Concept-Oriented Design in Chasm: Conversational Domain Language Inspired 3D User Interface Design and Development

Chadwick A. Wingrave

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 Bowman, Chairman
Dr. Godmar Back
Dr. Carlos Evia
Dr. Robert Jacob
Dr. Manuel Pérez-Quiñones

July 11th, 2008
Blacksburg, VA

Keywords: 3DUI, 3D interaction, HCI, Chasm, Concept-Oriented Design

Abstract

Title: Concept-Oriented Design in Chasm: Conversational Domain Language Inspired 3D User Interface Design and Development

By: Chadwick A. Wingrave

In my experience, novel ideas for 3D interaction techniques greatly outpace developers' ability to implement them, despite the potential benefit of these ideas. I believe this is due to the inherent implementation complexity of 3D interfaces, without sufficient support from methods and tools. Believing a developer-centric representation could overcome this problem, I investigated developer practices, artifacts and language. This resulted in the theory of Concept-Oriented Design and Chasm, a prototype realization of the theory. The key feature of Concept-Oriented Design is its use of developer-centric representations to create a multi-tiered implementation, ranging from an envisioned behavior expressed in conversational language to low-level code. Evaluation of Chasm by domain experts and its use in multiple case studies by volunteer developers has demonstrated that Concept-Oriented Design in Chasm addresses many of the problems of 3D design and development.

Acknowledgments

I wish to thank my wife and dog.

I wish to thank my family and friends. You know who you are.

I wish to thank my advisor, committee and academic friends. Its been a long trip.

Good-bye and thanks for the fish.

This page intentionally left blank.

Table of Contents

List of Tables

List of Figures

This page intentionally left blank.

Chapter 1 - Introduction

3D User Interfaces (3DUIs) are becoming commonplace with the maturation of the necessary hardware and computational technologies. To develop these 3DUIs, event-based architectures and other development approaches that were successful for 2DUI development have been applied but are problematic in 3DUIs for many different reasons [Gre91][Mye93][Her94][Jac99]. This has created problems in the development and reuse of 3DUIs despite the fact that many new 3DUIs are possible and developers are readily able to think about and discuss them. To avoid a potential stagnation in 3DUI development, this dissertation presents the development methodology of Concept-Oriented Design (COD) and its prototype toolkit Chasm that are based upon developer artifacts and language so as to scale with developer thinking. COD in Chasm improves the transition of developer 3D interaction ideas into a 3DUI implementation.

1.1. Motivation

New 3DUIs for medical, training, visualization and military applications [Bro99][Azu97][Abo00] as well as consumer gaming worlds like World of Warcraft, products such as the Nintendo Wii and smart phones like the Apple iPhone are being created. All of these 3DUIs, with their virtual worlds and augmentation of reality, are filling human entertainment and information requirements through spatial and novel interaction. Despite noteworthy application successes [Bro99][Azu97][Abo00], a problem looms. The problem is that 3DUI design and development is both complex and difficult [Gre91][Mye93][Her94][Jac99] with the potential to stall the growth of 3DUIs. As more interfaces shift their interaction from 2D to 3D, the potential effect of stagnation becomes widespread.

Advances in research fields have enabled new ideas and interaction to occur as well as making available new application areas. Some of these research areas include Virtual Environments, Augmented Reality, Ubiquitous Computing, Tangible Computing, Wearables, Perceptual Computing, Machine Vision and other areas that exist along the Mixed Reality Continuum [Mil94]. Together, these fields advance 3DUIs, though the approach or focus area may differ.

Advances in technologies have enabled new 3DUIs. This includes powerful commodity graphics cards, low-power processors, improved battery life, touch screens, high-resolution displays, brighter projectors, smaller and higher resolution cameras, etc. As the prices drop and robustness improves in these technologies, new application areas become addressable with 3D interfaces. Where once 3DUIs were for simple interaction in dangerous or expensive tasks, now complex interactions using commodity hardware in common applications are possible. For example, web interfaces, which were once considered the epitome of simplified desktop interaction, are becoming complex, interactive and mobile in the age of Web 2.0 and smart phones.

1.2. Problem Statement and Approach

The design space of 3DUIs is large, with a number of potential 3D interaction techniques for any given 3D interaction task. This can be seen in the work of taxonomy creation in 3D interaction [Bow99][Pou97][Dac01] and the large number of 3D interaction techniques that have been and continue to be created [Bow05]. But a problem exists for 3DUIs. Despite the ability of designers to think of and freely discuss 3DUIs, these interfaces are prohibitively complicated to design and develop.

Several problems are at the core of 3DUI design and development, leading to a 3DUI stagnation.

- Each additional feature of a 3DUI increases the number of actions and states of the user, environment and interface. Since each action must be considered in regards to each state, linear growth in actions and states result in non-linear growth of complexity in the implemented 3DUI. This results in successive interfaces that have only slightly more functionality being much more complicated to design and develop. This problem been noted as a classical problem of software engineering [Bro87], referred to as the State Space Explosion [Har98]
- Event-based architectures, successful in 2DUI development, are not a good match to 3DUI development. First, event-based architectures do not scale well with additional functionality [Mye93] as 3DUIs have many more states and actions compared to 2DUIs. One reason for more states and actions is that the user has to be represented in the environment, whereas in 2DUIs users act through devices. Second, event-based architectures do not encapsulate functionality. Because of this, event handlers and callback functions have to contain all functionality for all features of the system. Finally, in these event-based architectures there is no intermediate representation during development. As developers implement a system, they go directly from their ideas to code.
- Developers can talk about and discuss a 3DUI with others, even though that 3DUI is difficult to design and develop. Observations of conversational domain language and *developer artifacts* (any artifact created during development to help the developer understand their development problem such as notes on scraps of paper, journals, sequence charts, meeting notes and scenarios) show several structures useful for the encapsulation of complexity and for good decompositions. Unfortunately, current implementation representations in code do not parallel these structures. In many cases, current methods decompose a 3DUI as a flow that leads to bad system designs [Par72]. Additionally, developers are often not aware of their own assumptions and once implementation begins, developers often finds they do not fully understand what they are implementing, leading to changes in design.
- 3DUIs have short operational lifetimes that lead to problems reusing the implementations as well as problems composing higher-level meanings in development tools. One reason for this is changes in hardware and software over time, leaving old 3DUIs dependent upon old platforms. Another reason is that new 3DUIs have requirements that conflict or requirements that were never considered in the old 3DUI. Developers then are forced to choose between reuse, and thus the modification of code they do not fully understand, or the creation of new code for their 3DUIs' requirements. As newly added functionality or modification potentially breaks existing functionality, developers often opt to create new 3DUIs from scratch to avoid having to learn about the internals of the reused code. An additional reason to avoid reuse is that old code often does not externalize the necessary "hooks" [Pol06] for developers to gain access to the internals. The end result is little composition and reuse of code to create higher-level and more reusable 3DUIs, even though many systems have similarly functionality.
- Each 3DUI developer has to have many different skills to understand all aspects of 3DUI design and development. Non-standard hardware and

software systems as well as fragile hardware devices have to be understood. 3DUI developers must understand the physics behind 3DUIs (magnetism, optics, etc), they must have carpentry skills to build the environments, electrical engineering knowledge to develop hardware devices and artistic skills to model and design the 3D environments. Developers must create complex algorithms for graphics and parallel and distributed processing of data, all of which need to support a real-time response in the 3DUI. Developers must also understand how to develop 3DUIs that interact over time and in 6 degrees of freedom (DOF).

- New approaches to the design and development of 3DUIs, as well as architectures in software engineering, use many approaches that are improvements over event-based architectures but are still lacking. One approach is to use high-level representations for more condensed descriptions of 3DUIs. This enables better visualization and in some cases, code stubs can be generated for developers to place their low-level code. Development is iterative however so the high-level representations are constantly modified during the development process, requiring code refactoring and regeneration, which ultimately leads to longer development time and the potential for code loss. Another approach is to use declarative approaches to 3DUIs. Unfortunately, the design space of 3DUIs is large so such approaches run the risk of being too narrow to offer a useful level of expression or too large to be easily understood. Additionally, declarative approaches run the risk of not externalizing enough of the internals to allow sufficient modification for new 3DUIs. Finally, since 3DUI research is still early, it may be too early to know if a declarative approach is useful for the range of 3DUIs that need to be developed. A final approach is the use hierarchical state machines such as Statecharts [Har98]. While there are advantages to such an approach, there are still problems regarding the order of event processing and the global scope of actions, both of which allow newly added functionality to break existing functionality. Additionally, Statecharts was designed around an assumption of serial interaction and, though it does have additions to its syntax to describe parallel interaction, the notation is verbose.

To address these problems, Concept-Oriented Design in Chasm was created as a design and development approach that provides methods to help developers manage complexity and overcome the problems associated with implementing 3DUIs. This is achieved through the following features:

- A *concept* is the unit of development in COD, representing an encapsulated cohesive idea in an interface – a reusable chunk of functionality. A concept has a multi-tiered representation, each tier encapsulating a different type of information. These tiers externalize the concept's encapsulated functionality, for composition into concepts of higher meaning and for reuse.
- Conversational domain language and design artifacts guided the design of the tiers in a concept so as to match the growth in developer understanding during development. In this way, the representation scales with 3DUI interface complexity in a manner consistent with developer understanding. Additionally, as developer understanding changes during the design of a 3DUI, the changes are limited to the current tier of development. The decompositions achieved by developing according to conversational domain language leads to more reusable concepts, due to understandability and a closer match to how developers think of the 3DUI. The tiers also help incorporate new requirements into a concept while limiting the change

impact.

- Hierarchy and the reduction of global state improve on existing approaches. Similar to Statecharts [Har98], Chasm creates hierarchies of concepts to reduce the state space size. Similar to a state machine, Chasm's automata tier captures the internal flow of behavior in a concept. Similar to event-based architectures, causal statements in a concept's causality tier map a state of one concept to an action in another concept. In this way, Chasm removes the need for global variables maintaining the concept's state as well as global scope for actions that can lead to unexpected behaviors. Additionally, as opposed to event-based architectures, developers do not have to create and maintain explicit events in their implementation, instead relying on states already existing in the automata tier of a concept.
- The action processing in Chasm sets the order in which code runs in the system while minimizing the developer's involvement in setting that order. This is through the ordering of the causal statements and execution of the automata in each concept. This enables the developer to focus on adding new functionality while spending less time on the unintended behavior of that functionality. Reuse of concepts is improved for the same reason, as functionality can be pulled from one system and placed in another with minimal effort on the part of the developer.

1.3. Concept-Oriented Design in Chasm

Concept-Oriented Design in Chasm, based upon developer artifacts and conversational domain language so as to scale with developer thinking, improves the transition of 3DUI ideas into an implementation. Concept-Oriented Design is a methodology for software design and development based on five principles originating from an analysis of 3DUI development. These principles are envisioned behavior, artifact and language focus, tiered representation, longevity and community. Chasm is an implementation of Concept-Oriented Design with a four-tiered representation capturing first conversational language, then developer causal understanding between concepts, the internal flow in a concept and lastly code. The Chasm system handles the execution of the concepts at runtime, allowing for faster design and development and greater scalability and reuse.

The creation of COD in Chasm began with an analysis of how developers think about 3DUIs, as developers have little difficulty thinking about the same 3DUIs that are difficult to design and develop. This was achieved by interviews and code and developer artifact analysis. Because of the structures found in developer language in the developer artifacts, a study was performed to gather developer language descriptions of 3DUIs. From this corpus of knowledge, 3DUI problems and themes were identified which led to the creation of the five principles of COD and ultimately to the prototype system of Chasm.

The key ideas of Concept-Oriented Design in Chasm are the use of concepts and their organization into tiers to structure developer thinking as they design and develop a 3DUI. A *concept*, as the unit of development in COD, represents a cohesive idea in an interface – a small reusable chunk of functionality. That functionality has an envisioned behavior, which is a description in the conversational language of the domain. To develop a complete 3DUI, developers decompose the interface into concepts along domain language descriptions and then compose the concepts into flows of behavior. This is achieved by describing concepts in four tiers that separate out different types of understanding in the concept, as developer

understanding grows during development.

The four tiers in Chasm, shown in Figure 1-1, are envisioned behavior, causality, automata and code. The first tier in a concept, the envisioned behavior tier, is expressed in developer conversational language. An analysis of design artifacts shows that the next level of understanding is causality between concepts so the next tier is the causality tier, representing the causal behavior in causal statements such as "When X, then Y." These first two tiers allow the developer's ideas to be expressed and readily changed as their understanding grows. It also creates a decomposition approach that matches developer understanding. The automata tier is a state machine, representing the internal flow of behavior in the concept. External events in the system are mapped the concept's own actions in its state machine, causing transitions to new states. On entry to a state, the state entry function code in the code tier is executed. At runtime, Chasm manages the order of execution of the automata tiers using the causal statements. This separation of development into tiers allows developers to work all tiers simultaneously, with connections between tiers clearly defining the impacts of a change.

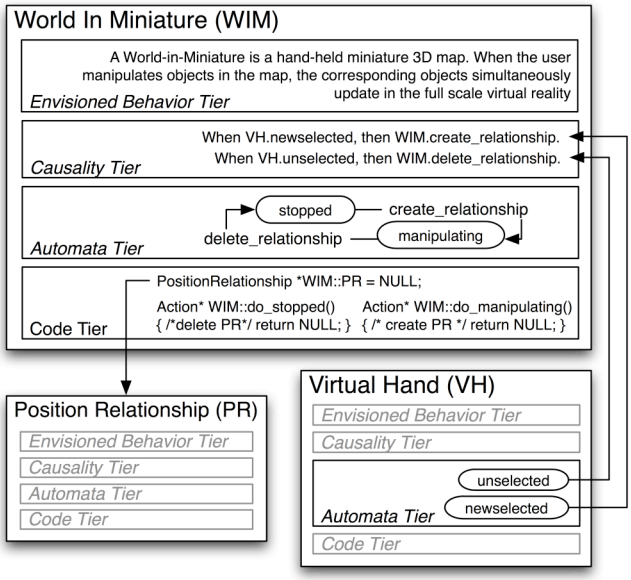

Figure 1-1. The four tiers of Concept-Oriented Design in Chasm represent developer understanding as they implement a 3D interface, shown here for the WIM [Sto95] technique. Decomposing development in this way creates cohesive concepts that are executed by Chasm as flows of events.

A mix of evaluation approaches achieved a robust evaluation of COD in Chasm. The evaluation types included an evaluation by custom criteria, an evaluation by a standardized set of criteria (the Cognitive Dimensions Questionnaire [Bla00]), the evaluation by domain expert, the longitudinal evaluation by developers as well as measures and metrics of 3DUI case studies. The mix of approaches created a robust evaluation while compensating for the weakness of each individual evaluation approach.

In summary, we claim that a 3DUI development methodology based upon developer thinking improves developers' ability to design and develop 3DUIs. The methodology is Concept-Oriented Design and Chasm is the prototype implementation. Therefore, we put forth the thesis that COD in Chasm improves the transition of developer 3D interaction ideas into a 3DUI implementation.

1.4. Research Questions

This dissertation is formed around the four research questions below (RQ1, RQ2, RQ3, RQ4) with a hypothesis and approach to each. Each question is addressed by its own chapter.

1.4.1. RQ1 – The Investigation - Chapter 3

Question - Why are current methods and tools insufficient for the design and development of complex 3DUIs?

Hypothesis – It is hypothesized that existing development methods and representations do not match how developers think of the 3DUI. The implementation should be readily understandable by developers.

Approach – This dissertation analyzed existing methods and tools of 3DUI developers, including their development processes and descriptions of 3DUIs.

1.4.2. RQ2 – Concept-Oriented Design - Chapter 4

Question - What are the core principles of an effective representation for the design and development of 3DUIs?

Hypothesis – It is hypothesized that a few core principles can embody the problems and incorporate the beneficial themes identified in RQ1.

Approach – Create the principles of Concept-Oriented Design as a methodology of 3DUI design and development.

1.4.3. RQ3 - Chasm - Chapter 5

Question – How can Concept-Oriented Design be instantiated as a tool to improve the design and development of 3DUIs for developers?

Hypothesis – The principles of Concept-Oriented Design can be created in a prototype tool that allows 3DUIs to be developed in ways that match developer understanding and execute with minimal developer consideration.

Approach – Select the most important principles of Concept-Oriented Design and implement them in a tool called Chasm. Work with real developers in real projects, changing and improving Chasm as problems develop. Identify problems caused by the instantiation of Concept-Oriented Design and overcome them well enough to be used by developers.

1.4.4. RQ4 - Evaluation - Chapter 6

Question - How effective is Chasm in the development of 3D interfaces?

Hypothesis – Experts should understand the benefits of Chasm for 3DUI development and with experience developers should be able to implement 3DUIs that match how they think, with implementations understandable and able to be reused.

Approach – Chasm evaluations are to be performed by comparison to software engineering criteria, by domain experts and by developers with long-term exposure to Chasm in the design and development of 3DUIs.

1.5. Motivating Example: SSWIM

In the extension of the World-In-Miniature (WIM) technique [Sto95] to have scaling and scrolling functionality [Win06], the problems associated with 3DUI development are demonstrated. Shown at the left of Figure 1-2, the WIM technique was described as "… a hand-held miniature 3D map. When the user manipulates objects in the map, the corresponding objects simultaneously update in the full scale virtual reality" [Sto95]. The WIM can be used to manipulate objects but also to travel by using the WIM to select a new location for the user and animating into the WIM, to move to that new location [Pau95].

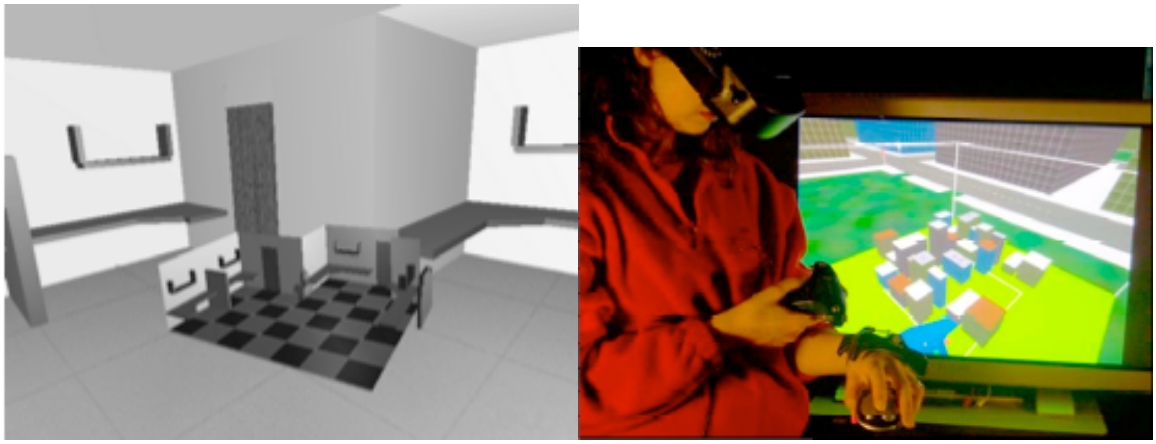

Figure 1-2. The World In Miniature (WIM) technique (left) allows the user to work with the world in the palm of their hand. The Scaled Scrolling WIM (SSWIM) (right) added scaling and scrolling functionality for larger world interaction but was much more complicated to design and develop.

The Scaling Scrolling WIM (SSWIM) [Win06] added scaling and scrolling functionality to allow the user to work with larger worlds, shown at the right of Figure 1-2. The scaling functionality was mapped to the movements of a mouse wheel: when the mouse wheel moved up or down, the world scaled up or down. Scrolling was performed by the user holding their hand at the edge of the SSWIM, causing the world to scroll. This was similar to the Mac OSX Finder or Windows Explorer applications.

The additional functionality added multiple conflicts and requirements to the existing implementation of the Virtual Environment containing the WIM. The conflicts that arose can be classified as generic, scaling, scrolling and hardware-related.

Generic Conflicts

- The cityscape used to test the SSWIM affected the user's cognitive model. It was determined that aligning the SSWIM with the world improved the user's ability to orient themselves. So, the SSWIM appeared attached to the user's hand as it remained aligned with the world.
- The orientation alignment changed the model users had of the SSWIM. In [Pau95], animating into the WIM was the preferred means of travel because it preserved the user's orientation in relation to the WIM. With the SSWIM being oriented with the world, the preferred travel animation was to animate the user moving in the world from their current location to the new location. This resembled map-like movement.
- Because the SSWIM was aligned with the world, it could not be attached to

the user's hand in the scenegraph, only updated each frame to appear attached to the user's hand. Functionality that assumed the SSWIM was attached had to be corrected.

Scaling Conflicts:

- To change the scale, the scenegraph had to be modified so changes to scale did not impact other objects and functionality of the WIM.
- The extremes of scale had the potential for the user to lose the world they were in. So, max and min scales had to be created and adjusted for each city brought into the SSWIM.
- While at a high level of scale, users could place their position far outside the workable area, so bounds had to be defined. Again, this had to be adjusted for each city brought into the SSWIM.

Scrolling Conflicts:

- A button had to be used to start the interaction with the SSWIM but a button was also used to travel around the environment. Using two different buttons for the two travel techniques proved too cognitively difficult for users. So, the same button had to be used with the location of the button press, inside the WIM or out, distinguishing between the user traveling or using the SSWIM.
- The users sometimes accidentally clicked in the WIM when they wanted to use the other travel technique. So, a smaller space was defined in the WIM to be the starting of the WIM manipulation. This smaller space was shown as a white box outline on the bottom of the SSWIM.
- Scrolling too fast created difficulties so an area in the middle of the SSWIM was defined as not scrolling and the area outside this was scrolling. The scrolling ramped up in speed the further the user moved from the center of the SSWIM.
- It was sometimes difficult to know if and in what direction the user was scrolling. So, an arrow appeared when scrolling to point in the direction of the scroll.
- The event handlers that implemented the SSWIM had to first establish the state of the system before performing a scroll. This was because all scrolling requirements were only active after the user began using the SSWIM.

Hardware Conflicts:

- The mouse sensor occasionally brushed up against the user and moved the mouse in the computer running the application. When the cursor went out of the application window, the mouse events stopped being sent to the 3DUI. This was eventually solved by a piece of tape over the optical sensor.
- A wireless mouse was used to remove problems with a cable being in the user's way but did present problems when the user moved too far from the mouse base station.
- In previous WIM implementations, magnetic trackers were used but with the SSWIM, a hybrid acoustic-accelerometer tracker was used. Since the user was holding one hand over the other during the interaction, the acoustic tracker's sensor was blocked which caused the user's hand to drift and stop responding. To resolve this, the virtual hand's position was offset such that though the hands appeared to be over top one another, they were actually a

few inches offset so the sensor was not covered.

This example demonstrates the types of problems typically encountered when designing and developing 3DUIs. The addition of seemingly simple requirements led to many conflicts, some of which could not have been predicted at design time. This requires constant modification to the 3DUI at many different levels and multiple iterations to achieve its usability. The effect was a long development time for this 3DUI. A development tool that matched how developers thought would reduce the time to implement the 3DUI and potentially offer reusable components.

1.6. Summary of Work

This chapter presented this dissertation's motivation, approach and research questions. This dissertation is hereafter structured in the following manner:

Chapter 2 discusses related work, including existing work related to software engineering and software paradigms. It also includes existing work in the design and development of 3D interfaces.

Chapter 3 investigates the developer's understanding of 3D interfaces. This includes the collection of development artifacts and a study collecting their language used to describe 3D interaction video clips. Notably, this investigation resulted in themes and five problems of current 3D interface development, which are also presented in the chapter.

Chapter 4 discusses Concept-Oriented Design, a methodology of design and development based upon the results in chapter 3. Concept-Oriented Design is based upon the five principles of envisioned behavior, language focus, tiered representation, longevity and community.

Chapter 5 discusses Chasm, a tool used to implement the methodology of Concept-Oriented Design. It discusses its inclusion of existing development methods, tiered representation, vocabulary and tools. Chapter 5 also includes Chasm's approach to problems that arose during its creation.

Chapter 6 discusses the evaluation of Chasm. This occurs through expert evaluations and long-term evaluations using ten case studies. The case studies were created by three developers and measures and metrics were pulled from these.

Chapter 7 concludes with a summary of contributions and future work.

Chapter 2 - Background and Related Work

This chapter begins with the background information required to understand 3DUIs and 3DUI design and development in this dissertation. Following this, the tools and approaches of the more general topic of software engineering are discussed. These include the common architectures already used and those that hold promise. The background information ends with the topic of forms of evaluation, not of 3DUIs, but of software engineering.

The related work is broken into three topics of general 3DUI tools, 3DUI specification and 3D behavior specification. It is here that promising work related to this dissertation is compared and contrasted. I conclude with a summary of both the background and related work and its impact on this dissertation.

2.1. Background: 3D Interaction Design and Development

As stated in Chapter 1, 3D interaction is "human-computer interaction in which the user's tasks are performed directly in a 3D spatial context" [Bow05]. This large space of possible interaction has been organized by a taxonomy of 3DUI tasks. The top-level categories are [Bow05]:

- *Selection* is the indication of one or more objects from a set of selectable objects.
- *Manipulation* is the modification of object attributes such as position and orientation. Selection and manipulation are often paired tasks.
- *Travel* is the movement to a new location or along a direction.
- *Wayfinding* is the "cognitive process of defining a path through an environment, using acquiring spatial knowledge, aided by both natural and artificial cues." [Bow05]
- *System control* is the sending of commands to the system. In 2D interfaces, these actions are commonly managed by widgets.
- *Symbolic input* is the communication of symbols to the system such as text and numbers.

With these task categories defined, the research community has been creating techniques to specifically address the needs of each task. These techniques have been researched and evaluated to identify the trade-offs and guidelines of use. The best match of technique to task, given the conditions of the system, can then be determined. Despite these problems in development of 3D interaction techniques, new techniques continue to be created [Win03] through such means as domain specific approaches, decomposition into of smaller tasks [Pou98][Pou99], hybrid techniques and assumption splitting [Pie01].

2.1.1. 3D Interaction Techniques

To understand this dissertation, several techniques need to be understood. These are discussed here in detail and involve the tasks of selection and manipulation of objects as well as travel.

First, the selection and manipulation techniques discussed in this dissertation are:

- Virtual hand techniques are the most easily envisioned selection and manipulation techniques. With these techniques, the user's virtual hand in a 3D

environment is mapped to the user's real hand position. This can be done in a one-to-one mapping or as an arm extension technique. Arm extension techniques scale the user's hand movements to allow the user to reach further into the environment. This scale can be problematic when detailed movements, especially due to tracker jitter, become erratic and hard to control.

- Go-Go [Pou96] is a technique that improves upon arm extension of the virtual hand technique. Go-Go uses a one-to-one ratio scaling function up close and a non-linear scaling function towards the end of the user's reach. This enables detailed interaction in the space around the user, as well as the ability to reach deep into a virtual environment.
- Raycasting selection is the use of a ray attached to the user's hand to indicate objects by pointing. This technique is useful for to the ability of the user to keep their hand in a non-fatiguing position at their side during selections. Also, Raycasting makes selection essentially a 2D task in that the user does not have to specify depth along the ray. This can improve selection speed. Problematically, Raycasting is limited in the manipulation of objects after selection because the object is attached to the ray, making depth manipulation difficult.
- Hand-centered Object Manipulation Extending Raycasting (HOMER) [Bow97] is a hybrid technique that combines the selection abilities of Raycasting with the manipulation abilities of arm extension. In this way, an object selected by Raycasting can then be manipulated by an arm extension technique, which has a scale factor based upon the object's original depth along the ray. This allows objects to be easily selected and manipulated, up close as well as at a depth.
- World-In-Miniature (WIM) [Sto95] is a technique that places the world in the user's palm, like a map, to visualize and manipulate. As opposed to arm extension techniques that scale the hand as it moves into the world, the WIM technique scales the world and then allows the hand to move normally in the scaled WIM. Using the WIM, users can move objects and see the results from both an exocentric as well as an egocentric viewpoint. The WIM also fulfills the travel task by allowing the user to select themselves and manipulate their own position in the WIM. On release, they are animated into the WIM [Pau95].
- Image plane selection is like Raycasting in that a ray extends from the user but instead of the ray attaching to the user's hand, it extends from their eye through something like their fingertip [Pie97]. Using image plane selection, multiple methods of selection are possible like occluding the object with a fingertip to select or placing two fingers around the object, encasing it on the top and bottom, to select. This was called the head-crusher technique. Occlusion selection techniques can have higher fatigue compared to Raycasting because the user's hand has to be held in front of their head for each selection.
- Voodoo Dolls [Pie99] is a hybrid technique that uses image plane selection [Pie97] to select objects and two hands to manipulate objects in the context of their surroundings. In Voodoo Dolls, the user selects an object and then dolls are temporarily created, for the selected object and surrounding objects, and attached to their hand like a WIM. With the other hand, the user manipulates the dolls with the doll manipulations updating their real-world equivalents. This allows the user the benefits of both the WIM technique and the ability to choose the scale of interaction, like HOMER, based upon the context of the selection.
- Scaled and Scrolling WIM (SSWIM) [Win06] is a WIM technique with the addition of scaling and scrolling to the WIM. This enables much larger worlds to exist in

the WIM and still allows detailed interactions by changing scales. The changes to scale were performed using a mouse wheel and moving the virtual hand to the edges of the WIM performed scrolling. The further from the WIM center, the faster the WIM scrolled.

Second, the travel techniques discussed in this dissertation are:

- Wand-based travel is a steering travel technique where users point a wand or similar device in a direction and perform an action, like pushing a button, to travel. The rate of travel is set at development time based upon the requirements of the environment. With steering techniques, the user can stay in one position and move through the world as if traveling in a vehicle.
- An improvement to wand-based travel is velocity control. One method of doing this is to map the distance of the user's arm extending into the environment to their rate of travel. For faster travel, users reach further into the environment and then pull their arm in close for slower, more controlled speeds.
- Gaze-based travel is another steering technique but travel is indicated by the direction of the user's gaze. This technique is often more natural and less disorienting. However, the coupling of control and viewpoint to the user's head removes the ability of the user to look around the environment while they are traveling.
- Target-based techniques are another method of travel but instead of steering to a location, a location is selected and the interface moves the user automatically. This technique requires both the selection of a point and the movement to the point. A point selection can be through any selection technique that can give a point such as arm extension, Go-Go or WIM. The movement to the point is generally through an animation, as immediate updates are disorienting to users [Bow97]. With slow in and slow out [Las87] animations, target-based animations are less jarring to the user.

2.1.2. Toolkit Features and Functionality

The functionality requirements for the 3D domain are quite large. This is attributable to the variety of applications, in various domains, using a variety of hardware and interfaces. These applications could be on highly immersive and realistic applications requiring realistic interactions, to lower fidelity graphics displayed in a web browser with little more interaction than just traveling around the scene. For example, some of the functionality required to develop a 3D environment are:

- Graphics: scenegraph, shadows, animations, data format importing (models, images, animations in models), specialized viewports (desktop monitor, HMD, various aspect ratios), lighting types, text, billboards, fog, radiosity, stereo
- Hardware Support: display hardware, interaction hardware, position trackers, buttons, physical devices, audio (stereo), portability (between computer platforms and the web)
- Performance: support for large scenes, realism, parallel and distributed processing (for computation and rendering), GPU programming and optimization, time-constrained computation, performance testing
- Content: model importing, simulation runs, CAD data, GIS data, reusable components (3D widgets, scripting, animations), 2D interfaces in 3D
- Specialized Functionality: collision detection, computer vision, virtual humans, physics simulation, scripting, networking for remote and distributed interaction
- Development: sockets, message-passing, event-based architectures, shared

memory, memory management, threads
- Development Support: documentation, user community, corporate support, example code, access to source code, specific computer language support (C/C++, Java, python, etc), OO vs. non, testing environments (cave simulator), debugging information and tracing, hooks into existing code

This functionality represents a wide range of varied requirements. As such, few toolkits come close to supporting all these requirements, as there is a tradeoff between: 1) supporting more requirements or 2) supporting the existing requirements well. Because of this, most tools focus on supporting a subset of the functionality well and with time, become more robust. Unfortunately, due to moving targets [Mye99], the underlying technologies change and leave many robust tools supporting obsolete feature sets [Ste08].

2.1.3. 3DUI Domain Tools

There have been many attempts to improve the design and development of 3D environments through the use of toolkits. Many of the tools are event-based architectures with some declarative approaches. Some event-based tools are VR Juggler [Bie01], DIVERSE [Kel02], Studierstube [Sch02], Panda3D [Gos04][Pan] and ARToolkit [Kat00]. Declarative approaches are found in the toolkits of Contigra [Dac01], X3D [X3D] and VRML [VRM]. A rise in game programming has lead to the creation of its own set of tools such as OGRE [OGR], Virtools [VIR] and Torque [Tor]. Other approaches to development are to write code around a scenegraph such as OpenSG (http://opensg.vrsource.org/trac) and OpenScenegraph (http://www.openscenegraph.org/projects/osg). VRPN [Tay01] has been successful at abstracting away device handling. More exhaustive lists can be found here [Ste08][Ray08]. Complicating the matter is the number of features and amount of functionality that must be provided to be useful in development.

As extant work has documented, a disturbing trend among 3D toolkits is their tendency to expire [Ste08]. For example, in a review of thirteen 3D toolkits reviewed in 1995 [Tay95], only one, Alice, is still in active development. Even then, Alice's focus has shifted to teaching programming, not 3D development. To address this, standardization has been attempted to improve portability between tools as well as enable features and functionality to be shared across toolkits. While many of the tools are similar in their approach, some of these have distinct approaches (see 2.4. below).

2.1.4. 3D Interface Development Difficulties

As shown in the extant literature, 3D interface development is not easy [Gre91][Her94][Dac06][Dac07][Lat08][Sha08][Mye91][Mye93][Ste08]. The Several characteristics that make design and development particularly problematic include: high bandwidth I/O, many Degrees of Freedom, real-time response, continuous response and feedback and probabilistic input.

As such, there is a mismatch of architecture and mindset between 3D interfaces and more typical 2D interfaces. As previous studies have shown, 3D interfaces need to be "truly interactive" and "the user's set of valid interaction techniques should not be restricted, no matter what state the application is in" [Gre90]. 3D interfaces are parallel interrelated dialogs, continuous as well as discrete, probabilistic, real-time and passive [Jac99]. These characteristics violate many assumptions of the traditional interaction models of interface development, which have characteristics similar to a compiler where actions are processed in a single stream of discrete

tokens. Most interfaces are implemented in event-based architectures that problematically spread functionality throughout a series of callback handlers, often resembling a spaghetti of callbacks [Mye91].

Additionally, many other non-architectural problems make 3D interface development difficult:

- 3D interfaces lack standard devices and displays and can be difficult to design for and maintain. The use of the devices themselves can be problematic as cables can get in the way of the user, they can be bulky and ill-fitting.
- These same devices make accessing the development environment problematic while immersed. This impacts the ability to debug code as well as reproduce errors.
- Not all information flows through the system. Near-field haptics and real-world objects exist completely outside the interface and have to be tracked.
- 3D coordinate systems are much more complicated to develop than 2D. Matrix transformations are difficult.
- Rendering can involve the simulation of visual, auditory, haptic and other sensory information, which can be difficult to do artistically and computationally.
- It is difficult to realize the assumptions of an interface when it is built. Reusing the implementation in a new situation can have small differences in requirements that lead to major implementation differences and impede reuse. Having access to deeper functionality, "hooks" [Pol06], assists reuse but it is difficult to know a priori what needs to be a hook.
- With parallel interaction, separate functionality operates on the same data, such as the location or color of objects, which ultimately lead to incorrect and unexplainable behaviors.
- Iteration is required to implement a good 3DUI and iteration is time consuming.

The human is at the core of 3D interfaces where in traditional interfaces, the human acts through widgets. This leads to the following:

- Much more contextual and human information is relevant to the interface, which is difficult to track and interpret. This includes body sizes [Pou97], fatigue, preference [Win05c], cognitive functioning, and so forth.
- In many cases, an interaction idea fails once the interface is "experienced," and this cannot be realized beforehand. This failure occurs after a long development phase.
- It is hard to know if something is wrong in a 3D environment because the user adapts to the interface. This can be advantageous as well for designing around technical limitations [Bur06][Raz02][Bow01].
- Poorly implemented systems have more consequences than just being hard to use. For example, this may be associated with fatigue, nausea and impaired functioning [Sta98], and the affects of these may be long-term, lasting up to 24 hours [Bla89].

2.1.5. Standard Reusable Components

By creating reusable components, such as widgets and interaction techniques, the components can be designed and built once and then reused as needed. The hope is

that 3D systems will begin to behave similarly and development will be simpler because of a standard set of components. CONTIGRA [Dac01], as an approach to declarative 3DUIs, has looked at this in depth, creating a taxonomy of 3D widgets. SVE [Kes00] has looked into the creation of reusable widgets and X3D is currently working on a standard for 3D interaction widgets [Pol06].

To date, reuse across toolkit barriers is limited. Creating standard reusable components continues to be problematic for several reasons. First, the exploration of the 3DUI design space is still on going, and it is not yet known what needs to be standardized. Early standardization runs the risk of locking-in a path of least resistance [Mye99] to bad 3DUIs. Second, standardization can limit creativity, making new or optimized interfaces hard to build. Third, standardized components create components hard to optimize for specific tasks. Fourth, access to the internal data of a standard is problematic [Pol06].

The forth problem, access to the internals, exists not just in standards but in existing tools for 3DUI design and development. The access to internal data structures as well as flow is required for the extension and reuse of existing 3DUIs. To date, existing tools have handled this in different ways. This notion was discussed as "hooks" [Pol06] to internal functionality in regards to X3D. In Panda3D [Pan], the creation of events and registering of callback functions allows access to the internals. Dwarf [Bau01] uses Corba objects to encapsulate its functionality as services that externalize "Needs" and "Abilities" through an XML format. Architecturally, these are similar approaches.

Despite these problems, standardization and engineering aspects of 3D toolkits is an active area of interest. This can be seen by a series of workshops around reusability and standardization at IEEE VR [Dac07][Lat08][Sha08], standardization of content around X3D, and a recent workshop at CHI 2008 [Sha08]. Notably, the latter had the largest attendance of any of the CHI workshops.

2.1.6. 3D Interaction Design and Development Summary

This section has reported background information and the status of 3D interaction design and development. The domain presents many problems to developers and requires a large amount of functionality. As such, many toolkits have approached the development of 3DUIs and have done so in many varied ways. As time and technology advance, these toolkits have expired and with a lack of reusable components, so have the systems developed in them. In attempting to address these problems, another toolkit attempt does not seem promising. Portability and extensibility seems to be the important factors.

2.2. Background: Software Design and Development

Software design and development is becoming more important for 3D environments. As discussed in the previous section, the creation of standard reusable components is important. This can be achieved by good decomposition and modular design. In addition, 3D design and development can be improved by several existing software engineering approaches.

2.2.1. Software Modularity and Decomposition

Software systems are complex. Methods to reduce this complexity to more manageable problem sizes continue to be a software engineering goal. These "divide-and-conqueror" approaches create smaller and smaller modules until

ultimately they are solvable. For example, it has been said that "the only way to cope with control of systems that are complex in terms of large numbers of information sources and devices for basic control actions is to structure the situation and thereby transfer the problem to a level with less resolution" [Ras85]. The exact way of doing this, which results in good features of software systems, is problematic.

2.2.1.1. Meyer's Criteria

According to Meyer [Mey97a], five criteria of modular design are decomposability, composability, understandability, continuity and protection.

A software system should allow for decomposability, or the ability to break a larger problem into smaller, more manageable problems. Also called a "divide and conquer" strategy, these problems can then be solved or can undergo more decomposition until they can. Two desired criteria of decompositions are high cohesion and low coupling. In high cohesion, internally the module is very interconnected and its parts highly related. In low coupling, the modules are only loosely dependent upon each other, which reduces the impact of change. More on decomposition in the following section (see 2.2.1.2).

A software system should allow for composability, or the ability to create modules of higher-level functionality or new types of functionality from the existing modules. This supports the process of reusability, which seeks to reduce costs and time of development.

A software system should allow for understandability, or the creation of a system that is quickly comprehended. This includes the ease of understanding a single module as well as minimizing the number of modules to understand. Module structure should match the problem being described and should be self-documenting, with its description paired with its implementation. With understandability comes reduced maintenance costs, more reusability and easier development. This is embodied in the philosophy of Literate Programming [Knu92].

A software system should allow for continuity, or the creation of a system where change impacts as few of its modules as possible. Change is inevitable in software development and maintenance, so modules should be designed for it. The notion of information hiding [Par72] selects what functionality in a module is externalized and what is internalized, or hidden. The externalized functionality then becomes the interface for the module, where modules then communicate through well-defined interfaces. By keeping well defined interfaces which are resistant to change, with information hidden, modules remain continuous.

A software system should allow for protection, or the confining of errors to a minimal set of modules. This includes errors coming from input or run-time behavior. This is especially important in module reuse where new systems can create unexpected cases of use for a module. Achieving protection comes from explicit interfaces as well as limiting communication and data exchange between modules. This limits the points of failure as well as making the possible points of failure clearly defined.

2.2.1.2. Decomposition

> Being abstract is something profoundly different from being vague... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> - E. Dijkstra

Decomposition, or the separation of concerns [Dij74], is the design and development

process that separates functionality of a system into smaller parts with as little overlap as possible. Decomposition occurs through the following: 1) select a piece of the problem initially, the whole problem, 2) determine the components in this piece using a design paradigm, 3) describe the components interactions and 4) repeat steps 1 through 3 until decomposition is no longer needed.

These approaches are generally thought of in two ways, functional and object decompositions. A functional decomposition proceeds from a generic description to a refinement of smaller and more specific steps [Wir71]. The result is a description of the desired functionality. An object decomposition is focused on the things in the system. For example, the object moto is, "Ask not first what the system does: Ask what it does it to! [Mey97a]." Both have their own problems.

The first problem with functional decomposition is finding the main purpose or top-most functionality in a system. Compounding this is the fact that often the top-most functionality in a system is fleeting. Good systems have new requirements added to them that change their focus. A system designed around a functional decomposition is immediately broken if the top-most idea in the system is no longer top. As Meyer stated, "real systems have no top" [Mey97a]. Parnas warned against this approach for the same reason [Par72].

A second problem with functional decomposition is a premature commitment to ordering [Mey97a]. The specification of a problem as a list of steps artificially locks the system into that order of steps. When the system and requirements change, the ordering of the steps is built into the system and is too inflexible to be changed.

An object decomposition is focused on the things or nouns in the system over the relationships between them. The purpose of identifying the objects first is that objects tend to remain more stable in a system. As the system changes over time, the objects have a better chance of remaining in the system whereas the functionality between them might change. As such, they make better candidates for reuse and composability.

A third decomposition exists in addition to the functional and object decompositions, which is similar to an object decomposition. An aspect decomposition [Kic96] addresses software concerns that are crosscutting through a system. Whereas an object is an encapsulated whole of a single concern in one location, an aspect is an encapsulated whole of a single concern which is weaved throughout a system in many objects. In this way it extends good modular design to the functionality spread through a system. An oft used example for an aspect decomposition is the placement of logging code, which is spread throughout a system, but is often repeated and has its own data and functionality needing encapsulation.

All three decomposition approaches are important. Where functional decomposition focuses on the hierarchical functionality and relationships in the system, the object-oriented and aspect-oriented decomposition focuses on the objects or things. All are required for a real system, so the approaches to decomposition must be supported. Decomposition improvements have focused on creating more dimensions for a system to decompose [Tar99].

The two characteristics of encapsulation and abstraction are important to the modules created in these decompositions. Encapsulation is important to decomposition, as it is the process of internalizing and protecting a module's features. Here, features refer to the module's concerns, functionality, design decisions, requirements, data, and so forth. This protection is achieved through information hiding means or the creation of an interface to the module. In this way,

both functionality and data access can be controlled at a module's interface. As the system changes, the interfaces remain constant, which limits the effect of change in the system.

Abstraction is also useful in decomposition, and focuses on the removal of details from a problem to create a simpler representation -- one better suited for current use. A good abstraction is the most abstract representation that still allows the design and development problems to be represented. In computer science, abstraction has been used to refer to the creation of abstract data types that separate the properties of the data from their concrete representation. Additionally in objects, behavior and data can be abstracted and substituted through polymorphism. Abstraction is a powerful tool in designing systems. As decomposition creates simpler problems to solve, abstraction can simplify each problem as it's being decomposed.

A problem exists with abstraction and that is that solving a problem as an abstract solution requires a higher level of knowledge [Hoc91]. So, asking for abstraction in design before the problem is designed can be problematic; better to wait for a solution before creating an abstract solution. Additionally, novices have problems understanding abstraction [Men91].

2.2.2. Software Engineering Approaches

Below, several approaches in software engineering are discussed in relation to this work.

2.2.2.1. Event-Based Programming

Event-based programming is concerned with the interaction between two modules: a listener that registers and responds to the event of a broadcaster. Whereas procedural programming specifies a series of instructions to perform, event-based programming involves waiting for an event to control the computation. The paradigm is powerful because the broadcaster does not have to concern itself with how its events are responded to or how the response is managed. This results in a loosely coupled system with components that are easily replaceable and generic. Reuse proliferates under these conditions.

Event-based programming is a popular architecture for graphical interfaces. They have been used in many languages such as Visual Basic and Java. The ability to register an event handler function with a broadcaster, such as a button or slider widget, is a good match to the out-of-order interaction in which users tend to work. Additionally, it is easy to create good events in such systems because of the naturalness of naming a click, push, drag or scroll event.

There are several problems with event-based systems [Mye91]. First is the loss of control. It becomes problematic to ensure the order of computation in such a system and different orderings can create different outcomes. Specifications in the event system can order the events but can become confusing. Second, the system quickly becomes complex as more listeners and broadcasters are added. There is a lack of modularity and encapsulation of these listeners and broadcasters. Third, the creation of a good set of events has to satisfy several conflicting requirements: cover the important ideas of the system, be few enough to be memorable and fine-grained enough to allow developers detailed control. Additionally, each event must be maintained in a system as every modification to a module can break an event. Fourth, events do not take into account the system context. The temporary conditions in a system [Jac99] play a role in the behavior for an event when it occurs yet this is largely ignored by most systems.

A similar architecture is publish/subscribe. It creates an even further decoupling by registering not with events on objects but on classes of information. In this way, publishers and subscribers can come and go and scalability is improved. For example, DIVE [Fré04] is a VE system with this type of architecture.

Event based architectures have been the typical implementation style for the first 3DUIs. However, as more complicated interaction has been required for 3D environments, this architecture has become overwhelmed.

2.2.2.2. Dataflow Programming

Dataflow languages originally focused on the potential for massive parallelism but more recent trends are concerned with the software engineering benefits [Joh04], especially dataflow visual programming languages (see below). While dataflow does address important issues of processing order and execution, it does so in a data-centric paradigm of data tokens passing through a network of connections, a very different development paradigm for developers. Flow-based programming [Mor94] dichotomizes the language into its computation and coordination features, instead focusing on data flowing between black boxes of code with care given to the ordering of events in such systems. This is still a data-centric approach to development however. It does not present any additional access to internals and flow decomposition has been noted as non-optimal for maintainability and development [Par72]. Dataflow can also lead to very confusing systems due to the number of boxes and flows.

Dataflow programming, especially flow-based programming, is a good representation for the continuous relationships in 3DUIs. Additionally, the parallel and discrete computation benefits of dataflow, required for many 3DUIs, is important. One example of a flow-based VR toolkit is FlowVR [All04], which makes great use of the computational benefits (see 2.6.1. below).

2.2.2.3. Visual Programming Languages

Visual programming languages attempt to increase the understandability of code through the use of a visual representation. Often, the visual languages are dataflow languages [Joh04]. The idea behind the visual representation is that in addition to being faster to absorb, the visual representation is actually a closer semantic match to the indented functionality of the programmer [Gre96]. Often such languages are aimed at specific programming domains [Lab] or for end-user programming [Joh04]. Despite this, it is unclear if visual is better than text [Whi97]. Visual programming languages have problems with change (viscosity) [Gre96] and their representational density is unclear as compared to text programming languages (the Deutsch Limit [Ram83]).

Despite this, the potential for a visual programming language for 3DUI is high. The domain of 3DUIs itself is very visual, making it arguably a better match to a visual representation. Many of the difficulties of 3DUI involve 3D behaviors that are easy for developers to perceive but difficult to program, such as the vector and matrix math of 3D graphics. The toolkit Alice [Con00] (see 2.4.2 below) demonstrates how visual representations can improve its usability with objects used in the environment appearing as the user is programming them. Both the PMIW [Jac99] and InTml [Fig02] tools have a visual representation as well. A noticeable difference between the use of visual programming in Alice and PMIW and InTml is that Alice shows the 3D environment while PMIW and InTml use visual representations for the behavior of

the 3D environment.

2.2.2.4. Unified Modeling Language (UML)

UML (http://www.uml.org) is a standardized object specification language for systems that has quickly become the industry standard. Its purpose is to assist in the representation and understanding of systems by the use of visual representations. To this end, it incorporates multiple diagram types to represent different information in a system. An active area is the translation of such diagrams into source code [Har97][Tog].

There have been many criticisms of UML [Hen05][Fra06][Hai06][Tho03], some satirical [Mey97][Bel05]. First, the specification of UML is bloated [Ast04]. Over time, the specification has grown and redundancies and constructs of little use have entered the representation. Second, UML is a model that quickly becomes out of sync with the implementation. To this end, the generation of source code from the model has addressed the problem but the bloat of the specification makes this transformation problematic [Ast04]. Third, UML tries to do too much and, in the end, becomes too generic.

UML is important for the visual aspects and broad exposure of its diagrams among software engineers. Whether or not UML is the appropriate representation for 3DUIs, its utility in representing the design and development of systems at a higher level of abstraction is considerable.

2.2.2.5. User-Interface Management Systems (UIMS)

The intent of UIMSs [Kas82] is to allow interfaces to be abstractly specified and automatically transformed to match the properties of the target platform. Popular for the ability to abstract low-level details, allowing developers to focus on functionality, the UIMS systems never created interfaces optimal for users [Kas82]. The interfaces created were unpredictable, with heuristics and rules being hard to specify and control. Often, this led to wild interfaces or those that had to be hand-tuned for each platform anyway. In the past, the standardization of the desktop platform removed the need for UIMS for desktop UIs [Mye99]. With a lack of standard interaction in 3DUIs and potentially an inability to standardize them [Bow05], UIMSs may become a promising technology once again [Sha08].

UIMSs are an important idea but have many limitations. The seemingly randomness of the generated interfaces is problematic but with work, UIMSs declared for the tasks of 3DUIs could be promising.

2.2.2.6. Natural Programming

Natural Programming is a human-centric approach to the design of tools to support development and debugging tasks. The inspiration for the work comes from applying the empirical studies of programmers and HCI methodologies to improve the development process. Typically, this is in the support of end-user programming tasks. The main projects have been Hands [Pan02] and Whyline [Ko04] with early inspiration from Alice [Con00]. Some of the key ideas from the empirical studies of programmers that Natural Programming uses are regarding control structures in looping, a focus on the closeness of mapping between the task and program and the effectiveness of visual versus text representations of programming [Pan96].

The focus on the human in the design and development task of 3DUIs is an important consideration.

2.2.2.7. Feature Interaction

The feature interaction problem has grown out of the telecommunications industry where the interaction among features has lead to significant cost, overrun and complexity. An example from [Zav08] is, "Bob has Call Forwarding, and is forwarding all calls to Carol. Carol has Do Not Disturb enabled. Alice calls Bob, the call is forwarded to Carol, and Carol's phone rings, because Do Not Disturb is not applied to a forwarded call."

A feature is an additional bit of functionality that is composed with existing features of a system, modifying each other, to create a new system behavior. The feature interaction problem stems from this, where features interact in unpredictable ways. This leads to complexity, bugs and ultimately extra costs and time in development. One aspect of the feature interaction problems is that in some cases, the "correct" behavior of the system is not a non-failure but is defined by the user of the system. To address this problem, research looks to detect, prevent and resolve these interactions. For an overview of the field, see [Cal03] and the conference Feature Interactions in Software and Communication Systems.

There are many approaches to the feature interaction problem that were grouped in [Cal03] into the three categories of software engineering, formal methods and on-line techniques. The software engineering attempts try to avoid interactions by good software engineering practices or introduce what they call process models to filter out unlikely combinations of features or reduce the features to safe combinations. Formal methods have been used to model the interaction between the features to prove specific requirements. These methods generally require the specification of the features in a special language and assume that the implementation correctly maps to the specification. On-line techniques check for interactions in real systems in the natural system environment and try to resolve issues on-the-fly.

Feature Interaction has many of the same design and development issues as 3DUIs. One example is a 3D interface with an arm extension selection technique that has a travel technique added to it. This 3DUI now has a feature interaction problem because both techniques use the same button for selection and travel. A second example, another 3D interface with a floating menu for system control tasks can visually interfere with a pen and tablet technique as the floating menus cover the tablet. A third example 3D interface, the audio output for one technique could interfere with another technique's audio. In 3DUIs, the potential for features to interact is high and not just in detectable discrete events like a button but in system IO, human perception, timing, ad infinitum. The techniques used in Feature Interaction to detect, prevent and resolve these interactions are applicable to 3D interface development.

2.2.2.8. Statecharts

Statecharts [Har98] are used to create visual and executable descriptions of reactive system in the form of a hierarchical state machine. There are many advantages to Statecharts. First, state machine modeling is familiar to many developers, increasing its potential uptake. As an example of this, a state machine representation has been included in the specification for UML 2.0. Second, Statecharts is a good way to describe behavior at a level detailed enough for simulation before development. Third, Statecharts improves on the representation of complexity and concurrency. Typical, the number of states increases rapidly and concurrently executing processes can be difficult to describe. With Statecharts, the complexity is reduced by a hierarchical decomposition of state. Additionally, Statecharts includes formalisms in

its representation for concurrency.

In its original use in modeling embedded systems, it has been successful at addressing the domain's problems —including timing issues, simulation of safety critical systems and complexity [Har98]. Statechart's creator even presents it as a method of overcoming two pessimistic arguments on the future of software engineering [Har92] argued by Brooks [Bro87] and Parnas [Par85].

Despite this, Statecharts have several problems, many covered in detail in [von94]. First, there are issues regarding the ordering of events in Statecharts. Statecharts handles events in terms of steps, where a certain number of events are handled in a time-step. This non-deterministic method of event handling creates problems such as the same set of events having multiple transition orderings. For example, if in a state A with valid transitions for events a and b, then if both a and b events exist, which transition should be taken? Additionally, if both an internal transition and a higher-level transition can both use the same event, which transition should be taken first? Statecharts, because of its utility and due to these problems, has many variants with differences in its syntax and event handling. This makes understanding the Statecharts diagrams difficult. In Chasm, the creation of rules to explicitly order the events removes this problem. A second problem is the global scope of the events in Statecharts. As such, Statecharts allows additional functionality to break existing functionality. As an example, a transition in Statecharts that creates a symbol b expects that symbol to be used locally. Unknown by the developer is that another part of the Statecharts system uses that event and causes the system to enter an incorrect state. To avoid this, Chasm keeps the scope of actions local. Causal statements map global state changes to actions internal to a concept. These first two problems, issues regarding the ordering events and global scope of events, mean that Statecharts are not compositional. If they were, Statecharts would be more easily verifiable and reusable.

Third, the notation of Statecharts, though it begins as a simple hierarchical state machine, becomes overly complex due to the many additions to the notation. For instance, the previously mentioned problem of two possible transitions in a state machine, both of which can be taken because both events exist, was rectified by requiring the developer to provide negated triggers which state which transition to take in the event that more than one event is possible. Considering every event is possible due to global scope, these negated triggers can be time consuming to develop. As Chasm orders events, it completely avoids the need for negated triggers. Another complexity of Statecharts is the need to retain the state of an internal state. Statecharts handles this with a developer's need to specify a history state that developers must create and manage. In Chasm as well as Virtual Finite State Machines [Wag06], each state machine is separate. So, by default all history is maintained. Other complexities in Statecharts include conditions, selections and the creation of its own programming language to improve Statecharts' expressiveness. Chasm avoids this by relying on existing languages familiar to the developer to express the needed conditional and selection functionality.

There are several advantages of using Statecharts in 3DUIs. Statecharts have characteristics useful to 3D interface design and development, including: continuous interaction with the environment, responding to interrupts, time-constrained requirements for responding to inputs, different operations to inputs based upon system state and handling of concurrently executing processes. However, since Statecharts evolved from a domain with a focus on timing and safety, many of the tradeoffs in Statecharts' representation are not applicable to 3DUIs. Complexity, concurrency and reuse are greater issues while the requirement for model checking

is lessened. Additionally, 3DUIs will use much more functionality external to the architecture in the form of toolkits which does not fit into the typical design and representation of Statecharts.

2.2.3. Software Design and Development Summary

Software engineering will play a larger role in the future of 3DUIs due to the need to manage complexity and build reusable systems. The event-based architecture, successful in 2D interfaces and used in the creation of early 3DUIs, has reached its limits. Architectures will be applied that focus on complexity and make it more manageable to the 3DUI developer community. This can be seen in the related work sections below.

2.3. Background: Forms of System Evaluation

Evaluation, especially of software systems, is a difficult process. Where several methods exist for the evaluation of 2D interfaces [Nei94] and 3D interfaces [Bow02][Hix99] these are not directly applicable to the evaluation of software systems.

2.3.1. Evaluation in Software Engineering

The types of evaluation performed on software engineering systems have been reported in multiple surveys [Sjo05][Tic95][Zel97][Gla02]. In general, software engineering does a poor job in evaluating its claims [Gla95]. The rationale for this is characterized by [Gla95] as "conceive and idea, analyze the idea, advocate the idea." Evaluation was lost in these steps, however, likely because of the cost and time involved in evaluation [Gla95]. An alternate rational was that software engineering research is often "developing a 'first of its kind' device or process so comparable data is not easily obtainable" [Zel97].

Multiple papers have tried to characterize evaluation in software engineering. As previous work has noted, the lack of evaluation is alarming [Tic95]. In 400 papers surveyed, 50% of the software engineering papers had no validation and only 20% devote at least a fifth of the paper to the validation. Of the evaluation used, software evaluation is very limited in the methods, with the majority being some form of conceptual analysis [Gla02]. The use of humans or case studies was relatively small. Said [Gla02], "There was very little use of the 'social' level of analysis categories." The lack of controlled experimentation[1] in software engineering was found to be low [Sjø05]. In a study of 5,453 articles from 1993-2002, only 103 articles had controlled experiments (1.9%). Thus, evaluation in software engineering is difficult and there are few guidelines for good evaluation techniques, especially for human-oriented evaluations [Zel97].

A survey of evaluation in software systems is presented in [Sha01]. This presented a list of five techniques used in the validation of software systems: persuasion, implementation, evaluation, analysis and experience. Shaw notes that early systems were originally supported by persuasion, but that as time passed, better methods were used. Shaw also warns of two inadequacies in validation. First, presenting a system without any comparison against the practice it improves and second, as

1 The term controlled experiment was defined as, "A randomized experiment or quasi-experiment in which individuals or teams (the experimental units) conduct one ore more software engineering tasks for the sake of comparing different populations, processes, methods, techniques, languages or tools (the treatments)."

claiming success in a system after comparison to toy problems.

2.3.2. Cognitive Dimensions of Notations

The Cognitive Dimensions of Notations [Gre89], created by Thomas Green, is a general method to evaluate a notation or even a programming language. Each dimension represents a single idea important to the utility of a good notation. When taken together, the guidelines create a high-level analysis of the process and activities of the notation. There are 14 original dimensions and others have been added since. These dimensions have been built into a questionnaire [Bla00] for use as a self-report or in a semi-structured interview. Some of the most important dimensions follow:

- Viscosity – How resistant to change is the notation? What are the costs of making a change in the representation?
- Hidden Dependencies – A relationship exists between two components but for at least one of them, the relationship is not visible.
- Premature Commitment – How much does the notation force the order of decisions such that decisions are made before the proper information is available to make the decision?
- Abstraction – What are the mechanisms in the notation for creating abstractions?
- Closeness of Mapping – How closely does the notation map to the domain being represented?
- Consistency – Are similar semantics expressed in a similar way?
- Diffuseness – How verbose is the notation?
- Error-proneness – What parts of the notation invite mistakes in its use?
- Hard mental operations – What parts of the notation place a high demand on the cognitive resources of the notation's user?
- Progressive evaluation – Can the existing work in the notation be checked as the notation is being used?
- Provisionality – What ability does the notation have for sketching or avoiding precision when the direction to proceed is unclear?
- Role-expressiveness – When reading the notation, is it easy to tell what each part of the notation is for?

