The Interaction Framework For Innovation: A Method to Create Reusable Three-Dimensional Interaction Techniques

Andrew Alan Ray

Dissertation submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
In
Computer Science and Applications

Dr. Doug A. Bowman

Dr. Shawn A. Bohner

Dr. Denis Gracanin

Mr. John T. Kelso

Dr. Christopher L. North

April 30, 2008

Blacksburg, VA

Keywords:
3D User Interfaces, Reuse, Virtual Reality, Virtual Environments, 3D Interaction Techniques

The Interaction Framework For Innovation: A Method to Create Reusable Three Dimensional Interaction Techniques

Andrew Alan Ray

ABSTRACT

The field of three dimensional interaction techniques (3DITs) provides a way for users to interact with virtual environments (VEs). Interacting with these environments helps provide the numerous benefits that VEs impart to their users. However, the methods and tools for building 3DITs are not well defined. Because of this, there is little reuse of techniques between different VEs. An example of this is the fact that the only reuse happening in the field of 3DITs is developers reading publications about techniques and implementing the techniques from scratch.

This work investigates the reasons behind the lack of reuse through personal experiences, a literature review, and a community survey. It then provides a solution for creating reusable techniques. This is done through foundational work and a toolkit designed to create reusable 3DITs called the Interaction Framework For Innovation (IFFI). We demonstrate that this toolkit can be used to build reusable techniques by studying the reusability of techniques built with the tool; by conducting both a conceptual and in-depth evaluation on creating reusable techniques. These three methods demonstrate that it is possible and feasible to build techniques that can be reused across both applications and toolkits.

This work is dedicated to Jesus Christ, my rock and my foundation.

This work is in memory of Dr. Sallie Henry. Without her guidance I wouldn't have gone to graduate school.

Acknowledgements

I wish to offer sincere appreciation to those who made this effort possible:

My parents and my family for their support all these years.

My advisor, Dr. Doug Bowman, for guidance and professional development.

My committee members and other faculty who have helped me through my years at Virginia Tech. I would especially like to thank John Kelso for his work over the years on DIVERSE and his editorial work on my dissertation.

Dr. Ron Kriz for funding part of my education and giving me the opportunity to work in the VT-CAVE which led to this work. Without you, the VT-CAVE wouldn't have 10% of the legacy that it currently has.

Patrick Shinpaugh for helping to create DADS and keeping DIVERSE and the VT-CAVE working properly. The VT-CAVE would have fallen apart years ago without you.

The 3DI research group for helping evaluate my work.

Contents

List of figures

Chapter 3

Chapter 4

Chapter 5

List of Tables

Chapter 2

Chapter 4

Chapter 1: Introduction

Imagine sitting in a chair and wanting to get a glass of water. In the real world you have to reach out and pick it up if it is sitting next to you. If it is across the room you have to get up to go pick it up. If it is in another room you have to get up, walk to the room, pick it up, and return to where you were seated.

Now imagine being able to stretch your hand out fifteen feet and grab the water glass, or being able to wrap your arm around a corner to pick up the glass. Imagine being able to look at a top-down view of your room with a small map held in your hands and being able to re-arrange the room by picking up miniature copies of the furniture. Imagine having a three-dimensional (3D) map of a city that provides real 3D detail, allows for scaling and scrolling, and can be used to fly you to a specific place in the city.

These are but a few of the different tasks that can be accomplished with 3D Interaction Techniques (3DITs). 3DITs are the methods used to accomplish tasks in virtual environments (VEs). 3DITs are often what makes the difference between an unusable or non-useful VE, and a usable or useful VE. 3DITs are also prevalent in almost every VE. Not every VE has the features described above, but almost every VE provides some sort of 3D interaction, such as a navigation or travel mechanism.

With this situation, what would one expect to find in the tools used to create VEs? At the very least there should be reusable navigation techniques present, and hopefully ways to help facilitate the creation of the other 3DITs mentioned. However, this is not the case. Tools specifically designed to help build and reuse 3DITs in VEs are nearly non-existent. Instead, the implementation of 3DITs for VEs is mostly accomplished with low-level tools not designed for 3DIT developed, making even the simplest 3DITs difficult (and even painful) to implement.

A quick look at the academic literature shows that the number of different tools available to build VEs is staggering [BOYD99] [CARLSSONC93] [DARKENR95] [DELIGIANNIDIS00] [DYKSTRAP94] [ELLIOTTC94] [FALKOK01] [FELSS98] [GOBBETTIE93] [GRIEPPT02] [HOLLERER99] [HUMPHREYS0] [KOOPERR98] [KOVEDL93] [OLWAL94] [PIEKARSKIW03a] [PONDERM03] [POUPREYVI97] [ROBERTD93] [RORKEM1] [SCHAFFER03] [SHERMANB1] [SHAWC93] [TANRIVERDVI01] [TRAMBEREND99] [WATSENK98] [WILLANSJ00c] [WILLANSJ00b]

[ZELEZNIK91]. This is by no means an exhaustive list. Why are there so many? The answer is that each facility with VR hardware usually creates their own tool and uses it to build their VEs. They usually do not share the tool with others, though there are notable exceptions. These tools mostly focus on abstracting VR hardware so that applications can be ported between platforms. None of them provide support for software reusability, and very few of them have any 3DIT support beyond inserting a draw callback to be displayed in the VE. This means that there is no reliable way to build reusable 3DITs with programs built using the same VR toolkit, let alone different VR toolkits.

There are a few 3DIT tools available (more on these in chapter 2), but they make an assumption that components of complete 3DITs are already available, and do not address the fundamental problem of creating reusable generic 3DIT components themselves. Until there is a way to create reusable generic 3DITs, the problems listed above will not be solvable with current tools and methods. Our research aims to develop a platform that not only allows the development of reusable VEs, but also enables reusable 3DITs to be developed with VR tools. Our work will also complement and interoperate with current 3DIT tool research.

1.1 Definitions

This section is devoted to explaining several of the major terms that are used in this dissertation. Some of these terms are used in different ways in other literature; we cite our source where possible.

Virtual Reality (VR) – A synthetic, spatial (usually 3D) world seen from a first-person point of view. The view in a VE is under the real-time control from of the user [BOWMAN05].

Virtual Environments (VEs) – Same as VR, these terms are interchangeable.

VR tools – These are the tools used to create VEs, and are often used to create interaction techniques. Examples are VRJuggler and DIVERSE.

VR toolkits – This is a common term used in academia that is identical to the VR tools definition.

3D Interaction Techniques – Human-computer interaction methods allowing the user to perform tasks directly in a 3D spatial context [BOWMAN05].

Reusability – In this dissertation, reusability refers to the ability to take an interaction technique

or technique component and reuse it without modification in other VEs.

1.2 Motivation for VR / 3DIT software

VR attempts to produce stimuli for a user's sensory system that replaces the sensory stimuli from the real world. It does this through a variety of different hardware devices. The two main senses actively researched are the sense of sight and the sense of sound. For our eyes, there are head-mounted displays (HMDs) to block out the real world, CRT monitors displaying 120 images a second and using active stereo glasses, and multiple sets of projectors creating an active or passive stereo effect in a single to multi-walled system. For the sense of sound there are headphones (both noise canceling and non-noise canceling), speakers (surround sound), and software to create 3D effects. In order to tell where a user is located, and what action they are currently performing, devices such as magnetic, acoustic, and optical trackers have been created. These devices can provide up to six degrees of freedom (DOFs), and have varying numbers of buttons and/or joysticks that are used to provide input to the computing system. Although hardware is what makes the VR experience happen, it is the software that manages the hardware where most of the development in VR and 3DIT research is done.

3DITs are realized almost solely at a software level. Their traditional modus operandi is to receive information from input devices such as the trackers listed above, and then to display to one or more of the user's senses through software commands that tell the hardware to perform the desired action. One example of a classic 3DIT is the Go-Go technique, which takes information about a tracker held in a user's hand and moves a virtual hand in the world with a non-linear scale. This allows for a user to reach hundreds of feet into the VE, instead of just a couple of feet in the real world [POUPREYVI96].

Managing all of the different hardware devices mentioned earlier can become a real challenge to a VE or 3DIT developer. In the middle 1990's, VR toolkits began to focus on abstracting the displays and input devices away from developers so that they did not have to directly address those issues inside the program. This allowed for VEs created with these VR toolkits to operate in many different hardware configurations without modification.

While this method is very valuable for VE creation, it does not directly address 3DIT creation, because simply managing complex sets of non-integrated hardware by proper software support does not fulfill all of the requirements for building 3DITs. In order to build 3DITs,

developers must be able to transform raw data from input devices into meaningful and useful information that is fed back to the user. For example, simply having your hand represented in a VE is a somewhat complex task. The device information must be queried, a transformation must be made from the device coordinate system to the VE coordinate system, the user's position in the VE must be taken into account, a model representing the hand must be loaded, and the hand model's position must be updated to account for both the user's position and the transformed device coordinates. All of this is just for moving one's hand around in the VE. This does not take into account selecting objects or moving objects that have been selected. Adding these conceptually simple tasks increases the complexity of the requirements, and therefore the complexity of the code, dramatically.

Below are a set of scenarios that show the progression of VR tool research, how 3DITs have been created in the past, and how they are currently created. These scenarios illustrate realistic situations and problems that we and our colleagues have encountered while developing VEs and 3DITs. They are primarily focused on 3DIT development and usage.

Scenario 1: Implementing Ray-casting

Chris, a graduate student, decides to implement ray-casting for selecting objects in his VE application. His VE is based on the GLUT software library, uses an HMD, and provides head tracking using a Polhemus tracking system. After spending time researching how to implement ray-casting, Chris finally gets it working. Bob (a fellow developer in the lab) likes the selection ability and wants to incorporate it into a VE that he has been working on. This would be possible, but Chris tightly integrated the interaction technique and the application together because it was the simplest way to make it work. This causes Bob to have to change both the architecture of his VE program and the architecture of the ray-casting technique that Chris implemented to get it to work in his program.

Chris's advisor likes the selection technique, but wants additions made to it so that it can snap to objects that are close to the ray to help reduce the amount of effort that it takes users to select objects. Chris makes more changes to the architecture and internals of the program to allow for more complicated selection mechanisms to be added instead of the simple line and plane intersection that is used for normal ray-casting. After some work Chris finishes the interaction technique and his advisor approves it.

Chris's advisor likes the interaction technique so much that he tells Bob to incorporate the new technique into his VE. However, Bob has already modified Chris' original technique to fit into his program. The new changes are completely incompatible with several assumptions that Bob made. Because of this, Bob has to start mostly from scratch and implement the interaction technique in a different way.

When the students demonstrate the interaction techniques in both of the different environments to lab visitors, everyone wonders why the techniques seem to be the same but behave slightly differently, and no one can tell exactly what the difference is.

Scenario 2: Implementing Go-Go

After Bob and Chris graduated, their advisor had several more ideas about how to improve selection techniques. His main idea was to use the way people naturally interact with environments and enhance this by taking advantage of the flexibility of VEs. He recruited a new student named Amanda to extend and enhance the environments that Chris and Bob had made.

When Amanda tries to run the programs that Bob and Chris had implemented, the programs don't work because the VR lab had upgraded its tracking system from a Polhemus to an Intersense unit and had gotten newer computers and operating systems. The customization that Bob and Chris had done to make their software work isn't documented, and Amanda can't figure out how to make anything work. She has to spend most of a full semester getting the system back into working order before she can even think about how to start on her original task.

When Amanda finally starts to implement the ideas that her advisor gave her and runs a small pilot study, she finds that she cannot directly compare her work to the previous techniques because of differences in implementation. Because of this she has to re-implement all of their previous work and combine it with her work before she can run a true comparative study on the different techniques.

Scenario 3: Using a modular toolkit similar to modern VR tools

Amanda's younger brother Jed has heard all the horror stories from his older sister, so he finds a university that has something called a VR toolkit that is supposed to help prevent the problems that his older sister ran into. The VR toolkit does indeed get rid of some of the problems, but Jed still has considerable difficulty with implementation issues.

He decides to compare interaction techniques that he has developed to techniques that have been developed at other universities that have different VR toolkits. Since the toolkits

don't interoperate, he cannot simply run the other techniques alongside his own, even if the code is available. This means that he has to implement the interaction techniques from scratch. He isn't able to exactly reproduce the techniques, because they have a number of nuances that the publications on the techniques don't describe.

This isn't the worst thing that Jed has to deal with, however. The VR toolkit that he uses provides good support for working in different hardware configurations, but it does not support 3DIT development at all. Everything has to be built from scratch in all of his applications. The only upside to this situation is that the VR toolkit was designed to be modular and support reuse, so it is easy enough for Jed to use his small interaction technique framework over and over in the different environments written in the same VR toolkit. Jed is also able to integrate a physics engine into the existing VR toolkit without much problem due to a good plugin architecture.

Scenario 4: Another modern VR toolkit scenario

Jed's brother John also attends a university where they have a VR toolkit, which they use with a CAVE display. The CAVE, originally driven by an SGI "graphics supercomputer," was now moving to a system powered by a cluster of PCs. The VR toolkit was designed to help with abstracting devices, but wasn't designed to synchronize program data among the different cluster nodes. Thus, John has real difficulty trying to implement his interaction techniques on the new cluster system.

Scenario conclusions

These scenarios, based on true stories and personal experience, paint a rather grim portrait of 3D interaction development. These scenarios do not present a complete list of problems in VR and 3DIT development; there are several others that were left out since they would only belabor the point. While it is possible to perform research under these conditions, addressing the problems illustrated above should be a high priority for VR/3DIT researchers.

What do these scenarios tell us about VR/3DIT software development? They tell us that there are a few concerns that have been well-addressed by current research trends, but many issues that are still problematic. These issues include the following:

- Often 3DIT developers integrate a 3DIT with an application, which leads to inconsistency among versions of the technique.
- Integrating the 3DIT with the application also causes problems in sharing the 3DIT among different people, even when you have access to the original developer.

- Moving 3DITs from one piece of hardware to another can also be problematic unless they are based on a VR toolkit that supports both hardware devices.
- Even when 3DITs are built with VR Toolkits they are never portable between different VR toolkits.
- Often, there is no VR toolkit support for building 3DITs.
- There are major issues with VR toolkits and 3DIT support on cluster-based systems.

1.3 Building 3DITs

Even with modern VR tools, building 3DITs is difficult. Why is this the case? What is necessary for 3DITs to be developed? What do current tools provide for building 3DITs? Answering these questions will help us discover what gaps need to be filled with software tools.

In order to answer these questions, we present in this section a brief description of several 3DITs, and what must be done to implement them. We will first look at simple interaction techniques, then go on to more complex techniques in order to expose the different parts of the techniques. By looking at these techniques the different requirements for building 3DITs will become clear.

The Go-Go technique

In the real world we naturally use our hands to interact with the world around us. The Go-Go technique allows the user not only to reach out and grab objects that are nearby, but also to reach and manipulate objects at a distance [POUPYREYVI96]. This is accomplished by providing a virtual arm that grows at a non-linear rate as the user stretches out his physical arm.

The steps that have to be performed in order to implement Go-Go are as follows:

1. Load and/or obtain a pointer to the object representing the virtual hand.
2. Determine the user's position in the VE.
3. Obtain the raw device coordinates of the user's physical hand.
4. Map the raw device coordinates to VE coordinates.
5. Calculate the physical hand's distance from the user's body and apply a non-linear scaling function.
6. Update the virtual hand object's position to the scaled distance.

The software used to build the Go-Go technique is not available, and was built from scratch [POUPYREVI97].

The HOMER technique

Go-Go is a useful technique, but it has a few limitations. Because of the non-linear movement it uses to attain large distances, precision is sacrificed when working at a distance. Another problem is that Go-Go has a finite range. In order to address such limitations, the HOMER technique was invented. This technique combined a simple selection technique called ray-casting with linearly-scaled hand movement, in order to get the best of both techniques [BOWMAND97a]. Ray-casting involves simply pointing at an object and then activating a selection trigger. This works very well for selecting objects, but not for manipulating them. By pointing at an object to select it, and then using an approach similar to Go-Go for manipulation, a user can select an object that is far away and manipulate it as if it were next to them. In order to do this, the following steps must be implemented in software.

1. Determine the user's position in the VE.
2. Obtain he pointing direction of the handheld tracking device.
3. Transform this direction vector into a ray emanating from the hand.
4. Calculate intersections between the ray and objects in the world.
5. Select the closest object intersected by the ray if more than one object is selected.
6. Move the virtual hand to the position of the selected object.
7. Attach the object to the virtual hand, and move the virtual hand based on the user's physical hand movements.

The HOMER technique did not reuse any code from pre-existing interaction technique development, and was implemented in a completely different VR tool than Go-Go. Neither the technique nor the tool is available publicly.

The SSWIM technique

The World in Miniature (WIM) technique [CONYWAYA00] is slightly more difficult to implement than the previous two techniques. This technique provides a miniature copy of the world attached to the user's hand. A user can pick up and move the miniature objects in the WIM in order to manipulate the full-scale versions of the object in the world. However, the WIM is usable only in small environments. If a very large environment is used, the miniature objects will be too small to allow easy selection and manipulation, making this technique less usable. The SSWIM technique addresses these problems with the WIM [WINGRAVC06]. This technique allows the user to scale and scroll the WIM so that any part of the world can be manipulated with ease. A handheld wand controls a 3D pointer that is used to select an object or location in the SSWIM. When the pointer moves outside the WIM area, the WIM is scrolled in

the direction of the pointer. To control the scaling of the enhanced WIM, a mouse with a scroll wheel is used. This technique was developed using a customized version of SVE [KESSLERG99] and is not available publicly.

Summary of previous 3DIT research

The prevailing trends of 3DIT development (as shown by these examples) are that techniques often reuse ideas or features from previous techniques, and that they tend to increase in complexity over time. Thus, the need for reuse is great, but is not supported with current tools (as we have argued in the scenarios above). The increasing complexity indicates that the implementation problems for 3DITs will only get worse if new tools are not developed.

In addition, these example 3DITs were all built with different software packages, were not built with reuse in mind, and are not available for reuse. Are these isolated cases, or is this an overall research trend? An analysis of the proceedings from the first symposium on 3DUIs reveals several interesting facts. Out of the 25 papers, only five of the papers used or created tools that are available to the public. Only one of these papers exclusively used freely-available software *not* developed by their research group. Nine of the papers didn't even mention the software tools used in creating the interfaces or techniques. Six created their own tools and did not make them available. An additional four used tools that their group created, but were not available for public review. One paper was pure theory and had no need to mention VR tools.

Software to build 3DITs is extremely varied and is rarely reused. From personal experiences and from conversations with other developers of 3DITs, building techniques is very difficult to perfect and tedious. When a developer wants to reuse an interaction technique, it must be implemented from scratch in the worst case, and cobbled together using existing source code that has already been written in the target toolkit in the best case. Even if there is a well-designed toolkit, there is precious little support for managing the complexity of 3DITs. Essentially, the prototype tools that were created to build these prototype 3DITs are the current state-of-the-art of VR tools.

1.4 Clarification of reuse

Software reuse is a general term that has different meanings. Each of these meanings usually maps to a specific sub-area of the Software Engineering (SE) reuse community. The areas of

reuse in the SE community that relate to this are domain, process, architecture, and module reuse.

At the highest level there is knowledge and domain reuse [SUTCLIFFEA00] [FRAKES98]. These methods seek to provide a "big picture" view of a specific area of work. These areas step away from the low level tools used to implement software and start looking at the background knowledge necessary to understand a specific domain. By being able to reuse this knowledge, it is possible for conceptual reuse to take place between different tools. This work does perform an informal domain mapping of VR toolkits and 3DITs. This was used to help develop both process and architecture reuse methods for building reusable techniques.

Another high level reuse area is process reuse. This area deals with how the processes used to develop software are reused [HOLLENBACHC96]. One example of a process used to develop software is the spiral model [BOHEMB88]. This work does not provide a specific method for developing software like the spiral model, but it does provide several steps that need to be added to whatever software process that will be used in a specific development effort. In addition to augmenting a process to develop techniques, it also ties this process together with component development because developing reusable techniques is realized through building reusable components. This is similar to the connection between processes and components outlined in Kellner's work [KELLNERM96].

Software architecture reuse allows developers to help reuse knowledge between systems in the same product family [MEDIVIDOVIC00]. While there has been some research into how similar architectures can be described, there is no way to reuse the actual architectures themselves. This work has analyzed the architectures for both VR toolkits and 3DIT specific tools, and has developed a method to allow for interaction techniques to work with any architecture in this specific domain. This model allows for tool reuse between frameworks written in different domains similar to the toaster model [BOEHMB96].

The lowest level of reuse is component based. There are several different types of reuse at this level. One is how components from one library can be used in another library without modification [DIAZR89]. A common question asked at this level is how to describe, search, and find components [FRAKESB94]. This work does not seek to provide a method to reuse components from one VR or 3DIT toolkit to another. It seeks to provide a higher level

architecture abstraction so that components built with IFFI can be used in any of the other architectures without modification.

Our definition of a reusable interaction technique contains four parts. From the acquisition of domain information we were able to identify four specific aspects of reuse that are important to the domain of 3DITs. The first part of reuse is that techniques must work between applications in the same VR toolkit. Evidence of tight integration between techniques and applications will be shown in future sections in this chapter. This definition is necessary because in order to allow for techniques to reusable they must be disconnected from the VE application they are used in.

The second part of the reuse definition is that techniques must work between applications written in any VR toolkit. In order for tools to be truly reusable they must be able to interoperate with any VR toolkit.

The third part of the reuse definition is that techniques must be reusable at a component level. This means that copying and pasting source code from one technique to another is not optimal, and that it must be possible for techniques (and parts of techniques) to be added or removed from the application during runtime. Defining exactly what a component consists of is the reusable component structuring step in the DARE approach. A description of a reusable component can be found in the plugin section of chapter four.

The last part of the definition of a reusable technique is that it must be possible for the technique to be broken down into multiple parts that can communicate with each other. These parts must be reusable at a component level inside other techniques. For example, if technique A contains components X, Y, and Z, then when technique B is built, it can use X and Y from technique A without having to modify X and Y.

In conclusion, this work is focused on reuse, but is mainly focused on providing a method for reuse at the architecture level within the VR toolkit and 3DIT domain. By providing software reuse at this level, it is possible for 3DITs built with IFFI to work in any toolkit in this domain. Lastly, this work does use component based reuse, but does not focus on making these components interoperable with other VR toolkits that use component based reuse. It also does not have a solution for how to describe, find, and reuse plugins. When we refer to a reusable technique it means that this technique can be used in a component-based manner in multiple

applications that can be written with any VR toolkit, and that parts of the technique can be reused in other techniques in a component based manner.

1.5 Research questions and approaches

Considering the arguments above, it is clear that there is considerable room for improvement in the development of 3DITs. The biggest contribution that can be made is in the realm of software development. Specifically, software that focuses on enabling the development of *generic, reusable 3DITs* will greatly improve the current situation.

At the highest level, our research question is how to come up with a better way to develop 3DITs. This question is too high level to be useful for performing reasonable research, so we break this question into three sub-questions. These questions, and the related approaches we will take to address them, are as follows:

Question 1

Why aren't current tools sufficient for the easy development of reusable 3DITs?

Why does such a problem exist? What is the best way to start working towards a solution for this problem?

Approach 1

Current tools are not sufficient because the focus has almost always been on the end product, not on what is used to improve the creation of the end product. Understanding the current state-of-the art through literature review, surveys, and the study of pre-existing tools will provide a foundation to start the design of a new tool.

Building better tools is never glamorous, but is necessary to further the development of 3DITs and interactive VEs in general. Using a programming language analogy, VE development is around the assembly language level of maturity. Not everything has to be done in excruciating detail for each program, but applications and techniques must be completely rewritten to work with a different tool. There is also very little support for any operation beyond the basics, and no concept of a standard template library. We have studied the current state-of-the-art and the needs of the 3DIT community to set the stage for a new 3DIT development tool; we report on this in chapter two.

Question 2

What are the requirements for a tool that does support the creation of reusable and generic 3DITs? How do we design a tool that fulfills these requirements, and addresses the problems identified in approach 1?

Is it possible to design, build, and demonstrate a working tool that will address the concerns we identify?

Approach 2

3DIT development should be based on a solid VR tool foundation. Current VR tools must be evaluated and reused where applicable to help build the 3DIT tool. Beyond this, new requirements for 3DIT development must be addressed in the design of the new tool.

We performed an evaluation of current VR tools to determine if any of them could provide the necessary foundation for our new tool; chapter three presents the work we did to adapt and extend a current VR tool as a foundation. Our tool (the Interaction Framework For Innovation - IFFI) is designed to integrate with the foundational VR tools in order to provide the proper structure and framework for reusable generic 3DIT creation. The requirements for our tool and the design of IFFI are described in chapter four.

Question 3

Can we demonstrate that the tool developed in answer to question 2 allows for reusable techniques to be created?

Is it possible to show that techniques built with the tool from question 2 are reusable?

Approach 3

Sample techniques and developer evaluations can be used to demonstrate that the tool produced in step two can be used to create reusable techniques.

In order to demonstrate this tool can produce reusable techniques, sample techniques must be built with IFFI. The techniques built with IFFI and the examination of their reuse is detailed in chapter five. Also, we must be able to demonstrate that developers can build reusable techniques with IFFI. We present our evaluation of IFFI in chapter six.

1.5 Benefits of reusable 3DITs

Improved reusability will benefit the 3DIT community in several ways. First, one recent research trend in our group has been the use of "flavors" to enhance common interaction techniques. For example, the standard ray-casting technique allows the user to point at an object and select it, but this technique can be enhanced by changing the color of the ray when an object can be selected, or snapping the ray to the nearest selectable object. Such "flavors" can increase its usability, but modifying a monolithic piece of code a each time a tweak is desired can be a time-consuming task that can require changing the design of the 3DIT. With reusable, modular 3DITs it will become much easier to take a working 3DIT and create a new flavor.

Second, in order to be able to implement and evaluate 3DITs, a short development cycle is desirable. This would allow researchers and developers to focus on the 3DIT itself, not on the implementation for the 3DIT. By using reusable components, people can rapidly evolve 3DITs in a manner not normally possible with current tools. Currently, a moderately complex 3DIT can take months to build and evaluate. What seems to be a simple tweak can often require changing the entire design for the 3DIT. By shortening the development cycle through reusable 3DITs, the development time will be lowered, thus providing an important benefit to 3DIT developers.

Finally, reusable 3DITs will enable researchers and developers to share their work with colleagues. Currently, it is almost impossible for a 3DIT developed at one research center to be used at another without significant code-level rework. This prevents direct comparisons of techniques developed by different groups and the sharing of techniques so that they can be extended. Reusable 3DITs, on the other hand, make possible the compilation of a library of common techniques, which will allow for much greater interactivity with VEs and will reduce the barrier of entry for developing new 3DITs.

1.6 Conclusion

In this chapter, we have introduced and motivated the topic of 3DIT development tools, showing the need for further research in this area. We will address three research questions in this area using the approaches outlined above. The flow for the rest of the dissertation is to investigate related work (chapter 2), describe the foundational work necessary for this research (chapter 3), describe the IFFI system (chapter 4), show several 3DITs built with IFFI (chapter 5), discuss the evaluation of IFFI (chapter 6), and lastly finish with conclusions, future work, and most

importantly the contributions that this research makes (chapter 7).

Chapter 2: Exploring why techniques are not reusable

2.1 Introduction

The first research question that this work addresses is why current tools are not sufficient for the easy development of 3DITs. This chapter addresses this research question by looking at the progression of research that has led to the current state of developing 3D interaction techniques and what the current state-of-the-art technologies are for building techniques. It does this through a literature review and a community survey. The literature review shows what has led to the current state-of-the-art, and also goes over the current state-of-the-art. The survey obtains opinions from experts and practitioners in the field to verify the conclusions from the literature review. These two activities combined with our personal experiences (discussed in chapter one) provide some answers for the first research question.

2.2 Literature review

This section focuses on three major areas related to and including 3D Interaction Techniques. The three areas are 3D graphics tools, VR tools, and 3DITs. Each of these areas is further broken down into more detailed sections except for 3D graphics development.

2.2.1 3D Graphics Development Tools

The introduction of PHIGS at the end of the 1980s and the commercial release of OpenGL in 1992 were the culmination of decades of computer graphics research and allowed for common desktop computers (in the mid 90s) to provide the capability to display VEs [HOWARDT91] [SEGALM98]. Other tools that were part of this era were CORE and GKS [NICOLCJ81] [ENDEDEG86]. More modern tools have all been based on OpenGL. However, graphics by itself cannot provide everything for a VR experience. Graphics tools are not typically concerned with input devices, sound, and tracking equipment. However, all of these are necessary for VR. The current state-of-the-art in VR graphics tools lies with OpenGL, Open Scene Graph, and VTK [OSG06] [VTK06]. These are graphics tools that are used by VR tools to provide the graphical part of VR.

2.2.2 VR tools

VR requires graphics, but also requires many other components to be true VR. This section will

focus on the beginnings of VR environments, the first tools available for creating VEs, components that make up VR tools, current VR tools, and current research trends with VR tools.

2.2.2.1 The first VR tools

VR tools are designed to help reduce the complexity of developing VEs on dramatically different hardware. One of the first VR tools was CAVELibs. This tool focused mainly on handling the display and provided input capabilities through a software program called TrackD [VRCO1] [PAPED96]. Other toolkits in this era were the MRToolkit [GREENM1], WorldToolKit [SENSE81], and Vega [VEGA1]. Each of these toolkits usually had a specific focus (graphics, scene graphs, scripting) that it handled reasonably well. However, there were usually one or two areas in the tool where it was deficient and would not meet all of the needs of researchers. This led to many people creating tools that addressed their own needs.

The main focus of these tools was on *abstraction*. The early VR tools showed that there was a need to abstract displays, input devices, and software incompatibilities to help provide access to the graphical tools that developers desired without having to worry about the lower-level hardware issues. Some of these tools focus on providing access to a specific scene graph, and some on augmenting existing scene graphs. Others work on collaboration, while still others provide cluster-based visualization. The prevailing trend in VR tool development is that the tools were built to support the equipment at the site they were developed at, and were designed to be used on certain types of projects. Some of them are flexible; most are not.

Because of the differing foci of tools and a lack of standardization, several more VR toolkits were developed in academic institutions. Some of the better-known tools were VRJuggler [BIERBAUMA01], SVE [KESSLERG00], DIVERSE [KELSOJ02], Alice [CONWAYA00], Lightning [LIGHTNING01], and Avango [AVANGO99]. These tools were developed in academia because there was (and still is) not a large amount of commercial interest in VR technology. There are many other VR tools that have been produced, but at a high level, these duplicate the same functionality as the ones listed above [BOYD99] [CARLSSONC93] [DRAKENR95] [DELIGIANNIDIS00] [DYKSTRAP94] [ELLIOTTC94] [FALKOK01] [FELSS98] [GOBBETTIE93] [GRIEPPT02] [HOLLERER99] [HUMPHREYS0] [KOOPERR98] [KOVEDL93] [OLWAL94] [PIEKARSKIW03a] [PONDERM03] [POUPREYVI97] [ROBERTD93] [RORKEM1] [SCHAFFER03] [SHERMANB1] [SHAWC93] [TANRIVERDVI01] [TRAMBEREND99] [WATSENK98] [WILLANSJ00c]

[WILLANSJ00b] [ZELEZNIK91]. Each one of the previous twenty-nine VR tools listed has a unique niche, but none are specifically targeted at 3D interaction technique development. Other than providing basic display and input device abstraction, there is no standardization between the approaches that the different tools use.

As examples of the different approaches taken by current VR tools, we compare Alice, VRJuggler, and DIVERSE. VRJuggler requires developers to derive from an application object that provides a handful of standard callbacks (configure, draw, and update) to access the VE. The driver program for this class has to configure and then hand off control of the program to the system, preventing VRJuggler from interoperating with other software. DIVERSE can also operate in the manner that VRJuggler does, but provides the ability to use as little or as much of the system as possible. This allows it to easily integrate into existing programs whereas VRJuggler requires a complete restructuring. VRJuggler and DIVERSE will be described in more detail in chapter three. Alice takes a completely different approach and tries to remove the complexities of C++ programming from the equation. It allows people to work within a higher-level scripting language and manipulate objects with commands such as move up, move down, etc. It is a tightly-integrated and specialized system with its own command language, integrated development environment, and graphics subsystem.

The next section will focus on the different areas that VR tools have historically supported.

2.2.2.2 Components of VR tools

This section breaks down the different parts of the VR tools and shows what is available to current 3DIT researchers. The main focus for most of the toolkits is graphics, followed by input, and then a variety of niche areas such as clustering, sound, portability, support for 3DIT development, and abstraction of low level concepts. Most VR tools have support for some level of graphics and input, but very few have support for all of the earlier features.

Graphical support in VR tools

In general there has only been one graphical package that has been supported. This is the standardized OpenGL language. DirectX has never been adopted by the VR community because it is not available on platforms other than Windows. VR has its roots primarily on SGI UNIX machines, and has evolved to using Linux workstations. Also, during the mid to late nineties

Windows was virtually unusable for serious software development and usage. Newer versions of Windows are much improved, but still have major issues such as security.

The way in which graphics have been supported is generally done in one of two ways. The first way is support like the GLUT toolkit offers. This is registering a callback to be called at a specific draw time in the application loop. This can be done procedurally like in GLUT, or it can involve deriving off of a C++ class that works in a similar manner. The second way to provide support is access to a higher level graphical toolkit that is built on top of OpenGL.

Historically SGIs OpenGL Performer has been a mainstay with traditional VR tools because it is a high level scene graph that draws OpenGL. This is allows for less work than with traditional OpenGL programming. It has been used in VR tools such as VRJuggler, DIVERSE, and Avango. Commercial support for OpenGL Performer has been discontinued and because of this several of its users are starting to embrace Open Scene Graph. Other scene graphs that have been used in VR are OpenSG, VTK, and Coin. The more generic the VR tool, the more scene graphs it will support. Some VR tools such as SVE only support their internally developed scene graph.

Scene graphs are normally used in VR tools in a couple of ways. The first way is just to provide a raw node that has nothing other than rendering capabilities. The second way is to provide a bit of structure to the scene graph to support general concepts. The majority of these are structured so that there is a high level container node, and then two child nodes. One of these children has navigation information, and the other does not.

Rendering OpenGL and scene graphs normally takes place on a variety of different systems. The ability to display on three or four walled systems, and on walls with different angles is necessary. Stereo also has to be supported in different ways. Also, coordinating rendering on multiple different computers to all render in a coherent manner is necessary in some cases.

Input device support in VR tools

Most VR tools have support for input devices. It is natural to want to take user input and have it influence the world, and different input devices have to be supported for this to be possible. Input has generally been covered in a generic manner given the myriad of different VR input

devices. In general input corresponds to values that are being continually sent to the system (tracker input) and discrete input events (button presses). The way input is accessible to the system is generally done in one of two ways. The first way is to push the input to the program, and the second way is to have the information available for the program to query when it needs it.

