Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Master in Computing

MASTER THESIS

Algorithms for a multi-projector CAVE system

Student: Javier Tibau
Directors: Carlos Andújar & Pere Brunet

September 2010

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Virtual Reality (VR) applications are very complicated to develop and deploy. Besides the common problems dealt with by any computer graphics programmer, a VR researcher must usually face a combination of the following issues:

- Uncommon input devices. Some of them have very specific and rare usage requirements and configuration. Many use legacy interfaces.
- Multiple displays. These must be skillfully placed, calibrated and configured.
- Complex projects require many PCs working together for multiple purposes:
  - Connecting as many input devices as the application needs. For example, some tracking technologies are based on optical sensors, requiring the system to connect a high number of cameras to provide redundancy, low latency, error checking and detect object occlusion.
  - Feed a higher number of displays than can be connected to a single PC.
  - Accelerate the rendering tasks, whether the speed up is needed online or offline. With an appropriate parallelization strategy, the frame-rate of the running application can be significantly accelerated.
- Whn using multiple PCs or Displays, synchronization and networking becomes an issue that is far our area of expertise:
  - All PCs must have a coherent time stamp.

  - Non-deterministic algorithms should be performed on a single PC, their results then distributed amongst all others.

- Furthermore, for a truly immersive experience, VR applications need to provide some form or other of 3D sound rendering. This being, yet again, a field of specialization out of the scope of this project.

While there are currently many software packages available, both freely available and commercial ones, there is always the tradeoff between:

- Programming experience required. Including the prerequisites and also learning how to program using the specific VR Framework.

- Flexibility and configurability.

- Cost: Initial investment, maintenance of the execution platform and upgradability. Bearing in mind that although the software may cost nothing, the administrator still has to spend a long time learning the quirks of the system.

By this definition and general experience, it is impossible to reach a state of perfection in all three of the aspects of this tradeoff, at the same time.

This document overviews some of the publicly available and most widely used solutions. By leveraging on their knowledge and experience a viable alternative is proposed.

The idea for this thesis sprung from the need to update the infrastructure of an old "Cave Automatic Virtual Environment" (CAVE) system from the "Centre de Realitat Virtual de Barcelona" (CRV). The previous incantation of the CAVE was made from commercial hardware and software components. Porting and deploying current state-of-the art applications was not an easy task. Upgrading the old system was too expensive and not viable in the long term.

ALIVE, the project here described has two main tangible objectives:

- To refresh of the old CAVE at CRV and be able to work with more modern hardware and software.

- Facilitate the development of VR applications, that will be deployable in the rehabilitated CAVE.

ALIVE stands for Abstraction Layer for Interactive Virtual Environments. It introduces an abstract Application Programming Interface, with a three-layered architecture, meant to provide an effective middle point that addresses the three aspects of the aforementioned tradeoff separately.

Instead of developing a VR library from scratch, we decided to use existing work on this field from community projects available freely online. Since each library has its strengths and shortcomings, an architecture that allowed the integration of many libraries was highly desirable. With such a thing in mind, a three-layered two-sided abstraction architecture was envisioned. The three-layers correspond to the following components:

Application Filled by the user's application-specific code. A sample polygon renderer application was developed to provide a concrete description of this layer.

Abstraction These classes enforce application flow and component connection. On instantiation these classes are fed with concrete implementations from the other layers.

Backend Backends, in the context of this document, are the wrapper classes that enable alive applications to interact with a specific VR Framework. At the time of writing, we provide a wrapper class for VR Juggler as well as sample configuration files for the VR Juggler environment.

The Abstraction layer masks the interaction between the polygon rendering code and the VR Juggler backend. Upon compilation, neither is meant to know about the other. They only meet at run-time, although only through the abstract interfaces in the Abstraction layer.

1.2 Objectives

With regards to facilitating development of VR applications, the main purpose of ALIVE is to reduce the amount of attention that the application developer has to dedicate to the issues that were described previously. In this project we aim to abstract the user from dealing with:

- Input devices.
- Display number and layout.
- Definition of the virtual cameras.
- Synchronization issues between cluster nodes.

Notably missing from the list are 3D sound rendering and synchronization for non-deterministic algorithms. These problems are out of the scope of this project and will be addressed in the future.

Summarizing the objectives of this project, we list:

- Provide an abstraction API, that facilitates development and deployment of VR applications.
- Create a polygon renderer application based on the proposed API.

- Update the CAVE software infrastructure.
- Abstract the users of the CAVE from installation and configuration issues.
- Facilitate de migration of VR Applications to this framework.

1.3 Document Organization

The rest of this thesis is organized as follows:

In Chapter 2 we overview the state of the art on parallel rendering techniques, libraries and 3D interaction methods. These are integral issues in a VR application. As the complexity of an application increases, so does the weight of these problems.

Chapter 3 details the design decisions applied for the implementation of ALIVE. Application flow and development is also explained as it pertains to the usage of the API.

A summary of the project is provided in Chapter 5, with our perceived achievements, conclusions and proposals for future work.

Lastly, from a practical point of view, the appendixes provide an indispensable help for installing, compiling and configuring the project and its dependencies.

Chapter 2

Project Background

As a proper introduction to the subject of parallel rendering, this first section introduces the Cave Automatic Virtual Environment (CAVE) by means of a brief history of its development. This introduction serves to illustrate the many issues present when developing multi-display systems.

The second section describes parallel rendering in more detail. Analyzing advantages and disadvantages of different approaches. Meanwhile the last section in this chapter surveys some of the available software packages that tackle the issues of parallel rendering from various angles.

2.1 Brief History of the CAVE

Developed by the Electronic Visualization Laboratory (EVL) at University of Illinois at Chicago, the first Cave Automatic Virtual Environment (CAVE), was shown at SIGGRAPH'92[1]. It quickly became popular as a VR system and has been replicated in many research centers worldwide.

The first generation CAVE used active stereo and three-tube CRT projectors to project 1280x1024 images onto $3m^2$ screens. The layout of the system included 3 rear-projected walls and a down-projected floor, this provided a novel sense of immersion. The user's head and hand tracking was performed by an Ascension, Inc. Flock of Birds electromagnetic trackers. Amongst its many shortcomings: the images were dim and had low spatial resolution; the 5 SGI Crimson workstations employed did not have enough power to provide more than 8 frames per second, therefore not achieving a sufficient rate for animation. The system was later updated with Marquee projectors and an SGI Onyx system.

For a second-generation CAVE in 2001, EVL focused on improving the image quality by utilizing Christie Mirage DLP projectors, which were brighter but more expensive than the previous equipment. Also improved was the frame rate, due to the usage of newer SGI Reality Engine, achieving around 25 frames per second. Spacial resolution was still a problem. Suc-

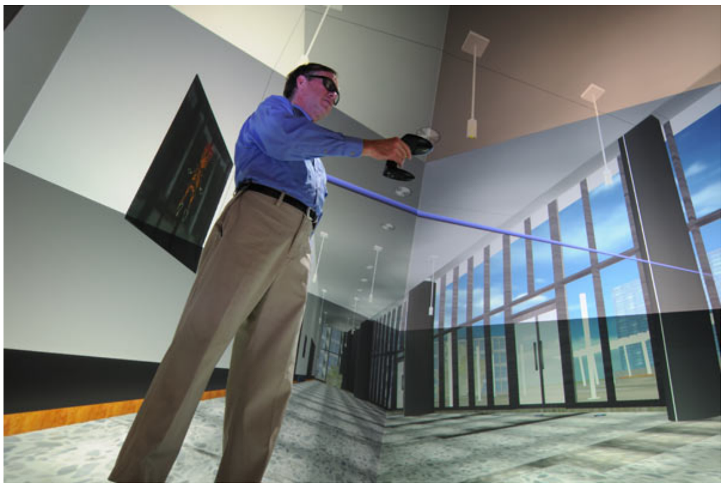


Figure 2.1: The StarCAVE, a “third-generation” CAVE [2]

cessive work from EVL was aimed at increasing the number of pixels per screen by usage of tiled displays.

The StarCAVE[2], a third-generation CAVE system, is a 5-wall plus floor virtual reality room. By using tiled-displays coupled with integrated circular polarization on LCOS LCD projectors, they provide a high-resolution environment viewable through lightweight polarized glasses. Furthermore, the basic layout of previous systems was discarded in favor of a pentagon-shaped room, with one rail-mounted wall to allow for access.

Universitat Politècnica de Catalunya, through the Centre de Realitat Virtual, owned an upgraded version of a first-generation CAVE system. Though the physical structure remains usable, not so does the old equipment, which is severely outdated and too expensive for a feasible renewal path.

In order to rehabilitate the CAVE, this and a sibling project sprung to life.

2.2 Distributed Computing Paradigms

The term “distributed computing” is generally applied to system of networked computing nodes, which perform a task together and coordinate their actions by message passing. There are two main paradigms that we need consider when dealing with distributed systems for VR.

2.2.1 Server-Client

In a client-server system, there are essentially two types of nodes. The server, which provides a service to be used by the many clients. On a VR platform, usually one machine acts as the input server, processing the input data from devices and providing digested information to the other nodes. The clients are constantly requesting updates updates of the data from the server.

2.2.2 Master-Slave

One master node controls task execution, and distributes the subtasks amongst slave nodes. The master PC may also have a slave process running. Most VR frameworks use some variation of master-slave architecture, where one master node controls that all clients remain synchronized and in the same frame step.

The VR Juggler cluster architecture also puts the master node in charge of sending the configuration parameters to the slave nodes.

Both of these distributed paradigms result in a network topology shaped like a star, centered around the server or master node. The other nodes depend on the central node to provide a service or instructions needed.

2.3 Parallel Rendering

Alongside the introduction of ever more effective display technologies and equipment, development of new algorithms for scaling the number of pixels, screens or image quality has also been underway. Most, if not all, of the effort has been focused on improving the parallelism of the rendering pipeline.

The appearance of high performance 3D commodity graphics cards in the 90's sparked the use of PC clusters for High Performance Visualization (HPV). A cluster implementation was already used to drive the first CAVE[1]. Nowadays, PC clusters provide their users with several advantages over dedicated supercomputers[3]:

- Low cost, being built from commodity components.
- High modularity and scalability that allows the cluster to adapt to its user's needs.
- Standards compliance and large range of Open Source Software solutions.

Asides from CAVEs, there are many other applications for PC clusters on HPV. Most commonly, an advance user may want an increase in image quality in order to properly perform a specific task. This will almost always push the envelope of performance for a single workstation. For an instance, if a user requires higher visual acuity on a large display, many problems arise besides the obvious need for a better screen:

- More calculation power is needed.

- With an increase in resolution, many imperfections of a low quality model may become visible. Hence, higher resolution data is desirable and this taxes available memory of a single system.
- More than a single display may be required to reach the desired spatial resolution. A separate graphics card might be needed, if not for its processing power, due to its added graphics ports.

2.3.1 Problems

The typical performance bottlenecks on the basic (single) rendering pipeline can be generally described as being:

Fill Rate The amount of pixels to be rendered can exceed the capabilities of the graphics card.

Geometry Transformations As in Fill Rate, the amount of triangles to be processed can become an issue.

GPU Memory Specially with volume rendering, the amount of data that needs to be held on the GPU might be larger than its available memory.

Bus Bandwidth Usually with dynamic data, when it cannot be kept on the GPU but must be generated or transferred on each frame, the speed of the available transmission channels might not suffice.

CPU Performance The CPU may also become a bottleneck, if it is unable to traverse the database to generate the rendering commands with enough speed. Tesselation or visibility computations might cause this.

Main Memory Occurring when the model data is larger than main memory

IO Bandwidth The speed with which the system can move the data from storage to the main memory.

There are many ways to parallelize the rendering pipeline. However, none of them is an optimal solution for all of the bottlenecks mentioned. Most attend to mitigate a specific issue, therefore making one appropriate for a specific visualization problem, but terrible for another.