Because of its high-level analysis, Cognitive Dimensions is a good method of evaluating a notation for qualitative impressions.

Cognitive Dimensions are useful to systems analysis, creating a set of common dimensions to evaluate a system and through repeated evaluations; meta-analysis of systems can be performed. The analysis is also able to be performed quickly and can lead to more detailed analysis. Additionally, with the creation of the questionnaire, the evaluations can be used to quickly gather a large number of responses from a community.

2.3.3. Ko's Framework for Studying Errors

In [Ko05][Ko03], Ko and Myers create a methodology and framework for the analysis of programming errors. Their framework focuses on three cognitive problems (knowledge, attentional and strategic) that can lead to breakdowns in

three activities (specification, implementation and debugging). These breakdowns are formed of a cognitive problem, an action and an artifact. Action and artifact types are specific to the activity type. Using this approach, they were able to analyze the errors participants made while programming and categorize them using this framework. Their methodology is (p 60):

1. Design an appropriate programming task.

2. Observe and record suitably experienced programmers working on the task, using think-aloud methodology [Eri84] to capture their decisions and reasoning.

3. Use the recordings to reconstruct chains of cognitive breakdowns by working backwards from programmers' software errors to their causes.

4. Analyze the resulting set of chains of breakdowns for patterns and relationships.

Their experiments were specifically on 3D worlds. The first was a study of three participants implementing projects for the "Building Virtual Worlds" course at Carnegie Mellon. The participants coded for 245, 110 and 50 minutes each and had six weeks of experience with Alice. The second study was the implementation of a simplified Pac-Man game. They programmed for 95, 90, 215 and 90 minutes each and had a 15 minute tutorial on how the use Alice before being handed the specifications for the application. Their method of analysis was Contextual Inquiry [Bey98] and led to the identification of 159 breakdowns by the participants. This investigation led to the creation of the Whyline [Ko04] application because they noticed "...programmers could easily verbalize their why questions during the think-aloud, [and thus] why not directly support an interface for asking these questions?" [47, p 77]. It is interesting their use of language in a tool to debug but not in a tool to design.

Ko's investigation of errors in programming can serve as a model for testing a programming system. Ko has reported on his experience with Contextual Inquiry, his observation of participants, the selection of suitable tasks, the selection of suitable participants and his experimental design. The task was implemented using Alice, a 3D environment toolkit. Unfortunately, the task of implementing Pac-Man was not 3D interaction so does not have all the domain problems associated with 3D development, making this approach problematic for this investigation.

2.3.4. Grounded Theory

Grounded Theory [Gla67] is an inductive approach, where ideas and theories are produced and supported through empirical methods. This emic approach to theory building is concerned with the process of the theory building. Grounded Theory's claim is that good process, in the collection, elicitation and analysis of data, will create good theory and thus always fit at least one set of data (the data it was created from). The typical domains are text, human behaviors or diaries. There are three approaches to grounded theory coding: Open Coding, Axial Coding and Selective Coding. Open Coding deals with the identifying and categorizing of data in a broad sense. Axial coding deals with relating codes to each other for causal relationships and selective coding deals with selecting one category and relating everything to it.

Grounded Theory's ability to create a theory from data, data created by the above evaluation approaches, is useful. It does not give insight into the generation or collection of the data, only the formation of theories from the data.

One useful technique in the organization of this data is affinity diagramming. It is

useful for finding relationships between collected data. The process is typically to post that data items on cards and constantly move and reorder the data to generate connections between data and themes. It is often used to work with data generated during a Contextual Inquiry [Bey98].

This work uses a Grounded theory approach in Chapter 3 to gather information from developers. It helped to guide the type of data collected and the collection methods. Additionally, its philosophy of extracting theory from data is a good exploratory approach when attempting to start with no preconceptions.

2.4. Related Work: General 3DUI Tools

The following tools are significant in their approach to 3DUI design and development. Virtools is an inclusive approach to the design and development of 3D environments. Alice has gone to great lengths to understand how developers think about 3DUIs. Its match to developer thinking has created a 3D development environment that is simple enough that it can be used to teach programming. X3D has a rich content specification that is portable between any of the X3D players. Where in the past, portability meant a loss of detail and power; this is no longer the case with X3D.

2.4.1. Virtools

Virtools (http://www.virtools.com) is a product of Dassault Systémes for 3D development. Its products have been used to create large-scale 3D applications for academics and industry and an active user group is game developers. Virtools includes a fully integrated development environment and exports to multiple forms, either a player or executables for different platforms. Virtools also imports most 3D formats and comes with support for virtual characters, collisions detection, lighting, shaders and other useful development features. One of the more interesting features is its Virtools Scripting Language that incorporates drag and drop scripting in a Virtools application as well as a debugging environment to assist developers.

Virtools is a complete platform for 3DUI development. Everything is included and the company updates and upgrades the platform for new requirements and functionality of the community. Interesting in Virtools is its ability to script predefined behaviors and reuse these between projects. Its integrated development environment, drag and drop functionality of scripts and portability across platforms make it a useful tool to 3DUI developers. Unfortunately its proprietary representation means that interfacing with Virtools will be problematic.

This work is concerned with the ability to easily create behaviors for many uses and reuses. Not just simple interaction but customizable interaction for optimized performance. This makes Virtools limited as a platform to build on, but with features to aspire to.

2.4.2. Alice

The toolkit Alice [Con00] was created for the design and development of 3D environments. Its design was optimized to match how developers think about 3DUIs in the hope that this would improve the usability of the toolkit. Their approach was to observe hundreds of Alice developers in hundreds of informal sessions over the course of four years. Their intent was "...to approach 3D graphics research not as a questions of rendering speed, but as one of authoring and pedagogy."

As a result, they rectified many mismatches between their toolkit and developer thinking. A large focus was on “controlled exposure to power” and language use.

With this power, developers could use commands simply and with experience, could increase the complexity of the arguments sent to a command. Their study of language use led to an improved API. The language of transformation were modified with the "death of XYZ" and used more natural terms. As an example, the "scale" command was replaced by "resize" and "translate" was replaced by "move." Additionally, commands such as "move up 1 meter" would move 1 meter along the Z-axis if Z was up in the world. Alice also introduced useful transformations such as "Place," "PointAt," "StandUp," "Nudge," "AlignWith" and "Pan".

Like Alice, this work seeks to understand how developers think to create a better 3DUI development tool. Because Alice is focused on teaching programming skills, it only focuses on better terms without seeking to create a representation that is a closer match syntactically. Alice also has an all-inclusive development environment, however, that isn't its main advantage.

Alice has gone to great lengths to understand how developers think about 3DUIs. Its match to developer thinking has created a 3D development environment that is simple enough that it can be used to teach programming. However, its applicability to large development is limited. Though it has terms that match to developer thinking, its architecture is still an event-based system, with all the problems of such an approach.

The other systems reviewed would do well to include the lessons learned in Alice's representation.

2.4.3. eXtensible 3D (X3D)

Extensible 3D (X3D) [X3D] and its predecessor Virtual Reality Markup Language (VRML) [VRM] are standardized file formats for the exchange of 3D information. Both are standards of the Web3D consortium and X3D follows the XML format, enabling its representation to be manipulatable by many XML tools. The content of VRML was mostly pure 3D vector graphics but it also specified animations in later versions. X3D, as the successor to VRML, has added audio and video data, interaction, scripting and networking information. Both formats are designed to transmit information on the web and are played in a browser, usually a plug-in to a web browser. They are also both declarative approaches to development where geometry and interaction are left up to the player to implement. This has the potential to simplify development but also can be limiting. To remedy this, X3D allows for scripting languages to be embedded.

X3D has an active community that is constantly adding to and improving its representation. Because these technologies are standards, the 3DUIs built with them will have longer lifetimes, run on more platforms, and have the potential for greater reuse.

X3D has a rich content specification that is portable between any of the X3D players. Where in the past, portability meant a loss of detail and power, this is no longer the case with X3D. Because X3D can be hooked into, many of the systems discussed below are using X3D as a content specification base, which is then manipulated. In this way, X3D content is transferable and tools, which manipulate its representation, are transferable to any X3D content. This split, with X3D as the model, the X3D player as the view and another development approach on top of X3D for the controller, fits nicely with the Model View Controller paradigm. This paradigm is popular and well knows to developers due to its heavy use in web design.

2.5. Related Work: 3DUI Specification

Three tools mentioned below approach the same problem of portability between toolkits with different approaches. IFFI, InTml and Contigra all look to provide a portal 3DUI specification, which is transferable between toolkits. IFFI and Contigra both take the approach that interfaces can be written to a generic specification and the specification transferred. InTml takes an alternate approach and that is to specify how data and information flow between elements in the system. In this way, it extends potentially beyond just an abstract interface specification.

2.5.1. Interaction Framework For Innovation (IFFI)

IFFI [Ray07][Ray08] is a framework focused on the problem of a lack of 3D interaction technique portability, specifically portability between toolkits. As 3D interaction techniques have been designed and developed, the ability to reproduce the interaction technique in a separate system requires the complete reimplementation of the technique. To address this, IFFI is a middleware layer between the toolkit and the application code. Using IFFI, the interaction techniques are written to IFFI's API for 3D interaction, with IFFI interfacing to the implementing toolkit. In this way, IFFI is following the Façade pattern with multiple Adapter patterns for each toolkit. The approach is similar to the ODBC project that attempts to be a common API to databases.

The advantage of IFFI's approach is that one interaction technique, written in IFFI is transferable in terms of code, to new systems and new toolkits. This overcomes a major hurdle regarding both technique and toolkit portability.

The disadvantages to the approach are that a standard API tends to become the lowest common denominator for the supported toolkits. Each new feature that is absorbed into IFFI will have to be recreated in each adapter for each toolkit in order to maintain its toolkit invariance. This becomes especially important for when a toolkit doesn't support specific functionality so a common interaction technique becomes unavailable or only available for only a subset of toolkits, which contain the required functionality. Keeping the API small avoids this but leads to lack of expressive power. An alternate approach is to write as much of the technique in IFFI and then write toolkit specific code however this limits the techniques cross-toolkit portability.

A final disadvantage is the performance of the adapters for each toolkit. Where developers can optimize an implementation over the set of all called functionality, a call to an adapter either has to make each operation atomic, at a cost of performance, or expose the commit action of the functionality to the developer.

IFFI is a mixed event and dataflow architecture, which uses an XML specification. However, it is unclear the methods it uses to order the events, handle event duplication and handle the issues that arise regarding complexity. Without these methods, the extent to which this becomes a problem is unclear, especially when IFFI is used in larger and more complicated systems. Additionally, the issue regarding performance of the adapters is not a large issue for most simple 3D interaction techniques.

2.5.2. InTml

Interaction Technique Markup Language (InTml) [Fig02] is a dataflow description of 3D interaction that sits atop the X3D and VRML specifications. In InTml, a filter is a unit representing a 3D interaction technique that has inputs and outputs. These are

used to define the flow of data and events at runtime in the system. Whereas FlowVR uses modules containing computation, InTml is a lower-level dataflow language where data is mapped between filters.

The dataflow architecture helps in the visualization of the interaction and creates a higher-level representation to assist developers. However, InTml suffers from the same limitations of dataflow architectures and that is a lack of hierarchy. Additionally, it could be problematic to represent the different states of the system and parallel and overlapping interaction common to 3DUIs.

InTml is similar to IFFI in that it relies on lower level APIs to implement the data that it connects. In this way, it becomes a portable definition of an interface, but one specified as a dataflow. InTml, as it is translated into an executable representation, does not face the performance problem of IFFI's adapters.

2.5.3. Contigra

Contigra [Dac01] is a declarative approach to 3DUI specification. Using Contigra's representation, 3DUIs are translated at runtime into the implementing representation, such as X3D. In this way, Contigra is a transferable representation between tools that additionally can create interfaces without the developer specifying how the interface works.

The benefit of declarative approaches is that developer declares what the interface does, leaving the system to implement it. The approach can lead to easily specified 3DUIs but can be problematic in the creation of more complex behavior. In this sense, it is a low threshold and low ceiling approach to 3DUI development [Mye99]. Additionally, implementing the system which interprets the declared 3DUI can be problematic, especially for multiple toolkits.

2.5.4. 3DUI Specification Discussion

There are several important issues to be mindful of with these approaches to 3DUIs.

A potential problem for 3DUI specification is that the same technologies that make them easily specifiable and transferable are also making existing environments easily transferable, such as X3D. In this way, the need to have a 3DUI that easily moves between toolkits is lessened because 3D content is already written in a transferable representation and these representations are working on interface descriptions too. As such, the approach falls victim to the moving target theme [Mye99].

Another issue is the amount of access to the internals of an implemented 3D interaction technique. Both IFFI and Contigra fall into the same trap of having to identify, create and maintain event sets per interaction technique that can lead to problems with extensibility as well as maintainability.

An advantage of these approaches is that 3D interaction techniques can be specified and reused. However, communal input is required to create consensus on a good representation structure. Currently, these three specifications describe 3D interaction techniques according to their own interests. How well that matches to the community at large is not known.

A final disadvantage is that 3D interaction techniques require more than just a specification of data and generic devices. Properties of 3D devices and hardware impact the usability of a technique. Recall the example from Chapter 1 where the implementation of the SSWIM technique required one hand to be remapped forward by about 20cm because the acoustic trackers failed when the technique required the

user hold one hand above the other. An additional example is the issues regarding changing the device used in a 3D environment. Though a toolkit might swap device drivers for the developer, there may also be required changes to the model loaded in the 3D environment to represent the device in the 3D environment.

Even with these disadvantages, at least a common API for portability becomes possible. As most 3D environments have only used minimal 3D interaction, this approach will be useful for many 3DUIs.

2.6. Related Work: 3D Behavior Specification

Several tools have looked at higher-level representations for 3DUIs and their behaviors. The earliest of tools focused on event-based architectures, similar in nature to 2D desktop toolkits. The limitations of the event-based architecture [Mye91] created a need for higher-level representations. This has resulted in several tools, which incorporate the benefits of event-based architectures along with state machine and dataflow representations. They also attempt to create representations that are more reusable, hierarchical and visualizable.

2.6.1. FlowVR

FlowVR [All04] is a dataflow architecture designed for the design and development of distributed interactive systems. Though not designed specifically for 3DUIs, FlowVR does claim to be for interactive systems. A running FlowVR application is a series of connected modules of code, where messages are sent between them to communicate data and events. To simulate the continuous timeline, FlowVR also has synchronizers to gather messages between modules distributed in the system.

There are several benefits to this approach. Due to its encapsulation of code and explicit coupling between modules, the ability for distributed and parallel processing in FlowVR systems is high. Additionally, because the specification ultimately runs code and is not a pure dataflow representation, development occurs in a language familiar to developers. Additionally, they can reuse existing toolkit functionality. Finally, the flow of information in the system is readily apparent, creating a higher level of understanding of the system for the developer.

A problem with the approach for interface design is, despite its ability to easily represent parallel processing, there is a lack of stateful information in the representation. This becomes problematic for the specification of interfaces that have functioning dependent upon system state. Creating interfaces in such a representation then requires replicating global system state in each module (global variables don't easily exist in distributed systems), creating synchronization issues, or passing global state along with each message in the system, increasing the message passing overhead. This can be problematic where systems with lots of modules can already be creating lag. Lastly, the ability to compose higher-level meaning with this approach is problematic as there is no support of hierarchy in its representation.

2.6.2. PMIW

PMIW [Jac99] is a User Interface Management System designed explicitly to represent non-WIMP interaction. Realizing both the discrete and continuous nature of 3DUIs, it uses discrete events to switch between the dataflow-like continuous relationships. In this way, it combines the benefits of dataflow while also matching the temporary nature of 3D interaction. Though it allows for abstractions, it does not

have structures to help it scale with complexity and decompose implementations into cohesive units. Like InTml, it is an academic system so has not been testing in full-scale development.

2.6.3. NiMMiT

NiMMiT [Boe07] is a toolkit for the design and development of 3D interaction, which has both a state and dataflow representation. In their requirements analysis, it was decided that a system for 3D design and development must be event driven, state driven, data driven and encapsulated for hierarchical reuse. In their system, the events are "user initiated actions" [Boe07] which drive the behavior of the system. The system responds to the events based upon the system's state and creates continuous behaviors that NiMMiT models as a dataflow. By capturing hierarchical behaviors, commonly repeated subtasks can be reused. Specific details of the interaction can be coded in a scripting language and NiMMiT currently uses LUA script. This representation is then fed to an interpreter.

There are a few interesting features of NiMMiT. In NiMMiT, its state driven representation lacks much of the structure of state machine or Statecharts representations. As such, the claims of encapsulation are problematic. Another issue is that in most dataflow systems, data is passed between parts of the system. In NiMMiT, the data flow is inside the states and as such is similar to PMIW.

2.6.4. Marigold

Marigold [Wil01] is a 3DUI specification toolkit that has a split representation between discrete and continuous behaviors, similar to PMIW and NiMMiT. For its discrete behaviors it uses Flownet [Smi99], which is a Petri-net representation, and its continuous behavior is specified by a dataflow representation. Additionally, it uses code for the description of the lowest levels of functionality. In Marigold, the separation of the discrete modeling of behavior from the implementation code allows behavior to be specified without concern for the lower level details. As stated in [Wil01], "…it reduces complexity to address such concerns separately as a refinement rather than at the same level of abstraction." The decision to separate behavior from lower level details was taken from the experiences of the HyNet [Wie96] project where exhaustive specification in a Petri-net led to a large and overly complex representation.

Marigold has the advantage of using higher-level representations so code can exist as code and behavior as a higher-level representation. In this way, the ordering of higher-level behaviors can be created before working on other functionality. It also merges the continuous and discrete behavior into one representation, reducing problems with keeping models synchronized. This is different from PMIW and NiMMiT. However, both dataflow and Petri-net representations are not commonly known to developers and requiring both, in the same diagram, can lead to learnability issues.

It is also unclear how its diagrams will grow as the complexity of the behavior increases. Such a representation is lacking in the ability to form human-understandable higher-level units of meaning. Abstractions are hard to create as well. Additionally, extending the diagrams is problematic without understanding all the details of the diagram. It is even difficult knowing which parts of the diagram are in different states and when they are concurrent.

2.6.5. ASADAL/PROTO

ASADAL/PROTO [Kim98] is a tool for the software engineering of virtual worlds. It focuses on the splitting of the implementation task into form, function and behavior, with representations in Statecharts [Har97], message sequence diagrams (sequence diagrams for the messages passed in the Statecharts system), dataflow diagrams and visual object specification. It also contains a language for programming behaviors called the component specification language (CSL) as well as a language for specifying the content of the scene, the Visual Object Specification (VOS). Through a processing step, these are combined into a running system.

The system uses many software engineering approaches to address the problems of design and development of 3DUIs. However, with so many customized techniques, the threshold requirements for using such a system are quite high. Even developers familiar with Statecharts would have problems due to the other approaches used (even a special purpose language CSL is used).

It is also unclear how the separate diagrams are kept consistent. For example, the MSD diagrams are used to understand the Statecharts diagrams but as such they maintain duplicate information. It is difficult to know how the system functions in this case, especially when the diagrams are not synchronized.

2.6.6. 3D Behavior Specification Discussion

Both PMIW and Marigold realize that discrete and continuous data need to be modeled in a 3DUI. Their representations differ in that PMIW uses discrete events to signal the switch between the continuous flows of behavior where Marigold joins the continuous and discrete into the same visual representation. Like PMIW, NiMMiT uses states to separate the continuous functionality of the system but unlike PMIW and Marigold, NiMMit also attempts to use hierarchy to create higher-level structures. All three representations have their own proprietary representation for developers to learn.

A fourth toolkit ASADAL/PROTO uses many standardized or modified standardized software engineering approaches. Its main representation is Statecharts and it uses sequence diagrams and dataflow to elicit requirements. With VOS as a standard interface, ASADAL/PROTO is similar to IFFI's approach of creating a standardized interface to develop to. In contrast to the previous toolkit's use of proprietary approaches, ASADAL/PROTO uses so many representations that the required knowledge of the developers is quite high.

A common thread of these tools is that they all realize a higher representation is required. Additionally, they merge the discrete and continuous representations. The advantages of these higher representations are greater reuse as well as visualization of the representation through custom tools.

There are several shortcomings too. First, these approaches do not appear to have any more use or any greater community support than any of the previous toolkits. In fact, the opposite is the case. Additionally, despite every developer's report that the representations are better able to visualize the 3D behavior of the system, the author is unable to follow the representations with their proprietary meanings. Finally, it is difficult to know if the tools have actually been used in the creation of 3DUIs and how they would work in the development of actual systems.

2.7. Summary

Thus, the extant literature has identified several requirements for a new system:

- The benefits of having an inclusive development environment for all developer-required functionality can be seen. This is the ability to use a common API and have the API deal with advances in technology regarding graphics, devices, tracking, audio, physics, ad infinitum.
- 3DUI specification can be seen as only going part of the way. While 3DUI specification is useful, 3D environments require behavior specification as well. Creating a separate 3DUI representation from behavior seems redundant. However, the three approaches of IFFI, InTml and Contigra provide three models of how developers think about 3DUIs.
- The thinking of the community used to develop 3D environments is not always the thinking of the mathematicians and computer scientists that developed computer graphics and write the APIs. The 3D developer community needs to have a representation formed in the thinking of their own community. Additionally, the 3D developer community is not necessarily a monolithic community.
- A specification will have to have input from a community. X3D has been worked on for years, coming to consensus between all the committee members. On a historic note, the graphics API PHIGS was designed by committee but was replaced by OpenGL, which was originally designed by a single company. The benefit of a single creator versus the stability of a collaborative committee needs to be considered in a specification.
- There is continuous and discrete behavior in 3D environments that has to be modeled as such. Many models have been created which all claim to be better. The opinion by the author, after an analysis of the visualizations, is that they are difficult to understand.
- A specification needs to allow access to the internals of the flow.
- A specification needs to allow the execution of code based upon the flow of the model.

From this chapter, we see a large number of features, too large to be addressed by one project. We see many different projects that have attempted to address 3D design and development with a tendency to expire. We see many problems that make the development difficult which is compounded by the difficulty in reusing past successes.

There is hope in software engineering's many potential approaches to the design and development of 3D environments. The overwhelming of event-based architectures of the earlier 3D tools has opened the door for other approaches like dataflow and Statecharts. In the new systems discussed above, it is telling how in each, a mix of these approaches is taken. Additionally, the new systems all operate on higher-level visualizations and abstractions. Unfortunately, they also operate without a guide for what makes a better visualization and even more importantly, what makes a better system. For this reason, the evaluation of systems needs to make its way into 3D design and development.

Lastly, X3D is having great success in becoming the de facto standard of 3D content but not as great of a success representing 3D behavior and interaction. This is evidenced by the existence of the work above. A warning as stated in [Win06b], "…I

have concerns that unless we are careful, a specification will have too many conflicts to be useful. We have a domain which is large and unexplored yet we are trying to create a specification which will cover it. Because of the unexplored nature of the domain, we will likely need it to change which conflicts with the concreteness of the domain. We want the specification to be direct and concise but because of the domain size, it will have to be large. Finally, we have very little ability as of now to judge the quality of the interfaces and ensure they will be good enough to be worth the effort of defining them in this specification".

A new approach to 3D design and development should focus its efforts not on content, which is being standardized, or behaviors and interaction, which is being explored, but on the representation and architecture of 3DUIs. The next chapter begins to address this by investigating how developers think and discuss 3D design and development, in order to develop a philosophy and later implement that philosophy in a system.

Chapter 3 - The Investigation

RQ1 - Why are current methods and tools insufficient for the design and development of complex 3DUIs?

Hypothesis – It is hypothesized that existing development methods and representations are insufficient because they do not match up with developer's models of the interface; they are system focused rather than the designer focused.

Because developers are able to communicate 3DUIs easily, this investigation focused on the developer's underlying understanding, representations, and language. In an effort to avoid any preconceived notion of problems regarding 3DUI design and development [Gre91][Mye93][Her94][Jac99], a Grounded Theory approach [Gla67] was chosen for this investigation. This is an emic approach, used here to inductively construct theories from a core of data without a priori beliefs. Additionally, as it is typically applied to text, human behavior or diaries, this approach corresponds well to source material. This investigation used this approach to report data findings as a series of themes and identified problems.

Two data collection procedures formed the corpus of 3D interface design and development knowledge. The first procedure gathered data from 3DUI developers in the form of developer artifacts (DA) and interviews. DAs consisted of meeting notes, UML, implementation code, graphics, hand-drawn pictures, diagrams, journal entries, conversations and any other form of communication relating to 3D interfaces for a developer's use. The second procedure was a collection of developer language by participants re-representing their observations of 3DUI video clips as language and pseudocode (further detail provided below).

Several themes and problems emerged from these, as shown in Table 3-1 and Table 3-2, below. As hypothesized, the problems identified here stem largely from the mismatch between the implementation and the developer's representations (see below for further details).

Table 3-1: Themes identified in the language of developer artifacts and language.

Streaming Thoughts	Statements captured as a stream of thought allow developers to quickly capture their ideas and first thoughts.
Domain Terms	The use of terms either from developer's domain of knowledge or defined ad-hoc for the situation allow developers to say a lot with only a single term.
Meta-Information	Meta-information describes the situation or context of the developer statements.
Composable Statements	Statements can be composed of other statement types.
Abstraction	Abstraction was used to create statements without going into the details of how to achieve the statements.
Specificity Order	The statements show a specific order of detail that allows developers to be abstract and specific as allowed by their understanding and whim.
Clustering and Paths	Statements cluster into temporary states in which they are active with paths between them formed by relationship statements.

Diagram Usage	Diagrams were commonly used to explain a part of an interaction or the scene in which the interaction takes place.

Table 3-2: Problems identified in the investigation.

Limited Understanding	Developers can describe their 3DUI at a high level, and the code to perform the actions at a low level, but are unable to describe the behavior between them.
Distance of Mapping	The distance of mapping between the envisioned behavior and its implementation limit the ability for developers to express common and important ideas.
Complexity	Each additional feature added to a system leads to a non-linear increase in implementation complexity with the potential to affect the functioning of the entire system.
Reimplementation over Reuse	Developers are unable to modify and reuse existing implementations for new purposes.
Hard Domain Problems	Several disparate types of knowledge and skill are required to implement 3D interfaces.

3.1. Collection Procedure

Data collection occurred through both developer interviews as well as a collection of their developer artifacts and source code. This created a corpus of knowledge about the 3DUI design and development process, which was then categorized into themes and analyzed for problems.

3.1.1. Interviews

Interviews with six developers provided insight into developer understanding of the 3DUI domain, tools and development process.

The first two interviews, with developer/researcher experts external to the author's university, were semi-structured and based on the Cognitive Dimensions Questionnaire (CDQ) [Bla00]. The CDQ is a series of questions used to analyze the ease of use of a notation system, or in this case the tools and process of the interviewees.

The third interview with an expert developer/researcher was conducted at the author's university. This interview focused on meaning and interpretative comments in the participant's journal entries.

The final three interviews consisted of developer/researchers external to the author's university. These were semi-structured interviews, based on the CDQ as well as experience from the earlier interviews.

In sum, there were six interviewees from six different universities and four different continents. These were augmented by several other 3D interface discussions, recorded during meetings and conversations at the author's university.

3.1.2. Developer Artifacts

Developer Artifacts (DAs) were obtained from both developers at the author's university and from members of the 3DUI mailing list. Because DAs are personal, informal, incomplete and domain specific, developers were hesitant to share their

DAs, even apologetic of the DAs' condition. Additionally, DA's are likely to be the first documents lost or destroyed during implementation, which further complicated collection.

DAs were collected from the author's journal entries (created years before this investigation), solicited from the author's university and solicited externally on the 3DUI Listserv (see Appendix A for the email solicitation).

Multiple artifacts were collected from seven different participants. Three participants submitted portions of their notebooks (this does not include the earlier developer/researcher interviewed with their journal). Because the notebooks had many entries, entries were often repetitive and difficult to understand. As such, the author focused on legible and interpretable entries relating to 3DUI development and those that were different from existing DAs. For example, if multiple images of a single interface were sketched, only one was collected. Other collected artifacts were notes developers had kept regarding a previous or current project and ideas they jotted down regarding an interaction.

This collection procedure did not result in a wide collection of DAs. The artifacts collected from the community were "polished" images, finalized UML diagrams or links to websites discussing a completed project. These are not DAs as they were not created early or during the design and development process and used to achieve the final implementation. This may be because 1) developers did not keep these early DAs. In the interviews, some interviewees mentioned that they frequently discarded early DAs because they were of little value after development began. Or, this may be because 2) developers were embarrassed of the DAs which are often difficult to read and understand. This may also be because 3) developers did not want to disseminate DAs. This is understandable; DAs are intellectual property and working ideas. Given this, it is likely that the author's rapport with local developers increased their willingness to release materials for this investigation. Additionally, it was found that nearly every local developer/researcher kept journals that meant they had more DAs to offer.

3.1.3. Source Code

Source code was collected from internal projects, open source projects and solicitation from the 3DUI Listserv. There were a total of twelve samples of source code selected in all.

3.1.4. Re-representation Experiment

To collect more language and developer representations, a re-representation experiment was performed. In this experiment, developer language was collected from nine participating developers who watched and then re-represented, in their own words, a series of 3D interaction video clips. This created multiple representations by different developers of the same 3DUI. Additionally, by removing developers from development tools, developers were not constrained by existing development forms. Using the Grounded Theory approach, the collection procedure in this investigation changed as the interaction between communication, developer and clip became apparent. This resulted in participants having slightly different amounts of time for some clips and later participants having additional post-experiment questions. For this analysis, the author spent an average of two hours with each participant developer.

Capturing the language used to implement a 3D interaction requires participants to 1) fully understand the interaction and 2) be uninfluenced by the researcher's

explanation. To achieve this, language was elicited from participants observing video clips of 3D interaction in a three step process (participants fully controlled the Quicktime video player in all steps).

In the first step, participants were asked to observe the clip and record their observations. Participants were told to pretend their notes were to be used to explain the observed interaction to another developer. Participants were encouraged to write descriptions, list the order of events and draw pictures.

The second step in this process was to validate the participant's understanding of the clip. This as achieved by verifying that a set of criteria was correctly mentioned and understood in their writing. The criteria set was determined a priori by three domain experts who observed all the clips (see Appendix B). When the participant's writing was incorrect or vague, they were questioned in as vague and unguided a manner as possible, with increasingly direct questions. This enabled the participant to come to the correct conclusions and expand their description of the vague criteria. It was the author's intent to be as unobtrusive as possible. These sessions were videotaped.

In the third step, participants were asked to "implement" the clip in pseudocode. An example of pseudocode was read to the participants so that they understood the task. Participants were informed that the purpose of this task was to see how they approached the problem--not to determine if they were correct in their implementation. This task was also videotaped. After each clip, participants rated the clip's complexity and their level of confusion. Participants were also questioned about the overall experiment after all tasks had been completed.

3.1.4.1. Clip Selection

Eighteen video clips were considered for use in this investigation. After two pilot tests, four were selected, with the first clip used as a practice trial. The observed interaction in the clips builds upon similar structures and interaction. This was purposeful so clips of more complicated interaction could be used in the experiment and the users would not be overwhelmed.

3.1.4.2. Practice Clip: Snap-To Raycasting Feedback

The practice clip of Snap-To feedback during a Raycasting task [Win05] is shown in Figure 3-1. The red selection ray extends off the user's finger while the yellow ray snaps to the closest object within 15 degrees of the red ray as the user moves their hand around the environment. This clip was chosen because the interaction was simple and mode-less. It allows the participants to experience the experiment's task without being overwhelmed by the 3DUI.

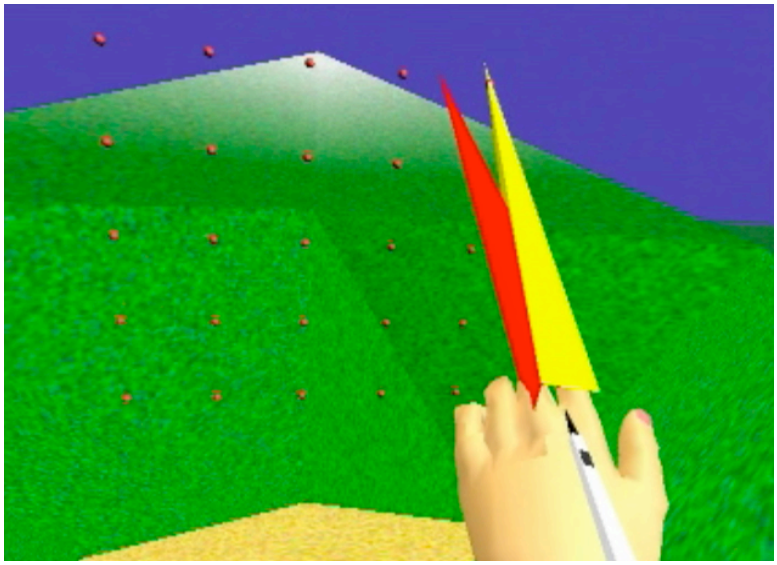

Figure 3-1: The Snap-To flavor feedback for Raycasting, used as the practice clip.

3.1.4.3. Clip 1: World-In-Miniature

The first clip used in this experiment was the World-In-Miniature (WIM) technique [Sto95]. In this clip (see Figure 3-3), a user manipulates furniture in a small room by dragging and dropping the WIM's proxy-objects for the furniture. This clip was used because it consists of a single, moderately complicated interaction technique.

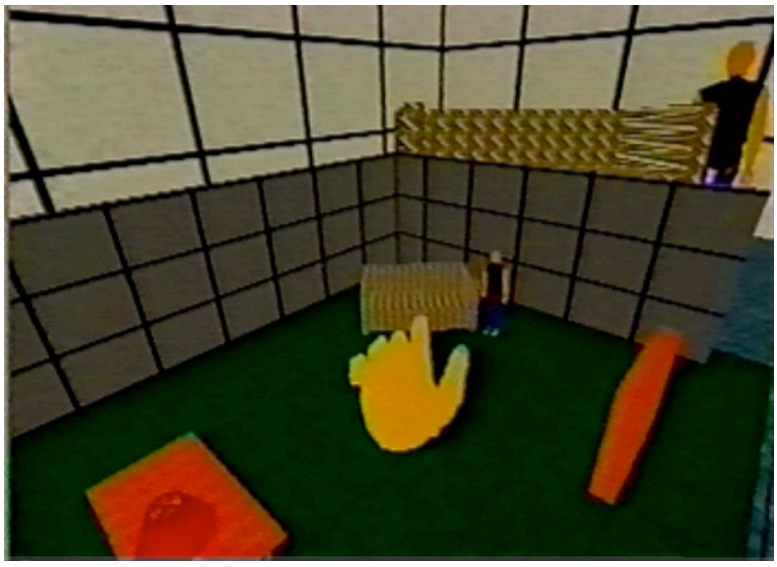

Figure 3-2: The World-In-Miniature (WIM) technique, the first clip in the experiment.

3.1.4.4. Clip 2: Georgia Tech Gorilla Exhibit

The second clip (see Figure 3-3) used in this experiment was the Georgia Tech Gorilla exhibit [Bow98]. Its main interaction is on a tablet that operates in much the same way as the WIM; the objects moved on the tablet move in the environment. This allows participants to reuse some of their knowledge from the first clip. Additionally, the user also moves around the environment by manipulating their position. This clip was chosen because it builds from clip 1 and is a combination of

simple and complex interactions.

Figure 3-3: The Georgia Tech Gorilla Exhibit, the second clip used in this experiment.

3.1.4.5. Clip 3: Scaled-Scrolling WIM (SSWIM)

The third clip observed (see Figure 3-4) was the SSWIM technique [Win06]. In this clip, the user moves themselves around a virtual city using the WIM to place themselves at a new position. The SSWIM technique adds scaling and scrolling functionality to the WIM, allowing for a more accurate placement of their representation and the ability to work with larger environments. Because the SSWIM augments WIM functionality, participant's knowledge of the WIM can be reused. The SSWIM technique was chosen because it is a complicated interaction technique with multiple simple interactions occurring simultaneously.

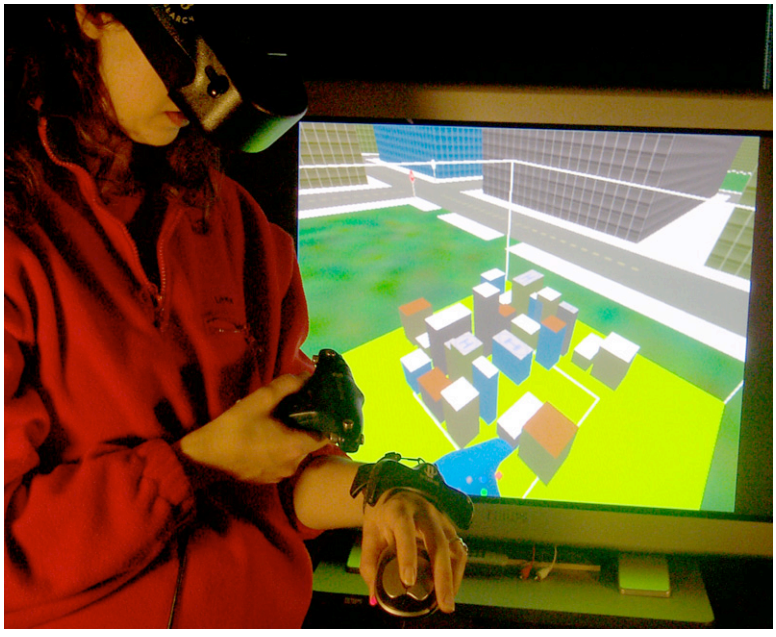

Figure 3-4: The SSWIM technique, the third clip used in this experiment.

Thus, the clips used in this study were structured to progress from simple to more complicated interaction. At the same time, the clips were showing roughly the same behavior; techniques to manipulate proxy objects for impact in the real world. This had several advantages. Participants could reuse their descriptions from earlier clips to understand increasingly complex interactions in later clips. This allowed the use of clips that otherwise would be too complex for an investigation of this kind.

The results of the Re-representation experiment are related to the statements found in the DAs from above. Because of this, all results are presented together in the statement classification (see section 3.2.2. below). The Re-representation experiment was found to have requirement statements (see 3.2.2.2.), relationship statements (see 3.2.2.3.) and problem-hypothesis statements (see 3.2.2.5.). These statements are then used in the support of themes in section 3.3. .

3.2. Data Analysis

Given the large corpus of data these experiments created, further analyses were conducted before themes and problems were identified. The DAs were classified into four different categories. The statements in the DAs and the language gathered in the Re-representation experiment were similar, so were pooled and classified. Finally, an analysis of issues in the source code was performed.

3.2.1. Classification of Developer Artifacts

Four types of DAs were classified: problem-solution (6.5), development note (5.5), interaction idea (8) and scenarios (1). Several other artifacts were reviewed but not classified. Additionally, the scenario DA was composed of 5 separate scenarios.

3.2.1.1. Classification Procedure

The DA classifications were created using a bucket sorting processes [Bey98]. The DA was examined and placed into a bucket that had similar features or a new bucket was created. The buckets were then resorted until no changes in buckets occurred. The justification for the scenario class, with only one item in it, is given below.

3.2.1.2. Problem-Solution

Problem-solution DAs describe a problem the developer has encountered and discuss hypothesized solutions to the problem. These types of DAs are similar to "reflection-in-action" which was described by [Sch83] as, "The practitioner allows himself to experience surprise, puzzlement, or confusion in a situation which he finds uncertain or unique. He reflects on the phenomenon before him, and on the prior understandings which have been implicit in his behaviour. He carries out an experiment which serves to generate both a new understanding of the phenomenon and a change in the situation." As developers explored design ideas in their journals, DAs were typically taken from these.

In Figure 3-5, DA#15 is shown as an example of a problem-solution DA. It has a problem statement on the bottom right, "- helps for large nodal systems, maybe use a cube for very large places w/diff levels." The solution is stated at the top, "For selecting occluded objects, have a frame they put over the objects and those objects de-occlude by mapping to a 2D surface by moving via forces." The bottom left contains a diagram. Note the informal text in these diagrams and how the developer used domain terms. This is common in the DAs collected in this study. In addition, some problem-solution DAs would have multiple steps, as each solution would lead to another problem.

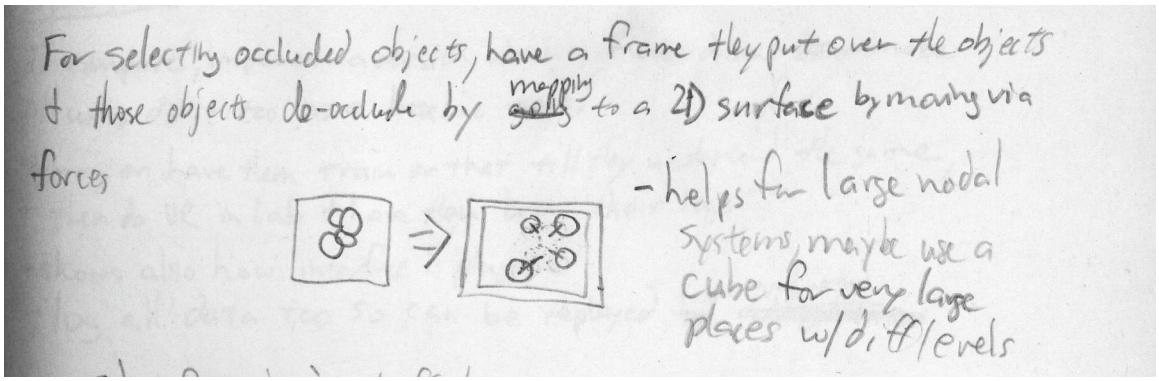

Figure 3-5. A problem-solution DA where the solution is stated first, then a small diagram and then the problem it is attempting to solve.

3.2.1.3. Development Note

Development notes are lists of tasks to complete, tasks completed, bugs, beliefs of the effectiveness of the 3D interface, schedules, useful notes about implementing the 3D interface and general thoughts (see Figure 3-6). Unlike the other artifacts, which were hand written, these were generally typed (two problem-solution artifacts were typed but it is unclear if this was their original form or if they were retyped for this collection of DAs). Typically, these were kept as developer project logs while they implemented a 3D interface. Some were found in journals but often they were electronic files. In the interview, and in discussion with developers later about these development note DAs, developers reported rarely coming back to these notes. Some, which were lists of items, had some items checked off with an 'X'.

Jan. 9

-finished integrating and testing the stylus API
-stylus now has new utility which […]
-created a rough texture to be used as […]
-tablet and stylus are now loaded into the environment
-changed […] to use 3 trackers instead of one
-old […] saved in […]

Jan. 10

-I thought I was going to have to rework my […]
-created map object and textured it, put it on tablet
-created button objects (4), in column on side of tablet
-got some button textures […]
-need a […]
-tested annotations and they work

Jan. 15
---[…]

Figure 3-6. A developer note DA, shown with identifying remarks removed, is a listing of tasks accomplished and things to do during development.

3.2.1.4. Interaction Idea

In an interaction idea, DAs describe how a 3D interface behaves (see Figure 3-7).

Where problem-solution DAs explore a problem, interaction idea DAs display either a full interface or a part of an interface. As such, they are a more refined DA with more information and interaction details, and possess application-specific comments and labels. The statements used in interaction idea DAs are similar to the statements collected in the developer language experiment. As with problem-solution DAs, interaction ideas are found largely in developer journals.

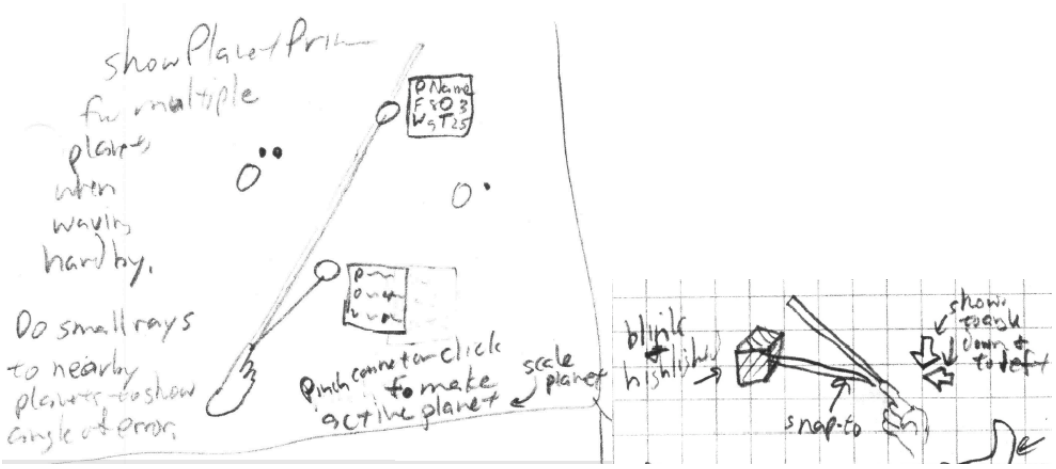

Figure 3-7. These two interaction idea DAs are taken from descriptions of 3D interfaces. Both describe Snap-To Raycasting and display how other task-specific information and feedback are shown in the 3DUI. Additionally, both were taken from a larger page of diagrams.

3.2.1.5. Scenario

Scenario DAs can be thought of as an interaction idea DA but in a text form and having a defined structure imposed on them from the scenario-based design literature [Ros02]. Because of this, they are classified separately. Scenarios try to be abstract, so the statements used reflect this. In Figure 3-8 below, the scenario DA describes a flow of behavior in the manipulation of pipes and electrical lines.

P5 – Pipe Path Specification

Brad wants to change the way that water pipes cut across a room. He brings the pipes down through the room to a point, has them make a 90 degree angle, come across, then another 90 degree angle. He then grabs some electrical lines that need to make a turn and he specifies an arc that they travel through since these lines don't make 90 degree angles. Another pipe he sets the angle that it crosses the room at by having it come down at a 90 degree angle into another 90 degree angle that rotates the pipe such that it can cross the room diagonally.

Figure 3-8. A scenario DA is a way of designing system functionality. The example scenario above would be classified as an Activity Scenario [Ros02]. This was one scenario of five that were used to describe the functionality of a system.

3.2.2. Classification of Statements

There were four classifications of statements in the collected DAs and developer language experiments: requirement, relationship, semantic and problem-hypothesis. The statement types were categorized by their information content and how the developer intended them to be used.

To identify the structure of each statement type, a general syntax was created for the statements (see Appendix C). Creating the syntax that accepted the developer statements was problematic as natural language has many variations. As such, the syntax is forgiving but allows statements to be rewritten and made clearer. In this way, they become well-formed examples. For example, the problem-solution DA in Figure 3-5 can have its solution statement: "For selecting occluded objects, have a frame they put over the objects and those objects de-occlude by mapping to a 2D surface by moving via forces," rewritten as a well-formed requirement statement: "When user places a frame over the objects, then those object de-occlude by mapping to a 2D surface by moving via forces."

Another interesting feature of statements is that the same content can take the form of any of the four statement types. Using the pairing of form and content, developers can convey their intent. Consider the developer needing to write about the content of using a travel technique in a 3DUI. The following are well-formed examples of the same content expressed in each statement type.

Requirement: "I need to travel in the environment."

Relationship: "When I push the button, I move along the ray's direction."

Semantic: "I like this travel technique."

Problem-hypothesis: "Because the environment is large, I need a simple way to travel around the environment."

3.2.2.1. Classification Procedure

These statement types were classified by applying a continuously refining syntax to the developer artifacts and developer language. After multiple passes through the statements, the grammar solidified. The grammar is inclusive and allows for many varieties of the statements types. This is important, as DAs were not expected to be seen by anyone other than the original developer, so there was little care or structure to make them readable. Additionally, many of these statements have value because they do not impose an order on the developer and operate as a stream of thought.

3.2.2.2. Requirement Statements

A requirement is a statement of what needs to occur, similar to functional requirements in software engineering. Examples of requirement statements can be found in Table 3-3. The requirement statements collected can be further refined into the sub-categories of factual, abstract and applied. A factual requirement statement is a statement of fact or status quo, such as "A model and user exist in the world" without any statement of action implied. The abstract and applied requirement statements are similar as both are a requirement for the system. However, the applied requirement statement describes how to achieve the requirement and the abstract statement does not. For example, the abstract requirement "User can rotate the model" becomes applied when stated as "User can rotate the model by looking around it." The addition of "by looking around it" is small but displays a greater level of developer understanding in their ability to connect the requirement to something in the interface.

Table 3-3. Example requirement statements from the developer artifacts and Re-representation experiment.

Developer Artifact	Re-Representation

- So, no matter how the selection box touches them, they keep their own property.
- Interface:
- - must be dynamic
- - tablet should at first contain a list of [cut]
- Take a scene of 5x5 red dots
- the user may move around the scene physically
- even the walls of the scene are movable
- user can scale the mini-rep to show more of less of the real scene
- User is not bound by objects placed on the map and may move through them if needed.

An additional requirement statement type, but not useful for implementation, was an observation statement, found in the re-representation experiment. Observation statements record what was observed by the participant but not written generically as a requirement. For instance, "move the cursor around" describes what they saw but it is not a requirement of "user should be able to move the cursor around."

3.2.2.3. Relationship Statements

Relationship statements are the building blocks of developer descriptions of 3DUIs (see Table 3-4 for examples). They capture contexts and create responses to the context in the system. The well-formed and simplified structure is: "When X then Y" with X representing the context and Y the response. The utility of this statement to development has been well documented [Pan02]. In this investigation, relationship statements were the most common statements found and the most useful and specific for describing 3DUIs. Interestingly, a relationship statement without a context can be considered a requirement (see the theme Composable Statements below).

Table 3-4. These are example relationship statements from the developer artifacts and Re-representation experiment.

Developer Artifacts	Re-Representation
• Pushing arm forward can speed zoom as much as the density of objects are (for instance, if have zoomed and only 2 objects in zoom and are filling zoom, don't fill any more). • If objects in front, then alpha them out or if zoom progresses, ignore them.	• If the user moves cursor over a object highlight object with yellow cursor • If the user clicks on a specific object then attach object to cursor • If the user clicks again, the object is placed where the cursor is at • Flash objects green when clicked

There are many ways to define the structure of relationship statements. In the simplest form, the relationship statement is a context, X, and a response, Y. Usually, the context is first and the response follows. As an example, consider the statement "When I push the button, I move along the ray's direction." This statement broken into its context X and response Y are respectively "When I push the button" and "I

move along the ray's direction." They can also be reordered to be Y-X as in "I move along the ray's direction when I push the button."

A context in a relationship statement identifies a state of the system to which a response is required--and can take many forms. The context is of causal or continuous type, often identified by starting with "When" or "While". The difference is that a causal context occurs instantly while a continuous context defines a period of time. The context can also use another subordinating conjunction, such as "After", "Before" or "As soon as", to provide clues to the ordering between statements (see the Clustering and Paths theme below). In the context, three types of state can be used: simple as in "X", conjunctive as in "X_1 and X_2" or disjunctive as in "X_1 or "X_2". Finally, a context can also be split around the response of a relationship statement, such as X_1YX_2. An example of this is "When I push the button I move along the ray's direction if the hand is at the user's side." In this case, the X_1YX_2 can be rewritten to be (X_1 and X_2)Y as in "When I push the button and user is holding their hand at their side, then move along the ray's direction."

A response in a relationship statement identifies the action to take in the system when the context occurs. As with contexts, a response can be a simple response "Y", conjunctive as in "Y_1 and Y_2" or disjunctive as in "Y_1 or Y_2". An example of this is "When I push the button then begin the SSWIM technique or move along the ray's direction." This allows the developer to specify two responses due to one context. This particular response, using "or", is important because a developer can specify two different responses in the system without having to give the exact cause for determining between the responses. When the developer knows this information, the response can be written to include conditional responses such as: "If x_1 then y_1, [else if x_2 then y_2], [else y_3]." An example of a conditional response of the form "If x_1 then y_1, else y_3" is "When I push the button, then if my hand is in the box, begin the SSWIM technique. Otherwise, move along the ray's direction."

Another feature of the context and response is the ability to be abstract or applied, like requirement statements. In the previous example statement "When I push the button, I move along the ray's direction," the context was instantiated as "When I push the button." The context could have been abstracted as "user wants to travel." Additionally, the response could have been abstracted as "I travel." As with leaving out conditionals in the response statement, there are benefits to being abstract.

Relationships statements can combine to create flows of behavior in a system. In the Re-representation experiment, developer statements classified as relationship statements were able to combine to create much of the functionality of the clips being observed. In developer artifacts however, the relationship statements, much like diagrams (see the Diagram Usage theme below), were used to explain particularly difficult interactions. As such, the functionality being described with relationship statements was incomplete or described at a high level without details. A reason for this may be that in their own artifacts, developers did not need to describe what they perceived as trivial behavior, leaving it as assumed. This is in contrast to the Re-representation experiments where this was the purpose.

3.2.2.4. Semantic Statements

Semantic statements consist of three types of statements: note, break and to-do (see examples in Table 3-5). Note semantic statements are used to provide non-critical but related information, often about what the developer was thinking about or where an idea for interaction came from. For example, "I got the idea when I considered how to avoid the problem with remote pointing." These contextual notes

are useful because many of the DAs are from journals that require the developers to remember what they were thinking at the time of the entry. Break semantic statements are used to signal a break from the current idea and the beginning of a new idea. This can be a line drawn across a page, a new title or even the start of a new paragraph. To-do semantic statements are common in development note DAs for listing items of work or scheduling tasks to complete during development. These statements are important for tracking developer ideas and keeping them organized. Semantic statements were found in the DAs but not the Re-representation experiment because the utility these statements offer were not required.

Table 3-5. Examples of semantic statements from the developer artifacts. The Re-representation experiment did not need the functionality of the semantic statements so they were not used.

Developer Artifacts

- Scenario - reaching for a specific orange in a fruit stand.
- ---
- I got the idea when I considered how to avoid the problem with remote pointing.
- BUGS:
- X- no textures yet on two top buttons
- X- [cut] doesn't work well for operators, leaves old stuff around

3.2.2.5. Problem-Hypothesis Statements

Problem-hypothesis statements describe a situation and a possible solution to it (see

Table 3-6). They are used to provide background for statements following them and typically start at the beginning of a problem-solution DA. An example problem is, "The problem with traditional techniques is that precise manipulation cannot be guaranteed if the objects are further away." Its following hypothesis (obscured for the anonymity of the participant) is, "Using [technique X], the world...". Sometimes, the problem is implied by a hypothesis like, "This technique can be used for pointing techniques for remote object manipulation." In this case, the implied problem is remote object manipulation.

Table 3-6. These are example problem-hypothesis statements from the developer artifacts and Re-representation experiment. Again, the Re-representation experiment found little use for these statements.

Developer Artifacts	Re-Representation
• in 9/12/2002, talked about a cone of probability • need more of a past history • w/ that cone, add prob. to each object selection and have it bleed off over time, then you get narrowing in and the ability to "circle around the object" • When we try to point to a remote location, it is hard to control the target position we are trying to pursue because of the increased sensitivity to user's hand/head movement. • [solution cut] • The sphere technique needs work • Idea 1) Bounding sphere on an object and [cut]	• Because the environment is large, I need a simple way to travel around the environment.

These statements can be specified using the other statement types. In this sense, problem-hypothesis statements are not identified by a structure but in the semantics of the statements. Additionally, these statements were rare in the Re-representation experiment, as the experiment did not require the developers to explore a problem but to describe how to implement what they were observing.

3.2.3. Source Code Analysis

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
>
> - Brian W. Kernighan

The source code was analyzed by reading through the code and tracing its execution while looking for noteworthy features of the code, architecture and API. Additionally, any stories told by the interviewees related to code are included. A limitation of this approach was that code is difficult to write and even more difficult to understand at a high-level. Another limiting factor was that a 3DUI developer analyzing 3DUI code will have problems finding noteworthy features due to familiarity with the domain. Despite these limits, some useful features were found and are discussed below.

First, source code is written with many assumptions. One assumption developers make is that the location of files in the system will not change (also mentioned in this study's interviews). Because of this, changes in the file system can break the 3DUI. Other developer assumptions are color or geometry, which do not always function between 3DUIs. For example, a highlight color used in one system would not work in another if the selectable objects were the highlight color. Another assumption is that code is operating alone in a system. This impacts initialization, such as loading in geometry for a 3DUI, and management of a critical resource, such as color or position of virtual objects.

Second, developers are often unaware of their requirements. When requirements

change, the code changes are widespread. Consider a virtual hand selection technique switching to a two-handed technique. The data structures, which previously could assume one hand, are now impacted in nearly every function. The code now has to identify which hand is performing selection, store the selected object according to the hand used in the selection and both hands have to be checked when looking for selected objects. Another example is switching a selection technique from single selection to multi-selection, which requires non-trivial changes to code. Other changes can be the addition of multiple selection techniques in a 3DUI or different sets of selectable objects dependent upon the state of the system.

The remaining issues are related to the event-based architecture used by many 3DUIs. First, event handlers must reestablish the state of the system when they are called. Because the same event can be used in multiple flows in the system, the event handler becomes packed with if/then conditional statements to know what actions to perform. Another problem with this is that the state is stored in global variables that are bad programming practice for several reasons [Wul73], such as a lack of information hiding, locality to use and poor documentation. The reason for the use of global data is that the API generally presets the parameters of the event handler functions preventing handlers that can pass in the proper stateful data. In one system, an undocumented global boolean variable called "UserBoxMoving" was set in one function and polled by functionality in not just another source code file, but another source code directory. This made the effect of its change difficult to understand, especially as it was not documented. One way around the reestablishing of state in event-handlers is by resetting the event handlers as the system runs. In this way, the use of global variables is not required. However, this approach suffers from problems with debugging and maintenance similar to self-modifying code.

A second problem with event handlers is that not all functionality is designed to be event-based and fitting into an event-based architecture requires the constant polling of data. This constant polling can lead to wasted processor cycles. Worse, it is possible that a value can change and then return to the previous value before the polling of the value occurs, resulting in a missed event. This is especially problematic if the polling is placed in the main event loop where application functionality can interfere with the polling frequency. For this reason, 3DUI systems use multiple threads or daemons for device data.

A third problem exists with event handlers. In event handlers, the decoupling between event-based architectures is good, in that the source of the event does not have to concern itself with how the event is interpreted. However, there can be multiple listeners of the event so the order of event processing can be problematic. As an example, one application was to set the color of an object but the color never changed. Code tracing revealed that the functionality was being called but that the object never changed color. After tracing further into the code, down to even the OpenGL code of the API, the problem was found to be a second function, running after the first in the same cascade of events that reset the color of object.

A fourth problem with event handlers is that special cases are often missed. This leads to random behaviors that are hard to reproduce and fix. Often, the source of the problem is a missed condition in the re-establishment of state in an event handler. This can result from a new use of the code that adds new requirements or simply a flow that was not tested. Part of the problem for this is that once a flow is implemented in the event handlers, it is difficult to reestablish or visualize in its entirety.

A fifth problem with event handlers is the destructive and wasteful overcalling of

functionality. Destructive refers to actions that are not undoable and wasteful to actions that result in more processing than is required[2]. One example was a system that changed the selection technique while the program was running. To do so, it first removed every event callback and then added in the new technique's callbacks. This was wasteful in that in many cases, callbacks were removed and then added. It was also destructive, as this approach would remove not just the selection technique but also any other functionality in the system when it removed all event callbacks. Another example of destructive behavior dealt with feedback in a selection technique. A material was applied to an object when it was selected and another material applied to the object as feedback when it was selectable. With both behaviors running, the following flow could occur; an object could exist with material #1, when selected it turns into material #2, pointing at it turns it to material #3 and then when the pointing stops, it turns back to its original material #1. This is a bug. The object is still selected so it should be material #2. In this case, the solution was for each frame to reset the selected object to material #3. This was a wasteful solution to a destructive problem. A final example of wasteful calling of functionality was from a function that normalized the scale of the user each frame, when it was only necessary when the scale changed (and this occurred infrequently).

The wasteful overcalling of functionality can also lead to terminal problems. This is the case when functionality can be called only once. For instance, consider a function that takes the currently selected object, changes its size and then releases the object. Calling the function twice could terminate the system as it tries to change the size of a NULL object.

Many 3DUIs create code that is good enough for its current usage. The previous examples show this -- the destructive and wasteful overcalling creates a 3DUI that follows its requirements. However, the created code is not reusable in other 3DUIs. Other examples of non-reusable code are simplified collision detection algorithms that only work for one environment or code written to predict the position of the user's torso. The code predicting the user's torso position assumed the user never bent at the waist or knelt, as both cases would have incorrectly positioned the torso under the user's feet. Additionally, the same code would not have been able to take advantage of an additional tracker that actually did track the torso's position!