The VRPN project supports the widest range of different devices, and focuses solely on these devices. It is designed to be used in conjunction with existing VR tools [TAYLOR01]. Other projects that have generic device support are VRJuggler [BIERBAUMA01] and DIVERSE [KELSOJ02]. VR Juggler requires that developers declare the types of input they are going to use (joystick, buttons, etc) in their code, and then has them query the information in the code. DIVERSE simply uses untyped chunks of memory and requires that developers reconstruct the data into meaningful device information.

Niche areas

There are many different areas that only one or two tools cover or try to address. None the less, these are important areas to mention because they can affect 3DIT development. This section goes over some of the more important niche concepts. This list includes scripting, higher level concepts, clustering, sound, and haptics.

Scripting – In order to get away from the compile, run, change, and repeat cycles higher level languages that are more flexible are starting to be used. Alice [PAUSCHR95] is the main user of this language, and VRJuggler and DIVERSE both have python bindings for their APIs.

Higher level concepts – Moving away from the XYZ coordinate systems and transitioning to more natural English language concepts is the distinguishing mark of Alice. This allows for developers to create and manipulate VEs without having to get into the gritty details of computer graphics.

Clustering – In order to coordinate many different machines together several different approaches have been taken. One method is to synchronize the input data on all of the machines and hope that the environment stays the same, and the other method is to force synchronization at the scene graph level or at the graphics level. VRJuggler is an example of the input synchronization method, while Avango and Chromium [HUMPHREYS02] are examples that

provide synchronization at the scene-graph/graphics level.

Sound – Adding sound to VEs makes is one way to make them more realistic. Support for environmental sounds, doppler effects, and 3D sound support are some of the most important types of sounds in a VE. VRJuggler along with several other projects have support for this, as does the OpenAL project. The OpenAL project is a generic sound architecture that can be incorporated into existing VR tools.

Haptics - Adding the sense of touch to VEs adds another level of realism. Haptic devices are not mainstream in the VR community, mainly due to cost. Because of the lack of availability, support form them is rarely incorporated into existing VR tools.

Component reuse – Reusing components that are pre-built with any particular VR tool is the first requirement for any modern form of software reuse. DIVERSE is one of the few VR tools available that has rudimentary support for this. All other VR toolkits require copying and pasting of source code to implement reuse.

Portability – The ability to develop and run VEs on multiple different operating systems with no changes is desirable because many users don't understand or appreciate the amount of work cross platform code takes. VR Juggler embodies this concept by creating the VPR, the virtual portable runtime system that is completely cross-platform. This is useful, except it creates considerable maintenance overhead. Most other VR Tools make sure that their tools work on different platforms and leave the rest to the developer.

3DIT support – SVE is virtually the only VR tool that has gone beyond providing just displays and input device support to provide utility functions for 3DITs. Some of these are a body centric scene graph structure kept up to date with tracker information, and object query / collide functionality.

2.2.2.3 Current VR tools and current research directions

The major point of this section is that although there are new trends (listed below) starting to come into vogue, there has been no major innovation in the VR tool research knowledge since the era when the CAVELibs, VRJuggler, and DIVERSE projects were created. The focus in VR tools has stayed with abstracting VR hardware and providing little support to other research areas that depend on VR (such as 3DITs).

From the pre-existing tools such as the CAVELibs a common way to build VEs was needed. Many institutions needed tools that were easier to use, more flexible, or simply didn't cost as much as the CAVELibs cost. The creator of the CAVELibs actually created an open source toolkit called VRJuggler to address these issues. Many other institutions took the same approach. DIVERSE and SVE are just two examples out of the dozens that could be cited. Another major change is the trend towards open source development. The first CAVE software was closed source and the graphics system that it used was also proprietary. Most of the newer systems (VRJuggler and DIVERSE) are open source and are free for others to download and try at their universities.

The decline and fall of SGI both in software and hardware performance has led to the next research trend in VR software. This is moving towards using clusters of off the shelf components to power VE's so that they can provide the same functionality that traditional big iron provided. However, as many VR designers found out, parallel computation and the software used for it is an open problem and there is no silver bullet.

Some of the earliest cluster solutions were a surround screen projection system done by Paul Bourke[BOURKEP00], and a version of VRJuggler (NetJuggler [ALLARD01]) retrofitted to use a standard parallel computation language called MPI. Through some unpublished work here at VT, performance between MPI and non MPI clustered systems used for VR purposes only differs by around 10% in favor of the non-MPI systems. This allows researchers to choose which method is better for them without significant tradeoffs.

Most cluster systems prefer to use their own custom solution in order to provide different synchronization methods between all of the machines in the cluster. These synchronization methods vary from system to system. Sometimes it is focused on the swap buffers function, other times it is done at a different level such as a frame counter. Some systems such as Avango focus on scene graph synchronization instead of lower level graphics issues such as the swap buffers. Avango does this by creating network aware containers for all of the different scene graph nodes. This is a good solution as long as the scene graph does not change its implementation or add new features. Other systems focus on synchronization of input devices and navigation information.

Another trend in clustered visualization is large hi-resolution walls. These single walls are often just several desktop computers powering multiple projectors. These computers are then

synchronized together to provide a coherent display system. Chromium is an example of a well known tool that has been built in order to provide support for these displays. It takes a novel approach to the display system by intercepting the graphics calls before they are sent to the graphics card to be rendered, and instead routes them over the network to be rendered elsewhere. By doing this it allows for multiple computers to render the same scene, or for one computer to render the scene and distribute the results to other computers.

2.2.2.4 What VR tools have provided for building 3D Interaction Techniques

Very little has been said about how VR tools have related to 3D Interaction development, particularly in the area of what tools have provided. This has not been addressed for a simple reason, VR tools rarely provide support for 3DIT development. Virtually every VR toolkit available for study (DIVERSE [KELSOJ02], VR Juggler [BIERBAUMA01], SVE [KESSLERG00]) provides essentially the same capabilities as the others at a high level. They all provide callbacks where you can place code to draw your application, and to update the code in the application. They provide the ability to have device information pushed via callbacks or allow it to be pulled to in an update callback. This device information is formatted differently for each VR tool, but it can be generalized to a variable number of DOF data and button presses. SVE also provides a scene graph structure that helps associate graphical objects to device information. It also provides easy object storage, retrieval, and manipulation capabilities. This is the most advanced support that a VR tool has ever offered for building 3DITs. However, SVE has several downsides that make it unsuited for further development. One is intellectual property problems that prevent it from ever being used outside of a handful of developers. It also does not support clusters without custom modifications. These may seem like technicalities, but they are of major importance to the research community. The largest flaw with SVE is that its foundation is not modular, is not designed for any modern type of component reusability, and cannot be added to SVE because of the intellectual property issues.

2.2.2.5 VR toolkit reuse summary

From the previous sections it possible to determine that VR toolkits do not provide much support for the four types of reuse that this work targets. The reuse of VR toolkits can be found in table 2.1. This shows that most toolkits provide reuse only at the copy/paste level and that hardly any tools have provided support for loading components.

Tool	Component Reuse	Between Apps	Between Toolkits	Between Techniques
VRJuggler		Copy/Paste		
DIVERSE	X	X		
SVE		Copy/Paste		

Table 2.1 Reuse characteristics for VR toolkits

2.2.3 3D Interaction Techniques

2.2.3.1 Early 3D interaction technique history and tools used to build them

While computer graphics algorithms and hardware were increasing in maturity, and VR tools were being created, researchers were looking at ways to interact with VEs. This is where 3D interaction techniques fit into the technological progression of VR toolkits. Some of the best known classical interaction techniques are Raycasting [JACOBYR94], Go-Go [POUPREYVI96], and the WIM [STOAKLEYR95]. These techniques are used in the areas of selection, manipulation, and in some cases navigation. These techniques were developed between the years of 1994 and 1996. All of these techniques were developed using SGI hardware, 6 DOF trackers, and were done with HMD devices.

The three interaction techniques mentioned earlier did not originate with the CAVE even though it was available at that time. While there have been CAVE specific interaction

techniques such as the Step WIM [LAVIOLAJ01], there has never been a major focus on creating interaction techniques solely for CAVE type systems. This is probably due to the hardware and software complexities that a CAVE introduces over being able to develop for a single desktop, or it could be due to the fact that HMD based interaction techniques generally port well from one VR environment to the other.

From these interaction techniques hybrid techniques such as the Voodoo dolls and HOMER techniques were popularized in the late 1990's [BOWMAND97a] [PIERCEJ99]. These techniques took parts from previous techniques and put them together to come up with more useful techniques. HOMER combines Ray-casting and body centric hand movement in order to combine the selection strength of Ray-casting and the strength of natural manipulation into one technique. The Voodoo dolls technique allows users to make a smaller copy of a far away object that they can manipulate combining both a WIM and glove based interaction techniques.

From these hybrid technique approaches, another trend is starting to emerge. This current approach is known as flavors. This is a method that takes existing interaction techniques and adds little tweaks to the existing technique to help improve usability. A small example of this is using the ray in ray-casting to snap to objects close to the ray to help with selection. Another example would be using the wand tracker for orbital viewing instead of the head tracker. A last example would be the SSWIM technique developed by Chad Wingrave [WINGRAVEC05]. This technique adds scaling and scrolling to the WIM technique. These additions greatly enhance the technique by allowing it to work in large environments where the existing WIM technique does not work well.

The developers of these interaction techniques often worked with or were creators of the VR tools they used to create their techniques. None of the tools created or artifacts created with the tools are publicly available for study and review. Because of this there has been a lack of feedback into the general VR tool research progression. This is a problem because there has been a large focus on improving usability in VEs with 3D interaction techniques, but very little focus on improving the tools used to create 3D interaction techniques. The next section deals with the few tools created specifically for developing 3D interaction techniques.

2.2.3.2 The first tools devoted solely to 3D Interaction Technique development

As with graphics research, there has been a lapse of time between generation of knowledge from research and the availability of the knowledge in a reusable manner through generic tools. As

stated in the previous section, early researchers into 3D interaction techniques ran into the problem of not having tools and created their own. Most of these tools were created alongside early VR tools and had a strong overlap with what they provided [KESSLERG99] [STOAKELYR95]. Because of this overlap, the major focus for these tools was not perfecting ways to build interaction techniques, it was to produce a system that allows for research to be conducted.

From these VR tools used to build techniques came three different approaches to building interaction techniques. One was the INQUISITIVE system [BOYDD99], the next is CHASM [WINGRAVC05], and the last is NiMMit [VANACKEND06]. These systems strive to make interaction techniques easier to develop by building techniques built out of multiple components. These components can theoretically be reused between techniques. None of these publications demonstrate components built with one technique being used in a different technique. This claim has only been stated at a theoretical level. The only difference between these tool's definition of reusability and the one used in this work is that these that these tools do not focus on making techniques reusable between multiple VR toolkits. These tools focus on GUI based tools to break down the different actions that make up interaction techniques, and focus on allowing a focus on the flow of information through different states in the program. They are also both built on top of existing VR tools (SVE for CHASM and MariGold for INQUISITIVE). How the states are represented in interaction techniques is very different between the two systems, but the same overall idea applies [WILLANSJ00b]. Following in the footsteps of how 3DITs have been built in the past, the actual tools themselves are not available, only publications on them. The advantage of these tools is that they provide a path for building more complex 3DITs, but their methods have one problem. This problem is that they do not provide any better way to build the components that they use to compose 3DITs. These components still have to be built from scratch using existing methods, which can be rather painful. And these components are not designed to be reusable with other VR tools also. Another issue is that some of these tools generate code for the user. This generated code can be difficult to understand and maintain over time.

The next section deals with an alternative approach to working with 3DITs by looking at specification and configuration issues for techniques using a higher level language than what has currently been used in the majority of VR tools.

The reuse characteristics of these 3DIT specific tools can be found in table 2.2. These tools are the closest to fulfilling the reuse requirements that this work requires for its definition of reuse. The major difference between this work and these 3DIT specific tools is that it focuses on allows techniques that are built to be reusable instead of just building techniques for a specific toolkit. The other difference between these tools and IFFI is that they have not demonstrated the reuse of components between techniques reuse feature that that they claim to provide.

Tool	Component Reuse	Between Apps	Between Toolkits	Between Techniques
CHASM	X	X		T
INQUISITIVE	X	X		T
NiMMiT	X	X		T

Table 2. 2 Reuse characteristics of 3DIT specific tools

2.2.3.3 Specification of VEs and 3DITs

Another research direction that some researchers are starting to take is the specification of 3DITs in applications. This research area deals with using XML to specify configuration options for VR applications and 3DITs. The advantages of this approach is that none of the lower system level details have to be handled by the user, and authoring tools can be made to abstract many of the difficulties of generating complicated XML. The downside to this direction is that it does nothing to help creation of 3DITs, only to allow configuration of existing components. This research front is nothing more than a set of tools that are created to work on a black box. While this is a valuable addition to the research field, its impact will be very limited until research happens that will start to improve the availability and to reduce the cost of developing 3DITs that make up these black boxes. There are two main research directions in this area. Contrigra is an

X3D based system that deals mainly with VEs themselves [DACHSELTR02], and InTml that deals with configuring 3DITs by specifying a data flow of information through existing components [FIGUEROAP02] [FIGUEROAP01]. Contigra is based on top of X3D based applications while InTml is designed to augment existing VR tools.

The reuse of tools in this approach can be found in table 2.3. This table shows that these tools are more reuse focused than traditional VR toolkits, but they have not demonstrated two of the four reuse characteristics.

Tool	Component Reuse	Between Apps	Between Toolkits	Between Techniques
InTml	X	X		T
Contigra	X	X		

Table 2. 3 Reuse characteristics of specification level tools

2.2.3.4 Current 3D interaction tool development status

The state of interaction tools for 3D interaction techniques is not mature. This can be shown by two different measures. First is the state of VR tools themselves. 3D interaction techniques are normally built on top of VR tools so that they can influence the VE. Because VR tools aren't well understood and available, it would be improper to say 3D interaction tools are mature. From the myriad numbers of VR tools listed above it is clear that there isn't one major source available. The second measure is looking through publications and generating statistics about the tools used in published 3D interaction research. The analysis of the 3DI symposium papers from Chapter 1 should also be considered.

The lack of reasonable artifacts and tools to recreate artifacts is disheartening. While it is theoretically possible to recreate many of the different techniques described in the papers without their implementation, there are always nuances and subtle differences that are usually lost in translation.

2.2.4 Summary of literature review

It is clear that there is a lot of work that needs to be done in the realm of VR tools. There are several different methods that are being pursued that have different advantages and disadvantages. While it is good to not have a complete monopoly and have one specific tool to be the magic bullet and solve all problems, it is possible to have a few well known and supported tools that people are willing and happy to use. There is also a trend starting towards making base level VR tools available for public usage. This helps spread and foster innovation by reducing the barrier of entry for new labs. An example of this is the giga-pixel display here at VT reusing the Chromium software to power their machines.

With 3D interaction techniques there is much work that needs to be done when it comes to tools for building them. If each researcher has to keep building their own tools and not reusing foundational work from others then significant difficulty in implementation thus leading to slowed innovation will persist. There has been some work on specifying what interaction techniques are composed of [FIGUEROAP01], but there has been precious little work on creating tools meant to be reused by others to help build interaction techniques [WILLANSJ00a].

2.3 Community survey

The previous literature review shows that tools currently rarely focus on reusability, that there is very little reuse of VR tools outside of the labs or businesses that have created them, and that the current approaches to building 3DITs only address part of the problem, but do not address the foundational problem of how to build generic and reusable 3DITs.

The first chapter explains the makeup of 3DITs, and the culture and developmental mentalities that have led up to what currently exists through scenarios generated from personal experience. It shows that very little thought often goes into how to make the 3DITs generic and reusable, most of the work is involved with making them work in a single application, let alone trying to make them work with different applications using a single VR tool. The first chapter also investigates a list of papers from the 3DUI symposium held recently and looks at the state-of-the-art for tools in the area of 3DITs.

The last approach to showing why current tools are not sufficient for building reusable techniques goes beyond literature reviews, stories heard in the trenches, personal observations, and personal experiences. It goes straight to the largest source of information available on the subject of 3DIT development. This is the 3DUI mailing list. This list includes many different

researchers that use all manner of VR tools, are located in different countries, have different focuses than what our research group has, but they all develop, research, or are at least interested in 3D user interfaces.

2.3.1 3DIT community survey

The survey consisted of twelve questions and was a mix of short answer and multiple choice questions. These were geared towards measuring what tools participants used to develop 3D interaction techniques, features they liked, and didn't like. The questions also were geared towards gaining insight into the link between 3DITs tools and the VR tools that they depend on. Below are a list of the questions, and a summary of responses for them. After the questions, a summary of the responses are listed.

Question 1: *What tools have you used to build 3DITs? Did you help develop the tools that you used? If possible, please provide URLs for the tools you mention.*

There were ten responses to this question, and only three of the responses did not create or combine tools to support their efforts. One used a commercially available tool (Virtools), one picked up a tool from another university, and one person used a tool that was developed at the same lab in a university. Four participants did not use a tool at all, they essentially cobbled together graphics and input packages to support their development efforts. Three more participants created the tools that they then used to build 3DITs.

Question 2: *What features of the tools that you used to build the 3DITs with did you like? How easy was it to build the techniques?*

There were nine answers to this question, and they are summarized with these five points.

- Abstractions from low level devices are good
- Scene graphs are useful
- Quick iterations are desired
- Tools do not meet the needs of the users
- People build the tools they use because the existing tools are “nearly useless”

Question 3: *What features of the tools you used to build the 3DITs with did you dislike? What made it difficult to build the techniques?*

From the nine responses come these nine major points (some are duplicated in multiple answers, some aren't).

- Time consuming due to the level of detail
- Hard to reuse parts of projects
- WYSIWYG tools do not produce good code
- Trackers need to be improved, and cross platform support is necessary
- Not enough examples to build off of
- Reuse is usually done through copying and pasting
- VR tool APIs are large and are very difficult to learn, which leads to everyone building their own.
- Certain projects require changes to the VR tools used, which requires source code access
- "Average multi-purpose tools are monolithic, all-encompassing, buggy, badly documented, poorly designed, poorly coded, and difficult to use. All at the same time. Single-purpose toolkits can be expected to work and perform some useful operations on programmers behalf."

Question 4: *Would you consider the tools you have used to be monolithic or modular? Do you prefer monolithic or modular tools?*

Out of ten responses, only one preferred monolithic tools, and only because the learning cost is high and modular tools would only be useful for multi-year projects. All of the other nine preferred modular tools, some cited that they were easier to use, and one even mentioned that VRJuggler's only useful feature was managing X11 windows and 3D projections! This user response is in direct contradiction to the design direction most VR tools have traditionally taken. However, implementing good modular tools is more difficult than with creating monolithic tools.

Question 5: *Do the tools you use to build 3DITs support the process you normally use to design and implement software or do you have to change your process to fit the tool?*

There were eight responses to this question. The answers varied from not being able to remember what people used to do with the tools, to yes there is change and change is very bad, to it doesn't change the process at all, to people building their own tools to support their design process.

Question 6: *In your opinion, on a scale of 1-5, how important is code reuse support in 3D interaction technique development?*

The scale was 1 being not important, 3 being somewhat important, and 5 being very important. There were ten responses to this question. No one submitted one or two. 15% of the total participants said it was somewhat important, while 62% said it was either a 4 (important) or 5 very important. The rest omitted this question.

There were several comments in this area. One of them was that users expressed a desire for a reusable library of common components to build 3DITs, and another was to be able to reuse previous work. Another interesting comment was that code reuse saves time, money, and prevents user re-training.

Question 7: *How mature do you think current VR tools are?*

The scale used was 1 being not important, 3 being somewhat important, and 5 being very important. There were ten responses to this question. No one voted 4 or 5 (mature or very mature). Six people (46%) said they were somewhat mature. Four people voted for two, and one voted for not mature. Comments for this question were there is considerable room for improvement, and that we are at the tip of the iceberg for what VR tools can do. Another comment said that VR toolkits require expert developers to use, and that most tools get in the way of the user. A lower level comment given was that better tracker support is needed.

Question 8: *In your opinion, how important is the ability to use different graphical toolkits with a VR toolkit?*

The scale was 1-5 again, with 1 being not important, 3 being somewhat important, and 5 being very important. There were eleven responses to this question. Eight of those eleven ranked it somewhat important to very important. Seven of those eight ranked it four out of five, and five people (38%) said it was very important. Two people ranked it a two, and only one ranked it not important. Comments for this question included a participant saying that writing your own graphical toolkit is difficult, and another saying that "building on other technologies is the only way to advance this field." Other comments were that using different graphical tools would produce significant time savings, and would foster reuse. One participant said they do not use external scene graphs because of the "long-dead scene-graph idea which is not the way to build non-trivial applications."

Question 9: *How important is public availability of tools that you might use to build 3DITs?*

This was on a scale of 1-5 with 1 being not important, 3 being somewhat important, and 5 being very important. There were eleven responses to this question. No one said it wasn't important,

and only one person ranked it a two. Four people 31% said it was very important, three people (23%) ranked it a 4, and 3 ranked it somewhat important. User comments for this question included the need to keep software updated, a need for faster and easier ways of creating tools, the benefits of ARToolkit to the field of AR research, universities must depend on tools that do not disappear because they are no longer being supported, and one recipient wants to be able to download and use our software, free of charge.

Question 10: *Do you think that VR tools and 3DIT tools should be part of the same toolkit, or should they be separate? Why?*

Out of the nine responses to this question, only two thought that they should be part of the same toolkit. Most thought that they needed to at least know about each other, and several thought that they must be separate. Lock-in to a specific tool is a very big problem to one of the responders. Also, the point that VR doesn't need 3DITs, but that 3DITs needs VR was mentioned.

Question 11: *What features of a tool would you consider useful for building 3DITs?*

Only six responses were given to this question. The main points from this question are access to objects in the environment, WYSIWYG, integration with the SG, easy access to the SG, multiprocess communication support, GPU based techniques, simplistic tools that follow a UNIX mentality, and predefined support categories.

Question 12: *Other comments or thoughts about building 3DITs and the tools used for this purpose?*

Only two people responded to this question, their points were to please include samples for others to use, and to think about novice users and allowing them to create a 3D user interface.

2.3.2 Summary of 3DIT survey responses

What does this survey show? It shows that 3DUI researchers desire tools that help them accomplish their work, and that they currently do not exist. Current VR tools are unwieldy and are not used in some projects. There is no support for reuse other than copying and pasting code that developers have written themselves, and that there are very few examples from which to base their work. It also shows that there is not even one area where the analysis from the other two approaches contradicts the results gathered from this survey. This verifies that all three approaches are in harmony, and that there is a grave problem with 3DIT development.

2.4 Summary of work done to answer the first research question

The first research question is why current tools are not sufficient to build reusable techniques. This has been answered through three different methods. The first method was the first chapter that motivated the need for reusable techniques. By investigating state-of-the-art VR tools and techniques we demonstrated that there is no evidence of technique reusability in current tools and techniques. One might say that our sampling of tools and techniques was not large enough, so a literature review was done in order to show the state-of-the-art for developing reusable techniques. This literature review covered VR toolkits, 3DIT specific tools, and specification level approaches used for building techniques. This showed that during the first stages of techniques development little thought was given to reusability. Due to the difficulty of building techniques, higher level 3DIT tools and specification tools were created. These newer tools were designed to make developing techniques easier by specifying different ways for components to work together. However, the newer tools focus was not on reuse. Because of this, these approaches do not address all of the four different types of reusability that this work addresses. These newer tools only demonstrated two of the four types of reuse, and theoretically provide a third type. Also, there are no examples of techniques being built with these tools in the literature. In order to verify that the outside research community agreed with our conclusions a survey of the 3DUI community was performed. This survey verified the conclusions from our personal work and literature review. Through these three activities we have shown that current tools are not sufficient to build reusable techniques because reuse has never been a focus of the VR tool community, there is little evidence of reusability with tools and techniques, and there is a desire for being able to build reusable techniques.

Chapter 3: Foundational Work

3.1 Introduction

The first and second chapters of this work have illustrated the need for reusability in the domain of 3DITs. The level of reusability targeted in this work is both within applications written with the same VR Toolkit and with applications written with a different VR toolkit. The type of reuse is meant to be component level, not copying and pasting source code. The first step toward creating this level of reuse is to decide what role VR toolkits will fill in this research. As stated in Chapter II there has been considerable work done with VR Toolkits and re-inventing the wheel would be rather arduous unless there is a specific need that existing tools do not fill.

An ideal VR toolkit would need to be designed to support reusability, work with the most popular graphical toolkits, and work with the current hardware trends (this work was performed during a period of transition from "big iron" to clusters of computers). When this work started in 2002-2003, there were no VR toolkits that met all of these requirements, so foundational work had to take place in order to allow for reuse research to take place. The purpose for this foundational work was to provide the ideal toolkits designed for reuse that could be used to complement the effort to make reusable 3DITs. The goal for this work was to be analogous to bricks stacked one on top of each other. The first brick is a VR toolkit that is suited for long term usage, and the second brick would be a tool that is designed to work with VR toolkits that is designed for creating reusable techniques.

The two major open-source tools available at the start of this work were DIVERSE (DPF / DTK) and VRJuggler. While VRJuggler had the technical features necessary, specific design considerations have historically kept the DIVERSE project from adopting the VR Juggler approach [KELSOJ02]. Because of this and the need to not alienate previous work done here at Virginia Tech specific additions to the DIVERSE toolkit had to be made in order to allow for modern reusability. When this work first started DIVERSE consisted of two different components: a utility (DTK) and a graphics layer (DPF). The utility layer contained the foundations for component level reusability, navigations, and a run time loop architecture. However, the graphics layer was completely integrated with a proprietary graphical toolkit (OpenGL Performer) that was becoming obsolete and is now no longer available.

The two new components built were the DIVERSE interface to open GL (DGL) and the DIVERSE Adaptable Display System (DADS). The goal for these tools is the same as other VR toolkits, to provide the ability to write a program once and use it on any VR hardware without modification. In addition to this goal, allowing for component level reuse is a major goal. For DGL, its focus is to provide a simple interface for displaying content from multiple graphical toolkits. For DADS it is allowing VR applications to run on a cluster system without requiring the programmer to focus on the normal technical challenges of cluster development.

3.2 DGL

DGL's main goal is to remain as invisible as possible to the outside user. The idealistic goal is to allow for content developers to work in their desired toolkit (VTK/Coin/Open Scene Graph), and to not have to focus on learning yet another VR tool for doing their work. In order to accomplish we identified several high level requirements for DGL. These are:

- Handle the VR specific aspects of the OpenGL environment
- Multi-threaded rendering where possible
- Allow context sharing between multiple displays
- Allow plugins to control every aspect of the architecture
- Provide interfaces that work with existing graphical programs
- Provide support for toolkits that render OpenGL (VTK/OSG/Coin)

3.2.1 The display architecture

In order to provide OpenGL support for any VR environment, and to handle the VR specific aspects of these environments, several considerations have to be taken into account. Details such as the number of windows, the viewports, the inter-ocular distance, configuring the frustum, head tracking, and stereo display code are just a few of the areas that require attention in a modern VR toolkit. However, there is a more fundamental aspect that has been ignored until now: creating standard 2D windows that display OpenGL. Each operating system has a specific way to create these windows. Some VR tools have written a non-trivial amount of code in order to add in cross-platform windowing components. In the spirit of reuse, it was decided to use the Open Producer project (which was originally part of Open Scene Graph) in DGL to provide a method for creating cross-platform windows. It was designed to be a non-center of the world tool so were able to use it transparently in the DGL architecture.

Beyond just creating windows, there are many other concerns to address when creating an environment suitable for displaying OpenGL in a multi-walled system. One is multi-

threading and another is how to handle render graphics in stereo. In order to handle these issues (and several others) a specialized display architecture was created. Figure 3.1 shows the architecture that was created to address these requirements.

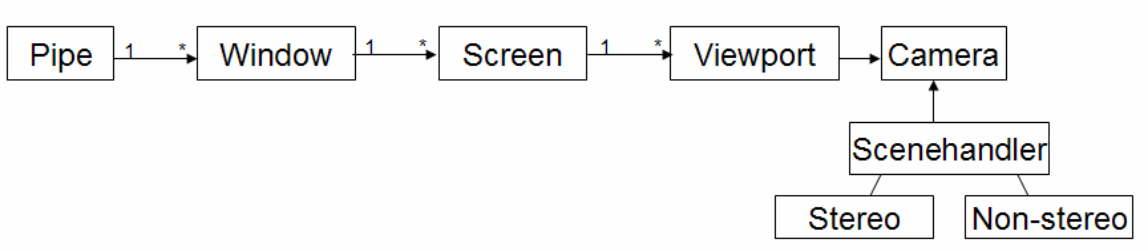


Figure 3. 1 The DGL Display Architecture

The starting point for the architecture is a pipe. This is inspired by the graphical pipes on the SGI "big iron" machines. In more modern terms this is a specific screen that is powered by a specific graphics card. Inside of each window is an Open Producer RenderSurface. This is the actual 2D window and associated properties that appears on our standard desktop. A pipe may have multiple windows. For example, passive stereo requires two windows within a single "pipe." Inside of each window is one or more screens. The screen is the component of the architecture that holds more detailed information about the specific display device that renders the window component. This includes an offset for the display (for the walls of the CAVE / immersive workbenches), the field of view, the interocular distance, and so on. One reason for allowing multiple screens are displays that have a large single window; they can break this window up into two different components (for passive stereo / CAVE systems). Inside of each screen there can be multiple viewports. These are standard OpenGL viewports, and allow desktop displays to easily mimic multi-walled immersive systems. Another use for viewports is to black out overlap on immersive screens. Each viewport has a Producer camera and scene handler associated with it. These will be explained later. Lastly, specialized stereo and non-stereo component designed to work with the scene handler have been provided.

This display architecture has allowed for several different types of displays to be written. Simple desktop windows are the norm with VR toolkits, but this architecture allows a six sided CAVE simulator using six windows or one window with six viewports to run on a desktop for debugging. Other uses of this architecture are the normal active / stereo walls used for CAVEs / PowerWalls.

Two important utility functions that the architecture provides are multi-threaded drawing and context sharing. The pipeline for drawing in DGL consists of the standard clear / draw / swap buffers cycle. In addition to this are two init callbacks that are called the first time the draw stage is reached. These are the initGL callback (called once per context) and the setupGL callback (called once per viewport). The threading works by having one update thread and one thread per context. Multi-threaded drawing can be turned on or off for an application by simply loading the appropriate plugin.

3.2.2 The Open Producer architecture

The previous section mentioned the usage of several Open Producer components, but did not explain their function. This section explains how Open Producer works and how DGL uses it to display OpenGL. Open Producer has a much simpler architecture, but does not provide direct support for VR displays, as it is. The purpose for DGL was to provide VR support using the strengths of Open Producer. The Open Producer architecture is listed in Figure 3.2.

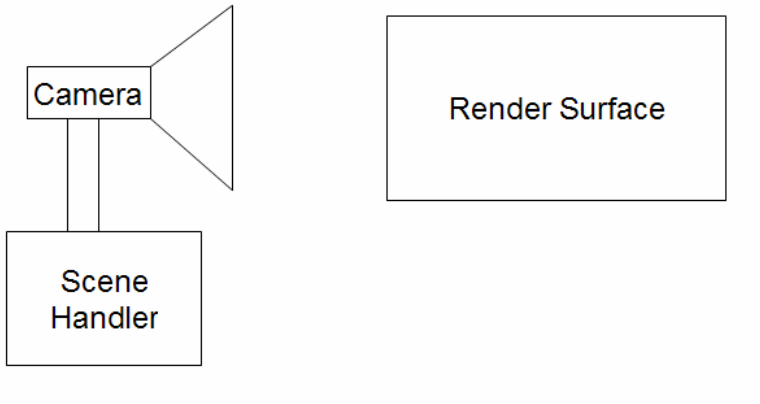


Figure 3. 2The Open Producer Software Architecture

The Open Producer architecture can be explained through a description of a movie producer. A producer sets up the scene to be recorded. This scene is represented by the render surface in Figure 3.2 and the component that handles setting up the scene is the scene handler. Next the producer may want to capture a specific part of the scene, and directs the camera crew to focus on this particular part. This is the camera in Figure 3.2. In terms of lower level code, a render surface is a 2D window that provides a surface to render graphics on. The camera is what causes the 2D window to render its contents to the world, and the scene handler tells the camera

what to do. DGL's display architecture manages multiple cameras, scene handlers, and render surfaces in order to create VR displays.

3.2.3 Programmer interface

The previous section describes the display architecture. While this is the centerpiece of DGL, the average developer generally does not directly interact with it. The purpose of the display architecture is to provide behind the scenes functionality for what the developer actually interacts with: the programming interface.

DGL focuses on integrating with the work flow that developers follow, not requiring them to completely change their existing code to work with DGL. For example, all that is necessary to convert most scene-graph applications is to take the root node for that application and make it a child of the world node that DGL provides. Another example is that the traditional GLUT callbacks in regular OpenGL programs can be replaced with corresponding DGL callbacks. Before the actual interface for integrating content into DGL can be displayed, the DIVERSE application loop has to be explained.

The application loop consists of the following steps: preConfig, postConfig, preFrame, draw, and postFrame. The preConfig step is used to configure the system before displays are created. The postConfig step is used to configure the system after the displays have been created. The preFrame happens before drawing happens, next the drawing happens, and lastly the postFrame is called. After post frame is called then loop jumps back to the preFrame step. This is illustrated below in Figure 3.3.

preConfig
- Creation of displays

postConfig
- Final chance to configure before drawing

preFrame
- Last chance to do data updating before the draw, usually used for tracker data before drawing.

draw
- init/setupGL are called before the first draw to allow for OpenGL setup
- Render OpenGL

postFrame
- Update after drawing

Figure 3. 3The DIVERSE Application Loop

Now that the flow for a DIVERSE application has been explained, the method that developers use to add their content to DIVERSE can be explained. There are two methods for displaying content in DGL: the augment and the C interface. The C interface to DGL consists of C functions that correspond to the above functions. They are similar to the GLUT programming standard. The only required change between GLUT and DGL programs are the input from the user. All of the input in DGL is managed by DTK shared memory segments. These have to be used instead of the keyboard/mouse specific callbacks used in GLUT. Shared memory is also used to provide the tracker information that GLUT does not provide but is necessary in several VR applications.

The second way content is added to DGL is through an augment. An augment is a C++ class that has callbacks for each of the steps listed above. These callbacks allow for user specific code to be executed at the desired step in the program execution. DGL augments are derived from DTK augments. They are designed to augment the scene in different ways and are plugins that can be loaded and unloaded at runtime. For example, a DTK augment may be a navigation plugin, but a DGL augment could be a piece of code that draws a helix. DGL comes with several augments. Three of these augments correspond to major graphical toolkits. Several open source packages can be made to render OpenGL on command through some creative "implementation gymnastics." These "gymnastics" done in augments are what provide DGL the ability to display content from Open Scene Graph, VTK, and Coin3D through the DGL display framework through augments. For example, with Open Scene Graph we implemented an

augment that understands the multitude of different components required to cause Open Scene Graph to render, and wrap up these areas in an augment. This augment is then given additional functions to provide Open Scene Graph specific functionality. One example of this functionality is a simple scene-graph that contains navigated and non-navigated scene-graph nodes that users can add their content to. This allows a developer to call a static function [DOSG::getWorld()->addChild(pointer)] and add their content without having to understand or worry about the major implementation details.

