2009

Real-time scenegraph creation and manipulation in an immersive environment using an iPhone

Brandon James Newendorp
Iowa State University

Real-time scenegraph creation and manipulation in an immersive environment using an iPhone

by

Brandon James Newendorp

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Human Computer Interaction

Program of Study Committee:
Eliot Winer, Major Professor
James Oliver
Stephen Gilbert

Iowa State University

Ames, Iowa

2009

Table of Contents

List of Figures

List of Tables

Abstract

Virtual reality (VR) display systems have undergone significant research and development since their introduction. Early systems used a head mounted display to provide users with a means of viewing a virtual environment. With the development of the CAVE Automatic Virtual Environment (CAVE™) that used multiple projectors and display surfaces, users gained a three-dimensional (3D) sense of the virtual environment and a sense of depth and immersion in the synthetic environment without bulky headwear.

One of the key challenges with creating VR environments is the creation and manipulation of 3D models to generate immersive scenes. Traditionally these models and scenes have been created on a desktop computer, using a two-dimensional display system. Although these systems have seen widespread adoption throughout academia and industry, they have significant drawbacks. When creating 3D models, the need to understand model size and spatial relationships between models is critical. This can be difficult to perceive on a 2D display system.

Another important challenge is controlling applications running in an immersive environment. Devices such as gamepads and wands are small and lightweight, making them easily carried inside an immersive environment. However, these devices require users to remember what behavior is tied to each physical button on the device. Other devices, such as Tablet PCs, overcome this limitation by offering a rich user interface, at the expense of being larger and usually requiring two hands to operate. Early handheld devices, such as PDAs, were investigated for use in

immersive environments and provided users with a graphical interface in a small device, but were limited by low resolution screens and poor hardware capabilities.

This thesis presents a two part solution to these issues, in the form of a VR application, known as iSceneBuilder, and a controlling iPhone application. Built using VR Juggler and OpenSceneGraph, iSceneBuilder allows users to create and manipulate a scenegraph — a common data structure for managing a 3D scene. By using a custom animation engine, iSceneBuilder smoothly animates changes to the scene, helping users understand how changes are being applied. iSceneBuilder was designed to run effectively on a large computer cluster and can take advantage of multiple processing cores by being designed for concurrency.

The iPhone application, which communicates with iSceneBuilder via a TCP/IP socket, provides users with a means of controlling the immersive environment. Built using Cocoa Touch, the application offers a rich user interface on a small, handheld device that, because of iPhone's capacitive touch screen, can be controlled with no additional hardware. This application allows users to browse the remote filesystem to load models into the immersive application. It also displays the scenegraph, allowing users to select a node to manipulate. Available manipulations include translation, rotation and scaling, as well as changing the transparency of a node. Additionally, users can navigate inside the immersive environment by using iPhone's built-in accelerometer.

Several uses for this system were demonstrated by creating new scenes, with varying levels of complexity. Both scenes were constructed inside an immersive environment, which allowed users to immediately perceive the size of models and their spatial relationships to other models. The first use case involved loading several models, then moving and rotating them into their final locations. The completed scene was saved as a single file that can be used in other applications. The second use involved creating several smaller scenes, then combining those smaller scenes into a larger scene. This use took advantage of iSceneBuilder's ability to manipulate components inside a larger scenegraph. Finally, this system shows promise for future development into an application that can support engineering design work.

Chapter 1: Introduction

To understand the motivation behind using an iPhone as the controller for building a 3D scene, it is necessary to understand the display systems used, the software powering those systems and existing techniques for generating and manipulating 3D geometry. A tremendous amount of research has been done to create the wide variety of state-of-the-art virtual reality (VR) systems currently available.

VR Display Systems

VR technology has gone through tremendous growth and change as it has evolved over time. Early VR systems were built around a head mounted display [1] that offered users a sense of immersion, but had limited display capabilities. These early systems had very limited fields of view (about 40°) and were very bulky, which drastically limited the user's movements. Since their introduction, head mounted displays have advanced in their abilities, offering higher resolutions and lighter weight models [2]. However, there are significant drawbacks to head mounted displays, despite recent advancements. One of the primary drawbacks of a head mounted display is that only a single user can use it. Additionally, the resolution of modern head mounted displays is still far lower than a typical desktop computer monitor. Typical head mounted displays run at 800x600px or 1024x768px, while a typical desktop LCD runs at 1680x1050px or higher.

To address some of the problems with head mounted displays, projection based VR display systems were created, which have seen significant growth in the last 15 years. Starting with the development of the CAVE Automatic Virtual Environment

(CAVE™) [3], a multi-sided immersive display system, more and more VR systems are built around one or more projectors. These systems typically use either active stereo [4] or passive stereo [5] glasses and hardware to provide a unique image to each eye. Both types of stereo glasses are able to block out images meant for the other eye. Passive stereo glasses are much less expensive than active stereo glasses, but can experience ghosting — seeing a faint double image in each eye. The difference between these two images, known as stereoscopy, allows users to perceive simulated images as three dimensional.

Projection-based VR systems are ideal when a group of people need to experience the same virtual environment at the same time. Although projection-based systems can range from a single screen to a fully immersive six wall CAVE™, they all require specialized software to generate three-dimensional (3D) content and run VR applications. Software such as CAVELib [6] and VR Juggler [7] exist to abstract the display system and input devices for software developers, simplifying the process of developing VR software for complex display systems, such as a CAVE™. By abstracting the display and input devices, developers don't need to write software specifically for a single system. Instead, developers can create VR applications that run with VR Juggler, then run their application on any VR system that supports VR Juggler.

Scenegraphs

As personal computer became capable of running 3D applications, a new market for graphics cards emerged. To ensure that software could be written to take advantage

of any graphics card, the OpenGL [8] standard was created. OpenGL is designed to provide a standardized means of describing graphical information to a graphics card, so it can render it to the display device. OpenGL, along with its competitor DirectX [9], is supported by nearly every operating system in widespread use today. While OpenGL excels at providing a low-level interface for creating graphics, it doesn't offer any capabilities for managing a complex scene or large amounts of geometry.

To make up for this shortcoming in OpenGL, a number of toolkits for managing a 3D scene's content, known as scenegraphs, have been created. Typically, a scenegraph will provide developers with a means of loading existing 3D geometry files, sorting the content within the 3D scene and manipulating the scene. Two popular open source scenegraphs today are OpenSceneGraph [10] and OpenSG [11]. Both OpenSceneGraph (OSG) and OpenSG offer similar features to developers, including a tree-based object hierarchy (see Figure 1), scene modification and extensive tools to manipulate content that is a part of the scene. However, while scenegraphs excel at managing existing content, they provide limited tools for creating new geometry from scratch. These tools primarily comprise of creating

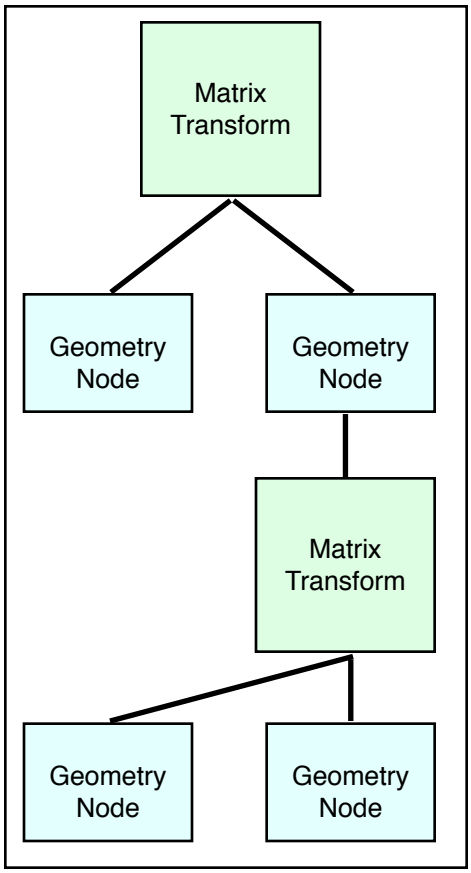

Figure 1: Example of a scenegraph tree-based object hierarchy.

basic geometric primitive shapes (e.g., cubes, spheres, and cones). More advanced tools are required to create and assemble a 3D scene.

One of the most commonly used scenegraphs is OpenSceneGraph (OSG). OSG has plugins to load a wide variety of 3D file formats into its native .osg file format. It also offers a large set of libraries that simplify the process of creating and using popular graphics techniques, such as on screen text, particle systems, volume rendering and terrain information. While OSG is capable of running on a cluster, it doesn't have any built-in provisions for sharing its scenegraph across multiple computers.

OpenSG, another popular scenegraph among VR application developers, was created specifically for applications designed to run on a computer cluster. OpenSG's developers focused on optimizing their scenegraph for running and rendering in a highly parallelized environment. The unique ability of OpenSG to share its scenegraph via the network enables it to easily run in on a large, multi-computer display system, such as a CAVE™.

3D Scene Creation Tools

A wide variety of tools exist to create 3D geometry today, including commercial 3D modeling programs, detailed engineering design tools, open source modeling tools and scenegraph editors. Two widely used commercial 3D modeling programs are Autodesk 3ds Max [12] and Autodesk Maya [13]. Both of these programs are designed for creating and modeling 3D objects with a high degree of realism as

shown in Figure 2. Although 3ds Max and Maya can be used to lay out an entire scene's content, they are primarily designed to generate a single model at a time. Another advantage of these programs is that they are able to layer complex colors and textures on models. Textures are images mapped onto the surface of a geometric shape with the purpose of giving it a more detailed and realistic appearance. One limitation of these programs is that, when exporting to a separate file, they save all the scene's content into a single model file, which doesn't preserve any hierarchy or information about the content of the scene.

Figure 2: Sample image from Autodesk 3ds Max. Image courtesy Autodesk.

Detailed computer aided design (CAD) software, such as PTC Pro/ENGINEER [14], Autodesk AutoCAD [15] and Dassault Systémes' SolidWorks [16] is widely used in industry to create detailed 3D models of products and parts such as in Figure 3. These programs are designed to allow engineers and CAD modelers to create extremely precise models of parts. However, they have little provision for modifying the color or texture on the models they create. They also are not designed to create or manage a large scene of 3D content. When exporting geometry, CAD programs often have

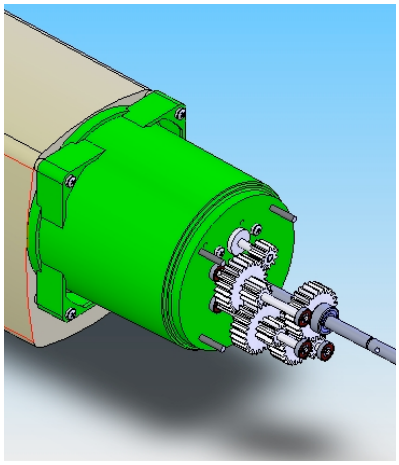

Figure 3: Sample image from SolidWorks. Image courtesy 3ds.com.

options to export a collection, or assembly, of parts that can be put together to form a larger model. However, these export formats are typically proprietary and are not easily imported into a 3D scenegraph.

Along with 3D modeling tools, there are also programs that are designed to modify and convert 3D models from one file format to another, such as Okino PolyTrans [17] and Right Hemisphere Deep Exploration [18]. The primary purpose of PolyTrans and Deep Exploration is to input a wide variety of 3D file formats, strip out extraneous data and export a final model in a format that can be read by popular scenegraphs. In particular, PolyTrans can import and export dozens of file formats. Neither of these programs are designed for creating 3D models from scratch — they primarily exist to modify and convert existing geometry. However, both of these programs are able to load multiple models and lay them out to create a larger scene.

The underlying problem with all of these programs is that they are only designed to run on a desktop computer with a two-dimensional (2D) interface — they are not designed for or capable of running in a 3D immersive environment. This is one of the central problems with most 3D modeling tools — they are used to create 3D scenes on a 2D display system. This requires the user to mentally map out the scene in 3D from a collection of 2D views as

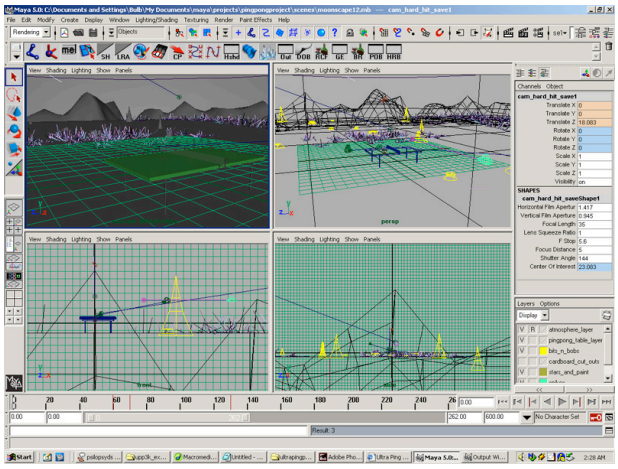

Figure 4: A typical 2D desktop program for 3D modeling. Image courtesy www.bulbmedia.net

shown in Figure 4. None of these programs are designed to run with a 3D display system that would show their content in its native form.