2.3.2 Taxonomy

Based upon the following taxonomy[4], it is easier to understand the strengths and weaknesses of a particular approach to parallelization.

Functional Parallelism

The rendering process is split into several clear and distinct functions. These functions will be applied sequentially to the individual data items. When a unit finishes working on a piece of data, it forwards its output to the next element and immediately takes a new instruction. This is the general structure of a rendering pipeline. While very successful in design and application, this approach suffers from being limited to the speed of its slowest processing unit, which becomes the bottleneck of the pipeline. Parallelism is also limited to the maximum number of individual steps that one may be able to define.

Temporal Parallelism

In the two previous approaches, one may consider only still images. Temporal parallelism is exploitable only for animations. If the time required to produce a high quality frame is t_f, rendering n frames would take $n * t_f$. For non-interactive animation, one may task m processors and render the complete animation in $n/m * t_f$ time. This may also be applied to interactive applications, trading quality for some latency since future object states have to be predicted.

Data Parallelism

While functional parallelism is used as a basic building block, considering its limits, further parallelization is achieve by contemplating multiple streams instead of a single pipeline. Simply, the input data is distributed amongst the processors. It has been widely adopted, while not trivial to implement, it is highly scalable since the number of elements can vary depending on the complexity of the rendering task. This type of parallelism deserves special consideration, as the distribution of the data and the merging of the output can be done on different stages of the pipeline[5].

While functional and temporal parallelism cases are trivial to realize and implement. Data parallelism, however easy to understand, suffers from the algorithmic difficulty of deciding when to distribute the data and when to share the results of individual processors.

Considering the following steps in the rendering pipeline:

1. Geometry Processing (GP): Transformations
2. Rasterization
3. Composition

Communication between processors can happen on three distinct places of the pipeline: Before processing the geometry, after applying the transformations but before rasterization and after rasterization.

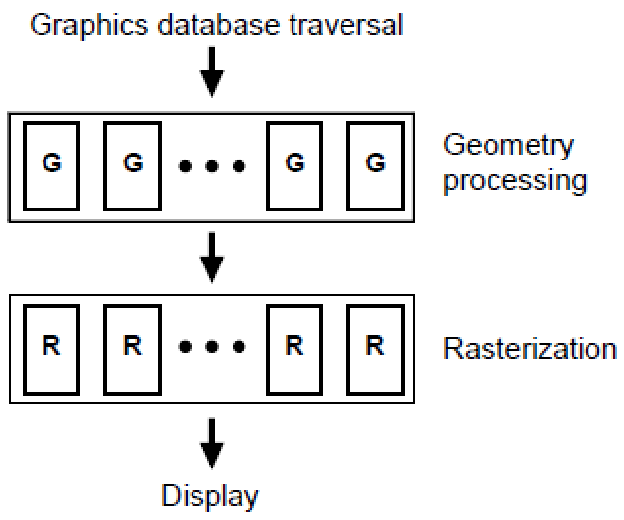


Figure 2.2: A simplification of the standard pipeline, as present on most single pipeline graphics processors [5]

Sort-first

Sort-first parallelization may be best described as a projection surface 2D subdivision problem. Each processor gets tasked with producing a tile of the entire final view. Compositing the final image just requires to place the sub-images side by side in a predefined layout.

The name is given under the consideration that the object database is distributed amongst all the processing units and communication occurs before applying the geometry transformations: each unit confirms that the objects it possesses falls inside its viewing frustum; If the object falls outside, the unit sends the object to the appropriate neighboring processor. Afterwards, rasterization occurs as if working on independent pipelines.

This is a very classical approach for clusters, as it entails little modification to the code structure. The most important benefits and drawbacks are now listed:

+ Low communication requirements. Very little information needs to be passed from node to node, mostly just input data from sensors.

+ Spatial coherence between rendered frames can be heavily exploited, this eliminates most of the initial communication overhead of exchanging objects with neighbors.

+ If enough memory is available, the object database may be replicated on each node. Further eliminating the communication overhead.

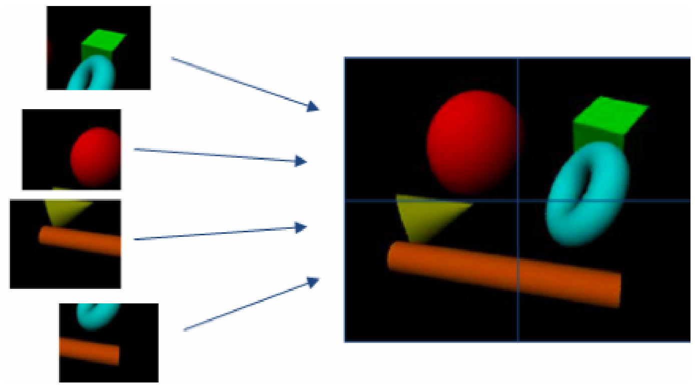


Figure 2.3: Sort-First: Spacial division of the scene representation [3]

- Input data is analyzed and processed in all nodes independently.
- Geometry transformations and overlapping primitives are also processed redundantly.
- If implemented without interprocessor communication, non-deterministic algorithms will obtain different results on each machine.
- Certain tiles may have to generate parts of the scene with far greater complexity than others. This is a highly undesirable load imbalance.

Sort-middle

Each processing unit applies all the geometry transformations to its objects. Communication between processing units occurs just before rasterization, the data for each object has already been converted to screen primitives and are ready for rasterization.

Geometry Processing units and Rasterizing units may be better considered as separate here. Since Geometry Processors are assigned arbitrary objects from the database, while the Rasterizers are still assigned a 2D section of the final view.

The redistribution of data occurs at an intuitive place in the pipeline, between geometry processors and rasterization. However, while this can be considered desirable design-wise, it does not work well with current popular techniques such as tessellation. If a high tessellation ratio r, is applied to the primitives, sort-first takes advantage in that only the raw primitives n are transmitted. Compared to sort-first:

- The cost of transmission is highly increased.
- Load imbalance is also present, for the same use cases.

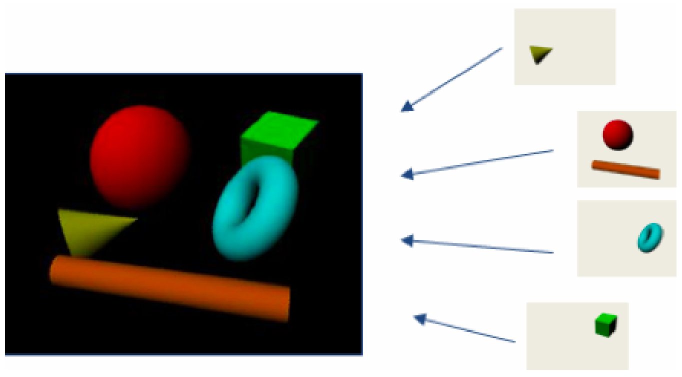


Figure 2.4: Sort-Last: Balanced division of the scene primitives [3]

- Commodity graphics cards don't allow the user to retrieve data before rasterization. This makes the approach unpractical for PC clusters, since the geometry transformations would have to be performed solely on CPU[3].

Sort-last

The application partitions the database so that each rendering unit renders completely an arbitrary set of the data.

The scalability of this method is high, maintaining some aspects of the pipeline fixed for each unit, such as: Memory usage, IO bandwidth, GPU usage and so on. However, the final compositing step has a linearly increasing cost depending on the number of processing units used.

The compositing unit merges the separate rendered images correctly depending on the alpha and depth values obtained by each rasterizer.

The high transmission cost of the compositing step is the main concern for this approach, however several great advantages occur:

+ It is easy to achieve load balance.

+ The rendering pipelines of each node are independent until composition.

+ With a hardware based approach, pixel composition may be greatly accelerated.

With or without hardware acceleration at the compositing level, extremely high pixel traffic may occur as this increases with the node count. Oversampling may also become prohibitive.

Hybrid Approaches

Combinations of the previous forms of parallelism are clearly possible. As hinted, a current HPV cluster may already implement functional and data parallelism if analyzed on a low-level. Very complex systems may result from such hybrid architectures, where appropriate combinations may serve to leverage the shortcomings of the different approaches.

For example, a combination of temporal and sort-last parallelism could be laid-out in the form of two separate high-performance clusters. If all nodes worked together to produce each frame, the amount of pixel data generated may surpass the network capabilities. If, however, the nodes are split into two separate subnets, the generated data would effectively be halved. Each frame would take longer to generate, but the next frame is already being rendered by the other cluster. All in all, such a design could generate a better use of the network at the expense of some latency.

2.4 Frameworks

Some software packages have achieved considerable recognition for CAVE development. Although a couple of the solutions are commercial, most of them originated in educational research environments and are freely distributed under open source licenses. This section is a brief description about the more predominant packages. Commentary about the strengths and weaknesses of each, as they relate to the previously mentioned taxonomy, is given.

CaveLib

The original API for the CAVE. It was created by EVL and today it is commercialized by Mechdyne [16].

It uses a replication sort-first approach. It was first deployed with the cluster of SGI workstations that powered the CAVE. Presently, it supports PCs with Windows or Linux operating systems.

As a low-level API, it abstracts the programmer from handling tasks such as:

- Viewport creation
- Cluster synchronization
- Data sharing

The user must only provide the graphics for the system. Integration with higher level OpenGL graphics APIs is supported.

VR Juggler

One of the most widely used software solutions. It allows the programmer to write an application regardless of display and interaction devices, without changing code or recompiling[6].

The configuration of a VR Juggler environment is achieved by means of an xml file, which defines:

- Number of PC Nodes, their available hardware and network addresses.
- Input Devices such as 3D trackers, virtual gloves, wands, etc.
- Displays, their physical layout and to which node they correspond.

Besides providing device abstraction, VR Juggler does not provide many additional features. The programmer must interact with the VR Juggler kernel by subclassing an App class in a process described by its developers as “filling the blanks”. This is a very straightforward and general purpose approach. VRJuggler is configurable at run-time and provides a GUI configuration tool to make changes on the fly. It supports most mainstream operating systems and rendering through OpenGL.

VR juggler’s parallelization strategy is one of application replication amongst cluster nodes. In other words, the master node remains in charge of state synchronization and sharing input data from the sensors. Each slave node is reproducing the same steps unaware of the other slaves.

Chromium

The main advantage of Chromium over other solutions is that it provides its users with a way to automatically execute their applications on a visualization cluster. It accomplishes this by intercepting the OpenGL calls sent by the application and redistributing them appropriately. It is unique in that no other system provides this kind of functionality, even commercial applications can be scaled, whereas other solutions require access to the source code[7].

As a parallel rendering API, however, it is considered to be rather limited. If a parallel rendering application is built from scratch, other solutions are preferable in that they offer the developer more flexibility and performance.

Equalizer

The Equalizer framework is one of the newer projects focused on parallel rendering. Self-described as ”GLUT on steroids”, it places severe emphasis on developing parallel application from the beginning, instead of expecting a separate solution to provide parallelization[8].

It is multi-platform, OpenGL based and distributed under the LGPL open source license. Being a good combination for both scientific and commercial applications.

One of its main selling points is the abstraction of display configuration. After developing an application using the framework, the user can choose between the many types of parallelism explained in Section **??** at runtime. This allows the exact code to be run at the highest possible performance on a tiled-display, a CAVE or a single screen backed by a remote visualization cluster.

OpenSG

Quite different from the other mentioned solutions, OpenSG is a portable scene graph system. It allows manipulation of the scene to asynchronous threads. It can be run on PC clusters with the data changes being applied automatically to the other nodes[9].

Rather than being a complete solution, it is used with other tools like VR Juggler, to provide data synchronization to the package.

FlowVR

FlowVR is a dedicated middleware for VR applications[10]. It proposes an approach for developing applications based on modules:

- Modules exchange data through a FlowVR network. Each module runs on its own thread.