3.2.4 Integrating DGL into existing programs

Creating a basic DGL program, or adding DGL support to existing applications requires little work. There is only one header file needed, and four required lines of code. This is illustrated in Figure 3.4. Building your application so that it compiles in the DGL libraries is supported by the dgl-config program. This program provides include and library information in a cross platform manner.

```
#include <dgl.h>
int main()
{
     dglInit();
     dglStart();
     while (dglIsRunning())
          dglFrame();
     return 0;
}
```

Figure 3. 4 An example DGL program that loads plugins and runs in any VR environment without requiring code changes in the user's application.

DGL is a singleton and is created by the initialization function call. The start function call does the preConfig/postConfig and calls the frame function call once to draw the first frame. This starts DGL, but does not keep it running. As long as DGL isn't told to quit (by a keyboard command or application specific feature) the running function call returns true. The main frame function call repeats the preFrame, DTK frame, and postFrame parts of the application loop.

3.2.5 The big picture

All of the components of DGL have been described in the previous section, but how all of the components fit together has not been described. This has been left until the end to reduce confusion. This is because DGL calls DTK which calls a specific part of DGL which then calls Open Producer which then calls another part of DGL, which produces a dependency nightmare

that is difficult to understand at first glance. The reason for such complexity is due to the fact that DGL is literally derived from DTK. This requires additional overhead for the creator, but not for the user.

The first step that happens in a DGL program is the initialization step. When the init call happens all of the variables inside of DGL are created and initialized. The high level overview of this is listed in Figure 3.5. The major product of this step is that the DIVERSE specific environmental variables are examined and DTK is called to load the specified plugins into the system. The next step in the application loop is the start call. When the start call listed in Figure 3.4 is called, DGL goes through and causes several steps to happen in both DTK, DGL, and Open Producer. These are listed in Figure 3.5. The major step in this phase is the creation of the displays. DTK was designed to be used as a base for graphical VR toolkits, and includes support for the concept of a display. The display has specific callbacks (creation, updating, etc…) and has a specific place in DTK's architecture. DGL derives its display system off of a DTK Display, and is powered by the DTK architecture. DGL causes the DTK architecture to be executed, which then causes the DGL Display to be created, configured, and updated. The DGL display is what calls the display architecture to use Open Producer to create the displays for DGL. When the frame call is made eventually the DGL display is called to draw. This then causes the display architecture to use Open Producer to render OpenGL. After all of the setup has been done with the scene handler then the DGL draw function is called which causes all of the augments to produce OpenGL. This is slightly complex, but in order to make three architectures with only marginal similarities work together to fulfill the requirements listed above, this design was necessary.

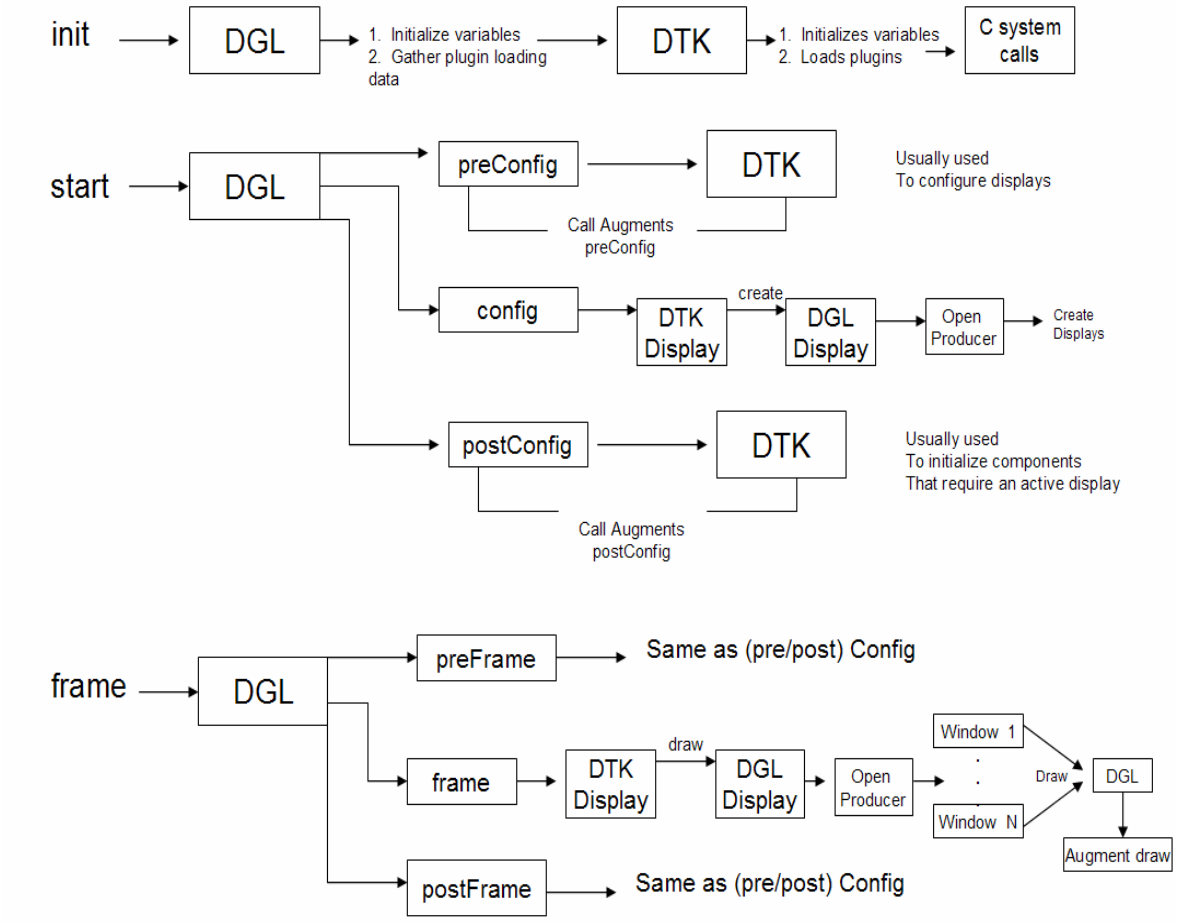


Figure 3. 5 The dependencies for the high level steps during the DGL application loop

3.2.6 Multi-threaded rendering

The discussion of multi-threaded rendering has been left out previously, because it adds yet another layer of complexity to describing DGL. However, multi-threaded rendering is required for reasonable performance in highly complex systems. When multi-threaded rendering is enabled the same basic functionality described earlier continues to happen, except in a slightly modified manner. The best analogy for threading in DGL is to imagine the components of the draw callback to be a circle. Now create one circle for each display, and one for the main program. Next, think about spinning the circles endlessly. Lastly, you have to make sure that the main circle and the display circles spin in a controlled manner so the application loop always happens in the order that was described earlier. These operations are visually described in Figure 3.6.

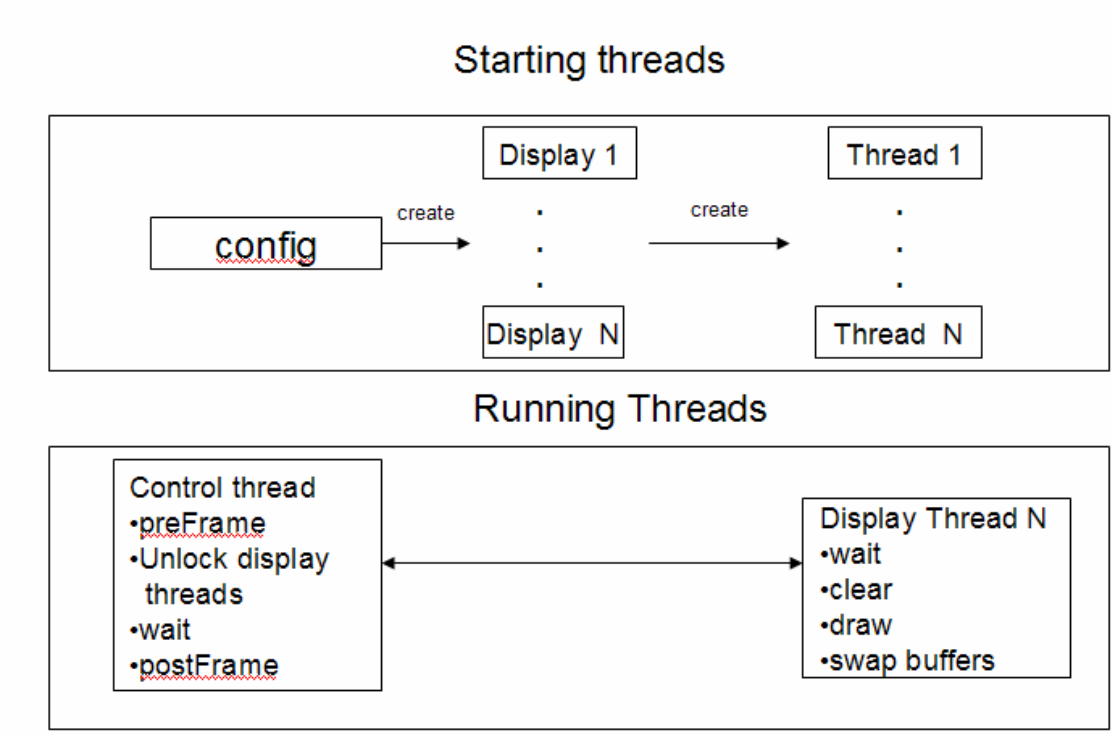


Figure 3. 6 Thread creation and runtime execution

Threads are started in-between the preConfig and postConfig steps in the runtime loop. The general idea behind the threading in DGL is that the application loop happens as it does in single threading, but during the draw step it splits off into N threads to do the drawing on different CPUs. The main thread does the preFrame before the draw threads start to draw. Once the preFrame is finished the threads clear and then draw. Once all the threads have done this, they all swap their buffers. This prevents screen creep. After this the main thread executes the postFrame callback, and the cycle repeats itself.

3.3 DADS

Several VR centers that had multi-walled setups originally used powerful single computer systems to drive the displays. The era for single computer systems powering a display was coming to a close at the time this research started. The major large scale vendor for these systems was restructuring their business, and there was only one way to go with hardware at that time: clusters. Only some VR toolkits supported using clusters, and several toolkits designed for clustering were not available at the time of this research.

In order to perform reusability research on multi-walled systems and to provide a foundation for other research needs we decided to update DIVERSE so it could power these systems. We performed a smaller literature survey in this area to discover the different approaches taken in creating clustering systems.

3.3.1 Previous cluster research

From the time before clusters became widely available VR Juggler was one of the few publicly available VR toolkits that was updated to have clustering capabilities. In addition to this tool, two new tools have emerged to fill different niches in the VR software world. These are Chromium and Szygy. Another closed source tool updated to work on clusters is Avango. Avango is similar to Syzygy, but has a reduced scope.

Szygy and Avango work by providing a limited scene graph that knows how to synchronize itself over a cluster. Chromium was developed to distribute graphics over the network in a cluster. This allows for one machine to render all images for several different monitors, or for several different machines to render different parts of the VE. Instead of typical video cards rendering all of the video, as is typically done, this allows for compositing of displays from multiple machines. It is mainly used for flat tiled display systems, though different types of projections such as CAVE type systems are possible. Chromium works by intercepting OpenGL library calls and causes rendering to be distributed to clustered machines. Current OpenGL based programs can work seamlessly on a cluster of machines this way. This method greatly taxes the network and is not suitable for demanding applications because Chromium is slower than other VR toolkits. Chromium has mainly been used with simple keyboard and mouse input, not with standard VR trackers [HUMPHREYS02].

Syzygy is a system designed from the start to support clustered systems. It is billed as a grid operating system with a focus on multimedia instead of on batch job control. The project goals were set so that Syzygy would be an integrated system for distributed graphics, sound, systems management, data I/O, and general I/O device management. It has the most flexibility and control options of any of the other VR cluster software systems available. It also allows for and supports multiple ways to tweak and optimize an application. The downside to this approach is that it is not designed to be transparently scalable from a single machine development platform up to a cluster based system. Also, the flexibility to use different graphical tools that different researchers want to use is not available with Syzygy. It provides its own distributed scene-graph,

pure OpenGL support, and nothing else for the end user. As long as a Syzygy user is content to use provided tools Syzygy is an option, otherwise development may be problematic [SCHAFFER03].

Avango is a closed software tool that is similar to Syzygy's in that it focuses on having a distributed scene graph, but that is its sole focus. It does not attempt to have any of the other cluster support systems that Syzygy provides. It works based on subclassing all of the graphic nodes in the scene-graph. This provides for well controlled synchronization, but causes significant problems when API changes happen, or new features are added [TRAMBEREND99].

VR Juggler is a VR toolkit that existed before clustered visualization systems became widely used. It has two different clustering solutions available: Net Juggler and Cluster Juggler. Net Juggler is a MPI based system developed on top of VR Juggler by a team of researchers not affiliated with VR Juggler, whereas Cluster Juggler was designed by the VR Juggler team to replicate the application across a cluster of computers. There are advantages and disadvantages to both methods. The Net Juggler approach takes advantage of the standard cluster computational language MPI, and uses it to take a standalone VR Juggler program and have it run on a cluster of machines. It works by extending VR Juggler components to work in a clustered manner. The advantage of this method is that uses a well supported and maintained cluster product to provide visualization. The disadvantage is that if any part of VR Juggler changes, Net Juggler will no longer function. Lastly, this requires complete use of the VR Juggler input, display, and configuration options. Using anything outside of the system will not work on a cluster based system [ALLARD01] [OLSEN].

Cluster Juggler performs the same operations as Net Juggler but does not require or interoperate with MPI. It replicates the VR Juggler program across multiple different computers and forces each machine to synchronize itself with the cluster after each draw frame. It distributes input to each of the nodes and guarantees that every machine in the cluster has the correct information during the applications lifecycle. It requires no modification to the VRJuggler programs.

3.3.2 Overview of DADS

We were not able to adopt any of the previous work to complete this portion of the foundational work due to technical constraints in the previous solutions. The largest issue was that the systems did not interoperate with the graphical tools necessary (DGL). However, ideas from the

previous solutions were incorporated in our solution. The solution we developed is called the Diverse Adaptable Display System (DADS). DADS takes a similar approach to Net Juggler [ALLARD01] and Cluster Juggler [OLSEN] by providing a default synchronization event at the end of every draw loop. Also, similar to the Juggler system is the ability to synchronize input for different devices. However, the use of plugins to implement the system (instead of having re-architect the core of the toolkit), and the usage of shared-memory differentiate DADS from VR Juggler. This section will go over the architecture of DADS, the communication components of DADS, and the requirements for making it work with existing applications.

When a user wants to use the DADS system they sit down on a traditional desktop computer and run their application the same way they did while they were developing it. However, by simply loading one plugin they can switch between standard desktop operation and full cluster operation. This single plugin works by loading several other plugins that create the setup described in Figure 3.7.

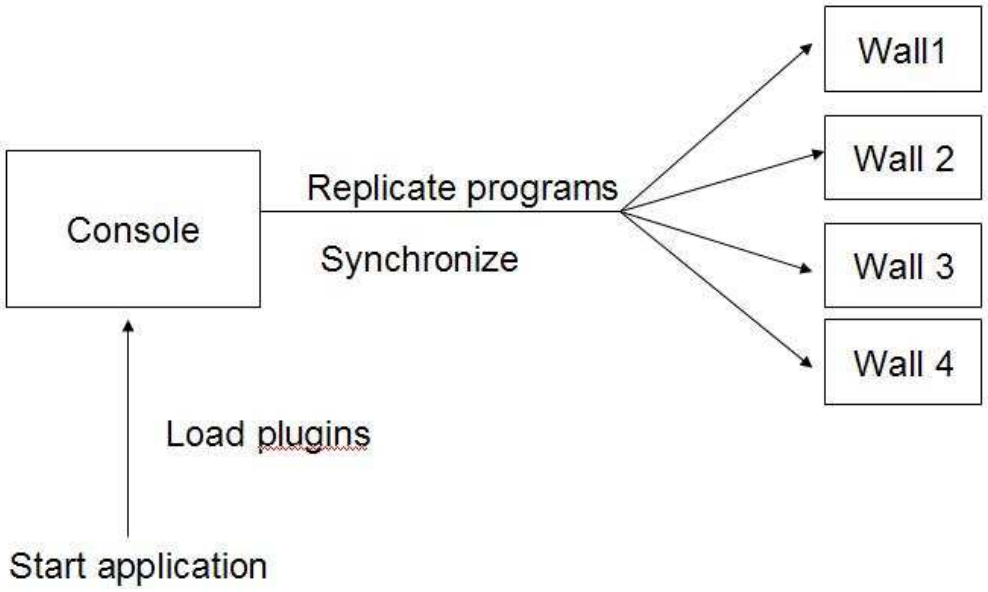


Figure 3. 7 Steps for starting application on the VT-CAVE with DADS

3.3.3 Dependencies for DADS

The DADS system has several environmental dependencies that require extra work to install before it can work. However, these requirements are often standard for cluster based

environments. These dependencies are to have a shared home directory between all the machines, and to use LDAP to handle the user logins across all of the cluster machines. Next all the software to be installed in the same location, and all the cluster machines to be running the same distribution and updates. Also, all of the machines must have unique ips and hostnames. Lastly, this only works on Linux, as is the case for several other VR configurations. This particular software configuration allows for DADs to make several assumptions. The first is that the same program can be compiled once and used on any of the cluster machines. Due to the plugin based nature of DIVERSE this allows each specific computer that powers a specific wall in the VR display to have plugins loaded for that particular wall. Because of the Linux requirement DADS can use the proc file system to determine how to replicate the program (directory the program ran in, environmental variables set for the shell, program name, and the arguments given to the program.

3.3.4 Plugins for DADS

As mentioned earlier, DADs works based on a series of plugins. There are three general plugin types used in DADS. The first are plugins used to start the programs, then plugins used to manage communication while the cluster is running, and lastly plugins used to shut down the cluster based system. In general there is a set of server plugins and a set of client plugins. On the console the user loads a group plugin which loads the starter, navWriter, and death plugins. The start plugin sends asynchronous messages to all of the client machines which then causes them to replicate the program started on the server. It also directs the client machines to load the navReader, clientDeath, and machine specific display / configuration plugins. The navWriter plugin reads navigation, device, and user specified information and sends this to the client machines during the postFrame step in the application loop. It also synchronizes the machines by causing the machines to stop at the postFrame step. The death plugin detects when the server application stops and causes the application to close on all of the client machines. The navReader plugin is the receiver of the data that the navWriter plugin sends. It sets the navigation and device information on the client machine it is loaded on (and also user specified information). The clientDeath DSO also pairs with the server death DSO to stop the machine. There is also a NVIDIA specific plugin that can be used to enable hardware specific synchronization. Because DADS is implemented via plugins, it is possible for other outside developers to implement and tweak synchronization methods for any DIVERSE application.

3.3.5 Communication across cluster machines

DADS provides for both synchronous and asynchronous communication across the cluster. The synchronous communication has been mentioned before, and will be explored in more detail here. The asynchronous communication happens over shared memory between the machines. A message abstraction on top of raw shared memory was created, and allows for communication across the cluster machines in a structured manner. Also, each client machine runs a daemon that listens for messages across DTK shared memory (this is how the programs are started). This allows for client machines to communicate information back to the server, and for non critical information state information to be sent across the cluster. The asynchronous communication is fast enough for handling GUI events, but not fast enough for sending navigation and device information across the cluster at over 30 FPS.

The synchronous communication mechanism for DADS consists of a customized client/server custom socket program. All this does is a standard send/acknowledge cycle for a block of data. This data is the users location and orientation, the device information (wand, buttons, joystick, and head tracker), and any shared memory segment that the user desires to be synchronously instead of asynchronously. These options can be used simultaneously without being processor or network intensive. This is not available with some VR clustering solutions.

3.3.6 Reflections on implementation

Several implementation notes about DADS should be mentioned. Due to the original design goals and modularity of the DIVERSE project, zero modifications had to be made to the original project parts (DTK/DPF) so that they could function in a cluster environment. This is unheard of when it comes to software projects that were not designed with clustering in mind. It was also flexible enough that another graphical front end (DGL) could be built on top of DTK the same way DPF was. Once this was done, DADS automatically worked with DGL. This proves that the reusability goals of the first DIVERSE paper were feasible and valuable in practice.

Another factor to consider is that 95% of the existing DIVERSE applications written could be ported to a cluster environment with zero modification due to the flexibility of DTK shared memory. This is another important testament to the value of reusability.

Lastly, it is possible to integrate MPI into a normal DIVERSE application and bypass using DADS. This is slower than using DADS, and requires rework for every application that uses it, but if a clustered display system powered by MPI is critical, DIVERSE can support it.

3.3.7 Conclusions on DADS

DADS is a modern cluster based solution that is flexible, easy to use, and allows developers to build cluster applications quickly and easily. It allows for DIVERSE applications to become cluster compatible without significant redevelopment costs. The flexibility and simplicity of DTK, DPF, and DGL are what have allowed DADS to be an effective solution. While a formal study comparing the application implementation using different VR tools has not been done, this system provides a good platform allowing for applications to be developed and reused quickly.

3.4 Summary

In summary this chapter has presented the foundational work necessary for creating a platform for reusability, the reasons behind this work, comparisons to other work, and an in-depth description of the foundational work developed for this research. By creating this foundational work we were able to create the reusable techniques described in chapter five with the tool created in chapter four. The foundational work has allowed for continued growth of our traditional VR research at VT, allowed for 3DIT research to happen on a cluster based system, and provided the possibility for our reusable techniques to work in a cluster based environment. DGL and DADS have been used in dozens of projects in the past few years and are the default software platform in use at the VT-CAVE.

Although the work in this chapter re-invented the wheel for VR toolkits, this was a necessary step. This was required to create an ideal VR toolkit designed to support reusability. The VR toolkit components we created work with the most popular graphical toolkits, and works with the current hardware trends. These toolkit components complement the effort to make reusable 3DITs. Using the brick analogy in the beginning of this chapter, the first brick towards creating reusable techniques has been laid. By removing the need for VR toolkit functionality from the development process for reusable 3DITs it will be possible to design a system to create reusable techniques at a higher level.

Chapter 4: The Interaction Framework For Innovation: IFFI

4.1 Introduction

This chapter addresses the second research question. It provides the requirements and design for a tool that allows for reusable techniques to be created. Reusability is a core requirement for any successful software effort. If developers had to write their own programming language to write their applications, software would not be as advanced as it is now. By addressing a foundational part of 3DIT development, the hope is that this will be a strong foundation that others can build considerably more advanced techniques than before. The structure of this chapter is as follows: the first part of this chapter will deal with the analysis of 3DITs with specific attention paid to reusability. Then we will revisit the requirements and design processes that previous tools used in order to understand how to build a tool designed to make 3DITs reusable. After this will be a final discussion on requirements for a reusable tool that takes into account analysis of both 3DITs and VR Toolkits. Next, a design of the reusable tool will be presented; then, a description of the subsystems of IFFI. Next to last will be a set of usage scenarios focused on how developers can reuse the system. These will be compared to the existing usage scenarios listed in chapter one. Finally, a comparison will be made against previous tools with a focus on reusability will be performed. Chapter five will focus specifically on the 3DITs built with this tool.

4.2 Analysis of 3DITs

One of the factors contributing to the lack of reuse is that 3DITs can be described simply, but cannot be implemented simply in many cases. This is due in part to a high degree of coupling between 3DITs, VR toolkits, and graphics toolkits. The reason for this coupling comes from the nature of design decisions the tools have made. In order to illustrate both the decisions that tools have made, and how a 3DIT is built using current tools, the Go-Go 3DIT [POUPREYVI96], and Voodoo Dolls [PIERCEJ99] will be deconstructed into a verbal description that the user can understand and then expanded into a list of requirements, and the dependencies of these requirements, in respect to VR tools, scene-graphs, and portability.

4.2.1 The Go-Go technique

Describing Go-Go verbally is simple: at a distance less than some threshold, the virtual hand follows the user's physical hand; beyond this threshold, the virtual hand's distance extends

farther than the physical hand based on a non-linear function. When the virtual hand touches an object, it can be picked up with a trigger event (e.g., button press), and an object in-hand can be dropped with the same trigger event.

This description allows for simple communication of the expected behavior in a VE. However, communicating the implementation of this 3DIT is considerably more complex. Next, we will look into requirements, design details such as device input, program flow, graphical objects and VR toolkit support. The design and implementation details greatly increase the complexity of a canonical technique, let alone a more complicated 3DIT. These are common design decisions that a 3DIT developer has to make. This is not a contrived scenario designed to make this research look better. To illustrate the design and implementation details, the next section shows one way that Go-Go can be implemented.

4.2.1.1 Requirements for implementing Go-Go

At a high level Go-Go has two major components. The first positions the virtual hand based on a non-linear function of the physical hand's position. The second takes a trigger event, and does one of the following: a) does nothing if the virtual hand is not touching an object, b) attaches the currently touched virtual object to the virtual hand, or c) detaches the currently held virtual object from the virtual hand and attaches it back to its previous parent.

More complexity arises when we drill down farther into the implementation of the 3DIT. From this high level description of components a list of requirements can be produced. These requirements are listed in Table 4.1. This table also specifies if the requirements are dependent on a VR toolkit, dependent on the scene-graph in a graphics toolkit, or are portable (not dependent on a VR toolkit or scene-graph).

Requirement	VR tool	Scene-graph	Portable
1. Knowledge of user body position	X		
2. Knowledge of user hand position relative to body	X		
3. Pointer to virtual hand		X	
4. Non-linear function			X
5. Update whenever user's body or hand moves	X		
6. Threshold distance			X
7. Feedback when hand touches object	X	X	
8. Trigger event	X		
9. Storage of state (none, touching object, object in hand)			X
10. List of selectable objects			X
11. Collision detection between objects		X	
12. Adjustment of object offset from hand so object does not "jump" when picked up		X	
13. Storage of selected object's parent		X	
14. Adjustment of object offset from hand so object does not "jump" when dropped		X	

Table 4. 1 Portability of requirements for implementing Go-Go

4.2.1.2 Analysis of 3DIT Requirements for Go-go

As Table 4.1 shows, despite the simplicity of the verbal description of Go-Go, there are at least fifteen low-level requirements that have to be implemented. Five of these requirements are VR toolkit-specific, six are scene-graph-specific, and only four depend on neither a VR toolkit nor a

scene-graph. Without significant thought and effort, the current tools often lead the developer to tightly couple the scene-graph and VR toolkit in their 3DIT. This is because of the nature of VR toolkits themselves. When the only functionality that a developer has is a generic callback, everything tends to be lumped into this callback (or supporting functions that the callback calls). This makes reusability dependent on both the scene-graph and the VR toolkit. Given the state of VR toolkits, it is not surprising that the majority of 3DITs are not reusable. If we look carefully at the non-reusable components, however, we note that they can be placed into three main categories. The first is object position and orientation (requirements 1, 2, 7, 12, 13, 15); the second is parent/child relationships (requirements 2, 3, 13, 14, 15); and the third is input (requirements 1, 2, 5, 8).

Thus, for a simple implementation of Go-Go, less than 30% of the software components are trivial to reuse (assuming that the technique is implemented from modular components). Next, analysis of a more complex 3DIT will be used to verify that the requirement distribution is similar.

4.2.2 Voodoo dolls

The voodoo dolls technique is a merger of image plane selection and WIM. A user starts out by selecting the objects they wish to manipulate using an image plane technique. Once these objects are selected, then a copy of this object is made and placed it in the user's hand. This is the process of creating dolls. The voodoo process is when the user moves the dolls in their hand and it influences the object that it represents in the world. The non-dominant hand acts as a frame of reference for the dominant hand. This allows for the user to move their hands into a comfortable position to manipulate the object. Users can be told to use the technique with the following instructions: stretch out your thumb and index finger. Place these fingers over top of the object you wish to select. Pinch your thumb and index finger together to create a doll of the object. Do not release your pinch until you are ready to release the object. The object in your right hand moves relative to the object in your left hand. You can pass objects between your hands by pinching when your fingers are over the doll.

4.2.2.1 Requirements for Voodoo dolls

This 3DIT is composed of three major states. The first is a normal VE state that just displays the world. The second is the creation and holding of the dolls. The third state is the release of the

dolls. The different states are toggled by a pinch event generated by pinch gloves. Depending on the state, a different pinch event may be used to trigger the next state. The first state is simple, it is just a place holder that waits for a pinch event to happen.

The second state is activated by a pinch event. When this event is generated the position of the hand must be determined and an image plane selection technique is used to determine if an object has been selected. If the selected object is already a doll then it is passed to the other hand, otherwise a copy of it is made. This copy is resized to properly fit on the bottom of the screen at the location of the users hands. Finally, whenever the right hand tracker produces change events the selected object must be moved via the mapping of the object in the right hand moving relative to the object in the left hand.

The last state is when an un-pinch event comes into the system. This removes the doll from the screen and stops the manipulation of the object.

Requirement	VR Tool	Scene-graph	Portable
1. Storage of state (none, doll creation / manipulation, doll destruction).			X
2. Ability to process pinch events.	X		
3. Ability to determine non-dominant versus dominant hand.	X		
4. Ability to pass objects between hands through pinch events.	X	X	
5. Ability to make copies of objects in the graphical world.		X	
6. Ability to manipulate scale, position, and rotation of objects in the world.		X	
7. Translation between local coordinates and world coordinates.		X	
8. Knowledge of user body position	X		
9. Knowledge of user hand position relative to body	X		
10. Ability to update objects in the world when body or hand moves	X	X	
11. Collision detection via an image plane selection technique		X	
12. List of selectable objects			X
13. Storage of doll / actual object pairs			X
14. Adjustment of object offset from hand so object does not “jump” when picked up		X	
15. Storage of selected objects parent		X	
16. Adjustment of object offset from hand so object does not “jump” when dropped		X	

Table 4. 2 Portability of requirements for implementing Voodoo Dolls

4.2.2.2 Analysis of requirements

As Table 4.2 shows, despite the simplicity of the verbal description of Voodoo Dolls, there are at least sixteen low-level requirements that have to be implemented. Six of these requirements are VR toolkit-specific, nine are scene-graph-specific, and only three depend on neither a VR toolkit nor a scene-graph. If we look carefully at the non-reusable components again, we note that they

can be placed into three main categories. The first is object position and orientation (requirement 6,10,14,16); the second is parent/child relationships (requirements 4,5,7,14,15,16); and the third is input (requirements 2,4).

Thus, for a simple implementation of Voodoo Dolls, less than 20% of the software components are trivial to reuse (assuming that the technique is implemented from modular components).

4.2.3 Summary of Analysis of 3DITs

From the requirements analysis of 3DITs in the previous paragraphs, it is clear that there is a strong coupling between VR toolkits, scene-graphs, and portable code. Therefore, creating reusable techniques is difficult. The amount of reuse that exists in the field is proof of this. This is not an insurmountable problem though; the previous analysis shows that most of the requirements fit into three main categories: input, position and rotation, and parent child relationships. If requirements in these three categories could be easily implemented in a reusable manner, then most of the techniques could be theoretically reusable. The remaining requirements that do not fit into these three main categories are generally parts that each VR toolkit provides in one form or another. In order to provide support for making the three requirement categories reusable, and to add support for the remaining requirements, we have designed a 3DIT development tool to augment current VR toolkit functionality. It is not possible to completely write the silver bullet of VR tools, but it is possible to make a large contribution in this area by creating a toolkit that can allow for reusable 3DITs.

4.3 Requirements discussion (3DITs with VR toolkits)

This section will go over the requirements used for building IFFI. At a high level the process for generating the requirements combines focuses of the previous research and determining what has been successful and what has not. The requirements process focused on the foundational work that was required to begin IFFI: current VR tools, the current 3DIT tools, and the previous analysis of a 3DIT as sources for gathering requirements. One last requirement is that the system must demonstrate that it can use a 3DIT in two different VR toolkits. The only known VR toolkits that support component reusability are DGL and CoVE so they were used to test techniques built with IFFI.

4.3.1 High level reuse considerations

At a high level, the ability to reuse the 3DIT in multiple applications and VR toolkits is the major goal behind IFFI. This will allow for the maximum amount of flexibility for 3DIT developers. Expanding upon this high level goal would be that the reuse would be at a component level, not a source code level. This research must avoid requiring developers having to custom modify their source code for each 3DIT that they want to use. The ideal is to have a framework added to existing code once, and this framework can then manage the addition and removal of 3DITs without having to write additional code.

4.3.1.1 Current VR toolkit considerations

Current VR toolkits are sufficient for providing device support and rendering options. Being able to use these two major components greatly enhances the reach and usability for IFFI. First off, creating a bridge layer between IFFI and these devices is a requirement, and there should be specific process to follow when creating this bridge. The bridge layer must also preserve the meta-data that comes from the VR Toolkit. 3DIT developers have to work closely with the devices, and preserving information about the environment is essential. An example of this would be the ability to specify keywords and metadata from a 6DOF device like an IS900 wand so that the developer can use the information to make decisions with their 3DITs.

From the foundational work, the key concerns is that the architecture of IFFI must not be a "center of the world architecture". It must be able to integrate with the current DIVERSE and CoVE architectures so that it can take advantage of the component reusability that these two toolkits offer. This requirement will allow for IFFI to integrate with the largest number of VR toolkits possible.

4.3.1.2 Technique considerations

All 3DITs require the ability to isolate the functionality for scene-graphs, VR toolkits, and device input. It is not feasible to write a new scene-graph or the ultimate VR tool; the specific operations that are the lowest common denominator are important to capture. It must also be possible to allow for "bare metal" coding with IFFI. This will reduce the reusability of the 3DIT, but could possibly increase the ease of development and amount of optimization possible. For scene-graphs the common operations necessary to support are position and rotation functions, and parent-child relationships. For VR toolkits, it is necessary to have a flexible architecture for

developers to use that augments their work flow instead of dictating it. For devices, the ability to model any device in IFFI and to be able to query information about the device is crucial.

From the directions of previous 3DIT tool research two major requirements were identified. The first is that there needs to be support for a data-flow architecture. This is the probable future for most 3DIT tools [BOYDD099] [WINGRAVC05] [VANACKEND06]. Three different research efforts have come up with variations for this method of work, and this research direction is the fourth. The second requirement is to allow for the specification of 3DIT components and device flow using a standard descriptive language. This will allow for tweaking of 3DITs to be much simpler than is currently the case. It will also allow for the higher-level configuration of input devices to 3DIT components than with traditional VR toolkits.