Desktop software in VR

Since most desktop tools attempt to display 3D content on a 2D display, such as a desktop computer, tools have been created to project their content into a 3D immersive display system, such as a CAVE™. One such program is Mechdyne's Conduit [19]. Tools like Conduit provide users with a better, more realistic experience for viewing the output of modeling programs. However, they offer limited interaction in a CAVE™ as the modeling programs were not designed for controlling a multi-screen environment. Because desktop applications are designed to run on a 2D display with a keyboard and mouse, it is difficult to provide both the desktop users and immersive viewers with good views of the virtual environment. Finally, they still suffer from the limitations of their desktop-only counterparts — they are not optimized for laying out a 3D scene.

Another approach to taking desktop software and running it in a 3D immersive environment is CaveUT [20]. CaveUT is a modified version of the commercial game Unreal Tournament 2004 [21]. While not a system for generating 3D content, CaveUT takes an interesting approach to running a desktop program in a multi-computer, large scale display system. CaveUT runs a separate copy of the game for each projector, which presents a modified view from the primary controller. However, CaveUT offers no provisions for controlling the game from within the

immersive environment — users still need to operate the game from a standalone computer.

Controlling VR applications

Because the traditional controls for a desktop computer (keyboard and mouse) are strictly two dimensional input devices, a number of different input devices have been used for 3D immersive environments. These devices range in complexity from an off the shelf gamepad to a Tablet PC. One very common VR input device is a gamepad [22], such as the Logitech Cordless Rumblepad 2™ [23] shown in Figure 5. These input devices provide users with numerous buttons and analog axes to configure as needed for a specific application. However, they typically are not tracked by the display system, so they are not able to provide a 3D input.

Figure 5: Logitech Cordless Rumblepad 2.

One alternative to gamepads is a 3D input device, known as a wand [24], which is tracked by the immersive environment. A wand is shown in Figure 6. Wands typically have a few buttons that can be used by software developers, but their primary advantage is that they offer six degrees of freedom within an immersive environment.

Figure 6: An Intersense IS-900 wand and tracking system.

These can be used in a variety of ways in a 3D immersive environment, but still are limited in what a user can do with them. One limitation, in particular, is that the user needs to remember what each button does.

Motivation

Numerous solutions exist to create new 3D geometry and modify existing 3D geometry. Some of these solutions are designed for creating detailed technical models, while others are better at creating artistic models. However, the vast majority of these solutions run on a desktop computer with a 2D display. There is room to improve on these systems by taking advantage of a 3D immersive environment. By creating scenes inside a 3D immersive environment, users have a better understanding of the models they are working with and how they relate to each other in the environment.

Additionally, many desktop tools are designed for creating single models, rather than laying out a larger scene. Although they are capable of laying out a scene, most desktop applications don't offer users the ability to easily compare objects to each other or view the scene in its real size. These are critical parts of creating a VR scene. Much of this process can be improved by bringing the scene layout into the VR environment directly, allowing users to see their scene as it's built.

Not only can the user experience of creating and laying out a 3D scene be improved by using a VR environment, the tools used inside the VR environment can also evolve. Most existing control systems for VR environments rely on the user's

memory to keep track of which buttons on an input device trigger different behaviors. Some of these systems lighten the load by using menu systems inside the application, where physical buttons control the menus.

However, mobile devices have drastically evolved recently, offering far better user experiences. Current mobile devices have higher resolution displays than their predecessors, which allows them to present richer interfaces for users. Not only have displays improved, so has the input system. While most devices use a stylus to interact with the interface, some new devices, such as Apple's iPhone, can be controlled with just a fingertip. These features, combined with built-in wireless communication, make iPhone an ideal tool for controlling a VR application.

Thesis Organization

This thesis discusses the issues of creating and manipulating a scenegraph in an immersive virtual environment and how to control applications in an immersive environment. Chapter 2 presents a literature review of past and current research in virtual reality applications, systems for controlling immersive applications and techniques for creating 3D models. Chapter 3, Methodology, first discusses how the immersive application is designed and built, then presents the iPhone application that is used to control the immersive application. Chapter 4 discusses some example uses of the applications. Chapter 5 contains a summary of the work and presents future work.

Chapter 2: Literature Review

Virtual Reality Application Development Systems

In addition to VR Juggler and a scenegraph, such as OpenSceneGraph, to create a VR applications, a number of simpler solutions exist to use virtual reality hardware without the difficulties involved in writing custom applications. Although these tools are easier for users to take advantage of, they also have a much more limited set of capabilities. These tools are developed with a specific use case in mind, then marketed for a specific purpose. While this provides for a powerful tool in certain cases, it is not always easy to adapt them for other purposes.

One of the simplest tools for running VR display systems is to modify the graphical output data from a standard desktop application — one that works with 3D data on a 2D display — and adapts it to a VR display. These tools, such as the open source Chromium [25], work by replacing the OpenGL stack on a computer with their own implementation of the OpenGL libraries. This modified OpenGL library will still generate output to the local display as normal, but it also sends the OpenGL calls to another computer that modifies them and displays them in a 3D VR display system. A key advantage to this approach is that no additional software needs to be written to run in a VR display system — standard desktop applications can be run without modification. Because of this, users don't need additional training to take advantage of a VR environment. However, desktop applications typically are not designed to run in this way. It can be difficult to control a VR application entirely from a desktop

application, and there are not going to be any VR-specific features that take advantage of the VR display system.

One such set of tools comes from ICIDO GmbH — the ICIDO Visual Decision Platform (VDP) [26]. The VDP is a collection of applications, which run in a virtual reality environment, that allow users to perform common actions in the engineering design process. Some of these applications include product reviews, ergonomic analysis and simulating flexible parts. Each of these features is a standalone application that serves a single purpose. A key advantage of this approach to virtual reality application development is that each tool can be highly optimized for its specific task. However, there is little room for users to customize the application for their specific needs. For example, if users wanted to use VR for city planning, none of the standard ICIDO applications would offer an ideal feature set for this use, and there's no easy way for users to create their own tools using the VDP system.

Another alternative for creating VR applications is Vizard [27]. Unlike the ICIDO system, Vizard allows users to create their own applications using the Vizard system. To create these applications, developers use the Python scripting language to create custom behaviors for Vizard objects. Essentially, Vizard presents a Python wrapper on top of standard VR application tools. By using Python, rather than C or C++, to script behaviors, the learning curve for new developers is reduced. This is because Python is a simpler language that doesn't have to be compiled like C++. However, it comes at a cost — users are limited to using the provided Vizard tools. Also, because Python is an interpreted scripting language, scripts written in Python

won't run as fast as machine code that is generated from a C++ compiler. When running complex VR applications with detailed models and visual effects, it is important to have an application that runs as fast as possible.

Scene Creation Tools

There are a number of tools created specifically for creating and setting up 3D scenes that will be used in a VR environment. Some programs, like OSGEdit [28], exist solely to assemble 3D models into a larger scene. OSGEdit, shown in Figure 7, can load files that are supported by OpenSceneGraph (OSG) and manipulate them as part of a larger scene. These manipulations include modifying the position, orientation and scale of an object, as well as adding new groups of scenegraph nodes. It can also save the complete scene out as a single .osg file, which is OSG's native file type. Although these capabilities allow OSGEdit to assemble a new scene, OSGEdit can't be used to generate new geometry. OSGEdit is also not capable of running on a VR display system; it only runs on a standard desktop computer. This can make it difficult for users to easily understand the 3D scene they are creating, especially if they intend to display the scene on a VR display system.

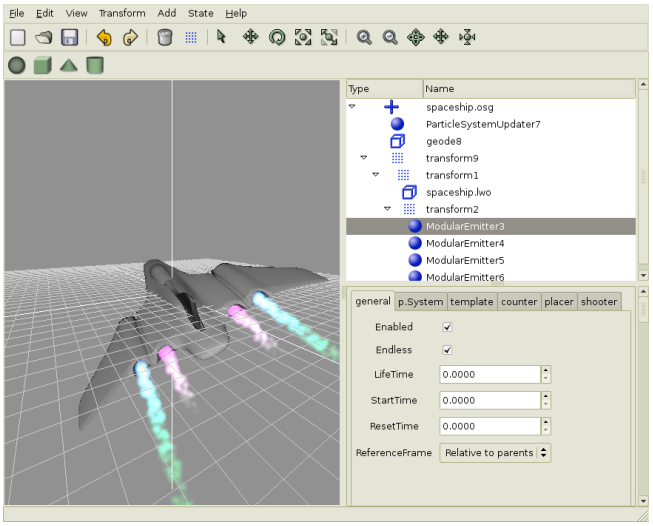

Figure 7: An example of the OSGEdit interface.

A number of programs have been

written to create 3D geometry using simplified 2D design tools, without the complexity of CAD. Zeleznik, et al. created VR Sketchpad [29], a tool that is designed to simplify the process of creating 3D geometry for architecture on a desktop computer. VR Sketchpad, however, is designed to simply create new geometry; it doesn't have provisions for importing or manipulating existing geometry. The basic premise of VR Sketchpad is that users can quickly create crude drawings on a desktop application, similar to Microsoft Paint [30]. Users quickly sketch out shapes and lines with different colors; the application translates these into 3D shapes that can be used in a virtual environment. While this is extremely easy for users to work with, this approach has a significant number of limitations. Because geometry is simply generated from 2D lines and shapes, users have no control over the height of the geometry. Additionally, VR Sketchpad offers no capabilities for modifying or managing existing scenes — it simply creates new geometry.

Another tool for creating 3D geometry, SKETCH [31], takes the idea of simple sketches on a desktop computer and combines it with gestures to create more complex models. In SKETCH, users are able to draw their ideas, as they might with pencil and paper, but can use some gestures to help define what kind of object they are drawing. SKETCH also has the ability to perform edits on geometry that has already been drawn by drawing the appropriate editing gesture. For example, users draw a set of orthogonal axes on an object to translate it within the scene. SKETCH manages a scene hierarchy based on where objects are drawn with respect to each

other. Despite these abilities, users of SKETCH are still required to mentally map the 2D views of their scene into 3D.

Some researches have investigated new hardware techniques for drawing 3D geometry using a 3D input system, rather than a keyboard & mouse. One such example, the 3D Tractus [32], uses a Tablet PC mounted on a height-adjustable stand with a sensor to monitor the height, as shown in Figure 8. This gives users a physical mapping between the height of the drawing tablet and where they are drawing in the 3D scene. By providing a 3D input system with an interface users can easily understand, this approach makes it easier for users to draw simple 3D content. However, the 3D Tractus doesn't offer users the ability to modify existing content, lay out a 3D scene, or take advantage of a VR environment.

Figure 8: The 3D Tractus drawing system.

Little work has been done in the field of creating 3D content from within a 3D virtual environment. Gardner, et al. investigated using a gamepad with multiple joysticks and buttons to draw lines in a 3D environment [33]. Their approach was to map three of the four axes on the pair of joysticks to cursor motion in the virtual environment. Each axis on a joystick would correspond to moving the cursor along a given axis. Users were able to draw 3D lines using the joysticks on the gamepad in an open 3D environment, which they found difficult and imprecise. Other buttons

on the gamepad were used to change colors of the line being drawn, display a help screen and reset the drawing area. Although drawing within a 3D environment is a good starting place for future research, this research doesn't address the concerns of how to draw more complex geometry in 3D, nor does it handle laying out or creating a new scene.

Controlling Virtual Reality Applications

Throughout the history of virtual reality, researchers have tried numerous approaches to creating a user-friendly interface for controlling and interacting with applications. These techniques have varied in both the on screen user interface (UI) and the physical devices used to interact with VR applications. While some researchers have attempted to convert traditional desktop interfaces, such as menus, to a VR environment, others have investigated more unique interaction techniques in VR.

In an effort to bring standard UI widgets to a 3D immersive environment, some researchers have ported a standard 2D desktop UI toolkit (Qt) to a CAVE™ [34]. This technique was implemented by displaying the 2D UI elements as textured objects within a 3D space. In order to control the interface, a wand replaced the behavior of the mouse on a desktop computer. An on-screen virtual keyboard was provided for text input. Test results show that the CAVE™ interface was considerably slower to users, by as much as 33% compared to a desktop keyboard and mouse interface. Although this interface will be familiar to the vast majority of

computer users, a desktop UI toolkit was designed for a 2D display and input system.