- Daemons running on the cluster's nodes are in charge of communicating modules on the network.

- The FlowVR network can implement simple connections between modules, or complex instances of data synchronization, frustum culling, etc.

Alongside, the same group develops FlowVR Render, a library built on top of FlowVR. It implements the necessary modules and network for a sort-first parallel renderer.

Comparison

Table 2.1 pits the aforementioned libraries against each other to better illustrate the purpose of each approach. Amongst them, OpenSG is the only one whose purpose does not generally allow direct comparison. It is a distributed scene graph, while the others are full-blown solutions to address the VR development issues.

	Distributed Paradigm	Parallel Rendering	Development	Input Management	Application Porting
CaveLib	Server-Client	Sort-First	Commercial	Yes	Re-implement with API
VR Juggler	Master-Slave	Sort-First	Free & Active	Yes	Re-implement with API
Chromium	Server-Client	Defined by Network	Free but Stale	No	Unmodified Applications
Equalizer	Master-Slave	Many Configuration	Free & Active	No	Re-implement with API
FlowVR	Server-Client	Defined by Network	Free & Active	No	Re-implement with API

Table 2.1: Comparison of VR Frameworks

2.5 Interaction in 3D

While users have grown accustomed to interacting with WIMP[1], these methods cannot be used as effectively on a Virtual Environment. The introduction of the aforementioned components, if done wrong, may even lessen the sense of immersion.

The are, essentially, three interaction tasks that a system has to offer to provide a fully interactive and immersive experience: Navigation, Selection and Manipulation[11]. All complex interaction inside a VE can be recreated by using the aforementioned tasks as building blocks.

A key ingredient for all interaction techniques is the use of a 6 degrees-of-freedom (DOF) 3D tracking device. A 6 DOF device can determine the position (x, y, z) and orientation (roll, pitch and yaw). This provides de researcher with a powerful tool to enable natural interaction.

Furthermore, there is the possibility of allowing for non-isomorphic representation of the input data. That is, if the user moves its hand over a space of 1 meter, the virtual object being manipulated might actually perform a translation of 2 meters. This non-isomorphic interpretation of the relation between physical space and virtual space can allow for very control of the virtual world.

This section presents some of the current approaches used to accomplish each of the basic interaction tasks.

2.5.1 Selection

Selection is the means to access one or many objects in a virtual world. It is intricately related to manipulation, which will be analyzed later, but bear in mind that some of the techniques are evidently shared.

Techniques

At the core, each technique must solve the issue of discerning amongst the collection of elements which is the one that must be picked out. At the same time, the selection technique might specify on when the selection should take place: On the press of a button, a hand gesture or other type of signal.

Often, finding if an object is selected is a hierarchical task. This involves, first checking against the axis aligned bounding box of a selectable object, and keep going down on the structure until verification with the primitives of the object.

Virtual Hand A very natural technique, involves simple collision detection tests between the geometry and the hand.

[1]Window, Icon, Menu and Pointing Device

Ray-Casting The metaphor applied here is that of a laser pointer. It is a very powerful since it allows selection of objects at a distance. It involves the creation of a ray whose origin and direction are determined by the hand's position and orientation vectors. Line-object intersections checks must be performed on the selectable objects.

Occlusion Technically similar to the above. The ray is meant to start at the user's eye, and be directed towards the hand. The metaphor being that the selection occurs when the object is "touched" at a distance.

Arm-Extension The Go-Go technique, inspired on the cartoon Inspector Gadget. Is a non-isomorphic representation as per the explanation at the beginning of this section. Before a certain distance threshold, the user's hand movement performs isomorphically, but passing this threshold its extension becomes exponential. A slight issue for this technique is the definition of regarding to which reference point to calculate the extension of the wand, in many cases this is point is located at the torso.

2.5.2 Manipulation

As mentioned earlier, manipulation is very much connected with selection. Since first of all, an object must be selected before applying any manipulation to it. t should be noticed that it is not desirable that we keep testing for intersections when an object is already being manipulated.

An issue that is most intrinsic to the manipulation task, besides applying a certain transformation that is, is to define what will happen to the object after the manipulation has taken place: The object might stay suspended mid-air, it might snap to a grid or it might also fall as by gravity until colliding with a fixed object.

Techniques

It is implied on the previous explanations of the selection techniques, that the same metaphors can be applied to manipulation. As such, two unrelated techniques will be explained instead:

HOMER The Hand-Centered Object Manipulation Extending Ray-Casting technique. It uses basic ray-casting for selection, and then moves the virtual hand to the objects position. All manipulation is thereafter referenced around the virtual hand's coordinates. For translation there is a similarity to the arm-extension technique, where a point in the torso is used as reference for translations. A displacement delta of the user's hand would incur in a directly proportional displacement of the object in regards to the torso.

World-in-Miniature This technique provides the user with a small version of the virtual world to interact directly with. Any manipulation that affects the objects of the miniature world will be mapped onto the real objects and scaled appropriately.

2.5.3 Navigation

Navigation is the movement of the user around the virtual world. It is a fundamental human task, of which we need to consider two problems: Travel and Way-finding.

Travel is the low-level, motor component of navigation. It corresponds to the physical representation of moving. Depending on the scenario, this may be related to walking, steering a wheel, or any other form of controlling the position and orientation of the user's viewpoint.

Way-finding is the high-level, cognitive component of navigation. It corresponds to the decision making and planning related to user movement. It involves spatial recognition, path planning, determination of the current position and mental mapping the environment and locations.

The user can perform a short number of subtypes of navigation. They may be, yet again, observed as independent use cases.

Exploration

In an exploration setting, the user has no explicit goal for his movements. The only objective of exploration is to collect information: objects and locations within the world, and building up knowledge of the environment.

The level of exploratory allowance greatly depends on the application's needs. On games, free exploration of a map may be part of the entertainment value; On another environment, focused on performing tasks in a relatively well known world, it might be better to induce search-based goal-directed travel techniques.

Search

Searching involves traveling to a specific goal or target location. On the extreme case, a *naïve search*, resembles exploration in that the user must navigate the world randomly but with a specific final location in mind.

There is no clear distinction between exploration and searching. A user may start the experiment by exploring the environment, until it gathers enough information about its surroundings and is able to guide himself towards the target destination.

Maneuvering

Maneuvering tasks take place in a local area and involve small, precise movements. The most common case, involves the positioning of the viewpoint more precisely in order to perform a particular task. For example, the user may need to examine an object from many angles, or read from a label on the virtual world.

Technically, the best solution for maneuvering tasks is to make use of body and head tracking. It is a straight-forward mapping and intuitive for the user to apply.

Specified Trajectory Movement

Specified Trajectory Movement does not engage the user in a locomotory way. The path to be traversed is previously set, therefore making it unrelated to travel and only to way-finding.

Techniques

In order to achieve these locomotory tasks, certain techniques have been developed. Depending on the situation, each has it advantages and problems. We introduce a few of the most common:

Gaze-Directed Steering Although the term gaze is a little mis-leading, since in most systems no eye-tracking is being performed. The user's gaze direction is inferred from the head's orientation. It is simple to implement but doesn't allow the user to look around the scene while navigating.

Pointing The user's hand orientation is used to continuously specify the direction of motion. Similar to gaze-directed steering, the considerable benefit is that direction of motion can be decoupled from line of sight.

Map-Based Travel A 2D scaled representation of the world is presented to the user. By pointing to the desired destination, the technique would smoothly transport the user to the new position. Several considerations must take place on implementation, such as terrain-height, boundaries, map scale and origin, as well as the smoothness of the transitory animation.

Grabbing the Air Analogous to the physical motion of pulling yourself along a rope, except the rope in the virtual world exists everywhere and can be pulled using hand gestures. For every frame, the movement that must be perform results from finding the transformation between the current hand's position to the previous' frame.

2.6 Summary

We presented a brief history of the CAVE system, with light technical information, highlighting the evolutionary direction that the paradigm has taken. There is a mention to the existence of an outdated system at UPC, whose physical infrastructure can be exploited to build a self-developed CAVE. This fact constitutes UPC's first interest in the continued development of the platform.

The listed software is powerful and multi-purpose. Regardless of the which solution, they carry either: A steep learning curve, in the case of the many APIs, or a very limiting set of features, as is the case for Chrome.

This project is presented as a tailored solution, which strikes closer to our user's needs in flexibility and ease of use.

Chapter 3

Design of the Abstraction Layer

The main purpose of the ALIVE framework is to simplify the development process of any type of visualizer, with different levels of complexity and organization. Some of the most representative projects at CRV implement the following visualizers:

- A Volume Renderer, used to visualize medical data from various sources.
- A Naval Ship Visualizer, that implements a space partition scheme to accelerate rendering of very large models.
- A City visualization application, implementing a quad-tree of relief-impostors[12].

Many other visualizers have been developed. The rendering code for each visualizer is very special-purpose, and yet, many development tasks are generally present present in all of them:

- Input Handling.
- Interaction Implementation.

It would already be very desirable to provide a unified library to handle these two development tasks.

Few of our researchers have dabbled with distributed systems and parallel rendering. From their development experience we gathered the following functional requirements that the framework should fulfill:

- It should completely abstract the developer from the running platform's layout. For an instance, while the CAVE might become the preferred Virtual Environment (VE) for performing user tests, the researcher should still be able to run the same application locally on his workstation for development purposes. Other VR display systems are available at CRV, that serve various levels of portability, immersion and price, the applications should be able to run in any of them with little effort.

- Input devices should be accessible through generic interfaces. It is extremely important that, similarly to the previous point, the application can make use of a Polhemus Fastrack tracker or an Immersion Flock of Birds setup. Though the devices might return different data types, the framework should perform the conversion steps and mask initialization and update procedures.

- Since all of them use different forms of interaction, if the framework is to handle it, allow for different configurations and customizations. Each visualizer has a preferred, or mandatory, method of interaction. Some, like the volume renderer don't require navigation, model visualization usually involves rotating the object to perceive a different perspective. The naval ship visualizer by comparison, provides an experience of ship inspection that is meant to facilitate navigation inside the model, more than one navigation method is implemented then.

- All visualizers use different file specifications for storing and retrieving model data. It was clear that loading models was not something that would be part of the framework. The developers prefer complete control while handling the data. The cities visualizer loads a cache of textures with all its relief impostors. The volume renderer needs to load some sort of volumetric data, most likely in the form of three-dimensional textures.

3.1 Architecture

Lets recount and generalize our problems into three main categories:

1. Abstraction. Throughly described in the introductory chapter, on the background analysis and further defined in the functional requisites we just presented.

2. Interaction. For which we have already stablished that it would be desirable, in terms of code reuse, to provide a shared collection of interaction techniques.

3. Clustering, Parallelism and Device Handling. Encompassing also system synchronization and administration.

Our proposed architecture is a multi-layered approach, specially designed to handle these three problems separately. Figure 3.1 indicates the framework's layers. The dark gray blocks in figure 3.2 represent the layers that have been concretely implemented as part of the current project.

We now define what does each layer mean and how they relate to each other and our problems.

Application The application layer is implemented by each researcher. It is meant as a wrapper class between the rendering libraries and the Abstraction Layer bellow. The sample polygonal renderer uses OpenSceneGraph [13] for generating OpenGL commands. When porting an application, if properly modularized, the rendering code of said application can also be treated as a separate package.

Abstraction With a name so self-explanatory, the only thing that should be made clear is that this layer provides the necessary means of communication between the user code and the backends. The Abstraction layer also defines the interface of communication between the applications and the Interaction Methods. This layer deals then with the first problems of Abstraction and sets the basis for the development of a collection of Interaction Techniques to solve our second problem.

Backend A backend is wrapper code that manages to mask the VR Library from the common ALIVE-based application. The juggler backend hides the implementation details of initializing, updating and controlling application flow with respect to the VR Juggler library. The backends, together with the VR Libraries deal with the issues relating to clustering, parallelism and devices.