3.3. Themes in Artifacts and Language

> Precise language is not the problem. Clear language is the problem.
>
> - R. Feynman

A thematic analysis of the collected artifacts and language is reported below. It was observed that the statements enabled developers to quickly report their ideas (Stream of Thought) and use terms loaded with meaning (Domain Terms). This created confusion so other statements assisted developer understanding (Meta-Information). Different statements were used as pathways to more detailed and structured organizations of developer thoughts. While the statements generally described single behaviors, they can be woven together to represent more complicated interactions (Composable Statements). In this way, statements describe complicated interaction and flows (Clustering and Paths). Statements could be left abstract (Abstraction) or progress toward more structure and detail as the need for a

[2] The harshness of the terms "destructive" and "wasteful" is not intended. In many cases, writing non-destructive and non-wasteful code is justified by the time saved in development.

more rigid definition was needed (Specificity Order). In this investigation, these themes are supported by quick sketches and hastily drawn diagrams (Diagram Usage).

Not all of these themes were as apparent in the developer artifacts compared to the Re-representation statements. This is understood after consideration of the motivation and context of each. The developer artifacts were designed to record ideas before proceeding to implementation or to explore 3D interaction problems and their possible solutions. The Re-representation was for fully describing a clip.

These themes are later used in Chapter 4 in the design of the principles of Concept-Oriented Design (COD).

3.3.1. About Thematic Analysis

These themes were created using an affinity sorting process [Bey98]. The evidence in the interviews, DAs, source code, Re-representation experiment and developer statements were used in this process. In the affinity sorting process, similar evidence was placed closer together and the process repeated until themes emerged. Typically, multiple subject matter experts are used to perform the sorting. However, in this investigation, only the author sorted, as the author was the only subject matter expert that watched all the Re-representation participants, looked through the source code, DAs, and so forth.

Additionally, a custom developed graphical tool, which simulated a surface where each piece of evidence was placed and moved about, assisted this analysis. With this tool, multiple surfaces could be created and the surface could be captured and returned to at any point in the future.

3.3.2. Stream of Thought

Statements captured as a stream of thought allowed developers to quickly capture their ideas and first thoughts. In the DAs, developers only wrote about what was important, leaving out details or connections. For example, developers listed alternative design possibilities, requirements for a technique, flow of the interaction and issues requiring resolution. They might then write about the trade-offs of each design--how to implement the requirements or issues. In the Re-representation experiment, developers recorded statements as they come to them and not necessarily how they flowed from each other. The interviewees stated they spent time understanding the flows of an interaction before they were built. To establish these flows, each were discussed and recorded in scenarios, sequence diagrams and storyboards. Interviewees acknowledged problems in coverage when describing interaction as streams of thought. Many times alternate flows of behavior or special cases were missed. For example, an expert describing a simple selection technique only reported on the happy path and failed to describe the path where a selection technique fails to select an object.

3.3.3. Domain Terms

Terms used from either the developer's domain of knowledge or defined ad-hoc for the situation allowed developers to convey much with only a single term. Terms such as “wave hand over”, “pinch”, “dock”, “snap”, “strafe”, “hover”, “highlight” and “grab”, succinctly expressed their intent. For instance, “grab” requires tracking a hand and waiting for a stimulus event like a button or a pinch to occur to determine what is grabbed. Just saying grab assumes that a grab occurred and the developer

can continue their flow of thought without concern for the "grab" process.

Domain terms were used heavily in the statements collected. For example, "Interactive, Dynamic Light sources that can be placed and changed," gave the requirements of object requiring selection and manipulation. "The joystick will provide the standard forward / backward / strafe left / strafe right" uses terms to describe how a device's data is mapped to movement. Another example regarding the manipulation of a pipe is, "puts another pipe, hovers over, and snaps. When release, it forms up." Here, the terms "hover over", "snap" and "release", convey certain functionality and leave the details of the functionality alone. For example, how long is the hover? How large is the distance between the pipe and hand for the snap to occur? What is the action to release? These details can be overlooked in the design phase, simplifying the developer's task.

There were problems with the use of domain terms, however, when developers had to implement them in code. The domain terms simplify design but have assumptions that are not always apparent and, as such, limit code reuse. These issues were discussed previously (see the analysis of source code above).

3.3.4. Meta-Information

Meta-information describes the situation or context of the developer statements. This information is important because many of the statements are written as a stream of thought and rely heavily on ad-hoc or domain terms. Recording meta-information helps the developer re-establish their original thoughts when they first wrote the statements. This can include problem-hypothesis statements that state the problem the developer is writing about. Additionally, semantic statements are used to support meta-information in that they provide notes to the author or reader. These statements are also useful for planning the project, seeing where work needs to proceed and what needs to be done for a 3DUI to be complete. With these statements, developers tracked interface problems or graphics and textures that needed to be fixed.

Because these statements sought to re-establish context or describe the statements themselves, they were not seen in the Re-representation experiment.

3.3.5. Composable Statements

Statements can be composed of other statement types. For example, a requirement statement can form the response of a relationship statement. Two relationship statements can be joined on the same context to form a single relationship statement with a conditional or compound response. Problem and hypothesis statements can be stated as relationship statements.

This can be seen in a description of the SSWIM technique. In SSWIM, one button was used to start flying and another to position the user inside the WIM. This can be described as two requirements: "User flies along the direction of the ray" and "User positions themselves in the WIM." By giving both of these requirements a context to start, we achieve: "When button is pushed, user flies along the direction of the ray" and "When button is pushed, user positions themselves in the WIM." These two sentences, because of a shared context, can also be composed as a compound relationship statement by defining when each is active: "When button is pushed, if user's hand is in WIM, then user positions themselves in the WIM otherwise user flies along the direction of the ray."

3.3.6. Abstraction

Abstraction was used to create statements without going into the details of how to achieve the statements. An idea could be written and captured before it was fully understood. In the statement types, requirements and the context and response of relationship statements were observed as being abstract and instantiated. The abstraction had three benefits for the developers:

- The developer can delay describing the details of the context or response for a later time. This allows the developer to focus on other parts of the system before returning to finish the full description of the relationship statement. This enabled the streaming thought theme above.
- The developer may not fully understand the exact context or response needed for the relationship statement. Being abstract allows them to record what they know and then continue to work at understanding the relationship further, refining their ideas.
- The developer may intend for the context or response to remain abstract so it can be redefined as needed for future usage. This is the process of creating reusable implementations. For instance, abstracting the context from "When I push the button" to "When I want to travel" allows the context to switch to alternate stimuli such as a pinch, voice command, eye blink or brain command.

3.3.7. Specificity Order

The statements show a specific order of detail that allows developers to be as abstract or specific as they wish, or as allowed by their understanding. The order of decreasing specificity by statement type is instantiated relationship, abstract relationship, instantiated requirement, abstract requirement and observation requirement. This order is achieved because relationships have a context and response while requirements have a response and lack a context.

The ability for a developer to be specific is due in part to an understanding of what they are explaining. As their understanding grows, statements written for a new specificity can be used. Observed in the developer language collection, relationship statements with the context before the response showed a higher level of developer understanding. This may be because putting the response first allows the developer time to consider what the context is and possibly leave it off if it cannot be determined. Additionally, the proper use of "when" and "while" to describe a relationship creates statements that are easier to follow.

The SSWIM technique's scrolling behavior in the developer language collection shows how the same behavior can be written as statements of different specificities. The scrolling behavior in the SSWIM has been described as:

"There was a large dead zone in the center of the SSWIM where no scrolling occurred, outlined by a white line after the first pilot evaluation. When the user moved their position outside of the box when positioning, the SSWIM scrolled the world in that direction (see Figure 3). The rate of scrolling was at first linear but was changed to a cubic function of the distance out of the box. This allowed for a smooth transition from slow to fast scrolling and accidental movements outside the box led to such small scrollings that users could easily position themselves while scrolling was occurring" [Win06].

The scrolling interaction, as represented by participants, is shown in Table 3-7. The table starts at the most specific statement for the SSWIM's scrolling behavior and

goes to the most abstract. Participant 9 gave a very specific declaration of the scrolling behavior. Though participant 8's description is a context/response relationship, the lack of "when" makes it difficult to follow. Participant 5's requirement statement allows them to state the behavior without the details of the scrolling. It is unclear if participant 5 gives a requirement statement from lack of understanding or for brevity. This also applies to participant 1's observation statement.

Table 3-7: Statements from the Re-representation experiment show the same SSWIM scrolling behavior as statements written at different levels of specificity. Because there was no example of an action/state relationship and abstract requirement statement, statements for participant 9 and participant 5 were rewritten for demonstration.

instantiated relationship (context/response)	Participant 9: "When the red cube is moved outside the white square, the miniature environment is strafed in that direction."
instantiated relationship (response/context)	Participant 9's statement rewritten: "The miniature environment is strafed in that direction when the red cube is moved outside the white square."
instantiated relationship w/o when or while (context/response)	Participant 8: "Move the red cube outside of the "inner" white box to move the city block in that direction."
instantiated requirement	Participant 5: "User scrolls through world using a red box to select the desired position to travel to."
abstract requirement	Participant 5's statement rewritten abstractly: "User scrolls through the world."
observation requirement	Participant 1: "The closer the cube gets to the edge of the small box, the faster it goes."

3.3.8. Clustering and Paths

Developers clustered statements and created paths between them with relationship statements helped by subordinating conjunctions. In this way, clusters of statements uses relationship statement to lead to new clusters. The subordinating conjunctions, such as "after", "before" and "now", helped to order the relationship statements. For example, in SSWIM, the scrolling functionality was dependent upon the user grabbing in the middle of the WIM to activate it and start a whole new cluster of active functionality. A relationship statement describes this, using a subordinating conjunction: "After grabbing in the SSWIM, begin to manipulation of the SSWIM." This describes the temporary continuous relationships discussed in [Jac99].

In developer artifacts, developers described functionality in chunks or groupings of statements. In many cases, this is centered on a diagram. However, most developers only describe the part of a technique that they are improving. When many ideas need to be described, the developer lists requirement statements. This avoids delving into details and maintains stream of thought.

In the Re-representation experiment, since their task was to completely specify the behavior in pseudocode, clusters and paths were readily created. However, clusters and paths had to be inferred from semantic break statements, such as the layout of

the statements, or the start and stop of a paragraph. For example, one participant used a relationship statement with an abstract response: "On the smaller display, let the user move the viewpoint along 3 axis." This was followed by multiple statements that operated in this defined cluster and then ended with: "If the user is at the edge of the smaller box, and moves outside, scroll their display proportional to how far they are outside of the box. Display an arrow on the ground indicating which direction they are scrolling in. Also the size of the arrow should indicate the speed of scrolling. Allow the user to zoom in and out of the smaller display so they can fine tune or move the viewpoint a great distance."

3.3.9. Diagram Usage

Diagrams were commonly used to explain a part of an interaction or the scene in which the interaction takes place. They are important to developers for describing ideas in an interaction even when they are little more than a scribble. They were hand drawn, but effectively augmented the explanatory power of the text. They tended to show the 3D scene or a snapshot of the 3D interaction at a particularly difficult point of the interaction. Hinting at the utility of the diagrams, every hand written developer artifact had some imagery with it. Table 3-8 displays the diagram usage per developer artifact type.

Table 3-8: Diagram usage by developer artifact type.

Type	Number	With Diagrams
Problem/Solution Technique	6.5[2]	3.5[1,2]
Development Note	5.5[2]	.5[2]
Interaction Idea	8	8
Scenario	1	0

[1] Two of these artifacts were retyped and it is unclear if the original document contained diagrams.
[2] The .5 refers to a developer artifact that fits into two categories because of two sections in the artifact.

While diagrams were requested in the Re-representation experiment and were seen in some of the practice clips, participants in later clips in the evaluation did not use diagrams. This is most likely due to time constraints, as evidenced by participant comments regarding diagrams: "I found they took too long" (participant 2), "With the time constraint I felt I was better able to just describe what I saw in words" (participant 4), "They took too much time to draw" (participant 5) and "I was too busy writing and did not have time" (participant 9). Because of this, the Re-representation experiment cannot be used to evaluate the utility of diagrams.

3.4. Problems in Existing Development

Several problems were uncovered in the investigation and are summarized below in terms of: Limited Understanding, Distance of Mapping, Complexity, Reimplementation over Reuse and Hard Domain Problems. These problems are addressed again in Chapters 4 and 6.

3.4.1. Limited Understanding

Developers can describe their 3DUI at a high level, and the actions to perform at

a low level, but are limited in describing the behavior between them.

This is apparent during communication and implementation. Knowing this, many developers use storyboards, discussions with others, scenario-based design, state diagrams and other formal and informal means to understand behavior. As expressed by one interviewee, "Behavior needs simulation in the head as they interact with each other." This enables much of the learning to occur before the developer implements the 3D interface in a computer language. Unfortunately, the implementation itself does nothing to support these informal means and the implementation's ability to convey the envisioned behavior is limited.

Part of the limited understanding problem is missed special cases in a 3DUI. For example, when describing the behavior of Raycasting, an interviewee successfully explained Raycasting but failed to explain its behavior when a button is pressed and Raycasting is not pointing at anything. This should have been identified in early stages of implementation -- but demonstrates that even the simplest of interaction can be elusive. As another participant stated, "All the exceptions are problematic--things you don't think about."

Another aspect of the limited understanding problem is that developers are often unaware of their own assumptions. This can be as simple as using static file references in code and assumptions of color or geometry for a 3D interaction technique. In addition, developers may assume that only one interaction technique will be active at a time; creating a Go-Go [Pou96] technique that assumes only one hand. Reusing the implementation for a two-handed technique can involve non-trivial change to implementation (see the Reimplementation over Reuse problem below).

Finally, user testing is the only means of understanding how the interface will actually feel to a user--and iterations take time [Mye93]. Tiny details affect the usability and can only be identified in pilot testing. One interviewee stated, "The unpredictable part of development is user acceptance. You go through multiple phases and iterations and still it isn't good. This comes from fuzzy requirements." Unfortunately, because of the time it takes to implement 3D interfaces, testing ideas takes too long. Another interviewee expressed the need for rapid prototyping tools, or "something like MATLAB, where we can prototype problems would be helpful." Interviewees also indicated that the literature on 3D interfaces played only a minor role in their development. When asked how much the literature helped, one interviewee responded that it was "hard to say... 40%" and another said the literature was only "somewhat helpful," with neither interviewee indicating much enthusiasm.

3.4.2. Distance of Mapping

The distance of mapping between the envisioned behavior and its implementation limit the ability for developers to express common and important ideas.

3D interaction is a series of "continuous relationships, most of which are temporary" [Jac99]. However, implementations are typically in implicit invocation [Sha96] architectures (event-based or publish-subscribe) where an event is broadcast to registered callback functions. This requires implementing the continuous, temporary relationships as short sequences of low-level instructions spread between multiple callback functions. Additionally, these callback functions begin with a series of if/then statements to reestablish state from global variables. Because of this, implementing 3D interfaces requires the simultaneous description of every envisioned behavior in a system at its lowest levels of detail across multiple callback functions. Changes to code face the same problems--making all changes large changes.

Several problems result from relying on the event-based architecture. It has been referred to as a spaghetti of callbacks [Mye91]; missing is the ability to create higher levels of meaning and encapsulate it as a flow of behavior to be maintained, tested and reused. As a result, specific points in the flow cannot be captured and responded to without considering every flow in the system. In 3DUIs, the notion of time and constraints are important but difficult to express. The envisioned behaviors must act over time but the functionality has to be discretized into slices of behavior to run across the callback functions. Additionally, developers must maintain their own constraints. Finally, to keep the architecture consistent, developers must simulate the event-based architecture when it is not available by polling data. This leads to the polling problems previously discussed. As one interviewee noted, "it's not the thought process [that is difficult], it's the details."

"Hooks" [Pol0] in existing code may improve this. Hooks give external functionality access to the internals of the 3D interface. In Panda3D and X3D, developers are able to create new events, which can be registered with on the fly. Unfortunately, because developers have a limited understanding of what will be useful for future developers, determining the proper events to create is difficult. Additionally, maintaining these events is problematic because every change to the code can potentially invalidate the hook. Because of this, developers often resort to quick fixes, or reimplementation of the 3D interface (see the Reimplementation over Reuse Problem). The previously mentioned examples of global variable use demonstrate potential problems with this.

When thinking of and representing ideas separate from development, developers use vastly different means of representing their envisioned behavior. Developers think of 3DUIs as flows of behavior, temporary relationships and hierarchical ideas built off of simpler concepts. To capture this, developers use storyboards, sequences of diagrams, finite state machines, "informal formalisms", dataflow diagrams, class diagrams, scenarios and text written on paper, journal logs, emails and even their hands. One interviewee stated that they "don't do formalisms because they need informalism". Another stated that, "formalisms are obscure so they use free-form text." Unfortunately, when going to implement their 3D interaction they lose the ties to their representations. Regarding their informal diagrams, one interviewee stated, "[we] throw [them] away once into it and never use it anymore. We don't look back." Still another interviewee said that "going from formalisms to C++… [we] don't have a way to do it."

Little of the utility of developer statements, collected in the DAs and Re-representation experiment, is contained in current implementations. Requirements, often collected before development, are maintained external to the code. Callbacks capture some of the utility of the relationship statements but only to the most generic event types and without the shown flexibility. Additionally, callback registration is embedded in the code and developers are not able to visualize all registered events and how the registered events change while the 3D interface is in different states. Finally, implementations have no support for diagrams.

3.4.3. Complexity

> *Each additional feature added to a system leads to a non-linear increase in implementation complexity with the potential to affect the functioning of the entire system.*

This is because each additional feature increases the number of actions and states of the user, environment, and interface. Since each action must be considered in

regards to each state, linear growth in actions and states result in non-linear growth of complexity in the implemented system. Existing methods of implementation are limited in their ability to express this growth in complexity. This is a classical problem of software engineering [Bro87] and referred to as the State Space Explosion [Har98].

Though also a problem of desktop interfaces, the impact of this is much larger in 3D interfaces. 3D interfaces typically have a much larger set of actions and states. In part, this can be attributed to the requirement of representing the user and representing their movements in the interface as compared to desktop interfaces, where users act through input devices. Compounding this problem is the volume of input data and uncertainty of the data that the interface must make sense of (see Hard Domain Problems below) [Jac99].

The representation itself also introduces complexities. Global variables tend to be undocumented or under-documented, unprotected by encapsulation and modified throughout an implementation. The callback architecture also makes it easy to miss all possible states for a callback to check for. Additionally, a coping strategy for the implementation complexity is to use destructive and wasteful calling of functionality. This slows the system and limits its future reuse.

Additionally, new envisioned behaviors also interact with existing envisioned behaviors to invalidate the functioning of both. Referred to as feature interaction [Cal03], this leads to retesting of all parts of a system.

3.4.4. Reimplementation over Reuse

> A worker answers an ad for painting highway divider lines. The first day they complete 10 miles and the foreman is happy. The next day they complete 5 miles and the following day only 1 mile. The foreman tells the worker, "Pick up the pace or you're fired." To this, the worker answers, "I'm trying, but do you know how far back that paint can is now?"
>
> - Anon

Developers are unable to modify and reuse existing implementations for new purposes.

Developers want to build interfaces like LEGOs®, with drag and drop ease. However, without reuse, developers are required to start at the beginning for each new 3DUI even though similar ideas have been implemented in the past. As a result, new ideas are slow to be designed and developed while old ideas cannot be extended. Creating good reusable code is a difficult process, and requires multiple times the effort to produce [Bro95]. However, if this were the only issue, 3DUIs would have developed good reusable code after years of effort, as several software projects are currently committed to doing just that.

Several reasons are to blame for the non-existence of reusable 3DUIs. These are a lack of code understandability, a lack of code configurability, a lack of internal access to an implementation and problems with how an implementation responds chaotically to modification.

First, a lack of code understandability impedes reuse. Already mentioned is how understandability is hampered by a single idea spread among a series of callbacks, the discarding of the original developer artifacts and the complexity of the implementation. Understandability is additionally compounded by the lack of good documentation of implementations. Interviewees said that they wanted libraries with better documentation, that was searchable by "not keyword but concept" with "high level comments, [as] low level is self-evident". Several developers also wanted

better navigation methods when developing, as “it is a problem finding parts of the code when looking.”

Second, code configurability allows developers to reuse code and adapt it to their needs, but developers are often unaware of their assumptions to even provide alternative configurations. This is hampered by a lack of limiting common uses and metaphors, the number of possible configurations and the problems typically found when extending code beyond its typical use.

There is a lack of limiting common uses and metaphors in 3DUIs. Unlike desktop interfaces, 3D interfaces are not constrained by well-defined I/O devices and common usage metaphors [Bow05]. The most applicable metaphor, the real world, is difficult to fully simulate and “magic” interfaces offer the ability to violate any rule of reality for the greater good of usability [Pie91]. As stated in extant work, “The human interface has too many interrelationships and too many possible manifestations for rigorous experimentation to fully explore them all in the near future, especially since all of the attendant trade-offs are changing almost daily" [Kru96].

The number of possible configurations in an interface can be seen in the simple interaction technique of Raycasting. The following six assumptions were found: 1) only single handed Raycasting (no two-handed selection), 2) only one object selectable at a time (no multiple selections), 3) only one set of objects able to be selected (can't have different object sets selectable at different states of the program flow), 4) the geometry and color of the ray was static, 5) the feedback type used in Raycasting was set to highlight the indicated object (there could have been none, expanding the Raycasting ray or displaying a bounding boxes around the indicated object), 6) the method of selecting objects with Raycasting was set to select the object with its center closest to the center of the ray (it could have been the nearest object the ray intersects or select multiple objects around the ray). In addition, there are a wealth of other selection strategies which exist in the literature regarding Raycasting [Ste04][Ste06][Pie97]. It is conceivable that a more complicated technique would only have more configurations.

There are problems faced when extending code beyond its typical use. This problem arises when reusing a 3DUI implementation in a way in which it was not intended. In [Mye99], the amount that a tool can do was called its “ceiling” and going beyond this was problematic. 3DUI developers often relied on external libraries to provide specific functionality but were limited in the ability to perform behavior even slightly outside the scope of these libraries. One interviewee commented on how helpful a library for virtual human animation was until they tried to synchronize their audio with the movement of the virtual human's mouth. They said, “sometimes libraries don't support something small so we have to hack it and that takes time. This can be a major problem because access to internals [of the tool] is problematic.”

Related is a third problem of extending an implementation. This requires internal access to the system that is not always provided. Through hooks [Pol06], external functionality can insert its code immediately following points in the execution of the existing implementation. Some existing 3D toolkits like Panda3D and X3D allow for the creation of new event types in an implementation. It is also possible for developers to implement this paradigm in their code [Gar93]. However, neither approach is a robust solution, as any extension to the implementation with the hook requires maintaining the event and enables each new developer the ability to invalidate all existing hooks. Additionally, knowing what is a useful hook is not trivial. The proper trade-off between detail and simplicity, in many cases, is dependent upon

the needs of the reuser and not the original developer. An example of hook creation comes from the code analysis where one developer used the polling of a global variable as a hook to signal other code, which has the potential to lead to problems.

Fourth, the usability of 3DUIs reacts chaotically to modifications, making reuse have unintended consequences. First, changes to an implementation can have unexpected consequences. As the implementation is spread across multiple event handlers and global variables, it is hard to know what code in the event handlers is for specific functionality. A full understanding of the system is required to be able to separate out the functionality and as such, developers opt for reimplementation over reuse. Second, hardware modifications, such as new trackers, displays and interfaces devices, chaotically impact a system. While the hardware seems comparable and its data similar, other issues arise which require modifications to the implementation. For example, one 3DUI changed tracking systems and because of the decoupling between devices and the running implementation, only a change to a configuration file was required. However, because the 3DUI hard coded the geometry to display for the interaction device, the users were shown holding the original device in their hand and many users were confused. Third, modifications to a 3DUI also can have a chaotic cognitive impact. For example, the WIM technique implemented in [Sto95] used an animation that appeared to move the user into the WIM when they changed positions. However, the implementation of the SSWIM caused the user to think of the WIM more as a map so the animation into the WIM was confusing. As such, an animation between positions was used instead, to match the users new cognitive model of the SSWIM (see Chapter 1).

Finally, this lack of reuse leads to problems with experiment repeatability. Because of the chaotic impact of a slightly different implementation, and because there is little reuse, the validity of usability meta-analysis is questionable. Interviewees commented that even in their own 3DUIs, it is hard to determine if the source of a usability issue is the implementation or the idea itself.

3.4.5. Hard Domain Problems

Several disparate types of knowledge and skill are required to implement 3D interfaces.

There are hard domain problems when implementing 3D interfaces that must be overcome, and in many cases, experience is the only tool. One hard domain problem is the amount of broad knowledge, skills and experiences required to build 3DUIs. Other hard problems are understanding domain specific problems and algorithms, making sense of the I/O and testing and debugging 3DUIs.

First, developing 3D interfaces requires a broad set of knowledge, skills and experience. 3D implementations can require knowledge of computer graphics, parallel and distributed processing, computer vision, operating systems, human perception, cognition and general physics. Additionally, developers need to be skilled electricians, artists, carpenters, 3D modelers, sound technicians and prop designers. Experience is required in the use of domain tools such as model converters. Several interviewees commented that experience was very important. "For simple changes, experts can do it easily but not novices. Large changes are very hard." Experience also is limited in its transfer between tools. One interviewee mentioned that they had spent 12 hours on a bug developing a 3D interface in a toolkit in which they were not familiar. They discovered the problem to be their assumption of the tool's handling of an event type and how it structured its scenegraph hierarchy. Another interviewee mentioned difficulties training developers, "New people have problems ramping up to

the speed of the group."

Second, 3D interface development requires knowledge of domain specific problems. Examples include Gimble Lock and specialized graphics hardware development. Multiple interviewees mentioned matrix transformations as a difficult problem, "3D graphics transformations are brutal." Additionally, algorithmic solutions to many types of problems facing developers are difficult to find. For example, one interviewee mentioned the problems they had converting between color scales.

Third, handling the I/O in a 3DUI is problematic. This is in contrast to desktop I/O, which has a standard mouse with two degrees of freedom (DOF) and two or three buttons, which provides precise and easily understood input. 3DUIs have no standard input devices, have 6DOF, buttons and joysticks and give large amounts of imprecise, continuous data [Jac99]. Compounding this is the real-time response required in 3DUIs. Additionally, 3D interfaces have much more contextual information such as the user's current view, body position, fatigue levels and nausea. Lastly, not all data passed through the system for the interface to understand. The effects of presence [Ken93], the ergonomics of near-field haptics [Bow05] and entangling cables [Mee03], as well as the social impact of the environment outside the virtual world, can play a role in how people interact. Even personal preference and game playing experience is important [Win05]. Typical approaches are to implement 3D interfaces as multi-threaded applications or distribute the computation across multiple machines, which are both difficult implementation paradigms. These approaches have to be taken, as even the smallest amounts of delay have been shown to impact 3D interface performance [Mee03].

Fourth, the testing and debugging of 3D interfaces is difficult. 3D interfaces have typical bugs, but the domain makes debugging more problematic. For example, there are issues like misunderstandings in the 3DUI APIs, working with non-robust hardware, fragility of custom-made physical devices and development in parallel and distributed processing environments. The following four issues make testing and debugging of 3DUIs difficult. First, the reproduction of bugs during development is problematic. Normal applications can be rerun or simulated but in 3DUIs, bugs can occur due to odd angles, positions or random tracker jitter that are difficult to reproduce. Stepping through a debugger, a typical method of debugging, is not possible in this situation. Second, compounding the problem of reproducing bugs is that bugs occur in time. 3DUIs require the user to proceed through a series of movements to reproduce the bug. Stepping through code with a debugger would break the timeline and prevent use of the debugger. Additionally, debuggers themselves can affect the performance of an application--that makes a bug disappear during normal interaction or even running animations. For example, one developer reported a case where a division by zero error during an animation delayed them for several days. Only by dumping the entire animation output to a file (a very large one) was the computational error discovered. What had made the bug so problematic was that even outputting small amounts of output to the screen slowed the application to the point that the bug did not occur. Third, when using a 3DUI, developers are generally not at the keyboard and mouse to use the debugging interface. This is made only more problematic by the encumbrance of HMDs and gloves and standing in cables. Many times paired debugging is necessary, with one debugging at the keyboard and the other developer holding odd positions while inside the 3DUI. Fourth, 3DUIs are also experiential and bugs do not appear until the user is inside the environment and even then, people adapt without realizing it [Bur06]. In one example, user's hands were moved a foot and a half up in a virtual environment and users believed them to be in the correct position even when

directly asked [Bow01]. Interviewees expressed frustration with the problem of debugging 3DUIs and reported common debugging problems such as "why isn't this on the screen" and "is it going to do what we want it to do". Solutions to this problem are problematic as though developers could write complicated scripts to perform tests of 3DUIs for the above reasons, it still would not address issues requiring the user to be inside and at the controls of the 3DUI. In some cases, the experience "just doesn't feel right" and time must be taken to identify why.

Finally, humans work well and are familiar with 3D perception and operation. Creating 3D interfaces which interface with humans in an effective manner has to effectively "fool" the user into believing that they can in fact work with the objects in front of them. Creating such realism is problematic for static content and worse for dynamic content, especially when attempting to be more realistic [Mor70].

3.5. Summary

This investigation into 3DUI development problems uncovered a large body of varied information. This can be attributed to the multiple approaches used in the data collection. This information was at first analyzed and then organized into eight themes and five problems of 3DUI development. The themes identified what is useful in developer artifacts and language. The problems identified what needs to be addressed in order to improve 3DUI design and development.

Chapter 4 - Concept-Oriented Design

Question - What are the core principles of an effective representation for the design and development of 3DUIs?

Hypothesis – Informed by the results of RQ1, it is hypothesized that the structure and information needed by developers can be created based upon developer understanding and the benefits of language.

The results of the investigation led to the creation of Concept-Oriented Design (COD). Concept-Oriented Design is a methodology for software design and development based on principles originating in developer representations and language. A *concept* is the unit of development for a COD system and encompasses a single software concern. It is a cohesive idea – a small reusable chunk of functionality. A concept is to COD what an object is to Object-Oriented Design. Like objects, concepts use other concepts to compose higher-level functionality. These concepts are called *component concepts* and are the reusable units of development and become the units of meaning to a community.

This chapter discusses the five principles of COD and gives support for them from the themes and problems of the investigation in the previous chapter. The five principles are Envisioned Behavior, Artifact and Language Focus, Tiered Representation, Longevity and Community. Later, in Chapter 5, these principles are used to design Chasm as an implementation of COD and an example of how COD in Chasm is used to implement real 3D interaction techniques is provided.

Table 4-1. The five principles of Concept-Oriented Design.

Envisioned Behavior Principle	Envisioned behavior is the "what is to occur" represented in a fluid and appropriately abstract representation while a concept and component concepts are the means of realizing the envisioned behavior.
Artifact and Language Focus Principle	Developers are able to conceive of 3D interaction despite the growth in complexity. A representation incorporating the themes identified in developer artifacts and language will be able to grow at the rate of developer understanding.
Tiered Representation Principle	No single representation is optimal for all developer needs but a tiered representation enables different information types to be represented optimally. Explicit connections between the tiers pull the system together as well as enabling a clear understanding of change impact.
Longevity Principle	The effort expended in creating a concept can be amortized over multiple implementations, and that COD systems will continue to perform as when they were first implemented.
Community Principle	The growth of a community is invaluable to a representation for the support and generation of a consensus of meaning it provides.

4.1. Envisioned Behavior Principle

Envisioned behavior is the "what is to occur" represented in a fluid and appropriately abstract representation while a concept and component concepts are the means of realizing the envisioned behavior.

An envisioned behavior, expressed in a fluid and appropriately abstract way, represents the single concern of a concept. It is the starting point of development that structures the developer's thinking as well as the concepts advertised behavior for reuse. It is the most precise definition of the concept at the level of abstraction deemed appropriate by the developer to explain the concept. The intent is not to be complete, but to be useful. An appropriate representation for an envisioned behavior could be a few sentences in conversational domain-specific terminology. This enables the developer to begin structuring their thoughts, designing and developing in a representation that is easy to work with and easily modifiable. Once complete, these same statements are easy for other developers to read and determine if a concept can be reused for their needs. The purpose is not to be detailed but fluid and appropriately abstract. For details, a developer will have to look deeper into the concept.

Envisioned behavior is inspired by many of the themes found in language. The theme of *streaming thoughts* allows developers to quickly record what they know at a high level. Developers are also able to use *domain terms* in the envisioned behavior to say more with fewer terms and can include the *diagram usage* to augment the envisioned behavior. *Meta-information* about their envisioned behavior also adds to its utility. Keeping the envisioned behavior with as few restrictions as possible enables the developer the freedom to create a clear description of the concept in the best fitting form.

Envisioned behavior seeks to address the problem of *limited understanding* from the investigation. Because developers at the start are limited in their understanding of what they are developing, the envisioned behavior allows them the ability to explain what they deem a necessary description of the concept to be. Developing the envisioned behavior allows them to think completely through many parts of the concept before they are burdened with further implementation details. Additionally, the same envisioned behavior is able to be quickly understood by a developer seeking to reuse the concept.

4.2. Artifact and Language Focus Principle

> *Developers are able to conceive of 3D interaction despite the growth in complexity. A representation incorporating the themes identified in developer artifacts and language will be able to grow at the rate of developer understanding.*

An artifact and language focus enables COD to scale with an implementation's complexity. This is due to developer artifacts and language abstractly representing 3D interaction as the most specific representation needed at a point in time. Additionally, developer artifacts and language were shown to compose more meaningful ideas from smaller and decompose large ideas into its cohesive components.

Incorporating artifacts and language in implementations and tools is important and not currently well supported. Evidencing this is the observation that developers spend time to create design artifacts, such as scenarios and storyboards, which indicate their value. However, once developers proceed into the development phase, they admittedly did not return to the artifacts. The design artifacts are capable of representing contexts, responses, system state and specific ideas which continue to change during development. Thus, they are useful during development and we can assume that the limited use of developer artifacts is not due to a lack of representational value but due to their disconnect from the implementation. This

disconnect seems analogous to the problems of documentation becoming outdated when separated from the implementation [Bro95][Mey97a] and becomes especially problematic when the implementation representation is very different from developer thinking. This creates a lack of cognitive directness in the development task [Hix93] and can readily be seen in event-based architectures where developers are forced to spread their implementation across multiple event handlers [Mye91].

By incorporating this principle, developers can progress from the abstractness allowed by the envisioned behavior towards a more specific representation, as seen in the specificity order theme from Chapter 3. This allows the developer to consider alternatives and explore the interaction in a representation that is provisional and open to change. Developer statements are also able to remain abstract as needed by developers. Two reasons for the abstractness are first, for when they are unsure on how to more concretely implement the system at the given time, and second, when they shift to work on other parts of the implementation. This gives the flexibility developers need.

Another reason to incorporate this principle is so developers can use language to assist in decomposing a concept and composing higher-level functionality. The domain terms theme suggests that terms are representative of highly cohesive functionality that can be used to decompose functionality into component concepts. Additionally, the relationship statements in developer language suggest how connections between concepts can be formed. As such, artifacts and language can be used to structure development.

Several problems of the investigation are addressed because of the artifact and language focus. First, the limited understanding problem is addressed because developer understanding at both high and low levels is captured in developer statements. As they decompose these statements into concepts and towards a more specific form, the developer is forced to address each detail of the interaction in a simpler component concept. Additionally, the representation is based directly on the developer's own chosen statements and structure. This is part of the solution to the second problem, the distance of mapping problem, which is reduced because the statements are based directly on developer statements and domain terms. Third, overall complexity is reduced because the functionality of the system is split into concepts along statement clustering and order, as well as domain terms. Though the same amount of functionality still exists in the system, it has been compartmentalized into component concepts.

4.3. Tiered Representation Principle

> *No single representation is optimal for all developer needs but a tiered representation enables different information types to be represented optimally. Explicit connections between the tiers pull the system together as well as enabling a clear understanding of change impact.*

Concept-Oriented Design is not a single but multi-tiered representation, taking advantage of each representation's strength. In the investigation, developers created different design artifacts for different parts of the implementation they were working on. This included explanations in language, flow diagrams, drawings showing mock usage of the 3DUI and pseudocode. Different information requires representation in different forms.

As shown in Figure 4-1, the first tier of COD is the envisioned behavior, based upon language, and successively lower tiers are not just more detailed language, but instead are representations best suited to explain the above layer's primitives,

keeping all ideas distinct. When a change needs to occur, developers can work with higher-level tiers before source code, which is very resistant to change. In this way, each tier explains *what* is happening, the tier below it explains *how* it is happening, and the tier above it explains *why* it happened. Explicit couplings, both across tiers and between concepts, create clear understanding of the system's functioning. Additionally, it shows the impact a change will have to the system by tracing the couplings.

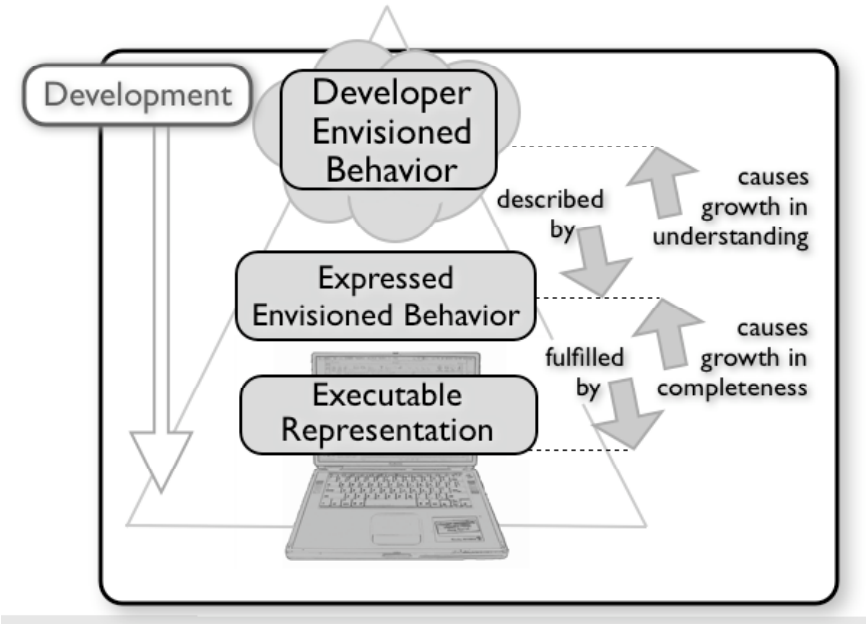

Figure 4-1. In Concept-Oriented Design, a developer's understanding is described in the envisioned behavior and fulfilled by the executable representation. These steps cause a growth in understanding and completeness of the representation.

The problems in Chapter 3 of complexity and distance of mapping are further addressed by this tiered representation. With the artifact and language focus principle, complexity was reduced by using component concepts to divide complexity. Building on this, a tiered representation separates complexity internally, again reducing the amount of interaction between ideas that give rise to complexity. This has the added advantage that each tier can be tailoring to match the information needs of each tier, reducing the distance of mapping problem. The goal is a clearer and less diffuse representation.

This is a well-understood issue in Formal Methods and Model-Based Design, as stated in [Bow95]: "Thus, larger applications often require a combination of languages. Indeed, many argue that no single notation will ever address all aspects of a complex system, implying that future systems will require combinations of methods. Process algebras and logics will become particularly important as systems become more sophisticated".

4.4. Longevity Principle

> *The longevity principle asserts that the effort expended in creating a concept can be amortized over multiple implementations, and that COD systems will continue to perform as when they were first implemented.*

The investigation, and evidence since the investigation [Ste08], has shown many

longevity issues with 3D interface implementations. 3DUI implementations are fleeting in that hardware and software change, source code is lost, developers moved-on in their careers, system usage directions and documentation are lost or forgotten, and so forth. Additionally, code reuse is problematic as developers commonly have changes in requirements and lack access to the existing code or choose to rewrite as opposed to modify the code. Because of this, the ability to compose higher-level meaning is limited.

Longer lasting concepts are more likely to be reused, which has multiple compounding benefits. To start, longer lasting concepts are likely to become familiar to developers, which increases the chances for concept reuse. With reuse, maintenance benefits each system using the reused concept. With reuse, testing benefits each system using the reused concept. Additionally, each system reusing the concept also serves as its own test as it is a new system under new requirements. This leads to a more robust concept with more reason to add functionality and extend the concept to new purposes. This cycle improves a concept with more maintenance, more developer familiarity, more testing and more functionality.

Concept-Oriented Design achieves longevity by the first three principles. The capturing of envisioned behavior maintains the concept's behavior and, through the artifact and language focus principles, makes its presentation understandable to others. This helps developers find and use the concept. With domain terms, it even helps to structure developer thinking, improving cognitive directness [Hix93]. Additionally, the tiered representation separates implementation ideas, which help to encapsulate the impact of change to a concept as well as seal-off external influences.

The problem of reimplementation over reuse is addressed by long-lived concepts. Developers able to rely on concepts being useful in their current and future work are able to apply their previous experience with the concept to their next implementation. If the concept is similar but needs slight additions of functionality, they are more apt to implement the improvements in an existing concept that they have previously been exposed to. This reduces developer time and also increases the value of the concept, for the original developer and others.

4.4.1. Concept Modifications

Any software system should be expected to be modified so that it improves and adapts to new purposes but modifications differ in their effect on the longevity of a system. It is important to manage these concept modifications of the envisioned behavior so the concept continues to function properly in past and future systems. Of the four types of modifications, the three acceptable are clarification, optimization and correction while change in a concept is not acceptable.

- Clarification – Modification that reshapes the concept's envisioned behavior's representation to better match its true envisioned behavior.
- Optimization – Modification to the concept's internals to more efficiently support the concept's functioning.
- Correction – Modification to the concept's internals to bring it more in-line with its envisioned behavior.
- Change – Unacceptable modification that alters or even replaces the envisioned behavior of a concept. Change should happen with a new concept.

Change to an existing concept is not allowed as it impacts all past uses of the

concept. If change occurs, the systems that use the concept will potentially fail. However, change is an important part of creating new concepts and meaning. Instead, change is performed with the creation of a new concept for the changed envisioned behavior. This is important so successful concepts remain available in past use, available for use in the future and continue to be created. A logical argument for why change of a concept is bad follows: *If an envisioned behavior were changed, it would have to be wrong. This cannot be the case since the envisioned behavior was what a developer at a point in time intended. Thus, the envisioned behavior could be of no use, but it cannot be wrong.* A reason for a change to a concept is because a developer wants functionality similar to an existing concept's.

Clarification can occur because the developer wants to improve the understandability of the concept or better document the envisioned behavior. Clarification should require no modifications to further tiers. A reason for a clarification of a concept is that it was written from one developer's viewpoint and not well understood by another developer.

Optimization should not impact existing use of the concept, due to encapsulation, but this is not always achievable. Proper testing is required and each system using the concept is a potential test. A reason for an optimization includes an improved data structure or algorithm.

Corrections improve a concept's robustness by correcting bug fixes as well as the handling of special cases. After corrections, any existing use of the concept should continue to function as it did before. If any existing use fails, then either the correction is invalid, improperly modifying the concept's functioning, or the existing use of the concept relied on side-effect functionality not specified in the envisioned behavior. In this case, the systems relying on the side-effect are tied to a version of the concept while other systems continue to receive the more correct concept. A reason for a correction is a bug fix or a case not realized by the original developer.

By following these modifications, longevity of a concept is improved.

4.5. Community Principle

> *The growth of a community is invaluable to a representation for the support and generation of a consensus of meaning it provides.*

An individual can build a concept but community builds robustness into a concept, supports new members using the concept and creates comprehensive solutions with multiple concepts to address a community's problems. In a community, concepts are refined in each system by testing under new conditions with bugs and patches contributed back to the community. Documentation for concepts is written from multiple viewpoints, clarifying the envisioned behavior. A community can also optimize a concept and add functionality not in the original concept but needed for another community member's needs.

Community also improves scientific analysis. Through shared tools, experimental conditions are more easily specified, reproducible and able to be rerun under similar conditions for new purposes. Experimental meta-analysis is also more reliable.

Community forms a consensus of understanding around its tools. With concepts being reusable, they become the units of reuse and discussion for the community. In this way, ideas can be composed to form higher-level units of meaning. Important concepts can be more heavily used and less useful concepts will be updated or ignored. In this way, successful concepts structure the thinking of a community while unsuccessful concepts are discarded.

Community creates and manages concepts. The community agrees to acceptable practices and organizes successful concepts into a useful vocabulary of concepts. This involves issues of versioning, managing modifications, categorization and distribution. Ultimately, higher levels of understanding and functionality can be conceived, as well as common ways of thinking. This has been seen in other communities such as CPAN for Perl, JSAN for Javascript and PEAR for PHP.

The investigation found very little broad community support. Nearly every interviewee, even at the host university, used a different toolkit to implement his/her 3DUIs. This is problematic for communities trying to share ideas and reproduce results. In support of this, participants in this investigation expressed the desire for common tools and broad communal support.

Community is supported by themes identified in language as well as the above principles of Concept-Oriented Design. First, with longer lasting concepts there is more time for a concept to become familiar and exert more communal influence. This, paired with the theme of *domain terms*, means that a concept becomes the realization of the term and allows further terms through composition. Second, the ability to store *meta-information* about a concept means that a community will understand not just what a concept is but why it exists and how it was conceived.

The principle of Community addresses the problems of *distance of mapping*, *reimplementation over reuse* and *hard domain problems*. First, because the community comes together to create concepts and propagates successful concepts by reusing them, the distance between developer's thinking and the representation in the concepts is reduced. Concepts come to embody the developer's knowledge and thought processes just as the envisioned behavior and tiered representation are mapped to developer thinking. Second, reimplementation over reuse is reduced not purely because needed functionality exists in the concepts but also because this reuse follows directly from reuse of previous developer thinking. Third, hard domain problems, once solved by a community member, can be shared in a community and reused by other members.

* * *

In sum, these principles are the methodology of Concept-Oriented Design, a new approach to the design and development of 3DUIs. As such, COD is a methodology and the next chapter, chapter 5, discusses Chasm, a COD system. Chasm's implementation of COD is a first attempt, an experiment in itself, used to demonstrate the ways in which COD may be implemented. As such, chapter 6 evaluates the principles of Concept-Oriented Design in 3DUI design and development and also addresses Chasm's implementation of Concept-Oriented Design.

Chapter 5 - Chasm

Question – How can Concept-Oriented Design be instantiated as a tool to improve the design and development of 3DUIs for developers?

Hypothesis – With careful attention to how developers think and work, it is hypothesized that Chasm can be created as a tool that implements the principles of Concept-Oriented Design.

To address this research question, Chasm was created to implement the principles of Concept-Oriented Design (COD). As such, Chasm is not a 3DUI API for domain specific content or a rapid prototyping tool for simple 3D interaction. Rather, Chasm is a tool that allows deep access to interaction details, where new ideas can be created and expressed, and even inspired, simply by working in the representation. Chasm is not an improved programming language, or just better visualization and management of source code. As such, Chasm may be viewed as a paradigm shift, focused on representing the thinking process of developers during design and development.

This is not to say that Chasm discards the benefits and advances of decades of programming language design. Instead, the tiered representation incorporates the strengths of current programming languages (C++) in its final tier of development, the code tier. This allows existing code libraries and developer experience to be used with Chasm.

Chasm is discussed below in four sections. First, the tiers of Chasm are discussed along with its dual representation in ChasmXML and C++ code. This representation is explored using an example implementation of the WIM [Sto95] and SSWIM [Win06] interaction techniques. Second, issues important to development with Chasm are presented. Third, Chasm's processing of actions is presented which is central to the proper functioning of a Chasm system; many special cases arose which compounded the problem's complexity. Finally, the fourth section examines the ways in which Chasm addresses the principles of COD and the problems identified in Chapter 3.

For a high-level Chasm understanding, the first and fourth sections of this chapter will explain Chasm's design decisions. For more detail, the second section can be read as well. Section three, as it discusses the internals of Chasm, will be for those interested in Chasm's implementation and internal functioning.

5.1. Tiers of Chasm

> We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.
> - H. Abelson and G. Sussman, The Structure and Interpretation of Computer Programs

Inside a Chasm concept, there exist four tiers that stratify Chasm's representation (Figure 5.1). Working through these tiers, the developer decomposes their envisioned behavior into concepts and component concepts. The first tier, envisioned behavior, stores developer's expressed envisioned behavior for a concept as conversational domain language. The domain terms suggest component concepts and help identify cohesive functionality. As developers learn about the concept, they begin to understand causal behavior between concepts.

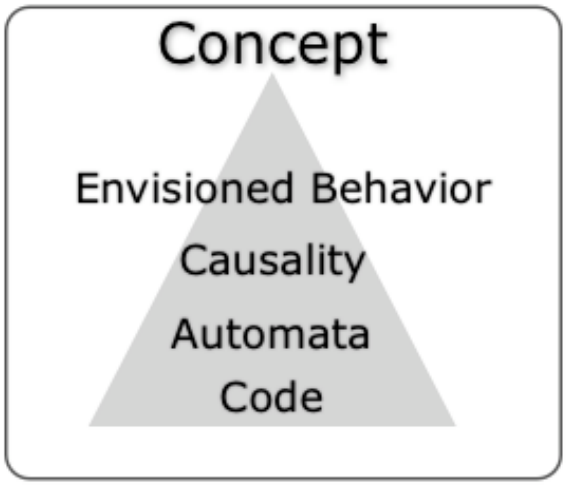

Figure 5-1. The implementation of a Chasm concept is stratified across four tiers of developer understanding.

The second tier, causality, makes explicit the causal relationships between concepts. These relationships use the response statements of "when X then Y" as identified in Chapter 3. This tier, by identifying all causality between concepts, creates a barrier between a concept's external influence and internal behavior. This allows developers in the following tiers to concentrate on the internal flow of behavior in the concept as external influences have been remapped to internal actions. This internal flow of behavior is the third tier, automata. The flow is captured in the concept as states, actions and transitions of a state machine. Once the flow is represented in the automata tier, the fourth tier, code, executes lists of C++ instructions in state entry functions when the flow enters a state. These tiers together represent, as well as document and support, developer's understanding as they implement a concept.

5.1.1. Representing the Tiers

Chasm's representation was designed to allow the transfer of existing tools and developer experience to Chasm. Chasm has two separate representations, ChasmXML and C++ code (see Appendix D for examples). This allows development to occur simultaneously at high levels in the ChasmXML and at low levels in the C++ code. It also allows developers to incorporate the power of existing tools and their expertise with those tools in the code tier. This includes source code analyzers, auto-documentation tools, versioning systems, refactoring tools and the like. This split representation is achieved through a special preprocessor step and a few special purpose Chasm command-line tools. The preprocessing step allows development of all tiers to proceed with minimal impact on the other[3]. Care was taken so that any Chasm preprocessor statements in the C++ code keep the code syntactically correct.

Alternatively, stub files could have been used; generated from the ChasmXML to hold the C++ code. With a stub file approach, the regeneration of stubs holds the potential for code loss when the stubs are overwritten. As development is iterative and every ChasmXML change would require a regeneration of stub files, there would have been a high probability of code loss. As such, Chasm did not use stub files.

---

3 One unhandled exception is changing a state's name in ChasmXML, as the state entry function in code is named in relation to the state's name. This results in a linker error until it is resolved by matching the state in the ChasmXML and code.

5.1.2. Envisioned Behavior Tier

> If you can't explain it simply, you don't understand it well enough.
> - A. Einstein

The envisioned behavior tier (see Figure 5-1) is expressed as conversational language to represent a single concern of a system. It represents "what is to occur" and the concept is the means of realizing that envisioned behavior. Recall from Chapter 3 that developers are able to state the main idea in a few statements. In the envisioned behavior tier of Chasm, these statements are used to structure development. Additionally, using conversational domain language as the representation for the envisioned behavior allows developers to express their ideas without having to transform their thoughts. In this way, Chasm matches the philosophy of Literate Programming [Knu92] with the belief that an implementation should be self-describing and developed like a novel. As the above quote is meant to illustrate, if the developer cannot explain the envisioned behavior in a simple representation like conversational language, they do not understand their implementation well enough.

In ChasmXML, the envisioned behavior is stored in the description tag. There are no structural requirements to this tier, which allows it to be easily written and changed as developer understanding of the envisioned behavior becomes more informed during implementation. With envisioned behavior, developers think of the full breadth of the behavior for the concept and discover and work through problems in their understanding early on. This is similar to the philosophy of CRC cards, that the "… purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code" [Hor05].

Approaches that help developers understand the envisioned behavior are use cases [Coc01] and scenario-based design [Ros02], which create an understanding of the flow of processing. However, this is generally the most apt to change in a system [Par72] so good decompositions focus on using information hiding to abstract ideas that are likely to change [Tar99]. An object-oriented decomposition addresses this by focusing on entities in the system and creating objects to represent them [Mey97]. The rationale for this is that objects in the system do not change as quickly as the processing of information or data structures. Unfortunately, this leads to implementations where the flow and processing of the system is spread among many objects and the original developer understanding, created by use cases and scenarios, is lost.

To address this, envisioned behavior's decomposes functionality based on language to help developer understanding grow as well as create a decomposition that is resistant to change. With language decomposition, developers use domain terms to refer to cohesive ideas they are familiar with. These same terms also express connections between other concepts. Given the community consensus of the use, meaning, and representation of certain domain terms, decompositions created using these terms are not as likely to change. This has the added benefit of improving reuse, as the envisioned behavior is more likely to match community thinking which improves understandability.

5.1.3. Causality Tier

The causality tier represents causality between concepts, the next level of understanding gained during development. This allows developers to match a stimulating context in one concept to a response in another, as can be seen in the

description of the WIM technique, "When the user manipulates objects in the map, the corresponding objects simultaneously update in the full scale virtual reality" [Sto95]. By representing causality as first-class objects in its own tier, called causal statements, Chasm achieves the sought-after design of loose coupling through causal statements, while the lower tiers in a concept are highly cohesive. In this way, the causal tier serves as a barrier to external influences on a concept. With the external influences defined, developers can continue their decomposition of the envisioned behavior into the lower tiers, with a simplified mental model.

A *causal statement* is a mapping of an external context in the system to an internal response in the concept. A causal statement takes the form of; "When X then Y" where X is the context and Y is the response. X and Y can at first be abstractly defined in conversational domain language but eventually are instantiated in the next lower tier, the automata tier, as a state(s) and an action. Because the causal statements of a concept defines the coupling between concepts, the statement can be redefined to easily:

1. Respond to a different context as the developer understanding changes.
2. Fulfill a new role in the system using the concept for a new purpose.
3. Respond to new contexts when reused in new systems.

Importantly, causal statements support how developer understanding grows during development by proceeding through three steps of increasing specificity. The first step is a text description that documents the meaning of the causal statement, i.e. its intent. Using the WIM statement as an example, the context and response text descriptions are "the user manipulates objects in the map" and "corresponding objects simultaneously update in the full scale virtual reality." The causal statement in this step is incomplete and leaving it as such is useful to the developer who may not fully understand the causal statement or would rather first complete other parts of the system's development.

The second step instantiates the causal statement's response to an action in the concept, so external influences are mapped to internal actions. In this way, the lower tiers of a concept have a list of influences that they must respond to. Any new external influences, due to new requirements or reuse, are then mapped to the internal influences using causal statements, leaving the lower tiers untouched.

The third step instantiates the causal statement's context to state(s) in the system, identifying the concept's external influences. Three possibilities exist; 1) the context is mapped to a state in an existing concept, 2) the context is mapped to a state in a new concept that the developer will implement, or 3) the context is mapped to a state of a reused concept, either the developer's, or from the vocabulary. In the investigation, developer language often did not include a context because developers only understood the action in the concept, not the external context. For instance, leaving the context for a Raycasting selection as abstract enables developers to later match the stimulus of the Raycasting to a button push or finger pinch, making it more composable.

One important issue with causal statements is event ordering (see Action-Processing Dictum below). By declaring causal statements, a developer is only specifying a partial ordering for the events in the system. For example, if three causal statements exist on a state, there is an ordering of which actions are sent first. However, those actions can cause other actions, internal to the concept and external by other causal statements. By the action-processing dictum, the developer is freed of the enforcement of this order and can focus on the other parts of the implementation.

Two improvements to the causal statement were added after Chasm was first implemented to improve its expressiveness; conjunctions and continuous statements.

- Conjunctions were used to form multi-state contexts. A conjunction allows the developer to say, "When X1 and X2, then Y" or "When X1 or X2 then Y". Conjunctions can also be nested. This enabled more powerful causal statement declarations.
- A continuous statement is "While X then Y", as opposed to "When X then Y." It is useful in describing continuous relationships that operate over time. Because continuous statements can be represented with two causal statements, it was at first considered 'syntactic sugar'. As such, it was added late in development and has yet to be used in any implementations.

5.1.4. Automata Tier

> If you cannot describe what you are doing as a process, you don't know what you're doing.
>
> - W. E. Deming

The automata tier represents the internal process or flow of behavior in a concept. As the causality tier mapped external stimuli to internal actions, the automata tier is focused only on internal behavior. As this reduces the concerns of the concept, a more rigid representation can be used. In Chasm, a state machine is used, more specifically a modified Moore machine [Mor56] with states, actions and transitions between its states. A benefit of a state machine representation is its familiarity to developers.

The automata tier consists of an alphabet of actions it accepts, Σ, a set of states, Q, a start state *s* and transitions that map a (state, action) pair to a new state $\delta: Q \times \Sigma \rightarrow Q$. Unlike Moore machines, Chasm's state machines do not use a separate output alphabet, output functions or multiple end states. Instead, each state in Chasm has a state entry and exit function, ω, which is evaluated when it is entered or exited. State functions are implemented in the code tier and have the option of returning an action in Σ, which will then be processed in the automata tier. There also is only one end state, used for determining if the concept is ready for collection by the system (i.e. calling the concept's C++ destructor). As Chasm is not a hierarchical state machine, each concept exists in its own state, allowing for concurrency in a Chasm representation.

As stated above, a concept is encapsulated by remapping external contexts to a list of internal actions. As actions are local to a concept and their number limited in practice through the use of good decomposition, exhaustive consideration of each action in each state is tractable. This reduces some of the unexpected behaviors commonly found in 3DUIs. As an aside, because hierarchical state machines do not have local actions, there is always the potential for additional functionality to break the existing.

5.1.4.1. States

States are clearly defined points in the internal flow of the concept and serve two purposes. First, they are used in development to define the internal flow of a concept. Second, they are the contexts of causal statements. Generally, once a concept is implemented, these states do not undergo drastic modification without changing the envisioned behavior (see Modification in Chapter 4). Because change to the envisioned behavior would require the creation of a new concept, developers are

able to rely on the state remaining unchanged.

States were found to have three types of uses; as a resting point, a decision state or a point-of-interest to external functionality.

- A resting point state describes a temporary state that the concept flows through when performing its functions. For example, a concept representing a button has resting point states of up and down.
- A decision state's entry function executes to determine the action to return to the concept. As an example, a concept representing Raycasting selection has a decision state *selectioncheck* that determines if an object was selected, returning the action *selection* if it was.
- A point-of-interest state is like a resting point state but exists only for use by external functionality. In this way, a state entry function may be split into two point-of-interest states. Another use would be by a decisions state that creates a point-of-interest state with no functionality. Using the Raycasting example from above, the decision state could return an action *noselection* so that Raycasting transitions to the point-of-interest state *nothingselected*. Once in that state, there is no state entry function but external concepts might find it useful.

A state satisfies the need for "hooks" in development [Pol06] without the associated problems of creating and maintaining events. First, using a state as a "hook" does not require additional development effort when the concept is created. Second, there is no cost associated with maintaining the states for external use, as the state is part of the concept, not an event that is maintained with it. Third, the states should remain unchanged in a concept, despite modifications (see Modification in Chapter 4) so use as a context for external functionality remains invariant. Fourth, Chasm manages the order of action processing such that all concepts responding to the entry to the state have a chance to respond before stateful information changes. As such, the developer does not have to be concerned about when the action is processed.

5.1.4.2. Actions

Actions are local changes that drive the concept from its current state to another state along a defined transition. First, Chasm actions are atomic in nature in that they do not contain additional information passed with the action (such as a float or string). This reduces any external concept from dependency on the current concept's data structures. If such information is needed, method calls in the code tier will provide that information. Second, an action only has local scope to a concept so it limits the change it can have in a system and greatly reduces the system complexity.

An action can occur through three methods (for a more detailed discussion see Ordering Rules and Event Types below):

- An action can occur through a causal statement that maps a context in the system to an action in this concept.
- An action can be returned by a state entry function, acting as a decision state discussed above.
- An action can be injected by the code tier manually using a method call, which bridges the automata and code tiers.

5.1.4.3. Transitions

Transitions are valid paths between states (or reentry to the same state) due to an action. When the concept is executing, the current state of the automata tier

changes along these transitions. When no transition exists for a state and action pair, an error occurs. Chasm helps resolve this error with debugging information and by halting execution.