Other developers have implemented various types of menu systems in 3D for user interaction. Typically, these have the advantage of having a single degree of control at a time — users can only move up/down or left/right at any given time. For example, the Spin Menu [35] uses a circular motion for users to select between given options. When users select an option, a new circle of options is presented to them, as shown in Figure 9. Other text menus [36] use linear menus or attach menu options to user-controlled objects in the VR scene. In fact, the concept of a linear menu system has been popularized in many consumer devices, such as Apple's iPod nano [37]. A key strength of a menu system is that actions are described to users — they don't need to memorize the behavior of a given action. However, it can be tedious for users to navigate through several levels of menus to reach a specific action. Also, a menu system can only present a limited amount of information at a given time without overwhelming the user.

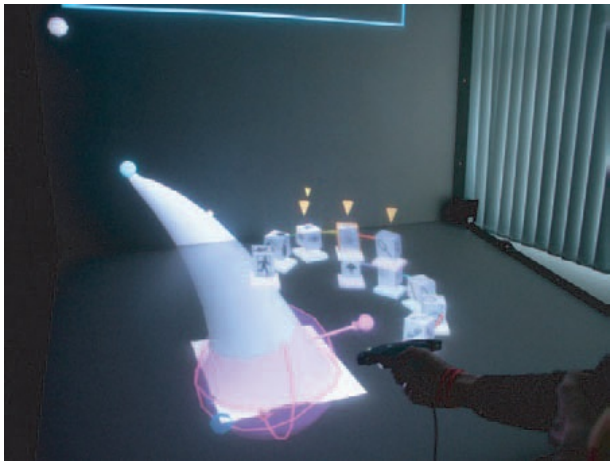

Figure 9: An example of the Spin Menu.

Another 3D interface system that is more specific for a 3D immersive environment was created by developers at ICIDO [38]. This interface allows users to select from

a number of functions at a given time by “pulling” a selector towards the desired option. One advantage of this interface is that it can vary the number of selectable items easily. However, if too many options were presented at once, it could become difficult for users to ensure they select the correct option.

A popular topic of research in VR is the use of gestures in a VR environment, which are typically performed by tracking the user’s hand or fingers [39]. With gestures, a user can perform various motions for the computer to recognize and interpret as a specific command. For example, a user can rotate their wrist to represent rotating a selected object. The concept of gesture-based controls was widely popularized with the film *Minority Report* [40]. There are a number of reasons that gross body gestures haven’t seen widespread use. First, it can be tiring for users to move their arms around for long amounts of time. Second, usability studies have found that gesture interfaces are typically, but not always, slower than traditional input systems such as a keyboard and mouse [41].

Similar to the use of gestures, full body tracking has also been researched to interact with VR environments. Many full body tracking systems use multiple cameras to track users, which eliminates the need for restrictive physical markers on the person being tracked [42]. One demonstrated use of full body tracking is to control avatars within a 3D environment [43]. Full body tracking can lead to intuitive control of a virtual environment, especially when compared to a menu system or smaller gestures. A key limitation of full body tracking, at this time, is the accuracy and reliability of the tracking systems. Often cameras are not able to provide very

reliable data about the position and pose of a person being tracked. These systems' tracking tends to "drift" away from the true position over time as well.

Hardware Devices

In addition to the numerous techniques investigated for creating a 3D user interface, researchers have created a wide variety of hardware devices for interacting with virtual reality applications. Many of these input devices are commonly used for other purposes, but are being applied in different ways to controlling a VR application. Other devices tend to be developed specifically for use with VR applications.

One approach to controlling an immersive environment is to create an application that runs on a standard desktop computer. These applications would communicate over a standard Ethernet network with the immersive environment to send commands. The Advanced Systems Design Suite [44] uses this approach of creating a feature-rich desktop application that controls a simple immersive viewer [45]. Figure 10 shows a laptop computer being used inside an immersive environment. There are several benefits to this approach. Users are often comfortable with standard desktop UI paradigms, making it easy to begin using the software.

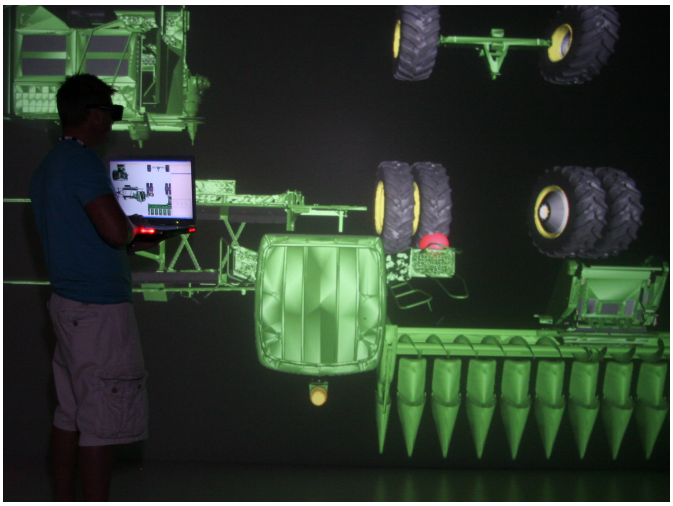

Figure 10: An example of a laptop computer controlling an immersive environment.

Also, a desktop computer typically has a significant amount of computing resources available, so very complex and powerful software can be created. However, a significant drawback to this technique is that a desktop computer cannot easily be used in an immersive environment. A desktop or laptop computer is bulky and usually requires two hands to operate, taking away from the sense of immersion.

An alternative to a desktop computer is the Tablet PC [46]. These devices provide users with a large, high resolution screen that offers a rich UI, similar to that of a desktop computer. Tablet PCs have been used to run desktop software [47] in immersive environment and they can be used entirely as a separate input device. One severe limitation of a Tablet PC, however, is that devices are both heavy and bulky. A user typically needs to cradle the Tablet PC in one arm, while using the other hand for the mandatory stylus. This greatly limits the user's mobility and freedom inside the immersive environment. Additionally, a Tablet PC usually requires the use of a stylus to interact with the screen. A stylus forces the user to be precise with their interactions, as UI designers assume the stylus can accurately select a small area on the screen.

One of the more VR-specific areas of research has been in the field of haptics — simulating the tactile sense of touch. While haptics have been popularized in the commercial market by incorporating rumble technologies into game controllers, such as a the Nintendo Wii [48] or Sony PlayStation 3 [49], more advanced haptics devices are being used in research labs [50]. Often these research oriented devices offer multiple degrees of freedom and can simulate the weight of virtual objects.

Often, haptic devices are used to simulate situations where a trained sense of touch is required, such as planning surgeries [51]. Despite these strengths, haptic devices are not necessarily a good choice for interaction in an immersive environment. Due to their size and space requirements, they easily can break a user's sense of immersion in a virtual environment.

Early in the growth of VR systems, researchers investigated the use of handheld personal digital assistants (PDAs) with immersive display systems [52]. An example of an early PDA-based interface is shown in Figure 11. Although they can't be used with a head mounted display, PDAs are certainly usable in a CAVE™. However, early PDAs offered significant limitations that hindered their growth as a VR input device. Early PDAs had no capabilities to communicate wirelessly with a standalone computer and used resistive touchscreens, which require the use of a stylus — requiring the use of both hands to operate the device at all times. Newer PDAs added some wireless communications capabilities, but still were limited by the screen's input system. Finally, PDAs typically have low resolution screens, which greatly limits what the UI can show. A typical PDA runs at a resolution of 320x240 or lower. At this resolution, very little

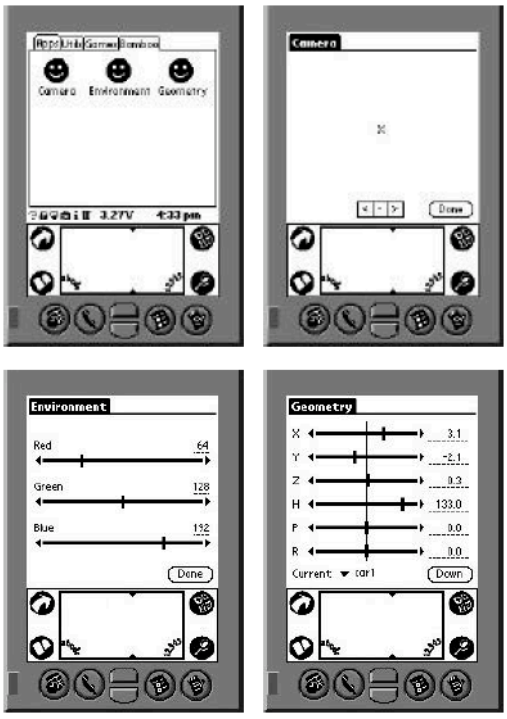

Figure 11: An interface for interacting with an immersive environment on an early PDA.

text can be shown on screen at a given time alongside user interface elements.

Motivation for mobile devices

Despite some of the limitations encountered with the earlier use of PDAs in virtual environments, recent advances in mobile computing have rekindled interest in their use. Current mobile devices have a number of new technologies that make them more suitable for use in virtual environments, including higher resolution screens, improved touchscreens, wireless communication and more advanced software development kits (SDKs).

The use of mobile computing devices has seen significant growth in recent years, with a number of organizations creating custom software for their own purposes. For example, iRobot has investigated using mobile devices for controlling their PackBot robot [53]. By taking advantage of a device with a built-in screen and controls, the amount of hardware required to control the robot is reduced [54]. Other researchers created tools to run augmented reality applications on mobile devices [55], such as smartphones and PDAs.

Until recently, touchscreen technology almost exclusively required the use of a stylus when fine, detailed actions were required. In particular, PDA and Tablet PC touchscreens were designed for operators to use a stylus. Although a stylus can ensure that users have precise control over the device, they tend to slow down user inputs and frustrate users [56]. One issue that users tend to encounter is parallax error — the difference in mapping user touch events to the actual displayed content.

If the touchscreen inputs are not perfectly aligned with the display, users have a difficult time accurately controlling the device. Users also need more time to precisely select an on-screen element with a stylus [57]. These issues have been mitigated through capacitive touchscreen technology.

The resolution of the screen on a mobile device is another key factor in the usability of mobile devices. A higher resolution screen is able to present more data to the user at a single time, and can display more detailed information. A popular area of research is using mobile devices to teleoperate robotic vehicles [58]. Many robotic vehicles include onboard cameras, which help remote operators see the world around the robot. Many interfaces will show these camera views on a mobile device, using the entire screen [59]. By being able to present more layers of information to users at a given time, users can have a better understanding of the remote environment. It is important, however, to not overload the user with too much information at once.

Despite the fact that mobile devices have higher resolution screens than their predecessors, it is still important to only present relevant information to the end user at a given time. In *The Design of Everyday Things*, Don Norman discusses a good user interface that provides good feedback to the user about their actions and only shows relevant parts of the interface at a time [60]. Although in his example, Norman is discussing a complex stereo control system, these design principles are just as applicable to software design, especially on a mobile device.

Overall, a number of solutions have been presented for interacting with virtual reality applications. Some of these solutions offer rich user interfaces at the cost of a large and bulky device, such as a Tablet PC. Other solutions use existing virtual reality hardware, like a wand or gamepad, but are more complex for users and can only show limited information on screen at once. Old PDA-based solutions started to address these problems but were still limited by the hardware capabilities at the time.

Research Issues

Based on the literature review of current research in scenegraph manipulation in virtual reality and systems for controlling virtual reality applications in immersive environments, two research questions have been identified. They are:

1. **Can 3D immersive display environments be used for creating and manipulating scenegraphs?**

   As described above, most scenegraphs and 3D models are created on two-dimensional displays, typically on a desktop computer. While this technique is widely used in industry, there is room for improvement. Rather than require users to mentally map 2D images of a 3D environment together, why not use a 3D display system to layout a 3D scene? This would allow users to intuitively create a scene, immediately understanding where objects are relative to each other.

2. Can an iPhone be a usable interface device for scenegraph manipulation in an immersive VR environment?

Numerous solutions have been presented for controlling applications in an immersive environment. However, all the presented solutions have their drawbacks, including large, bulky devices or relying on the user's memory to function properly. Recent mobile devices, such as Apple's iPhone, have a richer feature set that can improve on existing attempts at controlling immersive applications.

Chapter 3: Methodology

To address the research issues identified above, a two-part system was developed. The first part of this system is an immersive application that presents a 3D virtual environment to users. This application allows users to design, create and manipulate a scenegraph from inside the virtual environment. To interact with the scenegraph, a controller application was created to run on an iPhone. This chapter details how both of these applications were created.

Immersive Application

As described in the research questions section, one of the key issues that needs to be addressed is how to create and manipulate a scene in 3D. To this end, an immersive application was created to run in C6 at Iowa State University [61], a six wall fully-immersive environment. This section will detail the immersive application, known as iSceneBuilder, and how it was designed.