Referring back to the tradeoff defined in the introductory chapter:

- Flexibility is provided in our library by the ability to choose between the many VR Libraries for which we can write a backend, this allows for an enormous level of configuration and adaptability.
- The programming experience required to start developing with our library is fairly low. The classes in the Abstraction Layer are well-documented and follow a pattern that should be self-explanatory to computer graphics programmers.
- Only the backend developers are meant to deal with configuration and administration details of a VR Library. This specialization of tasks means that the rest of the team members will not have to spend time acknowledging these problems too.

3.1.1 ALIVE API

For the API design, the chosen methodology was to define an interface that would be analogous to VR Juggler. This ensures that our current users of

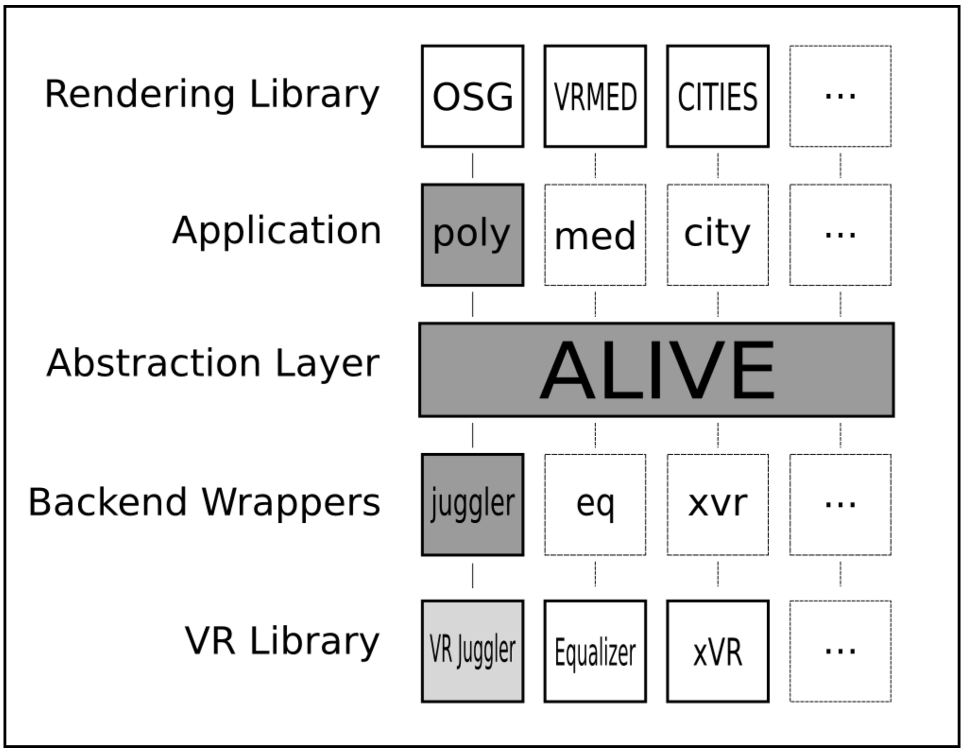


Figure 3.1: Layers of a sample ALIVE Application

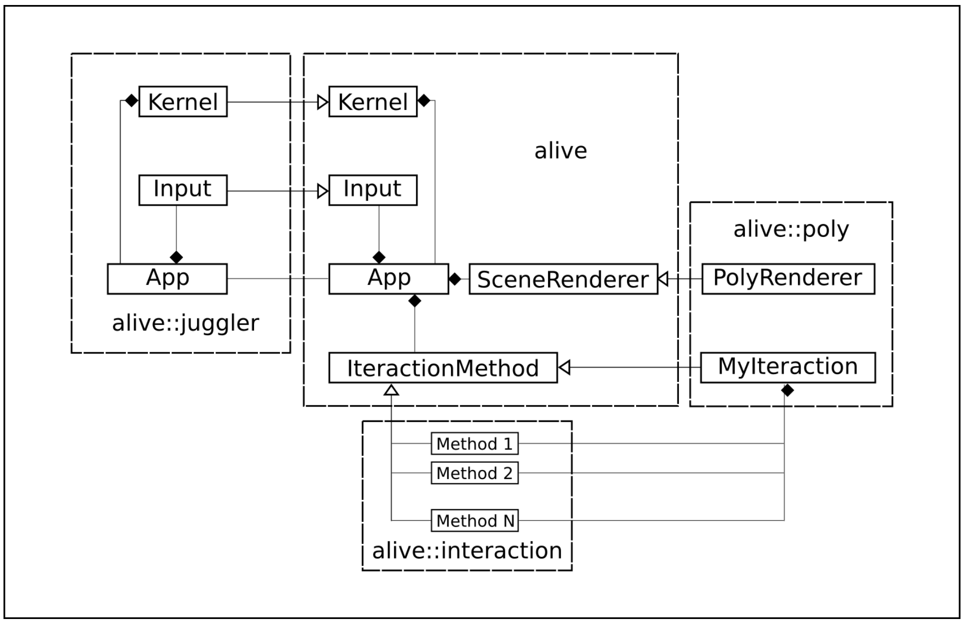


Figure 3.2: Main classes of our implementation

that framework will find it easy to transition their applications to the new API.

Five main classes were defined, of which the user must “fill the gaps” in order to develop an ALIVE application. These classes are inside the “alive” namespace of figure 3.2 and they implement the Abstraction Layer of our architecture.

alive::App Is a concrete implementation and whose methods act as glue code between the others. The methods in this class are placeholders, which will in turn pass the application’s instructions to running backend. The App object is in charge of controlling program flow, it contains references to the other objects, such as the alive::SceneRenderer, the alive::InteractionMethod, and the alive::Input input manager.

Listing 3.1: The main placeholder object enforces application flow

```
void alive::App::contextInit(){
    mSceneRenderer->contextInit();
}
void alive::App::preFrame(){
    mInteractionMethod->update();
}
void alive::App::draw(){
    mSceneRenderer->draw();
}
```

alive::Input The Input Manager class. Has indirect access to hardware and state resources, which is retrievable by the other classes by calling its methods. The abstract Input class has pure virtual methods where they correspond to backend-dependent information. Subclasses of this are the ones in charge of connecting the VR Framework to our API.

Listing 3.2: Provides generic representations for the input

```
float alive::Input::getTimeStamp(){
    return mTimeStamp;
}
bool alive::Input::getButtonState(int number){
    return buttonStates[number];
}
Vec3f alive::Input::getWandDirection(){
    return mWandDirectionVector;
}
```

alive::SceneRenderer The purely abstract alive::SceneRenderer class, is the blank canvas for OpenGL development. Its subclasses must define specific methods to provide the rendering code: Initialization routines, per frame updates, drawing calls.

Listing 3.3: General purpose interface for rendering encapsulation

```
class SceneRenderer{
    ...
    virtual void init() = 0;
    virtual void contextInit() = 0;
    ...
    virtual void draw() = 0;
    ...
}
```

alive::InteractionMethod InteractionMethod subclasses can implement any type of interaction (Navigation, Selection or Manipulation). Although optional to use, the API's design facilitates reuse of interaction code, regardless of the Scene. Through generic data types, the implemented methods are able to communicate the desired interaction to the Scene, which must deal with the data as appropriate for the rendering algorithms.

Listing 3.4: Interaction sample code

```
alive::InteractionMethod::InteractionMethod(int button)
    {
    mButton = button;
}
alive::InteractionMethod::update(){
    if(mInput->getButtonState(mButton)){
        // Update code
    }
}
```

alive::Kernel Is based on the idea of letting the backend handle framework initialization internally. The user instantiates an App object with a proper Scene and optional InteractionMethods, after which a Kernel is instantiated and gives application control to the backend framework's execution loop.

Listing 3.5: Code snippet from main.cpp

```
alive::Kernel* kernel = new alive::juggler::Kernel(
    application);
kernel->startAndWait(argc, argv);
```

3.1.2 The VR Juggler Backend

A backend provides the user with processed data regarding devices, displays and system state. The main implementation of backend functionality lies in subclassing the alive::Input class.

juggler::Input

The backend's juggler::Input defines functions for:

- Initializing and Updating Input Devices. It must convert the data in order to provide a generic representation for a Head and a Wand, as well as button states.
- Retrieving Viewport, Viewing Frustum, ModelView details for the current drawing context.
- Keeping time values for the application: Timestamps.
- It also holds state information, that allows asynchronous communication between the Scene and the InteractionMethod used.

Listing 3.6: VR Juggler specific code

```
void juggler::Input::init(){
    mWand.init(``VJWand'');
    mButton0.init(``Button0'');
    ...
}
int  juggler::Input::getCurrentContext(){
    return vrj::opengl::DrawManager::instance()->
        getCurrentContext();
}
```

VR Juggler was chosen as the first implementable backend, due to its available features and previous experience with it.

The backend must also provide a solution for context-dependent data, for dealing with multi-threaded processes. Each OpenGL generates object ids for data and instructions that are stored in the GPU's memory. If not handled correctly, different threads might overwrite this data which would result in run-time errors. VR Juggler provides a solution with the use of the template object ContextData, which encapsulates values in a hash table and returns the appropriate instance for the correspondent context.

juggler::App

App is the wrapper class, mediating between the backend and the user-facing API described in the previous section. It is not a subclass of alive's App, but rather an implementation of the necessary application structure as defined by the VR Framework (i.e. VR Juggler).

Listing 3.7: VR Juggler specific code

```
void juggler::App::App(alive::App* app){
    mApp = app;
}
```

```
void juggler::App::latePreFrame(){
    mApp->latePreFrame();
}
void juggler::App::draw(){
    mApp->draw();
}
void juggler::App::(){
    mApp->();
}
```

The Kernel object plays the minor role of ensuring proper framework initialization, it is instantiated in the main method of the application. Instantiates the backend's App object and yields control to the framework's control loop.

On instantiation the App receives the user's App object, which contains the applications Scene and InteractionMethods. It is the backend's App job to provide a concrete Input object to the user's App. Proceeding to initialization and appropriately keeping the devices updated.

3.1.3 The Polygon Renderer

The API has already been used in the implementation of an OpenSceneGraph-based polygon renderer. This application serves to demonstrate the API's functionality in allowing the programmer to write compact code that is relevant to his rendering algorithms. A secondary purpose of implementation is that this application should serve as a reference implementation of the platform.

The polygon renderer code is split amongst two classes:

- A SceneRenderer class that interfaces directly with OpenSceneGraph objects. It is responsible for calling the OSG methods appropriately and implement navigation, intersection and object manipulation methods with the OSG Scene. The init method for example creates the graph structure and loads some predefined models:

Listing 3.8: Initing the SceneGraph

```
        void poly::SceneRenderer::init(alive::Input* input)
            {
            alive::SceneRenderer::init(input);
            ...
            mMan = osgDB::readNodeFile(``man.3ds'');
            mManTransf->addChild(mMan);
            mHouse = osgDB::readNodeFile(``house.3ds'');
            mHouseTransf->addChild(mHouse);
            ...
            mNavigationTransf->addChild(mManTransf);
            mNavigationTransf->addChild(mHouseTransf);
            ...
            mSceneGraphRoot->addChild(mNavigationTransf);
```

```
}
```

A SceneViewer object is used by OSG to provide visualization capabilities of the SceneGraph we just defined.