4.3.1.3 Summary of high level goals

In summary, here is a list of the high level goals that IFFI addresses.

1. Works with multiple VR tools
2. Allows 3DITs to work both within applications written with the same VRToolkit and with applications written in a different VR toolkit.
3. Component level reuse, not copying and pasting source code.
4. Has to reuse VR toolkit display and input features
5. Straight-forward bridge process between IFFI and VR toolkits
6. Preserve meta-data about the environment that the toolkit provides
7. Must not be the center of the world
8. Capture the lowest common denominator operations for scene graphs, devices, and VR toolkits
9. Not get in the developers way, the system must allow for lower reuse targets if desired
10. Support a data-flow architecture
11. Support a declarative language for configuration of the system

These are the goals that must be achieved in order to provide a toolkit that can produce reusable techniques. These high level goals support three of the four types of reusability (component based, between applications, and between toolkits). The fourth requirement of reuse is implied with the third and tenth goals. In order to provide more detail for the design of IFFI a more expansive list of high level requirements will be listed.

4.3.2 High level requirements for IFFI

It is next to impossible to design software with only the high level goals listed above and still obtain a reasonable result. After careful study of literature, techniques, and toolkits a list of high

level requirements has been produced. This section describes a classification for requirements, and then lists the requirements.

4.3.2.1 Classification of high level requirements

The higher level requirements are grouped into several different categories. These categories help to isolate, group, and make the overall list of requirements easier to read.

Main categories for requirements
Output (Focused on graphics, but could be haptics or olfactory in the future)
Input (Currently 3D input devices, but could be computer vision, or AR in the future)
Toolkit (Requirements that stem from previous toolkit work and current requirements gathering)
Techniques (Requirements stemming from techniques)
System (Requirements from the systems designed for projected usage)

Requirements classification
Requirements will be denoted by a R as the first character in the requirement. Requirements will take the form of [R]-M.Num where R is for requirement or design and M corresponds to the main category listed above, and Num is the number of the requirement.

Abbreviations for main categories:
O – Output
I – Input
T – Toolkits
E – Techniques
S - System
Examples:
R-O.1 – The output system must support the use of HMD VR systems

4.3.2.2 Higher level requirements

4.3.2.2.1 Output requirements

R-O.1 - The output system must support providing graphical output to HMD, PowerWalls, RAVE, and CAVE type systems.
R-O.2 – The output system must be flexible enough to support other display devices and possibly even haptic and olfactory devices in the future.
R-O.3 – The output system must provide direct access to the provided rendering toolkit. For example, if the output system is chosen to be OpenGL then OpenGL must be accessible.
R-O.4 – The output system should be flexible enough to support any OpenGL based content, or toolkits that produce OpenGL content (for example Open Scene Graph, VTK, and Coin).
R-O.5 – The output system should support the display of common 3D object file formats (for example, 3ds, obj, pfb, iv, and wrl files).
R-O.6 – The output system must run on desktop systems and must be able to simulate the previous immersive options on the desktop.
R-O.7 – The output system will provide a standardized description language of the hardware capabilities.

4.3.2.2.2 Input requirements

R-I.1 – The input system should support Intersense IS900 trackers and wands for testing with VT equipment.
R-I.2 – The input system should be able to add additional hardware devices should they become available.
R-I.3 – The input system must support both pushing events to designated event receivers and the ability to pull device information from the input system.

R-I.3-i – The events either pushed or pulled from the system should identify the device that generated them, and allow for accessing the state of the entire device if desired.

R-I.4 – The input system must be able to support both polling the latest information from the device and obtaining the latest information from a queue.
R-I.5 – The input system will provide a profile system that will allow for device capabilities to be specified in a standard manner and accessible at runtime.

4.3.2.2.3 Toolkit requirements

R-T.1 – The 3DI toolkit should not re-invent the wheel when possible. It should take advantage of VR software that is already available.
R-T.2 – The 3DI toolkit will provide the ability to dynamically load and unload standardized components at run time that can be used for configuration, graphical, logical, and similar capabilities.
R-T.3 – The 3DI toolkit allows for any of its components to be used separately or independently based upon what the particular technique or application needs.
R-T.4 – The 3DI toolkit will provide for the concept of a virtual world that contains models, users, and user defined objects. Properties of objects contained in the virtual world must be easily accessible to the developer.
R-T.5 – The 3DI toolkit will provide the ability for code to be executed in a data flow architecture approach similar to previous 3DIT toolkit research.
R-T.6 – Must be able to integrate into existing workflows and not introduce its own development paradigm.
R-T.7 – The 3DIT toolkit must provide a simple declarative language for specifying techniques.

4.3.2.2.4 Technique requirements

R-E.1 – Standard model file formats must be able to associated with appropriate scene-graph nodes and accessible by interaction techniques.
R-E.2 – A concept of a world where objects live and are manipulated should be present.

R-E.2-i- The world should have an object lookup table.

R-E.2-ii- The world should support categorization and meta data of objects.

R-E.2-iii – The world will keep track of what techniques are available and allow them to communicate with each other if desired.

R-E.2-iv – The world must have a generic interface to allow for graphical, auditory, and other types of content to be encapsulated in it.

R-E.3 Interaction techniques must be able to turn off input streams coming to them, and to disable them outright if necessary.
R-E.4 Techniques will have a standardized interface with graphical, update, and input callbacks registrable by the user.

R-E.5 The system should be built to allow techniques to be created, deleted, added, and removed from the world listed in R-E.2 at run time.
R-E.6 Techniques must be reusable.
R-E.6-i Techniques must be reusable with different VEs.
R-E.6-ii Techniques must be usable between different VR tools.

4.3.2.2.5 System Requirements

R-S.1 – The software must be written so that it can be used on multiple platforms if necessary.
R-S.2 – The software must be able to be controlled locally or remotely.
R-S.3 – The software should seamlessly scale from one VR environment to another without requiring code changes.
R-S.4 – Where feasible, features such as threading should be used to help future proof the system.
R-S.5 – The system must simulate different VR environments on a desktop for testing purposes.
R-S.6 – The system should run on a cluster based VR system used to power CAVE type systems.
R-S.7 – The system must support component based reuse at the system library layer.
R-S.8 – The system must be open source to foster further adoption.

4.3.2.3 Mapping of requirements to goals

All of the goals map to requirements. Some of them map to multiple requirements, some of them map to a single requirement. The first goal of working with multiple tools is augmented by the fourth goal of reusing both input and output features. The output requirements are a description of the typical VR toolkit, and if IFFI were to not reuse VR toolkits these requirements would cause IFFI to be a "silver bullet" tool. The goal that requires techniques to work with multiple applications is met via the technique and system requirements. The component level reuse goal directly translates into requirement R-T.2. The reuse of techniques between VR tools goal is met by requirement R-T1. The bridge between multiple VR toolkits goal is met by having output and input requirements that require IFFI to work with multiple toolkits instead of reinventing existing VR toolkit functionality in a new tool. The metadata goal is met by requirement R-O.7. This allows for developers to be able to write techniques that can change their behavior depending on the hardware attached. The non-center of the world goal is a requirement in order to fulfill the working with multiple tookits goal, which is tied to the output and input requirement sections. The lowest common denominator goal is addressed by the toolkit and technique requirements. In order to provide the worlds and objects listed in these requirements the lowest common denominator approach must be used. This goal and requirement are the distallation of the results from the two techniques analyzed earlier in this chapter. The developers freedom goal is addressed R-T.6. By not requiring developers to only

use IFFI and only IFFI it is possible to achieve this goal. The data-flow architecture is requirement R-T.5. The declarative language goal for configuration of the system is met through requirement R-T.7.

4.3.2.4 Summary of higher level requirements

These requirements are at a high level, but provide a clearer picture of what is necessary to provide resuablity for techniques. These higher level requirements were broken down into multiple categories. The output category is designed to ensure that the VR toolkit that is used is flexible enough to handle the different environments where these techniques have been used in. The input requirement section assures that IFFI will work with any hardware device and will work with developers preferred method of operation. The toolkit section synthesizes best practices from existing toolkits and ensures that IFFI will be portable between tools. The technique requirement section assures that there will be support for building techniques beyond just a standard update callback. The system requirements section makes sure that the toolkit is portable, provides usablity for multiple platforms, and provides for requirements for testing.

4.3.3 Summary of requirements

This section has detailed the requirements for building a reusable technique from both a high level and a semi-low level perspective. The high level requirements were geared at synthesizing the definitions of reusability with existing literature. This procedure was repeated with the low level requirements in order to provide an overall list of requirements. With these requirements, the design for a IFFI can be described. The design of IFFI will point back to these requirements to show that it has covered all of them.

4.4 Design of IFFI

In order to address the requirements, and to develop a system conceptually coherent with users, the agile software practice of a system metaphor was used. This helps explain the design to the developer, and to the reader of this design section. The overarching metaphor for this system is to think of worlds that contain objects. The world knows about objects and knows how to work on these objects to perform actions that augment the type of object present. The objects in the world are containers that hold information and have appropriate helper functions for their data. Another important point is that objects register themselves with the system via a function call. When an object is registered all of the different worlds that it associated with are notified of its

presence. When unregistering an object, all of the worlds that are aware of the object process its departure. This by itself is the foundation for understanding the major parts of the IFFI system. In each section in this design section, requirements will be cited so that requirements and design decisions can be linked.

These worlds alone are not enough to describe how the design for IFFI works. With any large software project, there are a few general design choices that account for 80% of what a user does. There are three major steps for users of IFFI. They are init, configure, and update. Init starts the system, creates the worlds, loads plugins, and populates the world with objects. In the configure step 3DIT techniques are made, objects can go to other objects in the system and query information for setup purposes. The update function sends a heartbeat pulse to the system. This is called by the user's code. Requirements R-S.1-8, and R-T.1-8 are covered by the overall architecture design of IFFI. This allows for IFFI to co-exist with current VR tools. A graphical representation of how this works is listed in Figure 4.1. There are several concepts that are listed in Figure 4.1 that have not been explained yet, but it is meant to give you an overall feel for IFFI works.

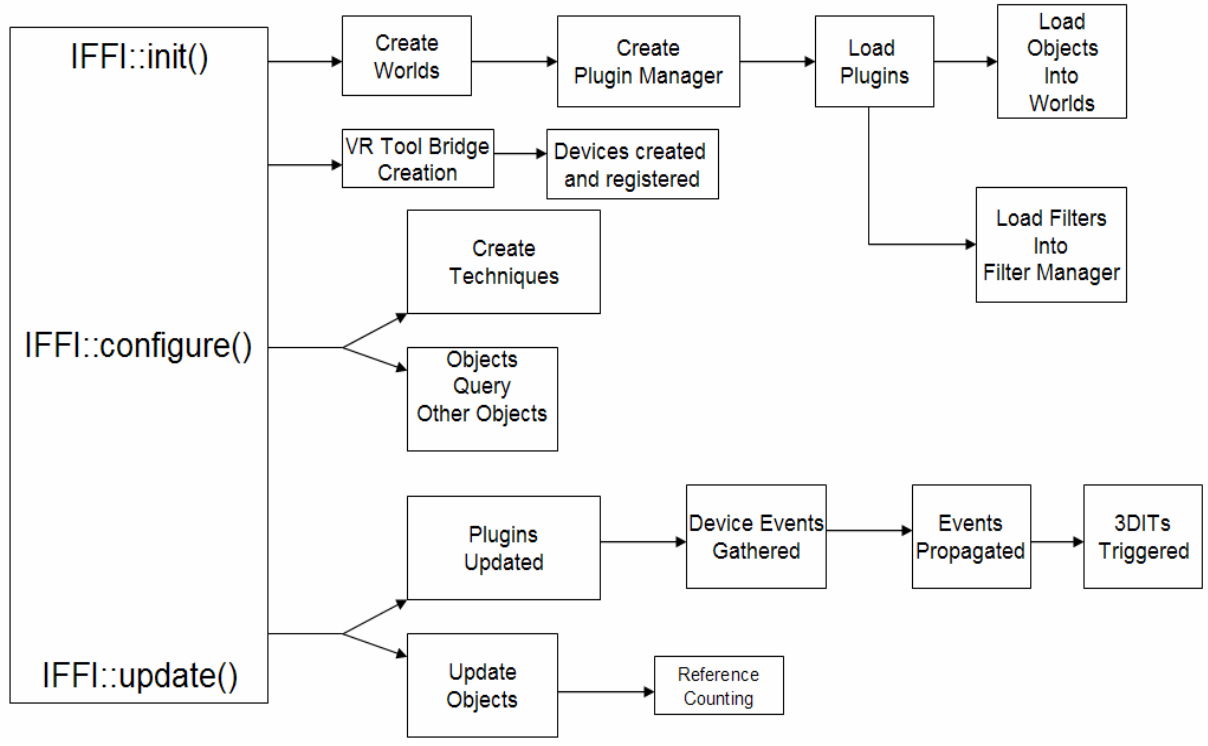


Figure 4. 1 IFFI program flow

A comparison is best used to understand how this architecture applies to typical VR toolkits. Figures 4.2 and 4.3 respectively show the architecture description for typical VR Toolkits and IFFI. With typical VR toolkits Applications and 3DITs are considered to be one and the same. Because of this, there is often a blurring of the line between VR Toolkits and 3DITs. With IFFI, 3DITs are first class citizens with their own category. This increases reuse because techniques are no longer forced to be coupled with particular applications. In general IFFI follows traditional VR toolkit design except that it is designed to work with other VR Toolkits, and provides a way to make techniques and applications reusable.

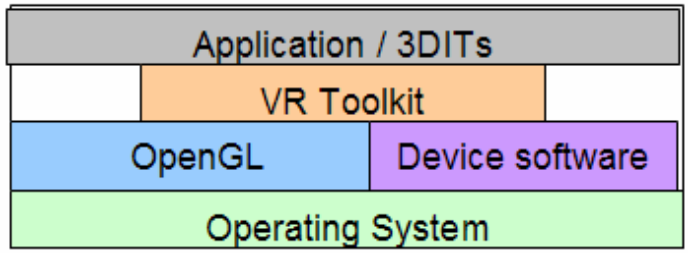


Figure 4. 2 Typical VR Toolkit Architecture Diagram

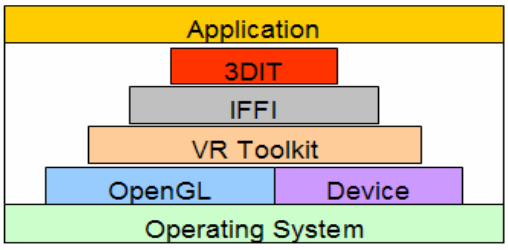


Figure 4. 3 IFFI Program Architecture

The major areas covered by this system are base objects, events, devices, filters, graphics, plugins, and graphics. There are also sub-areas that are added which are equivalent to "syntactic sugar." Each of the major areas are designed to map to the requirements listed in the previous section. Base objects fulfill R-E.2, events cover R-I.3-4, devices cover R-I.1-5, filters cover R-E.3-5, plugins cover R-E.6 and R-T.2, graphics cover R-E.1, R-T.1. It should be noted that the sole design decision to use DIVERSE / CoVE for graphical output takes care of R-R-O.1-7.

4.4.1 Description of IFFI subsystems

As mentioned earlier, the system metaphor for IFFI is objects and worlds. There are several objects and worlds that exist in IFFI. These are necessary in order to handle all of the different types of responsibilities listed in the previous sections. There are over ten "worlds" provided by

default with IFFI. These worlds cover the gamut of traditional VR toolkit support, add support for techniques, and provide flexility to add new functionality to IFFI. Because each world is specialzed, it would be trivial to add worlds and objects to support sound and physics functionality to this architecture.

4.4.1.2 Base Objects

In IFFI Base Objects are designed to provide metadata for describing objects and to allow them to be easily found by other parts of the system. Most objects in IFFI are derived from the BaseObject class. This class has a unique ID and groupID, a name, group (class) name, and can have a set of attributes. An attribute is a name-value pair that has a void* pointer for storing custom data.

4.4.1.3 Update Objects

VR toolkits often have objects that need to be updated for each system "tick". Update objects provide this functionality in a controlled manner. Examples of update objects are the plugin system, VR toolkit bridges, and reference counted objects.

4.4.1.4 Reference counted objects

Reference counted objects have reference and unreference methods that increment an internal reference counter. If this counter reaches 0, and the update callback is called, the reference world will delete the object. This makes it easy to handle memory allocation and deallocation in IFFI.

4.4.1.5 Events

Events are the "lifeblood" of information in IFFI. They are how information gets from one part of IFFI to another. The event system is based on a producer consumer model. Each producer and consumer is uniquely identified, and events are disseminated by type. The EventManager handles communication between producers and consumers. Producers identify which types of events they can produce and are called by the EventManager to produce events. Consumers identify which types of events they can handle, and are called by the Event Manager to consume events. The event manager makes it is possible for consumers to ask for all events of a specific type, specific events from a producer, or all events from a specific producer. Figure 4.4 shows how events are produced and consumed in IFFI.

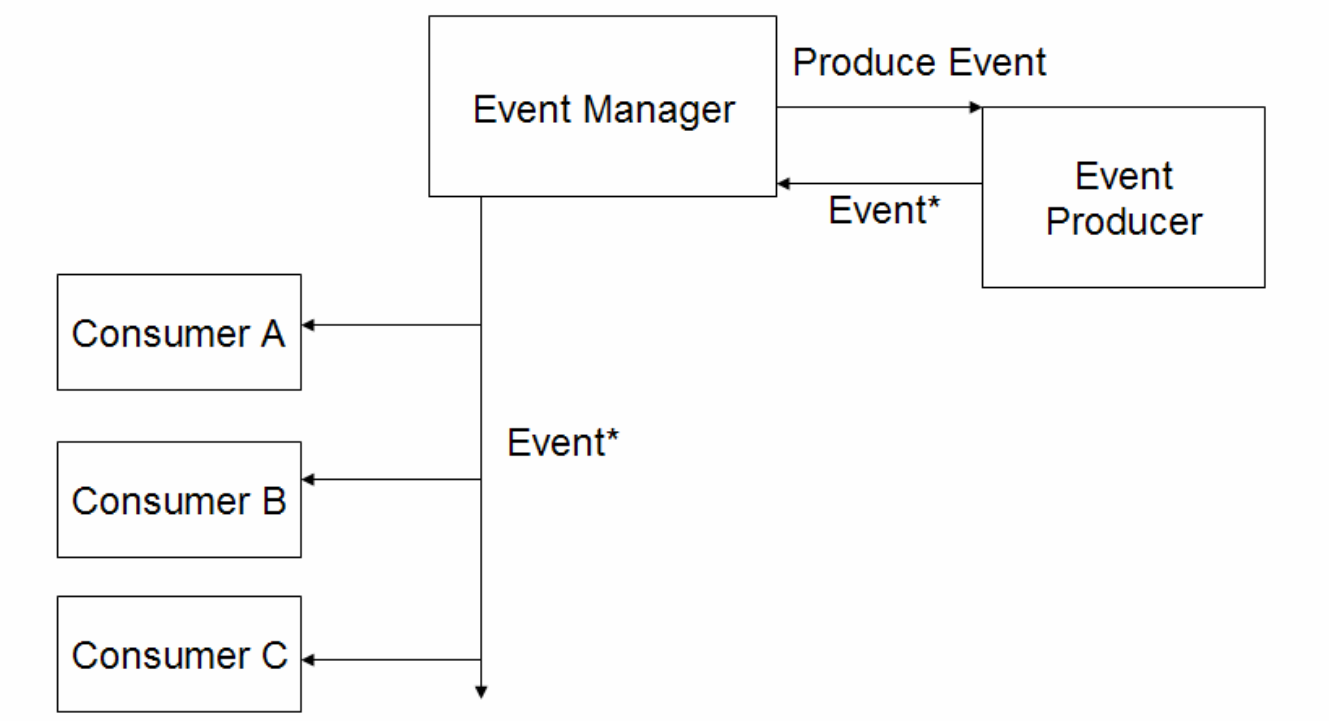


Figure 4. 4 IFFI Event lifecycle

4.4.1.6 Graphical objects

Graphical objects are designed to incorporate information about scene-graph nodes. They are derived from base objects and provide parent-child relationships, position, and rotation information about the nodes. The graphical world is a container for all of these graphical objects. It has functions to find Graphical objects in a simple manner. The goal of this design is not to implement a custom scene-graph for IFFI to use, but to provide a bridge layer between existing graphics packages. For example, Open Scene Graph nodes can be used to provide graphics functionality for a technique via these nodes. If another graphics toolkit is used, then another node type can be created that will provide the same functionality, and all of the techniques that used the original node can then use the new graphics toolkit without any modification. This greatly enhances reuse. Another use for the graphical objects is to provide a copy of the user's body in the graphical world that is updated by devices connected to the user. Figure 4.5 shows an example of this. The implementation of many techniques is aided by having the body of the user in the virtual environment.

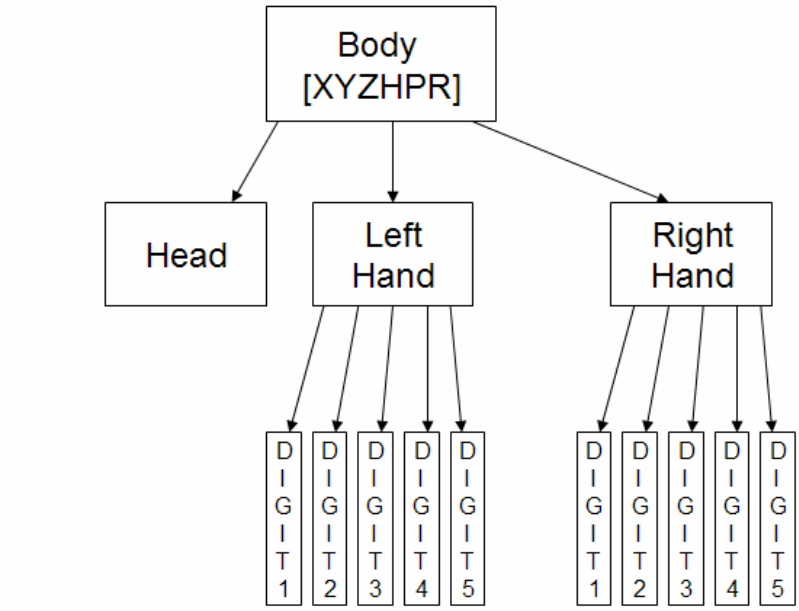


Figure 4. 5 The IFFI scene-graph representing the human body.

4.4.1.7 Devices

Devices are the heart of system information flow. They pump information from a VR device into the system. Devices are designed to be able to represent any piece of VR hardware. They are composed of a three-level hierarchy. The first is the device level, second is the component level, and third is the container level. At the device level there is a method to query or set meta data about the specific device, and to obtain information about any of the components that make up a device. Each component has a specific type (Button or Degree Of Freedom [DOF]) and some number of containers. Each container holds the specific data value for a device. For example, the IS900 has a DOF component that produces XYZHPR (position and Euler angle orientation) data. The IS900 device in IFFI has a DOF component with 6 containers. Each one of the containers represents one of the data values. It also allows for meta-data such as up, down, left, right to be associated with each data container. This allows for 3DITs to be built in a manner similar to the ALICE project. Figure 4.6 shows how an IS900 Wand is constructed with IFFI.

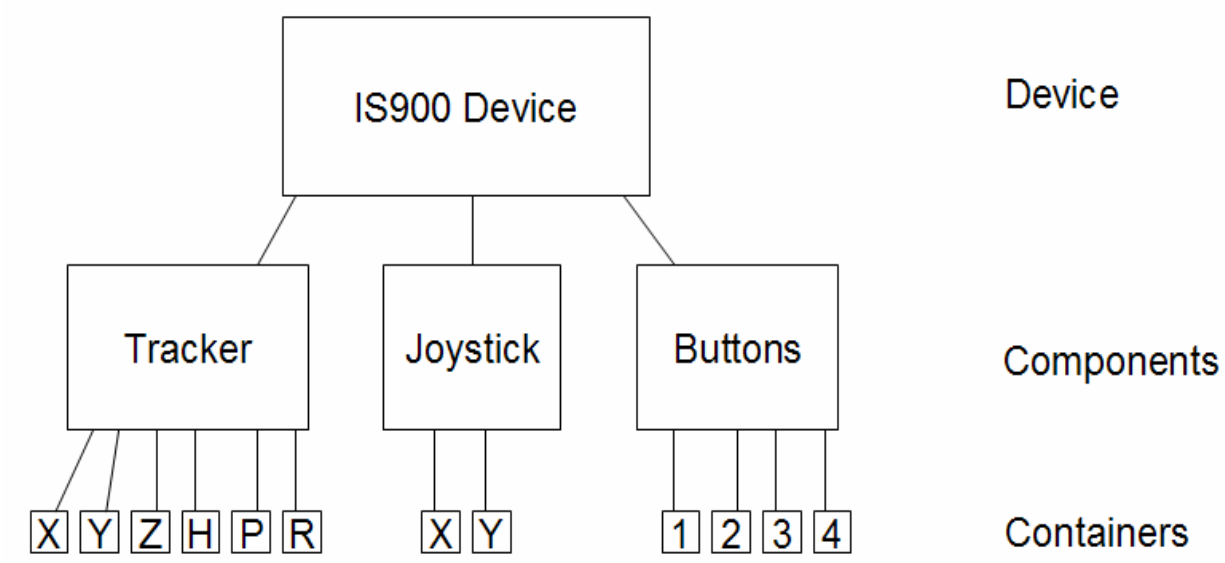


Figure 4. 6 IS900 Device represented using the IFFI device structure.

4.4.1.8 Filters

Filters are the arteries of IFFI. They are designed to take input from specific devices and to send it through a pipe and filter architecture to produce an interaction in the VE. Filters are managed by the FilterManager which handles feeding events to the filters. The filter manager works by keeping a list of the different filters in an internal list. Techniques are collections of filters created from clones of the filters in the FilterManager's internal list. These techniques are connected together in a manner specified either in the code, or by the configuration file. An example of how Filters work is listed in Figure 4.7.

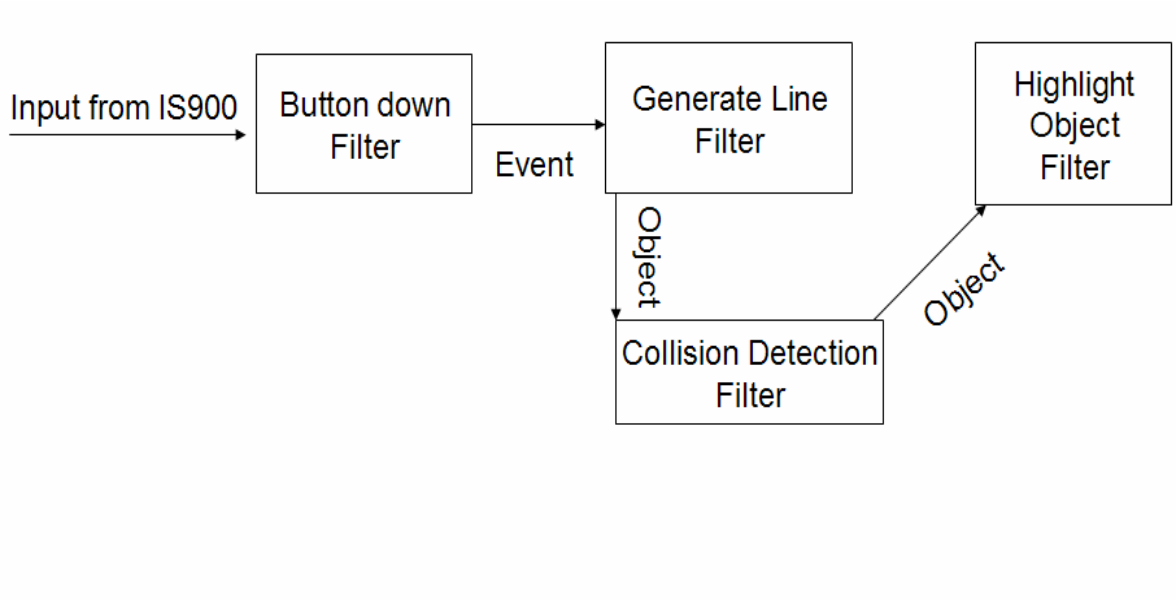


Figure 4. 7 Ray-casting technique broken down into IFFI Filters.

4.4.1.9 Zones

Zones are just containers for objects. They are designed to make developer's lives easier by providing a way to have multiple views of the same objects. For example, an object may be in a selectable zone, a viewable zone, a northern quadrant zone, and a simple geometry zone. This allows for easy book keeping for groups of objects where attributes may not be appropriate. Another example of their usage is that a traditional zone could be extended to be a container of graphical objects. A function could then be written to specify a boundary for the zone. Once this is done, another function can be used to tell the developer whether or not an object is in the

zone or not. When this is combined with the update object world it becomes a simple method to determine if users moved objects into a specific area during an experiment.

4.4.1.10 Plugins

Plugins are dynamic libraries loaded at runtime that have an init, configure, and update function. They also have priority levels for updating. In IFFI everything is a plugin, or at least loaded by one. The purpose for plugins is to populate the worlds in IFFI with objects, and to update information in IFFI. As listed earlier the first step IFFI goes through is an initialization step. This step is where worlds contained in the plugins are created, and the objects are initialized in the system. The next step is configuration. This is where the worlds loaded by the plugins may talk to each other. Lastly, the update "tick" provides a way for information from VR toolkits to be sent into IFFI worlds.

4.4.1.11 XML

The XML subsystem has been left until now. This is a meta level world that combines and controls the functionality of the existing world through an external source. This is an XML file written to describe devices, filters, zones, and graphical objects. The main goal for this subsystem is to provide for techniques to be built in code once, but to allow for techniques to be written outside of a traditional C++ coding environment. The next section provides examples of the XML files used to power IFFI.

4.4.2 Summary of the sub-systems in IFFI

IFFI contains over ten different sub-systems that are the "worlds," as mentioned earlier in this chapter. These systems provide the different functional components necessary for building reusable techniques. Plugins allow for functionality to be added to the system at runtime and provide a component level of reuse. The filter sub-system allows for techniques to be built using a data-flow. The device and graphical sub-systems allow for multiple VR toolkits to be used with IFFI. Several other sub-systems such as reference counting, zones, and XML provide useful functionality for creating techniques.

4.5 Implementation examples

There are multiple ways to use IFFI. In general it runs like any other VR toolkit. This involves creating an application, setting the proper configuration information, and running the application. Reuse is made possible via two specific design decisions. The first is the plugin system, the

second is the XML configuration system. Plugins can add functionality to any application, and configuration files specify how to use this functionality. Also, due to the nature of IFFI, there are no changes required in the technique code when a new toolkit is mentioned. First, an example of how the Go-Go technique can be implemented with IFFI will be presented, followed by next a full implementation of ray-casting with IFFI.

4.5.1 Designing the Go-Go technique

The Go-Go technique is a selection and a manipulation technique. The first component to consider is the state that Go-Go is in. The requirements analysis from an earlier section show that an object is either selected or it isn't. This state is normally changed via a button press event. Once this button press event happens, collision detection is used to attach an object to the hand that is moving in the world.

This description (along with the previous explanation of Go-Go) gives enough information to design the Go-Go technique for IFFI. One set of components that could be used to realize the Go-Go technique would be a state selection component, a button press detection component, a collision detection component, an object substitute component, a non-linear movement component, and an attach child component. All of these components are different filters.

The structure of how the filters are chained together is listed in Figure 4.8. The non-linear movement component would an independent component that would update a particular graphical objects position in the world to correspond to the location determined by the non-linear mapping definition in Go-Go. The component would take input from a 6 DOF tracking device and would use the user's body as a reference point. The state selection component would either call the collision detection filter or the attach child component to de-attach the object that was attached in the previous component. The input to the state selection filter can be anything, and its output is exactly what was passed in. As with the non-linear component, the state selector also only depends on IFFI and fulfills all four types of reuse. The object substitute filter would take any event as input and substitute another object as output. This allows for a button press event to come in and the right hand to come out. This is useful because it means that the state selector can send a button press event in and the next filter will know to work on the hand object instead of the button press event. The attach child filter takes an object as input and attaches it to a particular object or it takes any other type of event and detaches the object that it previously

attached. All of these components fulfill all four types of reusability because of IFFI's architecture. The collision detection component would take an object event (in this case the hand) as input and would check if it collides against all of the other objects in the world. The output of the collision detection filter would be the object that the hand collides with. This component will normally be scene-graph specific unless custom collision detection code is implemented in the filter.

These components show how it is possible to take a technique, decompose it into a set of requirements, and then have these requirements turned into components that can be used to realize the technique. There is not a one –to-one mapping between the requirements and the techniques, but there is some semblance of a mapping between requirements and components. The process used for producing these components and designing this technique is given in the next chapter. The method by which these components are actually specified and used in an application is described in the next section.

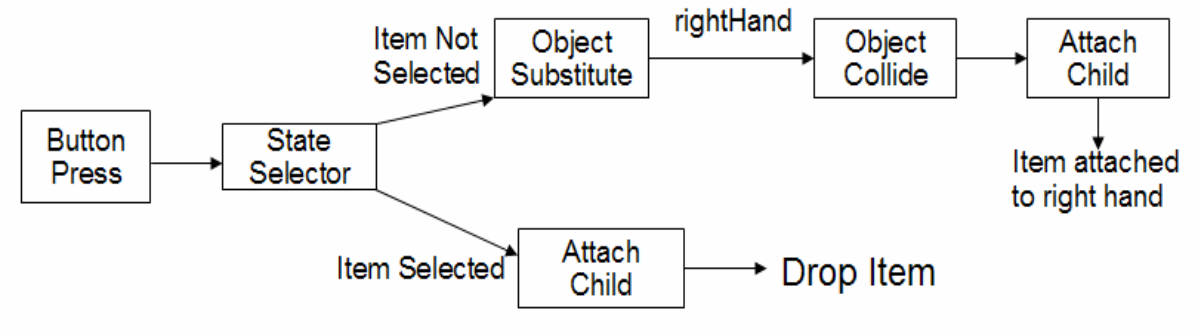


Figure 4. 8 Behavior connection diagram for the filters implementing the Go-Go technique

4.5.2 Example usage with Ray-casting:

This section gives an example of the actual script, XML, and C++ files that are used to construct an application that can load any virtual world or interaction technique. The script file starts a program designed to load configuration files into a skeleton environment and specifies what physical devices are connected to the system, the technique to load, and the virtual world that the user wants to load. The local file is an example of how specific devices are bound to common aliases and to specify which scene-graph to use. The ray-casting file gives an example of how the different filters for the ray-casting technique are specified and connected together. The world file demonstrates how to load graphical objects into the world. The C++ file demonstrates the

IFFI code needed to duplicate the functionality of the configuration files in the IFFI C++ API and how to initialize, configure, and run the IFFI system from within a C++ application.