VR Juggler

The underlying foundation of iSceneBuilder is built on the VR Juggler framework. By utilizing VR Juggler, iSceneBuilder can easily run on a wide variety of VR display systems, including C6, single wall displays and standalone computers. Although the VR Juggler suite includes numerous software tools to assist application developers, only a few features of VR Juggler were used in iSceneBuilder.

At the lowest level, iSceneBuilder launches from the VR Juggler kernel. The kernel is responsible for loading VR Juggler configuration files — these are used to

describe the environment the application is running in. For example, the C6 configuration file describes the computer cluster, the graphics output from each node in the cluster and the tracking system. Although they are not used in iSceneBuilder, VR Juggler configuration files are often used to describe input devices as well. When the application kernel launches, it determines from command line argument whether it is running as a cluster master node or a cluster slave node. If it's running as a master node, the application kernel sends a copy of pertinent configuration data to all of the slave nodes.

It is important to understand how VR Juggler runs applications on a cluster. Each node in a cluster runs a unique instance of the application. The application running on each node is responsible for maintaining its own memory contents and updating its graphics output. VR Juggler has provisions for sharing and distributing information across the cluster, which are described in the **Cluster Networking** section of this chapter.

Once the VR Juggler kernel is initialized, the application begins its own initialization process. The first step of the initialization is to initialize the VR Juggler input devices — in this case, the head tracker. After that, iSceneBuilder creates the base of the scenegraph tree. The scenegraph structure is described in the **OpenSceneGraph Integration** section of this chapter. Once the scenegraph has been created, the application initializes the networking system, which is responsible for communication with the controller application.

Additionally, there is some important configuration data that needs to be used as part of the application setup. This data can change based on the computer iSceneBuilder is running on — it includes the location of the application's data and a globally unique identifier (GUID) for the VR Juggler shared data. iSceneBuilder stores this data in a XML file, which provides for a human readable file. When the application launches, the data is read from the XML file and stored in variables for later use.

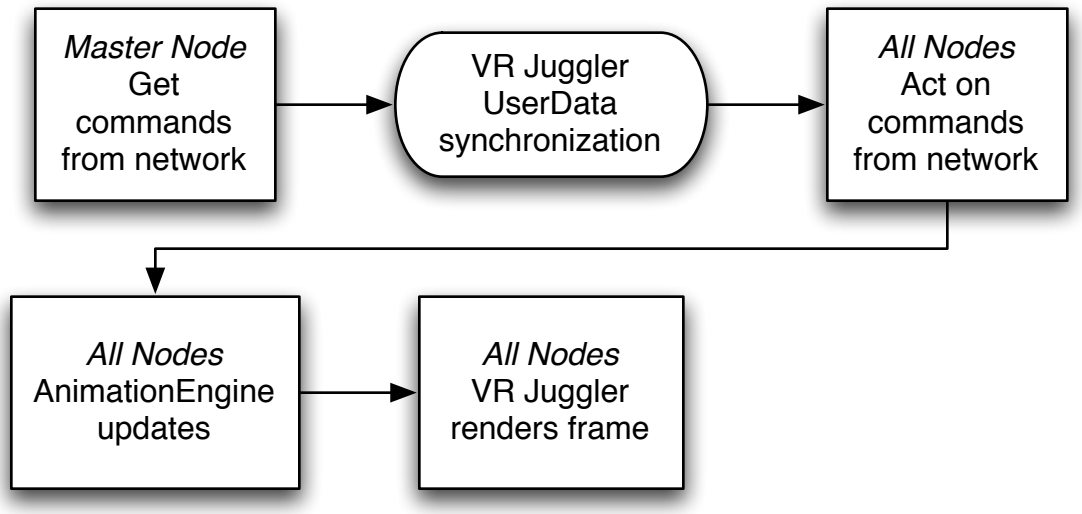

Figure 12: Steps taken to render a frame in iSceneBuilder.

Beyond the initialization of the application, VR Juggler is responsible for managing the main run loop of the application. There are a number of steps taken to draw each frame; these steps are controlled by VR Juggler. The first step in each frame is to clear the render buffer, then allow the application to update data before rendering the frame. There are two stages to updating data — the preFrame and the latePreFrame. In the preFrame, the master node receives an updated set of

commands from the controller application. Between the preFrame and latePreFrame, VR Juggler synchronizes this set of commands so that every node in the cluster has identical copies of the data. During the latePreFrame, each node responds to the incoming commands. The rendering pipeline for iSceneBuilder is shown in Figure 12.

Cluster Networking

As described in the **VR Juggler** section, VR Juggler runs a unique instance of the application on each node of the computer cluster. It is critical that each application have the same set of data to act on, so that each node runs the application identically to the other nodes. If any single node falls out of sync with the remainder of the cluster, the application will no longer operate normally and needs to be restarted. To address this critical issue, VR Juggler provides a UserData object that is shared and synchronized across the entire cluster.

iSceneBuilder extends the VR Juggler UserData object to maintain a list of commands that have been sent by the controller application and synchronize them with all the nodes. The first step in this process is to receive commands on the master node — the computer that is responsible for controlling the cluster. This computer stores the commands in a queue. Once VR Juggler is ready to synchronize data, the commands stored in the queue are serialized and sent to the other nodes. Other nodes de-serialize the commands and store them in another queue to be interpreted later. This process occurs as part of drawing every frame.

Networking & Concurrency

In addition to sharing commands between cluster nodes, iSceneBuilder also supports two-way communication with the controller application over a network. Specifically, iSceneBuilder uses a single TCP/IP socket. When the application is initialized, a TCP server socket is created, which listens on a designated port for incoming connection requests. Once it has accepted a connection, it begins a perpetual loop where it receives a block of data into a buffer, then sends any messages that have been queued to be sent. After iSceneBuilder receives a block of data, the buffer is parsed to find individual commands, which are then handled by the VR Juggler cluster shared data system. The message syntax used in iSceneBuilder is described in the **iPhone Networking** section.

A general trend in computing is to offering systems with multiple processors and/or multiple cores. In order to take advantage of these capabilities, developers need to consider concurrency when designing their applications. A typical desktop application only runs in a single thread, meaning it can only perform one task at any given time. By designing applications with concurrency, developers can enable their applications to perform multiple tasks simultaneously, taking advantage of more resources on the computer. One approach to concurrency is multithreading — using more than one thread within an application.

Because the TCP/IP socket is a blocking socket — meaning it will halt execution until it receives data — the socket cannot exist in the application's main run loop. If it was to exist in the main run loop, iSceneBuilder would stop rendering frames until

it received data over the network. Due to the sporadic nature of incoming data, this is an unacceptable behavior. To address this problem, iSceneBuilder runs all network traffic in a separate thread, utilizing a standard POSIX thread (pthread) [62]. A pthread offers an additional run loop, which iSceneBuilder dedicates to network traffic. This enables the main run loop to continue rendering frames without having to wait for network traffic. When the application is told to terminate, the TCP/IP socket is released from memory and the pthread is destroyed. By properly closing the socket, the port is immediately made available again for other applications to use.

Filesystem Integration

Because the controller application, which is running on a separate device, doesn't necessarily have access to the filesystem, iSceneBuilder is responsible for navigating its local filesystem. Specifically, iSceneBuilder needs to move to a specified directory and get a file listing from that directory. All of this information is sent to the controller application via the TCP/IP socket described in the **Networking & Concurrency** section. To manage these responsibilities, iSceneBuilder includes a class known as FileSystem.

Internally, FileSystem maintains a string that is iSceneBuilder's current directory. This is updated when the controller application tells iSceneBuilder to change to a new directory. The bulk of the work in FileSystem is to print the current directory's contents. This method opens the current directory and reads all of the directory's contents into a buffer. Then, it removes irrelevant data from the buffer — hidden

files and anything that isn't another directory or file. The final buffer of the directory listing is given to the networking system to transmit to the controller application.

OpenSceneGraph Integration

iSceneBuilder, as a scenegraph creation and manipulation tool, relies heavily on its internal scenegraph. As described in the Introduction, OpenSceneGraph offers a robust and powerful set of tools, which is why it was selected for iSceneBuilder. As iSceneBuilder initializes, the base scenegraph is constructed. This starts with a root node, which is an OSG Group — a node that can contain connections to other nodes. Attached to the root node are two other groups. One of these groups, called mNoNav, is maintained for objects that need to remain in a static position at all times, while the other group, known as mNavTrans, is where all user navigation commands are applied. All other geometry is attached to mNavTrans, so that any user navigation commands recursively affect the rest of the scene. This hierarchy is represented in Figure 13.

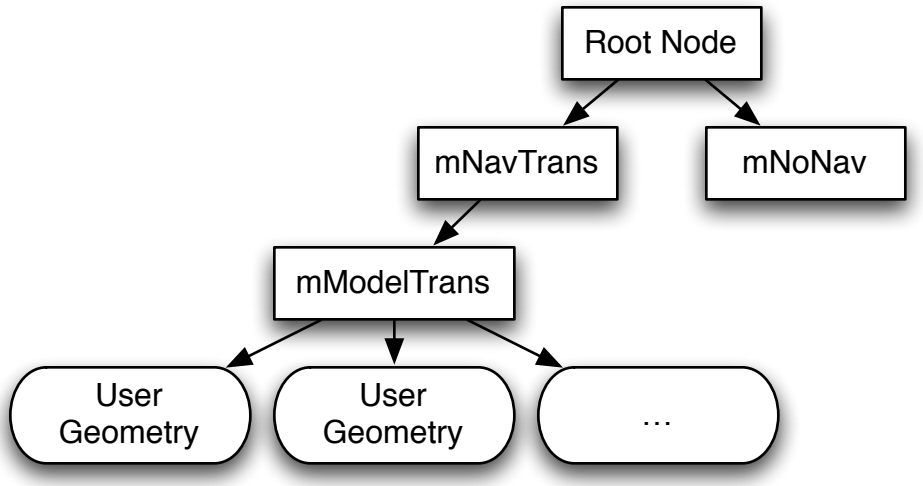

Figure 13: Diagram of the iSceneBuilder scenegraph.

Because the primary goal of iSceneBuilder is to manage and manipulate the scenegraph, a number of scenegraph tools were necessary. An internal class, known as ScenegraphControls, is a toolkit of scenegraph manipulation methods that are used for all of iSceneBuilder's functionality. One of these tools is used to change a node's internal name. A node's name has no impact on how the node is rendered; it simply exists for the user's sake. Another tool will generate a string containing key information about every node within the scenegraph. This tool is described in further detail in the **OpenSceneGraph NodeVisitors** section. A third tool in ScenegraphControls gets the detailed information about a node, including its name, unique identifier and current rotation values, and formats them into a string that can be sent to the controller application.

ScenegraphControls is also used for translating nodes, rotating nodes, scaling nodes and changing the transparency of a node. To implement these features, ScenegraphControls first needs to prepare the instructions. For example, the user sets a node's rotation using degrees, because degrees are easier for a user to understand. However, OSG internally uses radians for rotation data, so ScenegraphControls has to perform conversions to the appropriate data types. Once the data is prepared, ScenegraphControls creates a new AnimationCommand and adds the newly created command to the AnimationEngine. Both AnimationEngine and AnimationCommand are detailed in the **AnimationEngine** section.

In addition to manipulating the scenegraph, iSceneBuilder has intelligence built into how it loads geometry. Rather than simply loading a file when instructed to, iSceneBuilder maintains an internal list of every file it's loaded, how many copies need to be in the scene and how many copies of that model have been loaded already. When it's instructed to load a model, iSceneBuilder simply increments the counter for the number of needed copies. Every frame, iSceneBuilder checks if any new models need to be loaded, then adds them to the scenegraph if necessary. This intelligent model loading system ensures that every model has a unique name, helping the user keep track of what they have in the scene.

AnimationEngine

AnimationEngine is a state-based scenegraph animation system that can easily be incorporated into any application that uses OSG, such as iSceneBuilder. The goal of *AnimationEngine* is to make it easy for developers to add animations to their applications. By animating changes to the scenegraph, rather than snapping to a new setting instantly, users have better understanding of what is happening in the environment around them. iSceneBuilder uses *AnimationEngine* to power all of its object manipulation commands.

There are two key components to *AnimationEngine*: AnimationCommand and AnimationEngine. AnimationEngine is fairly simple — it maintains an internal list of active AnimationCommands and tells each active command to update itself every frame. When the developer adds a new AnimationCommand to the engine, it replaces any existing commands for that node with the new command. By ensuring

that only the latest command for an object exists in the AnimationEngine, there can't be a backlog of commands waiting to execute. The other advantage to this behavior is made apparent when a new command is given to an animation that is already in progress. For example, a command is halfway completed that moves an object from 0,0,0 to 100,0,0, meaning the object is currently at 50,0,0. A new command is given to the AnimationEngine that instructs the object to move to 50,50,0. Rather than first moving to 100,0,0, then proceeding to 50,50,0, the object will smoothly begin moving to its new goal of 50,50,0.