Listing 3.9: Initing the Context-Dependent Objects

```
void poly::SceneRenderer::contextInit(){
    // mSceneViewer has two levels of indirection,
        *mSceneViewer is the real pointer to the
        object.
    *mSceneViewer = new osgUtil::SceneView();
    ...
    // set lightning, clear color and other OpenGL
        context initialization code for our
        sceneViewer object.
    ...
    *mSceneViewer->addSceneData(mSceneGraphRood);
}
```

Drawing the scene entails setting the current Viewport, ModelView and Projection Matrices, applying any last minute scene processing and sending the rendering calls:

Listing 3.10: Initing the Context-Dependent Objects

```
void poly::SceneRenderer::draw(){
    *mSceneViewer->setViewport( mInput->getViewport
        () );
    *mSceneViewer->setProjection( mInput->
        getProjection() );
    *mSceneViewer->setViewMatrix( mInput->
        getViewMatrix() );
    ...
    *mSceneViewer->cull();
    ...
    *mSceneViewer->draw();
}
```

- A Sample InteractionMethod. Or rather a combination of methods, it implements the abstract alive::InteractionMethod but also possesses some alive::InteractionMethod references. Since this is supported by the language, it becomes a powerful way to combine Interaction Techniques. With continued development, it is possible that some of these combination techniques become popular and therefore worthy of joining the collection of single purpose methods, this would simplify yet another step of development.

Listing 3.11: Mixing Interaction Techniques

```
void poly::MyInteraction::MyInteraction(int button)
    {
        selection = new alive::interaction::
            RayCastingSelection(0);
        manipulation new alive::interaction::
            SimpleManipulation(1);
        navigation = new alive::interaction::
            DefaultNavigation(2);
    }
    void poly::MyInteraction::update(){
        selection->update();
        manipulation->update();
        navigation->update();
    }
```

3.1.4 Interaction Flow

The flow of Interaction requires some special attention, in order to ensure the proper communication between the SceneRenderer and the Interaction-Methods. We will keep using the classnames of our sample polygon renderer application.

Navigation is the most straightforward to implement. Listing 3.8 defined the graph structure. The navigation matrix starts as an identity matrix. The navigation method then modifies the navigation matrix and lets the poly::SceneRenderer know by means of the alive::Input object.

Listing 3.12: Navigation

```
void alive::interaction::BasicNavigation::update(){
    navigationMatrix = mInput->getNavigationMatrix();
    ...
    // modify navigation matrix
    ...
    mInput->setNavigationMatrix( navigationMatrix );
}
void poly::SceneRenderer::latePreFrame(){
    ...
    mNavigationTransf->setMatrix( mInput->
        getNavigationMatrix() );
    ...
}
```

Selection and Manipulation are slightly more complicated ventures, the selection method is fairly simple though. We just need to define the geometry that we will use to intersect with the scene. In this case a simple ray that points in the direction of the wand.

Listing 3.13: Selection Method

```
void alive::interaction::RayCastingSelection::update(){
    ...
    rayStart = mInput->getWandPosition();
    rayEnd = rayStart + mInput->getWandDirection();
```

```
        ...
        mInput->setRayStart( rayStart );
        mInput->setRayEnd( rayEnd );
    }
```

The code in the Manipulation method is, conceptually, also very simple. But, now we do have to deal with 2 flag variables: A button state that will determine if we are going to try and send manipulation instructions if there is a selected object currently.

Listing 3.14: Manipulation Method

```
    void alive::interaction::RayCastingSelection::update(){
        if(mButton && mInput->isObjectSelected() ){
            ...
            objectTransformation = mInput->
                getSelectedObjectTransformation();
            // We now use the data from the tracker devices to
                modify the transformation matrix for the selected
                 object.
            ...
            // And we give it back along with a flag telling the
                 SceneRenderer that it should update the matrix.
            mInput->setSelectedObjectTransformation(object);
            mInput->applyManipulation(true);
        }
        else mInput->applyManipulation(false);
    }
```

The SceneRenderer code carries the burden of having to handle interaction states. Our update interaction method should follow the next structure and be called inside the SceneRenderer's latePreFrame method.

Listing 3.15: Selection/Manipulation Handling by the SceneRenderer

```
    // In our case, if no ray has been casted, there is no need
        to try any further.
    if( mInput->getRayCasted() ){
        if( mInput->intersectionCheck() ){
            ...
            rayStart = mInput->getRayStart();
            rayEnd = mInput->getRayEnd();

            // Intersect the Scene with the ray defined by the
                above points.

            if( intersectionFound ){
                node = intersection->getNode();
                transfNode = node->getTransform();

                mInput->setObjectIntersected(true);
                mInput->setSelectedObjectTransformation(
                    transfNode->getMatrix() );
            }
        }
        else mInput->setObjectIntersected(false);

        if( mInput->getApplyManipulation() ){
            transfNode->setMatrix( mInput->
                getSelectedObjectTransformation() );
        }
    }
```

Some other application might prefer to always execute some form of manipulation, if for an instance they want to make the movement of an object dependent on the wand's orientation. We cannot implement interaction flow in the abstraction layer since it would disallow some forms of interaction flow.

3.2 Results

We have implemented an OSG-based polygon renderer that uses our library to gain access to an in-development CAVE system. The generated code is very compact and is mostly contained in a single SceneRenderer sub-class.

With approximately 200 lines of code this application:

- integrates head and hand tracking.
- Uses a specialized interaction scheme.
- Can be displayed on a multi-projector, multi-PC synchronized system.

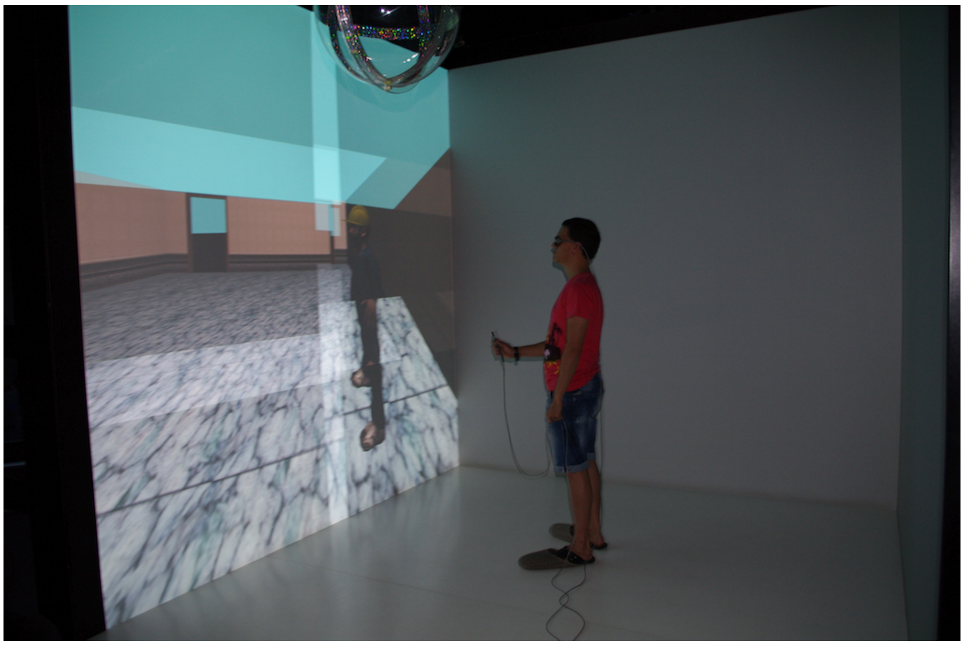


Figure 3.3: A user, testing the input devices

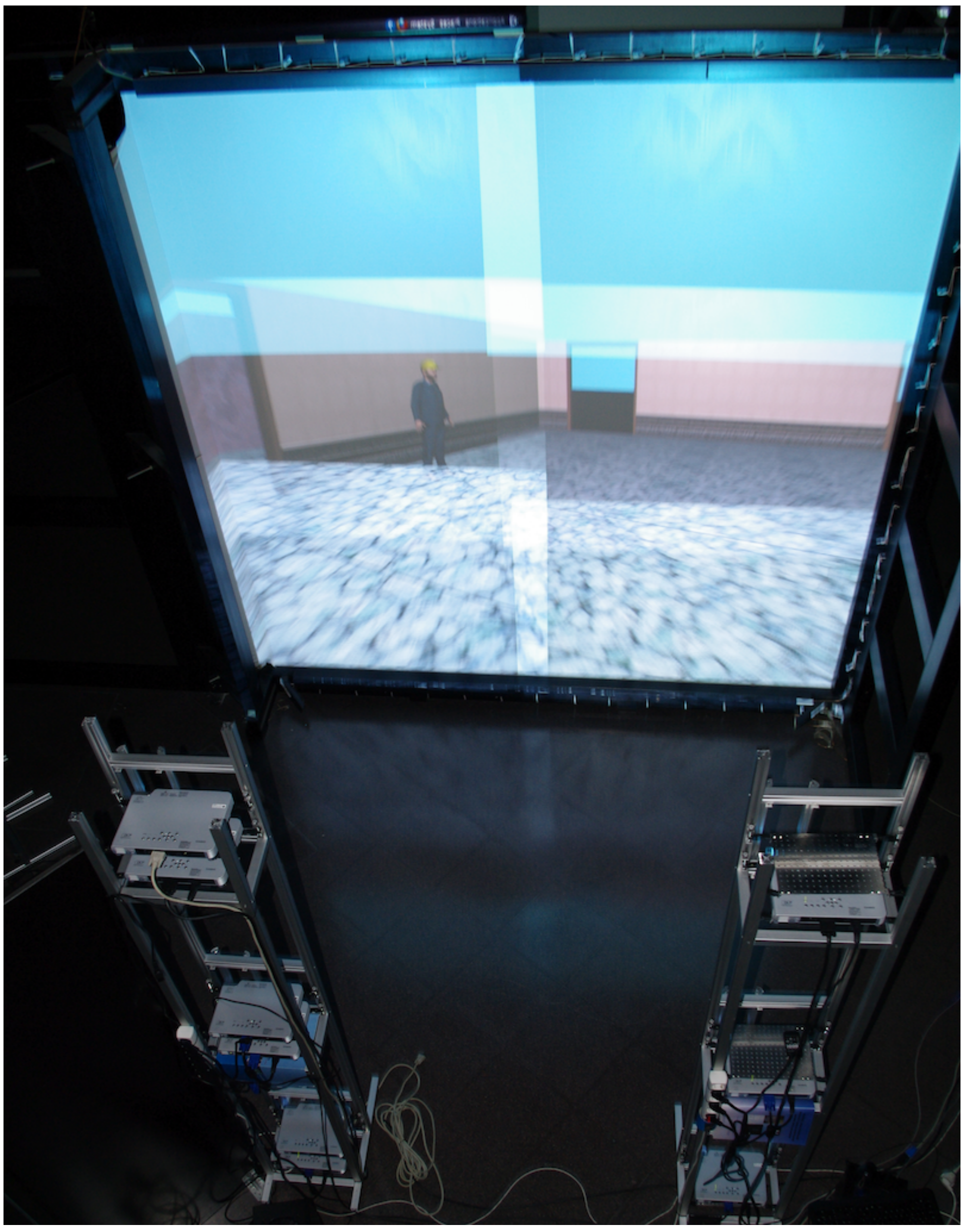


Figure 3.4: Left Wall of the CAVE

Chapter 4

Project Management

This appendix describes the planning as it was followed during the research and development of this thesis project. It should be noted that due to unexpected technical circumstances, it was impossible to adhere to the planned schedule. Where appropriate or relevant, comments will be made in regards to plan deviation.

Figure 4.1 shows the Gantt diagram for the main block of development time, during the months of July and August of 2010. Previous to this, research on the state of the art and tests were done, however they are not accounted in the final diagram and cost estimation.

Development did not proceed as planned. The complexity of the hardware and software used, generated unforeseen time sinks that largely disturbed the project workflow.