Example script file to start the application:

```
#Add basic interaction techniques to IFFI, the DGL compatibility layer, and
IS900 device support #through DTK
export IFFI_PLUGINS=basicITs:dglBridge:dtkIS900
#iffiLoader is a program that loads IFFI with the specified plugins
#local.xml holds the local device information for IFFI
#raycasting.xml specifies how the raycasting technique is put together and
#adds it to the world
#world.xml specifies the graphical world
./iffiLoader local.xml raycasting.xml world.xml
```

Example Local file:

```
<iffi>
<local>
        <inputMap value="wand"    source="is900Wand:wtracker"/>
        <inputMap value="trigger"
                    source="is900Wand:buttons:button2"/>
        <inputMap value="head"     source="headTracker:head"/>
        <inputMap value="position" source="navigation:position"/>
        <graphicalHandler name="OSGGraphicalHandler"/>
</local>
</iffi>
```

Example ray-casting file:

```
<iffi>
<IT name="RaycastingSelection">
      <filter name="GenerateRay" input="trigger"
             output="ObjectCollide" arguments="rightHand:1000.0"/>
      <filter name="ObjectCollide" output="HighlightObject"/>
      <filter name="HighlightObject" arguments="0.0:1.0:0.0:.30"/>
</IT>
</iffi>
```

Example world file:

```
<iffi>
        <!--This is a graphical object that is uniquely named,
            and is specified to be an OSG model.
            It has both position and rotation values.
            It can be included in one or more zones (by a colon
              separated list)
            It can have certain attributes added to it so that you
            can use these in your interaction techniques if
              necessary. You can also specify the parent of the node.
            By default IFFI provides a standard world, ether, and
            scene node.
        -->
        <graphicalObject name="ball1" model="ball.osg" type="OSG">
                <position x="10.0" y="2.0" z="3.0"/>
                <rotation h="0" p="180" r="0"/>
                <zone name="testWorld"/>
                <attribute name="selectable" value="false"/>
                <attribute name="positionLock" value="true"/>
                <childOf name="world"/>
        </graphicalObject>
```

```
        <graphicalObject name="bookcase" model="bookcase.osg" type="OSG">
                <zone name="testWorld"/>
                <position x="-10.0" y="0.0" z="0.0"/>
                <attribute name="selectable" value="wowlookatme"/>
                <attribute name="positionLock" value="true"/>
                <childOf name="world"/>
        </graphicalObject>
        <graphicalObject name="book1" model="book1.osg" type="OSG">
                <zone name="testWorld"/>
                <position x="-0.0" y="0.0" z="40.0"/>
                <childOf name="world"/>
        </graphicalObject>
</iffi>
```

These examples show how it is possible to use IFFI without writing any C++ code. It is possible to accomplish the same actions above by writing a program that follows the steps above. All of the filters are assumed to be loaded into the system through plugins during the init call. The C++ program that implements the configuration files would look like this.

```
IFFI::init();
//Tell the IFFI to load the config file then configure itself.
IFFI::loadConfigfile("devices.xml");
IFFI::configure();
//Create the raycasting technique
FilterManager::createTechnique("raycasting");
FilterManager::addFilterToTechnique("GenerateRay", "raycasting");
FilterManager::addFilterToTechnique("ObjectCollide", "raycasting");
FilterManager::addFilterToTechnique("HighlightObject", "raycasting");
FilterManager::getFilterFromTechnique("GenerateRay", "raycasting")
      ->setArguments("rightHand":1000);
//Connect the raycasting technique to the IS900 device
FilterManager::feedTechniqueByID("raycasting",
DeviceManager::getDeviceByName("IS900")->getID());
//Load an object into the world
GraphicalHandler* maker = (GraphicalHandler*)
IFFI::getObjectByName("OSGGraphicalHandler");
maker->loadModel("book.osg", "book");
GraphicalWorld::getGraphicalObjectByName("book")->setPosition(1,2,3);
GraphicalWorld::getGraphicalObjectByName("book")->setRotation(10,0,30);
//Run the program until completion.
while (1)
{
      IFFI::update();
}
```

There is no error checking in this code to increase readability. This example shows the general steps for IFFI (init, configure, and update), how configuration files can be loaded, a 3DIT can be built, and how graphical objects can be added and.

4.6 Reuse comparisons with other tools

In order to understand the significance of the design direction that IFFI has taken with respect to reuse, the above explanation of IFFI needs to be put in context with how other VR / 3DIT tools

work. It is impossible to have an equivalent test between any of the different packages, but evaluating each tool by itself should show the major differences in reuse. A VR toolkit (VRJuggler) and a 3DIT tool (CHASM) were available for study, and are examined below with a respect to reuse.

4.6.1 VR Juggler

VR Juggler is a VR toolkit that is probably the best known of the VR toolkits built [BIERBAUMA01]. The normal method of building an application with VR Juggler is to derive off of a class that provides various update callbacks. From there the recommended method of implementing your program is a fill in the blank procedure. Everything except for code for the specific displays and devices must be included in the application. This includes navigations and other pieces of code that are reused. This does not lead to reuse within VRJuggler applications. There are several examples in their source distribution that show how code reuse is not considered.

In the VR Juggler examples directory there are a series of example programs for different graphics packages. There is a constant theme for how to load content and to navigate through different models in these sample programs. However, each of these programs shares no code with any of the other examples. The only reused code is the base adapters for the different graphics packages. None of the navigation logic, scene-graph setup, behavior nuances, etc is reused between different applications. In some cases (Open Scene Graph, and OpenSG) navigation is considered simple, whereas with other graphics packages (OpenGL and Performer) this task is considered difficult. The sizes of these programs (in lines of code) also reveal another detail, trivial tasks such as navigation can be non-trivial to implement. For example, the OpenSG navigation program is 362 lines, Open Scene Graph is 454, OpenGL is 568, and Performer is 1043. All of these programs could easily require days worth of work for a novice to become familiar with before they could successfully modify them for their own needs, especially with the Performer application. Another point to consider is that all of these examples hard code the device configuration into the application. This is not the actual device used, but it requires that the devices on the system be configured in a certain manner for these programs to work properly. Another low level concern is the fact that each application has its own driver program included. All of these are essentially the same, except that each one has a different application class that contains most of the program's logic included in it. The result is that if one change

were to happen, over twenty different example programs would have to be changed, instead of changing just one program that uses a component based approach to loading content. Lastly, the fact that graphics and VR toolkit contain dependencies in all of these examples make it impossible to reuse most of the code in any other VR toolkit, or even with a different graphics toolkit in VR Juggler.

From the sample code provided with VRJuggler it is clear that reuse for the end-user was not a design concern, and that it can cause a significant amount of effort to re-implement concepts from one program to another. There is significant work in VR Juggler to facilitate the reuse of devices and graphics though. The entire goal of VRJuggler is to become a virtual portable operating system that allows a program to be written once and experienced everywhere.

4.6.2 CHASM

CHASM is a 3DIT tool that provides a data flow architecture [WINGRAVEC06]. It focuses mainly on the specification of techniques. A more detailed description of CHASM can be found in chapter 2. A developer uses CHASM by creating or reusing a certain number of states, then using a 2D GUI to configure how these states fit together, then generates skeleton code, and then fills in the blanks that this code created. This allows for code reuse via all of states, but increases the overhead developers have due to understanding how the generated code works. Below is an example of how an example configuration file for CHASM is constructed to represent the Raycasting technique.

```
<CHASMConcept name="/Users/cwingrav/code/chasm0.3/Paper/Raycasting"
height="199" width="305">
  <state name="ignore" x="243" y="145" height="30" width="50"></state>
  <state name="inited" x="28" y="71" height="30" width="50"></state>
  <state name="ended" x="228" y="13" height="30" width="50"></state>
  <state name="checked" x="98" y="164" height="30" width="56"></state>
  <state name="newselected" x="215" y="80" height="30" width="79"></state>
  <state name="unselected" x="92" y="27" height="30" width="70"></state>
  <symbol name="del"></symbol>
  <symbol name="init"></symbol>
  <symbol name="end"></symbol>
  <symbol name="rechecking"></symbol>
  <symbol name="selecting"></symbol>
  <symbol name="unselecting"></symbol>
  <transition from="inited" to="inited" by="init"></transition>
  <transition from="inited" to="checked" by="rechecking"></transition>
  <transition from="ended" to="ended" by="end"></transition>
  <transition from="checked" to="newselected" by="selecting"></transition>
  <transition from="checked" to="unselected" by="unselecting"></transition>
  <transition from="newselected" to="ended" by="end"></transition>
  <transition from="newselected" to="checked" by="rechecking"></transition>
  <transition from="unselected" to="ended" by="end"></transition>
  <transition from="unselected" to="checked" by="rechecking"></transition>
  <component name="button"
concept="/Users/cwingrav/code/chasm0.3/newvocab/Button" access="PROTECTED"
      cif="PASSED"></component>
  <response component="button" state="up" symbol="rechecking"></response>
</CHASMConcept>
```

This shows that CHASM components are focused on having the finest level of granularity. Instead of the three components listed in the IFFI example, there are nine in this example. The data passed between these components are also strongly typed whereas in IFFI they are loosely typed. There is a concept of input into the 3DIT via the component and response tags, but there is no way to abstractly map between a physical device and a button press event that is sent into this technique.

The next observation to notice about CHASM is the implementation of the technique itself. From the above file configuration file, ten different header and implementation files were generated. These files do not include the button concept. Each of these files has specific parts that must be filled in and customized to implement this technique. A cursory glance at these different files shows that there is a large learning curve to be able to work within the framework with this tool due to a heavy usage of macros and predefined implementation structure. The reuse of the components themselves is heavily considered in CHASM, but reusing specific parts of the application that the developer writes is not taken into consideration. The VR toolkit used

to realize the CHASM states is hard-coded into the files and states themselves. This removes any possibility of reuse between VR toolkits.

4.6.3 Summary of reuse methods with related VR toolkits

In summary, there are considerable differences between VR Juggler, CHASM, and IFFI. They have radically different approaches to implementation and require work at different levels (code / configuration files). Some are designed solely to abstract hardware and leave a set of blanks for the developers to fill in, while others try to go beyond the hardware aspect and to allow for specific parts of applications to be reused. Some pre-generate code and structure for you, others require you to understand and work within the structure that has been defined. However, neither of them allow for component reuse between both applications in the same VR toolkit, and between VR toolkits. IFFI is designed to address these concerns and to allow for reusability at the component level both between applications in the same VR toolkit and between applications using different VR toolkits.

4.7 Summary of IFFI

This chapter has presented the design and usage of IFFI. It showed how techniques are built, the similarities between techniques, and how these can be leveraged to build reusable techniques. After this, requirements were generated from investigating techniques, toolkits, literature, and existing experience. These were catergorized into high and low level requirements. From these requirements an overall design for IFFI was presented. A description of the sub-systems inside of IFFI and how they relate to technique development was presented. After this, an example of how to implement a technique with IFFI was presented. Finally, a comparison between reuse with IFFI and other toolkits was made. This showed that IFFI provides more reusability with less work than existing systems.

Even though an example of how ray-casting could be implemented with IFFI was listed in this chapter, an in-depth study of reusability using this tool is necessary to show that it is possible to build reusable techniques with this tool. The next chapter will focus on building techniques with IFFI and to investigate the effect of reusability when building multiple techniques. Lastly, IFFI needs to be evaluated by others, this is described in chapter six. The source for IFFI can be found at: iffi.sourceforge.net.

Chapter 5: Case study of Reusable 3DITs

5.1 Introduction

In the previous chapter we introduced IFFI, a system designed for to allow 3DITs to be developed in a reusable manner. The types of reuse that this system provides to techniques are between VR Toolkits reuse, between applications reuse, component level reuse, and the reuse of parts of techniques in other techniques. The purpose of this chapter is to prove that techniques built with IFFI exhibit these reuse features and to demonstrate how to build reusable techniques to outside developers. This chapter is a practical test for IFFI, for without reusable techniques to prove that the system is reusable, this research is essentially vaporware. This chapter will focus on how to build the varying aspects of techniques and the process used to build the techniques. This process entails for designing a technique, creating the software design for a technique, and finally the actual implementation of the technique.

Much research has been focused on techniques, but little of it is on practical reuse and construction of the techniques. For example, taxonomies have been used to help define the behavior of a technique, but there are no standard published methods for actually building techniques [BOWMAND04]. This is problem because from our personal observations, the lifecycle step that requires the most time is the implementation of the technique. Here at VT a normal lifecycle for a technique is two to four weeks time to design its behavior, eight to ten weeks of implementation, and then one to two weeks of evaluation. By increasing the reusability of techniques, IFFI will help decrease the time spent in the implementation cycle.

The sections of this chapter will be a description of the techniques used, a process for designing these techniques, analysis of reusability with these techniques, a description of how reusable components can be used to prototype a technique, and conclusions about these techniques. This chapter does not explore others experience with developing techniques with IFFI, only the creator's experience. The next chapter will deal with evaluating how others are able to build reusable techniques with IFFI.

5.2 Techniques used

There are a myriad of different techniques that could be used to show reusability. However, an exhaustive list of creating techniques and researching the various connections between them are

outside the scope of this work. In order to illustrate the different cases of reusability, three different techniques are all that are necessary. These three techniques are the ray-casting technique, the Go-Go technique, and the Homer technique. These are almost canonical techniques and are well known to most of the VR research community. While they are relatively simple techniques they are representative of more complicated techniques (see Voodoo dolls and Go-Go comparison in the previous chapter). After a short description of the techniques, the rationale for why they were chosen to demonstrate reusability between techniques will be given.

5.2.1 Ray-casting

The ray-casting technique is a selection technique. It works by selecting object(s) that touch a virtual ray. The ray is usually created from a 6DOF tracker that the user is holding. This ray is combined with a visual representation of the ray in the world to provide feedback to the user. Whenever a button press or other trigger event happens the object(s) are selected. The result of this selection is usually shown to the user by changing the object(s) color, or by placing a bounding sphere around it [MINEM95].

5.2.2 Go-Go

The Go-Go technique is a selection and manipulation technique. It is simple to describe verbally: at a distance less than some threshold, the virtual hand follows the user's physical hand; beyond this threshold, the virtual hand's distance extends farther than the physical hand based on a non-linear function. When the virtual hand touches an object, it can be picked up with a trigger event (e.g., button press), manipulated, and an object in-hand can be dropped with the same trigger event [POUPREYVI96].

5.2.3 HOMER

The HOMER technique is also a selection and manipulation technique. It is a combination technique that incorporates ray-casting and a modified virtual hand manipulation technique. This technique combines the effectiveness of each technique and uses each technique to address the limitations of the other technique. The HOMER technique works with the ray-casting technique at first, then once the user selects an object, the virtual hand moves to the object's location and a modified virtual hand technique is used. The modification to the traditional virtual hand technique is that the user can move the selected object to their body by moving their hand to their body. This works regardless of the distance to the object. This is done by a linear

scaling of the distance between the user and the object selected. The advantage of this technique is that it is easier to do fine grained rotation of an object than with the Go-Go technique. Once the object is moved into the desired location, then it is dropped and the HOMER technique switches to ray-casting [BOWMAND97a].

5.2.4 Rationale for techniques

In order to show that IFFI allows for reusability, the techniques chosen must exhibit some reusable features. These techniques were chosen because they are simple to understand and there are several possibilities for reuse with these techniques. As stated in the introduction, the features we are interested in showing are component level reusability, the ability to reuse parts of one technique from another, component level reuse, and most importantly, reusability between VR toolkits and applications written with the same VR toolkit. These techniques provide the ability to test these types of reusability.

For component level reusability the goal will be to show that techniques can be broken down into multiple parts, and these parts can be loaded and unloaded at runtime inside of VR applications. By using these components it will be possible to demonstrate that parts of techniques can be reused in different VR applications and between different VR toolkits. For reuse of parts from one technique to another technique, there are several possibilities. Ray-casting is used inside of the HOMER technique so it should be possible to reuse the components from ray-casting when building HOMER. Also, there is collision detection in all three of the techniques for selection so this should be able to be reused among all the techniques. In order to demonstrate independence between VRToolkits, the techniques will have to be demonstrated using two different toolkits.

5.3 Design Process for techniques

There has been a considerable mention of reusable techniques and a system given to help build them, but until now there has been no formal description of how to build one of these techniques. This section will describe how to use IFFI to build a reusable technique. However, this process is not tool specific. It is a process that can be used to build techniques with other 3DIT development tools. Many software development strategies can be used to build techniques, but there are a few key steps that need to be emphasized during specific phases of development in order to maximize the potential for reuse. These steps are breaking a technique down into a

series of steps, turning these steps into components, breaking the components into reusable parts, and reusing other parts if possible. This is not a completely novel approach to building techniques: it derives and expands ideas from Hierarchical task design [KIRWINB92] and task analysis [BOWMAND97]. These are two examples of previous research that covers how to break techniques down into respective parts. However, these research steps never take reusability into account, and halt at the theoretical level, they do not translate directly to implementation, whereas this work takes this step.

5.3.1 Breaking a technique into a series of steps

When creating a technique, the first step is to define the flow of actions for the technique. For example, with ray-casting the technique can be broken into the actions of pointing, selection, confirmation of selection, and feedback [BOWMAND04]. For the Go-Go technique the different parts of the technique are to extend the arm in a non-linear manner then check for collision detection with a virtual hand in the world with other objects. The purpose of this step is to crystallize the behavior of the technique. The next step is to define the information flowing into the technique, the transformations that this input takes, and how it flows until the technique is done. This allows for all of the information necessary for the next step in the process to be completed.

5.3.2 Turn steps into components

In order to move beyond the high level steps in the previous part to towards implementation the previous steps have to be broken down into components. The goal of this activity is to specify the components in a manner that can be implemented. For example, the high level action of selecting an object in the ray-casting technique could be broken down into an input trigger component, a selection component, and a highlight component. While creating these components, developers must specify how information flows from these components so that the components can communicate with each other.

5.3.3 Breaking components into reusable parts

The last design step before considering other reusable parts is to analyze the components for future reusability. For example, the input trigger component in the previous step is normally written to take a trigger event from a specific VR device, but to obtain higher reusability a generic ButtonEvent from IFFI can be used instead of a particular VR toolkit's button event.

This allows any technique that uses ButtonEvents to work with selection trigger. This step is not an exact science: it relies on the skill of the developer to recognize the ability of components to be reused in the maximum number of ways. The goal for this step is to generalize the components as much as possible to increase the potential for reuse.

5.3.4 Reuse other parts where possible

The last step is to see if there are other components available that can be used when building a technique. For example, if the trigger component in the previous step was created during the implementation of the ray-casting technique, then this component could be reused during the creation of another technique. The main source of difficulty with this step is that components often will not fit exactly with the needs of a different technique. For example, the input event to a component may be in a different format than the new technique provides. If this is the case, an event adapter that takes an input event and transforms it into the desired event type must be implemented. The worst possible case is that a component provides only half of the needs for a new technique, and there is not a feasible way to integrate an additional component that provides the missing half of the functionality into the new technique. If this is the case, a new component that encompasses all of the required functionality will have to be implemented. The source code from the existing component that provides partial functionality may be used to help build the new component, but this is not optimal reuse. However, this is still better than starting without any code at all.

5.4 Design process for the three techniques

This section will show the design process in action for the ray-casting, Go-Go, and the HOMER techniques. This will help provide an example of how to build these techniques, and will provide a small library of components that can be reused with other techniques.

5.4.1 Ray-casting

Step 1: Breaking a technique into a series of steps

The first step in the ray-casting technique is to point at an object. The next step is to select the object. The last step is to provide feedback from the selection. The information that flows into this technique is a button press and the data from a 6DOF tracker. If a button is pressed then the data from the tracker is sent into the selection component and is used to generate a ray in the world. Once this ray is created it is used as part of collision detection on a set of specified

objects. An object that is selected by collision detection is highlighted. Finally, when the ray is in contact with an object its color should be changed to provide feedback to the user.

Step 2: Turn steps into components

The steps of the ray-casting technique can be condensed into four components. The first is a button down component that activates the next component in the chain. The second component is a selection component that generates a line and does collision detection with other objects in the world. The third component is a highlight component that places a semi-transparent bounding sphere around the object. An object substitute component is necessary to change the color of the ray.

Step 3: Evaluate components for reusability

In order to maximize component reusability, components should be broken down into reasonable granularity. For example, the selection component mentioned in the previous section can be broken down into a generate line and a collision detection component. These components should then be implemented in a configurable manner. For example, the generate line component should allow for the size of the ray to be specified. The collision detection filter should allow for specific zones to be specified when checking objects. The highlight object should allow for the color and level of transparency to be specified. The object substitution component should allow for any object to be passed to it as input and another object identified by the arguments to be substituted in place of the incoming object.

Step 4: Reuse other parts where possible

This is the first technique created with this component so it is not possible to reuse components with this technique.

5.4.2 Go-Go

Step 1: Breaking a technique into a series of steps

The Go-Go technique is an arm extension technique, but it is also an selection technique. The steps of Go-Go are identical to ray-casting even though the technique is different. The first step in the Go-Go technique is to move your hand onto an object. The next step is to select the object via a button press event. The next step is to attach the object to the hand, and then move the object with the hand. Finally, after another button press event, the object is "dropped" and is no longer updated by the hand. The information that flows through this technique is a 6DOF tracker

event which is then transformed (perhaps non-linearly) into a position relative to the users body position along with the object being moved.

Step 2: Turn steps into components.

Go-Go has three components: the first component of Go-Go is a non-linear movement filter that takes device events from a 6DOF tracker and then changes the position to be non-linear if it passes a certian threshold. The next component is an attachment component that detects if the hand collides with another object in the world, and then makes that object the child of the hand or releases the object.

Step 3: Evalutate components for reusability

This technique has two states: an object is selected or it is not. A state selector component can be created that allows for the formal specification of these states so that this functionality is not added to the attachment component. The attachment component should be broken down into a collision detection component and an attachChild component. The collision detection component was already implemented with ray-casting. The attachChild component takes an object and makes it the child of a specified object.

These are the components necessary for creating the Go-Go technique. However, their interfaces are not defined yet, and this must be done.The non-linear movement component should take a DOF event, allow for a threshold and an exponent to define the non-linear movement, and a way to specify what object to move non-linearly. The attachChild should take an object event or also take no arguments to undo the attachment that was previously done. A more advanced attachChild / detachChild component pairing is possible, but is not necessary at this time. Finally, the state selector component will be a simple linear list of states that are advanced one at a time unless the filter chain that it calls returns an error, which means it will not advance the state. The are a myriad of different ways to handle state transitions, but this simple method will suffice for this technique and possibly for others.

Step 4: Reuse other parts where possible.

The button down component and the collision detection components from the ray-casting technique have already been implemented and can be reused in this technique.

5.4.3 HOMER

Step 1: Breaking a technique into a series of steps

The HOMER technique is broken into two other techniques. The first is ray-casting and the second is a modified virtual hand technique. The ray-casting technique has already been described, and the virtual hand technique is identical to Go-Go except that it moves linearly instead of non-linearly. The major difference with the HOMER technique is that the selection of the object is done with ray-casting instead of the position of the hand. Once this selection is done the user may move the object to their body by moving their physical hand to their body. This works regardless of the distance between the object and the user. The steps of the technique are to select with ray-casting and to move with the virtual hand. The flow of information is a button press and a DOF event that are transformed into a ray in the world. The ray is checked to see if it collides with other objects in the world. The collided object is then updated to the position of the virtual hand as it moves. Events from a tracker then map the movements of the virtual hand in the VE.

Step 2: Turn steps into components

There are several components in the HOMER technique. The first is the ray-casting component. This component is comprised of the components mentioned earlier. The second component is a non-linear updating component that moves the virtual hand and the object selected from the location of the object selected to the body of the user. The fourth component is the button down component described earlier.

Step 3: Evalute components for reusability

There is an element of state in this technique so a state selection component is necessary. The ray casting technique already exists and provides an object that it selects. This can be fed into a component that moves it and the virtual hand in a linear manner between the selection position and the user's body. This could be broken down further into a component that sends an event to multiple filters, a component that takes an object and provides a location / rotation event, and a component that moves objects between two positions in the world in a linear manner mapped to the user's hand. However, for this specific technique, the lowest level of granularity was not followed. The purpose for these components is not to be bare bones, but to promote reusability where possible.

Step 4: Reuse other parts where possible

The state selector, button trigger component, and collision detection are all the same with this technique. The only component to be implemented for this technique is the HOMER specific movement component.

5.5 Implementation

The previous section detailed the design process for these techniques. This section goes over the actual artifacts produced in the implementation. These artifacts are configuration files for each of the three techniques, and the lines of code (LOC) for the components implemented. The LOC are presented to illustrate the amount of effort necessary to implement and use these components.

5.5.1 Ray-casting configuration file

```
<iffi>
      <graphicalObject name="ray" model="redRay.ive" type="OSG">
            <position x="0.0" y="0.0" z="0.0" />
            <rotation h="0" p="0" r="0"/>
            <zone name="baseWorld:Ray-castingZone"/>
            <attribute name="selectable" value="false"/>
            <attribute name="positionLock" value="false"/>
            <childOf name="body:rightHand"/>
      </graphicalObject>
      <graphicalObject name="coloredRay" model="yellowRay.ive" type="OSG">
            <position x="0.0" y="0.0" z="0.0" />
            <rotation h="0" p="0" r="0"/>
            <zone name="baseWorld:Ray-castingZone"/>
            <attribute name="selectable" value="false"/>
            <attribute name="positionLock" value="false"/>
            <attribute name=”hidden” value=”true”/>
      </graphicalObject>
      <IT name="Ray-castingFeedback">
            <filter name="GenerateRay" input="wand:position"
            output="ObjectCollide"
                  arguments="rightHand:20.0"/>
            <filter name="ObjectCollide" output="ObjectSubstitute"/>
            <filter name="ObjectSubstitute" output="HighlightObject"
                  arguments="ray"/>
            <filter name="HighlightObject" arguments="1.0:0.0:0.0:.30"/>
      </IT>
      <IT name="Ray-castingSelection">
            <filter name="GenerateRay" input="trigger" output="ObjectCollide"
                  arguments="rightHand:1000.0"/>
            <filter name="ObjectCollide" output="HighlightObject"/>
            <filter name="HighlightObject" arguments="0.0:1.0:0.0:.30"/>
      </IT>
</iffi>
```

5.5.2 Go-Go configuration file

```
<iffi>
      <graphicalObject name="hand" model="hand.ive" type="OSG">
            <position x="0.0" y="0.0" z="0.0" />
            <rotation h="0" p="0" r="0"/>
            <zone name="baseWorld:GoGoZone"/>
            <attribute name="selectable" value="false"/>
            <attribute name="positionLock" value="false"/>
            <childOf name="world"/>
      </graphicalObject>
      <IT name="GoGoMovement">
            <filter name="NonLinear" input="wand:head:position"
                  arguments="hand:head:rightHand:5.0"/>
      </IT>
      <IT name="GoGoSelection">
            <filter name="StateSelector" input="trigger" condition="first"
                  output="ObjectSubstitute" arguments="first:second"/>
            <filter name="StateSelector" input="trigger" condition="second"
                  output="AttachChild"/>
            <filter name="ObjectSubstitute" output="ObjectCollide"
                  arguments="hand"/>
            <filter name="ObjectCollide" output="AttachChild"/>
            <filter name="AttachChild"  arguments="rightHand"/>
      </IT>
</iffi>
```

5.5.3 HOMER configuration file

```
<iffi>
      <graphicalObject name="homerRay" model="ray.osg" type="OSG">
            <position x="0.0" y="0.0" z="0.0" />
            <rotation h="0" p="0" r="0"/>
            <zone name="baseWorld:homerZone"/>
            <attribute name="selectable" value="false"/>
            <attribute name="positionLock" value="false"/>
            <childOf name="body:hand"/>
      </graphicalObject>
      <graphicalObject name="homerHand" model="hand.osg" type="OSG">
            <position x="0.0" y="0.0" z="0.0" />
            <rotation h="0" p="0" r="0"/>
            <zone name="baseWorld:homerZone"/>
            <attribute name="selectable" value="false"/>
            <attribute name="positionLock" value="false"/>
            <childOf name="world:body:rightHand"/>
      </graphicalObject>
<IT name="HomerSelection">
            <filter name="stateSelector" input="trigger" condition="first"
                  output="generateRay" arguments="first:second"/>
            <filter name="stateSelector" input="trigger"
                  condition="second" output="homerMovement"/>
            <filter name="generateRay" output="objectCollide"
                  arguments="length:20.0"/>
            <filter name="objectCollide" output="homerMovement"/>
            <filter name="homerMovement" input="wand:homerHand"/>
</IT>
</iffi>
```

5.5.4 Size of reusable components

The size of the components is an important artifact to consider. If these components are large enough to cause implementation and maintenance difficulties then IFFI has failed in its goals. In order to demonstrate that IFFI does not exhibit these characteristics, only the selection components were implemented, the feedback components were not. The purpose of this work is to show the feasability of this direction, not produce perfectly polished techniques.

The size of both the configuration and LOC for each component are listed in this section. For the configuration files, ray-casting and Go-Go are both 24 lines long. HOMER is only slightly larger at 28 lines. The LOC for the components used in this section can be found in Table 5.1. This table shows that in general components are not large enough to be problematic, and are generally managable. It also shows that the header files and configuration files are similar in size.

Filter name	Header LOC	Implementation LOC
AttachChild	27	175
GenerateRay	27	141
HighlightObject	31	115
NonLinear	39	144
HomerMovment	42	208
ObjectCollide	21	88
ObjectSubstitute	23	76
StateSelector	24	82

Table 5. 1 The LOC for the reusable components built for three 3DITs.

Starting with the ray-casting technique, 423LOC (header and implemenation) had to be written. This is a manageable amount of code for a devleoper to write. Next, for the Go-Go technique 699 LOC had to be written. Of these 699 LOC, 109 were reused by using the same ObjectCollide component in the Ray-casting technique. Finally, the HOMER technique required

934 lines of code, and 684 were reused from the previous techniques. This small example shows that component level reusability is possible with IFFI and that the more components available for reuse, the less code that a technqiue developer will have to write.

5.6 Analysis of reusability

This section will highlight the different aspects of reusability that the previous sections have discussed. The main areas of reusability addressed will be application and VR toolkit reusability, component level reuse, and reuse of components from previous techniques. The other types of reusability are inherent to IFFI's architecture.

5.6.1 VR toolkit reusability.

In order to test component level reusability, VR toolkits that support component level reuse in some form or fashion are necessary. To our knowledge, there are only two different VR toolkits that support component level reuse. These are DIVERSE and CoVE [KELSOJ02][COVE07]. Both of these tools allow for shared libraries to be used as plugins (components) that can add and manipulate functionality. Some examples of the plugins in these toolkits are navigations, displays, and configurations. Plugins from CoVE are not compatible with DIVERSE. This makes for an excellent test framework for IFFI. These two toolkits were chosen to avoid having to write low level plugin loading libraries for IFFI.

For DIVERSE the method of implementation was a simple driver program that loaded configuration files for IFFI and started DIVERSE. Once this was done DIVERSE plugins could be loaded with this skeleton program, and these plugins can contain IFFI. These DIVERSE plugins can be used with any application written for DIVERSE. For CoVE a small plugin was built. This plugin started and configured IFFI and can be used with any CoVE application. No changes to the source for either of these toolkits were made during the creation of the bridges for IFFI. Also, no changes to the techniques were made. The nature of the bridgework consisted of two major steps: configuration and update. During configuration the IFFI scene-graph was added into the VR toolkit provided scene-graph, and IFFI devices were created from the specific VR toolkit configuration. During updates, device and navigation information from the VR toolkits were pushed into IFFI.

5.6.2 Device Independence

Another measure of reusability for techniques not mentioned earlier is device independence. This is traditionally a VR toolkit level topic, but it is critical for techniques. Due to IFFI's structure and bridges, the devices used in these techniques do not matter as long as they can provide a button click event and provide 6DOF output. The xml configuration files show that only specific named links are referred to in the techniques. For example, the input for ray-casting is called trigger. The xml configuration section in the last chapter shows how a specific device can be linked to the input for any technique. For both CoVE and DIVERSE information from an IS900 was used to power these techniques.

A second concept to consider with device independence is that even though these techniques have been implemented to use traditional 6DOF trackers another possibility for device independence exists. Due to the nature of IFFI, it is possible for techniques to query the devices in the system and determine what their capabilities are. This is done via the metadata that is associated with each device. This allows for techniques to adjust their capabilities based on the runtime environment. An example of this could be a ray-casting technique that works with a traditional 6DOF tracker in an immersive environment, and works via a mouse on a desktop to select where to shoot the ray in the world.

5.6.3 Reuse of components

The reuse of components is important in two different ways. The first is that components themselves are reused wholesale, and that the reuse is not copying and pasting code from one file to another. The second is that components built during the construction of a previous technique were used in the creation of a successive technique. Reuse happened at the component level for the different techniques because each technique was broken into several individual parts that could be reused. For example, the HOMER technique was the best example of component level reuse in techniques because after building the Go-Go and Ray-casting techniques only 250 lines of C++ code had to be written in order to create this technique. The Go-Go technique was able to save some effort because one component was reused from the ray-casting technique.

5.7 Prototyping

One aspect of reusability that has not been investigated is the role reusable components take in prototyping new techniques. During the implementation of the HOMER technique, it was

possible to prototype its behavior from the components built with ray-casting and Go-Go techniques. With the Go-Go technique, the general structure of the HOMER technique was available, but not its specific linear movement between the selected object and the users body. This section will go over the structure of the Go-Go technique, the changes necessary for the HOMER technique, and how the technique was implemented.

5.7.1 The Go-Go technique

The behavior of the Go-Go technique implemented with IFFI can be found in Figure 5.1. The Go-Go technique is similar to the HOMER technique because it is both a selection and manipulation technique. It also deals with moving an object in a manner different than in real life. The differences between the GoGo and HOMER technique are that the GoGo technique works based on moving a hand around for selection whereas the HOMER technique uses ray-casting to select the object. Also, the GoGo technique uses non-linear movement of the hand whereas the HOMER technique uses a linear scaling between the object and the user and the distance of the users hand and the body.