The bulk of the capabilities of *AnimationEngine* are implemented in AnimationCommand. There are four types of AnimationCommand: translate, rotate, scale and adjust transparency. All of these command types have several things in common, including how many frames the command should take to complete its goal, the goal state and the node to modify. Each command is capable of updating itself every frame by linearly interpolating between the original state and the goal state. Rotation commands use quaternions for interpolation, while translate, scale and transparency commands are based on three-dimensional vectors.

There are a few key benefits to using *AnimationEngine*. The primary benefit is that *AnimationEngine* offers "fire and forget" animations. Once a developer adds an AnimationCommand to the AnimationEngine, they don't have to do any additional work to support the command — the engine will complete the animation and clean up after itself. Second, by animating changes to the scenegraph, users have a better understanding of the virtual environment and how they are impacting it.

Finally, in a situation where commands may have high latency (such as receiving commands over a slow network), animating changes will provide users with a smoother experience, helping minimize the visual impact of the latency.

OpenSceneGraph Node Visitors

There are two situations where iSceneBuilder needs to interact with every node in the current scenegraph. Rather than maintain a separate system for storing a pointer to each node, iSceneBuilder uses a pair of OSG NodeVisitors to interact with the entire scenegraph when necessary. A NodeVisitor is an object that is called recursively on every node in the scenegraph and can apply an operation to each node it finds. The first of these NodeVisitors simply builds a string with the name and unique identifier for each transform node it finds. This string is designed to be sent to the controller application. The second NodeVisitor adds a unique UserData object to every node in the scenegraph.

This unique UserData object has a few important pieces of information that are used elsewhere in iSceneBuilder. The first part of the UserData object stores the current rotation values of the node in degrees. By storing this data separately, less calculations are required when the controller application requests the rotation of a node in degrees. The other part of the UserData object is a unique identifier for each node, which is an integer. This is necessary because OpenSceneGraph doesn't have a unique identifier for each node. iSceneBuilder maintains an internal counter for every node added to the scenegraph, which is incremented for every

new UserData object. The controller application uses this unique identifier to tell iSceneBuilder which node it should apply changes to.

iPhone Software Development

Released in 2007, Apple's iPhone offers developers with a new hardware device that extends the capabilities of a traditional PDA [63]. Like mobile devices, iPhone is a small handheld device with a touchscreen. Unlike most mobile devices, iPhone uses a capacitive touchscreen. There are two key differences between resistive and capacitive touchscreens. First, a capacitive touchscreen is operated with a user's finger instead of a stylus. Second, resistive touchscreens are limited to detecting a single point of contact, while capacitive touchscreens can detect multiple simultaneous contacts (multi-touch).

Apple offers developers access to several key features of iPhone through their Cocoa Touch API [64]. In this chapter, any method calls that begin with the NS or UI prefix are part of the Cocoa Touch API. There are several unique features on iPhone that make it an ideal device for controlling virtual reality applications in an immersive environment. First, iPhone has built-in WiFi, which developers have access to [65], making it easy to connect iPhone applications to another computer or any device on a network. Second, iPhone has an accelerometer [66], which is capable of detecting the device's orientation. This can provide developers with additional means of controlling 3D applications. Recent models of iPhone also include an electronic compass, which can determine which direction the device is facing. Finally, Apple provides Cocoa Touch developers with the ability to draw custom user interfaces

with the CoreGraphics system [67]. With CoreGraphics, developers are not limited to the default UI objects when creating applications. iPhone OS is built on CoreGraphics, which makes it possible to create applications that are both visually appealing to users and consistent with the existing design paradigms on iPhone.

All of these features combine to make a compelling device for controlling applications in an immersive environment. Unlike early generation PDAs, iPhone has a high resolution screen that doesn't require a stylus for interaction. Additionally, iPhone has built in support for wireless networking, which makes it easy to interact with other computers. iPhone is also a small, handheld device that can be operated with one hand, leaving the user's other hand free. This contrasts with Tablet PCs, which need one arm to cradle the device, while the other hand uses the stylus to control the computer. Because of these reasons, the controller application for iSceneBuilder was written for iPhone. The remainder of this chapter describes how the iPhone application was built.

Application Delegate

The base part of the iPhone application is the application delegate. In Cocoa Touch, a delegate is a method that is registered to receive callbacks from other process. This class, which is a subclass of UIApplicationDelegate, is primarily responsible for responding to notifications from iPhone OS, such as launching, low memory warnings and the user terminating the application. In addition to these functions, the application delegate also controls and manages the socket used for communicating with iSceneBuilder over the network. Because the application delegate receives all

incoming network traffic and sends outgoing messages, it also needs to keep track of the view controllers, so that it can pass relevant messages to the appropriate receivers.

The application delegate also maintains the UITabBarController — this provides the tab buttons at the bottom of the screen, which are used to cycle between modes of the application. The iPhone application's tab bar is shown in Figure 14. The first button, Network, is used to connect to iSceneBuilder and save the current scene as an OSG file on the remote file system. The second button, File Browser, is used to browse the remote file system and add models to the current scene. File Browser is detailed in the **FileListingTableViewController Class** section. The third button, Scenegraph, provides users with a view of the current scenegraph hierarchy and allows them to edit the characteristics of a node. The capabilities of the Scenegraph view are described in the **ScenegraphTableViewController Class** section. Finally, the fourth button, Navigation, allows users to move around inside the immersive environment. The Navigation button is discussed in the **NavigationViewController Class** section.

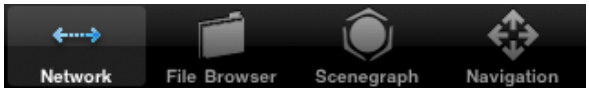

Figure 14: The tab bar items in the iPhone application.

iPhone Networking

Because the primary purpose of the iPhone application is to control iSceneBuilder, the networking system is of critical importance. Both applications communicate via a TCP/IP socket, which guarantees packets will be delivered to the recipient in the order they're sent in. Unlike iSceneBuilder, the TCP socket in the iPhone application doesn't need to be run in a separate thread. This is because NetSocket, the networking library used, is configured to use the current NSRunLoop. By utilizing the current run loop, the socket is non-blocking and will only briefly check for incoming data before allowing the program execution to continue. Because the application delegate is also the NetSocket delegate, it receives a method call when a handled event occurs on the socket: socket connected, socket disconnected and socket data is available.

Data must be formatted in a specific way so that both iSceneBuilder and the iPhone application can parse the data they receive. Below is an example message sent by the iPhone application to iSceneBuilder.

```
6:4:8.000000:0.051021:-10.000000;
```

Every block of the message is separated by a colon, while the message is terminated with a semicolon. The first block of the message is the command type, which is an integer. In this message, 6 instructs iSceneBuilder that this is a translate command. The next block, 4, specifies the unique node identifier of the node that

should be modified. The final three blocks specify the destination location of the node as floating point values.

In addition to specific commands, like the one above, other command types have no blocks beyond the command type block in the beginning. The most commonly sent message is known as the heartbeat message. The iPhone application has a NSTimer that repeats every 0.25 second. This timer sends a basic message to iSceneBuilder is used to ensure there is still an active network connection. If a certain number of these messages are not sent successfully, the application could automatically disconnect itself from iSceneBuilder.

The iPhone application also receives data from iSceneBuilder, which it needs to parse before it can handle the incoming command. To do this, the iPhone application uses a number of the string parsing capabilities of NSString. Primarily, the componentsSeparatedByString method is used to parse the separate blocks. This method returns an array containing the elements of a string that are separated by a specified delimiter — in this case, a colon. Once the blocks have been parsed, individual view controllers can enumerate through the array of blocks to interpret the command.

FileListingTableViewController Class

To allow the user to navigate through the remote file system and select models to load, the iPhone application needs to be able to display this information to users. The FileListingTableView, which is a UITableView object, provides this capability.

There are several key components to the FileListingTableViewController: receiving and parsing incoming data, creating UITableViewCells and handling user interactions with the UITableView. The FileListingTableViewController is shown in Figure 15.

When a message is given to the FileListingTableViewController, the controller needs to parse the incoming directory listing so that it only displays relevant information to the user. The FileListingTableViewController creates FileListing objects, which contain a fileType and fileName. When parsing the incoming data, the first step is to determine the file type — a folder, a file or the current directory. After this has been determined, the FileListingTableViewController identifies supported 3D model files. After the entire message has been parsed, the resulting FileListing

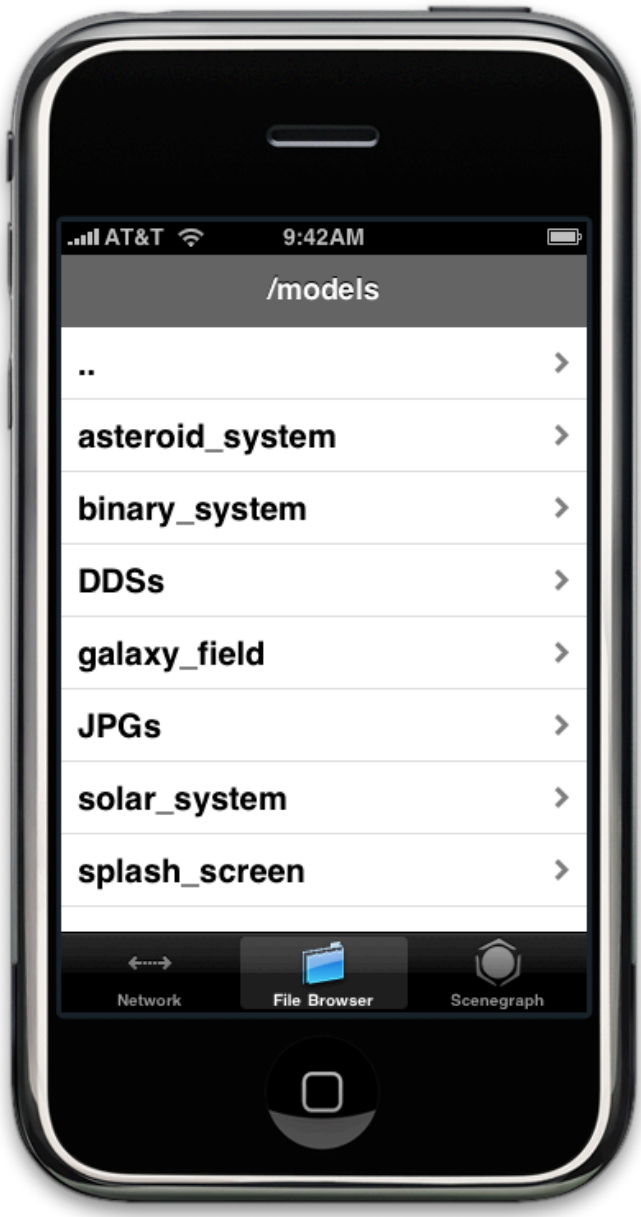

Figure 15: The FileListingTableViewController for the iPhone application.

objects are stored in a NSArray.

The data stored in the NSArray is used by the FileListingTableViewController to create UITableViewCells — the on-screen elements the user interacts with. These cells are created on demand, when the OS requests a new one be created and made visible to the user. By only creating cells as necessary, the memory overhead of the application is reduced — an important factor on a mobile device such as iPhone. An important property of a UITableViewCell is the accessoryType, which is the graphical element on the right side of the cell. The iPhone application sets different accessories based on whether the cell is displaying a folder or a file.

The final component of FileListingTableViewController is handling user interactions with the UITableView. There are two interactions that need to be accounted for — selecting a folder and selecting a file. If the user selects a folder by tapping anywhere on the cell, FileListingTableViewController creates a new network message instructing iSceneBuilder to change to the new directory and send back an updated directory listing. When FileListingTableViewController receives the new directory listing, it updates its collection of cells that are displayed to the user. If the user loads a model by tapping the accessory icon in the cell, FileListingTableViewController sends a network message to iSceneBuilder that contains the model's name. The model will immediately be loaded by iSceneBuilder and will become visible to the user in the immersive environment.

ScenegraphTableViewController Class

The third button on the tab bar, Scenegraph, presents the ScenegraphTableViewController, which is shown in Figure 16. This view displays the current scenegraph hierarchy to the user and allows them to select a specific node to edit. Similar to the FileListingTableViewController, the ScenegraphTableViewController uses an internal NSArray to store its contents and creates UITableViewCells that are displayed to the user.

