A Natural, Tiered and Executable UIDL for 3D User Interfaces Based on Concept-Oriented Design

CHADWICK A. WINGRAVE and JOSEPH J. LAViola JR.
University of Central Florida, USA
and
DOUG A. BOWMAN
Virginia Tech, USA

3D user interface (3DUI) design and development requires practitioners (designers and developers) to represent their ideas in representations designed for machine execution rather than natural representations, hampering development of effective 3DUIs. As such, Concept-Oriented Design (COD) was created as a theory of software development for both natural and executable design and development. Instantiated in the toolkit Chasm, Chasm is a natural, tiered, executable User Interface Description Language (UIDL) for 3DUIs resulting in improved understandability, reduced complexity and reuse. Chasm’s utility is shown through evaluations by domain experts, case studies of long-term use and an analysis of spaces.

Categories and Subject Descriptors: H5.2 [Information Interfaces and Presentation]: User Interfaces; D.2.2 [Software Engineering]: Design Tools and Techniques—User Interfaces; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms: Design, Human Factors, UIDL, 3DUI, Concept-Oriented Design

Additional Key Words and Phrases: Interaction Techniques, Natural Programming, non-WIMP interfaces, Chasm, UIMS

1. INTRODUCTION

User Interface Description Languages (UIDLs) for 3D User Interfaces (3DUIs) are the subject of a growing community of researchers [Dachselt et al. 2006; Dachselt et al. 2007; Latoschik et al. 2008; Shaer et al. 2008]. Curiously, UIDLs incorporate the approaches of User Interface Management Systems (UIMSs), formal languages and model-based approaches for the creation and specification of 3DUIs, despite the failing of these approaches to catch-on in traditional user interfaces [Myers et al. 2000].

In spite of the failings of these approaches, the differences between 3DUIs and
traditional interfaces justify their return. While no comprehensive list has been created, many have commented on design and development problems of 3DUIs [Green and Jacob 1991; Herndon et al. 1994; Myers 1991; Myers et al. 1993; Steed 2008; Jacob et al. 1999; Wingrave 2008; Wingrave and LaViola 2009]. Nearly all agree that the amount and type of IO found in 3DUIs is problematic. One aspect of this is that 3DUIs operate as recognition-based interfaces [Mankoff et. al. 2000] and have to deal with error-prone data. 3DUIs also have more degrees of freedom and are more unconstrained, basing their interface to some degree on the real-world [Jacob et. al. 2008]. Complexity also plays a role in 3DUI development, as each additional feature of a 3DUI must consider each existing feature of the system. As such, incremental improvements to 3DUIs lead to drastically more complicated interfaces to develop [Wingrave 2008]. This has been noted as a classical problem of software engineering [Brooks 1987] and referred to as the State Space Explosion [Harel and Politi 1998].

The consequences of these design and development problems are potentially far reaching. 3DUIs are incorporated in a variety interfaces such as virtual reality, mixed reality, augmented reality, tangible interaction, ubiquitous interaction, mobile computing and even entertainment computing and gaming [LaViola 2008]. If UIDLs or some other approach fails to improve design and development of 3DUIs, in some of the most promising and innovative interaction modalities the potential exists for an interface stagnation. This is surprising given that practitioners have little problem envisioning new interfaces and many avenues to improve interaction remain [Bowman et. al. 2006].

To address this potential stagnation, the theory of Concept-Oriented Design (COD) was created for the design and development of 3DUIs based upon the language and diagrams of practitioners¹ [Wingrave 2008]. The rational is that a UIDL based naturally [Myers et. al. 2004] upon practitioner thought, as opposed to machine execution, will promote reuse and scale in complexity with the practitioner’s envisioned 3DUI behavior. After studying practitioner journals and notes, performing interviews and capturing the language of developers used to describe 3DUIs, five principles of COD were created around the notion of a concept. In COD, a concept is the unit of development, representing an encapsulated cohesive software concern [Dijkstra 1982] – a reusable chunk of functionality. Keeping with practitioner’s natural representations, a concept has a multi-tiered representation, each tier encapsulating a different information type. These tiers represent, and externalize, the concept’s encapsulated functionality for composition

¹ Practitioners here refer to any individuals that designs and developers 3DUIs such as researchers and developers.
into more useful concepts, for reuse and in ways that integrate with existing tools and practice. The tiers are then executed in a consistent fashion such that the concepts remain understandable and continue to function according to the original practitioner’s intent, a critical aspect of COD. In this way, an individual concept’s envisioned behavior is declared in its tiers while its interaction with the entire system is observable as a flow of behavior at runtime.