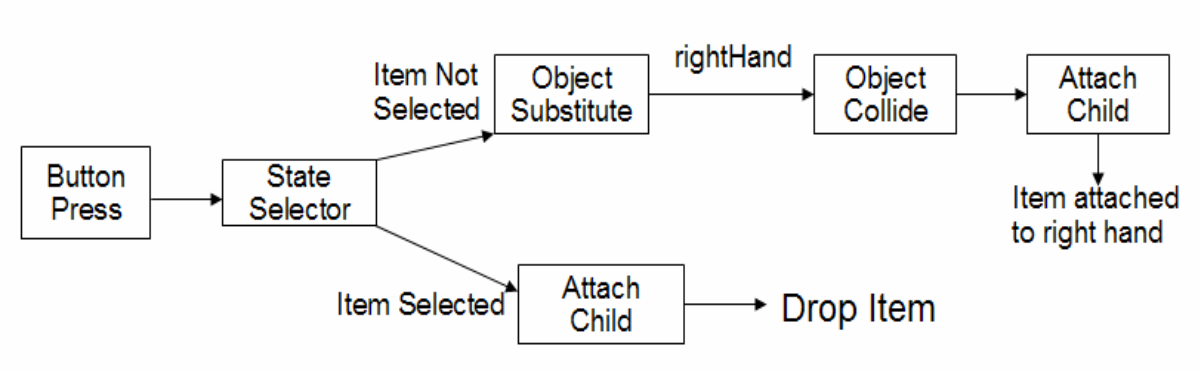


Figure 5. 1 Behavior of the Go-Go Technique

5.7.2 The prototype HOMER technique

In order to prototype the HOMER technique only minor changes from the existing Go-Go technique were needed. These were to change the object substitution plugin with the generate ray component and to move the object based on the wand, not the right hand. This allowed for an object to be selected via ray-casting instead of the virtual hands location in the world. This small change allows for the object to be moved in a manner similar to the HOMER technique, except the object will not move all the way to the users body. If an object is selected close enough to the users body then it will be possible for the user to see how the HOMER technique

works in comparison to the Go-Go technique. A description of the differences in behavior can be found in Figure 5.2.

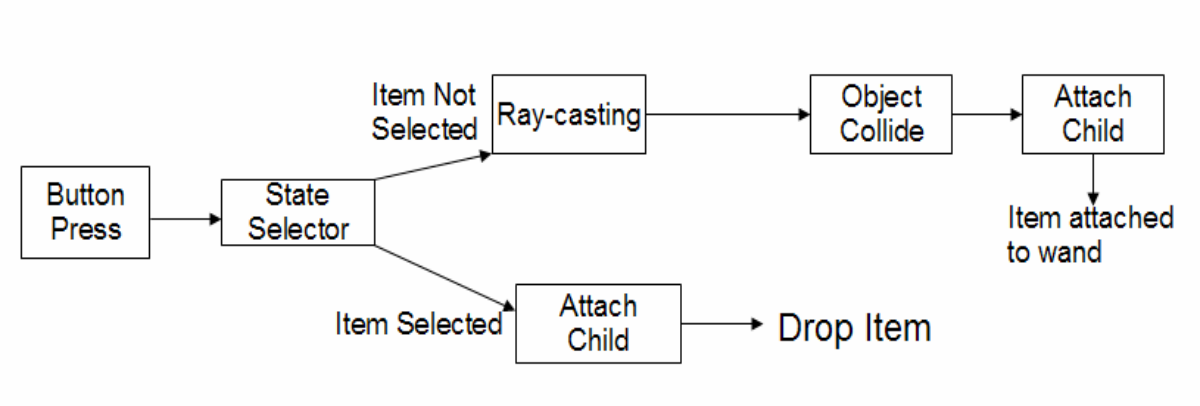


Figure 5. 2 Behavior of the HOMER prototype technique

5.7.3 The actual HOMER technique

In order to implement the actual HOMER technique only one component had to be implemented. This is the HOMER movement component. This component takes an objects and updates its position based on the wand movment. Whenever it receives a button press it stops updating the objects position. This was simply substituted for the Attach Child component listed in Figures 5.1 and 5.2. An example of the behavior of the actual HOMER technique can be found in Figure 5.3. The structure of the GoGo technique corresponds almost identically to the HOMER technique. The last point to consider is the actual configuration file change. Out of the five lines comprising the technique, only three changes had to be made. The first was to remove the object substitution component and replace it with the generate line component. The second and third changes were to replace the two references to the Attach Child componet with the Homer Movement component. This shows that it is possible to reuse configuration files as well as the components created with the techniques.

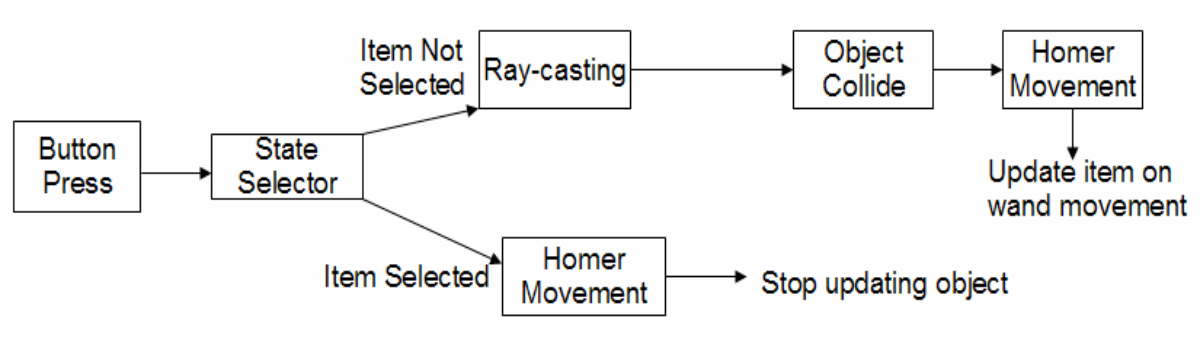


Figure 5. 3 Behavior of the HOMER technique

5.7.4 Summary of prototyping

This section has described prototyping a new technique from existing components. This another benefit of reusable components. This section describes the very minor changes that had to be made to an existing technique in order to allow for the prototyping of a new technique. These amounted to changing three lines in a configuration file to acheive general functionality of the technique. When the actual technique was implemented the results of the prototyping were able to be reused and only one component had to be added in order to make the technique behave in the manner described by the HOMER publicaiton.

5.8 Conclusion

This chapter showcases techniques built with the IFFI technique. It provides a process that goes from creation to implementation, and shows several different types of technique reusability by implementing three different techniques. These techniques worked on two different VR toolkits, between any application written with these toolkits, allowed component level reuse (not source code level), were device independent, and showed that components could be reused from a previous technique in a successive technique. HOMER was the best example of reusability because it was almost completely implemented completely from previous techniques. The behavior of HOMER was prototyped completely from existing techniques, showing another benefit to reusable techniques.

The next chapter will deal with evaluation of IFFI to see if other developers can build reusable techniques using the process listed in this chapter, and whether or not the research that they normally do can be done using this tool. Even with the advantages presented in the previous paragraph, this tool is relatively useless to the average practitioner if developers cannot easily build techniques from components.

Potential future work would be to build a library of techniques that collect metrics on the creation process over time. Even though construction of three different techniques was enough to show the potential power of this tool, a more in-depth study of techniques and their components would provide better insight into how reuse technique components. It is one of this work's goals to allow developers to achieve the reusability that this work achieved with the HOMER technique.

Chapter 6: Evaluation of IFFI

6.1 Introduction

The third research question was to demonstrate that the tool produced in chapter four could produce reusable techniques. In order to answer the third research question IFFI was evaluated in several different ways. The first evaluation of the tool was done in chapter five. A basic test of any tool is to verify the fact that it solves the problem it was designed to address. This test was carried out by building three interaction techniques. Beyond this, the techniques implemented produced a concrete demonstration that IFFI exhibits the four types of reusability that this work claims to address. However, just demonstrating that it is possible to implement reusable techniques with IFFI leaves out a major factor in demonstrating that reusable techniques can be produced. This factor is the users. The success of a tool is determined by those who use it. In order to demonstrate that IFFI can produce reusable techniques, other developers must be able to use it to create reusable techniques with it. In order to demonstrate this, two more forms of evaluation were done on IFFI. The first was a high level evaluation done by outside researchers, and the second was a more in-depth internal evaluation. These evaluations focused on verifying that we had identified an approach that could provide reusability, that it was possible for technique creators to build techniques using IFFI, and to identify the downsides of this approach.

6.2 Conceptual evaluation

In order to reduce bias from evaluating IFFI with people solely from the research group here at VT, it was decided to have IFFI conceptually evaluated by outside experts. We submitted and had a paper accepted at the ACM VRST conference [RAYA07]. Because of this, it was possible to interview researchers that work on VR toolkits and 3DIT tools who attended this conference. Due to the structure and time constraints of the conference we decided that it would not be appropriate to do a low level evaluation of the tool at this time. The goal for this evaluation was to gather high level impressions about the direction the work was taking, to verify the novelty of this approach, determine how others view it in relation to existing work, and to determine potential problems with this research direction. The structure of this evaluation consisted of

one-on-one interviews and several informal conversations during the conference. Due to the nature of the conference (spur of the moment conversations and location constraints) no audio recordings or formal transcripts are available.

6.2.1 Participant demographics

The participants we talked to at VRST were mainly VR researchers ranging in age from their early twenties to late fifties. The education level of the participants ranged from upper level undergraduate students to professors. The majority of the participants were tenure track professors or industry professionals. Three VR toolkit creators and one 3DIT specific toolkit creator participated in this evaluation. The other participants are active researchers in the VR field. The gender ratio was 90% male and 10% female. There were a dozen people that contributed to this evaluation over a period of three days.

6.2.2 Presentation

We designed a presentation to go over several aspects at the research at varying levels of detail. This presentation framed the research in context with current approaches, explained its purpose, and described the process for building techniques with IFFI. After an example of how to build the ray-casting technique a high level overview of the internals of IFFI was described. The goal was to show an overview of features inside of IFFI, but not bog the audience down with details. In order to achieve this, the concepts of worlds and objects were described, the body scene-graph was described, and then devices were explained. Finally, events and filters were explained and a description of how to put the information listed earlier into actual practice was given. This presentation allowed the audience to understand the context of the research, how to build techniques with IFFI, and how IFFI supported building techniques. The presentation used can be found in appendix A.

6.2.3 Informal interviews and conversations

This presentation was given to several people during the conference, and dozens of informal conversations ranging from a few minutes to over half an hour took place. The results of these two types of conversations can be broken into three specific areas. The first is the problem of reuse itself, the second is reuse in daily work, and the third relates to the quality of software engineering that this work exhibits.

6.2.3.1 The need for reusability

One of our first priorities was to verify that the problem we identified was considered a problem by the outside community. We found this to be true with two specific quotes: "The real problem is that nothing survives past its creator other than the publication" and "Reuse is a real problem." These two quotes show that reuse is important, and is one of the critical areas that considerably slows new research in this area. All of the participants in the study agreed with these two quotes. If the only reuse that is happening at a publication level, the months of work necessary to complete a publication must be repeated for each person that wants to reuse the work.

After the baseline of needing reuse was established, several other questions about the type of reuse normally came up. Two quotes from these conversations were "Not writing code is extremely important" and "What level of reuse are you going for, for example re-compiling?" These responses showed that people desired reuse, but had not put much thought into implementing reuse. This is probably due to the large hurdle required to implement reusability. The first of these comments points to the need for VR work to mature to the point where an advanced developer is not required to implement VEs and interactions. The success of the Alice project shows what can happen when non-developers can start to create VEs [CONWAYA00]. All of the participants agreed that the approach IFFI takes could theoretically provide the level of reuse we are targeting.

Several downsides to reuse were also mentioned in this area. The statement "Creating for reuse increases the complexity of development" was mentioned several times. Another concerned mentioned was the tool itself. There have been myriad multiple numbers of toolkits designed for creating VR applications over the years, so the question of how this approach avoided creating yet another tool was raised numerous times. A comment in the current level of reuse section also applies here. Even though there is a desire for reusability, the question of how people create reusable techniques was raised.

6.2.3.2 Current level of reuse in VR

The second area of discussion that came up while discussing IFFI concerned the level of reuse taking place in current work and the problems associated with creating reusable solutions. The people that talked about this area were developers and users of the higher level 3DIT tools mentioned in chapter two. These were a group of people familiar with the problem of reuse. Several of them had already developed one particular method to address this problem, even if it

reuse was not the central goal of their research. Two major quotes that came out of these conversations were "There is a large difference between application and components" and "There is a major learning curve with this type of work." Although efforts to create reusable components have started to take place, there are no published research results in VR. These two quotes show that reusability is difficult in this area, especially since most developers are used to working at a VR toolkit level and never considering reuse. The largest problem one of the researchers had was having their developers think in terms of how to take a problem and break it down into reusable components instead of just building an application that would solve the problem. In many cases this required a domain expert to be part of the design process to help steer the development in a reusable direction.

6.2.3.3 Contributions in Software Engineering

The last major area that the discussions covered was Software Engineering (SE). Some of these comments are controversial, but they raise valid points in three areas.

The first area was how IFFI related to existing toolkits. One major quote from this area is "This work correctly characterizes VR toolkits and the method to interoperate with them." This comment validates that we have correctly identified what a VR toolkit is, and how to interoperate with them. This is critical for the bridge concept to work properly. Finally, our system was described as "(IFFI) models what people need". This shows that existing VR toolkits are necessary, but people need another layer of abstraction so they can focus on their specific work.

The second area was the quality of IFFI. The foundational work has been used by dozen of developers, and we expect that this may be case with IFFI, so it was important to make sure that IFFI was high quality. One quote that expressed this was that "It collects the best practices of these [VR] tools." Even though there are several problems with VR toolkits discussed in chapters 1 and 2, there are several positive areas. By collecting the best practices that people are used to using, the learnability of IFFI is potentially increased. The last quote in the previous section also applies to this area. IFFI addresses the problem of reusability, and focuses on developer's needs, not on requiring "implementation olympics" to do their work. However, there has not been extensive usage of IFFI so it may not be learnable at the desired level.

The third area of comments dealt with the state of publishing software engineering work in VR conferences. The purpose behind these comments was to provide encouragement for continuing development of IFFI even if the publication venues did not accept it. The major

quote that exemplifies this section is “Software engineering is a real contribution and is not understood.” The conversation that led up to this quote made it clear that the architecture powering IFFI was a real contribution and was rarely seen at VR conferences. After this comment was made, a discussion ensued about VR paper review sessions, and how large numbers of systems papers are not accepted because most of the reviewers do not possess the knowledge necessary to understand the contributions of systems level work. In addition to this, Software Engineering conferences regard VR system work as too narrow focused for publication. Obviously, there is a strong need for authors of systems papers to describe their contributions clearly and convincingly, but if a well published expert who has served on review panels makes this statement, then this point should be not taken lightly. The point of stating this is not to blow the proverbial horn showcasing this work, but to bring attention to the difficulty of bringing solid problem-solving technical solutions to the attention of the VR community in general. Technical solutions that are well engineered like IFFI are required to solve the reusability problems that this work described.

6.2.4 Summary

The evaluation of the outside community has shown that IFFI contributes to the field of VR. Outside researchers have agreed that the architecture can theoretically provide the level of reusability proposed for techniques. There were three major areas that arose during this evaluation. The first was the need for reusability, second was the level of reuse in the field, and the third was the SE contributions of this work. Several downsides to the approach IFFI is taking were also mentioned. The training time that is necessary to learn VR tools can be daunting. Another downside is that designing and developing reusable components is difficult. Most developers are used to creating all-encompassing applications, not separating the technique from the application and additionally creating the technique in reusable parts. These criticisms apply to IFFI, but there are substantial differences between IFFI and previous approaches so these differences may mitigate some of these criticisms. In order to help address these concerns a developers guide to IFFI has been created and is available in appendix B. This covers the concepts of IFFI, provides a reference for the capabilities of IFFI, and gives examples of reusable technique development

6.3 In-depth developer evaluation

So far there have been two different methods of evaluation listed. The conceptual evaluation showed that IFFI could theoretically provide the support for building reusable techniques. Chapter five demonstrated that at a code level it is possible to produce these techniques. What has not been demonstrated is that developers can follow the processes that were presented in chapter four to build techniques from scratch and from existing components. Also, the low level internals have not been reviewed by others. Addressing these areas was the goal for the internal evaluation.

6.3.1 Evaluation procedure

The evaluation was broken into four different parts. The first part was a presentation (the same one given at VRST), next was a code review, then a test to see if it was possible to build a technique given a library of components, and finally a test to see if it was it possible to develop components from scratch to implement a particular technique.

The presentation given was the same one at the conference. The presentation provides a good overview of IFFI and provides a foundation so that the subjects can perform the next three parts of the evaluation. Instead of asking questions about IFFI based just on the presentation, questions were held until the next part of the evaluation happened. This was done to provide more information about IFFI and to allow to subjects to relate the theoretical information in the presentation with the practical information in the second part.

The second part was a code review. The presentation in the previous section contained an example interaction technique, and this part of the evaluation took one of the components from this technique and showed its implementation. The component chosen was a button down event filter. This was chosen because it was simple, yet demonstrated several different concepts of IFFI. For example, with a button press a certain device action triggers the technique to start. An event is produced which is then sent into the next filter in a technique. The generic event then has to be translated into a button event, and finally the button event has to be described as up or down. Once the button is determined to be down then the next filter should be called otherwise an error should be returned since the button down component does not handle button up events. These were the steps addressed during the code review.

Questions asked after the second part of the experiment

- What do you think of the program flow?

- Does this make sense?
- Will this allow you to do the work that you currently do?
- Is there anything that you don't see with the tool that you want to see?
- What is your initial impression of IFFI?
- How does this compare to systems you are familiar with?
- What are the benefits / downsides of IFFI that you see given this brief exposure to it?

After the code review the next step was ask the subjects to build the selection portion of the HOMER technique using a list of components. The subjects were asked to specify how the technique worked in the XML format that was described in chapter four. Below is a list of the components and their descriptions. Subjects were allowed to assume that input would come via a trigger event and would be sent to their filter.

Components and technique sample code for the third task

generateRay – A filter that takes a length of X and generates a ray that is passed on to the next filter. The arguments that it takes are the object from which to generate the ray from, and the length of the ray. For example arguments="hand:80".

stateSelector – A filter that takes a button down event and calls a filter specified for that state, then advances the state. I.e. state1->filter1, after a button press it would be state2->filter 2. The arguments that this filter takes are a colon delimited list of states. By default the first state is executed when the first button down event happens, then the second state for the second press, etc.. When the stateSelector reaches the end of the list, it goes back to the first state, i.e. this is a circular list of states. If a filter that the stateSelector calls returns an error, then the state is reset to its initial state and is not advanced.
An example of the XML that allows for the state selector to work with two states is:
<filter name="stateSelector" input="trigger" condition="first" output="firstFilter" arguments ="first:second"/>
<filter name="stateSelector" input="trigger" condition="second" output="secondFilter"/>

attachObject – Takes an object and attaches it to another object in the world specified by its arguments

objectCollide – A filter that that takes a specific object and determines if it collides with another object and passes the first collided object on to the next filter in the chain

Example Raycasting 3DIT with input from an is900 button

```
<iffi>
        <IT name=”Raycasting”>
                <filter name=”generateRay” input=”trigger”
                        output=”objectCollide” arguments=”hand:80”/>
                <filter name=”objectCollide” output=”highlightObject”/>
                <filter name=”
        </IT>
</iffi>
```

Questions asked after the third step

- How difficult was it to build this technique?
- What problems did you run into?
- Is the syntax intuitive?
- What are the benefits / downside of this approach?

The next step dealt with the subject's ability to develop a technique from scratch and to break it into a set of reusable components. The goal for this section of the experiment was to test the subject's ability to design a technique in a reusable manner. If this is possible, then previous work shows it is possible to implement these techniques and that other users can reuse these components in their techniques.

Question asked for the fourth step of the experiment

What filters would you create and how would you put them together for the Go-Go technique? Please provide a description of each filters functionality, and describe the information, and its form, that is passed to the next filter. Feel free to work with your own notation as long as the technique is broken down into components with data flowing through them.

Questions asked after the fourth step:

- How difficult was this task?
- Is this a process you normally go through when building a 3DIT?
- What were the benefits / downsides that you see for designing a technique this way?

After these questions were asked a final set of questions were asked to determine the subjects overall interest and impression of IFFI. The goal for these questions was to gather an overall impression of the system, how it related to other tools, how it supports building techniques, changes they'd like to see in IFFI, and to gauge future interest in the system.

Final questions for the internal evaluation

- What is your overall impression of IFFI?
- What did you like about IFFI?
- What didn't you like about IFFI?
- Is there a marked difference between developing with IFFI than with a traditional VR toolkit?
- Do you believe that IFFI makes developing 3DITs easier? Why?
- Does IFFI provide the support you need for developing 3DITs?
- What would you change about IFFI?
- Does IFFI match your design process for building 3DITs?
- Would you use it in the future?

6.3.2 Participant Demographics

The target group for this evaluation was graduate students that had technique implementation experience. The purpose of this group was to bring in "expert" users that could evaluate this work in terms of techniques they had developed or had experience with. Due to the nature of VR and technique development unless a subject has had prior development experience it is next to impossible to give valid answers to questions. VR and technique development is a small area and there were only four subjects qualified and available to evaluate this work. These were all male graduate students between twenty and thirty years old. For an expert evaluation this is a reasonable number of subjects.

6.3.3 Results

In order to analyze the answers each part will be individually addressed and each question will be addressed individually. Finally, a summary for each part and an overall summary will be presented. This section will contain all of the details collected during each part of the evaluation, but there will be a summary section that synthesizes the responses in each part. Finally, topics such as observational notes about each particular area such as critical incidents and questions that came up during each part will be mentioned in the summary sections.

6.3.3.1 Results for the Presentation / Code review

What do you think of the program flow?

The results for this question state that the program flow was learnable, clean, easy to understand, intuitive, easy to use, and straightforward. Beyond this it was stated this activity gave the participants a good idea for how IFFI worked. One other benefit to the code structure shown was that it was less work for the developer than with traditional VR toolkits.

Does this make sense?

The responses for this question were all positive. The participants mentioned that they understood how multiple filters could be put together in order to make a cohesive technique just from looking at one filter. They also mentioned that it would be nice to combine and use filters in different ways after implementing them. The other comments stated that IFFI followed standard coding conventions and was easy to understand.

Will this allow you to do the work that you currently do?

There were several discussions on previous and current research projects when this question was asked. The results from these discussions were that IFFI could address all of their previous and current research interests. The only downside discussed was that bridges for specific VR toolkits would need to be written to cover other VR tools. One particular discussion brought up the question of how IFFI could be used without the filter system. This was caused by the issue of animation. In order to solve this problem a developer would need to use the objects and worlds systems in IFFI in conjunction with a timer object. This would influence the object's positions in the world and accomplish the desired effect. This does not require the filter system, and shows that the data flow architecture is not required for every particular development scenario. However, if a developer wants to use the filter architecture it is also possible to create a timer object that generates an update position event which is then sent into a technique that knows which object to move in the world. Problems such as this show that IFFI is more than just filters, it was designed to take support for building VR applications to the next level, and thus allows for multiple solutions to a particular problem.

Is there anything that you don't see with the tool that you want to see?

This question exposed one deliberate area that was not heavily explained in the presentation. This is exactly how IFFI interacts with tools such as DIVERSE and Open Scene Graph. There were questions about how all of the pieces fit together, and how the abstractions made actually work. The next question raised was if it was possible to create a GUI tool to create the XML files. This is a valid concern and will be addressed in the future work section. There was also a question about how flavors of techniques could be created with this tool. This question was answered later in the evaluation when the participant had time to understand how techniques were built with different components.

What is your initial impression of IFFI?

The initial impressions of IFFI were quite positive. IFFI was called a promising approach, a "neat architecture", and that it could be promising if users were found to start using it on a daily

basis. Another benefit was mentioned was that IFFI provided cross platform development across both hardware and VR toolkits. IFFI was also described as a "great way to build flavors of techniques." One concern mentioned was that a library of components for reuse would be critical to the success of IFFI. This will take time to develop and is not part of the scope of this work. Another problem mentioned is that learning how to build components will be a problem. The last two problems that were raised by this question were that it would require time to adjust the way of thinking from traditional VR toolkit development, and that it was not clear how it would work with OpenGL programs. Good documentation and examples are the best that can be done at this stage of the research to address the transition from traditional VR toolkits. However, it was possible to describe the answer for the OpenGL question. Inside of the draw callback the filter manager would have to be called and this would allow for filters to be used to draw OpenGL graphics.

How does this compare to systems you are familiar with?

There were several different responses to this question. However, they can be broken down into two different categories. The first relates to lower level tools, and the second is the theory behind development of techniques themselves. It is interesting that answers about theoretical concepts came up at this question instead of the intended systems level answers. This may be due to the novelty of IFFI and the ideas that were stimulated from the presentation of this new system.

The first area of responses dealt with comparisons to other tools. These tools ranged from lower level graphical tools to traditional VR toolkits to higher level 3DIT tools. The following list contains most of system level comments about IFFI:

- Complements existing VR tools
- Similar to CHASM and its goals
- Similar to Virtools
- Better than DIVERSE for developing interaction techniques
- A "novel idea to separate the interaction from the application and lower level tools"
- Good solution for addressing the problem of plugin communication

One last comment that was interesting to note was that that in comparison to developing applications with tools such as Open Scene Graph, developing with IFFI was better by "leaps and bounds." The rationale behind why this type of comment was made is found in the next section.

The second type of comments this question received were more theoretical than practical. These comments were made because IFFI allows for a higher level of thinking when developing techniques, and it requires less attention to the lower level details. This was actually mentioned by a participant. This shows that IFFI is revolutionary in the way that it approaches the problem of technique development. The next comment in the theoretical evaluation was that IFFI was also structured differently from traditional VR toolkits, however it was not completely foreign. This shows that it is designed to help bridge the gap between traditional VR toolkits and newer 3DIT development tools. It is close enough to existing VR tools to allow people to reuse their existing experience with them, but does not fall into some design faults that they have. Finally, IFFI was described to have a "well thought out structure." This helps IFFI "remove design headaches and reduces the potential code refactoring cycle that happens when new requirements are introduced to a technique prototype." Participants were actively engaged and thinking about IFFI during this evaluation and were excited about what it could do for them.

What are the benefits / downsides of IFFI that you see given this brief exposure to it?
The benefits in this question were that it is a major benefit to be able to break a technique into parts and reuse them. The code for 3DITs produced with IFFI will be cleaner code than produced with traditional VR toolkits. It was also mentioned that reusability should be "a piece of cake", and that flavors of techniques would be easy to create. It was also mentioned that having a developer to support the tool is a large plus for the technique developer. This comment was made because several technique developers also have to maintain and enhance the VR toolkit they are working with, creating even more work for the developer.

The downsides of the approach were similar to the questions raised earlier. The animation question was raised again. The learning curve for the tool was also mentioned. In terms of techniques themselves, one question was how does one break a technique into filters and how does everything fit together with lower level tools. Another question about the implementation was a concern over hiding too much detail from the users. The OpenGL question was also raised. In addition, questions about state management were raised. The state management question could be solved by the incorporation of existing finite state machine literature.

Summary

In general the results from this part were quite positive. The participants were excited to talk about the system and there were several conversations about how it could be used in a wide range of contexts. All of the four different types of reuse were mentioned as a strength for IFFI at one point by participants. However, there were several questions and comments about the approach. Several of these resembled the results from the external evaluation. Some problems that were similar were the learning curve, training time, and relationship with existing VR toolkits. Unlike the VR conference interviews, participants wanted to talk about lower level tools and to get into the details of the system itself. Some of these questions related to animation, how multiple devices worked with the system, and how Open Scene Graph worked with the system.

6.3.3.2 Results for building a technique out of components

All of the participants were able to successfully complete this section of the experiment. All of the participants asked questions regarding the capabilities of IFFI filters, events, and the actual technique itself. Each participant took between ten and thirty minutes to complete this stage. The longer times were due to side conversations that came up. The answers to the specific questions are explained in detail below.

How difficult was it to build this technique?

The answer to this question has two parts.. The overall result is that it was not difficult to build the HOMER technique itself once the problem was well understood. However, the results of the comments relating to learnability and training time to become familiar with state machines became evident during this part of the evaluation. All of the participants thought that the task itself was not difficult once the technique was understood. However, it required time and patience to think about the problem in the right way. One item that all participants asked about was the behavior of the HOMER technique, even if they had worked and implemented it in the past. The next issue that a couple of participants asked about was the functionality of the filters. Overall, participants felt that building the technique would be faster with IFFI, even if they had to build the components themselves.

Another vein of questioning was raised about the filters themselves. The questions asked dealt with information flow, behavior changes, and looping. Several questions about how information was passed to and from different filters were raised. This prompted discussions on

events in IFFI. Questions about how one filter's behavior changed based on other filters were also explained from an event perspective. One question about state machines that was raised is that there is a potential problem with infinite looping that requires design diligence. On the positive end of comments about filter components, it was noted that it not be hard to build usable components. Also, understanding the states required for the technique is usually not difficult due to the inclusion of state based thinking in CS curriculums.

What problems did you run into?

The major problems that participants ran into were the descriptions of the components and technique itself. The major difficulties encountered were understanding how HOMER worked and how to break it down into pieces that reused the existing components. The xml syntax was also the source of several questions. These questions covered conditionals, input, output, and how arguments were given to components. However, it was a good sign that only one participant felt that it was difficult to think about interactions in terms of state machines. This same participant had questions about going from a sequence of actions to state machine transitions. However, there is a silver lining to this list of problems. If graphical representations for the techniques behavior, for the components behavior and how each line of the xml file related to a components behavior were present in the evaluation, most of these questions would have been nullified. This is an important point to take into consideration when designing techniques.

Is the syntax intuitive?

After taking some time getting used to the syntax, participants judged that the syntax was clear, and intuitive. The advantage of xml validation was also brought up. Three negative comments raised were that the input and output could be made clearer, and that colon delimited lists may need to be changed to a more xml centric solution.

What are the benefits / downside of this approach?

The benefits that related to this part of the evaluation were that techniques could be written quickly if the components already existed. A related benefit that was mentioned was that the state of your program could be handled in xml instead of C++. One downside mentioned during this question was that examination of code documentation is required to understand filter functionality. Questions on how to debug filters were also raised. The major downside mentioned was that it will take practice to become proficient at developing techniques in this manner, and getting the correct decomposition granularity will be an issue.

Summary

This section still contained several positive results, but it also showed several possible issues that developers will have when they start using IFFI. Understanding the behavior of a technique is critical to the process of breaking it into components. Graphical representations help immensely with this, as do taxonomies [BOWMAN06]. Also, the nuances of filters and their implementation will have to be emphasized during the development training sessions on IFFI.

However, on a positive note, all of the developers were able to complete this assignment with minimal help in answering questions about the functionality of the technique and components listed. The high level problem of breaking a technique down into components and then combining existing components to create a working technique was not as difficult as understanding the lower level details. The power of being able to reuse techniques was also mentioned. Finally, the comment that developing the technique with IFFI in a reusable manner should be faster than implementing it in a traditional VR toolkit is striking. Typically, creating reusable components is more difficult than by creating an all encompassing application. Obviously, this is at a theoretical level, but it will be interesting to see if this claim can be quantified in the future.

6.3.3.3 Results for building a technique from scratch at a component level and follow-up questions

All of the participants were able to complete this task. There were fewer questions asked in this part of the experiment than the previous experiment. Each participant also took less time on this part of the experiment than on the second part. This may be due to being more familiar with IFFI though. Due to the number of participants, the experiment was not counter-balanced so this claim does not hold statistical significance.

How difficult was this task?

None of the participants felt that taking a technique, identifying the component breakdown, and how these would be implemented in code was difficult. However, several comments on the change of design strategy and thought process were brought up. Because the Go-Go technique was one technique that all of the participants had developed before it was difficult to deviate from the methodology that they had developed during their previous implementation phase. It was stated that learning how to break one technique into many components was not obvious. This process was listed as frustrating by one of the participants due to existing experience with Go-Go.

One positive note about reuse is that all of the participants actually referred back to the previous questions components for ideas about how to create the components for this task. One participant even noticed that the structure of the solution was exactly the same as the previous question except for two minor changes. Participants did not feel that coding the solution would be that difficult. Also, it was stated that when the expert level is reached with this tool, a high level of development performance should be achievable.

Is this a process you normally go through when building a 3DIT?

The process used to build techniques with IFFI was described in several different ways. A list of these descriptions is found below:

- "Not bad"
- Easy to transition technique actions into filters
- Similar to the natural thought process
- Requires getting used to the syntax
- Works best if the technique is thought of as a flow of events through the system

One participant described the normal process they followed for developing techniques was to think mentally through each step, write it down, then use state machines to help determine the number of steps necessary for the technique. Next came the implementation and testing phase. This was related to the process of identifying a technique, breaking it into a components, designing these components for reusability, and refactoring them if possible so the participant came to the conclusion that IFFI does map to the process they normally used. Another participant said that IFFI does map well, but in a slightly different way. The only negative comment listed in this section was that the process is difficult to follow for familiar techniques, but it was noted that for new techniques it will be easier to follow.

What were the benefits / downsides that you see for designing a technique this way?

One of the major benefits of this approach is that one participant actually said they had a hard time finding downsides of the approach. A list of the benefits participants mentioned about designing a technique from scratch by specifying the components, and describing how they fit together can be found below:

- Not hard
- Seeing the result immediately is great
- Intuitive approach
- Provides reusability and flavors
- Easier modification of existing techniques than existing VR toolkits
- Does not force a specific paradigm on you
- Allows for reuse of techniques or parts of techniques
- Web repository of techniques possible for first time

The three downsides mentioned were that the transfer of knowledge between developers may be difficult if code documentation is not "up to snuff". Also, the need for bridges to be built to other toolkits was listed as a minor downside. The last downside was it was unclear how IFFI would integrate into a real application. This was purposefully left out of the evaluation due to the level of detail required and time commitment required.

What is your overall impression of IFFI?
A list of the different impressions that participants though about IFFI can be found below:

- Easy way to build 3DITs
- Allows for developers to think of interaction in an abstract way instead of thinking at a code level
- High level design is clean
- Where do I get it, how does it work with OpenGL
- This tool is different than existing VR tools
- XML file is not that desirable
- C++ configuration important

A problem that was mentioned about IFFI in this section was that a true evaluation by in-depth evaluation by having it used by technique developers, and a library of techniques would be required before it could truly be accepted.

What did you like about IFFI?
All of the participants had already expressed this at earlier stages in the evaluation.
What didn't you like about IFFI?
The only area mentioned was the difference between filters, objects, and worlds. However, it was mentioned that it was not a major problem, it was a concept that would take time to understand.

Is there a marked difference between developing with IFFI than with a traditional VR toolkit?
Participants said that this approach is very structured compared to existing VR toolkits. It was also mentioned that IFFI separates 3DITs from VR toolkits. The major difference listed was that IFFI allows developers to think in boxes instead of LOC. At a lower VR toolkit level, one participant mentioned that the data used in IFFI is much different than the shared memory in DTK.

Do you believe that IFFI makes developing 3DITs easier? Why?
Participants said yes it does, but will need a library of filters. Participants said that IFFI has the potential to make developing techniques easier to implement, but until IFFI is actually used this question cannot be fully answered. The only downside to this question is that the components must be made generic enough for others to use. One participant said yes and it is because IFFI provides structure and motivation for developing techniques.

Does IFFI provide the support you need for developing 3DITs?
All participants said yes. One said that it provides support for 95% of the cases one participant could think of.

What would you change about IFFI?
Surprisingly there were no major additions or deletions requested from IFFI. This will probably change after IFFI is used.