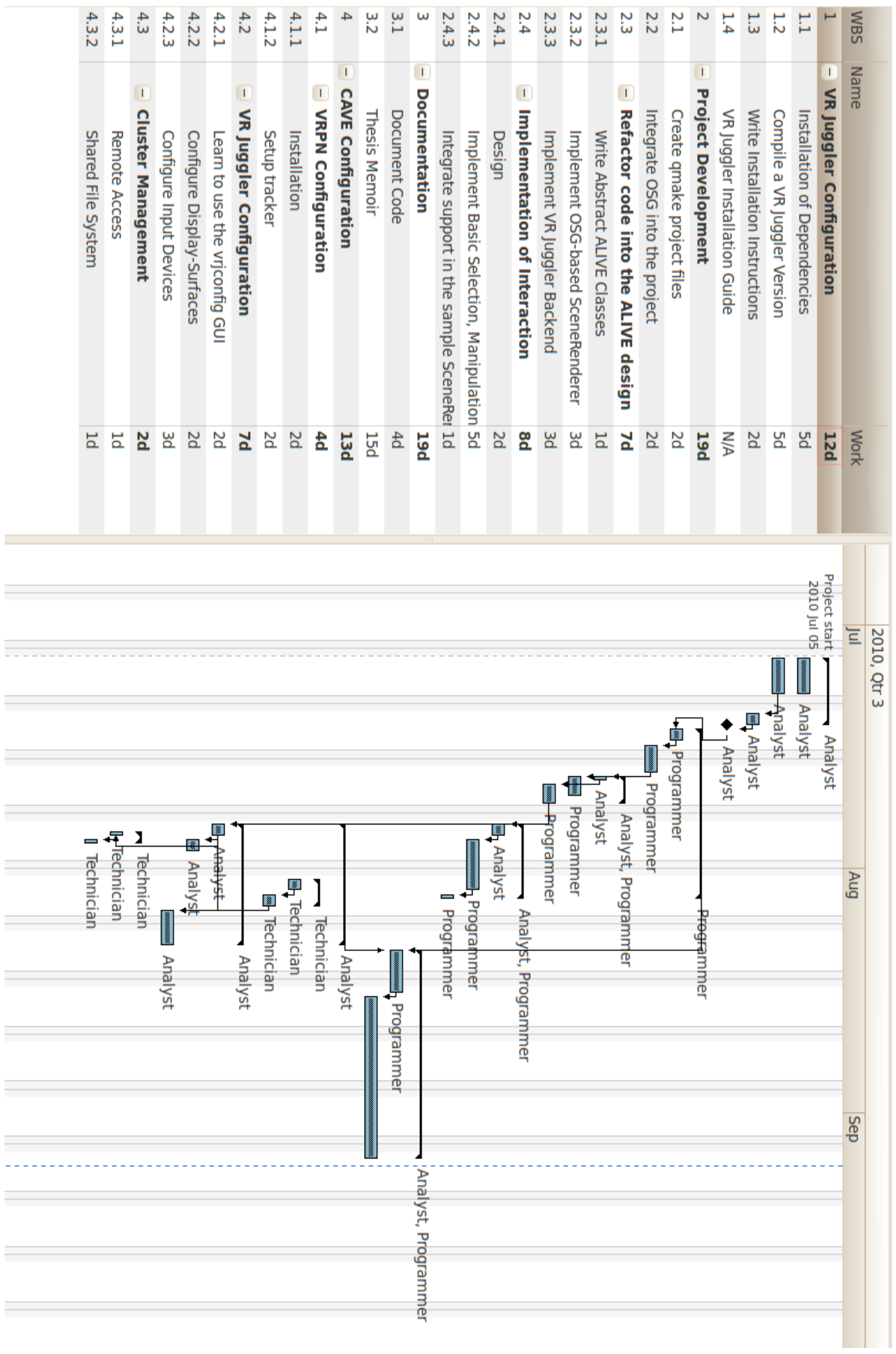
VR Juggler proved to be far more complex to compile than initially expected. Instead of the one week suggested for overcoming this chore, we were plagued by one annoying bug in the VR Juggler build system: It would not detect the presence of VRPN in our system and therefore would not build the required plugin.

It would seem that bugs were masking one another. Once the build system bug was fixed, another one appeared inside the VRPN driver, related to a recent redesign of the VR Juggler gadget interface. Apparently, nobody had noticed it because they had not tried to compile the VRPN driver.

When everything seemed solved and the driver was properly generated, three issues began showing during our testing:

1. The timestamps that we retrieved from the devices were zero-valued all the time, therefore incorrect. Another bug was reported to the VR Juggler user's list and it was solved in due time.

2. The VRPN buttons are reported as being pressed upon initialization, by the VRPN driver. A bug has been filed to the aforementioned user list, but a solution is still pending.



WBS	Name	Work
1	**VR Juggler Configuration**	**12d**
1.1	Installation of Dependencies	5d
1.2	Compile a VR Juggler Version	5d
1.3	Write Installation Instructions	2d
1.4	VR Juggler Installation Guide	N/A
2	**Project Development**	**19d**
2.1	Create qmake project files	2d
2.2	Integrate OSG into the project	2d
2.3	**Refactor code into the ALIVE design**	**7d**
2.3.1	Write Abstract ALIVE Classes	1d
2.3.2	Implement OSG-based SceneRenderer	3d
2.3.3	Implement VR Juggler Backend	3d
2.4	**Implementation of Interaction**	**8d**
2.4.1	Design	2d
2.4.2	Implement Basic Selection, Manipulation	5d
2.4.3	Integrate support in the sample SceneRei	1d
3	**Documentation**	**19d**
3.1	Document Code	4d
3.2	Thesis Memoir	15d
4	**CAVE Configuration**	**13d**
4.1	**VRPN Configuration**	**4d**
4.1.1	Installation	2d
4.1.2	Setup tracker	2d
4.2	**VR Juggler Configuration**	**7d**
4.2.1	Learn to use the vrjconfig GUI	2d
4.2.2	Configure Display-Surfaces	2d
4.2.3	Configure Input Devices	3d
4.3	**Cluster Management**	**2d**
4.3.1	Remote Access	1d
4.3.2	Shared File System	1d

Figure 4.1: Planification of this Thesis Project

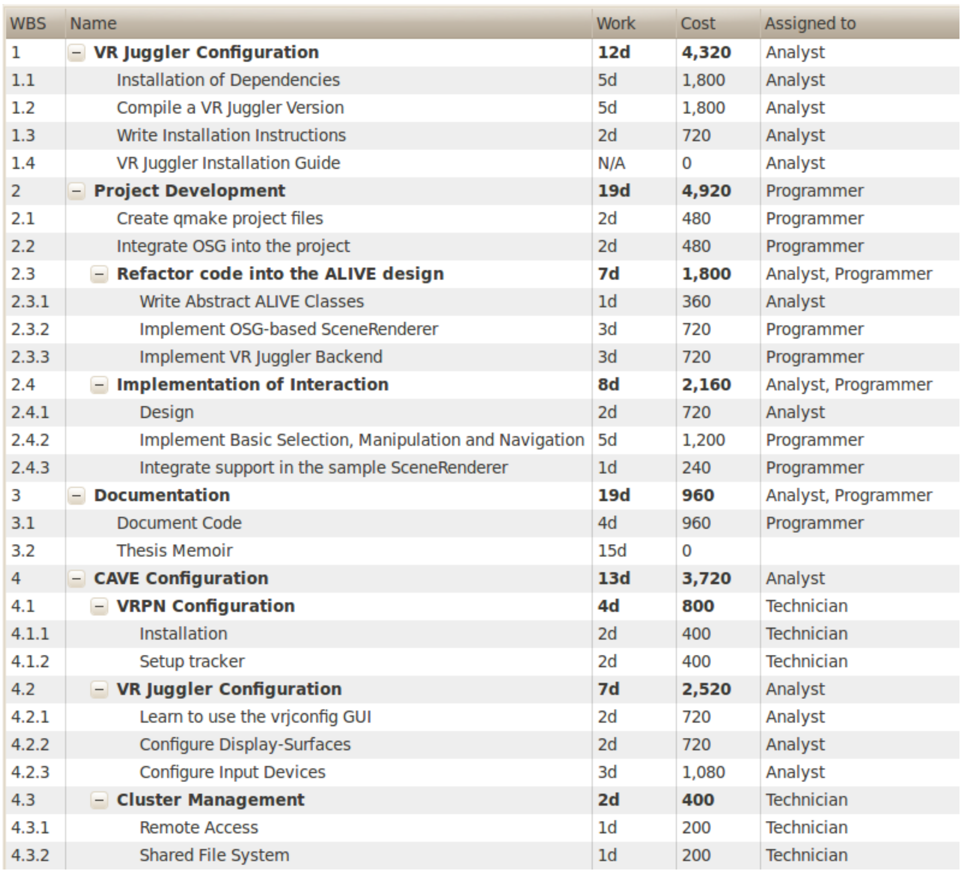
3. Our tracker seems to have initialization issues. It needs be reset a few times before reaching a correct init state. Often times it will return zero or infinity values, not detect the connected sensors or deliver an overly jittery sensor data.

This unfortunate combination of bugs pushed back development for about two weeks. Determining the identity of each bug was challenging due to the overlapping output they produced.

The issues with the hardware were the most unexpected and frustrating, since every morning it seemed that work had become undone during the night.

Economic Analysis

Considering the initial time allotment for tasks and standard fees for the development services, the estimated cost of development stands close to 14000, as per the analysis in figure 4.2



WBS	Name	Work	Cost	Assigned to
1	**VR Juggler Configuration**	**12d**	**4,320**	Analyst
1.1	Installation of Dependencies	5d	1,800	Analyst
1.2	Compile a VR Juggler Version	5d	1,800	Analyst
1.3	Write Installation Instructions	2d	720	Analyst
1.4	VR Juggler Installation Guide	N/A	0	Analyst
2	**Project Development**	**19d**	**4,920**	Programmer
2.1	Create qmake project files	2d	480	Programmer
2.2	Integrate OSG into the project	2d	480	Programmer
2.3	**Refactor code into the ALIVE design**	**7d**	**1,800**	Analyst, Programmer
2.3.1	Write Abstract ALIVE Classes	1d	360	Analyst
2.3.2	Implement OSG-based SceneRenderer	3d	720	Programmer
2.3.3	Implement VR Juggler Backend	3d	720	Programmer
2.4	**Implementation of Interaction**	**8d**	**2,160**	Analyst, Programmer
2.4.1	Design	2d	720	Analyst
2.4.2	Implement Basic Selection, Manipulation and Navigation	5d	1,200	Programmer
2.4.3	Integrate support in the sample SceneRenderer	1d	240	Programmer
3	**Documentation**	**19d**	**960**	Analyst, Programmer
3.1	Document Code	4d	960	Programmer
3.2	Thesis Memoir	15d	0	
4	**CAVE Configuration**	**13d**	**3,720**	Analyst
4.1	**VRPN Configuration**	**4d**	**800**	Technician
4.1.1	Installation	2d	400	Technician
4.1.2	Setup tracker	2d	400	Technician
4.2	**VR Juggler Configuration**	**7d**	**2,520**	Analyst
4.2.1	Learn to use the vrjconfig GUI	2d	720	Analyst
4.2.2	Configure Display-Surfaces	2d	720	Analyst
4.2.3	Configure Input Devices	3d	1,080	Analyst
4.3	**Cluster Management**	**2d**	**400**	Technician
4.3.1	Remote Access	1d	200	Technician
4.3.2	Shared File System	1d	200	Technician

Figure 4.2: Development Costs

All of the software used in this project is available through free software licenses. Hardware cost are also not considered in the economic analysis.

Chapter 5

Conclusions

A large amount of the time dedicated to this project was allotted to troubleshooting of devices, libraries and installation instructions. It is my opinion that these are tedious tasks that should be performed by a minimum of number of researchers. While most developers have the necessary skills to solve these problems, it is not productive to dedicate time to these issues. The backend layer has a secondary, but transcendental purpose, regarding this issue. Only the framework developers have to deal with the deployment issues. Once the wrinkles for a specific VR library have been ironed out, the new backend may be released to unsuspecting that ideally should only realize a change has been applied due to performance enhancements or other differences.