Two pieces of syntactic sugar are used by Chasm to simplify the specification of transitions. First, a transition can be defined as being ignored, reducing the size of the representation. Second, a method call can list all transitions for an action to go to a single state.

Finally, a note about the automata tier and decomposition. The automata tier naturally identifies concurrently executing behaviors, which may not at first be apparent. As the developer identifies the flow between states it becomes apparent that separate flows exist and that certain states can be held in parallel, resulting in further decomposition of the concept as the functionality in the concept is divided between two concepts.

5.1.5. Code Tier

> When the words are fuzzy, the programmers reflexively retreat to the most precise method of articulation available: source code. Although there is nothing more precise than code, there is also nothing more permanent or resistant to change. So the situation frequently crops up where nomenclature confusion drives programmers to begin coding prematurely, and that code becomes the de facto design, regardless of its appropriateness or correctness.
> - A. Cooper (in "The Inmates are Running the Asylum")

The code tier employs the strengths of procedural programming inside Chasm's representation. Procedural programming is able to precisely represent algorithmic solutions to problems by listing instructions to occur during execution. Developers are also familiar with procedural programming, having existing coding practices and many tools and toolkits to support their development. Despite these advantages, chapter 3 showed problems with procedural implementations being able to represent flows of behavior. Additionally, it is resistant to change and its documentation can become out of date [Bro87]. Using the first three tiers of Chasm to organize the flow as well as document the execution points for code, the strength of higher tiers compensate for the weaknesses of the code tier.

The code tier impacts the higher-level tiers of Chasm in four ways. First, concepts are self-aware of the structures in their tiers. This gives Chasm several useful properties to assist the developer in debugging. Second, each component to a concept is given a component initialization form (more below). This is used to generate runtime checking code in the code tier to validate that developers handled components appropriately. Third, the code tier can create an action in another concept that executes immediately. Fourth, a state entry function can return an action in a decision state. Putting the decision process into the code tier simplifies the notation of the automata tier and allows developer the full expressive power of procedural code to express the decision.

The code tier has several details important to its functioning discussed below in section 5.2.2.

5.1.6. Tiers in an Example Implementation

The benefits of Chasm’s tiers are best demonstrated through an example implementation using World-In-Miniature (WIM) technique [Sto95]. As a moderately complicated interaction technique, the WIM shows how a traditional decomposition is lacking as well as how Chasm's tiers resolve the problem. In addition, the WIM is

extended to include the scaling and scrolling functionality of the SSWIM technique [Win06] that shows how tiers enable greater reuse and extension.

5.1.6.1. A Traditional Flow-Based Representation

The first step of decomposition is gaining an understanding of what is to be developed. The WIM technique was described by a domain expert in the investigation as "a tracker's movement causes the virtual hand to move. When a button is pressed, check for selection of an object in the WIM (a proxy object). If an object is selected, move the full-scale object in the environment as the proxy object moves. When the button is released, release the proxy object from the user's hand and stop moving the full-scale object." This description is shown below as a flow of events (Figure 5-2).

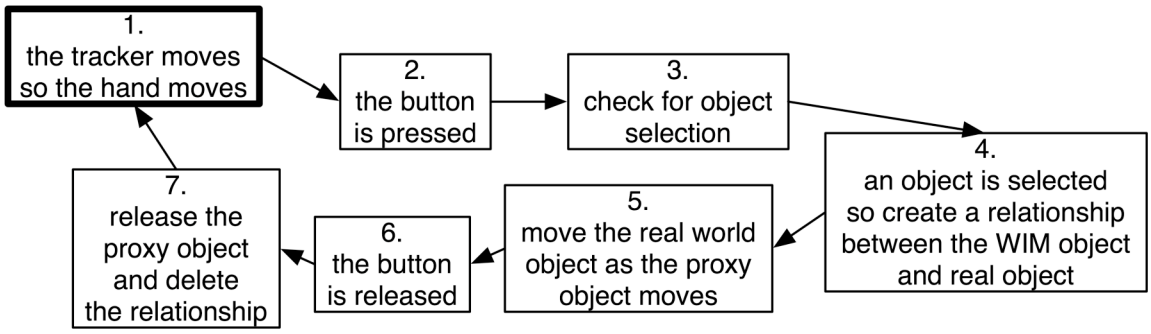

Figure 5-2. This is a full description of the WIM technique shown as a flow of functionality. There are several problems when using this flow in development.

Though the flow in Figure 5-2 is useful for its complete understanding of the technique, it has limited utility to a developer for the following reasons:

- The flow separates related events into distant steps. For example, steps 2 and 6 deal with a button and steps 4, 5 and 7 deal with the relationship between the WIM's proxy objects and the world's objects.
- There is no separation between discrete and continuous behaviors that have different ways of being developed. Steps 2 and 6 are discrete steps dealing with a button while steps 1 and 5 operate over time, dealing with the continuous updating of objects in the virtual world.
- Alternate paths are not easily identified in this representation. The domain expert did not identify the alternate path in step 3 which would handle the case of an object not being selected. This flow of events is only for the happy path where an object was successfully selected. Missing alternate paths during development creates unexpected behaviors and unplanned work. These problems are especially troubling when caught later in development.
- This flow of events requires the developer to maintain all the ideas of the WIM technique in the developer's head during development. This creates a complicated mental model of all the requirements of the hand, button, selection technique and object-to-object relationships. The flow does not indicate a decomposition of the WIM into simpler components to be developed separately.
- The flow is the first to change during the lifetime of a system [Par72]. Building a system along this flow will most likely result in major changes in the future.

5.1.6.2. The WIM in Chasm

In contrast, a Chasm representation is based on abstractions in conversational

domain language. The WIM technique was described by Stoakley et. al. in [Sto95] as, "A World-in-Miniature is a hand-held miniature 3D map. When the user manipulates objects in the map, the corresponding objects simultaneously update in the full scale virtual reality". Notice how this describes a complex interaction technique by depending on domain terms such as "3D map" and "hand-held". The second sentence also uses causal language to describe the behavior of the WIM. This new description is used in a four-tiered Chasm implementation of WIM, as shown in Figure 5-3.

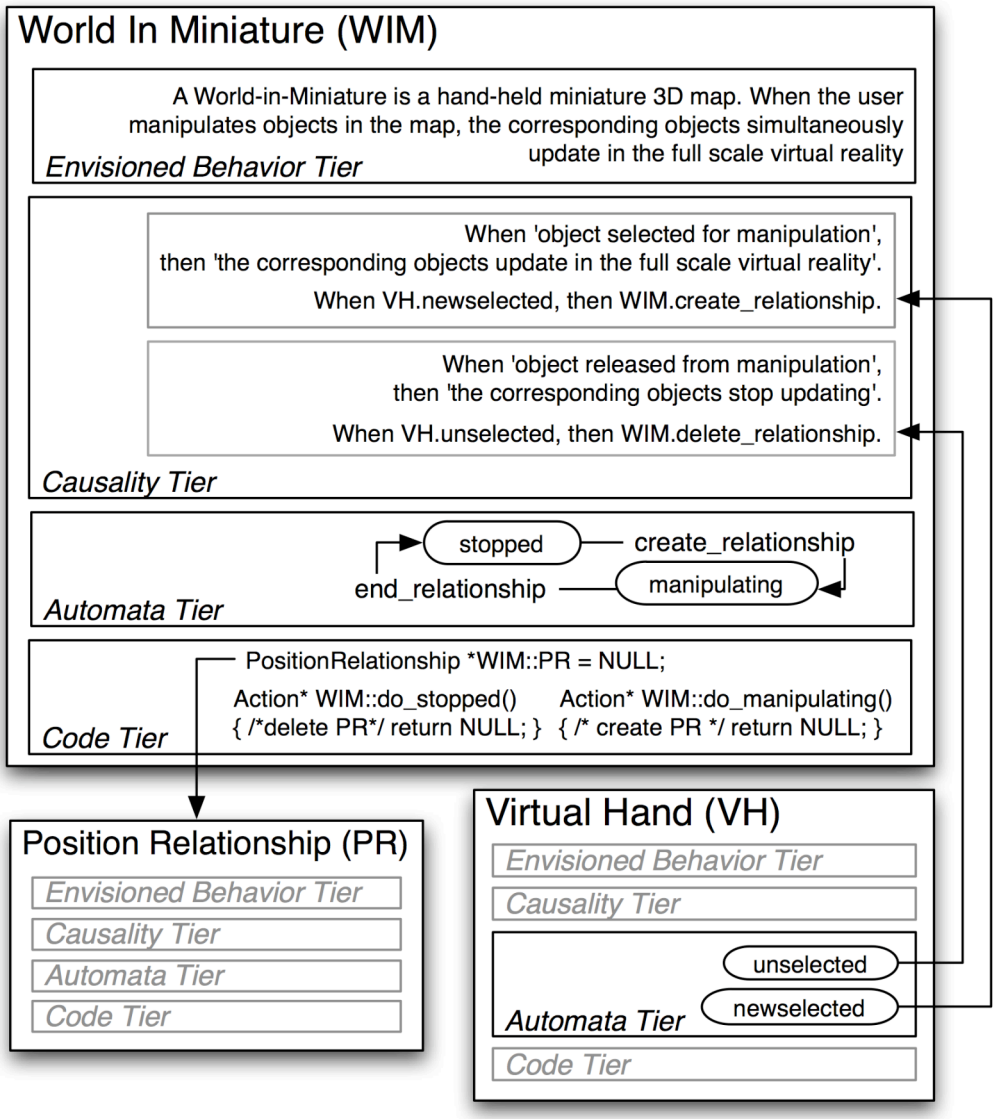

Figure 5-3. The four tiers of Chasm represent developer understanding as they implement a 3D interface, shown here for the WIM technique. Decomposing development in this way creates cohesive concepts, which are executed by Chasm as flows of events (refer to Figure 5-2).

The first step in a Chasm implementation of the WIM is the creation of a WIM concept. In this concept, the conversational domain language used to describe the WIM is placed in the envisioned behavior tier. This informs the deeper tiers with a

high-level, easily understood, language description. The first realization from this language is that a virtual hand is used to select and manipulate objects and that there is a position relationship between the virtual world and WIM proxy objects. These two ideas are realized as the concepts of virtual hand (VH) and position relationship (PR). Because these are common ideas in the domain, they have been implemented previously and are reused here.

The next level of understanding from the envisioned behavior is causality. The second sentence of the envisioned behavior implies that the grabbing of an object begins its manipulation and the release ends it, i.e. drops it at its current location. This understanding identifies the external influences in the system that impacts the WIM technique. This can be represented as abstract causal statements in the causality tier as:

- When "object selected for manipulation", then "the corresponding objects update in the full scale virtual reality".
- When "object released from manipulation", then "the corresponding objects stop updating".

These can even be shortened due to domain terms to the following:

- When "grab", then "manipulate".
- When "release", then "drop".

To realize these causal statements, the contexts of "grab" and "release" have to be connected to states in other concepts and "manipulate" and "drop" have to be connected to actions in the WIM concept. This is achieved by transforming the above abstract causal statements into the following causal statements:

- When VH.newselected, then WIM.create_relationship.
- When VH.unselected, then WIM.end_relationship.

This implies that the virtual hand's *newselected* and *unselected* states are used to cause create_relationship and end_relationship actions in the WIM. Chasm enables the developer to connect directly to the flow of behavior in external concepts, mapping that point in the external concept's execution to cause a response in the WIM. The coupling to these states in no way impacts the flow of behavior in the virtual hand concept, as it continues to execute normally. At this point, the developer has identified external influences to the concept and can begin to shape the internal flow of behavior in the WIM.

The automata tier represents the internal flow of the WIM technique. Where the decomposition by flow in Figure 5-2 identified seven states, now the WIM technique only has two: *stopped* (not moving the proxy object) and *manipulating* (moving a proxy object). The other states of the original flow still exist in the system but are handled by the virtual hand and the position relationship concepts. This greatly simplifies the WIM concept. The automata tier's transitions are then created so that its action of create_relationship sends the WIM into its manipulating state and the end_relationship sends it into its stopped state.

The code tier is used to implement in C++ the remaining behavior when the WIM enters one of its two states. At this point, the functionality required of each state entry function is clear because the flow has defined it. So, in the code tier, the state entry function of *manipulating* creates the position relationship concept, to update the virtual world's object, and the *stopped* state deletes the position relationship, which causes the object to stop moving. In Chasm's decomposition, the behavior of

the hand, button, relationships between objects and so on, are relegated to the concepts representing them. This greatly simplified the code tier of the WIM.

5.1.6.3. Execution of the Chasm Tiers

Analyzing the execution of the WIM implementation in Figure 5-4 shows how the execution reestablishes the flow of behavior in Figure 5-2. In this way, the Chasm implementation is cohesive, matching developer's understanding and simplifying development, and yet the flow of the WIM is recreated and able to be traced for debugging purposes.

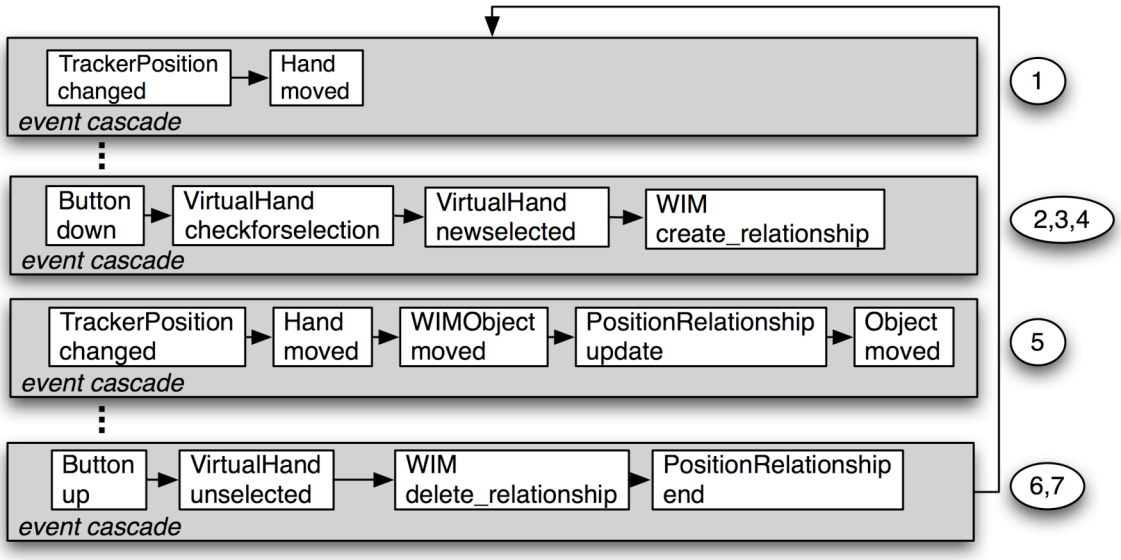

Figure 5-4. The processing of events in Chasm integrates the behaviors of multiple concepts as the flow in Figure 5-2 but allows development as cohesive concepts in Figure 5-3. The numbers on the right relate to the steps in the flow of Figure 5-2.

5.1.6.4. Extending the WIM to SSWIM

The tiers are also useful in the extending of a concept's functionality, shown in Figure 5-5 by extending the WIM into the SSWIM technique [Win06]. The SSWIM technique adds scaling and scrolling functionality to the WIM technique to allow the WIM to function in larger virtual worlds. Its functionality requires changing the flow of the WIM in step 5 and adding in two concurrently operating flows of behavior to allow the scaling of the world to change.

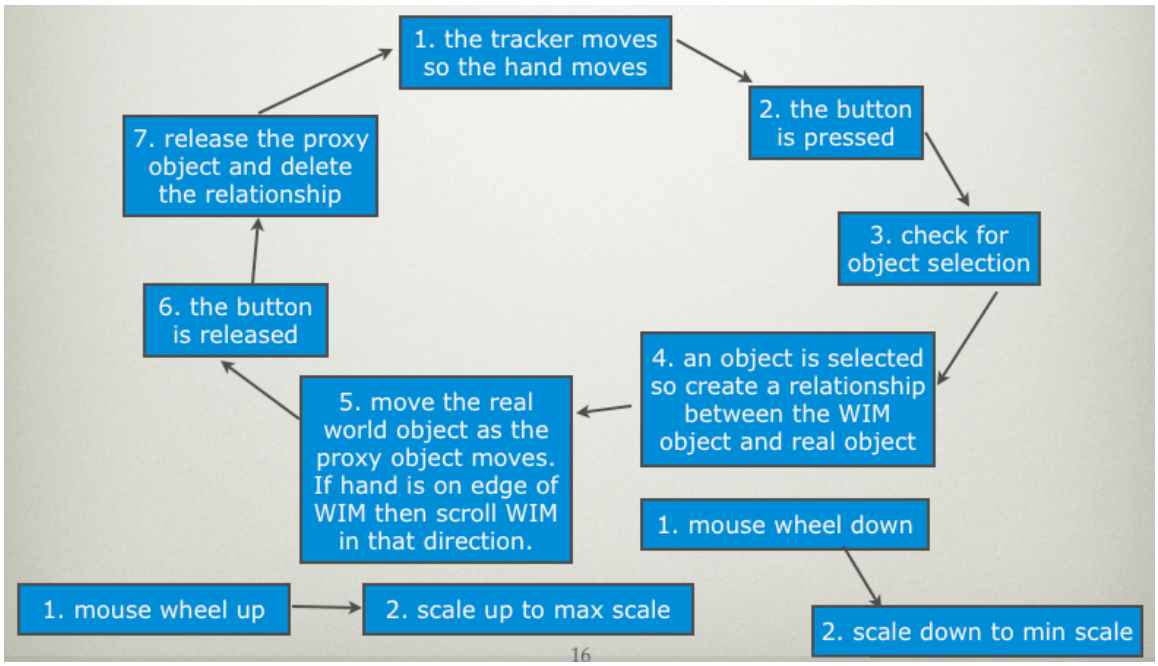

Figure 5-5. Extending the WIM to the SSWIM technique modifies step 5 of the WIM and adds two additional flows of functionality.

The implementation of the SSWIM technique proceeds in much the same way as the WIM (see Figure 5-6). First, the envisioned behavior of the SSWIM technique is set using conversational language: "Add scaling and scrolling to the WIM technique". This implies a WIM concept is required. Second, causality is identified and represented in the following statements:

- While "manipulating", then "check for scrolling and if hand outside WIM center then scroll".
- When "mouse wheel goes up", then "scale up (only to max scale)".
- When "mouse wheel goes down", then "scale down (only to min scale)".

These statements suggest that a mouse will be required so a mouse concept is added to the SSWIM. Additionally, we can see that scaling and scrolling are concurrently operating so we create another concept in SSWIM to handle the scaling functionality. This concept, ScaleObject, has the envisioned behavior of "Incrementally changing the scale of an object up to a max and down to a minimum scale." It can be implemented after the SSWIM technique has finished its implementation and even reused in future concepts. The SSWIM causal statements are then mapped to actions and states:

- While WIM.manipulating, then SSWIM.scrollupdate.
- When Mouse.wheel_up, then ScaledObject.scaling_up.
- When Mouse.wheel_down, then ScaledObject.scaling_down.

Notice how the causal statements do not have to map states to actions of its own. This is the case for mapping the mouse states to the ScaledObject actions.

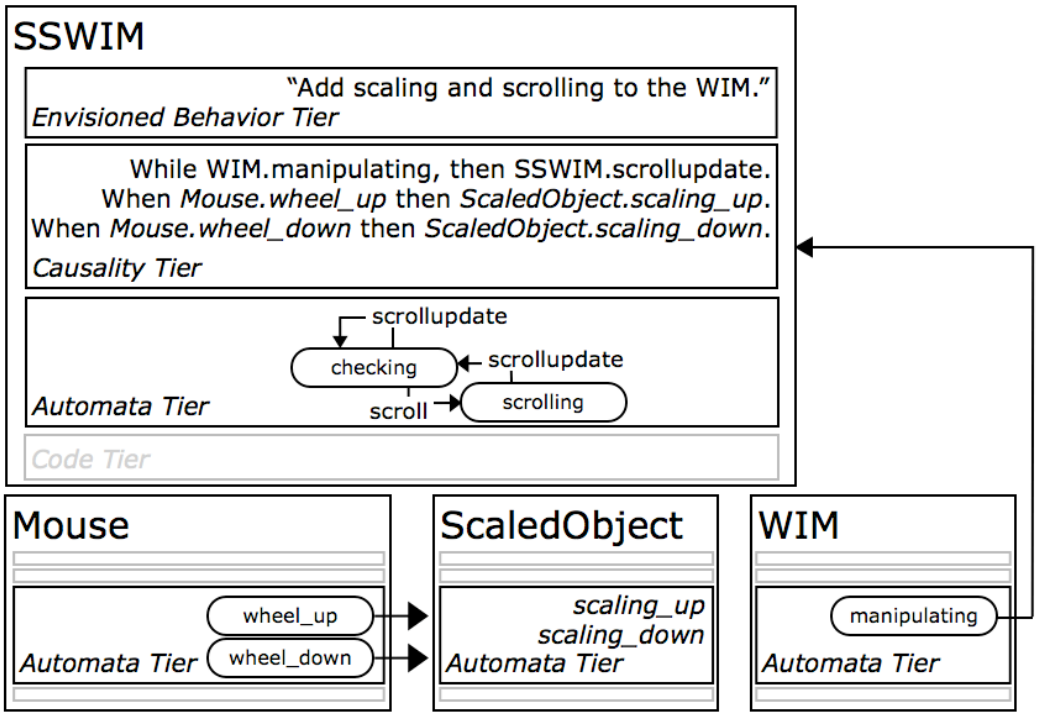

Figure 5-6. The SSWIM technique reuses the functionality of the WIM without modification.

As shown in Figure 5-6, the automata tier of SSWIM then becomes a few states into which the code to accomplish the behavior is inserted. In the code tier, the state of *checking* determines if the SSWIM is scrolling and the state of *scrolling* performs the actual scrolling and check to see if the SSWIM is continuing to scroll.

5.1.6.5. Adding Feedback to SSWIM

To improve feedback to the user in the SSWIM, the DirectionalArrow concept can use the SSWIM's states to implement its feedback functionality. However, the SSWIM does not currently have the needed states for DirectionalArrow as there is no ability for external functionality to know when scrolling starts and stops. So, the point-of-interest states of *startscrolling* and *endscrolling* are added to the automata tier of the SSWIM concept as shown in Figure 5-7.

Adding the DirectionalArrow feedback to SSWIM is with the envisioned behavior of, "Have an arrow point in the direction of the scroll while the user is scrolling" and is achieved with the causal statements of:

- When SSWIM.startscrolling, then "directional arrow appears".
- When SSWIM.endscrolling, then "directional arrow hides".
- While SSWIM.scrolling, then "align directional arrow with the direction of scrolling".

These actions are then realized in DirectionalArrow as:

- When SSWIM.startscrolling, then DirectionalArrow.show.
- When SSWIM.endscrolling, then DirectionalArrow.hide.
- While SSWIM.scrolling, then DirectionalArrow.update.

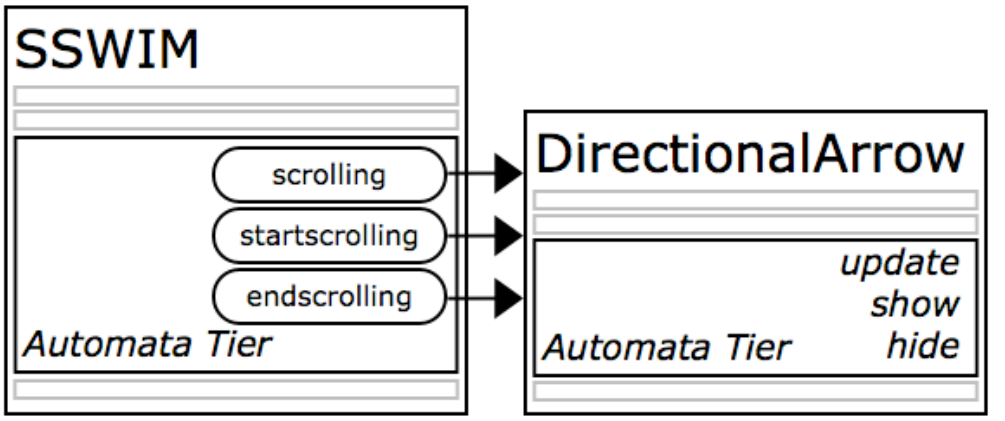

Figure 5-7. Additional point-of-interest states are added to the SSWIM technique to allow it to be more easily extended. Here, it is shown how a directional arrow, to assist in the feedback to the user, can use the SSWIM's states to implement its own functionality.

5.2. Developing with Chasm

Chasm developers require tools, documentation and assistance to support their use of Chasm in practice. Without supporting the developer, any new development approach is going to experience problems of unfairly competing against an established development approach with experienced users having robust tools. Without these approaches, Chasm would fail for reasons unrelated to its potential capabilities. Development with Chasm was improved by assisting the developer in: designing with Chasm, code tier management, runtime behavior and awareness, documentation and learning and a vocabulary of reusable concepts. Each of these are discussed in detail in the following sections.

5.2.1. Designing with Chasm

In Chasm, design is a process of composing and decomposing the envisioned behavior in the tiers of Chasm concepts. Design was assisted with a graphical tool, the ChasmGUI, which provides visualization and manipulation of concepts. A heuristic evaluation was performed on these tools.

5.2.1.1. Composition and Decomposition

Composition and decomposition in Chasm are designed to following good development practices and proceed along the dimensions in which developers think. To achieve this, there are multiple methods of composition and decomposition in Chasm. This includes object-oriented data encapsulation, inheritance and polymorphism as well as Chasm's tiers and component concepts. This allows the representation to change more readily during development because the representation and thinking are closer to the same form, i.e. cognitive directness [Hix93]. This helps to avoid what has been called the "tyranny of the dominant decomposition" [Tar99], which reduces reuse because the dominant decomposition makes the needed changes difficult. This occurs largely by the failure to externalize needed functionality or externalizing it in such a way that is not usable for a developer's needs.

The tiers of Chasm improve composition and decomposition. Chasm's top tiers create clear interfaces between the concepts in the system, with causal statements

connecting them, while lower tiers in a concept create flows with code describing the concept's processing. Thus, the envisioned behavior tier is a top-down decomposition along domain terms and developer thinking. Like object-oriented decomposition, it recognizes the important entities in the system. Then, the causality tier creates causal relationships between concepts, composing higher-level functionality. This allows non-essential functionality existing outside the concept to be easily incorporated into it. These top two tiers recognize the main entities of the system and specify the relationships between them. This approach has been thought of as identifying the nouns (concepts) and verbs (causal relationships) of system design [Mey97a]. This allows the concept to match object-oriented decompositions, event-based architectures and cross-cutting concerns.

The next tiers focus on the internal functionality of a concept. The automata tier decomposes the internal flow of the concept, without regard to the other concepts in the system. The code tier implements the concept in a procedural or functional decomposition and without regard to the rest of the system's code. This allows the code to be implemented while other concepts are still being designed. Additionally, placing the code tier last means it will be coded after the developer's understanding grew while progressing through the higher tiers.

In addition to matching developer understanding and assisting in the decomposition, the tiers match changes to the system:

- As new functionality is added to a system, the new concepts connect directly to the flow of the existing concepts, using the causality tier to connect to the internal flow in the automata tier.
- The stimulus of a 3DUI can change easily by changing the context of a causal statement in the causality tier. In this way, the stimulus to a 3DUI in one system can be easily changed in another system with a different set of requirements.
- A change in the automata tier has little impact on the external concepts in the system because external concepts only interface with the states of a concept and states do not change (see Modifications in Chapter 4).
- Because the flow of a system is in the automata tier, a change in the code tier can occur with minimal impact to external concepts.

5.2.1.2. Concept Inheritance

Concepts can inherit from one or more concepts, allowing a concept to incorporate the functionality in the parent's tiers as if they were defined in the child's. For example, envisioned behavior from the envisioned behavior tier is inherited, the causality tier inherits the causal statements and the automata tier inherits states, actions and transitions. The component concepts are also inherited. The state entry functions in the code tier function the same as C++ inheritance; so, the virtual child state entry functions replace those of the parent.

5.2.1.3. ChasmGUI: Visualization and Manipulation

ChasmXML, as with any XML representation, can be manipulated with a text editor or a generic XML editor but because of its domain specific meaning, a graphical editor called the ChasmGUI was created to assist developers. The ChasmGUI was implemented in Java and allows for the full manipulation and visualization of the ChasmXML representation.

As shown in Figure 5-8, the ChasmGUI is composed of six frames discussed below: palette, package, concept, envisioned behavior, action and component.

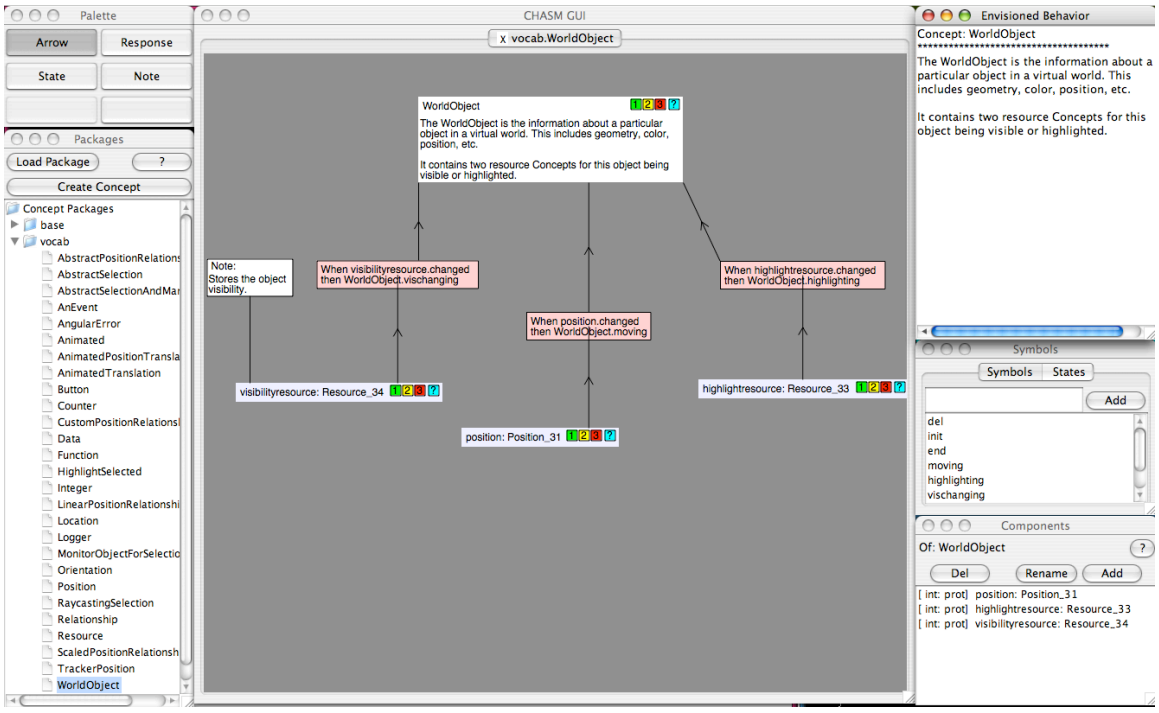

Figure 5-8. The ChasmGUI allows for the visualization and manipulation of the ChasmXML. The frames are: palette (top left), package (bottom left), concept (middle), envisioned behavior (upper right), symbol (since renamed to action frame) (middle right) and component (bottom right).

- Mid-display is the concept frame, the main visualization and manipulation frame (discussed further below). This frame is used in the layout of concepts, to add component concepts and responses and to add notes about component concepts. Though only one concept can be displayed, multiple concepts can be opened and toggled between using the tabs at the top of the frame.
- In the upper left, the palette frame is used to create responses and notes as well as switch between the arrow mode, used to manipulate concepts, and the state mode, for the creation of states.
- In the lower left, the package frame is used to load packages of concepts, creating a new concept and opening an existing concept in the concept frame. Since developers often separate concepts into different directories to organize the development, a package is a directory that contains concepts. This allows developers to load and unload concepts as needed and maintain their organizational structure for the system.
- In the upper right, the envisioned behavior frame is used to display the behavior of the highlighted items such as concepts, states, actions and the like. This is shown for WorldObject.
- Mid-right, the action frame is used to create and modify actions in a concept. Note that actions were originally called symbols but were renamed and the ChasmGUI was never updated.
- Bottom right, the component frame is used to create, delete and rename component concepts as well as display them, hide them and change their ordering. Hiding components is useful when there are too many components shown.

The visualizations of concepts and component concepts in the concept frame, shown in Figure 5-8 and Figure 5-9, have three different representations. These representations are able to be toggled by the icons, labeled 1,2 and 3, in the upper right of the concept: 1) compact, 2) envisioned behavior and 3) automata. These three representations show an increasing amount of information about the concept allowing the developer to access the proper amount of information for their needs. The last icon '?' is used to open a link to the Chasm help Wiki (see Documentation and Learning below). Also shown in Figure 5-8 are the causal statements between WorldObject and its component concepts.

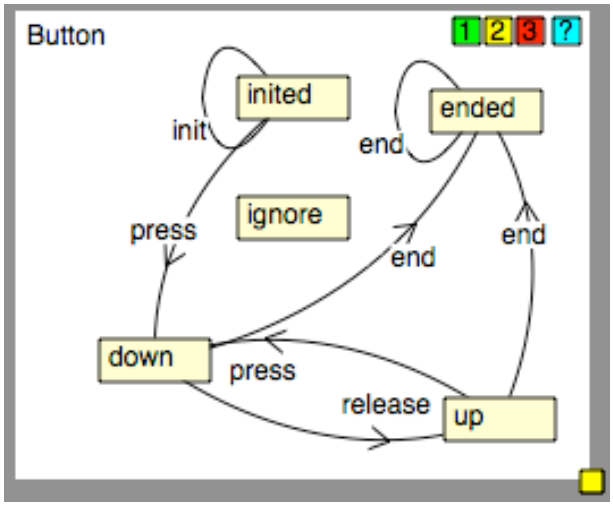

Figure 5-9. The button concept's visualization shows that it has two states of down and up with actions press and release. Other states and actions are standard for all concepts. Additionally, icons in the upper right allow for different views of the concept along with access to help.

The three representations of a concept are:

- Compact. This lists the concept's name and if it is a component, its name in the concept. For example, "position" is the name of the Position concept in the WorldObject in Figure 5-8. The compact representation also shows the four upper-right buttons. This representation is the default representation for component concepts. In Figure 5-8, the three components of WorldObject, visibilityresource, position and highlightresource, are in the compact representation.
- Envisioned behavior. This is the same as the compact representation but shows the envisioned behavior as well. In Figure 5-8, the WorldObject is in this representation.
- Automata. This shows the primitives in the concept's automata tier (ex. the Button concept in Figure 5-9). This is the default starting representation for the main concept in the concept frame.

The ChasmGUI is also is able to manipulate several other criteria regarding a concept. For example:

- Access of the states, actions and components allows developers to encapsulate information in the concept. Access is modifiable by the developer using the action and component frames. The levels of access are public, protected and private, matching C++ access levels.

- The developer is able select a concept's parent concept. The inherited automata primitives are displayed in the concept frame with inherited states in a slightly different color. Inherited components are listed as inherited in the concept frame.
- The component initialization form (CIF) for component concepts can also be set.
- The layout of the component concepts and states in a concept are reorderable. In ChasmXML, this is the manipulation of the x, y, width and height attributes.
- Notes, with further documentation, are attachable to component concepts and can be positioned in the display. These are saved in ChasmXML for the component.
- Causal statements are managed and can be created between automata states and actions in a concept and its component concepts or even components of component concepts.
- Transitions in a concept are modifiable. A table lists every state and action in a concept and a developer sets the state to transition to for the state/action pair. These transitions are then displayed in the automata representation (shown in Figure 5-9).

The ChasmGUI allowed for the manipulation of most every aspect of the ChasmXML specification as well as the visualization of concepts created by the developer and in the vocabulary.

5.2.1.4. Heuristic Evaluation of the ChasmGUI

A Heuristic Evaluation using Neilson's criteria for evaluation [Nei94] was performed to ensure a minimum level of usability of the ChasmGUI for developers. The goal was not for the creation of a walk-up-and-use application, as the ChasmGUI would only be used by long-term developers (see Evaluation of Chasm in Chapter 6), but for an application that allowed a moderately experienced developer to work with Chasm. From this evaluation, it was concluded that the ChasmGUI was successful in this goal though several improvements were suggested and implemented in the ChasmGUI in response to the evaluator's comments.

In the evaluation three usability evaluators were each given a six page tutorial on Chasm and COD, along with five scenarios to work through to evaluate the GUI. These documents are attached in Appendix E. The evaluations took between 1.5 to 2.5 hours. Though additional evaluators were planned originally, findings were saturated after three evaluators and the evaluation was concluded.

Two minor modifications to the ChasmGUI were made as a result of the evaluation. First, the evaluators believed state creation was through the actions frame. Because of this, a tab called "states" was added to the window's frame (see Figure 5-8) that displayed instructions on how to create a state. Though not ideal, this solution addressed the issue and developers had no further problems with this feature (state creation was achieved by clicking on the concept itself while in the automata representation or with the shift key pressed). Second, evaluators were not sure how to close a concept in the concept frame, so, an icon of an 'X' was added to each concept's tab. This was achieved previously by double clicking on the tab, though this was not immediately apparent in the first design..

5.2.2. Code Tier Management

The code tier has its own developmental issues. Below, tools, component initialization form, builders and supplemental files are discussed in turn.

5.2.2.1. Tools

Several tools are used, in addition to the ChasmGUI, to manage the transformation of Chasm's representation into a machine executable form. This is in addition to other tools such as XCode, eclipse, a compiler or make. The Chasm tools used include chasm2C, chasm-config and chasmprocess (see Figure 5-10).

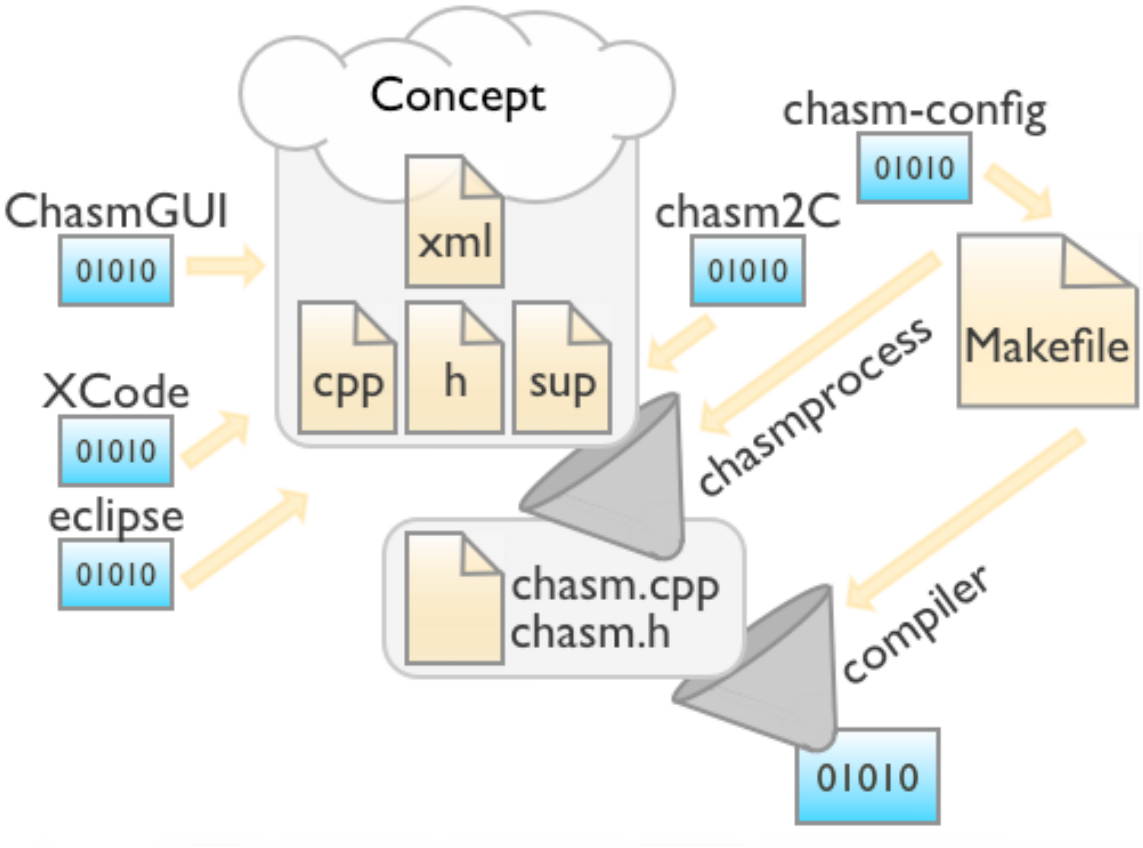

Figure 5-10. Several tools were created to manage Chasm's build process and management. This included the ChasmGUI for the ChasmXML manipulation but also chasmprocess and chasm-config for code transformation and makefile information.

The code files for a concept, though they could be carefully created by hand, are generated by a command-line tool, *Chasm2C*. This initial code generation has preprocessor statements and is shown in Appendix D in its three forms: C++ and header, builder and supplemental files (see below for information about Builders and Supplemental files). By default, Chasm2C generates files with help interspersed in the representation to assist developers. The developer then implements state entry functions, methods, data, constructor and destructor calls in the generated code. Existing development environments such as XCode or eclipse can be used for this. At the same time, the ChasmXML can also be manipulated.

Once a developer decides to merge the two representations of a concept into one, a preprocessing step is performed by *chasmprocess*. This tool first reads and parses the ChasmXML file, validating that it is properly formed, and then extracts the concept's and its parent's and component concept's information. It then opens the code files, replacing the preprocessor statements and generates the final C++ and header files for the compiler. These generated files are ready to be compiled without any further actions on the part of the user. The special cases handled by

chasmprocess include:

- Determining and including the appropriate header files for parent concepts and component concepts, as well as their parent and component concepts, ad infinitum.
- Line numbers in the output files are matched to the original files using cpp preprocessing statements. Without this step, the line numbers given by compiler errors would not match.

A final tool, *chasm-config*, writes configuration information to the command line such as version number, installation location and compile and linking information. This is useful for Chasm's integration with existing build procedures such as Makefiles. It also simplifies versioning and upgrading. This is modeled after a similar program in the GIMP project (http://www.gimp.org).

5.2.2.2. Component Initialization Form

Component Initialization Form (CIF) is the method of specifying how a component of a concept is initialized and in some cases deleted (see Partial Deconstruction Management below). The four CIF types are: external, internal, potential and vacant. External CIF means that the component exists previously in the system and is only passed to the concept as a constructor parameter. Internal CIF means that the component is internal and created by the concept during the initialization process. It does not need to be passed as a parameter to the concept's constructor and on deletion, it is told to delete as well. Potential CIF means that the component is potentially passed into the Concept--it can be NULL at initialization time. If it exists, it is created externally. Vacant CIF means that the component will be set later and not dealt with in the constructor or on deletion.

CIF serves three different purposes. First, CIF is used in the preprocessing step of chasmprocess to generate code that incorporates the component concept. If a concept is external or potential, then the constructor should have a parameter for the component. Additionally, potential CIF means that the default value for the parameter is NULL. Second, CIF is used to automatically delete component concepts, (if they have a CIF of internal) when the concept deletes. Third, as developers commonly forget to instantiate or pass component concepts the CIF is used to generate instantiation-time checks on component concepts. These checks halt execution and give warnings if the components are not handled in accordance with their CIF settings. This occurs frequently during development. Without this check, hard to identify terminal errors occur because the source of the error is at concept instantiation time while the realization of the error is at runtime.

5.2.2.3. Builders

A builder is a useful code tier abstraction, named after the Builder pattern [Gam94] that separates concept from toolkit-specific code. In a builder, the main concept is provided with a builder that holds toolkit specific functionality. When a method is called on the concept, the concept can use its own implementation for that method or forward that call to the builder. In this way, Chasm allows the same concept to be implemented for different toolkits. It also results in better inheritance. In Figure 5-11, a concept’s normal inheritance hierarchy is shown on the left while the builder inheritance hierarchies are mirrored on the right. Without builders, as in Figure 5-12 (below), concept code and toolkit specific code mix, making portability problematic, and any inheritance requires duplicating the concept code across the toolkit specific inherited builders. Given the potential large use of inheritance in Chasm, builders are required.

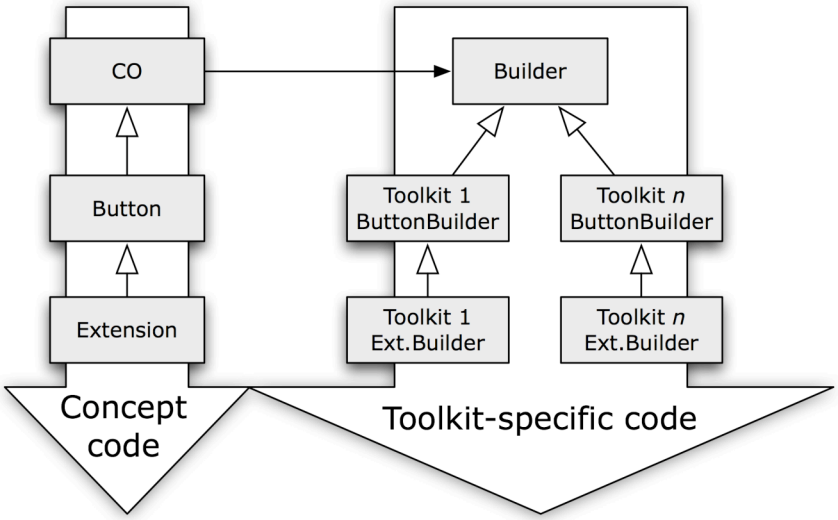

Figure 5-11. A builder separates concept code from toolkit code. In addition to simplifying a concept, it helps make concepts portable and inheritance hierarchies simpler. The Concept code on the left is the same for all toolkits.

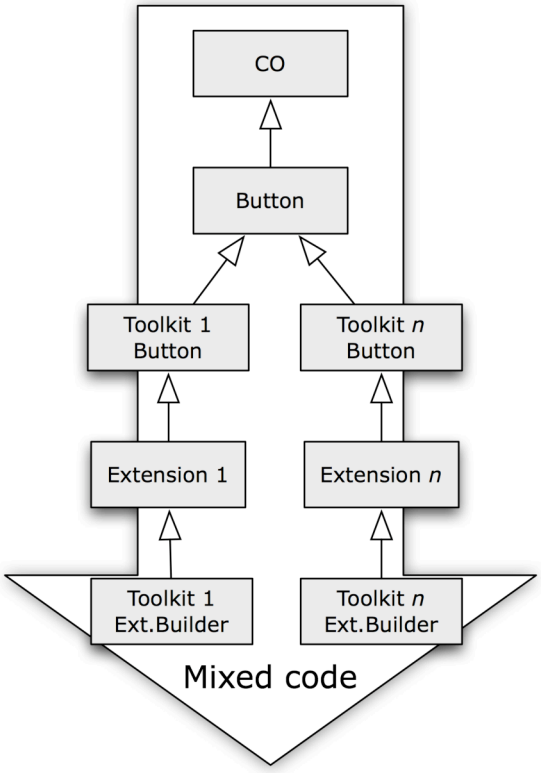

Figure 5-12. Without builders, the inheritance hierarchy is a mix of toolkit and concept specific code. Note how an extension to the Button concept has to inherit from every toolkit-specific class that implements a button. Also, the extension code has to be duplicated for every toolkit.

Builders are useful for increased portability of a concept that leads to increased longevity. First, a builder increases a system's portability by only having to swap builders to use a new toolkit. If a toolkit is not supported by Chasm, only the builders for the concepts have to be implemented. One feature of this approach is that as a concept composes higher-level meaning, concepts require less toolkit specific code because the meaning is composed from other concepts, not toolkits. Second, a builder increases a concept's longevity through its portable. In this way, a useful concept can be ported to a new toolkit by only the implementation of a builder. This allows a larger community to use and support the concept. This portability allows a community to shift to new tools and take their concepts with them. The community then does not have to define itself by its tools, but by what it does and the problems it addresses. In addition to being a much better definition of community, the community does not have its lifespan tied to that of a tool.

As an example, builders were used in the implementation of the WorldObject concept in SVE. A WorldObject represents an object and its geometry in a 3D environment. This includes functionality for position and visibility among other things. Using a builder, the functionality that interacts with geometry and the scenegraph was translated into SVE function calls. To use a WorldObject in another tool, such as VRJuggler or DIVERSE, only the toolkit specific code in the builder would be affected, not the original WorldObject concept. In this way, a system can be ported to a new toolkit with minimal effort.

5.2.2.4. Supplemental Files

Supplemental files were created to address the diffuseness and constant switching between header and code files during development. In the Chasm2C generated code, there were ten locations where developers needed to place code. In the header and code representation, these important locations were hard to find, even with documentation, and developers could potentially make changes that broke the preprocessing of the files, such as deleting the preprocessor statements. Because of this, supplemental files were created with the ten sections where developers need to place code. Developers can edit the supplemental files without having to deal with two auto-generated files with their extraneous code. Where the preprocessing step inserts ChasmXML information into code, a supplemental file inserts C++ code into the generated code and header files. This was implemented using the cpp preprocessor. See Appendix D's sample supplemental file for more information.

5.2.3. Runtime Behavior and Awareness

In object-oriented and structured programming, the developer can trace method calls and code to understand a system's runtime behavior. Chasm, however, weaves concepts together at runtime (see Action-Processing Dictum below). As such, the understanding of a system's flow and awareness of the state of the system at runtime are important to make explicit for debugging and testing. Because of this several additions to Chasm enable the developer access to this information: self-awareness, visualization, event tracing and an interactive shell. These are discussed, in turn, below.

5.2.3.1. Self-Awareness

A Chasm system and its concepts are instrumented to be self-aware, important for the to achieve the functionality that follows. To achieve this self-awareness, every concept registers with the system it is in and every primitive in each tier is a 1st class object in a concept, registering with that concept. Each concept is given a unique identifier in the system, the file and line number it was instantiated in, it knows its

class and can be given a unique name. As such, an individual concept can be found easily and can report information about its states, actions, causal statements and transitions. Additionally, each concept is required to implement a method displayMeInternal that enables developers to specify how they want to report information about the concept's current state (state also being the state in the code tier). So, a Button concept reports if it is up or down and a Position concept, representing a point in 3D space, would report its position in six dimensions. The displayMeInternal method could be called explicitly or at runtime using the interactive shell (see 5.2.3.4. below).

5.2.3.2. Visualization

Visualization of a Chasm system at runtime used two diagrams created with the Graphviz tool (http://www.graphviz.com). A transition diagram, shown in Figure 5-13, is of a single concept showing the states and the transitions between states by actions. This diagram also shows the envisioned behavior and documentation on each state.

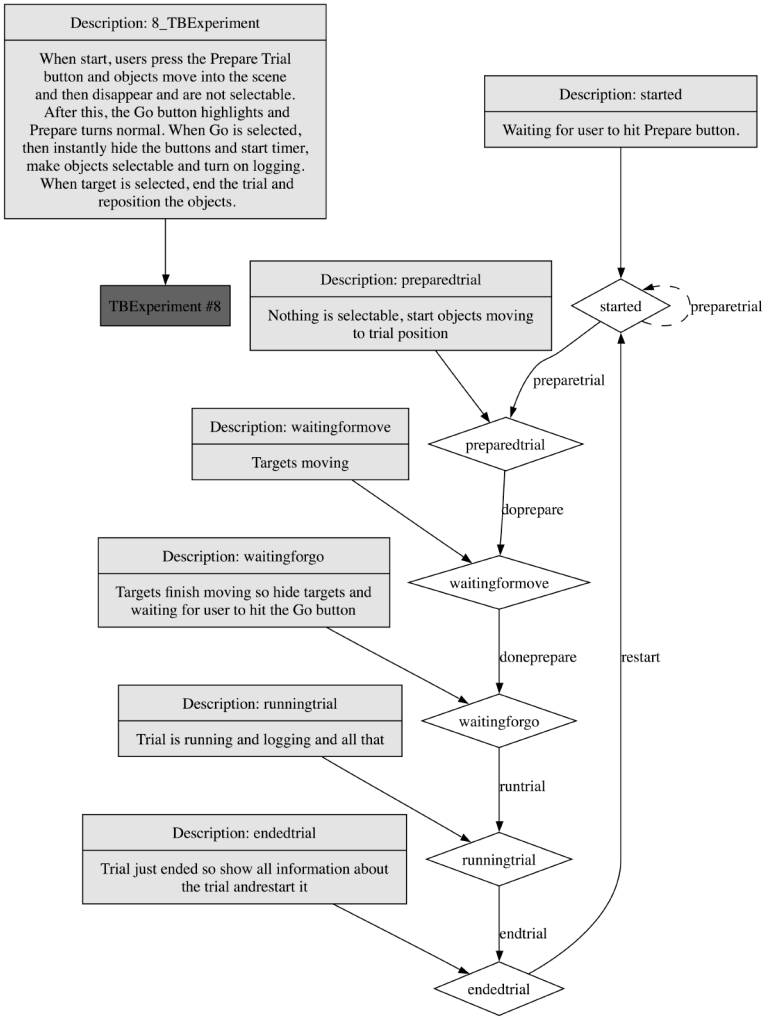

Figure 5-13. Chasm can generate diagrams like the transition diagram here that display the state machines, its states, actions, transitions and documentation.

A relationship diagram, shown in Figure 5-14, traces the stimulus and response of causal statements in the causality tier. The diagram is created by selecting a concept

and fanning-up (finding the states in other concepts that generate the concept's actions) and fanning-down (finding the states in the concept that generate actions in other concepts) its causal statements. Both diagrams were taken from the case study VDA1 discussed in Chapter 6.

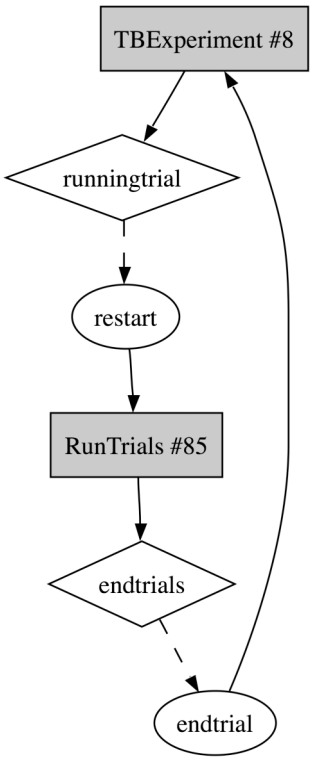

Figure 5-14. A Relationship diagram shows the connections between concepts by displaying the states as diamonds, actions as circles and the ability for a state to send an action as a dashed line. Not all states or actions are shown in this diagram, just those that connect the concepts TBExperiment and RunTrials from case study VDA1 in Chapter 6.

The two diagrams show different types of information. The transition diagram shows the inner workings of a single concept and can be useful for documentation purposes and learning. The relationship diagram shows the concept as used in an executing system. This information is valuable when the system complexity increases and becomes too complex for the developer to maintain in their head. These visualizations were created before the ChasmGUI and were influential in its development.

5.2.3.3. Event Tracing

Because Chasm reconstructs the flow of events at runtime, the tracing of events is valuable for confirming the proper execution flow. An event in Chasm is the point in time when an action is processed in a concept. With tracing, the concept receiving the action, the action and the states transitioned from and to are streamed to the command-line as shown in Figure 5-15. This enables the developer to see that events are being processed as they would expect, or if not, where the flow goes awry.

```
 in state 'ChasmTest_2.inited'
Event: (0) Cas(2) ChasmTest_2->'changing'
    ChasmTest_2 from inited to first via 'changing'
in first state
Event: (0) Cas(3) ChasmTest_2->'changing' from: 'ChasmTest_2'
    ChasmTest_2 from first to second via 'changing'
in second state
 in state 'ChasmTest_2.second'
Event: (0) Cas(4) ChasmTest_2->'changing'
    ChasmTest_2 from second to second via 'changing'
in second state
 in state 'ChasmTest_2.second'
```

Figure 5-15. Tracing statements, emphasized with a red and black background color, show the order of event execution as well as the source of actions. This is the flow of Chasm's tutorial 1.

When every event in a system is displayed, the flow quickly becomes overwhelming. To address this, system and concept-level control over the granularity of events was added. This includes the ability to turn on and off the event tracing before and after a critical section in the system. Tracing was controlled: at compile time by procedural calls, at load time by setting environment variables, by reading-in a configuration file and at runtime through the interactive shell. This gives the developer options to control the tracing output.

5.2.3.4. Interactive Shell

An interactive shell was added to Chasm, allowing a developer runtime control over the runtime behavior and awareness tools discussed above. The shell allows the developer to create the two diagrams discussed above (Figure 5-13 and Figure 5-14), change the event tracing, list the concepts in the system and display the state of a single concept in the system. The shell could be entered programmatically by a method call, by pressing Ctrl-C at runtime or on a system error before crashing. Once the developer finishes with the shell, execution is returned to the system to continue processing (or crashing if entered by a system error).

The shell is also extensible, allowing for the addition of new commands through a simple API. For example, to assist with the development of virtual environments in SVE, several commands were added to display SVE objects, the system's state and even manipulate SVE -- such as moving objects in the environment, changing the scenegraph or loading new objects from files.

5.2.4. Documentation and Learning

As a new approach to development, tools to assist developers in learning Chasm were created and took the form of documentation, tutorials and a Chasm Wiki. These were used in evaluation experiments in the next chapter. Documentation took the form of an early workshop paper [Win05] as well as a six-page Chasm manual (see Appendix E). These documents were the first step in understanding Chasm but lacked any real depth and step-by-step instructions in its use.

To fulfill the needs of the developers in the evaluation, a communal approach to documentation was taken in the form of the Chasm Wiki. As Chasm is a departure from traditional development approaches, several unanticipated questions arose over the course of use from developers. The answers to these questions were

incorporated into the Wiki pages for reference. Developers were encouraged to add to the Wiki, though developer involvement was slight. This may be because the developer's deadlines did not allow them the time to update the Wiki.

The Chasm Wiki can be found at:
http://people.cs.vt.edu/~cwingrav/wiki/index.php/Chasm[4].

Additionally, there are several examples and two tutorials packaged with Chasm. The examples are simple one or two line demo applications. The first tutorial focuses on the creation of a simple concept and the code generation process. It also explains the debugging of a Chasm application. The second tutorial involves a virtual environment created with SVE where the user points a ray at an object, pushes a button and the object travels to a new position.

Lastly, though a Listerv "Chasm-dev" was created and advertised it was not used.

5.2.5. Vocabulary of Reusable Concepts

The vocabulary of Chasm is a growing set of successfully used concepts, pooled together to encourage reuse in new Chasm systems. As the name implies, the collection of concepts is expected to grow and change like a language with useful concepts being refined and reused as per the community principle of Concept-Oriented Design. As this is the beginning of a collection of managed concepts [Win06b], which adhere to a certain level of stability, robustness and documentation, future work remains to develop additional concepts and create a better organization. This would include a means of distributing the concepts as well as versioning and bug reporting.

The vocabulary is currently split into the base and vocab directories. The base directory is composed of concepts so fundamental that their existence requires changes in the Chasm architecture to exist. The vocab concepts are the pool of concepts useful in multiple projects. There are six concepts in the base and twenty-eight in the vocab. The concepts in the vocabulary were used in the case studies of Chapter 6. A full listing is found in Appendix F with a few discussed below. These are discussed in terms of the issue their functionality addresses.

5.2.5.1. Data

> *Issue: Many times, changes to data sparked some reaction in a system.*

A Data concept represents some form of stateful value. It has accessor and mutator methods that are generic for different data sizes. Examples of inheriting concepts are Integer and Float, which specialize the accessor and mutator methods. External concepts can attach to the Data concept's state *changed* for notification when the data changes.

5.2.5.2. Set and MirrorSet

> *Issue: Developers commonly discussed behavior as groups of items.*

Developers naturally declare groups of items and operations on those groups as identified by [Pan00][Mil74][Mil81]. As such, a Set concept was created to hold multiple concepts. It has the states of *added* and *removed* for external concepts to build behaviors in relation to the concepts in the set. Additionally, there is an option

4 Because of a server upgrade of PHP, the Chasm Wiki no longer functions. The pages were salvaged using the Internet Archive (http://www.archive.org) and are currently online at the given URL but are not displayed as they were during the evaluation of Chasm.

to add a function to sort the items in the set.