Does IFFI match your design process for building 3DITs?
All of the participants said yes to this question. One participant did not care for the xml syntax, but the C++ interface matched their design process.

Would you use it in the future?
All participants said they would consider using IFFI in the future.

Summary

More positive than negative comments were made during this part of the evaluation. The participants thought that it was possible to take the description of a technique, break it into different parts, and specify how these parts fit together. Lower level details such as the toolkit and syntax were listed, but could probably be remedied with more training. A couple of discussions on how to integrate IFFI into a VR toolkit were also brought up. There was a strong desire to actually obtain the source for the tool by several participants. This was because participants believed that IFFI can make peoples job easier. One major contribution is that IFFI allows a developer to think at a much higher level when developing techniques instead of requiring technique developers to also be a code designer. Overall, participants said it was easy to see how IFFI worked.

6.3.3.4 Summary of in-depth evaluation results

In the in-depth evaluation several important benefits and problems with IFFI were mentioned. The major problems with the approach are that it will require time and effort. Learnability of the tool itself may be an issue if previous history applies. Current technique developers may have an issue developing techniques with IFFI that they have previously developed.

However, the positive responses greatly outweighed the negative comments. The four types of reuse targeted are listed as positives. In addition to this structuring, the development of techniques and providing support at a code level for building techniques help reduce the difficulty of implementing techniques. Perhaps the most important result from this internal evaluation is that it encourages technique developers to abstract away the code level and focus on the component level and how these components fit together.

6.4 Summary of results

Through the three different methods of evaluation we have shown that IFFI can produce reusable techniques, can be used to implement past and previous research, and is one solution to the problem of technique reuse in VR. By implementing multiple techniques we demonstrated that techniques built with IFFI fulfill the four types of reuse that this research seeks to provide. A conceptual evaluation showed that this research is of value, novel, and addresses the needs for both techniques and developers. It also points out that learnability of the system, design of the components, and complexity of reusable components are potential downsides for this approach. In-depth evaluations show that it is possible for others to build techniques using the process we developed, that it is feasible for technique developers to create techniques with this framework, and that we have produced an intuitive software solution. When all of these results are combined it is clear that IFFI can produce techniques that fit the description of reuse, and that others can use the process to develop techniques, and that the outside research community agrees with this research direction.

Chapter 7: Conclusions, Contributions, and Future Work

7.1 Conclusions

This work has focused on the development of reusable 3DITs. This work has allowed for techniques to be created that work with multiple VR toolkits for the first time. In chapter one the problems facing 3DIT development have been presented. This has shown that 3DIT development is rife with difficulty and reuse is seldom considered. In chapter two a literature review was presented that described the different approaches used to build 3DITs. The three major research directions were investigated in chapter two are VR toolkits, specification tools, and higher level 3DIT tools. The literature review showed that there is little existence of reuse in the field of VR, and has not been taken into serious consideration by the toolkit creator community. After the literature review a community survey was done in order to verify the results from the literature review. In chapter three the work necessary to lay the foundation for creating reusable techniques was presented. The existing VR tools were not feasible to develop reusable components with, and it was necessary to switch to tools that the wider community uses. In chapter four the requirements and design for the tool to create reusable techniques were presented. This tool produces techniques that can be used between applications written with the same VR toolkit, techniques that work between applications written in different VR toolkits, allows for component level reusability of techniques, and allows for parts of one technique to be reused in other techniques. In chapter five three sample techniques that demonstrated the four types of reusability were implemented; a simple process for developing reusable techniques was presented. The two VR toolkits used to demonstrate between toolkit reuse were CoVE and DGL. In chapter six the evaluation of IFFI was described. An outside evaluation and a more in-depth internal evaluation were performed to verify that techniques developed with IFFI can fulfill the definition of reusability.

7.2 Research questions and results revisited

The basic research question is that techniques have been built for over a decade so why do we not have a set of reusable canonical techniques available with existing tools? How do you prove that the tool created to produce reusable techniques can produce them, and that people can build techniques with the tool? These questions were crystallized into the following three research questions.

- **Question 1:** Why aren't current tools sufficient for the easy development of reusable 3DITs?

The first research question was answered by three components. The first was a literature review (chapter 2), the second was a community survey (chapter 2), and the third was our personal experience and review of existing work (chapter 1). The literature review detailed tools and design approaches for three different areas. The first are typical VR toolkits, the second are specification level approaches, and the third is 3DIT specific tools. All of these approaches showed that these tools were not designed to produce truly reusable techniques. The community survey verified the results from the literature review, demonstrated that the outside research community agreed that reuse was a problem, the current state of reuse had not progressed beyond copying and pasting, and there are no good solutions available for reuse. Our personal experience agreed with the community survey and literature review.

- **Question 2:** What would the requirements be for a tool that does support the creation of reusable and generic 3DITs? How do you design a tool that fulfills the requirements gathered in approach 1?

The second question was answered by an in-depth study of publications, artifacts, and personal experience. Through requirements analysis we were able to identify three critical areas that all techniques share. These are device input, scene-graph parent/child relationships, and position and rotation. By using smart abstraction we are able to create a simple set of abstractions that can map between multiple graphical and VR toolkits. In addition to this the best practices of previous VR toolkits, specification tools, and 3DIT tools have been synthesized into one coherent tool designed to help build reusable techniques. This resulted in the worlds and objects that can be found in the fourth chapter. The full explanation of the requirements and design can be found in chapter four. Foundational work was also necessary in order produce the toolkit for building reusable techniques. This was because VR toolkits at the beginning of this work were not designed for flexibility and reusability.

- Question 3: Can we demonstrate that the tool developed in answer to question 2 allows for reusable techniques to be created?

The third question was answered in three different ways. First three sample techniques were implemented that demonstrated the four types of reusability that we sought to provide with IFFI. The second form of evaluation was to have IFFI evaluated by outside experts in order to prove that it was novel research, was addressing the problem of reusability, and that our approach could theoretically provide the type of reuse we are aiming to provide. The third form of evaluation was a more in-depth internal evaluation done by experts. This evaluation showed that

the code in IFFI was usable for creating techniques, that it was possible to build techniques from existing components, and that it was possible to design techniques from scratch in a reusable component based manner. The details for these evaluations can be found in chapters five and six.

7.3 Contributions

This work provides several theoretical and practical contributions that are listed below. The contributions range from changes that could revolutionize the field of VR, to several lower level technical contributions that have improved the productivity of dozens of developers here at Virginia Tech.

7.3.1 Theoretical contributions

There are four theoretical contributions that this work has made. They cover providing the reusability of techniques, to validation of previous work, design steps for tools, and a process for developing reusable techniques.

7.3.1.1 True reusability of interaction techniques

The centerpiece contribution for this work is that for the first time interaction techniques can be shared between different institutions without requiring the reimplementation of the techniques from scratch. This is possible because IFFI is not tied to a specific VR toolkit and can work with multiple techniques. By solidifying the theory behind how to create reusable techniques from both a technique designer and technique implementer perspective it should be easier to develop techniques in the future. No longer will the better part of a semester be required to implement an existing technique in order to evaluate a new technique against it. This also allows for techniques to be distributed with VR toolkits for the first time. Until now the only techniques that were normally distributed with VR toolkits were navigation techniques. This will also significantly lower the bar for commercial companies to adopt academic research and may help increase the adoption rate of VR.

7.3.1.2 Validation of previous research

The second major contribution of this work is that we have validated previous 3DIT tool research. The specification and 3DIT specific tools have made an important contribution to the field of VR research because they laid the foundation for making techniques reusable without

writing code. These two research directions made an important contribution by introducing the data flow architecture and fully specifying the finite state machine paradigm; explaining the role these two concepts exhibit in implementing techniques instead of just relying on an update callback. By integrating all of these areas we were able to create a tool that combines the best practices and provides the revolutionary ability to create reusable interaction techniques between toolkits.

7.3.1.3 A revolutionary design step for VR tools

The third major contribution that this work made was the revolutionary design step of moving away from VR toolkits when developing 3DITs. By removing the dependence that specification level and 3DIT specific tools have on VR toolkits, a new way of developing becomes possible. This allows for technique developers to focus on their work, not on how to manage the details of integrating the technique, application, VR toolkit, and graphical toolkit together. By providing a structured pathway that developers can choose to use, IFFI allows for a revolutionary design step.

7.3.1.4 A process for developing reusable techniques

The last theoretical contribution is a simple process for developing reusable techniques. While this process is not heavily tested, it is a start for developing reusable techniques. This process was used in all three types of evaluation and was able to demonstrate a simple manner for developing reusable techniques.

7.3.2 Practical contributions

There are five major practical contributions that this work has produced. These range from the tool to the methods used to build the tool and the techniques built with the tool.

7.3.2.1 IFFI

The first practical contribution is the architecture of IFFI itself. Considerable work was done during the requirements and design stage of the tool that has produced a revolutionary architecture for building 3DITs. This architecture is what allows for techniques to be used in a reusable manner. The architecture itself may be useful in areas other than 3DIT creation, only time will tell. IFFI can be downloaded at iffi.sourceforge.net

7.3.2.2 Requirements analysis for techniques

The second practical contribution that this work has produced is requirements generation and analysis for 3D interaction techniques. For the first time, requirements analysis of actual 3DITs has been performed. This allowed IFFI to be able to obtain the level of reuse that it currently provides. This information may be valuable to future 3DIT developers as a way to help generalize a specific portion of their work.

7.3.2.3 Example techniques

The third practical contribution of this work is that example techniques have been produced and can be distributed to work with any VR application. Normally VR tools do not provide interaction techniques. Even rarer are VR tools that provide techniques that can be reused among applications written with the VR toolkit. IFFI provides three canonical techniques that can be used with any VR toolkit or application if the proper software bridge exists.

7.3.2.4 Open source tools

The fourth practical contribution of this work is the open source tools (IFFI, DGL, and DADS) that have been produced. By making this work open and free for anyone to use, it is possible that this work may be reused by anyone. Several tools in this research area have been hampered by closed source licenses, and this work avoids these problems and allows others to freely expand on this work.

7.3.2.5 Impact of foundational work on institutional research

The last practical contribution provided is the usage of DGL / DADS to power the VR facilities here at VT. These two programs that made up the foundational work have been used by dozens of developers over the years at Tech. Without these components other research would have been greatly slowed and less mature software would have been used to try and fill that gap that existed before this work started. This would have greatly increased the already long development time for VR applications here at VT. Because of this work four masters theses, an award winning undergraduate research project, and dozens of class projects were able to focus on their research goals instead of providing a foundation on which their research could be completed.

7.4 Future work

There are several major areas for future work with IFFI. The major goal for the future work is to continue lowering the barrier to producing reusable techniques. As with the contributions there are two avenues for future work: theoretical and practical.

7.4.1 Theoretical

Now that techniques are truly reusable, there is a completely new area of research that is possible. This is to investigate the types of reuse between different techniques. By allowing reusable techniques to be built from components, if a library of techniques exists several types of research can happen.

7.4.1.1 Granularity

The granularity of components for reusable techniques is one major area of future research. For example, existing 3DIT tools have broken down ray-casting into hundreds of reusable components whereas this work broke ray-casting into three simple components. Is it necessary to break techniques down into this many components? What are the advantages of breaking techniques down this far? How much time is saved in the short run for lower granularity choices, and what impact does this have in the long run for techniques?

7.4.1.2 Generalization of how to build techniques

By allowing techniques to be built in a reusable manner new ways of developing techniques may be possible. The related work section shows that there was a significant shift between the design and implementation of techniques between VR toolkits and the data-flow architecture approach. It may be possible for another new way to build interaction techniques to be possible if this work is used as a foundation. One example of this is that the requirements analysis of this work in chapter four shows that there are new ways to look at techniques that have not previously been done before. By allowing for techniques to be built new ways generalizing techniques may be possible that will help technique designers think about their work in a new way.

7.4.1.3 Technique generation

The last theoretical area possible is technique generation. Currently techniques are built by using VR toolkits and developers working at a low level code level. By moving to a reuse based paradigm it may be possible to change the way that techniques are generated. It may be possible

for a true interaction technique prototype generator to be built given enough techniques. This may allow for techniques to be generated in a completely different manner.

7.4.2 Practical

The other areas for future work are in the practical realm. These are areas that will greatly enhance the usability of IFFI, but may not be significant research contributions by themselves. The major areas for practical future work consist of supporting software that needs to be built around IFFI.

7.4.2.1 A GUI for building techniques

Currently the xml file for building techniques has to be generated and modified by hand. This is not difficult, and standard xml editors work with the IFFI xml structure. However, a coherent 2D environment that contains technique components and a graphical manner to generating these techniques will be extremely useful for developing techniques from existing components. This will allow for non-expert developers to build techniques or import existing techniques into their environments. It will also provide as a way to manage all of the components built using IFFI.

7.4.2.2 More techniques

This works purpose is to allow others to build reusable techniques. The obvious next step would be build techniques using this tool. This is one of the next steps towards reducing technique creation time. By building more and more techniques more components will be available for others to use.

7.4.2.3 Web repository for techniques

Another avenue for future work would be the implementation of an online database of techniques. The creation of standard metadata for techniques would be necessary for search functionality. Also a rating system and user ranking (novice, published, expert) would have to be taken into account. The creation and use of this repository would change the face of technique development. Instead of being locked behind access to academic publications, techniques will be available to the public. Not just theoretically, but in actual working form for the first time.

7.5 Summary

This work has motivated the problem of reusable techniques, proven that they are a problem, proposed and provided a solution for creating reusable techniques, and evaluated this approach. It has shown that it is feasible to create reusable techniques, and they can reduce the amount of effort required to implement future techniques. There are numerous contributions that this work provides, but reuse of techniques is the primary mantra that this work has stressed since chapter one. There is considerable future work left before reusable techniques become part of the everyday life of a VR developer. If current history repeats itself, this work may only be adopted by a handful of developers, but at least a theoretical and practical method for creating reusable techniques exists that has been tested and approved by a wide range of researchers. The follow-up research to this work will focus on creating reusable techniques and providing guidelines for how to develop these reusable techniques. This may change the face of VR due to the helping reduce the massive time investment required to create 3DITs for VEs.

Bibliography

[ALLARD01] Allard, J., Gouranton, L. L. V., Melin, E., and Raffin, B. *Net juggler guide*. Tech. Rep. RR-LIFO-2001-02, LIFO, Orlans, France, June 2001.

[AVANGO99] "Avango Homepage," http://imk.gmd.de/docs/ww/ve/projects/proj1_2.mhtml (current Oct 21, 1999).

[BALAKRISHNANR01] Balakrishnan, R., Fitzmaurice, G.W., Kurtenbach, G., "User interfaces for volumetric displays", IEEE Computer, March 2001, p. 37-45

[BELLEMANR03] Belleman, R. Interactive Exploration in Virtual Environments. PhD thesis, Universiteit van Amsterdam, 2003.

[BIERBAUMA98] Bierbaum, A. and Just, C. (1998). *Software Tools for Virtual Reality Application Development*, SIGGRAPH 98 Course 14 Applied Virtual Reality Notes, ACM SIGGRAPH 1998, Orlando, Florida, USA, 1998, 44p.

[BIERBAUMA00] A. Bierbaum, C. Just, P. Hartling, and C. Cruz-Neira, "Flexible application design using VR Juggler". Technical sketch presented at SIGGR,4PH 2000, New Orleans, July 2000.

[BIERBAUMA01] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "VR Juggler: A Virtual Platform for Virtual Reality Application Development" IEEE VR. March 2001

[BOEHMB88] B. Boehm: A Spiral Model of Software Development and Enhancement. IEEE Computer 21(5): 61-72 (1988)

[BOEHMB96] B. Boehm, S. Wolf, "An open architecture for software process asset reuse," *ispw*, p. 2, 10th International Software Process Workshop (ISPW '96), 1996

[BOURKEP00] Multiple Wall Projection by Paul Bourke. http://astronomy.swin.edu.au/~pbourke/projection/3walls/ Visited in July 2006

[BOYDD99] Boyd D. and Sastry L. Development of the INQUISITVE interaction toolkit – Concept and realisation. Workshop on User Centered Design and Implementation of Virtual Environments pp 1-6 1999 The University of York.

[BOWMAND97a] Bowman, D. and Hodges, L. An Evaluation of Techniques for Grabbing and Manipulating Remote Objects in Immersive Virtual Environments. *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, 1997, pp. 35-38.

[BOWMAN97b] Bowman, D. and Hodges, L. F.(1997b). Toolsets for the Development of Highly Interactive and Information-Rich Virtual Environments. International Journal of Virtual Reality, 3, 2, 1997, pp. 12-20.

[BOWMAND99] Bowman D. and Hodges, L. Formalizing the Design, Evaluation, and Application of Interaction Techniques for Immersive Virtual Environments. *The Journal of Visual Languages and Computing*, vol. 10, no. 1, February 1999, pp. 37-53.

[BOWMAND01] Bowman, D., Kruijff, E., LaViola, J., and Poupyrev, I. An Introduction to 3D User Interface Design. *Presence: Teleoperators and Virtual Environments*, vol. 10, no. 1, 2001, pp. 96-108.

[BOWMAN04] Bowman, D., Kruijff, E., LaViola, J., and Poupyrev, I. *3D User Interfaces: Theory and Practice*. Addison-Wesley, Boston, 2004.

[BROLLW95] Broll, W. (1995). Interaction and Behavior Support for Multi-User Virtual Environments. In Proceedings of *First Workshop on Simulation and Interaction in Virtual Environments, SIVE95*, (pp.256-263), Iowa City, IA: University of Iowa Computer Science Department.

[CARLSSONC93] Carlsson C., Hagsan, O.: *DIVE A platform for multi-user virtual environments*. Computers and Graphics, 17 1993.

[CONNER92] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-dimensional widgets. Computer Graphics (1992 Symposium on Interactive 3D Graphics), 25(2):183--188, March 1992

[CONWAYA00] Conway, Audia, Burnette, Cosgrove, and others, "Alice:Lessons Learned from Building a 3D System for Novices",ACM CHI 2000 Papers, pp 486-493.

[COOMANSM97] Coomans, M.K.D. and H.J.P. Timmermanns. Towards a Taxonomy of Virtual Reality User Interfaces. in International Conference on Information Visualisation (IV97). 1997. London.

[COVE07] CoVE Website. http://www.opentechinc.com Visited in October 2007.

[CRUZC93] Carolina, C, Sandin, D. J., DeFanti, T. A., (1993), "*Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE*". Proc. SIGGRAPH '93, pp. 135-142.

[CRUZ0C2] Carolina Cruz-Neira, Allen Bierbaum, Patrick Hartling, Christopher Just, and Kevin Meinert. VR Juggler -- An Open Source Platform for Virtual Reality Applications. In 40th AIAA Aerospace Sciences Meeting and Exhibit 2002, Reno, Nevada, January 2002.

[DACHSELTR99] Dachselt R. The challenge to build flexible user interface components for non-immersive 3D environments. Proceedings of 8 International Conference on

HumanComputer Interaction. Lawrence Erlbaum Associates. Part vol.2, 1999, vol.2. Mahwah, NJ, USA, pp.1055-1059.

[DACHSELTR02] Dachselt, R.; Hinz, M.; Meiner, K.. CONTIGRA: An XML-Based Architecture for Component-Oriented 3D Applications. Proceedings of the ACM Web3D 2002 Symposium, Tempe (USA), 24.-28.

[DRAKENR95] R. Darken, C. Tonnesen, K. Passarella. The Bridge Between Developers and Virtual Environments: a Robust Virtual Environment System Architecture. Proceedings of SPIE 1995

[DELIGIANNIDIS00] L. Deligiannidis, "DLoVe: A specification paradigm for designing distributed VR applications for single or multiple users" Doctoral dissertation, Tufts University, Feb. 2000

[DIAZR89] Prieto-Diaz and J.M. Neighbors, "*Module Interconnection Languages*", J. Systems and Software, vol. 6, no. 4, pp. 307-334,Oct. 1989.

[DYKSTRAP94] Dykstra, P. (1994). "X11 in Virtual Environments: Combining Computer Interaction Methodologies." j-X-RESOURCE **9**(1): 195-204.

[ELLIOTTC94] Conal Elliott, Greg Schechter, Ricky Yeung and Salim Abi-Ezzi [July 1994], "TBAG: a High Level Framework for Interactive, Animated 3D Graphics Applications", in Andrew Glassner, editor, Proceedings of SIGGRAPH `94 (Orlando, Florida), pages 421-434.

[ENDEDEG86] Endede, G., Kansy, IC, and Pfaff, G., Computer Graphics Programmin~ GKS-The Graphics Standard, Second Edition, Springer-Verlag, New York, 1986.

[FALKOK01] Falko Kuester, Bernd Hamann, and Kenneth I. Joy, "Virtualexplorer: A plugin-based virtual reality framework", in Proceedings of SPIE, R.F. Erbacher, P.C. Chen, M. Groehn, J.C. Roberts, and C.M. Wittenbrink, Eds., San Jose, California, USA, 2001, SPIE

[FIGUEORAP01] FIGUEROA, P., GREEN, M. A Framework for 3D Interaction Techniques In: CAD/Graphics'2001, 2001 **CAD/Graphics'2001**. , 2001.

[FIGUEROAP02] Figueroa, P.; Green, M.; Hoover, H. J. InTml: A Description Language for VR Applications. Web3D'02, February 24-28, 2002. Tempe, Arizona, USA.

[FELSS98] Sidney Fels and Kenji Mase. InvenTcl: A fast prototyping environment for 3D graphics and multimedia applications. In IEEE AMCP, pages 163--178, Japan, November 1998.

[FRAKESF98] Frakes F. Prieto-Diaz R. Fox C. DARE: Domain analysis and reuse environment. Annals of Software Engineering. Volume 5 Issue 0. 1998. pp. 125-141

[FRAKESB94] Frakes, William B. and Pole, Thomas, "An Empirical Study of Representation Methods for Reusable Software Components", *IEEE Transactions on Software Engineering,* V20 n8, pp. 617-630, 1994.

[GOBBETTIE93] Enrico Gobbetti, Jean-Francis Balaguer, and Daniel Thalmann. VB2: An Architecture for Interaction in Synthetic Worlds. In Proceedings of the ACM Symposium on User Interface Software and Technology, 167-178, November, 1993.

[GOBBETTIE99] Gobbetti, E., and Scateni, R., 1999, "Virtual Reality: Past Present and Future," Sardinia, Italy: Center for Advance Studies, Research and Development.

[GREENM1] Mark Green. MRToolkit home page. http://www.cs.ualberta.ca/~graphics/MRToolkit/

[GRIEPPT02] Griepp T. and Cruz-Neira C. XJL an XML Schema for the Rapid Development of Advanced Synthetic Environments. Proceedings of IPT 2002.

[HOLLENBACHC96] Hollenbach, C. and W. Frakes. "Software Process Reuse in an Industrial Setting". *Proceedings of Fourth International Conference on Software Reuse*. 1996. Orlando, FL: IEEE CS Press, pp. 22-30.

[HOLLERER99] Hollerer, T., Feiner, S., Terauchi, T., Rashid, G., and and D. Hallaway (1999). "Exploring MARS: Developing Indoor and Outdoor User Interfaces to a Mobile Augmented Reality System." Computers and Graphics **23**(6): 779-785.

[HOWARDT91] T. Howard, W. Hewitt, R. Hubbold, and K. Wyrwas, *A Practical Introduction to PHIGS and PHIGS PLUS*. Workingham -- Reading: Addison--Wesley, 1991.

[HUMPHREYS02] G. Humphreys, M. Houston, Y.-R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. T. Klosowski, "Chromium: A Stream Processing Framework for Interactive Graphics on Clusters," presented at SIGGRAPH, San Antonio, Texas, 2002.

[KANGK02] Kang, K.C.; Jaejoon Lee; Donohoe, P.; Software, IEEE Volume 19, Issue 4, July-Aug. 2002 Page(s):58 - 65

[KELLNERM96] M.I. Kellner, "Connecting reusable software process elements and components," *ispw*, p. 12, 10th International Software Process Workshop (ISPW '96), 1996

[KELSOJ02] J.Kelso, L.E.Arsenault, S.G.Satterfield, R.D.Kriz, DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments, Proceedings of IEEE Virtual Reality 2002 Conference

[KESSLERG99] Kessler, G. Drew. A Framework for Interactors in Immersive Virtual Environments. IEEE VR '99, Houston, TX, Mar. 1999. pp 190-7.

[KESSLERG00] Kessler, G., Bowman, D., and Hodges, L. The Simple Virtual Environment Library: An Extensible Framework for Building VE Applications. *Presence: Teleoperators and Virtual Environments*, vol. 9, no. 2, 2000, pp. 187-208.

[KIRWINB92] Kirwan B, Ainsworth, LK. A Guide to Task Analysis. London: Taylor & Francis; 1992.

[KOOPERR98] Rob Kooper and Brian Wills and Kevin Hamilton and Don Allison and Larry F Hodges COOL-VR: a Virtual Environments Toolkit (1998). http://citeseer.ist.psu.edu/370959.html

[KOVEDL93] L. Koved, W. Wooten, GROOP: An Object-Oriented Toolkit for Animated 3D Graphics, ACM SIGPLAN NOTICES OOPSLA'93, Vol. 28, No. 10, October 1993, pp. 309--325.

[KRIZR2003a] Kriz R.D., Farkas D., Ray A.A, and Kelso J.T. , "Visualization of Structure-Property Relationships: Spanning the Length Scales from Nano to Macro," Invited Paper, International Conference on Computational and Experimental Engineering Science, Corfu, Greece (July, 2003): 25-29

[KRIZR2003b] Kriz R.D., Farkas D., Ray A.A, Kelso J.T, and Flanery R.E., Jr., "Visual Interpretation and Analysis of HPC Nanostructure Models using Shared Virtual Environments," Conference Proceedings, High Performance Computing: Grand Challenges in Computer Simulations 2003, The Society for Modeling and Simulation International (SCS), San Diego, California (2003): 127-135

[LAVIOLAJ01] LaViola, Joseph, Daniel Acevedo, Daniel Keefe, and Robert Zeleznik. Hands-Free Multi-Scale Navigation in Virtual Environments. In the Proceedings of the 2001 Symposium on Interactive 3D Graphics,ACM Press, 9-15, 2001.

[LIGHTNING00] "Lightning Homepage”, http://vr.iao.fhg.de/vr/projects/Lightning/OVERVIEWen. html (current May 1, 2000).

[LINDEMANR01] Lindeman R.W. Siber. J.L, Templeman James N. The Effect of 3D Widget Representation and Simulated Surface Constraints on Interaction in Virtual Environments. Proc. of IEEE VR. 2001 pp.141-148

[LINDEMANR01] Lindeman R.W. **Making VR More Usable: The State of Effectiveness in Virtual Reality** *Journal of the Virtual Reality Society of Japan*, 11(1), 2006, pp. 24-27.

[MEDIVIDOVICN00] N. Medvidovic and R. N. Taylor, "*A Classification and Comparison Framework for Software Architecture Description Languages*," *Journal*, IEEE Transactions on Software Engineering, vol. 26, pp. 70-93. No. 1, Jan, 2000.

[MERNIKM01] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Multiple Attribute Grammar Inheritance. Informatica, 24(3):319--328, September 2000. 01] M. Mernik, U. Novak, E. Avdicausevic, M. Lenic, and V. Zumer. Design and implementation of simple object description language. In ACM Symposium on Applied Computing, SAC'2001, pages 590--594, 2001.

[MANTEREM01] Markku Mantere. **Visualization of Flow Data in Photo-realistic Virtual Environment (2001).** http://eve.hut.fi/Publications/visualization_2001.pdf

[MINEM95] Mine, M. (1995). Virtual Environment Interaction Techniques, UNC Chapel Hill Computer Science Technical Report, TR95-018.

[MULLER03] Mueller, A., S. Conrad, et al. (2003). Multifaceted Interaction with a Virtual Engineering
Environment using a Scenegraph--oriented Approach. WSCG, Plzen.

[MYERSB92] Brad A. Myers and Mary Beth Rosson. "Survey on User Interface Programming," *Proceedings SIGCHI'92: Human Factors in Computing Systems*. Monterrey, CA, May 3-7, 1992. 195-202.

[MYERSB97] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. "The Amulet Environment: New Models for Effective User Interface Software Development", *IEEE Transactions on Software Engineering*, Vol. 23, no. 6. June, 1997. pp. 347-365.

[MYERSB98] Brad A. Myers. "A Brief History of Human Computer Interaction Technology." *ACM interactions*. Vol. 5, no. 2, March, 1998. pp. 44-54

[NICOLCJ81] Nicol, C. J. and Kilgour, A. C. 1981. A Pascal implementation of the GSPC Core Graphics package. *SIGGRAPH Comput. Graph.* 15, 4 (Dec. 1981), 327-335.

[OLSEN] Eric Olsen. Cluster Juggler – PC Cluster Virtual Reality. Unpublished masters thesis, University of Iowa State

[OLWAL04] Olwal, A. and Feiner, S. 2004. Unit: modular development of distributed interaction techniques for highly interactive user interfaces. In Proceedings of the 2nd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia (Singapore, June 15 - 18, 2004). S. N. Spencer, Ed. GRAPHITE '04. ACM Press, New York, NY, 131-138.

[OPENAL06] OpenAL Website. http://www.openal.org/ Visited in August 2006.

[OSG06] Open Scene Graph Website http://openscenegraph.org Visited in July 2006.

[PAUSCHR95] Pausch, R., et al. Alice: A rapid prototyping system for 3D graphics. IEE Computer Graphics and Applications, Vol. 15, No. 3, pp 8-11, 1995.

[PIEKARSKIW03a] Piekarski, W. and Thomas, B. H. An Object-Oriented Software Architecture for
3D Mixed Reality Applications. In 2nd Int'l Symposium on Mixed and Augmented Reality, Tokyo, Japan, Oct 2003.

[PIEKARSKIW03a] Piekarski, W. and Thomas, B. H. Interactive Augmented Reality Techniques for Construction at a Distance of 3D Geometry. In 7th Int'l Workshop on Immersive Projection Technology / 9th Eurographics Workshop on Virtual Environments, Zurich, Switzerland, May 2003.

[PIERCEJ99] Pierce, J., Stearns, B., and Pausch, R. *Voodoo Dolls: Seamless Interaction at Multiple Scales in Virtual Environments*. Proceedings of the 1999 Symposium on Interactive 3D Graphics, pages 141-145.

[PONDERM03] M. Ponder, G. Papagiannakis, T. Molet, N. Magnenat-Thalmann and D. Thalmann, VHD++ Development Framework: Towards Extendible, Component Based VR/AR Simulation Engine Featuring Advanced Virtual Character Technologies, IEEE Virtual Reality, 2003

[PAPED96] D. Pape, "A Hardware-Independent Virtual Reality Development System," in IEEE Computer Graphics and Applications, Jul. 1996, vol. 16, no. 4, pp. 44-47.

[POLYSN04] Polys, N., North, C., Bowman, D., Ray, A., Moldenhauer, M., Dandekar, C. (2004a). Snap2Diverse: Coordinating Information Visualizations and Virtual Environments. SPIE Conference on Visualization and Data Analysis (VDA), San Jose, CA.

[POUPREYVI96] Poupyrev, I., Billinghurst, M., Weghorst, S., & Ichikawa, T. (1996). The Go-Go interaction technique: Non-linear mapping for direct manipulation in VR. *Proceedings of the ACM Symposium on User Interface Software and Technology,* 79 –80.

[POUPREYVI97] Poupyrev, I., *et al*., **A framework and testbed for studying manipulation techniques for immersive VR**. ACM VRST'97. 1997 pp. 21-28

[RAYA07] Ray, A. and Bowman, D. A. 2007. Towards a system for reusable 3D interaction techniques. In *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology* (Newport Beach, California, November 05 - 07, 2007). S. N. Spencer, Ed. VRST '07. ACM, New York, NY, 187-190

[ROBERTD93] Robert DeLine. Master's Thesis. *Alice: A Rapid Prototyping System for Three-Dimensional Interactive Graphical Environments*. University of Virginia,May, 1993.

[RORKEM1] Designing and Implementing a Virtual Reality Interaction Framework. Masters Thesis Rhodes University.

[SCHAFFER03] Benjamin Schaeffer, Camille Goudeseune, "Syzygy: Native PC Cluster VR," *vr*, p. 15, IEEE Virtual Reality Conference 2003 (VR'03), 2003

[SEGALM98] Segal, M. & Akeley, K. (1998), *The OpenGL Graphics System: a Specification* (Version 1.2).

[SENSE81] "WorldToolKit Release 8: Technical Overview," www.sense8.com (current June 3, 2000).

[SENSEABLE1] Sense8 Haptics toolkit homepage http://www.sensable.com/products/phantom_ghost/OpenHapticsToolkit-intro.asp (visited August 2006).

[SHERMANB1] Bill Sherman. The Free VR toolkit. http://www.freevr.org/

[SHAWC93] Shaw, C., Green. M., Liang, J. and Sun, Y. Decoupled simulation in virtual reality with the MR toolkit. ACM trans. on Information Systems. 11(3), pp. 287-317 1993

[SMITHS00] Smith S.Sp and Duke D.J. Binding Virtual Environments to Toolkit Capabilities. Computer Graphics Forum 2000 Volume 19 #3 pp. C81-C89.

[STOAKLEYR95] Stoakley, R., Conway, M., Pausch, R. Virtual reality on a WIM: interactive worlds in miniature. In Proc. CHI' 95,1995, pp. 265-272

[SUTCLIFFEA00] Sutcliffe, A.G. The Domain Theory: patterns for knowledge and software reuse. Lawrence Erlbaum Associates: Hillsdale NJ. 2000.

[TANRIVERDVI01] V. Tanriverdi and R.J.K. Jacob, "VRID: A Design Model and Methodology for Developing Virtual Reality Interfaces," *Proc. ACM VRST 2001 Symposium on Virtual Reality Software and Technology*, ACM Press, Banff, Canada, 2001

[TAYLORR01] ll Russell M. Taylor, Thomas C. Hudson, Adam Seeger, Hans Weber, Jerrey Juliano, and Aron T. Helser, "VRPN: a device-independent, network-transparent vr peripheral system," in Proceedings of the ACM symposium on Virtual reality software and technology. 2001, pp. 55--61, ACM Press.

[TRAMBEREND99] H. Tramberend. “Avocado: A Distributed Virtual Reality Framework.” IEEE Virtual Reality 1999:14-21.

[VANACKENL07] Vanacken, L., Raymaekers, C., and Coninx, K. 2007. Introducing semantic information during conceptual modelling of interaction for virtual environments. In *Proceedings of the 2007 Workshop on Multimodal interfaces in Semantic interaction* (Nagoya, Japan, November 15 - 15, 2007). WMISI '07. ACM, New York, NY, 17-24.

[VANACKEND06] Vanacken D., Boeck J.D., Raymaekers C., and Coninx K. 2006. *NiMMiT: a Notation for Modelling Multimodal Interaction Techniques*. Proceedings of International Conference on Computer Graphics Theory and Applications, 2006, pp. 224-231.