Of relative importance, it should also be noted that contributions to the VR Juggler project were made throughout the testing of the library. Bug reports were filed through their mailing list. Two of the bugs reported are listed out of the total of eight fixes responsible for reaching Release Candidate state for their long-awaited 3.0 release. Further collaboration between these projects would be a highly desired result, as it would ensure further improvement of both platforms.

Comparing our initial objectives to the achieved results:

- Abstraction from the common VR issues is provided by the VR Juggler backend. With the noted exceptions of 3D Sound Rendering and non-deterministic algorithms.

- The sample application has been tested on the CAVE. Complete rehabilitation of the CAVE still depends on porting of applications and technical issues unrelated to this project.

- The integration of OSG with ALIVE is completely contained within one class of the polygon renderer sample application. All in all, the code is around 200 lines of not too dense code, introducing almost zero programming overhead. My personal opinion is that these metrics help demonstrate that, porting well-designed applications should be fairly straightforward.

- CAVE configuration has already been performed and is well-documented. It should be easy to replicate it by following the instructions in Appendix A.

As demonstrated by the OSG-based sample polygon renderer and the VR Juggler backend, ALIVE-based applications:

- Will be written in, very compact, render-specific code.
- Are ready to run in our CAVE setup. And also any other configuration configurable by VR Juggler, such as the existing PowerWall or WorkBenches at CRV.

The desired benefits of this project will become even more tangible shortly:

- Our researchers will start porting applications. This is due to happen very soon, after some instructive sessions to demonstrate the API features.
- A comprehensive library of Interaction Methods gets implemented. Which should help the developers by semi-automatically, defining very controllable interaction schemes.

Future Work

Summarizing the road that is still ahead of us, we need to:

- Implement more Backends. VR Juggler is highly capable, there are however other frameworks that display great amount of potential for integration:
  - Equalizer. Whose great advantage over VR Juggler is the ability to parallelize the rendering pipeline by many methods. VR Juggler only allows sort-first parallelization by replication.
  - A VRPN backend. Although it would not provide any rendering functionality, could prove essential in bringing device-abstraction to other render-only libraries such as Equalizer.
  - xVR, which is also used by some of our group members.
  - Theoretically, the abstraction level of this library allows for integration with non-VR libraries. It would be interesting to explore the possibility of integrating a rendering framework for mobile devices.

  - Support for Sound Rendering is not yet included in the abstraction layer. 3D Sound Rendering is desired, to complete the immersion experience.
  - A solution for distributed random number generator should be implemented. This would allow some basic support for non-deterministic algorithms.

- Start porting the existing CRV applications.
- Implement a comprehensive library of Interaction Techniques.

Bibliography

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

[2] Thomas A. DeFanti, Gregory Dawe, Daniel J. Sandin, Jurgen P. Schulze, Peter Otto, Javier Girado, Falko Kuester, Larry Smarr, and Ramesh Rao. The starcave, a third-generation cave and virtual reality optiportal. *Future Generation Computer Systems*, 25(2):169 – 178, 2009.

[3] Bruno Raffin and Luciano Soares. Pc clusters for virtual reality. In *VR '06: Proceedings of the IEEE conference on Virtual Reality*, pages 215–222, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Thomas W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23(7):819 – 843, 1997. Parallel graphics and visualisation.

[5] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. Technical report, Chapel Hill, NC, USA, 1994.

[6] Vr juggler homepage. `http://vrjuggler.org/`.

[7] Chromium homepage. `http://chromium.sourceforge.net/`.

[8] Equalizer homepage. `http://equalizergraphics.com/`.

[9] Opensg homepage. `http://www.opensg.org/`.

[10] Flowvr homepage. `http://flowvr.sourceforge.net/`.

[11] Doug A. Bowman, Ernst Kruijff, Joseph J. LaViola, and Ivan Poupyrev. *3D User Interfaces: Theory and Practice*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[12] C. Andujar, P. Brunet, A. Chica, and I. Navazo. Visualization of large-scale urban models through multi-level relief impostors. *Computer Graphics Forum*, 9999(9999), 2010.

[13] Openscenegraph homepage. `http://www.openscenegraph.org/`.

[14] Chadwick A Wingrave and Joseph J LaViola. Reflecting on the design and implementation issues of virtual environments. *Presence: Teleoperators and Virtual Environments*, 19(2):179–195, 2010.

[15] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[16] Mechdyne. Cavelib homepage. `http://www.mechdyne.com/integratedSolutions/software/products/CAVELib/CAVELib.htm`.

Appendix A

Installation Instructions

Due to the many dependencies and the complexity of the development environment, we provide the following installation and configuration instructions.

The cluster administrator is expected to have sufficient gnu/linux experience and to be comfortable working with the command line.

A.1 Installation

A.1.1 Ubuntu pre-compiled packages

After a clean installation of ubuntu, the Operating System doesn't provide us with the required development environment. Here is a short description of what each of the required packages provides.

build-essential The essential development environment, according to the ubuntu devs. It contains the C, C++ compilers and gnu make.

subversion and git-core The SVN and Git packages you will need to download VR Juggler and ALIVE respectively.

automake1.9 Required version of automake.

scons A replacement for the autotools, needed to install some of the packages.

openjdk-6-jdk Java for the GUI-based configuration applications of the VR Juggler suite.

python2.6-dev The development headers for python.

libboost* The boost distribution of C++ libraries.

qt4-dev-tools The ALIVE libraries are built using qmake, and will soon add QT features.

libopenscenegraph-dev OSG libraries, necessary for the sample polygonal renderer.

qtcreator An optional development environment. Since we are already going to use the QT libraries. It also provides good integration with qmake and uses it as a its project file.

Type the following commands on your terminal to install everything on the list.

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt-get install build-essential subversion git-core \
automake1.9 scons openjdk-6-jdk python2.6-dev \
libboost-dev libboost-dbg libboost-doc \
libboost-filesystem-dev libboost-signals-dev \
libboost-program-options-dev qt4-dev-tools libopenscenegraph-dev
    \
qtcreator
```

Note that "$" is the command prompt and "\" is the the line break which allows to split the input of the long command. The qtcreator package is optional.

A.1.2 Manual Compilation Packages

Some dependencies could not be satisfied by the ubuntu repositories. Therefore, download the latest versions of:

CPPDOM http://sourceforge.net/projects/xml-cppdom/files/

GMTL http://sourceforge.net/projects/ggt/files/

Flagpoll http://code.google.com/p/flagpoll/downloads/list

Doozer http://sourceforge.net/projects/doozer/files/

VRPN ftp://ftp.cs.unc.edu/pub/packages/GRIP/vrpn/

We recommend to install all of the following libraries into a single folder structure. Since later they will have to be addressed to the compiler and linker. Also recommendable is to maintain the compiled objects separate from the sources.

```
$ mkdir -p $HOME/Share/juggler $HOME/Workspace/juggler
```

The folder in Workspace will the used for the sources and we will use the Share folder for file sharing between the nodes.

Now, we will create a temporary environment variable to facilitate giving the installation directory to each package. Bear in mind that if you close your terminal after this step, if you want to resume the installation at any step, you will have to set this again.

```
$ export VJ_BASE_DIR=$HOME/Share/juggler
```

We are ready to start compiling the dependencies. Unpack the downloaded dependencies to the juggler folder in Workspace.

CPPDOM

Change to the cppdom directory.

```
$ cd $HOME/Workspace/juggler/cppdom-1.0.1
```

Scons is having a problem detecting the correct architecture of the installed ubuntu system. Therefore there is a slight difference in the following command if you are using a 32 bit system or a 64 bit system.

For 32 bit systems type:

```
$ scons install prefix=$VJ_BASE_DIR var_arch=ia32
```

For 64 bit systems type:

```
$ scons install prefix=$VJ_BASE_DIR var_arch=x64
```

GMTL

A little simpler than the above.

```
$ cd $HOME/Workspace/juggler/gmtl-0.6.0
$ scons install prefix=$VJ_BASE_DIR
```

Flagpoll

Change to directory and install:

```
$ cd $HOME/Workspace/juggler/flagpoll-0.9.1
$ python setup.py install --prefix=$VJ_BASE_DIR
```

Flagpoll is used by the VR Juggler tools to help configure their build system according to the installed dependencies. There are some variables that need to be set now. Remember the above terminal regarding the issue caused by closing the terminal after this point. There is also a slight difference of commands between architectures.

For 32 bit systems type:

```
$ export ACLOCAL_FLAGS="-I $VJ_BASE_DIR/share/aclocal/"
$ export FLAGPOLL_PATH=$VJ_BASE_DIR/share/flagpoll\
:$VJ_BASE_DIR/lib/flagpoll\
:$VJ_BASE_DIR/juggler/lib/debug/flagpoll
$ export PATH="$VJ_BASE_DIR/bin:$PATH"
```

For 64 bit systems type:

```
$ export ACLOCAL_FLAGS=''-I $VJ_BASE_DIR/share/aclocal/"
$ export FLAGPOLL_PATH=$VJ_BASE_DIR/share/flagpoll\
:$VJ_BASE_DIR/lib64/flagpoll\
:$VJ_BASE_DIR/juggler/lib64/debug/flagpoll
$ export PATH="$VJ_BASE_DIR/bin:$PATH"
```

Doozer

Doozer also helps in the configuration/installation process of VR Juggler.

```
$ cd $HOME/Workspace/juggler/Doozer-2.1.6
$ ./configure --prefix=$VJ_BASE_DIR
$ make install
```

VRPN

The Virtual Reality Peripheral Network is used to facilitate integration of trackers, wands and other VR devices. VR Juggler detects its presence on the system and builds an input plugin for it.

The VRPN build system requires editing the Makefiles of its different subprojects, in order to specify your system architecture. Inside each Makefile, amongst the first line you must uncomment the appropriate line, defining the HW_OS as pc_linux for 32 bit systems or pc_linux64 for 64 bit systems.

Note that depending on the tool you use for unzipping the vrpn package (GUI or CLI), you might get both quat and vrpn inside another folder.

First install the quat library, change to the quat directory:

```
$ cd $HOME/Workspace/juggler/quat
```

Remember to modify the Makefile, you may use any text editor of your choice.

```
$ make
$ sudo make install
```

Now onto proper VRPN:

```
$ cd $HOME/Workspace/juggler/vrpn
$ make
$ sudo make install
```

There is a further step needed in order to allow VR Juggler to correctly detect the presence of VRPN in our system. The crude installation system of VRPN doesn't set the required symbolic links. We must therefore create them.

```
$ cd /usr/local/lib
$ sudo ln -s libquat.a libquat
$ sudo ln -s libvrpn.a libvrpn
$ sudo ln -s libvrpnatmel.a libvrpnatmel
$ sudo ln -s libvrpnserver.a libvrpnserver
```

Since configuration of individual VRPN devices is out of the scope of this document. We recommend heading to vrpn's homepage and learn how to configure your devices, as well as starting a vrpn server. Documentation is also available with the vrpn source code you downloaded.

A.1.3 VR Juggler

Now that we have satisfied the smaller dependencies, it's time to install VR Juggler. There are two lengthy steps required, downloading the latest SVN code takes a few minutes depending on your connection, and compilation might take about 15 minutes.

Change to sources directory and download the VR Juggler source code:

```
$ cd $HOME/Workspace/juggler
$ svn co http://vrjuggler.googlecode.com/svn/juggler/trunk
    juggler
```

Go turn on your coffee machine.

Change to juggler's source directory. Run the autogen script and configure the build inside a separate folder.