A second type of set was added called a MirrorSet. Inheriting all the Set's functionality, the MirrorSet creates a new concept in the MirrorSet for each item in a Set being mirrored. The effect is an association of additional data per each concept in the mirrored set. This was used in Raycasting to create a concept holding the selecting ray's angular error for each selectable object of the technique. The benefit was that the angular error, which was used in several computations, was only computed once per selectable object and placed, as a Float concept, in the MirrorSet.

5.2.5.3. Resource

Issue: Restricting access to a critical section of code or data is a common problem.

A concept in Chasm must continue to function in new systems where it cannot account for all access to its critical code or data. To assist the developer, the Resource concept manages access to a Data concept with access control and priorities. In this way, concepts can continue to function without knowledge of other concepts in the system because the Resource concept mediates between them. A Resource can handle access requests in a stack or queue form. Additionally, different access priority levels can be created.

5.2.5.4. Switch

Issue: Functionality becomes active or inactive as the system state changes.

A Switch concept controls the activation of a causal statement. When a causal statement is stimulated, the action goes to the Switch that, if on, sends the action to the responding concept. In this way, it is possible to disable and enable functionality in the system. The alternative is to remove the causal statement temporarily but this creates problems when visualizing the system, as the causal statements would only be shown when attached. With Switches, the visualizations will always show the causal statements.

5.2.5.5. Conjunction

Issue: In many cases, causality is dependent on more than one state.

Conjunctions enable more expressive causal statements such as "when X_1 and X_2", "when X_1 or X_2" and "when X_1 and (X_2 or X_3)". When all boolean conditions are satisfied and the context returns true, the action of the causal statement is sent.

5.2.5.6. Animate

Issue: In 3D interaction, many actions occur over time.

The Animate concept makes the declaration of continuous behaviors easier by assisting the developer in the discretization of a behavior that occurs over time. This is an important concept in 3DUIs [Jac99][Blo08]. The concept continuously updates over a developer-specified period of time with a developer-specified timeslice in milliseconds. The Animate concept then tells the developer the percent of animation time since the last timeslice so they know how much change to make in the continuous action. For example, an animation that moves an object 10 feet over 5 seconds, with 2 seconds since the last update, would tell the developer they need to move the object 2 seconds divided by 5 seconds, or 40% of its distance. This is 40% of 10 feet or 4 feet. There have been several concepts that have extended Animate, overriding its *animated* state's entry function for their intended behavior.

5.3. Chasm Action Processing

A critical component of Chasm is its ability to process actions in an order that maintains the order the developer intended, even when concepts are reused in new systems. A Concept-Oriented Design system that does not properly uphold an order of processing in the way the developers intend would not be useful because implemented systems would not be reliable. Event execution in systems like Statecharts and its variants is problematically difficult, and in some cases arbitrarily implemented, with similar systems managing events, *events displayed in the same syntax even*, in different orders [von94].

Chasm's action processing flows from the action-processing dictum, which is discussed below. This includes a series of ordering rules to implement it and examples showing it in action. Problematically, cycles can form when processing actions and several special cases have to be managed.

5.3.1. Action-Processing Dictum

The action-processing dictum of Chasm specifies the ordering of events in a Chasm system. In Chasm, an action is the automata tier and an event is how that action is being sent to the concept. The ordering of the events are defined such that concepts continue to function as the original developer of the system intended, despite change to the system, repurposing of the concept or reuse in a different system. This avoids side effects during development where new functionality reorders the existing action-processing order. Careful management of stateful data achieves this. If any stateful data of a concept, in the automata or code tier, is allowed to change before other concepts in the system are allowed to respond to the original stateful change, there is a potential for the system to behave erratically as well as the potential for any additional concept to break the existing functionality. Making the ordering of events problematic is the concurrent execution of the concepts and the fact that the causal relationship statements only specify a partial ordering of events.

The action-processing dictum is: *An action must be processed in Chasm before any stateful data that a concept depends on changes.* The enforcement of this affects all event handling structures, order and algorithms. Successful enforcement of this dictum removes the burden of ordering events from the developer.

5.3.2. Ordering Rules and Event Types

The four ordering rules of the action-processing dictum maintain the proper order of the five event types in Chasm. The Chasm event types are: response, continuation, invoked, continuous and future, discussed below.

- A *response* event is created by a causal statement as a response to a concept entering a new state. It is the most common event in Chasm.
- A *continuation* event is an event internal to a concept that continues the flow of behavior inside the concept. This allows a state in a concept to act as a decision state, using the state entry function in the code tier to decide the flow of events. This removes complexity from the automata tier, simplifying its notation.
- An *invoked* event occurs when the code tier causes a change that requires notifying the automata tier with an action. The event is handled immediately.
- *Continuous* and *future* events are not in the discrete timeline but the linear timeline. A *continuous* event continues a continuous behavior, i.e. one happening over a period of time, letting the event occur in the next processing of events. In

Chasm, this is how continuous behaviors, such as an animation, are discretized. A *future* event is an event set to occur at a point in the future. It executes when that time occurs.

The ordering rules for these events are; 1) partially ordered responses, 2) responses before response, 3) responses before continuation and 4) immediate action execution. Examples of these event types handled by these rules and the method of enforcing these rules are covered in the section Implementing the Ordering.

5.3.2.1. Partially Ordered Responses

In the action-processing dictum, responses for a single state happen in a partially specified order. Chasm assumes this order to be that of the requests by concepts. So, if a system is created where the causal statements are created in the order of: "When A.state, then B.b", "When A.state, then C.c" and "When A.state, then D.d", then the actions will occur in the partial order they were requested: B.b, C.c and D.d (see Figure 5-16 below).

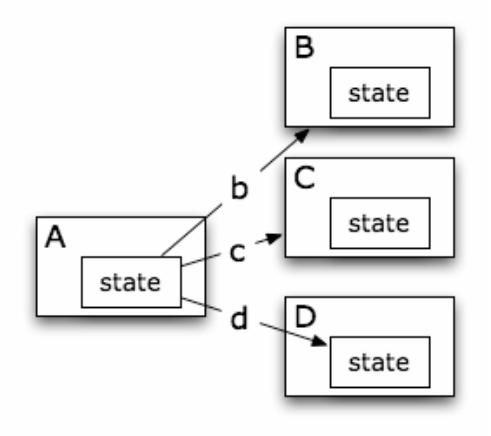

Figure 5-16. The order of events on entry to A.state is (B.b, C.c and D.d).

Chasm also has functionality to reorder responses such that an action has to occur before or after a list of other actions in a state's response. From the example above, it is possible to say C.c before B.b and D.d or C.c after D.d. The resulting orders would be (C.c, B.b, D.d) and (B.b, D.d, C.c). Note that these are reorderings and not rules permanently applied to the event orderings. It is also worth mentioning here that Chasm cannot specify a non-deterministic ordering of events and that this functionality is beyond the scope of this work.

5.3.2.2. Responses Before Response

A response's responses finish before the previous responses. So, given "When A.state then B.b", "When A.state then C.c" and "When B.state then D.d", then when A.state is entered, the order of events is (B.b, D.d, C.c), as shown in Figure 5-17.

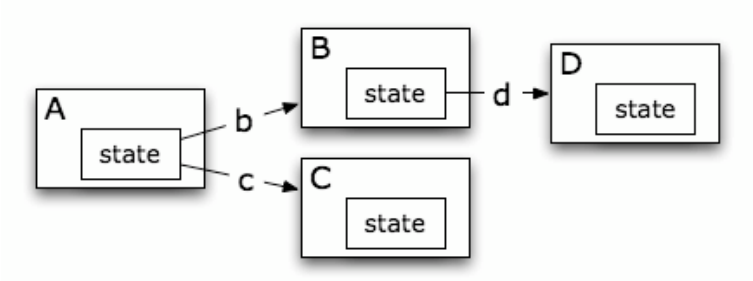

Figure 5-17. The order of events on entry to A.state is (B.b, D.d, C.c).

This rule addresses ambiguity. Above, the developer only specified partial orderings of (B.b, C.c) and (B.b, D.d) so possible orderings are (B.b, C.c, D.d) or (B.b, D.d, C.c). The rule responses before response says that (B.b, D.d, C.c) is correct. The rationale for this is: if C.c is handled before D.d, the potential exists for C.c to change stateful information that D.d operates on. There will always be the potential for newly added functionality to change existing functionality, because stateful information exists in the code tier, but the change should be limited to the newly added functionality.

For example, below are two cases in which a new concept is added to a fully tested and released system (refer to Figure 5-17).

- Case 1: Consider a fully tested and released system existing only as A, B and D in Figure 5-17. Now, C is added with the potential to change stateful information, either globally or in A, B or D. Because C.c is handled after all the responses to B.b (i.e. D.d), the changes C makes to the system are irrelevant to the existing functionality (it was already processed). Only C is responsible for any changes it makes to the system.
- Case 2: Now, consider a fully tested and released system existing only as A, B and C in Figure 5-17. Now, D is added with the potential to change stateful information, either globally or in A, B or C. Changes to A and B are irrelevant because they already processed. D can change things for C (because it processes before it) but this is fine. Since D is newly added to the system, its changes are expected to be either the correct behavior of D or a bug requiring correction in D. This means that changes to the system are limited to that caused by D, the newly added functionality.

5.3.2.3. Responses Before Continuation

A state entry function can return an action to itself as a continuation event which occurs after all responses finish. So, given "When A.state then B.b", "When B.state then D.d" and "When A.state2 then C.c" and A.state returns A.a, then when A.state is entered, the order of events is (B.b, D.d, A.a, C.c), as shown in Figure 5-18.

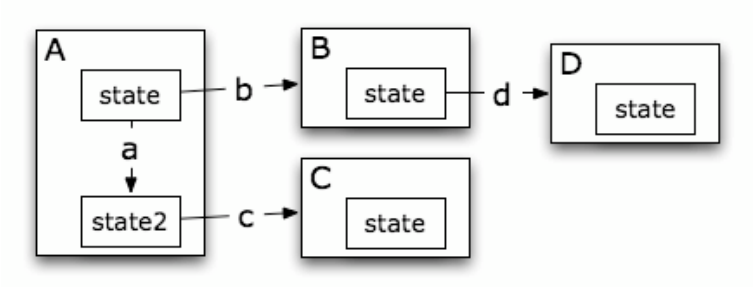

Figure 5-18. The order of events on entry to A.state is (B.b, D.d, A.a, C.c) where A.a is a continuation event.

If the continuation event occurs before the response events, then responding concepts would not be assured that they are responding to a concept just entering a new state. Consider a system A, B, D where C is added (as in Figure 5-18). If the continuation event A.a is processed before B.b then B and D, which expected to respond to A.state, will process when A is in state A.state2. Additionally, because of the rule responses before responses, C.c could run before B.b or D.d. This allows C the possibility to create changes to the system that could affect B and D before B and D are allowed to process. Therefore, processing the response event B.b before the continuation event A.a means that the system will continue to function as it did before C was added to the system.

5.3.2.4. Immediate Action Execution

The code tier has the potential to invoke events and these events are also expected to follow the action-processing dictum. As shown in Figure 5-19 below, there are four potential orders for invoked events to be processed: 1) immediately, 2) after the state entry function completes, 3) after the continuation events and 4) after the response events. Because every line of code in imperative programming can change the system's state, as can handling events, these invoked events must be handled immediately.

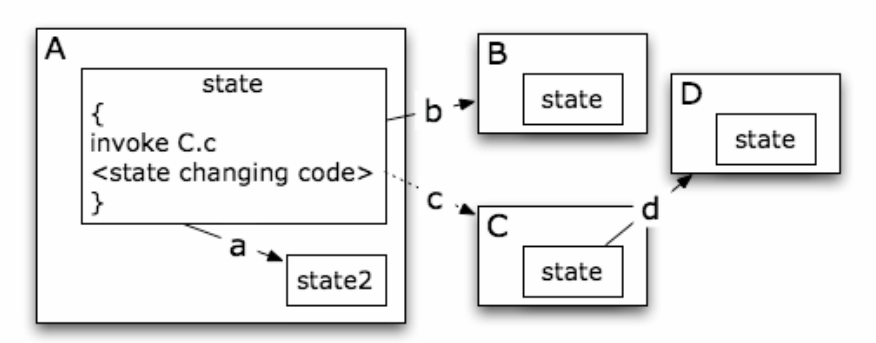

Figure 5-19. The order of events on entry to A.state is ((C.c, D.d), B.b, A.a) with C.c and D.d in parenthesizes as they run in a separate cascade of events.

The immediate handling of events creates a problem of handling not just the invoked event but also all of the events that follow and all the events before. In Chasm, this is addressed by using a separate event cascade, so that the existing cascade of events remains unchanged by an invoked event. In Figure 5-19, if C.state has response events or continuation events, those must all complete before returning to the execution of the state entry function of A.state. But, without a separate cascade to handle the events stemming from C.state, these events would be mixed with B.b

and A.a and their execution would be on a system that has changed since C.c was processed.

5.3.2.5. Final Ordering

A final example (see Figure 5-20) shows how the different event types are ordered. In this example, on entry to A.state, the events are handled as ((C.c, D.d), B.b, E.e, A.a, F.f).

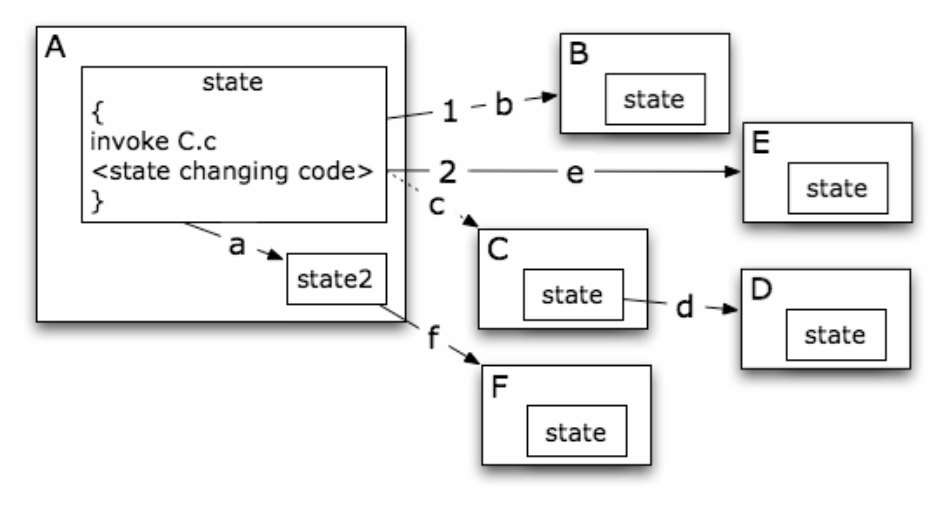

Figure 5-20. The order of events on entry to A.state is ((C.c, D.d), B.b, E.e, A.a, F.f) with C.c and D.d running in a separate cascade of events.

5.3.2.6. Continuous and Future Events

Continuous and future events are both linear time events. The difference between them is that continuous events happen at the next possible time the system is available for processing and the future events happen at a future point in time. When the maturation time occurs, the future event is processed as a continuous event. Continuous events are processed on an event cascade as a continuation event. These events can be originally created through invoked events, continuation events or response events[5] and given a time at which to mature.

5.3.3. Implementing the Order

The ordering rules of Chasm are implemented using three stacks, an ordered list and two algorithms. The response and continuation stacks hold events waiting to process. These two stacks in turn make one cascade entry in the cascade stack. The future list holds and temporally orders events waiting to be processed. The manipulation of the events and cascades in these stacks and lists are by an algorithm for event selection and an algorithm for event processing. This is explained further below.

5.3.3.1. Discrete Event Selection Algorithm

When the system seeks to process an event, the following algorithm determines which event to select for processing. Depending on a developer's need in their application, this algorithm can be set to run on request, in an event loop or

5 In the current Chasm implementation, response statements cannot create continuous and future events. This is a limitation of the syntax and will be rectified in the future.

whenever an action is invoked in Chasm. Its processing stops when all events have been processed. Chasm event processing has an assumption of termination--see Cycles below for conditions leading to non-termination.

The algorithm for determining the action to process next is as follows:

1. An action enters the system from the code tier or matures on the future list and is placed as an event in the continuation stack of a new cascade entry in the cascade stack.
2. The system retrieves its top-most cascade from the cascade stack. If there is no cascade, exit.
3. If an event is on the response stack in the retrieved cascade, process it. Return to 2.
4. If an event is on the continuation stack in the retrieved cascade, process it. Return to 2.
5. Remove the top cascade from the cascade stack. Return to 2.

5.3.3.2. Event Processing Algorithm

The algorithm for processing an event proceeds as such:

1. The event inserts its action into the action's concept, transitioning the concept to its next state along a transition. If no transition exists, the system generates an appropriate error message.
2. On entering the new state, run the state's entry function. If the state's entry function invokes an action, create a new cascade entry for this action and process that cascade immediately, returning execution to the state entry function. The entry function can return an action. If so, place this on the continuation stack of the current cascade.
3. For the newly entered state, add its causal statement's actions to the response stack of the current cascade.

5.3.3.3. Continuous Event Selection

The future list maintains the actions that occur in linear time, each of which has a maturation time. When the future list is checked, the events that have matured are handled as continuation events in the event processing algorithm above. The developer at runtime checks the future list. In this way, the checks for continuous events can be incorporated into the main loops of existing event based systems.

5.3.3.4. Potential for Parallel Execution

There is a great potential for parallel execution in Chasm as the connections between concepts are captured in the causality tier. Because of this, under some conditions, the processing of events could be treated as completely separate processes and run in parallel or even run in parallel and joined after both have finished. With a better algorithm for event processing, this could be automatically performed by the system without developer input. Unfortunately, the problem is complicated by the code tier that allows method calls between concepts.

Future work to automatically parallelize event processing could be through incorporation of the method calls into the event processing order or even extensions to the syntax of Chasm denoting non-determinism between response orders. Algorithms to load balance the processing of events, as in distributing of concepts among separate processing units, are also future work. Chasm was designed to

optimize developer time and not the execution, leaving this as promising work that is beyond this scope.

5.3.4. Cycles

The creation of non-halting cycles is a problem in the code tier and structure of Chasm. The four types of cycles that can occur are continuation, response , infinite cascade and maturation. Two of these cycles are identifiable by the system and one is blocked from occurring. Chasm assists developers with these cycles.

5.3.4.1. Continuation and Response Cycles

The first two cycles are similar in that they both are an infinite loop of events in a single event cascade. If the cycle occurs across more than one cascade of events, then it is potentially an infinite cascade cycle (see below).

Continuation cycles are created when state entry functions return actions that eventually lead to the first state, causing the cycle to repeat (see Figure 5-21). Below, state *a* returns a continuation event of action x causing a transition to state b: a-x→b. This continues with b-y→c and c-z→a. At this point, the cycle repeats.

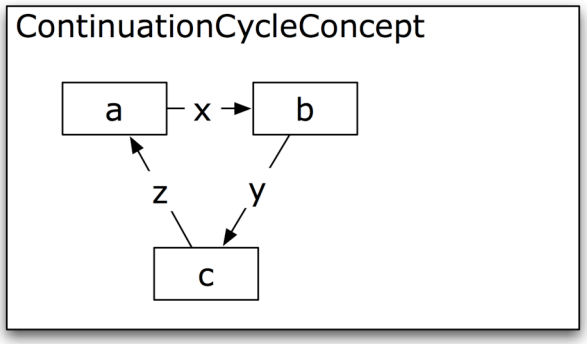

Figure 5-21. A continuation cycle occurs when the state entry functions return actions causing continuation events that loop. This is shown above as a never-ending loop abcabcabcabc, etc.

Response cycles are similar to continuation cycles but the events are both response events and continuation that create an infinite loop (see Figure 5-22). Below, when state A.state2 is entered, it starts a series of response and continuation events leading back to state2 being reentered. Ultimately, this ends up with a response cycle of events (A.state2, B.state1, C.state1, C.state2). In the diagram, notice how concept C, which returns C.x, loops through a continuation event.

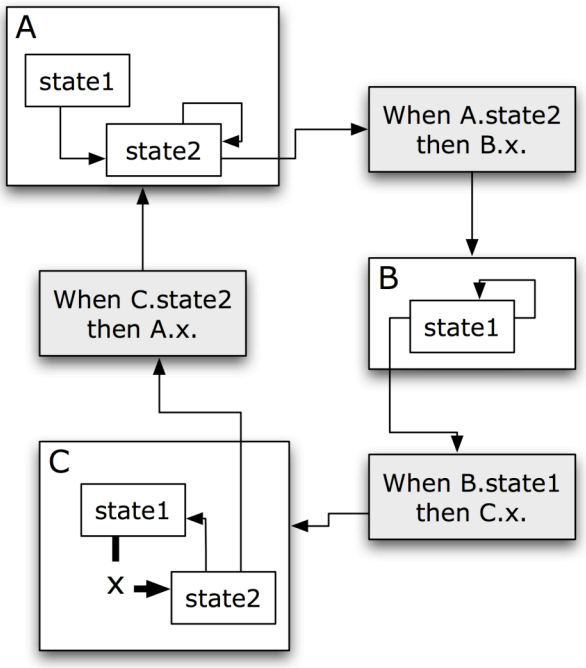

Figure 5-22. The response and continuation events started by the entry to state A.state2 create a response cycle. Here, state C.state1 returns action x.

When these cycles occur, they are caught by Chasm at runtime and reported with a printed error message. To catch these cycles, each state is entered into a cascade's state stack. If during the processing of a cascade, a state is re-entered, then the cycle is captured. A problem with this approach is that it incorrectly assumes the entire state of the concept is represented by the states in the concept's automata tier. The reality is that stateful information is also present in the code tier. This means that an error can be reported but in reality, stateful information is changing in the code tier that will break the cycle. For example, in Figure 5-22, the decision state C.state1 is shown to return C.x but could actually only be returning it on the 1st pass. If this is the case, the cycle repeats once and then quits without an infinite loop yet Chasm would warn that it is a cycle.

Without Chasm's help, it would be problematic to identify these cycles, especially when they occur across multiple concepts. Chasm's identification of the cycles greatly improves development. In Chasm, there are no means to override these checks to avoid the false positive. Thus far, this has not been a problem.

5.3.4.2. Infinite Cascade Cycle

Infinite cascade cycles occur when an entry function invokes an event causing a new cascade which, itself or during the processing of the cascade, recreates the same cascade. This can be caused by continuation or response events but would not be caught as a continuation or response cycle because the state is re-entered in a different cascade.

This cycle was at first common, caused by a state entry function changing a value which created an invocation event notifying itself that the value was changed, causing the cycle. The cycle was also difficult to track down until runtime awareness functionality was built into Chasm. The most common cause of the cycle was also addressed with the Resource concept (see Appendix F) making this cycle a rare

occurrence.

Chasm could capture this cycle automatically at runtime and print an associated error message. Because the cycle is easily identified currently with runtime awareness and tracing and its identification is susceptible to the same false positives as the previous cycles, this check was not implemented. If it were implemented, the existence of the cycle is identified by the current state in the top cascade being the current state in a lower cascade's state stack.

5.3.4.3. Maturation Cycles

Maturation cycles occur when a future event matures and then schedules an event to mature before it finishes processing itself. The effect being that processing is constantly trying to catch itself in a cascade and never returns to the processing of other events in the system. In many systems, timing is critical and there is plenty of research on achieving proper real-time behavior of systems [Alu94]. Because faster response rates have an impact on human performance [Adl05], Chasm operates on a the-sooner-the-better principle where important events can be set to run immediately in the next check of the future list. Chasm, however, does not assure any form of real-time behavior.

Chasm addresses the maturation cycle by maturing events as an atomic action. When the future list is checked for mature events to process, all events that are mature are processed before rechecking for future events. The effect is that processing returns to the system even if the future list has events that are mature. Events can still be scheduled to process before the current event is finished processing but it does not delay execution of the system. The burden of timing is placed on the developer to be aware of the time it takes to process their events and create realistic maturation times.

Avoiding much of this problem is the Animate concept, which assists developers in programming continuous behaviors as slices of time without expectation of uniformity of those time units. The Animate concept returns the amount of time passed since last handling of the continuous behavior as a fraction. It becomes easier to describe an animation as a function of time rather than a series of *n* timesteps. For example, animating the movement of an object 1 meter could be implemented two ways: 1) as 10 timesteps which move the object .1 meters with an assumption of 100ms between each timestep or 2) as a function that moves the object 1 meter per second. The Animate concept operates the second way. Since a machine can be bogged down, the 100ms assumption could break, leading to a slower rate of travel. Using the Animate concept, if the machine becomes bogged down, the animation becomes choppy, moving in larger steps to make up for lost time, but completes as expected.

There is no warning when an event is scheduled to mature before the currently executing process finishes. This is because the overhead to check timing by time-intensive system calls is not justified and the handling of the future list means that this hasn't been a critical problem.

5.3.5. Special Cases

Special cases in the development and event processing in Chasm can potentially lead to developer difficulties and an invalidation of a concept's behavior. These are handled where possible by Chasm.

5.3.5.1. ChasmXML Multi-Pass Loading

> Life can only be understood backwards, but it must be lived forwards.
> --Soren Kierkegaard

Loading a ChasmXML file requires three separate passes when loading. Without the multi-pass loading, each ChasmXML file would have required the developer to give extra instructions and spend time correctly ordering lines in the file in order to load each file.[6] This would bloat the ChasmXML file's notation as well as add complexity to a concept's development.

The three separate parses are parent, structure and response. The parent parse searches the ChasmXML file for the parent tag, parsing the parent, and its parents, first if it does not exist. This loads in inherited structures from the parent and errors out in the event that the concept tries to reuse names of automata tier structures. The structure parse loads all tags except causal statements. This ensures that all component concepts are fully loaded before the response parse. The response parse then loads the response tags, assured that all the states and actions of the component concepts are fully loaded and able to have causal statements attached.

5.3.5.2. Partial Destruction Management

Calling the destructor of an object in C++ is one means of changing the state of a concept. When improperly called, this can prematurely terminate an application or cause it to behave erratically. For example, deleting a concept that holds the state for a causal statement results in a dangling pointer in C++. This type of bug is hard to reproduce as well as fix. Making this more important in Chasm versus normal C++ is the fact that the developer is unaware at development time of all the uses of the concept. A new concept to a system can require the concept to exist longer than in previous systems and for different reasons, i.e. a new causal statement.

Chasm removed this concern from the developer by partially managing the deletion of concepts. First, if the destructor is directly called, it results in a runtime error with specific debugging information. Proper deletion occurs by calling a delete method of the Concept base class, CO::deleteCO(CO*). The CO::deleteCO(CO*) method wraps the concept's destructor to avoid directly calling it. This method first performs simple checks for NULL concepts and valid concept pointers and then passes the concept the *end* action to take it to the *ended* state. In the ended state, the concept checks if it is required by any other concept. If not, the *del* action is returned which calls the concept's destructor. Otherwise, it waits for one of the concepts which use it to request destruction before checking again. By using Chasm's event handling for deletions, events of the concept needing to process will be processed before the concept deletes. By partially managing deletions, additional concepts to a system will be assured the concepts they rely on will remain available.

Note that the current implementation does not catch circular dependency loops, the result being a potential memory leak. There are several simple algorithms, such as mark and sweep [McC60], to handle circular garbage collection that could be implemented in Chasm.

5.3.5.3. Gathers

A problem exists where a single state change incorrectly creates the same event multiple times. In the best-case scenario, there is wasted computation. In the worst-case scenario, there is unexpected behavior or system termination. This problem is surprisingly common and not easily identified by developers because it results from

---

6 A similar problem occurs in the ordering of include statements in C and C++.

the collective behavior of multiple concepts. This problem is handled by a special concept called a Gather that is automatically inserted into a system at runtime by Chasm to assure that the action is only passed once. Chasm checks for this problem each time a causal relationship is created.

Consider the example in Figure 5-23 (below) where the WIM is used to move pipes in a 3DUI. The user is trying to move pipes close together without them touching. When the pipes do touch, the held pipe is highlighted to a color to alert the user that pipes are touching, so the user can move them apart. Now, consider how the WIM operates. When a user grabs a proxy pipe object in the WIM to manipulate a pipe in the real world, called HeldPipe in the figure, the PositionRelationship concept that moves the real world pipe is dependent upon the proxy pipe's position regarding its frame of reference, the WIM. So, when the user's hand holding the proxy pipe moves or when the user's hand holding the WIM moves, the real world pipe updates. In Chasm, the tracker is the source of movement for both the left and right hand and, due to tracker jitter, nearly every check of the tracker will result in both hands moving at the same time. This causes the PositionRelationship to calculate twice -- resulting in the a twofold update of the pipe's position. In this case, it only results in extra computation. However, recall that when pipes touch, the pipe is highlighted a color for feedback. If the pipes touch, the first feedback highlights the held pipe and so does the second. However, since feedback in this case is boolean, i.e. it is either giving feedback or not, the held pipe will report an error and the application will halt as there is no transition from its highlight state to its highlight state. To the developer debugging the system, the system appears to crash when the objects touch. If events are traced, the developer will find (after a long string of events) that, for some reason, the HeldPipe is highlighted twice.

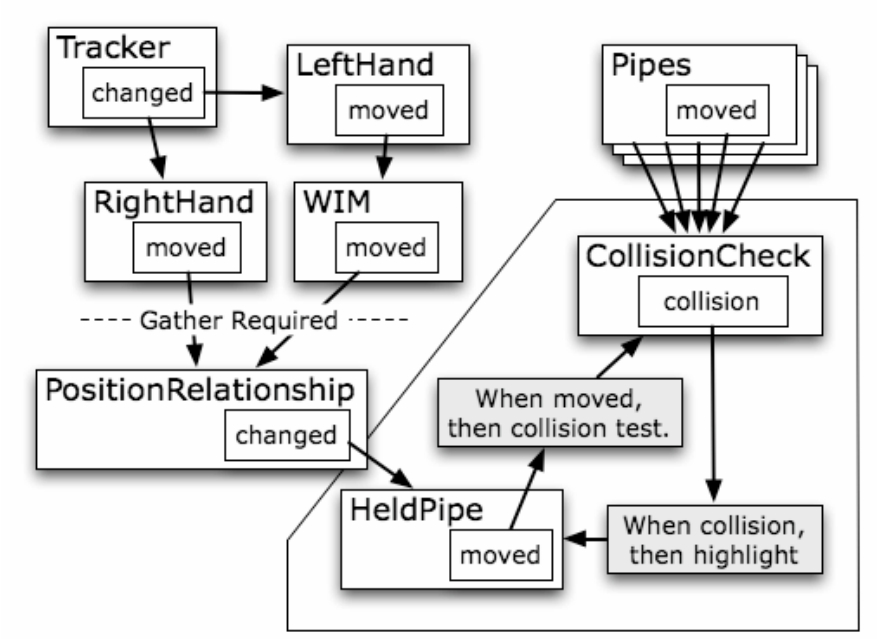

Figure 5-23. This pipe manipulation system via a WIM may crash because PositionRelationship will cause two highlight actions to enter the HeldPipe. Chasm identifies these issues and inserts a Gather in the position indicated. The Gather will collect both the RightHand and WIM actions and send them once.

A Gather works by routing the offending causal relationships into itself, waiting for the causal statements to finish and then sending the action once, as in Figure 5-24.

This occurs by first surrounding all the offending responses with two causal statements: “When 'offending state', then Gather.starting” and “When 'offending state', then Gather.going”. These two causal statements are #1 and #4 in Figure 5-24. It then routes the duplicate causal statements into the Gather with Gather.gathering and sets the Gather to send the offending action.

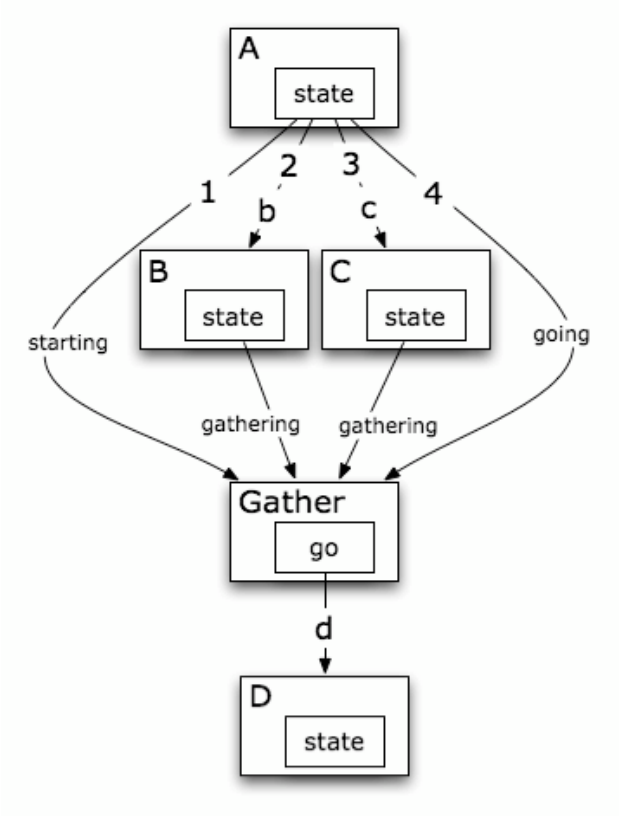

Figure 5-24. By using a Gather, the problem of the action D.d being sent by both B.state and C.state is resolved. This is achieved by wrapping the actions created by causal statements and a Gather sending D.d once when both B and C are finished.

Checking for this problem involves starting with an action and tracing its causal statements through to its states. If any action occurs more than once from the same state, a Gather is inserted. Because continuation events do not run before a response event, we only focus on causal statements and their response events. For actions caused while inside a state entry function, it is problematic. First, it is problematic to detect but more importantly, it is problematic to determine if it is an error. Most likely, the multiple events are the correct behavior and therefore not an error.

These checks have to occur when causal statements are added, which can lead to serious performance problems. In systems with deep causal statement graphs or high branching, the number of causal statements to search through is large. To optimize this, a secondary data structure stores every state that leads to an action. These states are referred to as ancestor states. The algorithm and data structure, not discussed here, greatly reduce the number of checks when looking for this problem, trading off storage for runtime performance.

5.3.5.4. Encapsulation Assumption

Much of the invariance of a concept's function is from the assumption that it fully

encapsulates its stateful data. Though this is clear in the automata tier, the assumption fails in the code tier because of its implementation in an imperative language. The effect is that states outside the concept, and not accounted for in the representation, can affect the functioning of the concept by placing it in a state it was not intended for or tested in.

Chasm places the burden on developers to properly encapsulate their concepts. If stateful data changes impact the states in the automata tier, the appropriate action should be inserted into the concept. This includes global state. This burden is typical for developers but is assisted in Chasm by its approaches to decomposition, which create simpler concepts and make the encapsulation process simpler.

As an example, consider a scoreboard concept that is updated every time a button is pushed (e.x. Figure 5-25). When the button is down, it increments a counter and when the counter is counted, the scoreboard is updated. The scoreboard relies on Counter to maintain the count -- it then asks the counter for the count to display it. Shown in Figure 5-25, there is also a method called Counter::incrementCounter which adds to the count. Incorrectly, it does nothing to invoke the action *increment*. So, calling the Counter::incrementCounter method would increment the score but not update the scoreboard. Thus, the next time the button enters the down state, the scoreboard will leap ahead by two. The correct implementation of Counter::incrementCounter would be to invoke the increment action instead of "count++".

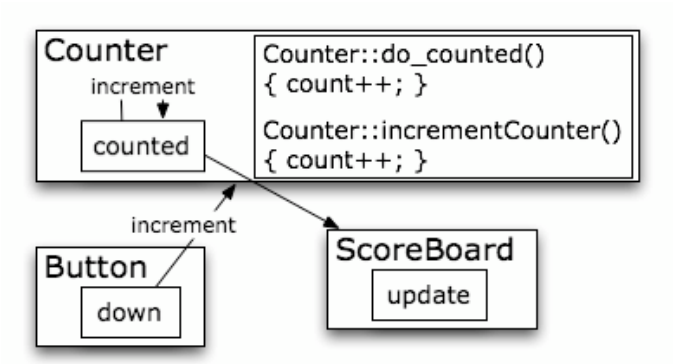

Figure 5-25. When the button is pressed, the ScoreBoard updates with an incremented score. However, calling the Counter::incrementCounter method only increments the counter without updating the ScoreBoard. The correct implementation would be to invoke the increment action in Counter::incrementCounter so the ScoreBoard would update.

5.3.5.5. Controlling Response Order

The ordering of causal statements affects the order events are processed and developers have a limited ability to specify that order in the ChasmXML. Currently, the order of the causal statements in the ChasmXML file determine the order of processing. However, the ChasmGUI has no ability to reorder causal statements.

While this problem is solvable it occurs infrequently enough that it is not addressed in Chasm. Chasm currently includes methods for programmatically reordering responses "before all of the following" and "after all of the following" responses. In the case studies, this problem never occurred, with the functionality only being used to implement the special case Gather concept, discussed above. This is not problematic because concepts that typically respond to the same state are component concepts in the same concept and can be ordered in that concept. This

shallow relationship makes any ordering problem easily apparent but deeper reuse, especially by developers that did not make the original concept, could increase the occurrence of this problem.

5.4. Match to Principles and Problems

> There is not now, and never will be, a language in which
> it is the least bit difficult to write bad programs.
> --Anonymous

The creation and implementation of Chasm was designed to address the problems found in the development of 3D interfaces as well as match the principles of Concept-Oriented Design. These problems were discussed in Chapter 3 and the principles of Concept-Oriented Design in Chapter 4. Both are shown in Table 5-1.

Table 5-1. The problems and principles behind Chasm's design are shown below.

Problems of 3DUI Development	Principles of Concept-Oriented Design
Limited Understanding	Envisioned Behavior
Distance of Mapping	Artifact and Language Focus
Complexity	Tiered Representation
Reimplementation Over Reuse	Longevity
Hard Domain Problems	Community

5.4.1. Problems

Chasm addresses each of these five problems, focusing largely on the complexity and reimplementation over reuse problem. Limited understanding is addressed by the envisioned behavior tier that lets the developers express their ideas in conversational domain language. The self-awareness, visualization, event tracing and interactive shell also help the developers to have insight into a Chasm system. The distance of mapping is addressed by Chasm's tiers where differing information types correspond to developer's thinking. This includes code in the code tier. The ChasmGUI also addresses this issue.

The complexity problem was addressed in several ways. Starting with the action-processing dictum, it allowed different concepts to process with little input from the developer. This was enabled by the encapsulation assumption, without which the action-processing dictum would fail. Another important complexity handling tool is the tiers, where the upper tiers define the coupling between concepts and the lower tiers deal with the internal functionality. Additionally, the representation is able to deal with temporary information while the understanding of the developer increases. As a result, Chasm's tiers divide complexity into concepts, into different information types and allow for development to proceed with temporary information. Other factors that reducing complexity include the cycle handling built into Chasm and the multi-pass loading that removed work from the developers.

The reimplementation over reuse problem was a major focus of Chasm, designed to allow concepts to be reused in new systems. Much of this came from its tiered representation, stored in ChasmXML, but other factors helped out as well. This includes the builders, which allow toolkits to be incorporated more easily, partial destruction management and Gathers, which ensure that a concept can be reused

without new functionality breaking existing usage. Because of this, concepts can be created and stored as a vocabulary for future reuse. Thus far, there has not been enough development to create a rich vocabulary, though the groundwork is laid and future development will create and improve concepts for this vocabulary.

Hard domain problems are important and consist of several different problems stemming from several different sources. To address this, Chasm creates a system where other researchers can develop, test and share their approaches with others. In this way, Chasm was careful in the design of its representation to ensure that existing tools continued to function and existing toolkits were easy to integrate into Chasm. In this way, when other researchers solve a hard domain problem, the solution can be easily incorporated into Chasm's vocabulary. The work of several concepts in the vocabulary is seen as first attempts at some of the problems of 3DUI hard domain problems. The Animate concept deals with continuous actions and the Resource concept deals with parallel behaviors. Future concepts in the vocabulary will continue to build on this base.

5.4.2. Principles

Concept-Oriented Design (COD) principles were followed in Chasm (see chapter 4 for a review). Most of the effort in Chasm was placed on issues important to the principles of tiers, longevity and community. These were critical for the initial creation of a COD system. As per the tiered representation principle, Chasm has a stratified design. This allows the envisioned behavior tier to address the envisioned behavior principle while the causality tier addresses the artifact and language focus principle. The tiered representation principle is also supported by its representation in ChasmXML and the different decomposition approaches in Chasm.

Chasm addresses the longevity principle first through the vocabulary, which serves as a library of organized and reusable concepts. Those concepts are valuable because of the other features of Chasm which enable them to be reused more readily – including the action-processing dictum, cycle handling, partial destruction management and gathers.

Chasm is also designed to reduce the barriers to entry for the community principle. The ChasmGUI is the beginning of a mainstream tool to reduce the time it takes to learn Chasm. Chasm is self-aware, it creates visualizations of running systems, traces events and has an interactive shell. This is all documented on the wiki and in several examples of Chasm systems. Chasm is also designed to allow existing tools and developer experience to be incorporated into the code tier. As the existing community uses a wide range of toolkits, builders were designed to simplify the process of supporting new toolkits. When Chasm was being used, some improvements were created to reduce developer problems, including supplemental files to simplify coding, and multi-pass loading of ChasmXML files.

In sum, Chasm implements the principles of COD and, because those principles arose from the problems identified early on, it also addresses the problems of 3D interface design and development. However, the remaining question is, “How well does Chasm actually work for real 3D interface development?” This is the subject of the next chapter, “The Evaluation of Chasm.”

Chapter 6 - Evaluation and Validation

RQ4 - How effective is Chasm in the development of 3D interfaces?

Hypothesis – It is hypothesized that success can be shown through carefully layered evaluations as well as by measures and metrics of case study applications.

Four evaluations of COD in CHASM were used to determine the ways in which 3DUI development may be improved by this system. These evaluations consisted of: criteria evaluation, evaluation by domain experts, longitudinal evaluation by developers and by measures and metrics of case studies. Though no evaluation type on its own is sufficient, by using a mix of evaluations, each overcomes the shortcomings of the other. The selection of the evaluations stems from the following considerations:

- The goals and results of the evaluations had to be considered. Is Chasm considered a walk-up-and-use interface or does it require expertise? For this evaluation, it was decided that Chasm is not a walk-up-and-use system.
- Separating the evaluation of COD in Chasm from the evaluation of the current tools is important. Tool creation is not the goal of this dissertation yet tools have to be created for evaluation and testing. Poor tool support would lead to a poor evaluation of Chasm, even though it would be the failing of tools and not COD in Chasm's potential.
- The role of experience impacts the evaluation of COD in Chasm. A head-to-head comparison of Chasm to development approaches, in which developers have years of experience, would not be a fair comparison. It takes time to gain experience before a comparison can be performed.
- The experience of the participants had to be considered. Developing 3DUIs requires experience with computer graphics, human-computer interaction and 3D hardware. Developers must also know the SVE toolkit used to handle Chasm's graphics and hardware. Additionally, Chasm is an object-oriented paradigm that uses state machines and an event-based architecture. Finding participants with experience in each of these areas restricts the potential participant pool.
- Testing COD in Chasm requires real problems, not simple 3D interfaces. The claimed benefits of Chasm come from the reduction of complexity in large projects and the reuse of code. Large amounts of development are required for this and need tools able to support real production environments. Therefore, simplistic 3DUIs would not show the extent of the benefits of Chasm.
- It takes developers, especially non-experts, time to develop 3DUIs. Finding volunteers with the discussed qualifications for lengthy periods of time is difficult. This is only made worse by the requirement that non-trivial 3D interfaces are required for a true test of this approach.

Shaw [Sha01] discussed five different validation techniques to software engineering research; persuasion, implementation, evaluation, experience and analysis. Shaw defines these validation techniques with statements such as:

- "I have thought hard about this, and I believe that...." (persuasion)
- "Here is a prototype of a system that...." (implementation)
- "Given these criteria, here's how an object rates...." (evaluation)

- "I evaluated these results based on my experience and observations about the use of the result in actual practice and report my conclusions in the form of...." (experience)
- "Given the facts, these consequences...." (analysis)

Recall that persuasion (Chapter 3) and implementation (Chapter 4) validations have been discussed. Chapter 3 discussed the design of the principles of COD while Chapter 4 discussed the issues involved in Chasm's development and how issues were addressed.

The four evaluation types, which address the above considerations and Shaw's remaining validation techniques, are discussed below.

- First, an evaluation by criteria (Section 6.1) of good software systems will show the comparative strength and weakness of Chasm along established metrics. This represents Shaw's validation technique of software engineering research. The criteria chosen are those of the Cognitive-Dimensions Questionnaire [Bla00] as well as a claims analysis by software quality attributes [Bar95][Boe76a][Boe79]. An expert with comprehensive knowledge of the system can perform this evaluation quickly. The downside of the approach is a possible bias in the choice of the criteria.
- Second, domain expert evaluations (Section 6.2) using the Cognitive Dimensions Questionnaire create an evaluation from a deep domain understanding and address the bias of the previous evaluation. This represents Shaw's experience validation technique of software engineering research. Exposing domain experts to Chasm can elicit their impressions in short periods of time regarding the approach of COD in Chasm. Importantly, their domain knowledge in 3DUI development will provide a differing perspective, leading to a broader evaluation. Unfortunately, the evaluation only provides a limited and potentially biased exposure to Chasm which also lacks the benefits of an experiential evaluation, a use in practice.
- Third, longitudinal evaluation (Section 6.3) by volunteer developers, with varying degrees of expertise in 3D interfaces, creates an evaluation from a deep understanding of Chasm. This too represents Shaw's experience validation technique of software engineering research. The longitudinal understanding is achieved by long-term use of Chasm in the creation of real 3D interfaces by four volunteer developers (VDB, VDC, VDD, VDE) and myself (VDA). Gathered information from the evaluation is from a Contextual Inquiry [Bey98] as well as a summative evaluation using the Cognitive Dimensions Questionnaire. Problematically, the volunteer developers lack the 3DUI development domain experience of the domain experts that will limit their evaluation.
- Finally, measures and metrics of case studies (Section 6.4), taken from the longitudinal evaluation as well as my own systems, create quantitative data for analysis and represent Shaw's analysis validation technique of software engineering research. This data can be used to support the themes and claims of the previous chapters.

The final section (Section 6.5) maps the evaluation results back to the five problems identified in Chapter 3. In this way, Chasm can be judged in relation to its performance on the problems of 3DUI development.

NOTE: A heuristic evaluation of Chasm's tools by usability specialists was covered in Chapter 5. This evaluation tested Chasm's suitability as a long-term development

environment and led to improvements in the ChasmGUI.

6.1. Evaluation By Criteria

The evaluation of systems by a set of criteria is useful but also problematic and can be performed by the use of a set of evaluator-selected criteria or a set of standard criteria. Using evaluator-selected criteria, the evaluator can select criteria that bring out important ideas in a system. They can also bias the evaluation by selecting only positive or negative criteria. In the literature, the criteria for analysis are often created arbitrarily [Kip97]. This is more akin to a claims analysis [Car00] than a true evaluation. Using a set of standard criteria for evaluation can show the value of a system in a generic way and avoid the selection bias. A standard set of criteria also has the potential to highlight positives or negatives that would have gone unnoticed. Cognitive Dimensions is useful in this respect and has been used to evaluate visual programming environments in the past [Gre96].

In the evaluation by criteria of Chasm, both approaches are used. First, Chasm is evaluated by Cognitive Dimensions and then by software quality attributes selected from the literature. By placing the evaluation by criteria section before the domain expert evaluation, longitudinal evaluation and measures and metrics evaluations, a general understanding of Chasm can be obtained. Then, the other evaluations can be understood in the context of these criteria.

6.1.1. Cognitive Dimensions Analysis

An analysis by Cognitive Dimensions (see Chapter 2 Cognitive Dimensions) was conducted to provide additional insight into COD in Chasm and this evaluation is reported below for each cognitive dimension. The cognitive dimension is given along with its brief description [Bla00]. Several ideas become apparent in this evaluation, and where applicable, I discuss the source of the problem. In most cases, the problem is due to the tools and ChasmGUI implementation rather than a limitation within Concept-Oriented Design in Chasm. Note that improvements based on this analysis are discussed in Chapter 7.

Visibility: ability to view components easily

The notation of Chasm was designed to be highly visible because it based upon language and diagrams of developers and domain experts. The multiple tiers approach allows developers to choose the tier they need for their type of work. Additionally, the causality and automata tier make the flow of the system much more visible compared to code. This lack of flow visibility in source code was problematic in current implementations so it was important to address. Unfortunately, some of the structures of the automata and causality tier are still in the code tier, becoming less visible when placed there.

The code tier has other visibility issues that were improved with supplemental files. For example, the preprocessor statements made it difficult for developers to understand what the final code would look like after it passes through the preprocessor. The supplemental files were a step forward in addressing this problem but were confusing to developers at first. Only Volunteer Developer B (VDB) used supplemental files consistently and they had positive comments about them. Hampering the uptake of supplemental files was their late introduction to the volunteer developers. Lastly, as Chasm lacked tool support for code in the code tier, instead relying on development environments developers were more familiar with.

It became apparent in this investigation that the ChasmGUI has avenues for

improvement in matching developer thinking. First, components of a concept were not shown within the concept. Second, the same concept could be a component of two concepts (aggregated) but was shown as two different concepts. Third, levels of scale in the concepts need to be visualized more like zooming graphical interfaces [Bed94], as the notion of delving deeper into a concept was not represented in the ChasmGUI. While the ChasmGUI has the ability to change scale, it was not successfully used in this investigation.

Viscosity: resistance to change

Chasm is easily modifiable with tier modifications having little impact on other tiers. One issue occurs when a state, acting in another concept as the context state of a causal statement, is renamed. Fortunately, this is caught at compile time.

Diffuseness: Diffuseness is verbosity of language

Chasm was successful in reducing the verbosity of the language in the causal tier but was limited in other regards. The automata tier was verbose regarding the definition of transitions. As a concept grows, new states and actions are added and the numbers of potential transitions in a concept rapidly increase. For most actions, the transitions were to the same state and there was no support for this in the GUI notation like there was in code. Concepts thus had very confusing automata tiers that were actually quite simple. Additionally, the code tier was verbose but the use of supplemental files addressed this. The ways in which the constructor parameters are handled could also benefit from further attention.

Hard Mental Operations: high demand on cognitive resources

Chasm attempted to reduce the number of hard mental operations during development through the use of tiers to support development as developer understanding improves. To this end, it made great strides. However, three issues remained.

First, decomposition of behavior is problematic in any system and Chasm's creation of new ways to decompose behavior can be overwhelming. In OO systems, "is a" relationships can be created by specialization of the object and "has a" relationships can be formed by composition and aggregation. In Chasm, because of the power of causal statements, it becomes difficult to know whether to specialize a new concept through inheritance or extend the concept with causal statements. Research on the utility and improvement of inheritance in OO systems is an active research area [Jon94][Bri01][Dal96].

Second, in the code tier, it can be hard to split functionality between the concept's state entry function and that of the builder's state entry function. The rationale is that generic code goes in the concept and the builder handles functionality that is toolkit or system specific. In practice, the volunteer developers had problems with this, likely due to their inexperience with the tools and the late introduction of builders to Chasm. This may also be attributable to forcing developers to create well-designed code at the same time they are prototyping an implementation. Code refactoring [Fow99] suggests that such structuring of code should occurre after its initial implementation.

Third, there was confusion in the way the transitions were presented in the ChasmGUI. Developers think in terms of, "When I am in state X, I go to state Y when action Z occurs." The ChasmGUI forced developers to think, "When I am in state X, action Z makes me go to state Y". Though a simple reordering, the developer is thinking about a flow in the automata tier and the ChasmGUI forces them to think

about actions.

Error-Proneness: the notation invites mistakes

There were few small, repeated problems. This was encouraging because much effort was put into avoiding these cases in Chasm's internals. Cycles were checked at runtime and error-checking code was generated at preprocessing time. For example, setting a component's CIF value as *passed* would create an error if the developer forgot to actually pass the component. Gathers also had automatic checks and one developer, VDB, had problems until gathers were inserted to the system (a software update during the evaluation of Chasm added gather support). VDB stated that their code worked after the update though they never realized why. One problem that occurred was developers thinking a concept started, not in the inited state, but in another.

Importantly, what was considered to be a potential problem of tracing the flow of a running system was not. There was a wealth of functionality to debug running systems such as the command-line interface and tracing (see Chapter 5 Runtime Behavior and Awareness) but none of the developers in this study used this, even after they were shown how it worked and tutorials were created. The flow worked for the developers without having to performing tracing flows. This was encouraging removing such difficulties from design and development was one of Chasm's goals.

Closeness of Mapping: closeness of representation to domain

The notation is a close mapping to developer's thought. However, the complete flow of the system is missing. Ideas that start out as scenarios and flows become broken into states and transitions inside the automata tier, with the original scenarios lost. The approach in this study improves on extant implementations that go from envisioned behavior to code though better support is possible. Additionally, visualization of continuous behaviors in Chasm are not displayed differently than discrete behaviors.

Role-Expressiveness: the purpose of a component[7] is readily inferred

Given that Chasm introduces new approaches to development, it presents developers with new representations they must learn. Overall, the top three tiers are understandable but the code tier has issues, which subsided with experience. Fundamentally, the issues are related to Chasm's current representation, not its core.

Starting at the envisioned behavior tier, developers had issues writing conversational domain language. Though developers could sufficiently explain what the concept did to an observer, their first instinct was to write code in the envisioned behavior tier. In this investigation, developers received some assistance with this in the beginning and they quickly understood the benefit of conversational domain language. This was especially true when they returned to their earlier code and could read the envisioned behavior.

The causality and automata tiers were also easily understood with practice. Though developers had exposure to state machines, they did not have experience working with them. Until developers gained experience, there were issues with the way Chasm causal statements inserted actions into other state machines. The ChasmGUI helped visualize this and also helped with concepts that required continuous

7 "Component" here refers to the general notion of a component of a notation, not a Chasm component concept.

behavior. Developers did not understand why a concept would continue sending its own action to itself to continue its behavior. However, after implementing continuous behaviors, it was quite natural.

The code tier was problematic for several reasons. First, Chasm makes use of the power of C++ such as access levels (public, protected and private) and inheritance. Though developers claim familiarity with OO development, confusion with this continued [Jon94][Bri01][Dal96]. Second, Chasm introduced a notion of CIF, which was confusing and was not supported sufficiently in the ChasmGUI, as mentioned in the heuristic evaluation (see Chapter4). Third, knowing where the developer was supposed to place code in the files and understanding the preprocessors statements was problematic. In part, this was due to the similar look and verbosity of preprocessor statements. For example, defining arguments for constructors requires six different preprocessor statements to handle all the possible cases of parameter type, name, default values and if passed to the concept's parent. To assist the developer, the automatically generated files were by default loaded with descriptive documentation (see Appendix D). Supplemental files helped improve the code tier problems.

Hidden Dependencies: important links between entities are not visible

Chasm is structured to catch hidden dependencies but requires more work in the visualization of inter-tier structures. Chapter 5 discusses Chasm's handling of behind-the-scenes issues, such as the action processing, partial destruction management, cycles and gathers. In this regard, Chasm was very successful. Chasm does have issues regarding the visualization of causal statements created in the code tier and passed components. Neither of these statements are visualized in the ChasmGUI.

Progressive Evaluation: work-to-date can be checked at any time

Chasm’s use of causal statements creates ways to progressively evaluate a Chasm system but an even larger potential exists. To test the flow of partially complete systems, state entry functions can be left as empty or return a single action, without any computation, to simulate the happy path's flow of behavior. VDB did something similar using key press actions to create an interactive system to simulate a flow of behavior in their incomplete 3DUI technique. After the flow was tested, VDB changed the stimulus of the action to the intended stimulus, like a button press or other user action. This progressive evaluation could be improved by creating a tool that inserts actions to running Chasm systems. For an example, review Volunteer Developer B’s case study (Chapter 5).

Provisionality: degree of commitment to actions or marks

As discussed previously, Chasm is provisional, allowing for changes to be easily and readily made. However, the ChasmGUI does not yet replace paper designing. As found by developers, it was best to continue to design on paper, especially for early designs. Once the paper prototypes were built in the ChasmGUI, developers continued to readily manipulate and make changes. Future work addressing the display of passed components in the ChasmGUI will improve this further.

Premature Commitment: constraints on the order of doing things

There are two premature commitments in Chasm, the first major and the second minor. First, the ChasmGUI forces the developer to consider a component's CIF at the time of the component's addition to the concept. At the time, the developer is generally only experimenting with the component. Having to deal with CIF creates an

extra decision, which is not needed. However, this is purely a limitation of the ChasmGUI, not Chasm itself. As a minor point, there is the previously discussed transition problem where developers think (state, state, action) while the ChasmGUI is (state, action, state). Again, this is a ChasmGUI issue.

Consistency: similar semantics are expressed in similar syntactic forms

As mentioned previously, causal statements can be created in the code tier as well as the ChasmXML. The is because, in some concepts, the causal statements change depending on the flow in the automata tier. This needs to be incorporated into the ChasmXML representation.

Secondary Notation: extra information in means other than formal syntax

There are multiple ways of capturing any secondary notation. This includes the envisioned behavior tier, notes on components and descriptions of states and actions. A useful addition to this would be to incorporate images, movies, media and hyperlinks, and this is an area for future work. Additionally, the ChasmGUI could better implement the causal statement. Currently, the action for the causal statement has to be given but ChasmXML does not require this, accepting an abstract text description.

Abstractions: types and availability of abstraction mechanisms

Components, tiers, inheritance and extension with causal statements allow developers to make abstractions quite easily. As mentioned in above (see Hard Mental Operations), this can lead to too ability to create abstractions, causing problems.

Novel Use: unusual usage of notation, unintended uses

The ChasmGUI and preprocessing limited much of the novel use of the notation. Additionally, the longitudinal developers suggested that they did not have sufficient expertise to attempt unusual uses with Chasm. Use of Chasm did seem to change their thinking of regular development as VDE designed a system for another class using Chasm's notation.

Improvements: obvious ways to improve the notation

This section will be covered in Chapter 7, in the future work section.

6.1.2. Software Quality Attributes of Chasm

Software quality attributes (sometimes referred to as Non-Functional Requirements or characteristics) are another way to evaluate Chasm. Below, each of these attributes are examined in relation to Chasm [Bar95][Boe76a][Boe79].

- *Accessibility* to the internals of a concept is improved in Chasm as developers have access to the internal flow of behavior in the concept. The concept also has the traditional method calls that developers can add.
- *Accountability* is improved by Chasm, as there now exist several higher-level tiers of understanding in which to measure a system. The measures and metrics (see Measures and Metrics in Case Studies below) are ways to quantify implementations. Additionally, Chasm has tracing information and could easily extend the tracing to measure the coverage of transitions in the automata tier or state entry functions in the code tier during testing.
- *Change impact* is localized to the tier of the change or below, greatly reducing the scope of testing and debugging.

- *Changeability* is improved during development. As development proceeds through tiers and as understanding grows during development, there are intermediate tiers in the concept that can be manipulated to quickly make high-level changes due to early design decisions. Externally, concepts can be added and removed quickly as well as reconnected using the causality tier. Because of this flexibility, development can progress in a top-down or bottom-up manner.
- *Composability* of concepts is achieved through the controls placed on a concept's state, the data encapsulation of C++ objects, the easy-to-understand envisioned behavior and internal tiers that model the concept's behavior. With this, the causality tier connections, inheritance and method calls readily compose new functionality.
- *Decomposability* is increased as the language of Chasm parallels that of the description of the task. The interaction of Chasm can be described with when, while, then, if and else, and this description flows directly into Concepts that can be generated through this description. Additionally, Chasm abstracts lower details away from the developer, reducing the complexity of their task to only the tier of the current development.
- *Efficiency* is both good and bad in Chasm. The overhead concerning the processing of event cascades degrades performance of the system. Additionally, the initialization of a Chasm system, including the insertion of Gathers, takes processing time. However, there is the potential for less repetition in the code tier due to multiple calls to functionality to reestablish the state of the system. Additionally, future work regarding parallelism and distributed processing based upon Chasm's representation can lead to a massive performance improvement.
- *Evolvability* of systems is achieved by swapping the cause of a stimulus or response in the Causality Tier.
- *Extensibility* is achieved with external concepts able to respond directly to a concept's flow, injecting their own events and code into a cascade of existing events.
- *Human Engineering* is improved in Chasm given its efforts to match developer thinking. The ability for the representation to assist developers to learn about their design and development is a major goal of Chasm. Additionally, because of the principles of Community and Longevity, developers will have communal support and a reusable vocabulary of concepts.
- *Internal consistency* is improved with concepts developed to match domain thinking and terms; and over time, these become the units of domain thinking.
- *Legibility* is addressed in Chasm with multiple tiers to read and understand the concept. When concerned with the code tier, legibility is addressed with the automata tier defining the state of the system in which the entry function operates. Using supplement files also improves the code tier’s legibility. For higher tiers, the ChasmGUI presents a visual representation for the developer and at runtime visual representations can be created.
- *Maintainability* is improved by the creation of both simple concepts and concepts that compose their behavior from other concepts. The tiered representation also enables a faster understanding of the system. Additionally, because of community support of concepts, the costs of maintainability are shared between projects.
- *Modifiability* in Chasm results from the four types of modification (see Chapter 4)

that can readily occur in a concept. It can occur in several tiers and have a limited impact on the system because of information hiding.