When parsing an incoming scenegraph list, the ScenegraphTableViewController creates ScenegraphListing objects. Similar to the UserData objects that are created for OpenSceneGraph in iSceneBuilder, ScenegraphListing objects stores the node's name, unique identifier, depth from the root node and rotation values. The node's name is used to generate the name of each UITableViewCell,

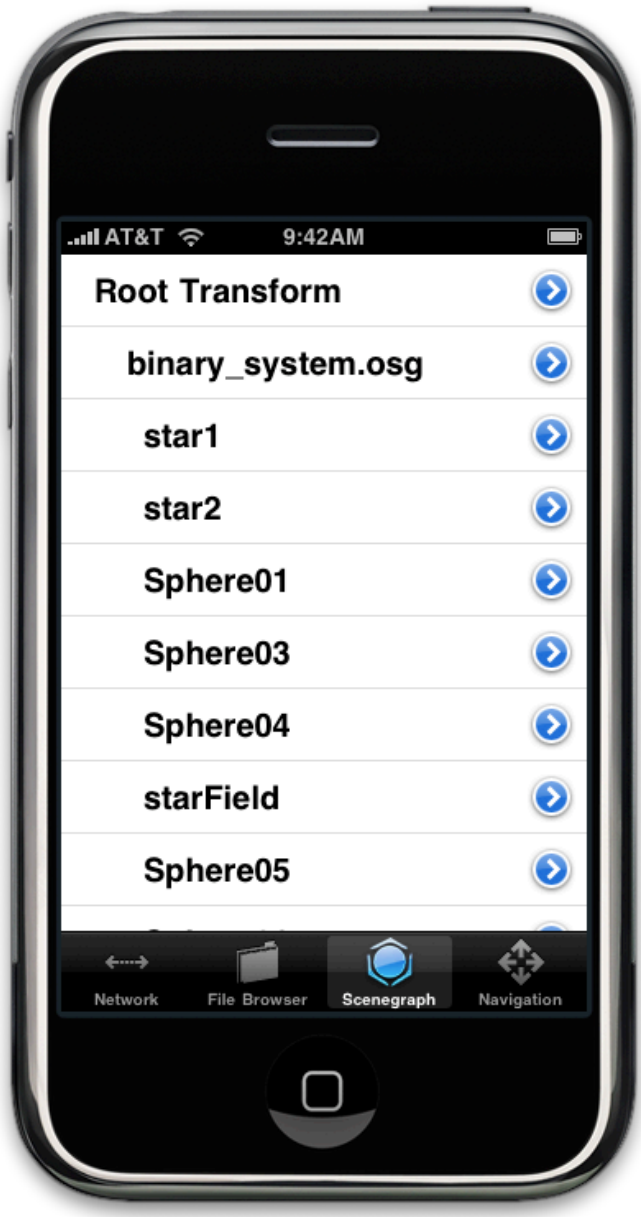

Figure 16: The ScenegraphTableViewController of the iPhone application.

while the depth is used to determine the indentation of the cell. Other data isn't visible to the user in the ScenegraphTableViewController.

When the user taps on the detail disclosure accessory on a cell (the blue arrow on the right side of the cell), the ScenegraphTableViewController determines which cell and node was selected, then generates a new command to be sent to iSceneBuilder. This command instructs iSceneBuilder to generate the detailed data for that node and send it back to the iPhone application. When that data is received, the NodeDetailViewController is created and made active. This view is described in further detail in the **NodeDetailViewController Class** section.

NodeDetailViewController Class

Perhaps the most important, or at least most used, view in the iPhone application is the NodeDetailViewController, shown in Figure 17. This view, unlike the previously described UIViewControllers, does not present a UITableView to users. Instead, it presents a customized UIView with a number of elements laid out on it. The purpose of the NodeDetailViewController is to allow users to manipulate important characteristics of a node in iSceneBuilder's scenegraph.

At the top of the NodeDetailViewController is a UITextField, which is used for editing the node's name. Below the UITextField is a UISegmentedControl, which has four segments, used to select a type of manipulation. Depending on what manipulation is currently selected, a set of sliders will be visible to the user.

These sliders, which are customized UISlider objects, are the most unique user interface element of the iPhone application. The standard UISlider, shown in Figure 18, is a horizontal slider with blue tracks and a plain white thumb. This contrasts with the customized UISliders in the iPhone application, which can be seen in Figure 17, that are vertical, have red/green/blue/orange tracks, thumbs with lettering and images on both ends of the slider.

Figure 17: The NodeDetailViewController in Scale mode.

In addition to their unique visual appearance, the customized UISliders have modified behavior, based on the selected manipulation. Typically a UISlider is used to select from a discrete range of values. In the case of rotation, this behavior is appropriate — users select a rotation value between 0° and 360° around

Figure 18: A standard UISlider.

each axis. Similarly, changing a node's transparency also is a discrete range of values, where users select a transparency value between 0% and 100% transparent. However, translation and scale commands do not operate on a discrete range of values. Instead, the sliders have a custom "spring-loaded" behavior where they will reset to zero when the user isn't touching them. This behavior is similar to how a physical joystick or gamepad would behave. Because of this behavior, users can move objects precisely in small areas and quickly across large areas with the same interface.

Typically a user will manipulate geometry along a single axis at a given time, so three sliders are presented to users in translate, rotate and scale modes. The sliders are colored to correspond to the standard colored axes in virtual reality applications. Additionally, the icons at the top and bottom of the sliders represent which direction the slider controls. In the event that the user wants to manipulate an object in two or three axes simultaneously, the sliders are multi-touch enabled. A user can drag two or three of the sliders at the same time, in different directions if desired. This is a feature that takes advantage of iPhone's multi-touch display that is not found on many other devices. Scale mode contains a fourth slider that will scale the node in all three axes simultaneously, because users will often want to make the object larger or smaller, rather than stretching it along one axis.

Finally, the NodeDetailViewController takes advantage of Core Animation [68] to provide a smooth, rich interface to users. Similar to *AnimationEngine*, used in iSceneBuilder, Core Animation is used to give users a better sense of their

interactions with the application. In the case of the iPhone application, the number of sliders on screen at any given time can vary between one, three and four. As users select different manipulation modes, the application presents different sets of sliders to the user. Core Animation moves the sliders around on screen and fades them in and out, as necessary. Additionally, Core Animation is used when the value of sliders is changed programmatically, rather than immediately moving the thumb on the slider to the correct position.

NavigationViewController Class

Rather than forcing users to view the immersive environment from a fixed position, users need to be able to freely explore the scene they are creating. In order to allow the user to move around inside the immersive application, the NavigationViewController was created. This class, which is activated by the fourth button on the UITabBar, takes advantage of iPhone-specific hardware.

One of the simplest classes in the iPhone application, NavigationViewController has a single, large UIButton that covers the entire screen, which is shown in Figure 19. When a user taps and holds on this button, its image changes with new text, telling the user to tilt to navigate around. At the same time, when iPhone OS detects a touch down event on the button, it stores the current orientation of the device from the accelerometer. As long as the user is still touching the button, the iPhone application gets the current accelerometer position every 0.1 seconds, finds the difference between the current orientation and the stored orientation and sends the difference to iSceneBuilder. By storing an initial orientation and finding the

difference, a new “neutral” position is set every time the user begins navigating. This provides for a better user experience when controlling the application, because users aren’t forced to hold their iPhone in a specific orientation to navigate properly. When the user lets up on the button, the initial position is erased and the application stops responding to accelerometer events.

In addition to the button that controls user navigation, there are two additional controls that modify how the user navigates. By default, user navigation moves on a horizontal plane, along the X and Z axes. However, users occasionally need to move up and down as well. To enable this behavior, a UISwitch was placed at the top of the view. When toggled on, user navigation occurs in a vertical plane, along the X and Y axes. In this situation, tilting their iPhone towards the user moves up, while tilting their iPhone away from the user moves down.

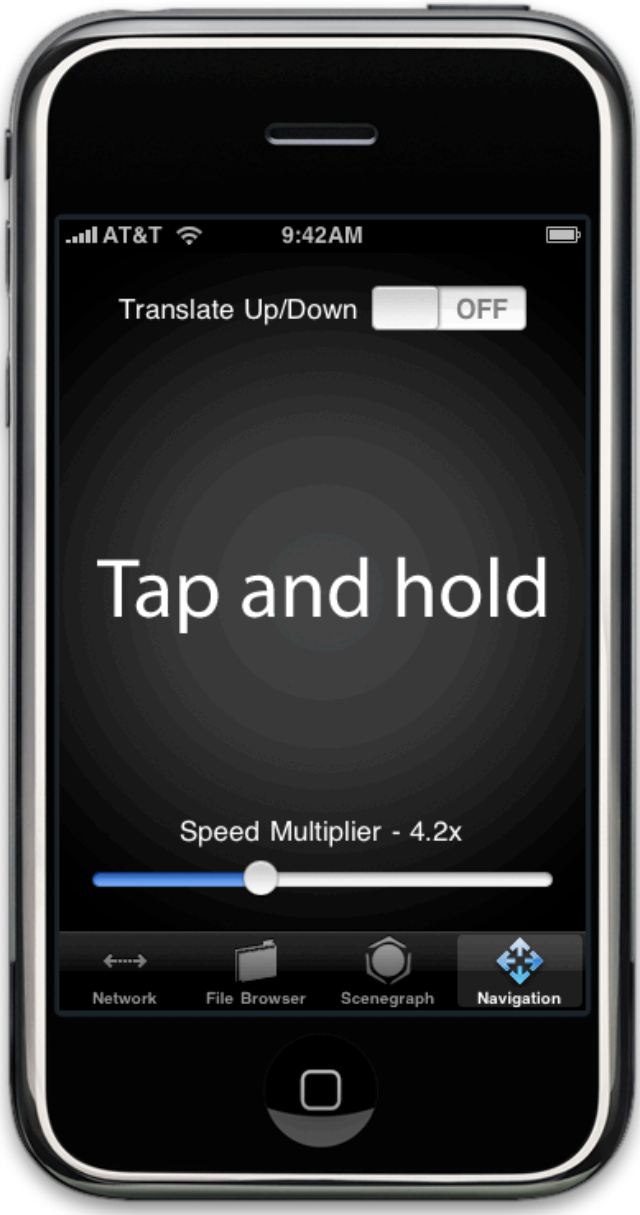

Figure 19: The NavigationViewController of the iPhone application.

Some scenes can be fairly large, so the user needs to move from one part of the scene to another. However, the user also needs precise speed controls in smaller areas. To facilitate these needs, the UISlider at the bottom of the view controls a multiplier for the navigation speed. With values ranging from one to ten, the accelerometer values are multiplied by the current value of the slider to get the final navigation speed. This gives the user slow and precise or fast navigation as necessary.

The following table, Table 1, summarizes all of the custom classes in iSceneBuilder and the iPhone application that have been described in the Methodology chapter.

Class Name	Application	Purpose
ScenegraphControls	iSceneBuilder	Set of tools for manipulating the scenegraph in iSceneBuilder
AnimationCommand	iSceneBuilder	A single command to animate changes to the scenegraph
AnimationEngine	iSceneBuilder	Maintains a list of AnimationCommands and automatically updates active commands
NodeVisitor	iSceneBuilder	Recurses through the scenegraph and returns data from or makes changes to each node
UserData	iSceneBuilder	Custom data that can be attached to nodes in the scenegraph
ApplicationDelegate	iPhone Application	Responds to events from the OS and manages the network connection to iSceneBuilder
FileListingTableViewController	iPhone Application	Allows the user to navigate the remote file system
ScenegraphTableViewController	iPhone Application	Represents the scenegraph and allows the user to select a node to edit
FileListing	iPhone Application	Object containing a file type and file name
ScenegraphListing	iPhone Application	Object containing data about a specific scenegraph node
NodeDetailViewController	iPhone Application	View for making changes to a scenegraph node
NavigationViewController	iPhone Application	Allows the user to navigate in the immersive environment

Table 1: Description of custom classes in iSceneBuilder and the iPhone application

Chapter 4: Results

In order to demonstrate the capabilities of iSceneBuilder and the iPhone application for building and managing scenegraphs, two different scenes were created. Each of these scenes had a different purpose, and different techniques were employed to achieve the final result. Both scenes were created inside C6 at Iowa State University and are presented in this chapter. In addition to these two scenes, a potential real-world use case is discussed at the end of the chapter.

The first demonstration of the capabilities of iSceneBuilder was to create a simulated space battle, using models from the original *Star Wars* movies. This scene, which includes several copies of each model, could be used as part of a larger space application or as a standalone model. The goal for the scene was to have six X-wing fighters approach three TIE fighters, which would be escorting an Imperial shuttle. Because the scene is set in space, a star field is an appropriate background.