```
$ $HOME/Workspace/juggler/juggler
$ ./autogen.sh
$ mkdir build
$ cd build
$ ../configure.pl --with-boost-includes=/usr/include \
--with-gmtl=$VJ_BASE_DIR \
--with-flagpoll=$VJ_BASE_DIR/bin/flagpoll \
--with-vrpn=/usr/local --prefix=$VJ_BASE_DIR
```

VR Juggler is ready for compilation. By now your cup of coffee should be ready too. Type the following command and do something else for the next 15 minutes:

```
$ make build install >& install.log
```

The output of the make command is being forwarded to an install log file. You can view it interactively by typing the following into another terminal:

```
$ tail -f $HOME/Worskpace/juggler/juggler/build/install.log
```

When make is done, if you followed the instructions correctly (the coffee part is not mandatory), VR Juggler should be installed into the shared directory we defined at the beginning.

A.1.4 ALIVE

For the following steps we use qmake, all are pretty straightforward. The only decisions being the definition of a home folder for the libraries and the correct compilation order based on the dependencies.

First we need to define the home folder for ALIVE. Analogous to the folder we chose for VR Juggler and its dependencies, we will create and set:

```
$ mkdir -p $HOME/Share/alive
$ export ALIVE_HOME=$HOME/Share/alive
```

Now we need the source code. It is available in a github repository.

```
$ cd $HOME/Workspace
$ git clone git://github.com/jtibau/alive.git
```

Compiling all of the enclosed folders involves the same exact procedure, we just have to be careful about the build order.

First we need to change into the appropriate project folder. Each project has a .pro file inside. Then we enter the following commands:

```
$ qmake PREFIX=$ALIVE_HOME
$ make install
```

The preferred building order is:

1. The abstract classes, in the subfolder alive.
2. The Interaction Methods, inside the subfolder interaction
3. The VR Juggler backend wrapper, inside the juggler subfolder.

Now everything is installed in your shared folder. You could start developing your own application now, or test the installation with the sample polygonal renderer.

A.1.5 The Sample Polygonal Renderer

After installing the abstract libraries, the provided interaction methods and the VR Juggler backend, you have all the necessary tools to build your own alive-based application.

The building procedure is the same as for the previous components. Remember by this point you still need the environment variables set earlier.

```
$ cd $HOME/Workspace/alive/poly
$ qmake PREFIX=$ALIVE_HOME
$ make install
```

You will need to add the another environmental variable to tell the system where to look for the dynamic link libraries in your system.

Refer to the configuration section of this appendix to learn how to permanently configure your system for development.

```
$ export LD_LIBRARY_PATH=$VJ_BASE_DIR/lib:$VJ_BASE_DIR/lib64
```

You should also set the path that OSG should use to find the resources your models will need. We recommend putting the models inside a data directory on the shared folder. If you don't, OSG will render only the meshes with no textures, but will still work.

```
$ mkdir -p $HOME/Share/data
$ export OSG_FILE_PATH=$HOME/Share/data
```

The PREFIX variable appended to the qmake command ensures that upon installation, the produced files go to the shared folder's alive directory.

In order to run the polygonal renderer, with a standalone configuration, type the following commands (replacing modelname with your own model filename):

```
$ cd $HOME/Share
$ ./alive/bin/poly ./data/modelname standalone.jconf
```

You should now be running the polygonal rendering with a standalone VR Juggler configuration.

A.2 Configuration

As mentioned previously, there are a few configuration steps you may want to perform in order to ease your day to day development tasks:

- Provide a way to access the nodes remotely. This is fairly important in terms of comfort, as you will be able to administer and run the nodes remotely from a single machine.
- Set the environment variables as a permanent configuration, so you don't have to do it every time.
- Configure the shared folder for the cluster, or else you would have to find another way to distribute application changes between cluster nodes.

A.2.1 Remote Access to the Nodes

You may have reached this point configuring only the master node of the cluster. If you are not developing cluster-based system and only want to use ALIVE as an abstraction library, you may skip this and the shared folder subsections.

If you do want to configure a cluster, remote access is probably a highly desired feature. We use OpenSSH for this. It is very easy to configure in ubuntu and it should be comfortable and straightforward to use for any command line user.

At this point, you will most likely have not installed anything on your slave nodes. If appropriate, perform a clean Ubuntu 10.04 installation, use the same architecture for all nodes.

For sanity's sake, it might also be important to introduce an easy to remember naming scheme for the cluster nodes: node1, node2, node3. It is also recommended that you use the same, generic, user name in all machines.

The installation package for OpenSSH is available in the ubuntu repositories. The slave doesn't need to operate an ssh server and the client is installed in the default ubuntu installation. Be sure you have an up-to-date installation and type de following command to install the server:

```
$ sudo apt-get install openssh-server
```

If you have connectivity between your nodes, you can now move away from the nodes and do all your work from the single master node.

Lastly, there is an easy step that you can perform so that every time you login to the one of the nodes it doesn't ask you for a password. You must first generate an rsa key pair in the master node, leave an empty passphrase when asked.

```
$ ssh-keygen
```

Then for each node, type the following command replacing N for the node number (or using the naming scheme of your choice):

```
$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub username@nodeN
```

Now if you want to access one of your nodes you can type on a terminal:

```
$ ssh username@nodeN
```

It won't ask you for a password. The output will show a welcome message and change the prompt to reflect that your are in nodeN.

A.2.2 Setting the Environment

Configuration of these variables is fairly easy. We just need to append the export lines to the end of the bash configuration file.

We recommend editing your personal file. You may use gedit, vi or the text editor of your choice, append the following lines:

```
export VJ_BASE_DIR=$HOME/Share/juggler
export ALIVE_HOME=$HOME/Share/alive

export ACLOCAL_FLAGS=''-I $VJ_BASE_DIR/share/aclocal/''
export FLAGPOLL_PATH=$VJ_BASE_DIR/share/flagpoll\
:$VJ_BASE_DIR/lib64/flagpoll\
:$VJ_BASE_DIR/juggler/lib64/debug/flagpoll
```

```
export PATH="$VJ_BASE_DIR/bin:$PATH"

export JDK_HOME=/usr
export LD_LIBRARY_PATH=$VJ_BASE_DIR/lib\
:$VJ_BASE_DIR/lib64\
:$ALIVE_HOME/lib:$ALIVE_HOME/lib

export OSG_FILE_PATH=$HOME/Share/data
```

That's all there is to it.

A.2.3 Shared Folder

We provide instructions for configuring the Network File System protocol, as it is the preferred way for linux systems. You may also configure samba/cifs if you know your way around it. It is however crucial that you use the same folder, as we have used it previously during the installation and environment configuration steps.

As mentioned before, the purpose of the shared folder is not having to compile everything for each node. Since each node pulls the latest object files from the same place in the network. Keep in mind that depending on your data you may need a very fast connection.

The NFS Server

The master node should be the server. First install the nfs server:

```
$ sudo apt-get install nfs-kernel-server
```

Create the shared folder, it should actually be outside your home so create it:

```
$ sudo mkdir -p /srv/Share
```

Now we need to change a few configuration files. With the editor of your choice, edit: The /etc/default/nfs-kernel-server file by finding the following flag and setting it to no:

```
NEED_SVCGSSD=no
```

The /etc/default/nfs-common file should have the following:

```
NEED_IDMAPD=yes
NEED_GSSD=no
```

Lastly, append the line that defines our share to the /etc/exports file. Your network IP and mask could may look like this: 192.168.0.0/24.

```
/srv/Share network_ip/mask(rw,sync,no_subtree_check,
    no_root_squash)
```

You can now restart the server to apply the changes we just made:

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

The NFS Clients

All nodes, including the master nodes, should apply the following configurations.

Although the master node already possesses the shared folder, remember it is not as of yet mounted on the correct position. For simplicity, you should just treat is a regular node in regards to the mount. However, if you mount over your existing $HOME/Share folder you won't be able to access the contents you have created over the course of this guide. Move the contents of your shared folder to a temporary location:

```
$ mv $HOME/Share $HOME/temp
$ mkdir Share
```

The clients now need the nfs client daemons. Install them from the repository:

```
$ sudo apt-get install nfs-common
```

To ensure that the folder is mounted upon log in, append the following line to your /etc/fstab configuration file:

```
master_ip:/srv/Share /home/username/Share nfs rw,hard,intr 0 0
```

Change the line appropriately with your master node's ip and your username on each machine (including the master).

In order to mount the shared folder, you may restart your machine, or type the following command:

```
$ sudo mount -a
```

All that is left is to move the contents of the Share folder back to its correct location:

```
$ mv $HOME/temp/* $HOME/Share
```

The Slaves

This special subsection is dedicated to the slave nodes. If you applied the previous instructions to the master, it will be easy to infer the exact steps to follow for configuring the slave nodes. Here is however, a quick listing of them.

First install the tools and dependencies available through the ubuntu repository. Note that some things are only needed at the node that will be doing development, such as the svn and git applications. This command also includes nfs and openssh.

```
$ sudo apt-get install build-essential \
automake1.9 scons openjdk-6-jdk python2.6-dev \
libboost-dev libboost-dbg libboost-doc \
libboost-filesystem-dev libboost-signals-dev \
libboost-program-options-dev qt4-dev-tools libopenscenegraph-dev
    \
nfs-common openssh-server
```

Configure NFS and the environment variables per the previous sections.

The dependencies and the polygon renderer application are already available through the shared folder.

Appendix B

CAVE Configuration

The new CAVE at CRV is meant to be highly scalable with easy to find and replace components. This were features that became very desirable when the previous CAVE started to show its age.

That said, in the current state of deployment, this project runs on the left wall of the CAVE with head and hand tracking.

We recently noticed that the walls of the CAVE depolarize the reflected light and therefore are only capable of stereo with active components. Solutions are being weighted but the project can demonstrate its validity by displaying the single monoscopic multi-projector wall all the same.

B.1 Network and Devices Layout

The network layout is fairly simple, with all the nodes connected to a single network switch and accessible through public IPs.

A total of 6 projectors illuminate the 3x3 meter left wall of the cube. The are positioned to project into 3 rows and 2 columns matrix. Each node is in charge of projecting a single row. If a polarized stereo solution is found, two superimposed matrices of 3x2 will be used to generate the left-eye and right-eye images respectively. Each machine is able to feed 4 projectors, through 2 video cards.

Tracking is provided by a Polhemus Fastrack tracker, connected via serial port to the master node. There are two devices attached to this tracker:

- The head sensor, glued to a pair of polarized glasses.
- A stylus, to represent the wand an possessing one button for interaction.

Ideally a third device should be considered, a wand, providing more than one button. But we only had one serial port available and it had to be occupied by the Polhemus device.

For all administrative purposes, we recommend running the nodes from the ssh command line interface.

The master node then has direct access to the devices, by means of a serial connection to the tracker. It must then raise a vrpn service, so that all nodes can access the device data remotely. Alternatively, the vrpn service may be hosted on a separate single purpose machine on the network.

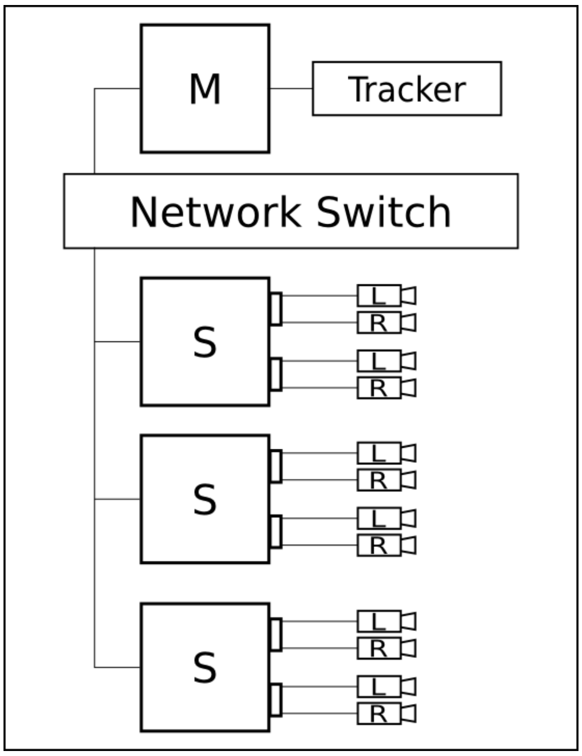


Figure B.1: Network and Devices Layout