- *Portability* to new platforms, environments, and toolkits is less problematic in Chasm, as only the builder in the Code Tier needs reimplementation.
- *Rapidity* in development is improved as developers can rapidly create a domain specific set of concepts, like terms in task relevant grammars [Kel77]. These concepts can quickly cover a large range of behavior.
- *Reliability* is improved by Chasm's ability to produce simple concepts that have exhaustively considered all state/action combinations. Additionally, the Action-Processing Dictum and the special cases that are handled for the developer improve reliability. External concepts that reuse a concept, by attaching to a state in its flow, can rely on the meaning of the point in the flow staying the same even if the actual flow is modified. When a true change occurs, it results in a new concept so the current concept continues to function for the developer.
- *Reusability* occurs through cohesive concepts that are composed to form higher levels of meaning. To reuse a concept involves attaching directly to the flow of another concept's automata tier, with no need to artificially create "hooks" [Pol06] that would have to be maintained. Created concepts are placed into a vocabulary of concepts for reuse in future systems.
- *Robustness* of a concept's behavior is achieved by maintaining the order of processing in each tier and across concepts. This is handled internally in Chasm.
- *Scalability* in the current form of Chasm is a non-issue as Chasm relies on other toolkits to perform the graphics and hardware handling. This computation generally eclipses the system load coming from interface design in Chasm. In the future, this might change as more complicated interfaces are created with Chasm. However, the potential for parallelization of Chasm (see Chapter 5: Potential for Parallel Execution) could create impressive scalability.
- *Self-containedness* in Chasm is achieved as each concept is designed to address one envisioned behavior. Internally, this is decomposed into the concept's implementation.
- *Self-descriptiveness* exists in a Chasm representation because of the use of domain terms and a tiered structure that matches developer thinking. To gain understanding, the code can be traced and run-time information displayed to the user.
- *Structuredness* exists in Chasm as the tiers organize the interdependent parts of the implementation.
- *Testability* is greatly improved, as unit testing becomes a very effective approach. This is because a state machine, which the automata tier is based upon, can exhaustively consider all actions in all states. There are two problems with this and Chasm addresses both. First, the number of state and action combinations to test quickly rise in any interestingly sized state machine. Chasm addresses this problem by breaking the state machine into smaller state machines in the automata tiers of the concepts in a system. The second problem is that Chasm has more state than just what exists in the automata tier, so the assumption of exhaustive tested fails. In practice however, Chasm emphasizes the full encapsulation of state in each concept. In this way, it becomes easier to spot additional stateful information relevant to unit testing. In the future, unit tests can be developed and maintained for each concept in the vocabulary.

- *Understandability* (similar to learnability) is improved as developers have multiple tiers to understand a concept, choosing the correct tier for their needed understanding of the concept. The structure in the tiers allows for visualization of the system during development and runtime. Additionally, Chasm's similarity to existing state machines lowers the knowledge threshold for developer. However, thinking as Chasm and in terms of state machines is not the norm for developers. This leads to problems in learning Chasm.
- *Upgradeability* is improved because changes to a concept are internalized due to the automata tier and the C++ encapsulation. Future work to version concepts and create a system of concept management will improve this further.

6.2. Domain Expert Evaluation

In the expert evaluation, the entire process of Chasm was evaluated using domain experts familiar with their own tools and development methods. They were given a tutorial on Chasm (see Appendix E) before they began and then given a task to perform with Chasm. The entire process took roughly two hours for each of the five experts. Though all experts were associated with the 3D Interaction Lab, they have familiarity with very different toolkits. They were to use the Chasm tutorial 2 task, which deals with the creation of a simple virtual environment (shown in Figure 6-1) described as:

> *This scenario demonstrates how to solve a simple 3D interaction problem inside Chasm.*

The description of the problem is this:

> *When the user selects the end object out of a line of objects with Raycasting, have that object move to the center and above the other objects.*

The tutorial required that the experts proceed through the development process in Chasm step-by-step, according to a script. They were instructed to use their domain experience to compare Chasm to the tools and methods in which they were familiar. The Cognitive Dimensions Questionnaire [Bla00], used as a semi-structured interview, was used in this investigation to gather comments from the five participants. The experts had the option to opt out of the recording of the interview so only interviews two through five were recorded. The comments were then gathered and grouped into themes through an affinity diagramming process.

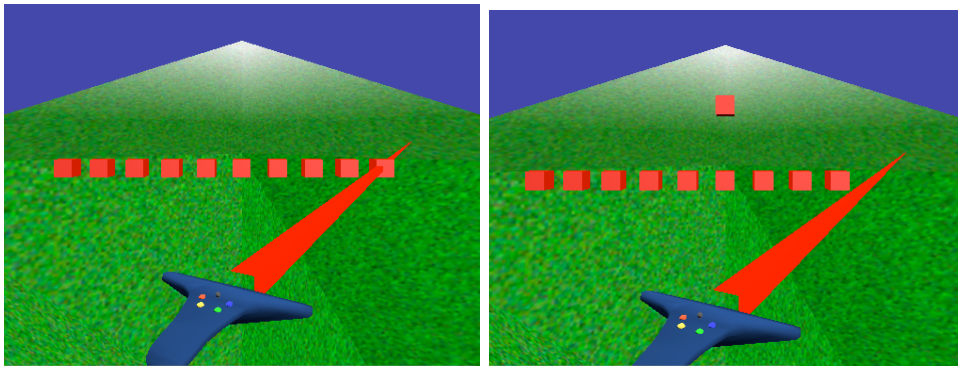

Figure 6-1. The expert evaluation had domain experts build a simple virtual environment using tutorial 2. When the far right cube was pointed at (shown on the left) and the button pushed, the cube animated to its new position above the other cubes (shown on the right).

Experts in this evaluation repeatedly stated that Chasm was not particularly difficult, but that there was a lot to learn. For example, in expert 3's words: "At first when I sat down with this, I wasn't sure how it all was going to fit together... until the end" and "It's fairly concise. It's good. I think there is a problem with all ... languages or tool that there is a certain type of dialect you need to learn." Expert 5 stated: "There were a lot things that required a lot of mental effort but that was because I was learning the system. If I were using the system for a while, my response would be a lot more targeted" and also, "I don't know if I can pick out anything that was stranger or harder than other [notations]. Again, it goes back to that I was groking the concept and GUI at the same time." Some specific examples were difficult to learn as expert 2 states: "How to manage the states and work and the custom places in code to put things." Expert 3 listed several issues that led to problems such as the use of camel case, capitalization, underscores and naming conventions in the code. This expert stated that it was "...good that [it was the] same in the gui and c file. Consistent." Other issues listed by expert 3 were the use of, "So many files (supplemental, code and header). So many files even with good feedback." These experts also regarded this to be a frequent problem with command line development.

The following four themes were developed after performing an affinity diagramming process of the collected comments; match to thinking and development, notation, abstracting and reusing content and visual representation.

6.2.1. Match to Thinking and Development

Chasm matched how experts thought about the design and development of 3D interfaces and there is room for improvement. This is covered below in three sub-themes. First, Chasm's causality matched how developers think, making development easier. Second, Chasm limited the scope of development, allowing for partial development and changes to occur easily. Lastly, although Chasm matched developer thinking, the next step to complete when developing with Chasm was not always apparent.

Chasm successfully matched the ways in which experts thought during development. As stated by expert 5, "I don't think your system restricts people from programming a certain way but I do think it supports the way that most people think." Most experts agreed that they thought in causal terms, like the causal tier: "You think, when I do this, the system does that. When I do this, then I do that. So, from a programming perspective, I think that you often have to think that way" (expert 5).

Chasm was successful at limiting the scope of a development task. The experts liked the ability to stop in the middle of development and check their work. For example, expert 1 said it was possible to build in both a top-down and bottom up manner. And expert 2 said, "It's very easy to just save the file, build the framework or skeleton you need, and test out the different parts. Being able to go in, take the next part and put it back together. Seemed very simple." This expert also stated that, "You can put everything out there and figure out how you want to string it together." Expert 5 said, "'That's how a lot of people program, right?' They start out with a very simple thing and they're like 'Let me see if I can get it to read the button, now I want the button to do this' when ultimately they want the button to do a lot more. But, they sort of build up the capability and relationships and your system, the Chasm system, seems capable of supporting that type of work." This expert also said, "...you could piece together these things and run them incrementally as you add more functionality". Much of this was enabled by the ease of changes in Chasm. This was supported by expert 4's comment that "changes are fairly easy" and expert 3's

comment that it is "...not particularly hard to change anything". Expert 2 stated that it was not difficult to make changes but that it was, "just knowing what to change and where."

The disadvantage to this developmental flexibility was the difficulty in understanding the next step in development, i.e., in a concept. For example, expert 3 stated that they wanted support in the form of "...a workflow guide, so [you] know where you are in your implementation for each concept." Developmental disadvantages also include knowing which tool to use next and which representation to modify. For example, expert 5 attributed their confusion to the multiple representations used, saying, "[You are] essentially working in 3 different environments at the same time. You'd have your GUI, your editor, and your command line shell." Expert 3 echoed this point with: "What is next? [I've] generated all these files... I don't know what that would be." One solution was offered by these experts; bringing the code tier into the ChasmGUI.

6.2.2. Notation

Though the notation was received well, experts offered suggestions for its improvement. For example, expert 2 stated, "Not overly verbose. Seems fine." Similarly, expert 3 stated, "it's fairly concise. It's good." Expert 4 commented, "it does make sense for the most part – fairly straight forward example." Regarding notation details, expert 2 stated that "this is the first actual tool I've seen which tries to put structure on interaction technique development. Most of the others are – 'Here is a toolkit, you have fun building whatever framework you need to build to build your interaction technique.'"

While some of the comments regarding the causal relationship notation were ambiguous, they were largely positive. For example, expert 4 stated that, "some of the [causal statements were] a little confusing," but that this may be "because [I'm] not accustomed to using it." Expert 5 stated that they "liked the notation of the [causal relationships]." They also stated, "A lot of the times in programming, logical operations are the ones you make mistakes on. But, the way you have the dialog boxes set up to created the [causal] relationships, [it] helps you visualize that piece of the code."

Experts expressed three main notation issues. As expert 3 stated, the "XML files have mixed content and presentation in [one] file," and this may lead to problems. During the evaluation, when exploring the automata tier of the Switch concept, expert 5 stated, "I remember one of the concepts I opened up... It had a bunch of stuff in it and you said, 'This is an easy concept but it is a complicated graph.' Well, that would be problematic." Expert 5 also stated that the similarity between concepts and component concepts may be problematic.

6.2.3. Abstracting and Reusing Content

Experts also expressed that Chasm had a good ability to create abstractions and reuse content, largely through the use of inheritance, extension and component concepts in the vocabulary.

Considering Chasm's overall abstractions and reuse, expert 3 stated, "I think the level of abstraction and framework is good, it's just that there is definitely a learning curve." Experts 2 and 5 questioned how useful the vocabulary was in its current size. For example, expert 5 said there "...could be problems with more complicated techniques as more components are needed." Expert 2 similarly said that it is "hard to understand the concepts and what they all mean". However, this expert also

stated that this was, "one of the major benefits of this, because most tools, unless you build the components yourself and know exactly how everything fits together, it's impossible to try different things out and put things together." In contrast expert 3 had few problems with the vocabulary, stating that Chasm "... provides a lot of terms so [developers] can be expressive." There were also concerns of reuse, as expressed by expert 2: "potentially [there could be] problems if [developers] broke out of the structure a little bit…[I]would have to go in and understand how Chasm works under the hood." Interestingly, in the section below (Behind the Scenes), the longitudinal evaluation suggested that developers did not need to know how Chasm worked 'under the hood' to effectively use the system.

6.2.4. Visual Representation

Many of the expert's comments were centered on the visual representation of Chasm in the ChasmGUI. Several issues emerged though the representations were well-received. For example, expert 1 said, "nice GUI", "intuitive GUI" and "I like the way the descriptions appear... very helpful." Expert 2 similarly stated that the "...GUI made things easy to see." Expert 5 stated that the, "graphical components, you might say the direct manipulation, the pointing and clicking, to create things and to access properties... those were efficient" and "it wasn't too hard to find things." However, expert 4 related that the "layout of the GUI could be improved."

Additionally, experts discussed: the visualization of the causal statements, concepts and components, automata tier visualization, and concept package display. The causal statements were "very well done" (expert 1) but there were "... problems with relationships visually" (expert 2). Expert 5 stated that the response relationships allowed visualization of logical operations in the system. Both experts 2 and 3 commented on the fact that not all connections between the concepts and components were shown, as some causal statements were added in the code tier. Regarding this, expert 3 recommended that, "there should be something connecting all things on the board. Nothing [is] represented until supplemental code later on."

Many comments also concerned the visualization of concepts and components. For example, expert 5 had trouble seeing "how the boxes fit together", referring to the box-like visualization of the concepts and components. Expert 2 stated that the "components and concepts look[ed] similar." Expert 5 also stated that the way the components were displayed they "visually didn't seem tied together, yet they were." Expert 5 liked the ways in which the inherited states in the automata tier were colored differently.

Expert 3 many comments display and organization of concepts in the GUI, which are displayed alphabetically in each package of concepts. The expert had difficulty finding concepts because they didn't know concepts' names and suggested a hierarchical display of similar concepts or inheritance for better organization. This expert also suggested that this information be exported thorough the creation of a documentation tool so the information could be read outside the ChasmGUI.

There were also concerns that the visual representation would not scale to larger systems. As stated by expert 2, in relation to the visualization of the Raycasting concept's automata tier, "a problem is that we are doing simple Raycasting and [it] used 45% of screen space just on it." However, expert 3 stated that they "...liked the graphical aspect of state machines and transitions."

Several of the other improvement recommendations concerned:

- Components, which were passed into the concept or passed to another

component of a concept, had no visual representation for this passing.

- The automata tier visualization needs work in transition layout to avoid overlapping actions and lines. Additionally, it needs to overcome its seemingly complex representation for even simple concepts.
- The experts like the ability to create notes for themselves. They admitted they kept notes while developing normally so this feature was seen as useful.
- The code tier may need to be in the ChasmGUI. As expert 5 stated, "an editor for the code tier would be useful." This was explicitly avoided so existing tools, which work well with code, can continue to work with the code tier. However, this comment suggests developer work between the tiers needs to be supported.

Overall, the domain experts liked Chasm and had positive comments and insights.

6.3. Longitudinal Evaluation

Initially included in the longitudinal evaluation of Chasm were four volunteer developers and the author. Three developed systems and completed the evaluation. Volunteer Developer B (VDB) was a graduate student in the 3DI Group at Virginia Tech who used Chasm to develop their Masters Thesis. The remaining three other graduate students used Chasm in their semester long project in the CS 5754: Virtual Environments class. Due to a mid semester group change (unrelated to Chasm), one of the groups split, reducing the number of Chasm developers from the class.

It was through this process that the ten reported case studies (see below) were created. The experiences of the volunteer developers with Chasm were captured through observation, similar to multiple Contextual Inquiries [Bey98], and a summative Cognitive Dimensions semi-structured interview [Bla00], the results of which are reported as themes at the conclusion of this section.

The author was available during the volunteer developer's use of Chasm answering questions, meeting during the semester and summer as necessary. The feedback and comments from these developers resulted in multiple Chasm improvements, some during the evaluation and others as areas for future work.

6.3.1. Volunteer Developer A (VDA)

The author (hereafter VDA) has been developing 3D interfaces for several years and has years of experience with the SVE toolkit. VDA has built several event-based applications in several languages with differing types of tools.

With Chasm, VDA created three case study applications (VDA1, VDA2, VDA3). The first two were research projects created with early Chasm versions. VDA1 has been published [Win05] while VDA2 has not. The third case study, VDA3, was a tutorial created to help developers understand Chasm as is packaged with the Chasm distribution. It also was used as the evaluation task for domain experts (see section 6.2.).

6.3.1.1. Baseline Factors (VDA1)

As shown in Figure 6-2, Chasm was used to implement a testbed application to evaluate factors affecting Raycasting selection [Win05]. This allowed for the evaluation of Chasm's utility in the development of a real system. Chasm was used to implement the Raycasting selection technique with highlighting feedback when the ray was pointed at an object. Chasm was also used to control the flow of the experiment, from the animation of target objects to new locations, to the pushing of

buttons to start and stop the experiment. With these uses, Chasm demonstrated that it is more general than a system to be used for 3DUI creation.

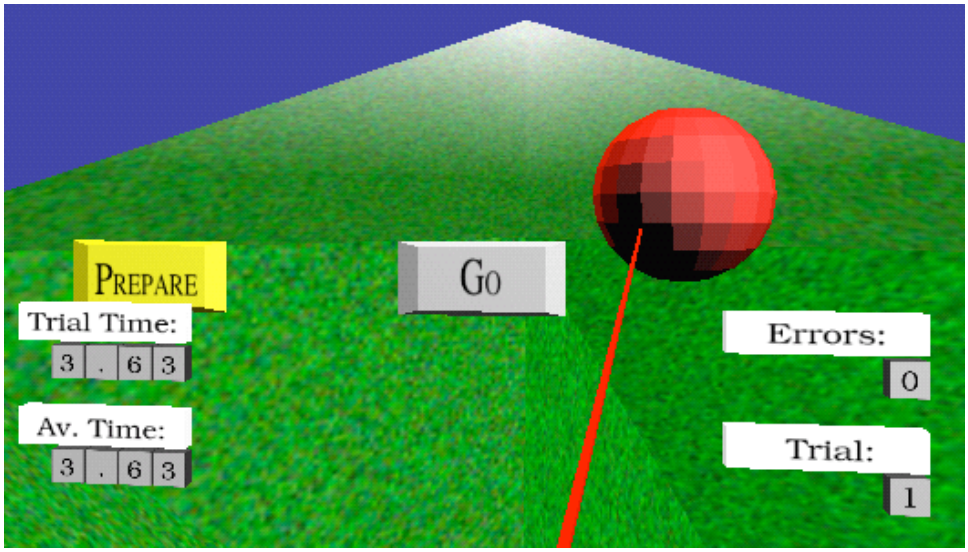

Figure 6-2: A test environment created in Chasm for the study of Raycasting.

While developing the experiment and describing the experiment's progression in common language, the implementation was naturally split into two concepts. TBExperiment consisted of 6 states, and controlled the experiment flow. RunTrials had 5 states and controlled a selection trial. All states were important for either their state entry function or in communicating with other concepts. For instance, the RunTrials.endtrials state notified TBExperiment that the trial was over by sending the TBExperiment.endtrial action. Also, the TBExperiment.runningtrial state notified RunTrials to start by sending the RunTrials.restart action. The entire application used a total of 87 state machines (33 unique), encompassed 201 states and 200 actions[8]. The total implementation time was not determinable as Chasm was in development while the case study was being implemented.

Chasm was particularly important in four aspects of the system's development. First, Chasm provided concepts specifically for time and animation. In the experiment, many of the target objects had to be animated moving and scaling. Extending the Animated concept and placing the move and scaling code into the state entry function performed this animation. Second, modifying the experiment's timing was relatively effortless due to the causal relationship. For example, the causal statement of "when the animation ends, advance to the next state" was associated with a dependence directly on the end of the animation. Therefore, changing the animation times did not require re-syncing of other parts of the experiment. Third, during pilot studies using the 3DUI, participants practiced a selection trial before the experiment began which affected the results. Because of this, the target object and selection ray were never to be visible at the same time, until the start of the trial. This new requirement could have been problematic to implement, as the target object and ray were already invisible and visible at different points in the trial due to other system behavior. In an event-based system, this functionality would have to have been inserted into multiple event handlers, potentially modifying all the existing behavior. As such, the entire application's functionality would require complete retesting. With

8 Due to the decision to report arrays of concepts as a single concept, this was reported earlier in [Win05] and had different values.

Chasm, the necessary modifications were easily added to the three states, which represented points in the experiment flow, of TBExperiment. A fourth situation illustrating Chasm's utility was the reuse of the Raycasting concept. Though originally developed for the selection of the target object in the experiment, its functionality was exactly what was needed for the selection of 3D buttons. These 3D buttons allowed the user to control the experiment flow. Reusing the Raycasting concept for this new task only required the creation of a new Raycasting concept. When the buttons were selected with the new Raycasting concept, they returned actions to TBExperiment to advance the experiment.

6.3.1.2. Flavors (VDA2)

"Flavors" of Raycasting was an experiment implemented in Chasm. Raycasting flavors were similar to the original Raycasting, but contained additional feedback or features to improve on the original interaction (see Figure 6-3 below). The first flavor was gravitational, used to pull selectable objects closer to the user as they point nearer to an object. This helped increase the amount of visible feedback to the user. Figure 6-3 (left) shows how the spheres closest to the ray were brought closer to the user. The second flavor was a yellow Snap-To Ray that extended from the user's hand, to point at the nearest selectable object (see Figure 6-3 right). The yellow ray disappeared if no object was within 30 degrees of angular error.

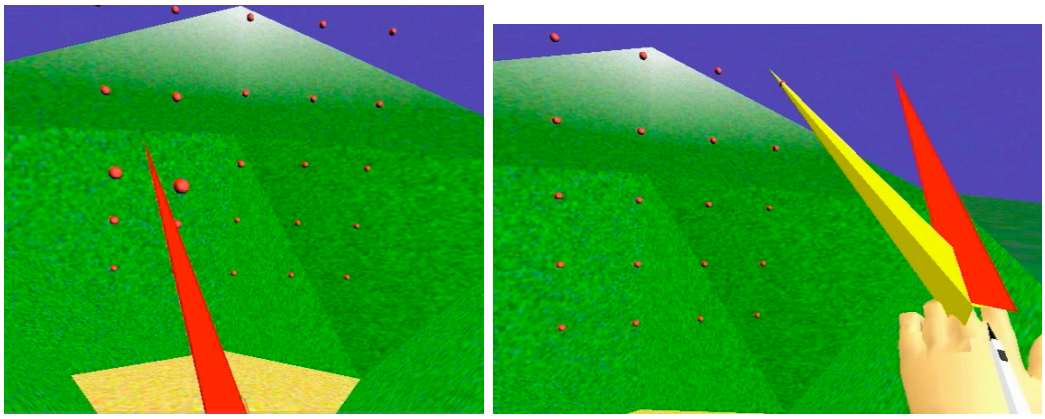

Figure 6-3: The addition of "flavors" to Raycasting easily created two techniques which improved on Raycasting, which would have been problematic in a normal implementation. The flavor on the left pulls objects closest to the ray closer to the user and the flavor on the right uses a yellow Snap-To ray to point to the closest selectable object.

This case study reused almost entirely the concepts from VDA1. This is not surprising given that the experiments were so similar though the degree of reuse was unexpected. In the automata tier, the concepts of Raycasting, TBExperiment and RunTrials required no changes and could be reused entirely. In the code tier, a change occurred only in TBExperiment. This involved the addition of a few lines of code to switch between flavors between trials. Though the code was inserted directly into TBExperiment, the same effect could have been achieved without modifying it , i.e., creating an extending concept that responded to states in TBExperiment. However, it was quicker to insert the code into the TBExperiment concept. As the ChasmGUI did not exist at this point, creating a concept took a few minutes of time.

The creation of flavors, which would normally require major code changes, was accomplished with only the addition of concepts. By responding to the Raycasting concept's states, the causal statements inserted functionality directly into the flow of Raycasting. Method calls to Raycasting provided any extra information needed about

the target objects. In addition, though each flavor was designed without regard for the other, VDA1 was surprised to see that Chasm allowed all flavors to run simultaneously without problems (see Figure 6-4).

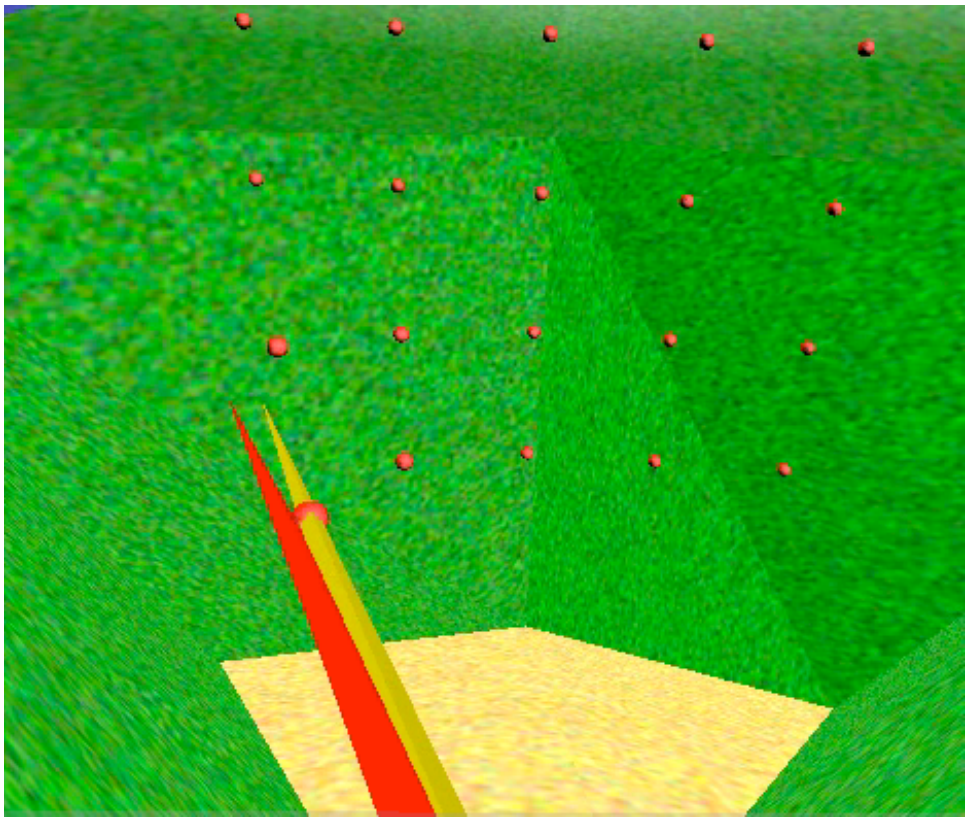

Figure 6-4. Both flavors of Raycasting were able to run concurrently in Chasm without being designed to do so. In this figure, both rays are shown and the spheres closest to the red ray are pulled closer to the user.

In Table 6-1 (below), the total number of lines of code to implement the flavors is shown in the first column. Of those lines of code, the number of lines of code directly calling Chasm functionality is shown in the second column. The Chasm code created states, actions and transitions for the concept and was only a small addition to the implementation. It is important to note that this code was later automatically generated by Chasm from the ChasmXML and this case study was performed before the creation of ChasmXML.

Table 6-1. The implementation details of the flavors of Raycasting show low overhead from the Chasm implementation.

	C++ Lines of Code	Chasm Lines	States	Actions
Flavor 1	98	14	3	3
Flavor 2	187	24	4	4

6.3.1.3. Chasm Tutorial 2 (VDA3)

The Chasm tutorial 2 demonstrated how to solve a simple 3D interaction problem inside Chasm. The tutorial gives step-by-step instructions and eventually creates a simple application. The description of the tutorial interaction is this: *When the user selects the end object out of a line of objects with Raycasting, have that object move*

to the center and above the other objects. The before and after pictures of the environment are shown in Figure 6-1 above. The tutorial has concepts for a button, a row of objects that can be selected, a Raycasting technique for the selection and an animation concept to move the object.

6.3.2. Volunteer Developer B (VDB)

VDB used Chasm to create three selection and manipulation techniques for their Master's thesis. VDB was experienced developing 3D interfaces and had a semester's experience working with the SVE toolkit and VE hardware. Their Chasm interaction occurred across a semester and into the summer. The selection and manipulation techniques developed by VDB are presented below and were incorporated into a single application. In this application, only one of the following techniques ran at a time, specified by a command-line argument. It is important to realize that none of the techniques implemented by VDB were simple interaction techniques as they involved at least some special aspect to improve the interaction. As such, VDB experienced Chasm in challenging tasks.

Outside of the author (VDA), VDB likely had the greatest exposure to Chasm and over the longest period of time. As such, VDB was particularly informative for issues within the automata tier in Chasm. For example, VDB believed that actions were sent to states after exposure to the ChasmGUI, where the arrows in a causal response go into the concept and not a state, VDB's understanding of the system was adjusted accordingly. Additionally, VDB found it was interesting how a state can operate as a decision state by returning an action and controlling the branching of behavior in a system. Currently in Chasm, there is no explicit difference between a state and a decision state. Lastly, VDB at first found the idea of "looping until something happens" to be non-obvious. This looping was used for creating continuous behaviors in a concept. During VDB's use of Chasm, they found the "looping" to be quite useful and a "good design model".

VDB also realized other benefits of Chasm. First, VDB used and liked the supplemental files. Second, VDB realized that tiers allowed them to design at various levels of complexity. For example, when VDB was designing technique VDB2 below, VDB designed and tested the flow of the technique by connecting to keypresses. After VDB2 was tested, VDB connected it to the correct contexts.

VDB did have trouble with builders, however. Specifically, at issue was understanding what code to place in the builder and what to place in the concept. Since the builders add complexity to development, by forcing developers to consider splitting the code at the same time they are designing it, this problem was understandable.

6.3.2.1. Go-Go (VDB1)

The Go-Go technique [Pou96] is an arm extension virtual hand technique used for the selection of objects. Go-Go improves over the traditional linear scaling of the user's hand position by using a non-linear function to place their hand after the user's hand extends a reasonable distance from their body. In this way, the advantages of working close to the body remains but users gain the ability to select objects at a distance. It was described in [Pou96] as: " a technique [that] uses the metaphor of interactively growing the user's arm and non-linear mapping for reaching and manipulating distant objects. Unlike others, [the] technique allows for seamless direct manipulation of both nearby objects and those at a distance." VDB did not specify an envisioned behavior for the concept. In terms of interaction, this was the simplest of the three techniques implemented.

6.3.2.2. HOMER (VDB2)

The HOMER technique [Bow97] is a hybrid technique for selection and manipulation. It uses Raycasting to select objects and an arm extension metaphor for manipulation. In this way, HOMER gains the benefits of Raycasting selection and the ease of manipulation of arm extension techniques. It was described as "the most basic technique…the user grabs the object with the light ray, as before, but instead of the object becoming attached to the light ray, the virtual hand moves to the object position and the object is attached to the hand. When the object is dropped, the hand returns to its natural position. This allows simple grabbing and manipulation with no extra work on the part of the user" [Bow97] .There was no envisioned behavior specified by VDB.

6.3.2.3. Voodoo Dolls (VDB3)

The Voodoo Dolls technique [Pie99] is another hybrid technique for the selection and manipulation of objects. It allows for the selection and manipulation of objects at a distance, using an image plane selection technique [Pie97], but still having an egocentric viewpoint of seeing the objects situated in the environment. It was described in [Pie99] as "a two-handed interaction technique for manipulating objects at a distance in immersive virtual environments…. With the Voodoo Dolls technique, the user dynamically creates dolls: transient, hand held copies of objects whose effects on the objects they represent are determined by the hand holding them. … When a user holds a doll in his right hand and moves it relative to a doll in his left hand, the object represented by the doll in his right hand moves to the same position and orientation relative to the object represented by the doll in his left hand. The system scales the dolls so that the doll in the left hand is half a meter along its longest dimension and the other dolls maintain the same relative size; this allows the user to work seamlessly at multiple scales." There was no envisioned behavior specified by the VDB.

6.3.3. Volunteer Developer C (VDC)

VDC created four travel techniques to compare in a testbed evaluation for a graduate level virtual environments class at Virginia Tech. The first three techniques were common travel techniques and the fourth integrated two of the previous. VDC had almost little to no computer graphics or 3DUI development experience pre-Chasm. As such, many of the issues encountered by VDC centered on normal 3DUI development issues. One of their major problems dealt with limiting their travel technique to travel along the horizontal pane, which was not a Chasm related issue. Additionally, there were issues in the use of the SVE toolkit. After Chasm, they wished there were more tutorials, especially involving SVE and the supplemental files.

6.3.3.1. Wand-Based Travel (VDC1)

The wand-based technique is a simple point and travel technique commonly used for travel [Bow04]. When the button is pressed, the travel occurs in the direction they are pointing. In the concept VDC created to represent this technique, they specified the envisioned behavior as: "Basic travel, const velocity, const direction and orientation. When user presses button, get initial heading and orientation from wand, travel at const velocity in that direction until button released."

6.3.3.2. Variable Wand-Based Travel (VDC2)

This travel technique is the Wand-Based technique of VDC1 with the added ability to change velocity during travel. The means of this is the extension of the hand into the environment to increase velocity and pulling the hand in closer to decrease speed.

VDC gave the envisioned behavior as: "inherits from wandBasedTravel, adds variable velocity based on distance btw user's chest and wand [newline] when in state:traveledcontinue setVelocity equal to a function of the distance between two positions."

6.3.3.3. Gaze-Based Travel (VDC3)

Whereas the previous two techniques took the direction of travel from the wand, the gaze-based travel technique takes the direction of travel from the head. VDC gave the envisioned behavior of: "basic gaze based travel. Inherits from oneBaseTravel. Takes direction from HMD, not wand." The oneBaseTravel refers to Wand-Based Travel as the participant misinterpreting the term "wand" as "one". VDC was later corrected though this was never updated in the envisioned behavior of VDC3. Interestingly, VDC designed this technique as inheriting off of VDC1, and only had to change the concept from which the orientation for travel came.

6.3.3.4. Jump Travel (VDC4)

The jump travel technique is a dual travel technique that incorporates the variable wand-base travel technique as well as the ability to travel to selected points. The second form of travel, traveling to selected points, used a scaled arm technique to reach into the environment. When the 2nd button was pressed, the user would travel to that point in the environment. The envisioned behavior of this technique was described by VDC as: "the user travels via wandBasedTravelVariable, constraining motion to the horizontal (xz-plane). The '1' Button on the wand is used for wand-based travel motion. The user can select a point in space by positioning the wand at that location via arm extension and travel to that point by pressing the '2' Button." Both travel techniques were able to interact without conflicting with each other's functionality. Additionally, both techniques had travel constrained to the 2D plane.

6.3.4. Volunteer Developer D (VDD)

Recall that VDD was in a graduate-level virtual environments class at Virginia Tech. VDD was an experienced developer and had experience in computer graphics. With a mid-semester project group split, VDD decreased the scale of their project and eventually stopped using Chasm (for reasons unrelated to Chasm). Despite this, however, VDD was exposed to Chasm development enough such that VDD was able to map-out (in paper format) several of their techniques in a pseudo-Chasm representation before development in pure SVE.

During their early use of Chasm, and later when just developing with SVE, the majority of issues VDD had were not Chasm related. For example, VDD had compiler and linker error messages, troubles loading object and material files in SVE, difficulties using the trackers and problems debugging a running SVE application.

6.3.5. Volunteer Developer E (VDE)

VDE was also in a graduate-level virtual environments course. Though VDE was new to computer graphics, they were an experienced developer. Similarly to VDD, VDE also used Chasm initially but stopped when the project group split (though this occurred in a different apart from VDD's) and VDE also continued to design their system using Chasm's visual representation on paper. VDE did not have a summative interview but their use was observed over a significant period of time throughout the semester.

Taken together, volunteers B, C and D represent a spectrum of Chasm experience and exposure. Recall that VDD had the shortest exposure, resulting in comments

centered on learning and Chasm first impressions. VDB had the greatest exposure, resulting in more in-depth, thoughtful comments that largely overlooked earlier conceptions and beliefs. Finally, VDC's experience fell somewhere in-between these. This can be seen in their comments in the following themes.

6.3.6. Identified Themes

At the end of the volunteer developer's use of Chasm, a semi-structured interview was completed with VDB, VDC and VDD using the Cognitive Dimensions Questionnaire [Bla00]. Using a multiple pass affinity diagramming process [Bey99], four main themes were developed. These include representation, structuring thought, designability and learnability. Note that because these themes are complex, additional sub-themes used to support these 4 were also developed.

6.3.6.1. Representation

Included in the theme of representation are comments concerning notation, visualizing the representation in the ChasmGUI, issues in using code with Chasm's higher level tiers and behind the scenes questions.

The notation of Chasm was largely understood though some issues emerged. For example, as VDC stated, with time it was "very easy to see the various parts of the notation" and it was "no problem once [I] got the hang of it." VDB stated that "the GUI notation was short" and that the notation "was easy... Chasm states were pretty short too."

Some problems concerned the interwoven transitions and causal statements. VDB stated that the invoked actions, continuation actions and causal statements "were similar and there might have been some confusion on this at the beginning." VDD similarly noted that the causal statements were confusing. Some of these experts related, however, that with experience this issue was overcome (VDB). From my observations, it seemed developers had difficulty realizing that actions were local to a concept. For example, in early paper drafts, the volunteers mistakenly drew actions of one concept entering into another concept's state.

There were additional issues, largely in the ChasmGUI, where the tools became intrusive. For example, VDB said "starting up was a hassle, making a new concept for something trivial and small." This was evident in the author's (VDA) case study as well. In VDA2 (flavors) it was easier to modify a concept, TBExperiment, than to extend it with a new concept. VDD noted that in the ChasmGUI, "there would be certain ways of doing things just to make things work." Much of the problem is due to the ChasmGUI's inflexibility and not the representation. This issue will be discussed further in the Designability theme below.

Finally, the representation's split between the top tiers and the code tier was confusing. The developers were not always sure how much state to contain in the automata tier as opposed to the code tier. VDB noted that, "you may not have intended keeping the state within itself….like Voodoo Dolls, keeping track of what's going on separate from the state [of the concept]." One of Chasm's benefits is that the developer decides where to place state. Better documentation and guidelines of this feature would help resolve this confusion.

The representation was visualizable in the ChasmGUI, which helped the developers. The developers considered its representation to be natural and familiar. However, there were some problems, as important aspects of the representation were not visible. VDB commented that the "[ChasmGUI] made things easier to visualize. It helped a lot when I was deciding relationships between concepts." While developing

VDC said that, "most of the dependency changes are visible [and that] you can see what states affect others and what messages." VDB employed their own strategy here, using the ChasmGUI in one window, to reference information like state names, CIF and component concepts while they were developing the supplemental files. The visualizations were also useful for learning from the existing code. As VDC noted, "a couple of times I've looked at [the author's] code and open[ed] the ChasmGUI...[and] for the most part figure[d] out what is going on."

The developers also found the ChasmGUI to be familiar and easy to think about. For example, VDD stated that "because Chasm is kinda a UML and things that I am familiar with... the ideas of state diagrams helps me to organize things a little bit better." Also, "The way the notation is, [that is] how I would sketch it out. That's certainly nice. There is a direct mapping on how I would toy around with it on paper and how it's represented in the system." VDC said that "It's really easy for me to think about this. Ok, what's happening. So, now I'm here and get the reading ... whatever. That's just a really good way for me to think... graphically." There were some issues with meaning in part of the representation, as noted by VDD. "Sometimes figuring out how to get the information you need is kinda difficult. They might be represented by a red square or a green square... once you understand what everything means, it's really easy." This comment regarded visualizing states in the automata tier of a concept that were inherited from a parent concept. In general, the visualization seemed to help the developers, *when it was complete (see below)*.

Two major problems with the visualization are: passed components and Chasm structures created in the code tier, which are not visualized in the ChasmGUI. For instance, in state entry functions, developers often needed to create component concepts and causal statements. These structures, since they were not stored in the ChasmXML, were not visualized in the ChasmGUI. Regarding passed components, VDB commented on a situation where, "I had opened something, the HOMER technique, with a bunch of components, and I was trying to describe [the implementation] to Dr. [X]. It had a button over here that didn't have any causal relationships tying it to any concepts. It's actually doing something but you have to go into the code. And that happens a lot. You are making concepts that are separate components and not connected with causal relationships."

The code in the code tier was unusual for the developers, which also led to problems. This was due to the use of static constructors, builders, preprocessing of files and knowing where to place code in the code and header files. As VDB stated, "I think there is a steep learning curve. Not Chasm itself, but using it in conjunction with C. It's pretty complicated. Just the way that you have everything set up. Like, static constructors. I've never used that. You never have to use it but it is there. So many files like chasm.h gets erased. I've never used that but it was pretty explicit not to use it." In some cases, basic OO issues became a problem. Access levels were problematic for VDB who said, "the public, private, protected I would get mixed up."

The use of builders and supplemental files helped this process. As VDC stated, "I used supplemental files the whole way along so logically, it helped me structure things a bit." Similarly, VDB stated, "I relied on the code and the supplemental files in isolating that code so it was easier to see." Regarding builders, VDB was surprised how useful they were, stating, "The builders were very helpful for separating out functionality. I didn't think I'd use it." The same developer had commented earlier on problems with the builders and it is likely that their continued use of builders led to greater understanding of these processes over time. VDB was the only volunteer to use the builders.

CIF, important in the connection of the automata and code tiers, was understood and was not associated with further problems for the programmers. For example, on CIF, VDC stated, "I have not run into any major problems along those lines" and VDD similarly suggested that "the CIF was pretty straight forward."

Chasm handled several important activities behind-the-scenes. This included event handling and passed components which were handled without their knowledge. VDB noted that they were "not sure" how events were handled and that they "never really got into that." Also, though VDB said "I really didn't care about the order because I knew the Chasm system would call them in whatever order it used," VDB did recognize that this could be an issue. They said "it was in the back of my mind that it could call it in any order and it could mess up." VDB asked, "How does it do it?" This same developer did have problems with events, though this issue was rectified with an updated version of Chasm with Gathers. For example, VDB stated, "at first it was a problem, but then, as I got the revisions, it disappeared." Passed components were also successfully handled without developer knowledge. VDB stated that "all I needed to do was pass in a component and ... it added that extra functionality. I didn't have to type in anything." As with the event handling, developers were unaware how this functioned. One comment was, "I can't think off the bat how that happens. It's pretty neat." The main reason developers were able to ignore these behind the scene activities was because the debugging output provided enough information. As stated by VDB, when "something would happen and it would error out... it would tell me what [action] was sent and what state it was in and that there was no transition for that [action] in this state. So I would know what exactly what was happening." There were no comments on these behind the scenes activities of Chasm from VDC and VDD, suggesting that these developers also had positive experiences.

6.3.6.2. Structuring Thought

Chasm structured the volunteer developer's thinking. By forcing them to think in Chasm's terms, they were able to consider development in a linear fashion, decompose and compose functionality easier and create implementations that are more understandable. These benefits can be attributed to Chasm's state machine representation and forcing developers to think in terms of changes and response to change.

All the volunteer developers commented on how Chasm forced them to think differently. This was problematic at first but they liked it in the end. VDC had several comments about Chasm's way of thinking. Their initial impressions were, "At first, [thinking of it as a state machine was a problem] cause it's not the usual Object-Oriented kinda way and not really thinking about the data so much as what changes. Once I got into doing that, it wasn't so bad." VDC also stated that "thinking in the states machines, in terms of that, it's a logical way of thinking but it's not something I do every day." After spending more time with Chasm, the developers concluded that "at this point, it seems like a very logical way to think about designing the technique." VDB stated, "Yeah, [it] definitely changed how I thought" and, "at first, [the way Chasm makes you think] was difficult." VDD, the volunteer developer with the least Chasm experience, had more issues with this than the other developers. For example, he said, ."it kinda requires that extra step for me to really get to the end result. It is hard for me to get to the end of the problem" However, even VDD recognized that "the whole way of representing this problem, solution, scenarios is a little different for me, I think it is a really good one once you get there."

Chasm not only changed the way that the volunteer developers thought about development, it also enabled the volunteer developers to develop in a linear fashion

with fewer side effects. In a sense, developers had fewer concerns, making implementation easier. As expressed by VDC, "I'm not envisioning all the issues that I have to deal with as I start coding and resolving the issues, keeping up with what I am dealing with in each state and documenting that." VDD also said that Chasm "lets you focus less on the drudgery of making an event system work and more of, ok, I'm at this point in the progression, what do I do now?" VDB recalled a situation where a small change created only a small impact despite being late in design, "I had, say, a four state interaction technique... and then later on I figured out I needed another state that was ending the interaction technique... so, I just added it. It was easy to add. And where I needed it, I could add a causal statement to that new state and everything worked fine. Everything that had previously been defined worked and everything that was newly defined worked. Just added a state and a causal statement fixed everything that I needed. And that was way later in design."

Composition and decomposition of behavior, largely through modular thinking, is improved in Chasm. As VDD suggested, "at a really high level, it forces you to be modular, which is a lot more maintainable, implementable, especially because Chasm takes care of a lot of the low level event handling and stuff like that." This additionally occurs through the specialization and composition of existing concepts. VDC stated, "It was really easy to kinda say, ok, well when I did that variable speed wand based travel, to just inherit from that and then [play] with things." VDC also stated that "looking at it at a higher level, we were able to implement wand based travel without the variable speed and using inheritance, we can say that was the partially completed project and it was a very simple matter to use inheritance and add in the variable speed or to use inheritance and make a gaze based travel." The ability to compose concepts was important too, as developers realized. VDD said, "I think in certain ways, the system provides you with concepts that are already defined and you have your own abilities to extend these concepts or create new ones." Additionally, VDB recognized how concepts can be extended in many ways. For example they said, "if I needed to add something to Raycasting, it would be *outside* (emphasis added by developer) it versus inside it. It depends what I was adding but I could fashion it in that way or in another way like you added a Switch inside Raycasting and I could make jumps through or add a concept inside a concept."

The structure of the implementations was understandable, due in part, to the importance of naming. As VDC stated "I can look back at [my implementations] and actually figure out very quickly what is going on." Much of this came from the importance of naming in Chasm, where one word can carry a lot of weight. Or as VDD says, "Because it requires you to name it something significant, there is [an] amount of your own association that you put with it. "You can be descriptive enough with just the name of it if you want to be a brevist. If you want to be more descriptive and really tell about something there is a description field and most of the things you can describe in plain English, describe what you are doing. That actually helps organize things a lot, even more than a name. Some things can be descriptive in just a state change." VDC goes on to say that "changing the name [of a state] can be a lot quicker if I'm, 'Oh, travel is not a good name, lets call it went.' Its easier for me to think like that."

Overall, it was helpful for the developers to structure their thought processes in terms of causality between state machines and sequence. The volunteer developers had positive comments about the structure, relating that this helped them to identify the sequence in their systems. For example, VDC commented that there is a "certain order... which starts with describing it in common language and then coming up with

states." VDD stated that when they though about it Chasm's way, they "start[ed] to see states and transitions instead of transitions and things that have to happen." VDD stated that this was helpful and VDC said that this was a "logical way of thinking." Here is an example of VDD's thinking proceses: "So, ok, I'm in the inited state then the user is going to press the button and I'm going to the travel state. Its really easy for me to think about this." As VDC stated, this made it possible to "sketch out what [were] the states, what was going to happen in the state, what sort of messages were going to be passed in the state, what sort of events are going to trigger these." Causality was not an issue for the volunteer developers, as illustrated by VDC, who stated that "the notion of events and responses seems to be... and passing a message and going to a new state, seems as logical and straight forward as it could have been." The ChasmGUI helped this as VDB noted, "The GUI helped to write things out in a flow chart sort of thing."

There are areas for improvement, however. Notably, Chasm likely put too much emphasis on the automata tier and not the flow or sequence. This is supported by VDD's observation that "the difficult thing about the notation... people with a lot of programming background tend to orient problems in a sequential set of solutions. You look at the problem and this needs to happen and this needs to happen... In this notation, most of this stuff is abstracted out and we are more concerned about how you get from this position in the implementation to this position in the implementation and that ordering, at least for me, is really difficult. Because, I don't want to say, 'Well, why don't I stay in this position in the implementation until something else happens and there is a transition.'" This likely explains similar comments from VDC, who noted that "nothing was that difficult to describe once you get into the frame of mind of thinking in terms of states and state changes and messages between states." Thus, as with the "not normal thinking" section discussed above, experience matters in how the developers will understand these causal relationships. This was expressed by VDB's comment that "at first, it was difficult."

6.3.6.3. Designability

Chasm allowed the developers to sketch out their ideas, if not in the ChasmGUI, then at least in its notation on paper. VDD's overall comment as, "I think essentially what you have at the high level is a really nice design tool. It kinda makes sense of all the craziness of all the programming." Chasm allowed the developer to choose the level of precision and evaluate the system as they worked, and this helped. Also, Chasm controls changes, making the addition of a new functionality relatively easy. They did have concerns about the network of causal statements and transitions as well as feeling that a Chasm implementation has a slow start.

Developers used Chasm for sketching their design. As VDB related, the "GUI helped to write things out in a flow chart sort of thing. It wasn't really a flow chart. It was kind of a standard notation." This developer also said that "everything was designed on the fly. I never really had a rough draft, just straight to implementation." VDD had similar comments: "the way the notation is, is how I would sketch it out. That's certainly nice. There is a direct mapping on how I would toy around with it on paper and how it's represented in the system."

Chasm's use altered how they sketched their designs. VDD said, "Actually using those steps as a filter for pulling out this important stuff. When I think about it that way, I start to see states and transitions instead of transitions and things that have to happen. I found that really helpful just in us doing that." Regarding sketching out ideas, VDC similarly stated that this was "very possible. Just using the state machine and message passing. … Just with pencil and paper, trying to sketch out what was [*sic*] the states, what was going to happen in the state, what sort of messages were

going to be passed in the state, what sort of events are going to trigger these... just thinking about it with pencil and paper is maybe a little bit easier than [the ChasmGUI]." VDB said, "I needed something to happen here. And, I'd realize it was because I needed this concept to do something to be in a state. If it entered a state, I could have it do something. Or if it entered a state, I could have other concepts do something based on that. So that knowing that, I could design based on that. If I needed this concept to do something, I needed this concept to move."

Part of being able to design is choosing the level of precision at which to develop, to be as specific or as abstract as needed. In addition, this includes the ability to build something and test it fully or in small parts at a time. In terms of level of precision, there were comments from VDD about descriptiveness and brevity (see 6.3.6.2. above). VDB also said that "if I didn't want to be too precise, I wasn't. That was doable." This was beneficial because developers often did not know what they wanted at a point in time, as VDD stated: "you can say this is my concept that is going to do all this stuff and I'm not going to be precise about it right now because I don't really know what it needs to do. But, I know this state needs to exist." This is in concordance with VDB's other comments on design and adding to a system, "Yeah, you can try out partially completed versions. I remember doing that quite a bit. That's one of the strengths brought up in your paper. It's having compartmentalized design. You can shut something off and try it out. That's more on the concept level. If you were making a concept that conflicts a concept, you can make a smaller concept and then design a more complex concept or you can work from the top down, design the larger concept and have little black boxes that don't really do anything. Design those later. I think it didn't matter which way I went."

Developers did not have problems making changes to Chasm systems, during implementation or for reuse as a new system. For example, VDD stated: "I would say changes are very easy to make." VDC also said, "most of the changes at this point in my proficiency [pause] it's easy to make changes. If it is a structural change and I need to make a change to how a concept works then I can pull up the GUI, save it and remake. That's very nice and easy. Any other changes are no different than any other C++ code. In terms of flexibility, it is very flexible." VDB also thought that it was easy to use the system itself to make changes. They said that "adding a new action or adding a new state, that really didn't change or cause bugs as far as I could see." As VDB expressed, this may be because "everything was pretty much isolated. If you had something working on a certain state and add a new state, it wasn't going to change anything. You could add functionality that way."

There were several specific cases of this; see VDC's earlier comment regarding their implementation of wand-based travel in 6.3.3. above. VDB had also said that "an example of adding a new state: I had say a four state interaction technique ... and then later on I figured out I needed another state that was ending the interaction technique... so, I just added it. It was easy to add. And where I needed it, I could add a causal statement to that new state and everything worked fine. Everything that had previously been defined worked and everything that was newly defined worked. Just added a state and a causal statement fixed everything that I needed. And that was way later in design." With regards to the reuse of concepts during their use of Chasm, VDB said that Chasm "allows for plug and play. I can use the same [concept], like highlight selected. I could use the same [concept] for Go-Go, HOMER, Raycasting."

Chasm also helped protect the concept, as noted by VDB: "It was hard to do something radically different from the parent. I definitely used inheritance a lot."

6.3.6.4. Learnability

The volunteer developers comments had many associations to the learnability of Chasm. The developer's comments on learning centered on how to think like Chasm and how to use the tools that Chasm provides. The developers discussed the documentation available and also commented on the fact that there were many confounding factors to their learning.

Though Chasm represents a different way of thinking for the developers (recall the structuring thought them in 6.3.6.2.) they were largely pleased with this shift. Recall what VDD stated, "The whole way of representing this problem, solution, scenarios is a little different for me. I think it is a really good one once you get there." The thinking process may have been different for these developers because they were thinking in terms of state machines as well as responses between state machines. In relation to thinking as a state machine, VDC said "its not the usual object oriented kinda way and not really thinking about the data so much as what changes. Once I got into doing that, it wasn't so bad." VDC also expressed that it was not difficult to describe "once you get into the frame of mind of thinking in terms of states and state changes and messages between states." VDB similarly expressed that Chasm "definitely changed how [they] thought." Supporting this, the author observed several participants using a Chasm representation for other unrelated work, including other class projects and homework.

Causality between concepts and actions were initially problematic, though this issue was resolved with increase experience. Recall that VDD was the volunteer developer with the least Chasm experience. They stated that "responses kind of confused me and I found myself making mistakes with that quite a bit." VDB agreed, saying that "I guess [actions] being sent and [causal statements] were similar and there might have been some confusion on this at the beginning." VDB also stated, however, that this was clearer later. For example, "I think they were just confusing because they were just [actions] being sent either way." VDC had no issues with causality, saying that "the notion of events and [causal statements] seems to be... and passing a message and going to a new state, seems as logical and straight forward as it could have been." Again, experience matters.

Along with this, the design of certain concepts was, at first, confusing, though they learned to use them. Concerning continuous behaviors, which are implemented by concepts continuously returning an action to themselves, VDC said that they were "still not sure [they] got the animated move totally figured out" and that they "have an overall idea but ... not at a low level." Similarly, VDB stated that the "loop back to itself" was "backwards" and that they did not "think about things like that, but it is actually more practical than a way [they] would have thought." Another useful idea was lazy evaluation, or only updating a concept when it was needed. Regarding this, VDB stated, "Your concepts were very confusing... like for example, the [lazy evaluation]. I had no idea what that was for. It was not self-explanatory but after looking through the code and something that used it, I could see how it could be useful. And actually, it is in 6 or 7 of the concepts I have made. I use it quite frequently."

The developers commented about difficulties in learning the system of Chasm. These comments centered on the Chasm and SVE gap as well as access and CIF. In relation to Chasm and SVE, VDC stated, "I was trying to figure where Chasm and SVE work together and stuff. It may have added to the confusion..." as well as, "...initially finding exactly where to insert which SVE functions or C++ code in the main.c file or entry function [was hard to understand]." VDD stated, "I would get my [actions] and states done and I would say, 'Ok, now I've got my program.' There is a little bit of

gap between what that is and what SVE needs to have to actually make something like that work." Access and CIF were also problematic. VDB said, "The public/private/protected I would get mixed up." VDC similarly had issues "trying to figure out what sort of relationships [they] need[ed] when bringing concepts into the system." For example, VDC wondered, "is it going to be passed or protected, internal, external, inherited or have an amalgamation? Has-a versus is-a? Aggregation [is problematic]." Much of the CIF problem may be due to issues with premature commitment in the ChasmGUI, not CIF itself.

There were other system issues that had to be learned. From my observations, it was noted that developers had problems knowing that certain causal relationships existed but weren't visualized, understanding the visualizations in the ChasmGUI (especially for inheritance) and using the ChasmGUI and tools in general. VDD summarizes this issue well with, "I guess there is a lot of blanks to fill in and it is not always inherent what blanks need to fill in that transition." VDB similarly said, "I remember just running into problems. I would have to figure out workarounds. That happened a lot."

Documentation and help was useful though the developers largely learned by the examples provided in Chasm. VDC noted that "You have lots of good examples and code." However, VDC said that the tutorials were only "somewhat useful." This may be because "there is a hole in that the first one doesn't use the ChasmGUI and the second one starts with the code in place. We don't have a tutorial that builds something from the ground up." The author's implementations were used as helpful examples as VDC implies: "in almost all the example code, you used supplemental files and I was modeling a lot of my code after the selection and manipulation techniques you used... it helped me a lot." These developers also used the envisioned behavior and notes in the author's implementations. VDC stated that they had not "run into any major problems [regarding CIF]" because the author "has been helping with that."

There were confounding factors that hampered learning of Chasm. While developers were learning Chasm, they were dealing with other tasks as well, which increased their levels of confusion. This included programming issues in C++ (Java was the language they were the most familiar with) such as normal compile and linker errors, information hiding with access levels (public / private / protected) and problems with include files. SVE itself presented many problems to the developers, including loading in models and so forth. VDB understood this well, saying that "the most difficult thing was working with SVE. It had nothing to do with [Chasm]." Graphics and coordinate systems were also problematic for the developers. VDC had only moderate graphics experience before Chasm and stated, "once I figured out the coordinate system and how my position moved, I made mistakes getting used to 3D graphics, but I don't think there were Chasm mistakes I was making." Additionally, Chasm used several coding methods which were likely problematic to the developers, including the builder pattern, preprocessing of files and static constructors. With more experienced developers (i.e., domain experts), it is likely these would not be issues.

6.4. Measures and Metrics in Case Studies

Quantifiable measures and metrics from the case studies above are used to make claims about Chasm and compare Concept-Oriented Design in Chasm to other development representations. Measures are reported where possible. This includes measures of the automata tier structures as well as reuse, collected by looking at component concepts and inherited parents. Metrics are computed from these

measures. These metrics represent a higher-level understanding of the case studies that the simpler measures would not express. Metrics are reported for the state space, action space and interaction space. These measures and metrics are used to validate many of the claims of Chasm.

6.4.1. Measures

The measures of the case studies were collected by code walk-throughs of the C++ and ChasmXML representation. Table 6-2 shows the measures collected for each concept. These measures are a means of understanding the complexity of the concept's implementation.

Table 6-2: The countable measures of cases studies, collected by code walk-throughs of C++ and Chasm.

c – concept	p - parent
s – state	m - component
a – action	r - response
t – transition	f – state function

6.4.1.1. Collection Issues

Several issues, including time, repeated similar elements, and inheritance and components are discussed below. These issues further complicated the collection of the measures.

First, development occurred over time. During this time, different versions of the vocabulary were used. This resulted in concepts that were nearly identical, or even identical, but with different names. Because of this, similar concepts were marked as duplicates and counted as the same concept. However, if the concepts were markedly different in their measures, i.e. had extra states or components, they were marked as two separate concepts, even though they represented the same envisioned behavior.

Second, many concepts were used repeatedly for similar purposes and separately counting them would lead to a grossly overrated measure of complexity for the case study. For this reason, when a concept was repeated for a similar purpose, the duplicates were counted as one concept. For example, a concept with a Set of 20 Integer concepts would be measured at 21 (1 Set concept and 20 Integer concepts). As each Integer in the Set is treated similarly, this measure of 21 concepts is a much higher complexity rating than it should be. So, the repeated Integers would count as a single concept. Without this reduction step, simple concepts would have been overrated in their complexity, making the measures and resulting metrics less representative of the true measure of complexity.

Third, counting these measures was complicated because of the use of inheritance and component concepts. For example, the number of states of a concept is the sum of all states of its own and any in its parents. In addition, component concepts add states to the concept, which increase its representational complexity. But, these states are not actually in the concept increasing its design and development complexity. These cases require terms to clarify the measures.

6.4.1.2. Computing the Measures

The measures of a concept are summed with the following general summation equation in Figure 6-5 (below). These equations compute the countable measure for

Z from the internal measures Z_i and its component concepts Z_m. Internal measures, Z_i, are composed of the current concept Z_c and its parent concepts Z_p. So, s_c (see Table 6-2) for a concept is the number of states in the current concept and s_p is the number of states from its parents making $s_i = s_c + s_i$.

$$Z = Z_i + Z_m$$
$$Z_i = Z_c + Z_p$$
$$Z_p = \sum^{p_c} Z_i$$
$$Z_m = Z_{pm} + Z_{cm}$$
$$Z_{pm} = \sum^{p_c} Z_{cm}^{p_i}$$
$$Z_{cm} = \sum^{m_c} Z^{m_i}$$

Figure 6-5: This general summation equation is used to compute most measures for a concept. In this equation, Z refers to the measures in Table 6-2.