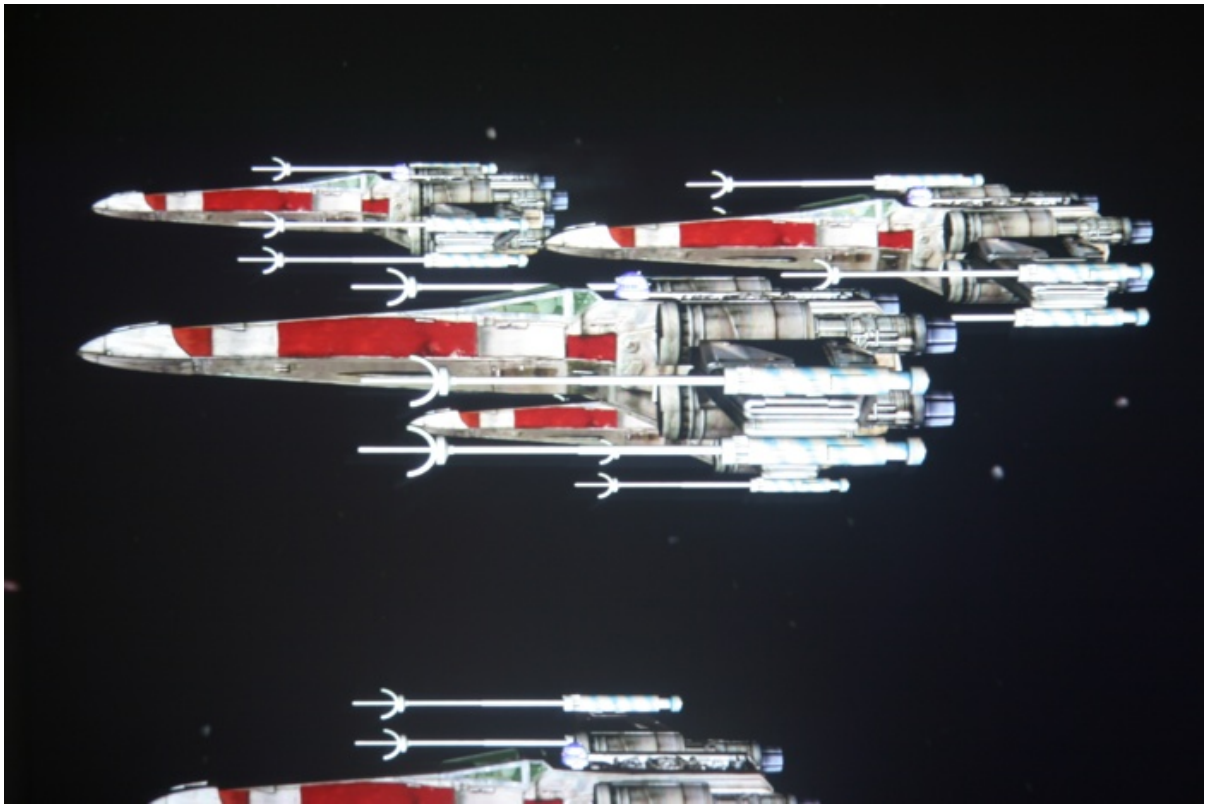

Figure 20: The fleet of X-wings.

The first step in creating this scene was to load a single X-wing model and place it in the scene. As is common with many 3D models, the internal rotation matrix didn't match the desired rotation of the model. Because of this, the first step was to rotate the model so it was upright and facing the correct direction. Once the model was moved into place, five additional X-wings were loaded and configured similarly. A portion of this X-wing fleet is shown in Figure 20.

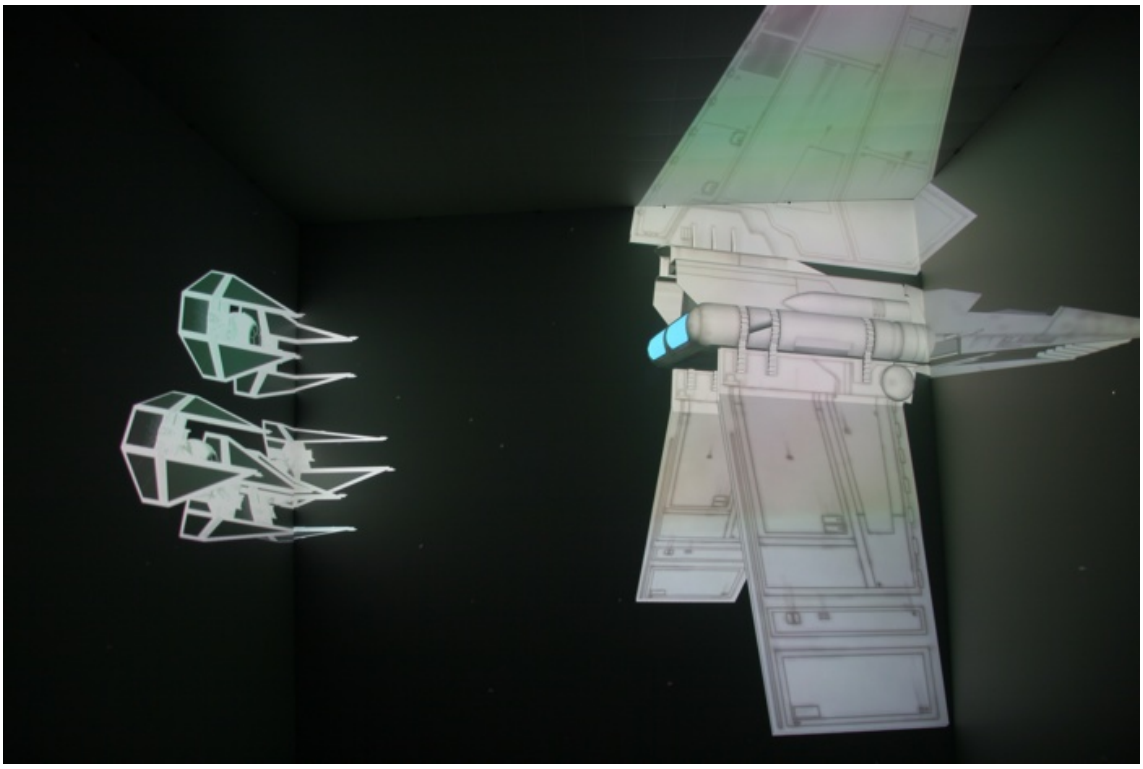

Figure 21: The TIE fighters and Imperial shuttle models.

After the X-wing fleet was configured, the user navigated away from the X-wings to where the Imperial fleet was to be place, then loaded the first TIE fighter model. Like the X-wing models, this model also needed to be rotated to the appropriate orientation before placing it. A total of three TIE fighters were loaded and moved into a tight formation that faced the X-wing models. Finally, an Imperial shuttle model was loaded and placed in between the TIE fighters and X-wing models, as shown in Figure 21. Finally, a star field model was added to the scene. Because this model is considerably larger than the ship models, it serves as a “sky dome” that gives a sense of a background in the scene.

This example demonstrates the ability of iSceneBuilder to load a variety of existing models, place them in a scene and manipulate them to create a new scene as the

user wants. By creating the scene inside the VR environment, the user immediately sees how large models are compared to each other and how the scene looks in VR.

The next scene demonstrates additional capabilities of iSceneBuilder by creating a larger, more complex scene. Rather than create a new scene from nothing, this example recreates a scene from the Virtual Universe [69], a space exploration application created at the Virtual Reality Applications Center. Specifically, the Virtual Universe contains an asteroid field environment, which contains thousands of asteroids in a pseudo-random pattern. The asteroid field scene was originally created using 3ds Max by creating a pattern of several asteroids, then duplicating that pattern many times to generate a larger field of asteroids. One of the significant challenges when creating the original scene was understanding how large the asteroids were and how tightly they should be spaced.

Figure 22: The base set of nine asteroids.

The first step in recreating the asteroid field was to load a small number of asteroids and begin placing them. A variety of asteroid models were used, each with a unique shape and size. In order to create a sense of randomness, each asteroid was rotated to arbitrary values and moved into a position near another asteroid. This base collection of nine asteroids, shown in Figure 22, was saved as a .osg file for future use.

Figure 23: Several sets of asteroids.

Figure 24: The completed asteroid field.

After saving the set of asteroids, a new, blank scene was created. Then, the user loaded several copies of the previously saved collection of asteroids into the scene, as shown in Figure 23. Because each copy was loaded from a separate file, each set of asteroids was grouped underneath a single MatrixTransform. This made it easy to move around groups of asteroids at once. Each group was rotated to arbitrary values and scaled so that they weren't readily recognizable as duplicates. Because iSceneBuilder allows users to manipulate parts of the scenegraph, the user was still able to modify the properties of individual asteroids as necessary, independently of the larger group. Again, after laying out several groups of asteroids this way, the scene was saved as a .osg file for later use. This process was repeated another time to generate the large asteroid field with thousands of asteroids visible at once. A portion of the final scene is shown in Figure 24.

By using the iPhone application to control iSceneBuilder, the user was able to get immediate, real-time feedback about their actions in the immersive environment. This feedback included loading models and moving them inside the scene. Additionally, building the scene in the immersive environment made it possible for the user to realize the size of models, how close they were to each other and what the model would like in its final use. This strongly contrasts with the experience of developing the same scene on a desktop computer, where users cannot easily understand how large a model is or what it will look like in the immersive environment.

A final example of using iSceneBuilder is presented as a discussion for how it can be applied to a real-world situation. One of the key strengths of iSceneBuilder is that users see their changes to the scenegraph in real-time, inside the VR environment. A common use for VR is for engineering design. Rather than building costly and time consuming physical prototypes, VR environments can use existing CAD geometry to give engineers information about the product during the design process.

For this hypothetical use case, a company is placing a new display system by the operator's seat in their vehicle. There are a number of potential locations for this display, but engineers are concerned about how the display will block line-of-sight to critical areas the operator needs to see at all times. With many existing VR tools, an engineer would have to generate a finite number of models with the display in potential locations, then load each model into the VR environment, one at a time. If the engineering team wanted to see the display in a different location, a new model would need to be generated, forcing the team to reconvene at a later time.

iSceneBuilder can easily facilitate this use case, by saving time before the design review session and offering the engineering team more flexibility during the session. Rather than generate a finite set of models, the team would only need a model of the vehicle and the display, as two separate models. At the beginning of the design session, an engineering, using the iPhone application, would load the models of the vehicle and display. Then the team could investigate positions for the display by moving and rotating it in the environment with the iPhone application. When they found a potentially acceptable location for the display, the current scenegraph could

be saved for future reference. By using iSceneBuilder, engineers can investigate an unlimited number of positions for the display without having to leave the VR environment or waiting for additional models to be generated.

Chapter 5: Future Work & Conclusions

This thesis presented a system for creating and manipulating a scenegraph in an immersive environment, controlled with an iPhone. By generating the scenegraph in the VR environment, rather than on a desktop computer, users have a better understanding of how models relate to each other. Users are also able to see their changes to the scene in real-time, rather than being required to make changes to a model, then bring the modified model back into the VR environment.

The first part of this system, known as iSceneBuilder, is an application that runs in an immersive environment. Built using VR Juggler and OpenSceneGraph, iSceneBuilder maintains an internal scenegraph of the current scene that can be modified by the user. Because many VR systems run on a computer cluster, iSceneBuilder is designed to share incoming commands with all nodes in the cluster, so that each computer behaves identically to the master node. iSceneBuilder also is designed with concurrency in mind, as typical computers have multiple processing cores available. When making changes to the scenegraph, iSceneBuilder utilizes *AnimationEngine,* a state-based system for animating changes to nodes in a scenegraph. By animating these changes, users have a better sense of the changes they are making to their environment. Finally, iSceneBuilder utilizes a proprietary TCP/IP socket to communicate with its controlling iPhone application.

Rather than use existing tools for controlling VR applications, iSceneBuilder is controlled with a custom-built iPhone application. This controller application both receives data from iSceneBuilder and sends numerous commands to iSceneBuilder.

Built using Cocoa Touch, the iPhone application has a number of different views, each with different capabilities, that the user can take advantage of. The FileListingTableViewController allows users to navigate through the remote file system, view directory listings and load models into iSceneBuilder. In order to view the scenegraph hierarchy and select models to manipulate, the ScenegraphTableViewController was created. After selecting a node to edit, the NodeDetailViewController presents a set of customized UISliders that allow users to translate, rotate and scale models, as well as changing the transparency of a node. Finally, users are able to navigate in the remote environment by using the NavigationViewController. While this view is active, users take advantage of the built-in accelerometer in iPhone by tilting the device to control their navigation. By developing this application for iPhone, rather than other control systems, users have a small, lightweight, handheld device they can easily use in an immersive environment. Because iPhone has a capacitive touchscreen, users don't need a stylus or other device to interact with the application. Cocoa Touch, combined with the high resolution display, allows for a detailed and informative user interface.