[VEGA1] "Vega: The Comprehensive Software Environment for Realtime Application Development", http://www.multigen.com/support/dc_files/Vega_brochure.pdf, Visited 10-01-01.

[VRCO1] VRCO Website, Home of the CaveLIBS," www.vrco.com (current June 20 2006)

[VTK06] VTK Website http://vtk.org Visited in July 2006.

[WATSENK98] Watsen, K. and M. Zyda. Bamboo - A Portable System for Dynamically Extensible, Real-Time, Networked, Virtual Environments. in IEEE Virtual Reality Annual International Symposium. 1998, 252-259. Atlanta, Georgia

[WILLANSJ00a] Willans J.S. and Harrison M.D. A 'plug and play' approach to testing virtual environment interaction techniques. Virtual Environments 2000 pp. 33-42 . Springer Computer Science, NY.

[WILLANSJ00b] Willans J.S. and Harrison M.D. A toolset supported approach for designing and testing virtual environment interaction techniques. Proceedings of the 6th Eurographics Workshop on Virtual Environments. 2000 Volume 55 #2 pp. 145-165.

[WILLANSJ00c] Willans J.S., Harrison M.D. Smith S.Sp. Implementing virtual environment object behaviour from a specification. User Guidance in Virtual Environments: Proceedings of the Workshop on Guiding Users through Interactive Experiences – Usability Centered Design and Evaluation of Virtual 3D Environments. 2001 pp 87-91

[WINGRAVEC05] Wingrave, C. & Bowman, D. (2005) CHASM: Bridging Description and Implementation of 3D Interfaces. New Directions in 3D User Interfaces Workshop in IEEE Virtual Reality.

[WINGRAVEC06] Wingrave, C., Haciahmetoglu, Y. & Bowman, D. "Overcoming World In Miniature Limitations by Scaling and Scrolling" IEEE 3DUI Symposium 2006 pp. 11-16.

[WLOKAM95a] WLOKA, M. Interacting with virtual reality. In Virtual Environments and Product Development Processes (1995), Chapman and Hall.

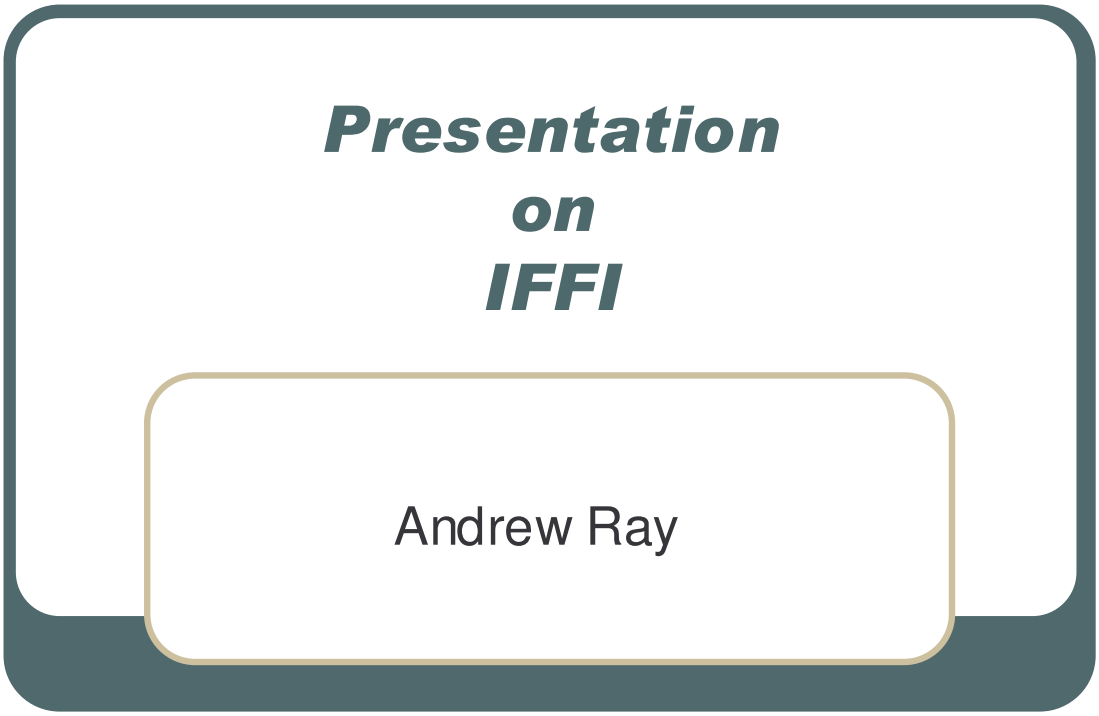
[WLOKAM95b] Wloka, M., Greenfield, E., "The Virtual Tricorder: A Uniform Interface for Virtual Reality," Proc. of UIST '95, (1995), pp. 39-40

[ZELEZNIKR91] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. Computer Graphics (SIGGRAPH '91 Proceedings), 25(4):105--112, July 1991.

Appendix A: Evaluation experiment materials

This appendix is made up of two parts. The first part is the presentation given during the evaluation and the second part is the example filter used in the evaluation.

A.1 Evaluation presentation



Experiment flow

- 15-30 minute presentation
- Build a very simple filter
- Make a 3DIT based off of existing components
- Design how you would implement a 3DIT with IFFI
- Answer questions after each step

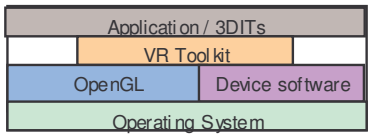
Outline

- Introduction
- Place in VR toolkit world
- Sample 3DIT creation process
- Subsystem description
- Question and answer

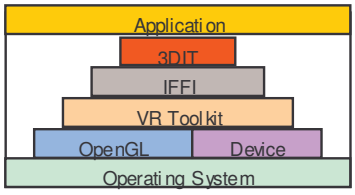
Intro

- IFFI – Interaction Framework for Innovation
- Purpose – Write your technique once, use it in any VR toolkit that has an IFFI adapter (~200 lines of code)
- Designed to make your life easier as a 3DIT developer

VR toolkit comparison

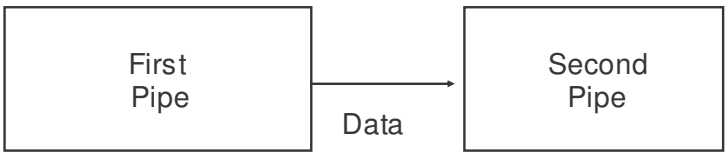


Typical VR Toolkit architecture
(update callbacks)

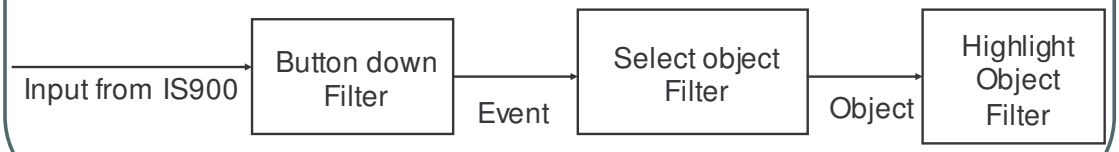


IFFI architecture
(data flow architecture)

Data flow architecture example



An example 3DIT built using this architecture

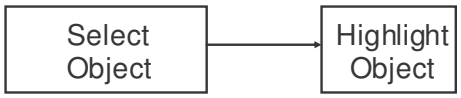


Sample 3DIT with IFFI development process

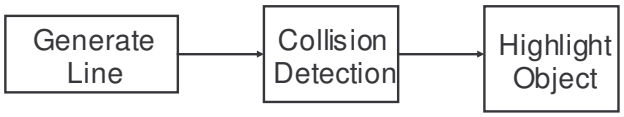
- Identify technique
- Identify breakdown of technique
- Consider reusability of parts and refactor design as necessary
- Implement parts of technique as filters
- Write a specification file to tie everything together

Sample 3DIT development process with IFFI

- Identify technique: Raycasting
- Break technique into parts:

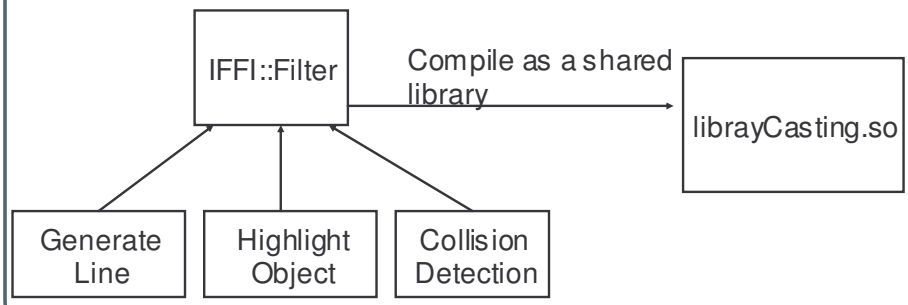


- Redesign for reusability



Sample 3DIT development process with IFFI

- Implement parts of technique as filters



Specification file for hooking everything together

- <iffi>
- <IT name="RaycastingSelection">

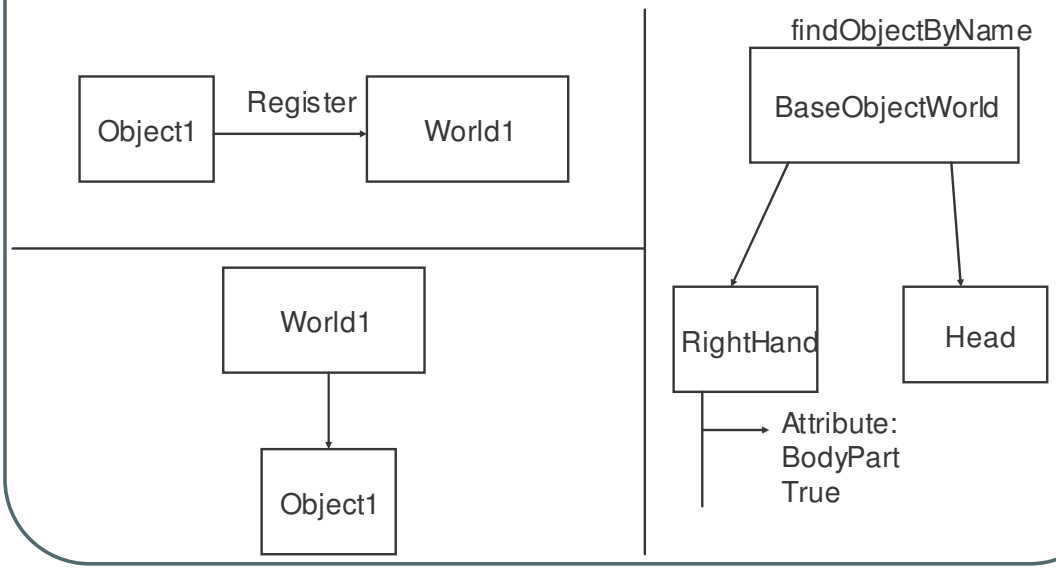
```
<filter name="GenerateRay" input="trigger"
   output="ObjectCollide"
   arguments="rightHand:1000.0"/>
<filter name="ObjectCollide" output="HighlightObject"/>
<filter name="HighlightObject"
   arguments="0.0:1.0:0.0:.30"/>
```

- </IT>
- </iffi>

Subsystem description

- IFFI is based around objects and worlds
- Objects have a specific function (information, graphics, devices, etc…)
- Worlds know how to manage, coordinate, and make use of multiple objects
- Objects register with the system and through inheritance, know how to add themselves to all the different parts

Objects / worlds example



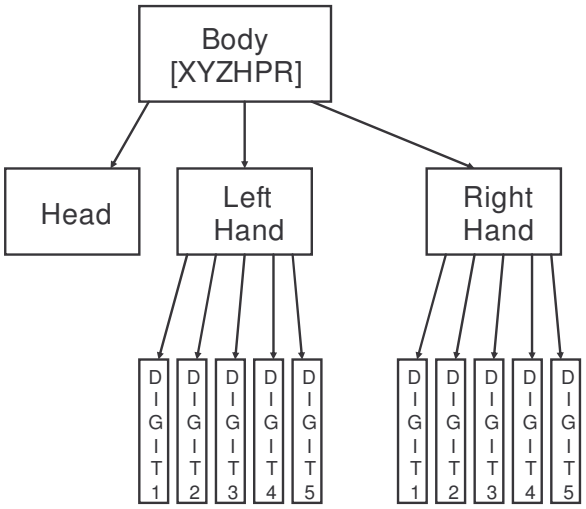
Basic objects and world

- Name, GroupName (class name), a unique ID, a group ID, and attributes
- Attributes are a name->value pair that can also store a void* pointer to hold whatever you want
- World allows you to find objects based on names, IDs, groupIDs, by attributes, etc…
- How you find objects easily and tag them to have specific properties

Graphical objects

- Extend the basic objects to provide graphics related functionality
- Goal – position, rotation and parent / child functionality.
- Also a body scenegraph so you can use it when creating 3DITs
- body (at the navigated location)
- head, leftHand, rightHand, [lr]digit[1-5]

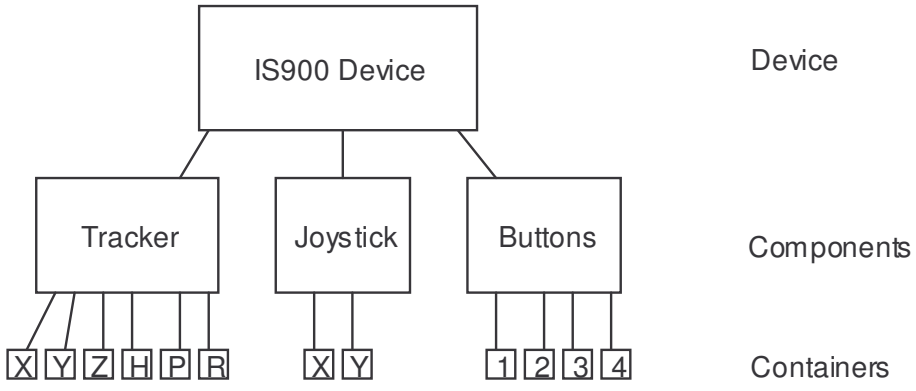
Sample Body scenegraph



Devices

- Devices have components that then have containers holding the data
- IS900 -> 6dof component, joystick component, button component
- 6dof component->6 containers each with one DOF
- Devices have meta-data
- DeviceManager has ability to find specific devices based off of meta-data
- Devices produce events

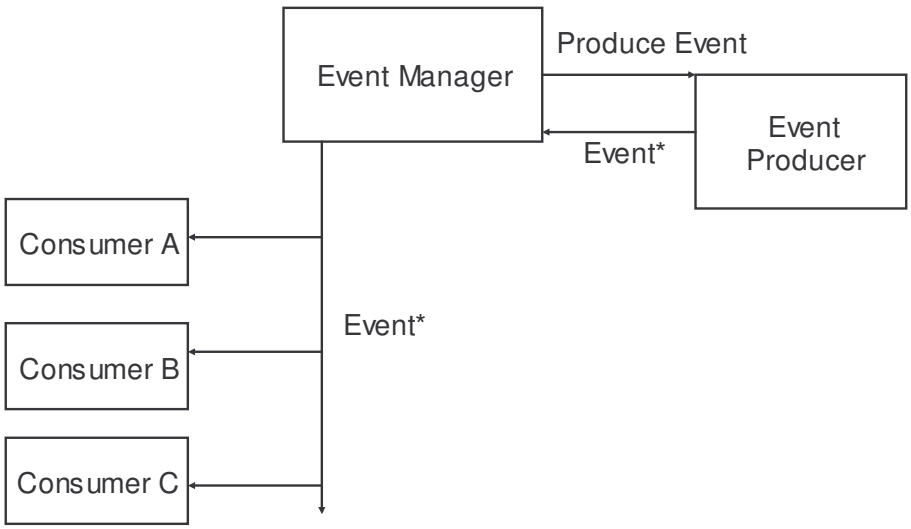
Device Example



Events

- Events have a specific type and are uniquely identified, reference counted, and are usually derived children
- Producer / consumer model with several choices
- EventManager handles subscriptions / shuttling events to the system

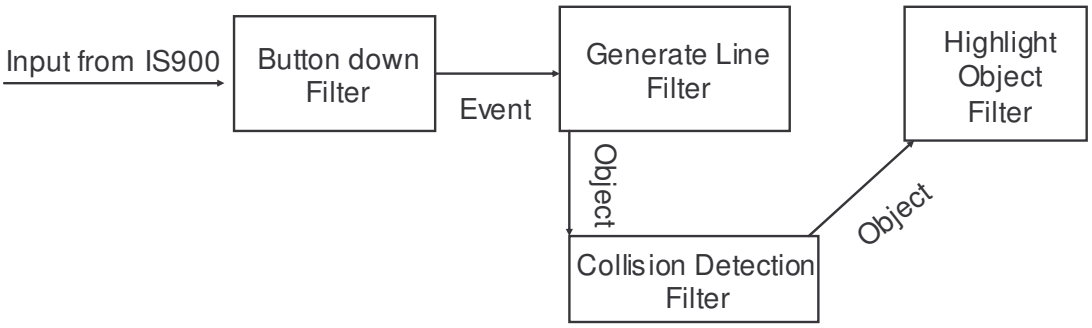
Event Diagram



Filters

- Specialized event model for a dataflow architecture
- A filter simply takes an input and produces an output
- The filter manager collects a group of filters into a technique, feeds the technique with specific input sources
- filterEnd(event) – design feature

Filters Example



A.2 The code used during the code review section of the evaluation

```
#include <string>
#include <stdio.h>
#include <dgl.h>
#include <idgl.h>
#include <iffi.h>

using namespace IFFI;
using namespace std;

//Create / configure IFFI / load sample configuration files
int config();

//This is a skeleton filter class that defines all of the necessary
//functions
class ExampleFilter : public Filter
{
	public:
	//Stub destructor
	virtual ~ExampleFilter() {};
	//Handle an event passed to the filter.
	//Identify the class type
	virtual string getGroupName()
		{ return "ExampleFilter";};

	//Take an event, process it, and then send it on to the next filter, or
	//return if there is an error.
	//This function is normally called by the FilterManager, and then becomes self
propogating.
	virtual int filterEvent(IFFI::Event* event)
	{
		printf("Got event of type %s\n", event->getGroupName().c_str());

		//This is the block of code necessary to determine if a button event has
		//been sent into the filter, and whether or not the button event is a button
		//press.

		//Wrong type of event, return an error
		if (event->getGroupName() != "ButtonEvent")
			return -1;
		//Convert the base event into a button event
		ButtonEvent* bevent = (ButtonEvent*) event;
		//Exit normally if the event is a release event
		if (bevent->buttonAction != ButtonEvent::PRESSED)
			return 0;
```

```
                //Tell the system that this function is finished with the event
                event->unref();

                //Pass on the event to the next function in the filter if
                //necessary
                return filterEnd(Filter::DEFAULT, event);
        };
        //Tell the filter manager what types of events we can handle
        //Filters are only asked to handle events that they can handle
        virtual std::vector<std::string> handlesEvents()
                { std::vector<std::string> Return; Return.push_back("ButtonEvent"); return
Return;};
        //This is used to handle pass in arguments from an XML file
        //Or your own custom method
        virtual int handleArguments(const string& args)
                {return 0;};

        //Used so that each technique can have its own copy
        //of this filter
        virtual Filter* cloneFilter()
                { return new ExampleFilter;};
};


int main(int argc, char** argv)
{

        //Configure IFFI
        config();
        //Create an example filter
        ExampleFilter filter;
        //Register it with the system
        filter.Register();
        //Update IFFI
        while (1)
        {
                IFFI::update();
        }
        return 0;
}



//Configures IFFI to use DGL
```

```
//Also loads input files
int config()
{
	IFFI::init();
	DGLPlugin* dplug = new DGLPlugin;
	dplug->Register();
	IFFI::configure();
	FileParser::loadConfigFile("local.xml");
	FileParser::loadConfigFile("iffi.xml");
	return 0;
}
```

Appendix B: Developer guide for 3DITs

Developing 3DITs with IFFI
by
Andrew A. Ray

Introduction
This is a guide to building 3D Interaction Techniques (3DITs) with the Interaction Framework For Innovation (IFFI). There are four major sections that this guide covers. The first is a big picture overview that helps sets the mindset for designing reusable 3DITs. The second section deals with the internals of IFFI that are needed when building 3DITs. Next, examples of how a few 3DITs have been built. The last section is installation information for IFFI.

The big picture
In order to understand how to build reusable interaction techniques, you need to put yourself in the right mindset. The first step is to think of a 3DIT as a series of steps that are accomplished in some order. An assembly line is a real world example of how a reusable 3DIT can be built. Information from devices is sent into the system where each station performs some action on the data, and passes the result on to the next station. Because software is not bound by physical limitations it is possible to reuse these components in any number of 3DITs without any cost, provided they are built with IFFI.

Your goal as a designer is to consider the different components that make up your 3DIT. Finding the right granularity of components for your 3DIT can be difficult. It is worthwhile to do so, because considerable amounts of your 3DIT could be used in other techniques. Another reason to do this is because if there is a need to tweak your 3DIT during testing, this process makes incorporating these tweaks much easier.

IFFI provides objects with special properties, and worlds that contain and work on these objects. Most objects are designed to help solve a problem in a specific subdomain of 3DIT work.

The high level steps for using IFFI in your programs
Include <iffi.h> in your program.
Use iffi-config to provide the includes and linking information for the program.
Call IFFI::init(), then IFFI::configure(), and then IFFI::update() in the loop part of your program.

Major guidelines for creating your own IFFI components
Always implement getGroupName for your class to uniquely identify it.
Always call Register / UnRegister on your object when appropriate.

Building 3DITs with IFFI

Implementing 3DITs with IFFI is generally done by breaking down a 3DIT into a certain number of filters and then implementing them. In IFFI, Filters are specialized EventConsumers. They are designed to allow for events to be easily consumed by Filters, and sent to other filters. When you are building a 3DIT you essentially have two tasks to handle. How to parse the event you have been handed, and how to create an event to send off to the next filter.

How to create your own filter:

- Create a class that derives off of IFFI::Filter.
- Override getGroupName to uniquely identify your filter
- Implement int filterEvent(Event* event) to handle the filter sent to your class
- Implement handleArguments as a virtual function that can parse information from the XML configuration files.
- Implement handlesEvents to return the types of events this filter can handle.
- Implement cloneFilter so that the filter can make a copy of itself.
- Add the filter to the IFFI system by calling instance->Register(); before you call IFFI::init; (It is possible to Register the filter after init, this is the simplest method though)
- Add the filter to the XML configuration file
- Configure the filter to take input from an input device or another filter

A skeleton framework for creating a filter can be found in IFFI/simpleProgram.

The filter class is a specialized event consumer in IFFI. It allows for an easier way to control the flow of data in IFFI. Take a look at the example code in IFFI/include/filters for a guideline to build your own filter.

It is rather simple to implement the getGroupName function, it just returns a string. The handlesEvents just returns a vector of strings that detail what events a filter can take. A common list of these events are ButtonEvent, DOFEvent, and EventMessage. The reason for the clone filter function is because a filter may be used in multiple techniques and need to have different copies. The handleArugments function simply takes a string and lets you parse it however you want. A function called IFFI::parseColonList(string) takes a : delimited list and returns a vector of strings that correspond to the different elements in that list. This simplifies argument parsing.

The most complicated part of creating a filter is creating the filterEvent(Event* event) function. Here is a skeleton for creating this functionality

```
filterEvent(Event* event)
{
        //Error check function for bad event and bad event type
        if (event != NULL && event->getGroupName() != "DOFEvent")
                return -1;

        //Event handling code
        EventMessage* nextEvent = new EventMessage;
        nextEvent->setSubject("DOFEvent");
```

```
        //We are done with the event, so unreference it for garbage collection
        event->unref();

        //Send the event to the next filter
        return filterEnd(nextEvent);
}
```

This is a pretty simple function, but there are a couple of important parts to notice. All events in the system are reference counted, so the unref takes care of garbage collection. The filterEnd function is the centerpiece of the filter class. By simply passing an event into this function, you do not have to worry about how to call the next filter, the IFFI framework takes care of this for you.

XML configuration

In order to use the filters you have created you need to do two actions. The first is build a program that supports IFFI (IFFI/simpleProgram). Next you need to use an XML file to tell IFFI to use your filter in a technique. This is accomplished by using an XML file. Here is a simple XML file that describes how to include a model and 3DIT for a virtual world.

```
<iffi>
        <graphicalObject name=”hand” model=”hand.ive” type=”OSG>
                <childOf name=”world”/>
                <position x="0.0" y="0.0" z="0.0" />
                <rotation h="0" p="0" r="0"/>
        </graphicalObject>

        <IT name="RaycastingSelection">

                <filter name="GenerateRay" input="trigger" output="ObjectCollide"
arguments="rightHand:1000.0"/>
                <filter name="ObjectCollide" output="HighlightObject"/>
                <filter name="HighlightObject" arguments="0.0:1.0:0.0:.30"/>
        </IT>
        <local>
                <inputMap value="wand"     source="is900Wand:wtracker"/>
                <inputMap value="trigger"  source="is900Wand:buttons:button2"/>
                <inputMap value="head"     source="headTracker:head"/>
                <inputMap value="position" source="navigation:position"/>
                <graphicalHandler name="OSGGraphicalHandler"/>
        </local>
</iffi>
```

The iffi tag tells the program that this portion of the file is for the IFFI framework to handle. You may add your own tags to the file and parse them manually. TinyXML is used for the XML handling in IFFI.

The graphicalObject tag names an object, tells it which file to load into the world. The childOf sub-tag allows you to specify where the object lives in the graphical world. The position and rotation are what is used to specify what position the object has in the world.

The IT tag is used to create an interaction technique in IFFI. Each technique must have a unique name. A technique is composed of different filters. Each filters name is an instance of the getGroupName function in your filters implementation. When a technique is created via an xml file, the system finds the filters in the technique, creates clones of them, and groups all of the clones together in a single technique. This is so that filters can be used more than once in the system.

Each filter can be fed by multiple sources of data. These sources are either input devices or other filters. For example, the filter GenerateRay takes input from the trigger event, and sends its information to the ObjectCollide filter. The Object Collide filter sends its output to the HighlightObject filter. The arguments option in the filter tag collects the field and sends this output to the handleArguments function in your filter.

The local tag is where you specify how a specific device in the system corresponds to an input for filters. This is done so that different devices can be used in the system without requiring changes to existing 3DITs. The syntax is to have a particular name (wand/trigger/head) correspond to a particular component or container in a device. The syntax for the source field is deviceName:componentName:containerName. The graphicalHandler specifies which graphics engine should be used to load the models in the XML file.

Sample xml files can be found in examples/osgTest.

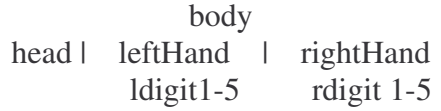
Scenegraph provided with IFFI

IFFI provides a simple scenegraph for loading models into the world. It consists of a scene node that has two children, the world, and the ether. The world is where object affected by navigation are put, and the ether is where objects that are not affected by navigation are put.

In order to get access to these nodes you can call:
GraphicalObjectManager::getGraphicalObject("head");

Representation of the human body in IFFI

One feature that will be of interest to you with developing 3DITs is the representation of the human body in IFFI. Here is the simple SG for the body in IFFI.



It is possible to add adjust this to add in support for other body parts, but in general this lets you use head and wand tracking, which is normally supported for most VR installations. The body

holds the users location in the world, and all of the children are populated with the data values provided by the VR toolkit's trackers.

In order to get access to these nodes you can call:
GraphicalObjectManager::getGraphicalObject("head");

Subsystems of IFFI
Each of these sub systems is described in the include/iffi directory. Feel free to look at the header files to determine the exact interface you need to use.

Objects
In general to use an object in IFFI, you need to do two steps. The first is create it. The second is to have the object register itself with the IFFI framework: myObject->Register(). All of the objects in IFFI are derived from a base object. This base object provides the following functionality:

1) Variable number of attributes for each object (name / value / void* triplet). This is useful for tagging and providing information about an object
2) A name for the object
3) The ability to identify the particular group of objects that the object belongs to
4) A unique ID for the object and its group
5) The ability to handle arguments provided by another part of the system
6) A static container for all of the objects with various methods of finding objects.

Reference counted objects
All objects that derive from the ReferenceObject class have a ref and unref function that keeps track of the number of parts of your program that point to the object. By default the reference count is 0, the object is deleted. Whenever IFFI is updated it automatically takes care of freeing unused objects.

Graphics
GraphicalObjects are designed to be abstract versions of a node in any graphics package. The only current implementation of a GraphicalObject is the OSGGraphicalObject class. This allows for open scene graph nodes to be used with IFFI.

The current functionality of GraphicalObjects is to determine parent / child relationships, and the position of the object in the world. There are add/remove Parent/Child functions, and get/set Position and Rotation functions. There is other functionality stubbed out in the header files, but it doesn't work.

Events
Events are the circulatory system for IFFI. The event system is based on producer consumer relationships. Producers are uniquely identified and can be referred to by name in the system, and the events they produce are identifiable by type. Consumers can choose to receive all events from a producer, events of a certain type from a producer, or all events of a certain type that the system provides. The exact API calls for these are in the Events.h file.

In order to produce your own events in the system, make a class derived from EventProducer. Event producers are updated every time IFFI::update is called. In this update function, create your events, tag them to be from the specific producer, and add them to the EventManager by calling EventManager::addEvent(Event*); The Event system takes care of the rest for you.

In order to consume events, make a class derived from EventConsumer for most cases, or EventHandler (if you want to handle all events of a specific type). EventConsumers have two main functions that have be implemented. The first is a handleEvent function that actually handles an event given to the consumer. The second is a handlesEvents function. This function returns a vector of strings that identify what events the consumer can handle.

Because of the Register / Unregister functionality in IFFI, all that is needed to connect producers and consumers are the following functions:
EventManager::(dis)connectProducerConsumer(sourceID, targetID);
EventManager::(dis)connectConsumerToEventTypes(consumerID, vector<int> events);

By default the event system doesn't provide functionality in the events other than what is related to the event architecture. In order to reduce the amount of work developers have to use to get their information from part of the system to another, the EventMessage is provided. The EventMessage is a class that allows for a developer to put any number of primitives(and strings) into a message, and then allows them to read them in the same order they were written. This provides a way to populate events with information and to send them to other parts of the system. The only disadvantage of EventMessage is that it requires that you do not use tabs in any of your data.

Using the getSubject method and comparing the result to EventMessage::BAD_MESSAGE can determine whether or not an event message is valid.

Example of how to create an EventMessage (should be done in a function of a class that is derived from EventProducer):

```
EventMessage* eMsg = new EventMessage;
eMsg->setSubject(“A sample message”);
eMsg->addInt(1);
eMsg->addFloat(2.3);
eMsg->addDouble(3.44);
eMsg->addChar('a');
eMsg->addString(“Testing”);
eMsg->setSourceID(getID()); //Set the source ID for the message to be this producer
EventManager::addEvent(eMsg); //Send the event into the system
```

Example of how to get data from an event message:

```
something(Event* evt)
if (evt ==NULL || evt->getGroupName() !=”EventMessage”)
return -1;
EventMessage* eMsg = (EventMessage*)evt;
printf(“Subject is %s\n”, eMsg->getSubject().c_str());
```

eMsg->getInt();
eMsg->getFloat();
eMsg->getDouble();
eMsg->getChar();
eMsg->getString();

Making a graphical object into an event can be done with the following function: Event* IFFI::makeGOM(GraphicalObject*). You can get a graphical object from a message by using GraphicalObject* IFFI::getGOM(EventMessage*).

Devices
Devices are part of an architecture that allows for any VR input device. The system is designed so that devices are can be made up of any number of components. Each of these components have several different containers that hold the raw device data. There are two main types of containers, DOF and button containers. DOF components emit events, while button containers emit events. This is so that X number of events have to be sent when an immersive tracker moves.

Most of your work with devices will be with the events that they emit. There are two types of events that devices in the system produce. The first type is ButtonEvent, the second type is DOFEvent. These are also what you get from the getGroupName() function. Both events also allow you to get the component or container that produced the event.

DOFEvent capabilities:
This event contains a vector of doubles that represent the number of DOF by a device. It is a public member called dof. There is a getNumDOF function and a getDataFor(string) function that allows you to get the data associated with a specific dof (i.e. x or y).

ButtonEvent capabilities:
This event contains information about a button (the member button of type int), and its action (the member buttonAction). Currently buttons are just integers that tell you whether the button was pressed or released (see the buttonAction). There is also a getChar function if the button event was a keyboard event.

In order to write your own device the IDevice class is the easiest way to build such a device. It provides the following functionality:
add/get Button/DOF components
Allows for the source ID for these components to be found.
Allows for a sourceID to be manually set for button devices (i.e. keyboards only having one source instead of X sources (X is the number of keys))
Easily sending data into the components via a string / device mapping

Filters included with IFFI

AttachChild -
GenerateRay -

HighlightObject -
NonLinear -
ObjectCollide -
ObjectSubstitute -
StateSelector -

Building IFFI:
All of the requirements for building IFFI are as listed below:
Open Threads (build with make install)
cvs -d :pserver:cvsguest@openscenegraph.org:/cvs/openthreads co OpenThreads
Open Producer (build with make install)
http://www.andesengineering.com/Producer/download.html
Open Scene Graph (cmake . && make && make install)
http://www.openscenegraph.com/osgwiki/pmwiki.php/Downloads/Downloads
DTK (build with ./configure && make install)
DGL (build with ./configure –with-osg && make install)
http://sourceforge.net/project/showfiles.php?group_id=70895
Finally to build IFFI:
wget http://snoid.sv.vt.edu/~anray2/IFFI-0.1.tar.gz
tar xfz IFFI-0.1.tar.gz
cd IFFI
./bootstrap && ./configure && make install

Using DGL with IFFI:
In order to use DGL with IFFI you need to include idgl.h, link to libidgl, and create/register a DGLPlugin object after the IFFI::init.

CoVE support in IFFI:
Check out cove with:
svn co https://secure.opentechinc.com/svn/cove/branch/dev
Build cove via:
./configure && make install
Build the cove plugin
cd IFFIsrc/cove
qmake
make
./run (this will load raycasting with CoVE and a sample world)
Look at the run script to see how to use it with your CoVE application

How to use the demo 3DITs:
export PATH=/path/to/iffi/bin:$PATH
In examples/osgTest type make
./RUN-raycasting or ./RUN-homer or ./RUN-gogo

How to build a 3DIT:
Put iffi-config in your path (export PATH=/path/to/iffi/bin:$PATH)

Type make in the directory where your 3DIT is present

Low level decisions to make a 3DIT:
Do you want your 3DIT to be a plugin or a program? You can build plugins using DIVERSE and CoVE and have them load filters into the system, or you can integrate your 3DIT with the source code that you have. For examples of plugins, see the DIVERSE or CoVE examples. For an example of how to build the filters into your source code see examples/osgTest.