These terms of the general summation equation (Figure 6-5) are explained in more detail below.

- Z is the total count for the measure in a concept. This recursively includes all parents (and their parents) and all components (and their parents and components).
- Z_c is the concept's own value for the measure without parents and components. This is the magnitude of the addition to the measure from this concept.
- Z_p is the sum of all the concept's parents' internal measures. This includes each parent's parents through the entire inheritance hierarchy. This is the magnitude of the measure this concept is incorporating into its own representation.
- Z_i is the sum of the concept's internal measures, which includes itself as well as all its parents. This is a good measure of the design and development size for the concept.
- Z_m is the sum of a concept's components for a measure. This includes its parent's components (Z_{pm}) as well as its component's own (Z_{cm}). This is the magnitude of the measure that was not considered by the developer during design and development.

Table 6-3 (below) shows the measures for each case study. The number of unique concepts in a case study is shown in parenthesis.

Table 6-3: Measures for the case studies are reported below. Highlighted are the highest and lowest values for a measure.

Case Study	c (unique)	s	a	f	t
VDA1	61 (20)	130	131	105	266
VDA2	87 (33)	201	200	186	510
VDA3	25 (16)	51	50	43	102
VDB1	28 (21)	66	67	48	162
VDB2	26 (22)	58	60	44	141
VDB3	82 (22)	201	208	145	498
VDC1	12 (9)	29	28	19	65
VDC2	15 (13)	37	35	25	86
VDC3	18 (12)	43	42	30	101
VDC4	26 (17)	58	56	41	122

6.4.1.3. Reuse

The case studies showed that developers were able to reuse concepts in the vocabulary and compose new concepts from the existing. There were a total of 94 unique concepts in the case studies. After removing duplicates, as discussed in the above section, 80 distinct concepts remained. At the time, 22 of these were in the vocabulary, 25 were created by long-term developers, 23 were created by the author and 10 were the top-most concepts of the case studies. Of these 80 concepts, 33 inherited from another concept and 61 had component concepts. There were, on average, 38.0 concepts in each case study where 18.5 (mean) were unique. Some of these concepts were later added to the vocabulary.

Some concepts were heavily reused across projects. Removing the 10 case study concepts and 7 false-start concepts (those not used in any final implementation), 63 concepts remained. Of these, about 45% were used in more than one project (28 of 63). Additionally, about 24% of the concepts were used in 4 or more case studies (15 of 63) with 5 used in all (7.9%).

Comparing the number of concepts across the implementations suggests that developers were able to compose functionality out of many different concepts. This includes the use of reuse, extension and inheritance.

6.4.2. Metrics

Three metrics are computed from the above measures for a concept. These metrics are the state space (S_{sp}), action space (A_{sp}) and interaction space (I_{sp}). The state space is the total number of states that the system can exist in. The action space is the total number of distinct actions that can occur in the system. The interaction space is the number of state/action combinations that are possible. The term space is used in these metrics because it conveys the notion of exploration during development. As such, a larger space means more exploration needs to be performed to exhaust all possible states, actions or interactions, which implies a more complex implementation. Each space is discussed in more detail below in regard to the following three model types.

Three different models for these spaces are used in the following discussion: naïve (N), hierarchical (H) and Chasm (C). The naïve model is without any reductions applied to the representable space. The use of a single state machine to represent an entire application would be an example of a naïve model. The hierarchical model refers to a reduction in space due to hierarchy. This is the type of space reduction typical of Statecharts [Har92]. The final model is Chasm's model (see Chapter 5). Each model can be used for the spaces above. Because of this, the model type will be used as a prefix to the space to define the model/space pairing. So, NS_{sp} is a naïve state space and CI_{sp} is a Chasm interaction space.

Though the case studies were implemented in a Chasm model, the other models can be computed from Chasm's representation. This allows comparisons between the different modeling types in the same case study. As shown in Table 6-4, the Chasm models are equal to or smaller than the naïve or hierarchical models for the case study spaces.

Table 6-4: The three spaces in the three model types per each case study show the decreasing implementation complexity. Both hierarchical and Chasm models reduce the state space but Chasm's reduction of the action space is a large improvement over even the hierarchical model's interaction space.

	HS_{sp}& NS_{sp}	NA_{sp}& CS_{sp}	HA_{sp}	CA_{sp}*	NI_{sp}	HI_{sp}	CI_{sp}
VDA1	3.29e+22	201	200	2.30	6.57e+24	40200	734
VDA2	2.21e+15	130	131	2.15	2.90e+17	17030	399
VDA3	1.73e+06	51	50	2.00	8.64e+07	2550	144
VDB1	1.15e+08	66	67	2.39	7.72e+09	4422	219
VDB2	7.68e+06	58	60	2.31	4.61e+08	3480	189
VDB3	5.25e+23	201	208	2.54	1.09e+26	41808	772
VDC1	2.25e+03	29	28	2.33	6.30e+04	812	100
VDC2	2.25e+04	37	36	2.40	8.10e+05	1332	130
VDC3	5.62e+04	43	42	2.33	2.36e+06	1806	154
VDC4	4.05e+06	58	56	2.15	2.27e+08	3248	191

** CA_{sp} was zero because these are the top concepts in the system. Because of this, the average number of actions per concept is reported.*

6.4.2.1. State Space

A state space (S_{sp}) is a measure of the number of different states during execution that must be considered by the developer. Both Chasm and hierarchical models collapse the S_{sp} in a similar fashion, retaining the original representable state space, by reducing states that cannot exist or are irrelevant based upon the system functionality. This greatly simplifies development, as can be seen by comparing NS_{sp} to HS_{sp} and CS_{sp} (Table 6-4).

Computing the S_{sp} is different for the three models; HS_{sp} and CS_{sp} require the counting of states while NS_{sp} requires the equation (in Figure 6-6 below). HS_{sp} and CS_{sp} are the sum of all the states in the system, or *s*. Because both the Chasm and hierarchical models collapse the state space in a similar manner, the HS_{sp} is the same as the CS_{sp}. The equation in Figure 6-6 calculates the naïve state space by considering that a concept can potentially exist in each of its own states for each state of each of its component concepts [Har98]. Both CS_{sp} and HS_{sp} greatly reduce their state spaces over the NS_{sp} in this manner.

$$NS_{sp} = (NS_{sp})_i (NS_{sp})_m$$

$$(NS_{sp})_i = (NS_{sp})_c + (NS_{sp})_p$$

$$(NS_{sp})_m = (NS_{sp})_{pm} (NS_{sp})_{cm}$$

$(NS_{sp})_{cm}$ *is the product of every component's* NS_{sp}

$(NS_{sp})_{pm}$ *is the product of every parent's* $(NS_{sp})_{cm}$

$$(NS_{sp})_c = s_c$$

$$(NS_{sp})_p = s_p$$

Figure 6-6. Computing the NS_{sp} from the Chasm model has to consider that each state in a concept can exist in each state of its components' states. This leads to a much larger state space.

In the ten case studies, CS_{sp} was much smaller than NS_{sp}. In fact, the CS_{sp} is linearly related to the log of the representable NS_{sp}, as shown in Figure 6-7 (below). Thus, a linear increase in state in the Chasm model allows a concept to represent an exponential growth in state. This is a major success given the complexity issues identified in Chapter 3.

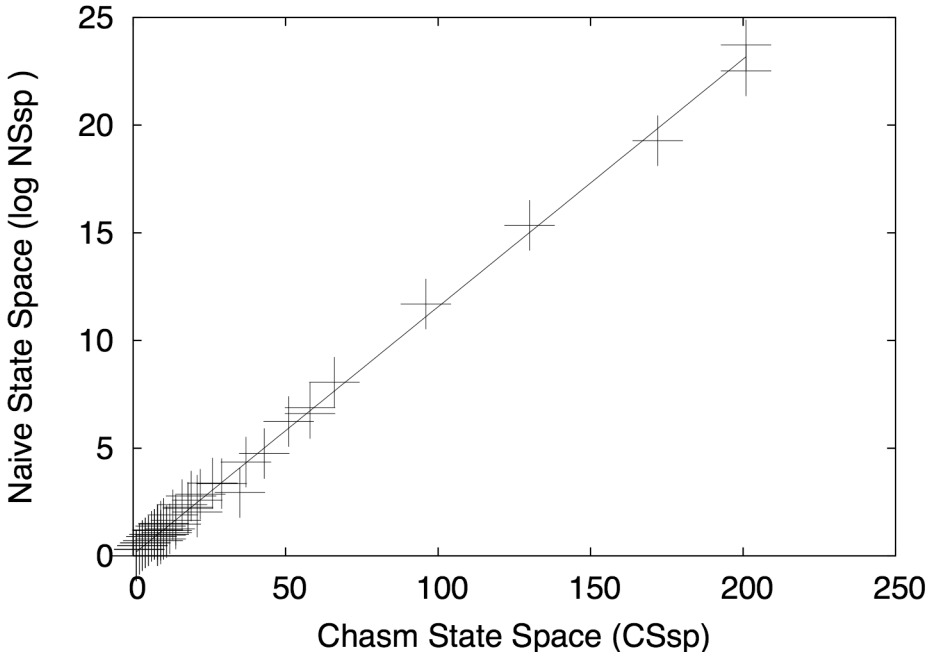

Figure 6-7. The CS_{sp} is linear related to the log of the NS_{sp} for the concepts in the case studies. As such, a concept in Chasm represents an exponential growth in complexity with only a linear increase in a Chasm concept's state space.

6.4.2.2. Action Space

An action space (A_{sp}) is the number of actions that a developer must consider during development. A Chasm action space CA_{sp}, due to the local scope of actions, is only the number of internal actions of the concept, or a_i, and *not* the total number of actions in a concept, a. In contrast, NA_{sp} and HA_{sp} have no local scope for actions so their action space is the total number of actions in the system, or a. This allows the potential for side-effect behaviors in those models when an action affects part of the

system it was not intended to impact.

In practice, hierarchical models only use a small number of actions outside of their level due to the cohesion of the hierarchy. However, this only reduces the frequency of this problem as there is still a potential for unexpected interaction. When it occurs, it is difficult to fix [von94]. This is similar to the problems of globally scoped variables in imperative languages, which have been well-documented [Wul73].

In the ten case studies, the per-concept average between the models is very different. These averages are 2.54 for CA_{sp} versus 22.0 for NA_{sp} and HA_{sp} actions.

6.4.2.3. Interaction Space

The third space is the interaction space (I_{sp}), which represents the number of action/state combinations in a concept. As such, the interaction space reflects the developer's work to design and develop a concept. Additionally, it is the total number of interactions that must be considered to exhaustively remove all unintended behaviors. Because it represents both the states in the system and the actions in the system, it is a better measure of complexity than the state and action spaces.

The computation for each model is shown in Figure 6-8 (below). The simplest case to compute is the naïve interaction space, NI_{sp}. The NI_{sp} for a concept is the product of the concept's naïve state and action spaces ($NS_{sp}NA_{sp}$), as every state has to be considered regarding every action.

$$NI_{sp} = NS_{sp}NA_{sp}$$

$$HI_{sp} = HS_{sp}HA_{sp} = HS_{sp}NA_{sp}$$

$$CI_{sp} = s_i a_i + \sum^{m} CI_{sp}$$

Figure 6-8. : The reduction of the interaction space of a representation reduces the developer's design and development work. A Chasm interaction space is smaller than a hierarchical interaction space, which is smaller than the naïve interaction space.

For hierarchical models, the interaction space is reduced due to the reduction in the state space (see Figure 6-8). In this case, the interaction space is the product of the hierarchical state and action spaces ($HS_{sp}HA_{sp}$). Because HS_{sp} is much smaller than NS_{sp}, the HI_{sp} is much smaller than NI_{sp}.

In Chasm, the interaction space for a system is further reduced because actions have local scope (see Figure 6-8). Because of this, the CI_{sp} is the product of its internal states and actions as well as the sum of its component's CI_{sp}'s. Because Chasm reduces the state space as hierarchical models do and in addition reduces the action space, the CI_{sp} is much smaller than both the naïve and hierarchical models. For example, compare the growth of CI_{sp} to HI_{sp} (Figure 6-9).

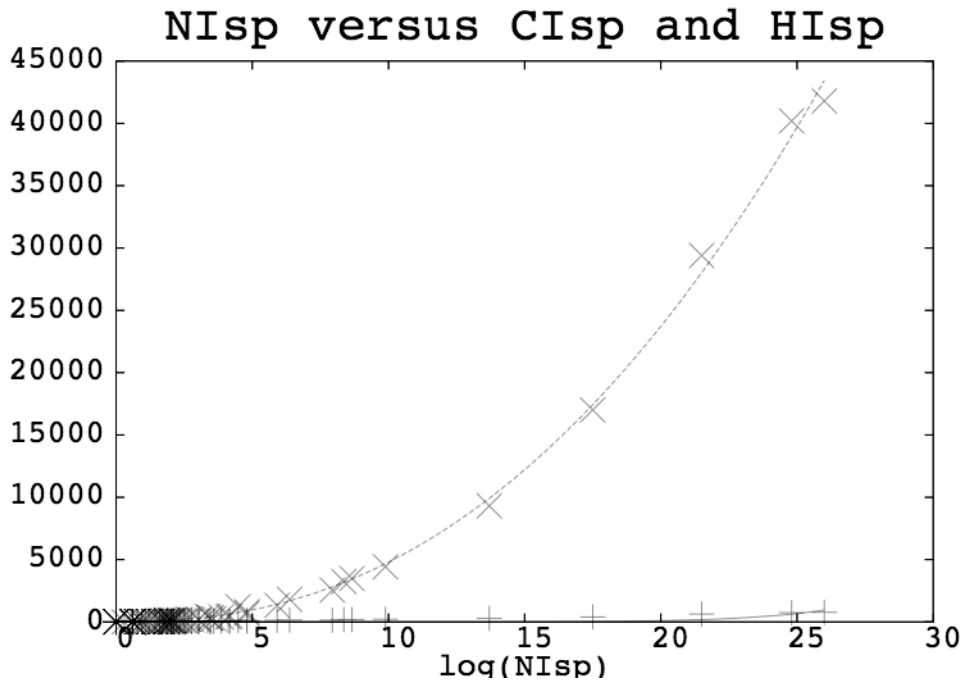

Figure 6-9. The growth of CI_{sp} is much smaller than HI_{sp} when compared to NI_{sp}.

6.4.2.4. Analysis of Spaces

The scale and growth of the three spaces in the three models show how implementations grow in complexity (see Table 6-4). In the ten case studies, CI_{sp} is smaller than HI_{sp}, which is smaller than NI_{sp}. The ratio of these spaces is shown in Table 6-5. Taking these three spaces as metrics of implementation complexity suggests; 1) the reason developers are constantly encountering special cases and 2) the importance of locally scoped actions in addition to hierarchy to collapse the state space.

Developers are constantly encountering special cases because the interaction spaces are extremely large, too large, in fact, to consider every possible condition even with a large number of developers. For example, a medium-sized case study has a state space of around 10^8 interactions, using a naïve model. If each interaction in the system is considered in 1 second[9], then checking every interaction requires 27777.7 man-hours. With even a hundred developers, roughly seven 40-hour weeks of effort would be required. For the largest case study of 10^{26} interactions, 100 developers would require $2.78*10^{23}$ hours or $3.17*10^{19}$ man-years working every hour of every day of the year. This is not a tractable approach.

Hierarchy is effective in reducing the interaction space. For the same 10^8 interactions, hierarchy roughly reduces the interactions to 4000. At one second per interaction, this becomes a little over an hour of effort. However, for the largest case study with 10^{26} interactions, hierarchy reduces the interactions to 42,000. This would require 11.67 hours. While this is possible, few developers would subject themselves to such a tedious task!

With Chasm, locally scoped actions reduce the interaction space yet again. For the moderately sized case study, VDC2, 229 interactions have to be considered, or about

9 One second is chosen for simplicity of the calculation. During development, to consider such an interaction could take considerably longer.

four minutes of effort (using the fanciful assumption of 1 second per consideration). For the largest case study, VDA2, 772 interactions have to be considered, or 13 minutes of effort. With a Chasm representation, the problem of considering every case in development becomes tractable even for the largest case study.

In addition, because concepts are being reused, the total number of interaction considerations is smaller still per project. In the reuse section above, it was found that about 45% of the concepts in the case studies were used in more than one case study. Making the assumption that the reused concept has already had its interactions considered by the concept's original author further limits the number of interactions.

One additional benefit of a Chasm model is that the interaction space is not only smaller but more dense. This is achieved because the collapsing of the interaction space removes cases where no interaction occurs. Exhaustively considering sparse interaction spaces is prone to human error, especially in a repetitive task, even with high levels of vigilance. With a Chasm model, the remaining interactions to consider are much more likely to be important, making the task smaller and less repetitive.

Table 6-5: The ratio of spaces shows the reduction of the interaction space due to a hierarchical model, a Chasm model and the benefit of Chasm over even a hierarchical model.

	HI_{sp}/NI_{sp}	CI_{sp}/NI_{sp}	CI_{sp}/HI_{sp}
VDA1	6.12e-21	1.12e-22	0.02
VDA2	5.88e-14	1.38e-15	0.02
VDA3	2.95e-05	1.67e-06	0.06
VDB1	5.73e-07	2.84e-08	0.05
VDB2	7.55e-06	4.10e-07	0.05
VDB3	3.83e-22	7.07e-24	0.02
VDC1	1.29e-02	1.59e-03	0.12
VDC2	1.64e-03	1.60e-04	0.10
VDC3	7.64e-04	6.52e-05	0.09
VDC4	1.43e-05	8.42e-07	0.06

6.5. Evaluation Results

The results of these evaluations suggest that Concept-Oriented Design (COD) in Chasm was successful in addressing multiple 3D interface development problems. In this final section, the evaluation results are mapped back to RQ1's five problems (see Chapter 3).

6.5.1. Limited Understandings

> *P1 - Developers can describe their 3D interaction at a high level and the actions to perform at a low level but are limited in describing the behavior between them.*

In the case studies, developers using Chasm overcame the limited actual understanding problem. There was positive feedback about Chasm conceptually. One domain expert stated that Chasm was "good because we are starting at concepts without committing to any kind of implementation or details." Another stated, "You can put everything out there and figure out how you want to string it together." Though volunteer developers were satisfied with Chasm, they suggested that there was a learning curve. Recall, for example, that one volunteer stated that "nothing

was that difficult to describe once you get into the frame of mind of thinking in terms of states and state changes and messages between states."

Two instances in the case studies also address this problem. In the VDA1 case study, describing the application as a flow led to the realization that the flow was actually two separate flows that drove each other. So, the envisioned behavior was divided into two simpler concepts with causal statements between them. In a second instance, a long-term developer realized a requirement late in implementation. A new state was added in an existing concept with little impact to the rest of the system. Recall that they commented, "So, I just added it. It was easy to add ... Everything that had previously been defined worked and everything that was newly defined worked ... And that was way later in design."

Identified themes map to and support the limited understanding problem. The designability sub-theme of *choosing precision* highlighted the volunteer developer's ability to work through their problems from multiple approaches. In addition, the structuring thought sub-theme of *importance of a name* highlights how the volunteer developers could name different aspects of a Chasm system to give understanding while they are developing. In the domain expert theme *notation*, experts liked how they could make notes to themselves. In tandem, these show how Chasm approached this problem -- by allowing development to proceed from many different angles, simultaneously allowing developers to give descriptive names and create notes to themselves. The volunteer developer's structuring thought sub-theme of *sequence versus causality* brought up a potential problem of not enough focus on the flow of behavior through the system and too much focus on the causality in a concept. The domain experts in the *visual representation* theme had concerns about the chosen representations for the visual layout. Potentially, this visual representation could cause problems of limited understanding in its own right, especially in larger systems.

6.5.2. Distance of Mapping

> *P2 - The structural differences between the implementation representation and developer's understanding create problems for developers.*

Chasm's approach to the distance of mapping problem was validated by domain experts and long-term developer comments, as well as in case studies. One domain expert commented, "You think sequentially. You think, 'When I do this, the system does that.' … So, from a programming perspective, I think that you often have to think [Chasm's] way." One volunteer developer stated, "It's really easy for me to think about this."

The case studies also supported Chasm's approach to this problem. In one instance, a volunteer developer's experience of adding a button to drive an interaction showed the naturalness of working with Chasm. They added the concept for the button and a causal statement from the button to their system, and were amazed at how it just worked and did not interfere with any other parts of their system. In another case study, the author wanted to create slight variations of the basic Raycasting technique to improve selection. With Chasm, there was no modification needed to the original Raycasting concept, only causal statements attached to Raycasting's states [Win05b]. As a testament to Chasm's ability to keep implementations separate, all the variations of Raycasting could run simultaneously without interfering with each other.

The identified themes map to and support the distance of mapping problem in three different ways. The first way is Chasm's similarity to developer thinking. The experts

agreed that Chasm matches how they think, as evidenced in the *match to thinking and development* section above. The volunteer developers were leery about the match, as discussed in the *structuring thought* sub-theme of *not normal thinking* above. In this section, they state that it was different at first (see the *learnability* section under *Chasm thinking)* but that they liked it with increased exposure to Chasm. Given that Chasm was based on the thinking of domain expert's and these experts liked Chasm's method of thinking, it is likely that this is a reciprocal relationship. The volunteer developers did like how Chasm allowed them to sketch out ideas (see the *designability* sub-them *sketching tool)*.

The second way the themes map is in regards to the visualization, layout and interface of Chasm. The domain experts had many ideas put together in the *visual representation* theme about the visualization of concepts, components and causal statements, as well as the package view. They did have concerns about the size of the visualization, especially for larger systems. The volunteer developers liked the visualization of the representation but realized it could be improved upon.

The third way the themes map is in regards to learning Chasm. The volunteer developer theme of *learnability* listed several issues with learning Chasm but overall said it just takes time and they were successful once Chasm was learned. The domain experts also stated that Chasm experience was required.

6.5.3. Complexity

> *P3 - Added functionality to a 3D interface leads to a non-linear increase in implementation complexity and affects the functioning of the entire system.*

Chasm's approach to the Complexity problem was validated by domain experts, volunteer developers and measures and metrics. It was encouraging that domain experts could see how Chasm overcomes complexity even with their limited exposure. One domain expert stated, "You capture [the implementation] while you think of it." Volunteer developers were satisfied with the ways in which Chasm helped them think about developmental processes. Their comments were: "at a really high level, it forces you to be modular, which is a lot more maintainable, implementable, especially because Chasm takes care of a lot of the low-level event handling."

An analysis of the case study implementations can also show how Chasm overcomes the Complexity problem. A comparison of Chasm's model to that of the naïve and hierarchical models show it has a state space improvement over the naïve model comparable to the hierarchical model, but also is an improvement in the action space over both the naïve and hierarchical models. This greatly improves its interaction space model, which was argued here to be a better measure of development complexity.

Identified themes map to and support the complexity problem. The volunteer developer's theme of *structuring thought* reduces the scope of development in Chasm by creating modular, smaller and more manageable, units of development. This was also echoed in the domain expert's theme of *match to thinking and development*. However, the experts were concerned that Chasm's structuring of thought may not scale; this included debugging systems, abstraction and reuse, and the visualization of Chasm's structure. This was dispelled by the volunteer developer's experience, captured in the representation sub-them of *behind the scenes*, where the expert's concerns never materialized.

The analysis of the measures and metrics of the case studies provides additional

support for the idea that Chasm addresses the complexity problem. By comparing the measures and metrics of the case studies, it was shown that a linear growth in the Chasm representation was able to represent an exponential growth in the state and interaction spaces. This was not just shown for the author's case studies, but also those of the volunteer developer; they were able to create similarly complex systems and interactions. Additionally, there was a large amount of composition and inheritance, meaning that developers were successful in reusing concepts. This helped to address complexity by black-boxing it in component concepts.

6.5.4. Reimplementation Over Reuse

> *P4 - Developers are unable to modify and reuse existing implementations for new purposes.*

Chasm's approach to the reimplementation over reuse problem was validated three times in the case studies and measures of case study reuse. VDA1 required that users in the system select buttons to advance the experiment. Raycasting was the subject of the study, but it also fulfilled the requirement to select buttons. So, another Raycasting concept was simply added to the system. In VDA2, three flavors of Raycasting were created by extending, without modifying, the original Raycasting concept. The third instance was the reuse of a concept in the vocabulary by VDB. When the volunteer developer updated their version of Chasm, the issue they were having was resolved, and the developer described this as "magic." Case study results also suggest that developers are able to reuse concepts through inheritance, extension and reuse as component concepts with close to 45% of concepts being used in more than one case study.

Identified themes map to and support the reimplementation over reuse problem. The domain expert theme *abstracting and reusing content* supported this idea. The volunteer developers also had supportive themes. The *structuring thought* sub-theme of *composition and decomposition* represents how Chasm concepts were later reused in new systems. This was also supported by the *designability* theme that highlighted the ability to quickly make changes and sketch out ideas. The learnability sub-themes of *chasm thinking* and *documentation* also helped in that the volunteer developers could learn about concepts for easy reuse through documentation and the tiered architecture.

6.5.5. Hard Domain Problems

> *P5 - Several disparate types of knowledge and skill are required to implement 3D interfaces.*

There are unaddressed hard domain problems in Chasm. Instead of addressing them, Chasm instead relies on the COD principle of Community. The hard domain problem includes issues in computer graphics, evaluation, device I/O and developers required to have a broad set of knowledge and skills to build 3D interfaces. This issue is discussed in detail in the *confounding factors* in learnability.

However, several concepts in the vocabulary have started to address hard domain problems in development, including the animation and resource concepts. With the community principle of COD, Chasm benefits when community members successfully address a hard domain problem. This includes improved reuse of successful concepts between systems, the incorporation of toolkits in the code tier, and using builders to make concepts toolkit-agnostic. Each of these is a step in creating a means to share successes between developers.

Thus, Chasm was not designed to specifically solve hard domain problems, but does make the dissemination of the successes more fluid.

* * *

In summary, Chasm's approach to 3DUI design has had its approach evaluated by criteria, domain experts, long-term volunteer developers and by measures and metrics in case studies. From these, it was concluded that Chasm may successfully address four of the five problems earlier identified and discussed in this work (see Chapter 3). Chasm may also have the ability to address the fifth problem once a community is formed (see Principles of COD in chapter 4).

Chapter 7 - Conclusions and Future Work

This final chapter summarizes this study's research questions and hypotheses, major contributions and implications for future work.

7.1. Summary of Research Questions

The following four research questions are summarized in how they guided and framed this research.

RQ1 - Why are current methods and tools insufficient for the design and development of complex 3DUIs?

The investigation studied developers to identify the themes common across their 3DUI design and development. Included in this investigation were developer artifacts, domain language and code. This was collected by interviews, an experiment to collect developer descriptions of 3DUIs and the solicitation of artifacts and code from the 3DUI community. This resulted in eight themes (see Table 7-1) and five problems (see Table 7-2) that emerged and were used to inform the design and evaluation of Concept-Oriented Design in Chasm.

RQ2 - What are the core principles of an effective representation for the design and development of 3DUIs?

RQ1's themes and problems led to an innovative methodology for software design and development, Concept-Oriented Design (COD). The five principles (shown in Table 7-3) formed the basis of COD along with the development and reuse of one cohesive idea in a concept.

RQ3 - How can Concept-Oriented Design be instantiated as a tool to improve the design and development of 3DUIs for developers?

Chasm was created as an instantiation of the principles of Concept-Oriented Design. Chasm's design focuses on a four-tiered representation. The tiers progress through: the envisioned behavior tier with conversational domain language, the causality tier with causal statements connecting concepts, the automata tier as a state machine and the code tier in C++ code. As a system, Chasm enabled developers to work between the tiers to visualize and decompose their envisioned behavior of a concept. To this end, a graphical interface was created, the ChasmGUI, along with several command-line tools. Additionally, an emphasis was placed on making the runtime behavior of the system apparent through visualization and event tracing. Several issues were overcome to enable this to work smoothly including event ordering, according to the action-processing dictum, managing cycles and the handling of special cases.

RQ4 - How effective is Chasm in the development of 3D interfaces?

Finally, this study was concerned with Chasm's effectiveness in the development of 3D interfaces. To examine effectiveness, this study used criteria, domain experts, longitudinal evaluation by volunteer developers and a comparison by measures and

metrics of case studies. Themes and models were then created which were ultimately mapped to the five problems of 3DUI design and development.

Domain experts viewed Chasm favorably, especially regarding its ability to address complexity and design 3DUIs. However, they were concerned about its visual representation and its ability to scale. Long-term developers liked how Chasm structured their thinking and offered suggestions on improving Chasm's code tier. Learning to develop 3DUIs with Chasm took time but there is evidence that 3DUI issues, and not Chasm, were the most problematic. Chasm's internal functionality, such as the action-processing dictum, cycles and special cases, successfully removed concerns from the long-term developers. Both participant groups felt that Chasm was good for understanding and representing flow of a 3DUI.

Ten non-trivial 3DUI case studies were created. From them, the state, action and interaction spaces of Chasm were compared against naïve and hierarchical representations. While the state space is traditionally reported, and Chasm reduced the state space comparable to the hierarchical model, it was argued that the interaction space is a better measure for implementation complexity. The interaction space, which represents state-action interactions, is reduced by the local scope of Chasm's actions, making its representation better than even hierarchical models. Additionally, there was a large amount of reuse in Chasm's case studies with about 45% of all concepts across all case studies used in more than one case study.

7.2. Major Contributions

This study contributes to the extant literature and the development of 3DUIs by:

- Identifying themes in developer artifacts and language
- Identifying five problems in existing 3DUI development
- Creating the software methodology of COD based on five principles
- Instantiating COD in Chasm
- Planning a practical approach for system evaluation

These contributions are summarized in more detail in the following sections.

7.2.1. Themes in Developer Artifacts and Language

Several themes were identified (see Table 7-1 below) when looking at developer artifacts, language and code. It was observed that *streams of thought* were the normal way developers discussed their envisioned behavior and the use of *domain terms* allowed words to carry much more weight and meaning. They also used *meta-information* about their artifacts to better contextualize the streamed statements and supplemented this by *diagram usage*. The developers also used *composable statements* that weaved together, as *clusters and paths*, the full envisioned behavior. These statements could also be of different levels of *abstraction,* which (overall) may develop within a *specificity order*.

Table 7-1. In the investigation, themes identified useful structures in developer artifacts, language and code that should be supported.

Streaming Thoughts	Statements captured as a stream of thought allow developers to quickly capture their ideas and first thoughts.
Domain Terms	The use of terms either from developer's domain of knowledge or defined ad-hoc for the situation allow

	developers to say a lot with only a single term.
Meta-Information	Meta-information describes the situation or context of the developer statements.
Composable Statements	Statements can be composed of other statement types.
Abstraction	Abstraction was used to create statements without going into the details of how to achieve the statements.
Specificity Order	The statements show a specific order of detail that allows developers to be abstract and specific as allowed by their understanding and whim.
Clustering and Paths	Statements cluster into temporary states in which they are active with paths between them formed by relationship statements.
Diagram Usage	Diagrams were commonly used to explain a part of an interaction or the scene in which the interaction takes place.

Several theories of development correspond to the above themes. For example, scenario-based design [Ros02] and sequence charts in UML (http://www.uml.org) have attempted to capture the streams of thought in development thinking. Both focus on the elicitation of requirements and not a representation used as development, though some attempts have been made to generate code from UML [Har97][Tog]. Literate programming [Knu92] is a philosophy that code should be first and foremost, human readable. To this end it emphasizes many of these themes but is only a philosophy, not a tool. Visual programming languages [Whi97][Ken02] have tried to capture development thinking in visual representations. Some domain specific visual programming languages have also become mainstream, including Labview [Lab]. However, few systems incorporate all of these themes, putting them together in one system.

7.2.2. Five Problems in Existing 3DUI Development

The five problems in existing 3DUI development were identified in Chapter 3 (see Table 7-2 below). These problems are limited understanding, distance of mapping, complexity, reimplementation over reuse and hard domain problems. Some attempts to better match developer thinking in the 3DUI domain have been performed [Con00]. Many systems and methodologies try to address the complexity problem, such as Statecharts [Har92], Model-Driven Engineering [Ken02] and dataflow programming [Joh04]. Often, these create many problems of their own [von94] or don't scale in the case of dataflow languages [Joh04]. Additionally, many of these systems are too domain specific and developers are hesitant to take the time to learn a non-generalizable system [Mye99].

Table 7-2. 3DUI design and development problems.

Limited Understanding	Developers can describe their 3D interaction at a high level and the actions to perform at a low level but are limited in describing the behavior between them.
Distance of Mapping	The structural differences between the implementation representation and developer's understanding create problems for developers.

Complexity	Added functionality to a 3D interface leads to a non-linear increase in implementation complexity and affects the functioning of the entire system.
Reimplementation over Reuse	Developers are unable to modify and reuse existing implementations for new purposes.
Hard Domain Problems	Several disparate types of knowledge and skill are required to implement 3D interfaces.

The traditional approach to improve 3DUI development involves domain specific APIs to address hard domain problems. Many APIs have been created, many are no longer supported and many more are being developed [Ste08][Ray08]. Each of these APIs claims to be a better tool than the previous and support a certain subset of features of 3D interaction. Typically, the choice of API for a project is made based on the features needed for an application, developer familiarity with the API, access to API experts and the not-invented-here syndrome [Ste08]. However, these APIs do address specific parts of the hard domain problems.

Hopefully, the recognition of these five problems will guide the development of future toolkits. Existing approaches do a poor job in assisting developers in their limited understanding of their envisioned behavior. Directly coding creates inflexible code and locks-in bad design decisions. In this sense, the rapid prototyping approaches to development, such as Extreme Programming [Bec99], make sense. There is also a large distance of mapping in existing implementations between developer thinking and the form of implementation. Most systems are event-based, the problems of which are well documented [Mye91]. While higher-level approaches to development and specification are becoming more popular research topics [Dac06][Dac07][Lat08][Sha08], the ability to support the development iteratively in the model and the code is problematic [Mye93]. Also, developers prefer general rather than domain specific tools that are difficult to learn [Mye99].

7.2.3. Concept-Oriented Design (COD)

COD, with its five principles (see Table 7-3) and a concept as its unit of development, was created to address the identified problems and support the identified themes found in developer artifacts and language.

Table 7-3. The five principles of Concept-Oriented Design

Envisioned Behavior Principle	Envisioned behavior is the "what is to occur," represented in a fluid and appropriately abstract representation, while concept and component concepts are the means of realizing the envisioned behavior.
Artifact and Language Focus Principle	Developers are able to conceive of 3D interaction despite the growth in complexity. A representation incorporating the themes identified in developer artifacts and language will be able to grow at the rate of developer understanding.
Tiered Representation Principle	No single representation is optimal for all developer needs but a tiered representation enables different information types to be represented optimally. Explicit connections between the tiers pull the system together as well as enabling a clear understanding of change impact.

Longevity Principle	The effort expended in creating a concept can be amortized over multiple implementations, and that COD systems will continue to perform as when they were first implemented.
Community Principle	The growth of a community is invaluable to a representation for the support and generation of a consensus of meaning it provides.

The first principle of *envisioned behavior* represents the understanding of what is to occur in a concept at a high and abstract level and in a fluid representation. The second principle of *artifact and language focus* recognizes the ability of developers to understand and think about interaction, even when it is difficult to implement. The principle encourages incorporating artifacts and language into the development process. The third principle of *tiered representation* captures development information as it becomes available and progressively drills down to a finalized form. Additionally, connections between tiers create traceable impacts when development ideas change. The fourth principle of *longevity* recognizes the benefits of concepts that last, creating more opportunities for reuse. This justifies the costs of development, extension and maintenance. The final principle of *community* acknowledges how a community makes sense of and structures development domains. Additionally, communities encourage new users, maintain the representations and share information and successes.

Included in the discussion of longevity is how modifications to a concept can clarify, optimize and correct it, but changes to the envisioned behavior result in a new concept. This assures that concepts are long-lived and systems continue to function as intended while incorporating modifications that improve the system.

7.2.4. Chasm

Chasm is a prototype implementation of COD. Its representation is in four-tiers, the first three in an XML representation, ChasmXML, and the last in C++ code. Chasm is a series of tools to assist the developer on the command-line, in build procedures, in a graphical interface, for runtime awareness and runtime visualization. There is online documentation (previously a functioning wiki), a six-page manual, two tutorials applications and several smaller example applications to assist developers in the learning of Chasm. It incorporates the SVE [Kes00] toolkit for the VE functionality and has a beginning vocabulary of concepts for reuse.

The creation of Chasm had to overcome many technical problems. The internals of Chasm perform many behind-the-scenes functions to keep the complexities of executing the representation removed from the developer's concern. The use of ChasmXML to represent the internals of Chasm separates the higher tiers from the source code that developers are familiar with. The development of the action-processing dictum keeps the events ordered in a way that developers expect the system to operate. In addition, it allows existing concepts to operate consistently when new concepts are added to a system or the concepts are reused in other systems.

Three issues in Chasm arose and created side-effects when reusing concepts; cycles, gathers and destruction. The addition of new concepts to a system can create cycles. Chasm identifies these cycles, reporting them to the user at runtime. Another issue was dealt with by gathers; when a new concept is added to a system, there is the potential to add an extraneous stimulating action into an existing concept. Multiple

stimulations would break the envisioned behavior of the concept. Gathers were successful in addressing the problem by gathering all events into a single event. Because gathers are inserted automatically by Chasm, the issues were completely removed from the developer's concern. The final issue was the destruction of concepts. Since a concept can be extended for uses that were never known at development time, previous assumptions regarding the destruction of the concept can fail. By Chasm's partial management of the destruction of concepts, a concept checks if it is in use before allowing itself to be destroyed.

Chasm has had several requests for its release and plans are in the works. However, with so many ties to old APIs (SVE and the author's own C++ tools), its release will be for academic purposes and without support.

7.2.5. Practical System Evaluation

This study used a practical approach to evaluate Chasm and this approach may be applicable to other systems. Chasm's evaluation used a carefully selected series of techniques to evaluate a design and development system. This resulted in a large body of data that was then analyzed and clustered into themes.

The first evaluation is by author-selected criteria to allow for important characteristics of the system to be targeted. This targeted evaluation is at the cost of evaluation breadth. Following this, the evaluation by a set of standard criteria, agreed upon by a community, addresses the problem of breadth as well as allowing for meta-comparisons between systems that have used the same standard set of criteria. This addresses a potential selection bias when the system's author selects the evaluation criteria. For Chasm, the Cognitive-Dimensions [Bla00] questionnaire was used.

While the author of a system is the most expert and most able to perform these evaluations, they are biased and multiple evaluators need to be used. Unfortunately, expert evaluation is problematic because domain experts have only a limited exposure to the system under evaluation, there may be difficulties contacting experts or there may be relatively few experts. Additionally, expert time is expensive and limited. To overcome these problems, evaluations from both a domain expert and a system expert viewpoint were performed. Domain experts were given a short training script to introduce them to the system and then performed the evaluation. System experts were non-experts given tasks to complete over a long period of time and then performed the evaluation. In this way, the evaluation gathered domain-expert as well as experiential data. Additionally, the long-term evaluations created case studies to be used for measures and metrics. The end result was a practical evaluation on custom and standardized criteria, by experts in the domain and experts of the system, with quantifiable measures and metrics to support the themes that were drawn. With most systems lacking useful evaluations to support their success, this approach to evaluation can be effectively applied to other systems. Additionally, the use of a standard criteria set, like Cognitive Dimensions, allows for meta-analysis between systems.

This approach to evaluation creates a large body of information. Methods to handle such information can be found in Contextual Inquiries [Bey98] and Ethnographies [Spr79]. Chasm used an affinity diagramming process [Bey98] to create themes.

7.3. Implications for Future Work

This study identified several areas for future work, including: improved structuring of

envisioned behavior, better tool support, capturing of flow, improvements to the automata tier, incorporating Chasm into traditional software development and the creation of a vocabulary.

7.3.1. Structuring Envisioned Behavior

Future work on examining the ways in which the developer's envisioned behavior may be structured during development is needed. Currently, Chasm captures envisioned behavior as language but does little to assist the organization of this into the lower tiers. This may be improved by the handling of the envisioned behavior as it proceeds from abstract and interconnected ideas into a fully instantiated system. The order of specificity discovered in the language of developers is a good starting point. There is also potentially more structure in developer envisioned behavior than is currently being used. This could lead to the ability to rewrite statements into more structured forms, having impact in the causality tier as well as the automata and code tiers. For example, statements in developer language such as "Do X when Y" can be rewritten as "When Y do X" and nested conditional statements such as "When X, if X1 then Y1" hint at decision states in the automata tier. Finally, further tiers can be created. Requirements statements and observational statements both seem to hold important information. Additionally, information regarding the rationale for a concept and its design is not supported in Chasm.

7.3.2. Tool Support

There are several opportunities to improve Chasm's tool support.

First, improved visualizations need to be created to address the shortcomings identified in the evaluations. Second, incorporating the code tier into the ChasmGUI may be useful. Originally, this was avoided because Chasm needs to work with existing tools and developer workflow. However, comments from the long-term evaluators and domain experts suggest that this causes confusion. This can be achieved by exporting code from Chasm and then using a 3rd party parser like Harmonia [Bos01] to then import the code. Third, a better way to elicit the envisioned behavior from developers needs to be created. This would discourage developers from writing code in the envisioned behavior. Possible ways to do this are through better instructions and design or by simulating a conversation with the user, possibly by the use of animated agents [Bal99]. Finally, system simulation may be used for testing purposes. This would allow developers the ability to walk-through their system.

7.3.3. Flow Capture

Chasm's intent was to match developer's thinking. One problem, discovered in the evaluation, was that the automata tier of the ChasmGUI does not support the flow of a single continuous thought. Instead, the ChasmGUI forces developers to consider a flow from a state to all possible states -- a minor albeit important difference. The creation and understanding of this flow has been an important idea to scenario based design [Ros92] and needs to be better supported since flows are how developers first think about the system. There has been work on the structuring of scenarios into an automata and would be interesting to reproduce this in Chasm [Gli95].

Additionally, backstory information to the scenario needs to be maintained. Future work involves supporting the developer when new flows break existing flows. There is also the possibility of new statement types or the rewriting of causal statements to match thinking in the flows. For example, a causal statement "When X then Y" could

be represented in a flow as "Because of X, continue to Y" or "Continue to Y because of X". In this way, the structures of the flow match that of the causality tier and form a bridge between it and the envisioned behavior.

7.3.4. Automata Improvements

There are several available improvements to the automata tier's representation.

First, the current stateful automata is overkill for many of the concepts which exist in the vocabulary. In many cases, a stateless automata, where the original state of the automata does not play a role in the transition, could greatly reduce the complexity of the concept. There were comments from both the domain experts and volunteer developers on this point. This effectively made simple concepts appear complex.

Second, exhaustively considering all transitions that can occur can reduce unexpected behaviors. However, Chasm does not support any "checking off" of transitions to support developers performing exhaustive consideration. Additionally, Chasm does not allow the developers to make notes about why transitions can or cannot occur.

Third, as mentioned previously, conditional statements in the envisioned behavior can hint at the creation of decision states.

Fourth, modifying the automata tier to differentiate between external and internal actions can reduce the number of transitions in a concept. This smaller number of transitions equates to less work for developers.

Fifth, many times the state of a concept switches on and off the causal statements between concepts or even the creation and destruction of concepts. Because this was not handled in the automata tier, it had to be handled in the code tier, which means it was not visualized in the ChasmGUI. Modifications can be made to the automata tier to handle this.

Finally, the concept that is connected to a causal statement changes based upon the computation performed in the state entry functions. For example, this occurs in a selection technique where the currently selected object changes according to a selection function. This has to be performed in the code tier and without any visualization and tool support in the ChasmGUI. An addition to the Chasm syntax called a concept reference would improve the expressiveness of Chasm. A concept reference operates like a pointer in a programming language and allows the referenced concept to change during the lifetime of a concept.

7.3.5. Chasm in Software Development

There are several additions to Chasm that could improve its use in software development practice. For example, planning is a major part of development. The ability to select critical paths of development and select completion dates would be useful. Additionally, the information in Chasm needs to export to documents such as use cases, a project glossary and test cases. Also useful would be the ability to reference external documents and images, especially in the envisioned behavior.

7.3.6. Expanded Vocabulary

The current vocabulary is a small group of reusable concepts. Despite this, they were highly reused in the case study applications (about 45% were used in more than one case study). Many new concepts have yet to be created and these will likely add to the reuse. Other improvements to the vocabulary are a better structure and

organization. This may include a distribution system, versioning format, organizing scheme and documentation standards. A unit testing facility for each concept needs to be created as well so that the invariance of existing functionality is ensured. Finally, a means to organize the builders is also needed. These approaches have been used in existing systems, including RubyGems for Ruby, CPAN for Perl, JSAN for Javascript and PEAR for PHP.

References

[Abe88] Abelson, H. & G. J. Sussman, (1988). LISP: a language for stratified design. BYTE, 13(2).

[Abo00] Abowd, G. & E. Mynatt, (2000). Charting Past, Present, and Future Research in Ubiquitous Computing. ACM Transactions on Computer-Human Interaction, 7(1).

[Adl05] Adelstein, B.D., E. M. Burns, S. R. Ellis & M. I. Hill, (2005). Latency Discrimination Mechanisms in Virtual Environments: Velocity and Displacement Error Factors. Proceedings, 49th Annual Meeting of the Human Factors and Ergonomics Society.

[All04] Allard, J., V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin & S. Robert, (2004). FlowVR: a Middleware for Large Scale Virtual Reality Applications. Euro-Par.

[Alu94] Alur, R. & D. L. Dill (1994). A Theory of Timed Automata. Theoretical Computer Science, 126(2), 183-236.

[Ast04] Astesiano, E. & G. Reggio, (2004). An Attempt at Analysing the Consistency Problems in the UML from a Classical Algebraic Viewpoint. in Recent Trends in Algebraic Development Techniques, Lecture Notes in Computer Science, Springer.

[Azu97] Azuma, R., (1997). A Survey of Augmented Reality. Presence: Teleoperators and Virtual Environments, 6(4).

[Bal99] Ball, G., D. Ling, D. Kurlander, J. Miller, D. Pugh, T. Skelly, A. Stankosky, D. Thiel, M. Van Dantzich & T. Wax, (1997). Lifelike Computer Characters: The Persona Project at Microsoft Research. In J. Bradshaw (Ed.), Software Agents (pp. 191-222), AAAI Press/The MIT Press.

[Bar95] Barbacci, M., M. Klein, T. Longstaff & C. Weinstock, (1995). Quality Attributes. Technical Report CMU/SEI-95-TR-021, Carnegie Mellon University.

[Bau01] Baur, M., B. Bruegge, G. Klinker, et. al., (2001). Design of a Component-Based Augmented Reality Framework. ISAR.

[Bec99] Beck, K., (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley Professional.

[Bed94] Bederson, B. & J. Holland, (1994). Pad++: a zooming graphical interface for exploring alternate interface physics. Symposium on User Interface Software and Technology.

[Bel05] Bell, A., (2005). UML fever: diagnosis and recovery. ACM Queue, 3(2), pp. 48-56.

[Bey98] Beyer, H. & K. Holtzblatt, (1998). Contextual Design: Defining Customer-Centered Systems. San Francisco, CA: Morgan Kaufmann.

[Bie01] Bierbaum, A., C. Just, P. Hartling, K. Meinert, A. Baker, & C. Cruz-Neira, (2001). VR Juggler: A Virtual Platform for Virtual Reality Application Development. IEEE VR.

[Bla00] Blackwell, A. & T. Green, (2000). A Cognitive Dimensions Questionnaire Optimized for Users. 12th Workshop of the Psychology of Programming Interest Group, pp. 137-154.

[Bla89] Blatzley, B. R. Kennedy, K. Berbaum, M. Lilienthal & D. Gower, (1989). The time course of post flight simulator sickness symptoms. Aviation, Space and Environmental Medicine, 60(11), pp. 1043-1048.

[Blo08] Blom, K. & S. Beckhaus. (2008). On the Creation of Dynamic, Interactive Virtual Environments. SEARIS Workshop, In IEEE Virtual Reality.

[Boe07] Boeck, J. D., D. Vanacken, C. Raymaekers and K. Coninx, (2007). High-Level Modeling of Multimodal Interaction Techniques Using NiMMiT. J. of Virtual Reality and Broadcasting, 4(2).

[Boe76] Boem, B. W., J. R. Brown & M. Lipow, (1976). Quantitative Evaluation of Software Quality." Proceedings of the 2nd international conference on Software engineering.

[Boe76a] Boehm, B. W., (1976). Software Engineering. IEEE Trans Computers, Dec., pp. 1226-1241.

[Boe79] Boehm, B. W. (1979) Software Engineering – As It Is. Proceedings of the 4th international conference on Software engineering.

[Bos01] Boshernitsan, M., (2001). Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools. Master's Thesis. Technical Report CSD-01-1149, University of California, Berkeley.

[Bow95] Bowen, J. & M. Hinchey, (1995). Ten Commandments of Formal Methods. IEEE Computer, 28(4).

[Bow99] Bowman, D. (1999) Interaction Techniques for Common Tasks in Immersive Virtual Environments: Design, Evaluation, and Application. Georgia Tech Dissertation.

[Bow02] Bowman, D., J. Gabbard & D. Hix, (2002). A Survey of Usability Evaluation in Virtual Environments: Classification and Comparison of Methods. Presence: Teleoperators and Virtual Environments, 11(4).

[Bow97] Bowman, D. & L. Hodges, (1997) An Evaluation of Techniques for Grabbing and Manipulating Remote Objects in Immersive Virtual Environments. In Proceedings of the 1997 Symposium on Interactive 3D Graphics.

[Bow04] Bowman, D., E. Kruijff, J. LaViola & I. Poupyrev, (2004). 3D User Interfaces: Theory and Practice. Addison-Wesley Professional.

[Bow98] Bowman, D., J. Wineman, L. Hodges & D. Allison, D., (1998). Designing Animal Habitats Within an Immersive VE, IEEE Computer Graphics & Applications, 18(5), pp. 9-13.

[Bow01] Bowman, D. & C. Wingrave. (2001) Design and Evaluation of Menu Systems for Immersive Virtual Environments. Proceedings of IEEE Virtual Reality, pp. 149-156.

[Bri01] Briand, L., C. Bunse & J. Daly, (2001). A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented

Design. Software Engineering.

[Bro95] Brooks, F., (1995). The Mythical Man-Month. Addison-Wesley, 2nd ed.

[Bro87] Brooks, F., (1987). No Silver Bullet: Essence and Accidents of Software Engineering, Computer Magazine, April.

[Bro99] Brooks, F., (1999). What's Real About Virtual Reality. IEEE Computer Graphics and Applications, Nov/Dec.

[Bur06] Burns, E., S. Razzaque, A. T. Panter, M. Whitton, M.R. McCallus & F. P. Brooks, (2006). The Hand is Slower than the Eye: A quantitative exploration of visual dominance over proprioception. Journal on Presence: Teleoperators and Virtual Environemnts, 15(1), pp. 1-15.

[Cal03] Calder, M., M. Kolberg, E. H. Magill & S. Reiff-Marganiec, (2003). Feature Interaction: A Critical Review and Considered Forecast. Computer Networks, 41(1), pp. 115-141.

[Car00] Carroll, J., (2000). Making use: scenario-based design of human–computer interactions. Cambridge, Mass: MIT Press.

[Coc01] Cockburn, A., (2001). Writing Effective Use Cases. Addison-Wesley Longman.

[Con00] Conway, M., J. Pierce, R. Pausch, et. al. (2000). Alice: Lessons Learned from Building a 3D System For Novices. ACM CHI, pp. 486-493.

[Dac01] Dachselt, R., (2001). CONTIGRA - Towards a Document-based Approach to 3D Components. Proceedings of the Workshop 'Structured Design of Virtual Environments and 3D-Components' (WEB 3D '01), Paderborn, Germany.

[Dac06] Dachselt, R., P. Figueroa & I. Lindt, (2006). Workshop on the Specification of Mixed Reality User Interfaces: Approaches, Languages, Standardization. IEEE Virtual Reality.

[Dac07] Dachselt, R., P. Figueroa, I. Lindt & W. Broll., (2007). Workshop on the Mixed Reality User Interfaces: Specification, Authoring, Adaptation. IEEE Virtual Reality.

[Dal96] Daly, J., A. Brooks, J. Miller, M. Roper & M. Wood, (1996). Evaluating inheritance depth on the maintainability of object-oriented software. Empirical Software Engineering.

[Dij74] Dijkstra, E., (1982). On the role of scientific thought. In Dijkstra, E. W., Selected writings on Computing: A Personal Perspective, New York, NY, USA: Springer-Verlag New York, Inc., pp. 60–66.

[Eri84] Ericsson, K. & H. Simon, (1984). Protocol Analysis: Verbal Reports as Data. MIT Press, Cambridge, MA.

[Fig02] Figueroa, P., M. Green & H. J. Hoover, (2002). InTml: A Description Language for VR Applications. Web3D.

[Fow99] Fowler, M., (1999). Refactoring. Improving the Design of Existing Code. Addison-Wesley.

[Fra06] France, R., S. Ghosh, T. Dinh-Trong & A. Solberg, (2006). Model-Driven Development Using UML 2.0: Promises and Pitfalls. Computer,

39(2), pp. 59-66.

[Fré04] Frécon, E., (2004). DIVE on the Internet, PhD Thesis, IT University of Göteborg.

[Gam94] Gamma, E., R. Helm, R. Johnson & J. Vlissides, (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

[Gar93] Garland, D. & C. Scott, (1993). Adding implicit invocation to traditional programming languages. Software Engineering.

[Gla67] Glaser, B. & Strauss, A. (1967) The discovery of grounded theory: Strategies for qualitative research. London: Weidenfeld and Nicolson.

[Gla95] Glass, R. (1995) A Structure-Based Critique of Contemporary Computing Research. Journal of System Software, pp. 283-7.

[Gli95] Glinz, M. (1995) An Integrated Formal Model of Scenarios Based on Statecharts. Software Engineering – ESEC, Springer Berlin.

[Gos04] Goslin, M. & M. Mine, (2004) The Panda3D Graphics Engine. Computer, 37(10), pp. 112-114.

[Gre91] Green, M. & R. Jacob, (1991) SIGGRAPH '90 Workshop Report: Software Architectures and Metaphors for Non-WIMP User Interfaces. Computer Graphics, 25(3).

[Gre89] Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) People and Computers V. Cambridge, UK: Cambridge University Press, pp. 443-460.

[Gre96] Green, T. R. G. & M. Petre, (1996) Usability analysis of visual programming environments: A 'Cognitive Dimensions' Framework, J. Vis. Lang. Comput. 7, pp. 131–174.

[Hai0] Hailpern, B. & P. Tarr, (2006) Model-driven development: The good, the bad, and the ugly. IBM Systems Journal, 45(3).

[Har92] Harel, D. (1992). Biting the silver bullet: toward a brighter future for system development. IEEE Computer, 25(1), pp. 8-20.

[Har97] Harel, D. & E. Gery, (1997). "Executable object modeling with statecharts," IEEE Computer, 30(7), pp. 31-42.

[Har98] Harel, D. & M. Politi, (1998). Modeling Reactive Systems with Statecharts: The Statemate Approach. McGraw-Hill Publishing.

[Hen05] Henderson-Sellers, B. et. al., (2005). UML— The Good, the Bad or the Ugly? Perspectives from a Panel of Experts, Software and System Modeling, pp. 4-13.

[Her94] Herndon, K. P., A. Van Dam & M. Gleicher, (1994) Workshop on the Challenges of 3D Interaction. SIGCHI Bulletin, 26(4), pp. 1-9.

[Hew08] Hewett, T., R. Baecker, S. Card, T. Carey, J. Gasen, M. Mantei, G. Perlman, G. Strong & W. Verplank, (2008) ACM SIGCHI Curricula for Human-Computer Interaction. http://sigchi.org/cdg/index.html, cited June, 2008.

[Hix93] Hix, D. & H. R. Hartson, (1993). Developing User Interfaces; Ensuring Usability Through Product & Process. New York, John Wiley & Sons,

Inc.

[Hix99] Hix, D., J. Edward Swan II, J. L. Gabbard, M. McGee, J. Durbin & T. King, (1999). User-Centered Design and Evaluation of a Real-Time Battlefield Visualization Virtual Environment. IEEE Virtual Reality, pp. 96-103.

[Hoc91] Hoc, J. & A. Nguyen-Xuan, (1991). Language Semantics, Mental Models and Analogy. In J. Hoc, T. Green, R. Samurcay & D. Gilmore, Psychology of Programming, London, Academic Press, pp. 139-156.

[Hor05] Horstmann, C. (2005). Object-Oriented Design and Patterns. 2nd ed., Wiley.

[Hut85] Hutchins, E., J. Hollan & D. Norman. (1985). Direct Manipulation Interfaces. Human-Computer Interaction, 1(4), pp. 311-338.

[Jac85] Jacob, R., (1985). A State Transition Diagram Language for Visual Programming. IEEE Computer, pp. 51-59.

[Jac99] Jacob, R., L. Deligiannidis & S. Morrison, (1999). A Software Model and Specification Language for Non-WIMP User Interfaces. ACM Trans. On Computer-Human Interaction, 6(1), pp. 1-46.

[Jac98] Jackson, M. & P. Zave, (1998). Distributed feature composition: A virtual architecture for telecommunications services. IEEE Transactions on Software Engineering XXIV, 10, pp. 831-847.

[Jon94] Jones, C. (1994, June). Gaps In the Object-Oriented Paradigm. Computer.

[Joh04] Johnston, W., J. Paul Hanna & R. Millar, (2004). Advances in Dataflow Programming, ACM Computing Surveys, 36(1).

[Kas82] Kasik, D., (1982). A User Interface Management System. Proceedings of Computer Graphics and Interactive Techniques, pp. 99-106.

[Kat00] Kato, H., Billinghurst, M., Poupyrev, I., Imamoto, K. & Tachibana, K. (2000). Virtual Object Manipulation on a Table-Top AR Environment. In Proceedings of the International Symposium on Augmented Reality, pp. 111-119.

[Kel77] Kelly, M. J. & A. Chapanis, (1977). Limited vocabulary natural language dialogue. International Journal of Man-Machine Studies, 9, pp. 479-501.

[Kel02] Kelso, J., L.E. Arsenault, S.G. Satterfield, R.D. Kriz, (2002). DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments. In Proceedings of IEEE Virtual Reality.

[Ken93] Kennedy, R., N. Lane, K. Berbaum & M. Lilienthal, (1993). A Simulator Sickness Questionnaire (SSQ): A New Method for Quantifying Simulator Sickness. International Journal of Aviation Psychology, 3(3), pp. 203-220.

[Ken02] Kent, S. (2002), Model-Driven Engineering, Integ. Form. Methods.

[Kes00] Kessler, D., D. Bowman & L. Hodges, (2000). The Simple Virtual Environment Library: An Extensible Framework for Building VE Applications. Presence: Teleoperators and Virtual Environments, 9(2),

pp. 187-208.

[Kic96] Kiczales, G., J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda & A. Mendhekar, (1996). Aspect-Oriented Programming. ACM Computing Surveys, 28(4es).

[Kim98] Kim, G., K. Kang, H. Kim & J. Lee, (1998). Software Engineering of Virtual Worlds. Virtual Reality Software and Technology.

[Kip97] Kiper, J., E. Howard & C. Ames, (1997). Criteria for evaluation of visual programming. J. Vis. Lang. Comp., 8(2), pp. 175–192.

[Ko03] Ko, A. J. & B. A. Myers, (2003). Development and Evaluation of a Model of Programming Errors. IEEE Symposia on Human-Centric Computing Languages and Environments, Auckland, New Zealand, October 28th-31st, pp. 7-14.

[Ko04] Ko, A. J. & B. A. Myers, (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. CHI, pp. 151-158.

[Ko05] Ko, A. J. & B. A. Myers, (2005). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. Journal of Visual Languages and Computing, 16(12), pp. 41-84.

[Knu92] Knuth, D. (1992). Literate Programming. Center for the Study of Language and Information - Lecture Notes.

[Kru96] Krueger, M. (1996). Virutual (Reality + Intelligence). In Gorayska, B. & Mey, J., Cognitive Technology: In Search of the Humane Interface. eds. North-Holland.

[Lab] LabView (version 8.5) [Computer Software] Austin, TX: National Instruments, 2008.

[Las87] Lassiter, J., (1987). Principles of Traditional Animation Applied to 3D Computer Animation. Computer Graphics, 21, pp. 35-44.

[Lat08] Latoschik, M. E., D. Reiners, R. Blach, P. A. Figueroa Forero, R. Dachselt, (2008). Workshop SEARIS - Software Engineering and Architectures for Realtime Interactive Systems. IEEE Virtual Reality.