By combining these two tools, users have a powerful system for creating and manipulating scenes from inside an immersive environment. Rather than using a 2D desktop computer for scene creation, which make it difficult to understand the spatial relationships between models, building scenes in a 3D enable users to immediately understand how large models are and how they relate to one another. This can be beneficial, not only for creating new scenes to use in other applications, but also as

a tool for engineering design. Using an iPhone to control the immersive application gave users a tool with a graphically rich user interface that was easy to manipulate. Users weren't required to memorize the functionality of specific buttons or navigate through VR menu systems to control the application. The iPhone helped users control the immersive application without being intrusive in the environment.

If further analysis of the described system was needed, a number of user studies could be performed to quantitatively assess the system. These studies could first analyze how effectively users can build 3D scenes in VR compared to using a traditional 2D tool, such as 3ds Max. In this study, users would be asked replicate a 3D scene using one of the tools while being timed. Feedback could also be gathered from the users while they work and through an exit interview. A similar study could compare controlling the immersive application with an iPhone versus a gamepad, wand or Tablet PC.

To continue the development of iSceneBuilder and the iPhone application, there are several areas that could benefit from additional research and development. Currently, the iPhone application is only capable of manipulating scenegraph nodes that are imported from a file. Support for additional node types, such as lights or particle systems, could be added. Also, the ability to create new OSG Groups in the iPhone application would assist users in managing more complex scenegraphs.

Acknowledgements

I would like to thank everyone who has helped me throughout my time at Iowa State that helped me get where I am today. First, I want to thank my family for their love and support throughout my life — I can't imagine where I would be today without you. Second, Dr. Eliot Winer, both for bringing me on as a graduate student and for allowing me to freely explore my interests, trusting that I would do something interesting or useful.

Also, I want to thank the faculty and staff at the Virtual Reality Applications Center (VRAC) for everything they do to support the students in the lab. Working at VRAC has been a dream of mine since high school, and now that I've had the opportunity to work here, it's better than I ever imagined. This is entirely because of the wonderful people that work at VRAC every day.

Finally, I want to single out two colleagues for their help with my thesis work: Christian Noon for always looking at my work & offering suggestions and Eric Foo for the numerous times he read my thesis & offered advice for improvements.

Bibliography

1. Sutherland, I.E., *A head-mounted three dimensional display*, in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. 1968, ACM: San Francisco, California.

2. *I-O Display Systems: i-Glasses i3TV*. May 23, 2009; Available from: http://www.i-glassesstore.com/ig-hrvpro.html.

3. Cruz-Neira, C., D.J. Sandin, and T.A. DeFanti, *Surround-screen projection-based virtual reality: the design and implementation of the CAVE*, in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, ACM: Anaheim, CA.

4. *StereoGraphics CrystalEyes 3*. May 23, 2009; Available from: http://reald-corporate.com/scientific/crystaleyes.asp.

5. *Barco Reality Stereo Glasses*. May 23, 2009; Available from: http://www.barco.com/VirtualReality/en/stereoscopic/glasses.asp.

6. *Mechdyne Corporation - CAVELib*. May 23, 2009; Available from: http://www.mechdyne.com/integratedSolutions/software/products/CAVELib/CAVELib.htm.

7. Bierbaum, A., et al., *VR Juggler: A Virtual Platform for Virtual Reality Application Development*, in *Proceedings of the Virtual Reality 2001 Conference (VR'01)*. 2001, IEEE Computer Society.

8. Woo, M., *OpenGL programming guide*. 1999.

9. Bargen, B., *Inside DirectX*. 1998.

10. Burns, D. and R. Osfield, *Open Scene Graph A: Introduction, B: Examples and Applications*, in *Proceedings of the IEEE Virtual Reality 2004*. 2004, IEEE Computer Society.

11. *OpenSG*. May 23, 2009; Available from: http://www.opensg.org/.

12. *Autodesk 3ds Max*. May 24, 2009; Available from: http://www.autodesk.com/3dsmax.

13. *Autodesk Maya*. May 24, 2009; Available from: http://www.autodesk.com/maya.

14. *Pro/ENGINEER - 3D Product Design*. May 24, 2009; Available from: http://www.ptc.com/products/proengineer/.

15. *Autodesk AutoCAD*. May 24, 2009; Available from: http://www.autodesk.com/autocad.

16. *SolidWorks :: 3D CAD Design Software*. May 24, 2009; Available from: http://www.solidworks.com/.

17. *Okino PolyTrans*. May 24, 2009; Available from: http://www.okino.com/conv/conv.htm.

18. *Right Hemisphere - Visual Product Communication and Collaboration*. May 24, 2009; Available from: http://www.righthemisphere.com/products/dexp/.

19. *Mechdyne Corporation - Conduit*. May 24, 2009; Available from: http://www.mechdyne.com/integratedsolutions/software/products/conduit/conduit.htm.

20. Jacobson, J. and M. Lewis, *Game Engine Virtual Reality with CaveUT.* Computer, 2005. **38**(4): p. 79-82.

21. *Unreal Tournament*. May 24, 2009; Available from: http://www.unrealtournament2003.com/.

22. Moritz, E., et al., *Usability of multiple degree-of-freedom input devices and virtual reality displays for interactive visual data analysis*, in *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*. 2007, ACM: Newport Beach, California.

23. *Logitech > Gaming > PC Gaming > Gamepads > Cordless Rumblepad 2™*. May 24, 2009; Available from: http://www.logitech.com/index.cfm/gaming/pc_gaming/gamepads/devices/287&cl=US,EN.

24. Dang, N.T., et al., *A comparison of different input devices for a 3D environment*, in *Proceedings of the 14th European conference on Cognitive ergonomics: invent! explore!* 2007, ACM: London, United Kingdom.

25. Humphreys, G., et al., *Chromium: a stream-processing framework for interactive rendering on clusters.* ACM Transactions on Graphics, 2002. **21**(3): p. 693-702.

26. *Visual Decision Platform - Virtual Reality standard software*. June 17, 2009; Available from: http://www.icido.com/en/Products/VDP/.

27. *WorldViz : Vizard*. June 17, 2009; Available from: http://www.worldviz.com/products/vizard/index.html.

28. *OSGEdit - An open editor for an open scenegraph*. May 24, 2009; Available from: http://osgedit.sourceforge.net/.

29. Do, E., *VR Sketchpad, Create Instant 3D Worlds by Sketching on a Transparent Window.* Proceedings of CAAD Futures 2001 (Eindhoven, 2001: p. 161-172.

30. Microsoft. *Windows XP - Microsoft Paint overview*. June 17, 2009; Available from: http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/mspaint_overview.mspx.

31. Zeleznik, R.C., K.P. Herndon, and J.F. Hughes, *SKETCH: an interface for sketching 3D scenes*, in *ACM SIGGRAPH 2006 Courses*. 2006, ACM: Boston, Massachusetts.

32. Lapides, P., et al. *The 3D Tractus: a three-dimensional drawing board*. in *Horizontal Interactive Human-Computer Systems, 2006. TableTop 2006. First IEEE International Workshop on*. 2006.

33. Gardner, H., et al., *Line drawing in virtual reality using a game pad*, in *Proceedings of the 7th Australasian User interface conference - Volume 50*. 2006, Australian Computer Society, Inc.: Hobart, Australia.

34. Andujar, C., M. Fairen, and F. Argelaguet. *A Cost-effective Approach for Developing Application-control GUIs for Virtual Environments*. in *3D User Interfaces, 2006. 3DUI 2006. IEEE Symposium on*. 2006.

35. Gerber, D. and D. Bechmann. *The spin menu: a menu system for virtual environments*.

36. Bowman, D.A. and C.A. Wingrave. *Design and evaluation of menu systems for immersive virtual environments*. in *Virtual Reality, 2001. Proceedings. IEEE*. 2001.

37. Apple, Inc. *Apple - iPod nano*. 2009 June 18, 2009; Available from: http://www.apple.com/ipodnano/.

38. Rossler, A., R. Breining, and J. Wurster, *Three-Dimensional User Interface For Controlling A Virtual Reality Graphics System By Function Selection*, USPTO, Editor. 2004, ICIDO Gesellschaft Fur Innovative Information Systems: USA.

39. Kim, H. and D. Fellner. *Interaction with hand gesture for a back-projection wall*. 2004.

40. Spielberg, S., *Minority Report*. 2002.

41. Cabral, M.C., C.H. Morimoto, and M.K. Zuffo, *On the usability of gesture interfaces in virtual reality environments*, in *Proceedings of the 2005 Latin American conference on Human-computer interaction*. 2005, ACM: Cuernavaca, Mexico.

42. Hasenfratz, J., M. Lapierre, and F. Sillion. *A real-time system for full body interaction with virtual worlds*. 2004.

43. Konrad, T., D. Demirdjian, and T. Darrell. *Gesture+ play: full-body interaction for virtual environments*. 2003: ACM New York, NY, USA.

44. Zhang, R., et al. *Immersive Product Configurator for Conceptual Design*. in *Proceedings of the ASME 2007 International Design Engineering*. 2007. Las Vegas, NV: ASME.

45. Noon, C., et al. *An Immersive VR Application For Interactive Product Concept Generation And Qualitative Evaluation*. in *Proceedings of the World Conference on Innovative VR 2009*. 2009. Chalon-sur-Saône, France: ASME.

46. Neugebauer, R., et al., *Virtual reality aided design of parts and assemblies.* International Journal on Interactive Design and Manufacturing, 2007. **1**(1): p. 15-20.

47. D. Weidlich, L.C., T. Polzin, D. Cristiano and H. Zickner, *Virtual Reality Approaches for Immersive Design.* CIRP Annals - Manufacturing Technology, 2007. **56**(1): p. 139-142.

48. Nintendo. *Wii.com*. June 20, 2009; Available from: http://wii.com/.

49. *PlayStation.com - PLAYSTATION®3*. June 20, 2009; Available from: http://www.us.playstation.com/PS3.

50. Salisbury, K., F. Conti, and F. Barbagli, *Haptic rendering: Introductory concepts.* IEEE Computer Graphics and Applications, 2004. **24**(2): p. 24-32.

51. Hutchins, M., et al., *A networked haptic virtual environment for teaching temporal bone surgery.* Studies in Health Technology and Informatics, 2005. **111**: p. 204-207.

52. Capps, K.W.a.R.P.D.a.M.V., *A Handheld Computer as an Interaction Device to a Virtual Environment.* Proceedings of the Third Immersive Projection Technology Workshop, 1999.

53. iRobot. *iRobot Corporation: PackBot*. June 20, 2009; Available from: http://www.irobot.com/sp.cfm?pageid=171.

54. Gutierrez, R. and J. Craighead. *A native iPhone packbot OCU*. 2009: ACM New York, NY, USA.

55. Wagner, D., et al. *Towards massively multi-user augmented reality on handheld devices*. 2005: Springer.

56. Plimmer, B., *Experiences with digital pen, keyboard and mouse usability.* Journal on Multimodal User Interfaces, 2008. **2**(1): p. 13-23.

57. Park, J., T. Song, and J. Jeon. *Usability analysis of a PDA-based user interface for mobile robot teleoperation*. 2008.

58. Keskinpala, H. and J. Adams, *Usability analysis of a PDA-based interface for a mobile robot.* Human-Computer Interaction, 2004.

59. Adams, J. and H. Kaymaz-Keskinpala. *Analysis of perceived workload when using a PDA for mobile robot teleoperation*. 2004.

60. Norman, D. and B. Collyer, *The design of everyday things*. 2002: Basic Books New York.

61. *Virtual Reality Applications Center*. June 22, 2009; Available from: http://www.vrac.iastate.edu/c6.php.

62. *pthread*. BSD Library Functions Manual.

63. Apple, Inc. *iPhone Dev Center - Apple Developer Connection*. May 24, 2009; Available from: http://developer.apple.com/iphone.

64. Apple, Inc. *iPhone Application Programming Guide*. May 24, 2009; Available from: http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html.

65. Apple, Inc. *CFNetwork*. May 24, 2009; Available from: http://developer.apple.com/iphone/library/navigation/Frameworks/CoreOS/CFNetwork/index.html.

66. Apple, Inc. *UIAccelerometer Class Reference*. May 24, 2009; Available from: http://developer.apple.com/iphone/library/documentation/UIKit/Reference/UIAccelerometer_Class/Reference/UIAccelerometer.html#//apple_ref/occ/cl/UIAccelerometer.

67. Apple, Inc. *Core Graphics*. May 24, 2009; Available from: http://developer.apple.com/iphone/library/navigation/Frameworks/Media/CoreGraphics/index.html.

68. Apple, Inc. *Core Animation Programming Guide*. June 25, 2009; Available from: http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40004627.

69. Newendorp, B., et al. *Development Methods And A Scenegraph Animation API For Cluster Driven Immersive Applications*. in *Proceedings of the World Conference on Innovative VR 2009*. 2009. Chalon-sur-Saône, France: ASME.