://code.google.com/p/omegalib/.

36. OpenGL (Open Graphics Library). http://en.wikipedia.org/wiki/OpenGL.

37. OpenSceneGraph. http://www.openscenegraph.org/projects/osg.

38. Cyclops programming in OmegaLib. http://code.google.com/p/omegalib/wiki/BasicCyclops.

39. Visualization Toolkit. http://www.vtk.org/.

40. W3C HTML Working Group. http://www.w3.org/html/wg/.

41. WHATWG HTML Living Standard. www.whatwg.org/html/.

42. The WebSocket API. http://dev.w3.org/html5/websockets/.

43. libwebsockets HTML5 Websocket server library in C. http://git.warmcat.com/cgi-bin/cgit/libwebsockets/.

44. Hammer.js A JavaScript library for multi touch gestures. http://eightmedia.github.com/hammer.js/.

45. The FreeImage Project. http://freeimage.sourceforge.net/.

46. Safari Web Content Guide: Handling Events. http://developer.apple.com/library/ios/.

47. VisualRoute - Traceroute and Reverse trace. http://www.visualroute.com/.

VITA

NAME Daniele Donghi

EDUCATION

2011 - present	Master of Science in Computer Science, University of Illinois at Chicago, Chicago, USA.
2010 - 2012	Laurea Specialistica in Computer Engineering, Politecnico di Milano, Milan, Italy.
2007 - 2010	Laurea in Computer Engineering, Politecnico di Milano, Milan, Italy.