[Mee03] Meehan, M., S. Razzaque, M. C. Whitton and F. P. Brooks, (2003). Effect of Latency on Presence in Stressful Virtual Environments, IEEE Virtual Reality.

[Men91] Mendelsohn, P., T. Green & P. Brna, (1991). Programming Languages in Education: the Search for an Easy Start. In Hoc, T. Green, R. Samurcay and D. Gilmore, J., Psychology of Programming. London, Academic Press, pp. 175-200.

[Mor70] Mori, M. (1970). Bukimi no tani. The uncanny valley. (Trans.) K. F. MacDorman & T. Minato. Energy, 7(4), pp. 33–35.

[McC60] McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine," Communications of the ACM.

[Mey97] Meyer, B., (1997). UML: The Positive Spin. American Programmer.

[Mey97a] Meyer, B., (1997). Object-Oriented Software Construction. 2nd ed. Prentice Hall PTR.

[Mil94] Milgram, P. & A. F. Kishino, (1994). Taxonomy of Mixed Reality Visual Displays. IEICE Transactions on Information and Systems, 77(12), pp. 1321-1329.

[Mil74] Miller, L. A. (1974). Programming by Non-Programmers. International Journal of Man-Machine Studies, 6(2), pp. 237-260.

[Mil81] Miller, L. A. (1981). Natural Language Programming: Styles, Strategies, and Contrasts. IBM Systems Journal, 20(2), pp. 184-215.

[Mor56] Moore, E. (1956). Gendanken-Experiments on Sequential Machines in Automata Studies. Annals of Mathematics Studies, 34, Princeton University Press, pp. 129-153.

[Mor94] Morrison, J. P. (1994). Flow-Based Programming: A New Approach to Application Development, van Nostrand Reinhold.

[Mye91] Myers, B., (1991). Separating application code from toolkits: eliminating the spaghetti of callbacks. Proceedings of the User Interface Software and Technology, pp. 211-220.

[Mye93] Myers, B., (1993). Why are Human-Computer Interfaces Difficult to Design and Implement? Technical Report CMU-CS-93-183.

[Mye99] Myers, B., S. E. Hudson & R. Pausch, (1999). Past, Present and Future of User Interface Software Tools. ACM Transactions on CHI.

[Nei94] Nielsen, J. & R. L. Mack (Eds.). (1994). Usability Inspection Methods. John Wiley & Sons, New York, NY.

[New05] The New Oxford American Dictionary. (2005). Second Edition, Oxford University Press.

[OGR] OGRE, http://www.ogre3d.org/, version 1.4.9, 2008.

[Pan] Panda3D, http://panda3d.org, version 1.5.0 as of 2008.

[Pan02] Pane, J., (2002). A Programming System for Children that is Designed for Usability. PhD Dissertation, Carnegie Mellon University.

[Pan96] Pane, J. & B. Myers, (1996). Usability issues in the design of novice programming systems. Carnegie Mellon University Technical Report, CMU-CS-96-132.

[Pan00] Pane, J., C.A. Ratanamahatana & B.A. Myers, (2000). Studying the Language and Structure in NonProgrammers ' Solutions to Programming Problems. International Journal of Human-Computer Studies.

[Par72] Parnas, D. L., (1972). On the Criteria to be Used in Decomposing Systems into Modules. CACM, 15(12).

[Par85] Parnas, D., (1985). Software Aspects of Strategic Defense Systems. Communications of the ACM, 28(12), pp. 326-335.

[Pau95] Pausch, R. F., T. Burnette, D. Brockway & M. E. Weiblen, (1995). Navigation and locomotion in virtual worlds via flight into hand-held miniatures, SIGGRAPH, pp. 399-400.

[Pie01] Pierce, J., (2001). Expanding the Interaction Lexicon for 3D Graphics. PhD, Carnegie Mellon.

[Pie97] Pierce, J., A. Forsberg, M. Conway, S. Hong & R. Zeleznik, (1997). Image Plane Interaction Techniques in 3D Immersive Environments. Symposium on Interactive 3D Graphics, pp. 39-43.

[Pie99] Pierce, J., Stearns & R. Pausch, (1999). Voodoo Dolls: Seamless Interaction at Multiple Scales in Virtual Environments. Symposium on Interactive 3D Graphics, pp. 141-145.

[Pol06] Polys, N. & A. Ray, (2006). Supporting Mixed Reality Interfaces through X3D Specification. In Workshop on Specification of Mixed Reality User Interfaces: Approaches, Languages, Standardization, IEEE Virtual Reality.

[Pou96] Poupyrev, I., M. Billinghurst, S. Weghorst & T. Ichikawa, (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 (UIST).

[Pou97] Poupyrev, I., S. Weghorst, M. Billinghurst & T. Ichikawa, (1997). A framework and testbed for studying manipulation technique for immersive VR. VRST, pp. 21-28.

[Pou98] Poupyrev, I., S. Weghorst, M. Billinghurst & T. Ichikawa, (1998). Egocentric object manipulation in virtual environments: empirical evaluation of interaction techniques. Proceedings of EUROGRAPHICS, Computer Graphics Forum, 17(3), pp. 41-52.

[Pou99] Poupyrev, I. & T. Ichikawa, (1999). Manipulating Objects in Virtual Worlds: Categorization and Empirical Evaluation of Interaction Techniques. J. of Visual Languages and Computing, 10, pp. 19-35.

[Ram83] Ramsey, H. R., M. E. Atwood & J. R. Van Doren, (1983). Flowcharts versus program design languages: an experimental comparison. Communications of the ACM, 26, pp. 445–449.

[Ras85] Rasmussen, J., (1985). The Role of Hierarchical Knowledge Representation in Decisionmaking and System Management. IEEE Transactions on Systems, Man and Cybernetics, SMC-15(2).

[Ray08] Ray, A., (2008). The Interaction Framework For Innovation: A Method to Create Reusable Three Dimensional Interaction Techniques. Virginia Tech Dissertation.

[Ray07] Ray, A. & D. Bowman, (2007) Towards a system for reusable 3D interaction techniques. VRST.

[Raz02] Razzaque, S., D. Swapp, M. Slater, M. C. Whitton & A. Steed, (2002). Redirected Walking in Place. Eurographics Workshop on VEs.

[Ros02] Rosson, M.B. & J.M. Carroll, (2002). Usability Engineering: Scenario-Based Development of Human Computer Interaction. Morgan Kaufmann.

[Sch83] Schön, D., (1983). The Reflective Practitioner. How professionals think in action, London: Temple Smith.

[Sch02] Schmalstieg, D., A. Fuhrmann, G. Hesina, Z. Szalavári, L. Miguel Encarnação, M. Gervautz, W. Purgathofer, (2002). The Studierstube Augmented Reality Project. Presence, 11(1).

[Sha08] Shaer, O., R. Jacob, M. Green, K. Luyten, (2008). User Interface Description Languages for Next Generation User Interfaces. CHI.

[Sha01] Shaw, M., (2001) The Coming-of-Age in Software Architecture Research. International Conference on Software Engineering.

[Sha96] Shaw, M. & D. Garland, (1996). Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.

[Sjø05] Sjoberg, D. I. K., J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. Liborg & A. C. Rekdal, (2005). A Survey of Controlled Experiments in Software Engineering. IEEE Transactions on Software Engineering, 31(9), pp. 733-753.

[Smi98] Smith, S. & D. Duke, (1999). Virtual environments as hybrid systems. In Eurographics UK 17th Annual Conference, pp. 113–128.

[Spr79] Spradley, J. (1979). The Ethnographic Interview. Wadsworth Group / Thompson Learning.

[Sta98] Stanney, K. M., R. Mourant and R. S. Kennedy, (1989). Human factors issues in virtual environments: A review of the literature. Presence: Teleoperators and Virtual Environments, 7(4), 327-351.

[Ste06] Steed, A. (2006). Towards a General Model for Selection in Virtual Environments. IEEE Symposium on 3D User Interfaces.

[Ste08] Steed, A. (2008). Some Useful Abstractions for Re-Usable Virtual Environment Platforms. SEARIS Workshop at IEEE VR.

[Ste04] Steed, A. & C. Parker, (2004). 3D Selection Strategies for Head Tracked and Non-Head Tracked Operation of Spatially Immersive Displays. 8th International Immersive Projection Technology Workshop.

[Sto95] Stoakley, C. & R. Pausch, (1995). Virtual Reality on a WIM: Interactive Worlds in Miniature, CHI, pp. 265-272.

[Tar99] Tarr, P., H. Ossher, W. Harrison & S.M. Sutton, Jr. (1999). N-Degrees of Separation: Multi-Dimensional Separation of Concerns. Proceedings of the International Conference on Software Engineering (ICSE).

[Tay95] Taylor, R. (1995). Requirements and Availability of Application Programmer's Interfaces for Virtual Reality Systems. UNC Tech Report.

[Tay01] Taylor II, R. M., T. C. Hudson, A. Seeger, H. Weber, J. Juliano, A. T. Helser, (2001). VRPN: A Device-Independent, Network-Transparent VR Peripheral System, Proceedings of the ACM Symposium on Virtual Reality Software & Technology.

[Tho03] Thomas, D. A. & B. M. Barry, (2003). Model-Driven Development: the Case for Domain-Oriented Programming, Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, pp. 2–7.

[Tic95] Tichy, W.F., P. Lukowicz, L. Prechelt & E. A. Heinz, (1995). Experimental Evaluation in Computer Science: A Quantitative Study. Journal of Systems and Software, 28(1), pp. 9-18.

[Tor] Torque, http://www.garagegames.com, version 1.5, 2008.

[Tog] Together (version 6.0) [Computer Software]. Raleigh, NC: TogetherSoft.

[Wag06] Wagner, F. (2006). Modeling Software with Finite State Machines: A Practical Approach, Auerbach Publications.

[Web] Web3D Consortium. (http://www.web3d.org/).

[Wei93] Weiser, M. (1993) Some Computer Science Issues in Ubiquitous Computing. Mobile Computing Communications Review, 3(3).

[Whi97] Whitley, K. N., (1997). Visual Programming Languages and the Empirical Evidence For and Against. J. of Visual Languages and Computing, 8, pp. 109-142.

[Wil01] Willans, J. S. & M. D. Harrison, (2001). A toolset supported approach for designing and testing virtual environment interaction techniques. International Journal of Human-Computer Studies, 55(2).

[Win03] Wingrave, C. (2003). The Future of VR and Related Technologies and Changes Required for it to Occur. Young Virtual Reality.

[Win05] Wingrave, C. & D. Bowman, (2005). Baseline Factors for Raycasting Selection, Proceedings of HCI International.

[Win05b] Wingrave, C. & D. Bowman, (2005). CHASM: Bridging Description and Implementation of 3D Interfaces. New Directions in 3D User Interfaces Workshop, In IEEE Virtual Reality.

[Win06] Wingrave, C., Y. Haciahmetoglu & D. Bowman, (2006). Overcoming World in Miniature Limitations by a Scaled and Scrolling WIM. 1st IEEE Symposium on 3D User Interfaces.

[Win05c] Wingrave, C., R. Tintner, B. Walker, D. Bowman & L. Hodges, (2005). Exploring Individual Differences in Raybased Selection: Strategies and Traits. IEEE Virtual Reality.

[Win06b] Wingrave, C. (2006). Understanding the Limits and Benefits of a 3D Interface Specification. Workshop on Specification of Mixed Reality User Interfaces, In IEEE Virtual Reality.

[Win08a] Wingrave, C. & D. Bowman, (2008). Tiered Developer-Centric Representations for 3D Interfaces: Concept-Oriented Design in Chasm. IEEE Virtual Reality.

[Win08b] Wingrave, C. (2008). Chasm: A Tiered Developer-Inspired 3D Interface Representation. User Interface Description Languages for Next Generation User Interfaces Workshop, In CHI.

[Wir71] Wirth, N. (1971). Program development by stepwise refinement. Communications of the ACM, 14(4).

[Wul73] Wulf, W. & M. Shaw, (1973). Global Variable Considered Harmful, ACM SIGPLAN Notices, 8(2), pp. 28-34.

[X3D] X3D, http://www.web3d.org/x3d/, 2008.

[VIR] Virtools, http://www.virtools.com/, version 4.1, 2008.

[VRM] VRML, http://www.w3.org/MarkUp/VRML/, 2008.

[von94] von der Beeck, M. A, (1994). Comparison of Statecharts Variants.

Lecture Notes in Computer Science, vol 863. Proceedings of the Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 128-148.

[Zav08] Zave, P. (2008). FAQ Sheet on Feature Interaction. http://www.research.att.com/~pamela/faq.html.

Appendix A – Design Artifact Solicitation

This email was used to solicit design artifacts on the 3DUI listserv. It can be found here: http://listserv.vt.edu/cgi-bin/wa?A2=ind0506&L=3DUI&P=R1107&I=-3

Date: Thu, 16 Jun 2005 17:35:52 -0400

Reply-To: "3DUI (3D User Interfaces) Mailing List" <[log in to unmask]>

Sender: "3DUI (3D User Interfaces) Mailing List" <[log in to unmask]>

From: Chad Wingrave <[log in to unmask]>

Subject: Design artifacts requested for reflection on 3D interface design

Content-Type: multipart/mixed;

I am currently reflecting on the process of designing 3D interfaces for my dissertation. For this, I am soliciting design artifacts from the community. These design artifacts are anything written, typed, drawn or created during the implementation or planning phases of a 3D interface. This includes meeting notes, hand sketches, informal notes, scenarios, design documents, UML charts, dataflow diagrams, pseudo code, implementation code, journals which log the implementation, journals which log the design, emails about the project or anything else which can be recovered that pertains to the designing process of a 3D interface. I have created methods of coding design artifacts from our research group but require a dataset from the community at large to feel comfortable in the coverage. If you have items which you would like to submit but would first like to discuss it with me, please contact me directly. I will be happy to go over my project and answer your questions:

-Chad Wingrave ([log in to unmask])

3D Interface Group, Department of Computer Science, Virginia Tech

http://csgrad.cs.vt.edu/~cwingrav

This project has been reviewed by the Virginia Tech IRB Review Board and has been deemed exempt. Steps will be taken to keep personal information confidential. A consent and release form should be attached for your review. Please include it with your submitted design artifacts.

human_consent.pdf [application/pdf]

Appendix B – Clip Criteria

The information below is the criteria set per clip that the participants needed to identify in the Developer Language experiment of the Investigation.

Snap-To Flavor

1. When object withing X angular units, show Snap-To ray aligned with closest object.

WIM

1. WIM attached to hand
2. WIM represents a smaller version of the world
3. Changes in WIM affect real world
4. Potentially: Changes in real world affect WIM objects
5. Green highlight when touched object (Clip 1.6)
6. no collision detection (clip 1.7)
7. objects rotate at hand position (clip 1.5)
8. objects translate at hand position (clip 1.4)
9. WIM is world aligned

Gorilla

1. Tablet is a WIM of the environment
   * touch on it to go to that location
   * move items on it
   * when over item, it highlights
2. No collision detection
3. Animated movement to a location is linearly rate based
4. Moving items
   * rocks, trees, self and walls
   * changing in tablet, changes in world and changes in world change tablet
   * highlight feedback
   * stylus moved as scaled arm extension
   * when grabbing, turn off highlighting
   * items stick to ground
5. Creating items with tablet by pressing button and positioning

SSWIM

(From WIM #1, #2, #6, #9)

1. WIM is scaled
    * unspecified: using wheel on mouse
    * unspecified: limits to max and min
2. WIM is scrolled
    * line box represents the area of wim
    * scrolling occurs faster the further out from the center
    * gold arrow when scrolling shows direction
    * scrolling at rate determined by scale
    * positioning can occur when scrolling WIM
3. Buildings come and go from scene with scale and scrolling
4. Travel when release
    * when release in a position, travel to that position
    * Grabbing of self when in center
    * travel rates ramp up and ramp down to a maximum velocity (could have been time)
    * unspecified: when release outside of WIM, leave at current position
5. WIM size shown by white wireframe
6. Showing positioning
    * Height is shown by line to ground
    * position on surface is shown by red circle
    * position of self is shown by red cube
7. fly in direction of wand
    * unspecified: push button to fly
8. Button flies and grabs self and knows when by position in WIM core our outside
9. No collision detection
10. Details not noticed between worlds
    * roads and stop-signs shown on world and not WIM
    * buildings have different textures
11. Rooftops noticed

Appendix C – Statement Type DTD

This is the DTD specification of the statements identified in Chapter 3, the Investigation. This is used on developer language and design artifacts.

From version 8.

```
<!--
                This new grammar has a lot more tags to it but captures more information
correctly. The conditions
     are handled better (breaking them from the context part and putting them into the
response) and
     this allows the context of a relationship to be spread out into multiple places. In
a well formed
     response, the context will be all SA. This also addresses the issue of empty action
tags in previous
     grammars.

     I have not looked much at the issue of semantics yet and those categories. That
hopefull fits in.

     This break most statements into context, requirement, observation and relationship
with requirement
     and observation having a response, context having states and a relationship being a
response in a
     context.
-->


<!ENTITY % imp_structures "context|relationship|requirement|observation|response">
<!ENTITY % stateactionorder "SA|AS|SAS|A" >
<!ENTITY % nd "nd|interest" > <!-- notes and interest can go anywhere -->
<!ENTITY % contexttype "causal|continuous" >
<!ENTITY % rtype "abstract|applied|factual" > <!-- requirement types -->
<!ENTITY % satype "abstract|specific" > <!-- state/action types -->

<!ELEMENT dva (section)*>

<!-- A section of text with a name to it. I generally only do one section of text per
file
     but that is not necessary. -->
<!ELEMENT section (section|(original,annotated+))*>
  <!ATTLIST section title CDATA #IMPLIED >

<!ELEMENT annotated (section|semantic|%nd;|%imp_structures;|image)+>
  <!ATTLIST annotated by ID #IMPLIED> <!-- Allows annotator codes -->

<!-- This is an intesting thing about the following text. Use this to call out imporant
things learned. -->
<!ELEMENT interest (#PCDATA) >


<!ELEMENT semantic ANY >
  <!ATTLIST semantic type (problem|note|break|hypothesis|todo) #REQUIRED>
  <!ATTLIST semantic name ID #IMPLIED>
        <!ATTLIST semantic ref  IDREF #IMPLIED>

<!ELEMENT relationship (#PCDATA|%contexttype;|response)* >
  <!ATTLIST relationship order (%stateactionorder;) #REQUIRED >
  <!ATTLIST relationship name ID #IMPLIED >
  <!ATTLIST relationship ref  IDREF #IMPLIED >

<!-- A context is a state or situation to embed the following information. It has no
when/while because that
     is assumed. -->
<!ELEMENT context (#PCDATA|state)*  >

<!ELEMENT causal (#PCDATA|state)* >
  <!ATTLIST causal proper (true|false) #REQUIRED >
```

```
<!ELEMENT continuous (#PCDATA|state)* >
  <!ATTLIST continuous proper (true|false) #REQUIRED >

<!-- states can be abstract or instantiated.
     ex. When <state>the operation is over</state>  - abstract
         When <state>the button is released</state> - instantiated -->
<!ELEMENT state (#PCDATA|semantic|%nd;)*>
  <!-- ATTLIST state satype (%satype;) #REQUIRED -->

<!ELEMENT response (#PCDATA|action|concurrent|order|conditional|%nd;)* >
<!ELEMENT action (#PCDATA|semantic|%nd;)* >
  <!ATTLIST action name ID #IMPLIED>
  <!-- ATTLIST action satype (%satype;) #REQUIRED -->
<!ELEMENT concurrent (#PCDATA|response)* >
<!ELEMENT order (#PCDATA|response)* >
<!ELEMENT conditional (#PCDATA|condition)* ><!-- (((state)+,response)+,(response)?) >-->

<!ELEMENT condition (#PCDATA|state|response)* >
  <!ATTLIST condition proper (true|false) #REQUIRED >
  <!ATTLIST condition order (%stateactionorder;) #REQUIRED >

<!ELEMENT requirement (#PCDATA|response|semantic)* >
  <!ATTLIST requirement name ID #IMPLIED>
       <!ATTLIST requirement ref  IDREF #IMPLIED>
       <!ATTLIST requirement rtype  (%rtype;) #REQUIRED > <!--NA|false|true|non-
functional) #REQUIRED -->
<!ELEMENT observation (#PCDATA|response|semantic)* >
  <!ATTLIST observation name ID #IMPLIED>
       <!ATTLIST observation ref  IDREF #IMPLIED>
       <!-- ATTLIST observation rtype  (%rtype;) #REQUIRED -->
<!-- Applied
         - applied means that the requirement is applied to a task, object or how the
requirement is applied is explained.
              ex. User can rotate the model. vs User can rotate the model by looking
around it.
              - if the requirement is not an action or behavior, then it is 'na'
              ex. There is a 3D world in front. <- this has not action or behavior, it
just is
              - applied requirements are nearly relationships but lack specific details
to explain the state
              ex. 'using the cursor' is a state but not specific about how to use the
cursor.. click, tilt, how?
              - functional / non-functional requirements determine if applied is needed.
i.e. non-functional don't have a
                need to be applied or not, they are NA

        ? So, should we break this into functional requirements which are applied and
unapplied and non-functional
          requirements which are NA? In this case, unapplied functional requirements are
then what is needed for an
          abstract requirement. When it is applied, it is because it has come time to
determine the context or cause of
          the requirement. This splits the specification of the implementation as one can
write unapplied functional
          requirements and non-functional requirements up front and then follow through
with applied functional
          requirements.

-->



<!-- This is the original text of the artifact. It can be updated to clarify text using
the add, cut and replace. -->
<!ELEMENT original (#PCDATA|add|cut|replace|image)* >
<!ELEMENT add (#PCDATA) >
<!ELEMENT cut (#PCDATA) >
<!ELEMENT replace (cut,add) >
<!ELEMENT image EMPTY >  <!-- A link to an image file -->
  <!ATTLIST image src CDATA #REQUIRED >
```

```
<!-- this can be any type of node, it is unspecified and useful for separating the
language on a 1st pass -->
<!-- this is useful for sectioning off text for a 1st pass and then identifying its
meaning by setting the
     tag to something else. A good document will not have any 'nd' tags when finished -->
<!ELEMENT nd       ANY >
```

Appendix D. ChasmXML and Generated Code

Chasm Code Development: Process and Tools

chasm2C

Chasm2C generates the files for users to place their implementation in. The basic types of files it is capable of generating are '.h' and '.C' files for a single concept. So, for a concept named 'A', it will create two files called A.h and A.C which are ready to be compiled. It is also capable of generating Suppliment files '_sup.h', which insert code into specific parts of the .C and .h files, and Builder files, which allows for specializing a concept at runtime. Both of these files are discussed later. Chasm2C generated files are immediately compilable and by default inserts instructions for inserting code into the file. Chasm2C will not overwriting .C, .h or _sup.h files, giving warnings and stopping if the files exist.

The statements in the .C and .h files are listed in Appendix C. Some examples are:

CHASMCONCEPT : The name of the concept.

CHASMPARENT : The parent of the concept.

CHASMDECLARESTATES_PUBLIC : The concepts states which have public access.

chasmprocess

Chasmprocess is the command-line preprocessor for turning the .C and .h files into the chasm.h and chasm.C files which are passed to the compiler. It is passed a ChasmXML file for reading in information about a concept and then transforms the passed files into a concept based upon the ChasmXML file.

See Appendix C for more on the preprocessed statements. Some examples of the statements transformed, for the Button concept are:

CHASMCONCEPT : Button

CHASMPARENT : CO - because button has no parent and CO is the base concept class

CHASMDECLARESTATES_PUBLIC : States *down,*up;

chasmverify

Chasmverify is used to validate the ChasmXML file and report errors. This can be a malformed XML file, an uninterpretable XML file or a reference to another concept which can not be found.

chasm-config

Chasm-config is used mostly for compilation and install issues. It reports information regarding directory information and compile flags. It can be very useful for makefiles and setting the CHASM_HOME environment variable discussed below.

The following is a listing to help understand Chasm's code generation.

ChasmXML and its DTD

The following is a list of the ChasmXML tags with the same information written as a

DTD specification.

Example XML

```
<CHASMConcept name="" abstractbuilder="{YES|NO}" height="" width="" abstract="{YES|NO}">
  <parent name="">description</parent>
  <state name="" access = "{PUBLIC|PROTECTED|PRIVATE}" x="" y="" height="" width="">
       description</state>
  <symbol name="">description</symbol>
  <transition from="" to="" by="">description</transition>
  <component name="" concept="" access = "{PUBLIC|PROTECTED|PRIVATE}"
            cif = "{EXTERNAL|INTERNAL|VACANT|PASSED}">
    <note title="" x="" y="">description</note>
  </component>
  <{when|whenexit|while|whilenot} state="" symbol="" x="" y="">description
  </{when|whenexit|while|whilenot}>
</CHASMConcept>
```

ChasmXML DTD

```
<!ENTITY % relationship "when|whenexit|while|whilenot" >
<!ENTITY % accesstype "public|protected|private" >
<!ENTITY % ciftype "external|internal|vacant|passed" >

<!ELEMENT CHASMConcept
(description|parent|state|symbol|transition|component|relationship)* >
       <!ATTLIST CHASMConcept name CDATA #REQUIRED >
       <!ATTLIST CHASMConcept abstract (yes|no) "no">
       <!ATTLIST CHASMConcept abstractbuilder (yes|no) "no">
       <!ATTLIST CHASMConcept height CDATA #IMPLIED >
       <!ATTLIST CHASMConcept width CDATA #IMPLIED >

       <!ELEMENT description (#PCDATA) >

       <!ELEMENT parent (#PCDATA) >
              <!ATTLIST parent name CDATA #REQUIRED >

       <!ELEMENT state (#PCDATA) >
              <!ATTLIST state name CDATA #REQUIRED >
              <!ATTLIST state access (accesstype) #REQUIRED >
              <!ATTLIST state x CDATA #IMPLIED >
              <!ATTLIST state y CDATA #IMPLIED >
              <!ATTLIST state height CDATA #IMPLIED >
              <!ATTLIST state width CDATA #IMPLIED >

       <!ELEMENT symbol (#PCDATA) >
              <!ATTLIST symbol name CDATA #REQUIRED >

       <!ELEMENT transition (#PCDATA) >
              <!ATTLIST transition from CDATA #REQUIRED >
              <!ATTLIST transition to CDATA #REQUIRED >
              <!ATTLIST transition by CDATA #REQUIRED >

       <!ELEMENT component (#PCDATA|note)* >
              <!ATTLIST component name CDATA #REQUIRED >
              <!ATTLIST component concept CDATA #REQUIRED >
              <!ATTLIST component access (accesstype) #REQUIRED >
              <!ATTLIST component cif (ciftype) #REQUIRED >

              <!ELEMENT note (#PCDATA) >
                     <!ATTLIST note title CDATA #REQUIRED >
                     <!ATTLIST note x CDATA #IMPLIED >
                     <!ATTLIST note y CDATA #IMPLIED >

       <!ELEMENT relationship (#PCDATA) >
              <!ATTLIST relationship state CDATA #REQUIRED >
              <!ATTLIST relationship symbol CDATA #REQUIRED >
              <!ATTLIST relationship x CDATA #IMPLIED >
              <!ATTLIST relationship y CDATA #IMPLIED >
```

Chasm Preprocessor Statements

```
CHASMCONCEPT
CHASMCONCEPTORIGNAME
CHASMDESCRIPTION
CHASMPARENT
CHASMCOMPONENTCLASSES
CHASMCOMPONENTINCLUDES
CHASMDECLARESTATES_PUBLIC
CHASMDECLARESTATES_PROTECTED
CHASMDECLARESTATES_PRIVATE
CHASMASSIGNSTATES
CHASMASSIGNENTRYFUNCS
CHASMDECLAREENTRYFUNCS
CHASMDECLARESYMBOLS
CHASMASSIGNSYMBOLS
CHASMASSIGNTRANSITIONS
CHASMASSIGNCOMPONENTS
CHASMDECLARECOMPONENTS_PUBLIC
CHASMDECLARECOMPONENTS_PROTECTED
CHASMDECLARECOMPONENTS_PRIVATE
CHASMCHECKCOMPONENTS
CHASMCHECKCOMPONENTSBUILDER
CHASMCOMPONENTPARAM
CHASMCOMPONENTP
CHASMDECLARERESPONSES
```

Sample Generated .h File

```
#ifndef _HEADER_CHASMCONCEPT__
#define _HEADER_CHASMCONCEPT__

// Generated from a ChasmXML file. Chasm version 0.3

CHASMPARENTINCLUDE;
class State;
class Symbol;
class CHASMCONCEPTBuilder;
CHASMCOMPONENTCLASSES;
#include <chasm/COBuilder.h>



// Constructor Defines: NOTE: all followed by comma!
// These defines can be used to save the rewriting of Constructor
// parameters. You don't have to use.
#define CHASMCONCEPT_PARAMS__F
  // --- Use to define constructor parameters. Saves writing them in
  //     multiple places. ex. char* _f,
#define CHASMCONCEPT_PARAMS__A
  // --- Abridged: Use to define constructor parameters. ex. _f,
#define CHASMCONCEPT_PARAMS_D
  // --- Use to define constructor parameters with default parameters.
  //     ex. int a=0, char* c=NULL,
#define CHASMCONCEPT_PARAMS_DF
  // --- Use to define constructor parameters with default parameters
  //     but without the default. ex. int a, char* c,
#define CHASMCONCEPT_PARAMS_DA
  // --- Abridged: Use to define constructor parameters with default
  //     parameters. ex. a, c,
#define CHASMCONCEPT_PARAMS__P
  // --- Use to define parameters passed to parent in constructor
  //     parameters. ex. a,

class CHASMCONCEPT
  : public CHASMPARENT
{
  friend class CHASMCONCEPTBuilder;

  // ---------------------------------------------------------------------
```

```
  // Chasm functionality
  protected:
    CHASMCONCEPT(CHASMCONCEPT_PARAMS__F
                 CHASMCOMPONENTPARAM,
                 CHASMCONCEPT_PARAMS_D
                 char* _name = "CHASMCONCEPTORIGNAME");
  public:
    static CHASMCONCEPT* cCHASMCONCEPT(
           CHASMCONCEPT_PARAMS__F
           CHASMCOMPONENTPARAM,
           CHASMCONCEPT_PARAMS_D
           CHASMCONCEPTBuilder* _b = NULL,
           int _line = -1, char* _file = NULL,
           char* _name = "CHASMCONCEPTORIGNAME");
    ~CHASMCONCEPT();
    CHASMDECLARESTATES_PUBLIC;
      // --- Public States...
    CHASMDECLARECOMPONENTS_PUBLIC;
      // --- Public Components...
    CHASMDECLARESYMBOLS;
      // --- Symbols...

  protected:
    CHASMDECLARESTATES_PROTECTED;
      // --- Protected States...
    CHASMDECLARECOMPONENTS_PROTECTED;
      // --- Protected Components...
    void displayMeInternal(int _i = 0, char* _s = '\0');
      // --- for all Concepts, this is a method to print something
      //     about it for debugging.
    CO* createClone();
      // --- for duplicating this Concept. NULL by default
  private:
    CHASMDECLARESTATES_PRIVATE;
      // --- Private States...
    CHASMDECLARECOMPONENTS_PRIVATE;
      // --- Private Components...
  // ---------------------------------------------------------------------

  public:

};



// ---------------------------------------------------------------------
// Builder Object
// ---------------------------------------------------------------------
  // --- Its a Design Pattern. Look it up, read a book.
  // --- Used to composite extending libraries with the Concept.
  // --- Stores all functionality of the Concept so extending libraries can modify it.
class ButtonBuilder
  : public CHASMPARENTBuilder
{
  // ---------------------------------------------------------------------
  // Chasm functionality
  public:
    ButtonBuilder(char* _name = "CHASMCONCEPTBuilder");
    ~ButtonBuilder();
  // Add your functionality for the concept here...
  public:
    CHASMDECLAREENTRYFUNCS;
      // --- State entry functions...
  protected:
    void displayMeInternal(int _i = 0, char* _s = '\0');
    void builderInitialization();
      // --- Sets responses and cheks that all components are set.
      // --- Called by do_init so all extending objects of this Builder
      //     will have called all their functionality.
  // ---------------------------------------------------------------------
```

```
  public:

};

#endif
```

Sample Generated .C File

```
#include "CHASMCONCEPT.chasm.h"

// Generated from a ChasmXML file. Chasm version 0.3

#include <chasm/State.h>
#include <chasm/Symbol.h>
#include <chasm/CHASM.h>
#include <cwlib/cwDebug.h>

CHASMCOMPONENTINCLUDES;
    // --- includes header files for component concepts

#define DIRECTOR ((CHASMCONCEPT*)director)
    // --- Use this define as a shortcut to the Builder's Director
    // --- as the appropriate type describes the Concept
#define BUILDER ((CHASMCONCEPTBuilder*)builder)
    // --- Use this define as a shortcut to the builder while in the director

// ------------------------------------------------------------------------
// Static Constructor
// ------------------------------------------------------------------------
CHASMCONCEPT* CHASMCONCEPT::cCHASMCONCEPT(
       CHASMCONCEPT_PARAMS__F
       CHASMCOMPONENTPARAM,
       CHASMCONCEPT_PARAMS_DF
       CHASMCONCEPTBuilder* _b,
       int _line, char* _file, char* _name)
{
  CHASMCONCEPT* retval;
  retval = new CHASMCONCEPT(
       CHASMCONCEPT_PARAMS__A
       CHASMCOMPONENTP,
       CHASMCONCEPT_PARAMS_DA _name);
  retval->setLineInstantiation(_line);
  retval->setFileInstantiation(_file);
  if ( _b == NULL ) _b = new CHASMCONCEPTBuilder();
  retval->setBuilder(_b);
  retval->initCO();
  return(retval);
}


// ------------------------------------------------------------------------
// Constructor
CHASMCONCEPT::CHASMCONCEPT(
    CHASMCONCEPT_PARAMS__F
    CHASMCOMPONENTPARAM,
    CHASMCONCEPT_PARAMS_DF  char* _name )
  : CHASMPARENT(CHASMCONCEPT_PARAMS__P _name)
{
  CHASMDESCRIPTION;
    // --- describes the Concept

  CHASMASSIGNCOMPONENTS;
    // --- assigns the variables to the Components

  CHASMASSIGNSTATES;
    // --- assigns the public States here

  CHASMASSIGNSYMBOLS;
    // --- assign the Symbols
```

```
  CHASMASSIGNTRANSITIONS;
    // --- define the transitions.


  CHASMCHECKCOMPONENTS;
    // --- make sure that NO created concept is set yet as
    //     these should be created in builder. Also, make sure
    //     all passed components are set.
}


// ---------------------------------------------------------------------
// Destructor
CHASMCONCEPT::~CHASMCONCEPT()
{
}


// displayMeInternal
// ---------------------------------------------------------------------
void CHASMCONCEPT::displayMeInternal(int _i, char* _s)
{
  depthprintf(_i,"\033[31mCHASMCONCEPT_%d\033[0m",getCID());
  if (_s != NULL) printf("'%s' ",_s);
  BUILDER->displayMe(_i,_s);
  putchar('\n');
}


// createClone
// ---------------------------------------------------------------------
CO* CHASMCONCEPT::createClone()
{
  CO* retval;

  retval = NULL;
    //new CHASMCONCEPT(CHASMCOMPONENT__P,pointAtName(),pointAtDirectoryFull());

  return retval;
}



// =====================================================================
//  Builder - Begin
// =====================================================================

// ---------------------------------------------------------------------
ButtonBuilder::ButtonBuilder(char* _name)
  : CHASMPARENTBuilder(_name)
{

  // User code here...

}

// ---------------------------------------------------------------------
ButtonBuilder::~ButtonBuilder()
{
}

// ---------------------------------------------------------------------
// This is here so that it can be called after ALL inheriting concepts
// have passed through their constructor
void ButtonBuilder::builderInitialization()
{


  CHASMASSIGNENTRYFUNCS;
    // --- the entry functions to the states.
```

```
  CHASMCHECKCOMPONENTSBUILDER;
   // --- Checks that passed and created Components are set

  CHASMDECLARERESPONSES;
   // --- define the Responses. All should be set now.

}

// ---------------------------------------------------------------------
void ButtonBuilder::displayMeInternal(int _i, char* _s)
{
}
```

Sample Supplemental File

The sections in a supplemental file are: header include files and abstract class definitions, code include files, required parameters, header member definitions, builder header definitions, constructor code, destructor code, builder initialization code, data to display about the concept when asked and remaining implementation code.

Note: The suppliment file has a modified .h and .C file which, using C++ preprocessor statements, include parts of this Supplemental file.

```
// Files to include in the header file: ex. 'class cwList'
#ifdef CHASMSUP_HINCLUDES
#endif

// Files to include in the code file: ex. '#include <cwlib/cwList.h>
#ifdef CHASMSUP_CINCLUDES
#endif

// Add code to the Concept parameters
#ifdef CHASMSUP_PARAMS
#  define Body_PARAMS__F
#  define Body_PARAMS__A
#  define Body_PARAMS_D
#  define Body_PARAMS_DF
#  define Body_PARAMS_DA
#  define Body_PARAMS__P
#endif

// Add code to be displayed in the Concept class definition here
#ifdef CHASMSUP_H_Concept
#endif

// Add code to be displayed in the Builder class definition here
#ifdef CHASMSUP_H_Builder
#endif

// Add code to be displayed in the Concept constructor
// NOTE: Passed Components and parameters are valid to
//       use here. ex. a component named 'star' would
//       be accessible by '_star'.
#ifdef CHASMSUP_Constructor
#endif

// Add code to be displayed in the Concept destructor
#ifdef CHASMSUP_Destructor
#endif

// Add code to be displayed in the Builder builderInitialize
#ifdef CHASMSUP_builderInitialization
#endif

// Add code to be added to the displayMe method of the Builder
#ifdef CHASMSUP_Display
#endif
```

```
// Add code to be displayed in the code body here
#ifdef CHASMSUP_C
#endif
```

Appendix E. Evaluation Documents

This appendix contains the following documents:

- 6 pages - A six page tutorial on Chasm and Concept-Oriented Design
- 2 pages - The five scenarios of the heuristic evaluation of the GUI
- 6 pages - Chasm tutorial 2

NOTE: Chasm tutorial 2 was on a wiki which no longer exists at this time due to a software update of the host. Attached is the text of the webpage.

Introduction to Concept Oriented Design and Chasm

Concept Oriented Design (COD) is a design and development process which at first supports the developer in understanding their intended design and then supports the splitting of the design into an implementation of reusable and integrable concepts weaved into an application. It was developed to support the development of complex interactive systems, specifically 3D interfaces, after the domain was reflected upon and several issues were uncovered which made development with existing methods and tools problematic. Chasm is a COD system which seeks to keep the strengths of existing development while allowing for COD. It is a group of command line tools, a Java GUI and a library of reusable Concepts.

Main Points of COD with Chasm

There are several main points of development with Chasm.

- Design is an iterative notion and requires shaping ideas as new issues arise through development. Hierarchical levels of implementation are used to discover issues early and allow for easy changes. As ideas become more concrete, more detailed levels of implementation are used.
- Even a simple requirement can grow to the point that it is difficult to understand all the complexity of it. Placing a requirements into a single concept and supporting the fracturing of the requirement into multiple concepts allows for complexity to be contained and understood in manageable units. Additionally, this allows developers to put off explaining complexity into a concept while they focus on other parts of the development.
- Describing a concept is not necessarily the hardest part of implementation but also important is the describing of the concept interacting with other concepts. This is managed by using a custom state machine formalism to keep ideas distinct, define hierarchical connections between them and mediate issues which arise with minimal input from the developer.
- Sequential programming is good at calling functionality at a known point of execution in an application and is well understood by developers and supported by development tools. Chasm allows C++ code to be executed when the state machine enters a state; even allowing the called functionality to cause changes in the custom state machines.
- Reusablity and modifying and extending existing code is problematic and essential. This is supported in Chasm by both the multiple levels of description in COD, which allow designers to choose the level of detail needed for their task, and also the Chasm internals which assures concepts always function despite added functionality.

Chasm Terminology

In Chasm, a *system* refers to an application and all of its implementation. What is to occur in a system or a part of a system is the *envisioned behavior* and it is described in conversational language. In a sizable system, the envisioned behavior is complex, interconnected and in many cases not fully understood when development begins.

Despite this, conversational language is capable of describing it at a high level – and the process of describing it causes a growth in the developer's understanding of the system. To simplify the concept's envisioned behavior, it is then decomposed into smaller envisioned behaviors each of which describes part of the system. Potentially, each part can then be decomposed further until ultimately, the resulting envisioned behavior is simple enough to be easily understood and thus easily implemented by conventional means.

In Chasm, a *concept* encapsulates a single, stateful and complete implementation of an envisioned behavior. It is represented as a custom state machine with states and symbols with transitions to new states caused by symbols. *States* are for clearly defined points in the flow of the envisioned behavior and represent either a resting point of the envisioned behavior, a split in the path of the envisioned behavior or a point of interest to external functionality of the envisioned behavior of the concept. A *symbol* is a change someplace in the system which means something specific to this concept and signals that a change in the envisioned behavior needs to occur. *Transitions* are valid paths in the envisioned behavior between states due to symbols. This results in a clear flow between important parts of the envisioned behavior due to changes in the system and more importantly, directly shows and limits the concerns, as well as the proper functioning, of the concept.

Concepts can respond to changes in other concepts. When a concept enters a new state, it signals a change in its envisioned behavior which can drive the flow of behavior in external observing Concepts by causing a *response*. The response was modeled after our observations of developer language use which typically followed a When/Then or While/Then syntax. A response is created by an observing concept registering a symbol of itself with a state in an observed concept which is called a *component concept*. When the component concept enters the observed state, it fulfills its duty by sending the registered symbol to the observing concept. The observing concept has no concern of how the observed state is reached and the component concept has no concern of what the symbol means making a clear separation of concerns. Because a concept is considered stateful and only affected by its own symbols and encapsulated state, the concept will continue to function correctly despite additional concepts to the system.

Multiple descriptions of the envisioned behavior exist in Chasm making it easier to reuse and understand. The concept itself is given a descriptive name, its envisioned behavior is described in language and its states and symbols are given descriptive names and comments. The lowest level of implementation is the *state entry function* where functions and method calls are placed in typical sequential programming style. Depending on the level of understanding needed, the developer can choose which level to learn about the concept as opposed to relying on documentation which could be missing, out of date or incomplete. In Chasm, much of the documentation of the concept is the concept's structure.

Development Steps

Overall, development is iterative and exploratory in large part we feel due to developers not fully understanding what they want until they have to implement it. Chasm's use of description at different levels of resolution as well as concepts which fully encompass single ideas and put off details until later, works to fulfill these observations. Chasm's development method generally iterates through the following steps for each concept and

multiple concepts in a system can be in different steps of development.

Step 1) *Describe the envisioned behavior in common language.* The developer creates a description of the envisioned behavior in a top-down manner. This creates a record of the envisioned behavior for the developer, their collaborators or future maintainers or re-users. This language is important as language is closer to the envisioned behavior and thus more correct than the implementation which is an attempt at creating the envisioned behavior.

Step 2) *Decompose each behavior into a concept, reusing concepts as possible.* After the developer has described the envisioned behavior, they have to break the description into concepts. This proceeds in a top-down as well as bottom-up manner. We have observed that humans have a natural ability to describe complete concepts at appropriate levels of detail.

Note about steps 3 & 4) The order of steps 3 and 4 tends to depend on the situation. If the developer is creating the first concept, step 4 occurs first, else step 3. This is because the creation of the first concept is about finding important changes in the system which drives the concept and component concepts are created to supply states which send symbols into the concept.

Step 3) *Identify useful states in a concept.* For the concept, the developer must define a set of useful states where a state is a useful point in the concept's flow. Some states are useful internally to a concept whereas other states potentially are useful to other concepts. These states should then be described so we know what it means to be in that state.

Step 4) *Identify the cause of every change in a concept and create a symbol for it.* These symbols should be described so the developer knows the meaning of the symbol to the concept so as changes occur in the system, new causes of the symbol can be identified by checking it against the description of the symbol.

NOTE: Every state change in a concept does not need to be a state! Existing programming languages are very good at implementing a sequence of functionality and should be used to do so.

Step 5) *Determine the effect of each symbol on each state.* The effect of a symbol in a state could be a transition to a new state, a transition back to the current state, the ignoring of the symbol or an error. Considering all combinations in this manner is feasible in Chasm because of the local scope of symbols. Because it is feasible to consider all the combinations, the occurrence of unanticipated errors is greatly reduced.

Step 6) *Implement the state entry function.* Each state has the option to have a state entry function associated with it that is called when that state is entered. Because much of the complexity is described with the Chasm structures, the implementation of the state entry functions should proceed smoothly and be called precisely when needed.

Concepts of Chasm

The concepts of Chasm are divided into the base and vocab libraries. The libraries are constantly growing as new concepts are created and added. The following lists the most common and used concepts.

In base:

Conjunction – Merges the state space between two concepts by allowing multiple boolean operators.

Switch – A concept for enabling and disabling responses like a switch.

Set – Holds a group of concepts.

MirrorSet – Mirrors a Set with a concept here for each item in the Set.

in vocab:

AnEvent - This is a one state concept which can be triggered by external functionality.

Animated - This is used to map a given time to a 0-1 scale and notify actions to occur over that timeline. It can be extended for a lot of uses.

Button – Represents an abstract button with up and down states.

Data - This represents a generic data item. When the value of it changes, it enters the changed state. Examples are Integer or Location.

Function - It maintains a value and enters the incorrect state when its dependencies change. It is recomputed either when explicitly asked for its value or whenever its dependencies change.

Logger – Logs user defined information in response to a change in the system.

Resource - A Resource concept allows for mediation of its data. Different concepts can be given control of the Resource at different levels of access and there are different ways to change the overall data based upon how the updates come in.

WorldObject - The WorldObject is the information about a particular object in a virtual world. This includes geometry, color, position, etc.

A Domain Example

Let us look at a simple implementation of the Raycasting technique and how it would progress under Chasm's design method. The first action we take is to describe the Raycasting technique. We do so as (step 1): *When the user pushes a button, set the selected object to the currently pointed at object*. This is placed into a concept called Raycasting (step 2), shown in Figure 1. A concept for the button is pulled from the library with its two states of up and down. The useful state in Raycasting is *checked*, meaning check for a selection, but since external functionality will most likely want to be notified when the selected object changes,

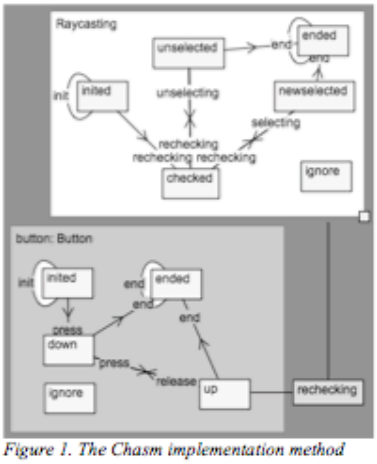

Figure 1. The Chasm implementation method quickly breaks the desired behavior into the Chasm architecture which later implementations can extend.

we create the states *newselected*, meaning a new object has been selected, and *unselected*, meaning nothing was selected (step 3). Next (step 4), we determine changes which would indicate a need to recheck for selection and the point when a new selection is made. We create symbols to indicate this, calling them *rechecking*, *selecting* and *unselecting*. The button is connected with a response to send Raycasting's rechecking symbol when the button's up state is entered. The selecting and unselecting symbol is given by the state entry function as needed (shown below). We then determine the effect of each symbol in each state for (step 5). Since we have two symbols and three states, we have to consider six cases and create the transitions as shown. The last step is (step 6), the implementation of the state entry functions, which is easy since only the state checked has one:

```
WorldObject* selectable = getSelectableObject();
if ( selectable != selected )
{selected = selectable; return selecting; }
else {selected = NULL; return unselecting; }
```

The return value in Chasm is used to return a symbol, if needed, back to the system based upon the processing that occurred. Also, notice the selective use of Chasm's structures. Using a Concept to encode the function getSelectableObject() would be tedious but quite easy with procedural programming.

Let us also add in feedback for highlighting selected objects. The description can be given as: *When the user releases a button and something is selected, unhighlight the selected object. Then, select the new object, highlighting it red. If nothing is selectable, then unhighlight the selected object and set the selected object to nothing*. We can keep our existing implementation of the Raycasting technique and create a new concept with the two states of *nothighlighted* and *highlighted* as shown in Figure 2. The transitions between the states are the symbols of *on* and *off* and all transitions are valid and covered by a symbol. For our purposes, we create the response relationships of:

```
When selection.unselected
then HighlightObject.off

When selection.newselected
then HighlightObject.on
```

Notice how the response relationship does not require the developer to have access to the original code, just the states. The final step is the implementation of highlighted's state entry function which is:

```
if (current != selection->selected)
  selection->selected->unhighlight();
current = selection->selected;
current->highlight(color);
```

And unhighlighted's entry function which is:

```
if (current != NULL)
  selection->selected->unhighlight();
```

There are no changes to the existing concepts of Raycasting or button because Chasm weaves this functionality around the existing concepts. The two states and two symbols of the new concept HighlightObject have nothing to do with the selection of an object in Raycasting. This shows how Chasm limits code interdependencies. If there are problems with the implementation, the developer knows specifically on which component concept and response to focus.

Development with Chasm

Chasm integrates into existing develop practices and tools and comes with a standard library of reusable concepts. The structure of Chasm is developed with a custom graphical tool called *chasmgui* which outputs ChasmXML files. The structure in the ChasmXML files and the state entry functions are then integrated into C++ source code to be managed with C++ development tools and methods. This is done with Chasm preprocessors, run from the command-line, which take in both the ChasmXML file and source code and turn the resulting implementation over to the compilers. This allows both the ChasmXML and implementation files to be modified simultaneously and iteratively without imposing an unnatural order on the developer. Chasm also comes with a standard library of reusable concepts which encapsulate their state, ensuring that these concepts can be reused and integrated into new systems. More information about development with Chasm can be found in the Chasm tutorials.

Chasm GUI Heuristic Evaluation

This evaluation tests the Chasm GUI used to manipulate the chasmxml files through a Heuristic evaluation. Nielsen's 10 Usability Heuristics are provided. The following scenarios will be used to guide the evaluation. For each issue you uncover, you will list the heuristic(s) it violates, a description of the issue and a rating for severity on a scale of 1-5. A description detailing what each level of severity refers to will be provided, as will a form to fill in the issues you uncover. During this evaluation, please feel free to ask the researcher questions about the interface.

Please read the introduction to Chasm and Concept Oriented Design

Secondly, please read through the tutorial on Chasm and Concept Oriented Design now.

Scenario: Concept Creation

Open a new package directory called /Users/cwingrav/code/chasm/heuristic/h<id>. Create a concept named "AandB". To this concept, create the symbol "aing" and "bing". To this same concept, create the states "aed" and "bed".

We will now give descriptions to the states and symbols. Set the description of state "aed" to "state of a" and to state "bed" set its description to "state of b". Now, resize the concept's boundaries so it is larger. For the symbol "aing", set its descriptions to "action of a" and for the symbol "bing", set its description to "action of b".

We will now create transitions the following transitions:

- For state "inited", when it receives the symbol "aing", go to state "aed".
- For state "inited", when it receives the symbol "bing", go to state "bed".
- For state "inited", when it receives the symbol "aing", go to state "aed".
- For state "aed", when it receives the symbol "ending", go to state "end" and when it receives the symbol "aing" ignore it.
- For state "bed", when it receives the symbol "ending", go to state "end" and when it receives the symbol "bing" then go to "bed".

Move the states around so they display nicely. Rename the concept to "AandBConcept" and place it at a good position visually on display. Rename state "aed" to the improper c++ variable names "1aed" and "ae d".

Scenario: Components

Add a component to AandBConcept of Integer called "anint" and make it public and external. Delete it and add a new component called "aninteger" which is protected and internal. Create a response so that when aninteger.changed occurs, it sends symbol "aing" to AandBConcept. Delete this response and make that same state send symbol "bing". Move the response to a good layout location. Place a note on "aninteger" stating that "This is an integer which affects the AandBConcept". Move the note to a good location. Hide the component "aninteger". Close the concept and reopen it.

Scenario: Inheritance

Chasm also supports inheritance so we will be extending AandBConcept in this scenario. So, create a concept called "ExtendToC" and give it a description. Inherit off of "AandBConcept" by setting "ExtendToC"'s parent to "AandBConcept". Notice how

inherited components, states and symbols are added. Now, to "ExtendToC", add state "ced" and symbol "cing". Give transitions into state "ced" from "inited" by "cing", from "aed" by "cing" and from "bed" by "cing". Now, create two more components on ExtendToC, of default settings, of type switch called "aswitch" and a component of type Integer called "anotherint". Create two responses such that when component "anotherint" enters the "changed" state, have it give the "cing" symbol to "ExtendToC" when the switch is enabled. Lay all the Chasm structures out nicely on the panel.

Rename the inherited states and symbols in "ExtendToC". Open "AandBConcept" and rename state "aed" to "aaed" and find "aaed" in "ExtendToC".

Scenario: Modification

Create a concept, delete it. Open concept Raycasting and unhide and layout its components, making sure to scale and scroll the panel as needed.

Scenario: Understand Raycasting

Open the Concept Raycasting. Using the GUI, learn about how raycasting works and describe it to the researcher. Describe how the components of Raycasting affect Raycasting and what the states and transitions mean. How well does the GUI support your task in understanding the Concept of Raycasting?

NOTE: This scenario is different from the above scenarios because it is looking at how well Chasm designs its information layout, not it functionality.

Final

The researcher has some questions for you now which will be recorded. If you have any objection to the recording of the questions, please let the researcher know.

Please fill out the demographics questionnaire now.

Thank you for your participation!

ChasmIntro_tut2

This is part of the ChasmIntro series.

The second introduction to Chasm deals with creating a world in SVE where the user points at an object, pushes a button and the object moves to a new location.

This section demonstrates how to solve a simple 3D interaction problem inside Chasm. The description of the problem is this: When the user selects the end object out of a line of objects with raycasting, have that object move to the center and above the other objects.

Before the selection

tut2.png

After the animation

tut2a.png

Enter the Directory and Load the GUI

Go to the directory chasm/tutorials/tut2_move_object where you will find a Makefile, the tutorial2_base.C file and the directory answers which contains finished versions of the code. There are also files for SVE such as sve.init, termio.h and tut2.world.

* Set your CHASM_HOME variable to include the current directory.

* Open the chasmgui in the current directory.

Create the Concept with the GUI

Go through the following steps. Use the help button "?" as needed. Some of the descriptions are purposefully vague. If you become stuck, see the answers directory.

* Open the current directory as a package using the "Load Package" button and selecting, with the GUI's directory selector, tut2_move_object.

* Create a new Concept and call it Scenario_MoveObject and place the description into it of 'When the user selects the end object out of a line of objects with

raycasting, have that object move to the center and above the other objects.'

* Since you know you are using RaycastingSelection and pointing at a certain object, create a protected component of RaycastingSelection, which is in the vocab directory, which is passed and called "raycasting".

* Open raycasting and review its states, symbols and components until you know how it works. Also, see its parent concept. Hide raycasting and then unhide it by clicking on it in the Component frame.

* Create a component WorldObject which is protected, passed and called "targetedobject".

* Since you are monitoring an object for selection, create a protected component concept of class MonitorObjectForSelection and call it "mfs". The intent is that it will enter the state selected when the targetedobject is selected by raycasting. You will call the static constructor of this concept later in the c++ code since you are setting the CIF to created.

* Create a note on mfs of MonitorObjectForSelection which states "Determines if the targeted object is selected" and place the note in a good position to be read.

* The intent of this concept is to make the targetedobject move when it is selected. We have already represented the targetedobject being selected with the MonitorObjectForSelection so now we need to create a protected animated translation concept to move the targetedobject. To do this, create a component concept of class AnimatePositionTranslation and call it "apt".

* Create a response from the mfs.selected state to the apt.animating symbol.

* Save the concept.

* Open the file Scenario_MoveObject.chasmxml to see what you created.

Set up the code

* Create a supplemental file and .C and .h file from the Scenario_MoveObject.chasmxml file. Do this by typing:

chasm2C -s Scenario_MoveObject

* Again, type the command:

chasm2C -s Scenario_MoveObject

Notice how it warns you that the Scenario_MoveObject_sup.h and .C and .h files exist. Chasm makes sure you do not overwrite your code files.

* Move the tutorial2_base.C file to tutorial2.C.

Implement the C++ Code

* You now need to add C++ code in the tutorial2.C. Open tutorial2.C in any editor you use, and follow the following steps.

* Let us create the Scenario_MoveObject Concept. First, include its header file by uncommenting the include.

#include "Scenario_MoveObject.chasm.h"

Notice this uses the .chasm.h version and not the .h version. The .chasm.h version (which you have not created yet) has had a preprocessing command (chasmprocess) run over the Scenario_MoveObject's .h and .C file using the information in the .chasmxml file to create the .chasm.h and .chasm.C files. These files hold information from the user's code and the Chasm structure and are overwritten constantly so do not modify them directly. Only modify the .C, .h and _sup.h files.

* Now, create an instance of Scenario_MoveObject so in the main function, add the declaration:

Scenario_MoveObject.chasm* scenario = NULL;

and before Chasm_InteractiveShell() function call, add the call to the static constructor:

scenario = Scenario_MoveObject::cScenario_MoveObject(
raycasting, testcube[NUMCUBES-1]);

testcube is a WorldObject and wand and raycasting are the passed in Concepts to Scenario_MoveObject.

* Create these concepts, wand and raycasting, to pass in as well as ray, the WorldObject used to point in Raycasting. Declare them both:

```
Button* button;
Raycasting* raycasting;
WorldObject *wand,*ray;
```

and create instances of them:

```
wand = WorldObject::cWorldObject("wand",
            new WorldObjectBuilderSVE(
            SVE_loadObject("wand.obj","wand")));
SVE_attachToObject(SVE_findWorldObject("wand"),
            SVE_findWorldObject("SVE hand") );
ray = WorldObject::cWorldObject("ray",
            new WorldObjectBuilderSVE(
            SVE_loadObject("ray.obj","ray"),
            SVE_findWorldObject("SVE cursor")));

button = Button::cButton(new ButtonBuilderSVE(1));
raycasting = RaycastingSelection::cRaycastingSelection(10.,
              s_selection,selectable,ray,
              new RaycastingSelectionBuilderSVE());
CHASM::WHEN_state_THEN_symbol(raycasting,button->down,
                        raycasting->rechecking);
```

Much of this complexity deals with SVE calls and creating the SVE builders.

* Also, you need to create something which tells the Raycasting concept to check for a selection. Create a Button concept and response.

```
Button* button;
...
button = Button::cButton(new ButtonBuilderSVE(1));
```

Notice how Builders are used in the Concepts. Read about builders here. They allow the same concept to be implemented in different toolkits by abstracting their functionality in Chasm structures.

* Compile.

make

* Run

./tutorial2

Finish implementing the Concept

* Notice the error you receive.

Err: Scenario_MoveObject.C line(134) in builderInitialization: ERROR: you did not set Component variable 'mfs' in Scenario_MoveObject_1

We have to implement the Component concepts in Scenario_MoveObject.

* Add the lines to the supplement file in the Constructor section:

```
setComponent("mfs",
    mfs = MonitorObjectForSelection::cMonitorObjectForSelection(
    _targetedobject,_raycasting));
setComponent("apt",
    apt = AnimatedPositionTranslation::cAnimatedPositionTranslation(
    -0.8,0.4,0.,5.,
    _targetedobject->pointAtPosition()));
```

Notice you do not need to include header files as this is done automatically for you. These components are created here because you told Chasm that the Concept was responsible for them by setting the CIF to created.

Also notice how "mfs" and "apt" are the names of the components you set using the Chasm GUI.

Running tutorial2

* Compile it again.

* Run it again.

* When you run 'tutorial2', you will see the world in front of you. Use the vtsve command 'cmd' to control the user.

To move the user head to a good position type: s 0 1.8 0 0 0 0

To move the hand, type:

t 2

s 0. 1.5 -.5 28. 0. 0.

Use the 'fly' command to move the wand and shift arrow keys to change the angle of the ray. When you are pointing at the last object (as per the image above), click the button using the command: b 1

Appendix F. Vocabulary

Base: (7)

Gather Set MirrorSet ConjunctionCheck Conjunction Switch Continuous

Vocab: (28)

Button AnEvent Logger Function Relationship Data Location Orientation Position Integer Counter Resource Animated AnimatedTranslation AnimatedPositionTranslation WorldObject AngularError AbstractSelection AbstractSelectionAndManipulation RaycastingSelection HighlightSelected MonitorObjectForSelection TrackerPosition AbstractPositionRelationship AbstractSelectionAndManipulation LinearPositionRelationship ScaledPositionRelationship CustomPositionRelationship