Concepts are then as much the means of communicating ideas as developing them, just as languages form by the selection of useful terms and expressions. COD relies on community members to pass judgment on concepts by picking the useful and release their concepts back to the community to be judged. The “specification” then becomes the commonly used concepts; like a vocabulary of terms. This vocabulary of concepts addresses the problem that any single specification for 3DUIs runs the risk of being too simple to be useful, too complex to learn, too slow to adapt or too dependent on a single group for progress. Focusing on naturalness in the representation allows practitioners to form their envisioned behavior in a form closely matching how they think and then create concepts that match.

To explore the principles of COD, we developed the toolkit Chasm, named for the divide between the practitioner’s envisioned behavior and their intended 3DUI. While many design and development approaches focus on either declarative functionality with low thresholds to learning, event or callback architectures for high ceilings [Myers, Hudson, Pausch 1999] or object decompositions focused on the “nouns” of the system and not the behavior, COD in Chasm unifies each approach in one natural representation. As such, several ways exist to design systems: 1) functionality can be declared by reusing existing concepts, 2) causal relationships between existing concepts can compose new functionality, 3) a formal definition of behavior and changing state can be defined in an automata, and 4) concepts can be programmed with existing languages and using existing APIs. In this way, high-level expression and low-level control are unified in a single representation with multiple tiers, allowing practitioners to choose the level of control and understandability they require.

In this article, we describe how Concept-Oriented Design was developed as a UIDL for 3DUI design and development. In section 2, we discuss how 3DUI development differs from 2DUIs and work related to COD in Chasm. Following this, section 3 discusses the naturalness [Myers et al. 2004] of the UIDL and the creation of COD, separate from its implementation in Chasm. In section 4, we discuss the resulting tiered model inside a concept as well as its ChasmXML representation. In section 5, we discuss
the execution of the UIDL. The evaluation of COD in Chasm is covered in section 6 and we conclude with a discussion and future work. Throughout this document, we will return to the World-In-Miniature technique (WIM) [Stoakley and Pausch 1995] as an example of the benefits of COD in Chasm. As a moderately complicated interaction technique, the WIM shows the naturalness of COD, how Chasm's tiers represent the WIM and Chasm’s execution recreates the original practitioner’s understanding of the WIM’s flow.

2. BACKGROUND AND RELATED WORK

2.1. Why UIDLs Make Sense for 3DUIs

UIMSs, formal languages and model-based approaches, which are incorporated in UIDLs, offered promise to traditional user interface development yet failed to catch-on [Myers et al. 1999]. Part of the failure is attributable to the lack of control, high learning thresholds, a focus on direct manipulation over dialog management and limitations on the expressible interfaces. However, probably a more important reason was the standardization of the WIMP metaphor for 2D interfaces, creating a standard metaphor for which to develop toolkits. As such, these approaches were no longer needed as standards took the place of innovation and exploration.

3DUI design and development differs from traditional interfaces in such a way that approaches above are still applicable. For example, 3DUI design and development is more difficult and beset by many domain difficulties. To start, there is no common metaphor for 3DUIs like WIMP for traditional interfaces and more importantly, it is probable that such a metaphor may never be realized [Bowman et al. 2004] due to the variety of devices, interaction styles and contexts of use. Additionally, traditional user interfaces were constrained by the desktop but 3DUIs, in a similar way to post-WIMP [van Dam 1997] and reality-based [Jacob et al 2008] interfaces, are based upon real-world themes such as naïve physics, body awareness, environment awareness and skills and social awareness and skills. Another example is how traditional interfaces have an emphasis on layout and interaction with graphical icons and widgets whereas 3DUIs define behaviors, context, movement and reaction to the user, objects and the environment. Regarding event-based architectures used in development, traditional interfaces have a generally understood event propagation mechanism and common event sets (i.e. a button has a ‘click’ and a scrollable window ‘scrolls’) that are lacking in 3DUIs. This is made more difficult by many events in 3DUIs being probabilistic or recognition-based [Mankoff et al. 2000]. For example, an event could be, “When almost
touching an object” or “If the vase is released on top of the table”, both events being subjective and context dependent. Finally, 3DUIs are better represented through dialog management [Green 1986] over direct manipulation.

For the above and the following reasons, UIDLs for 3DUIs are growing in importance. UIDLs manage the tedium of events, create a higher-level model for better abstractions and can speed-up development, allowing more time for design iterations. In this way, UIDLs are useful for automating the creation of an interface for multiple hardware configurations and avoiding the hardware porting problem. This is important, as 3DUIs have no standard hardware configuration to standardize an interface design. In this way, a UIDL can potentially create consistency in 3DUI interfaces, standardizing the way practitioners design, develop and discuss 3DUIs. A UIDL’s model can even be used to port a 3DUI between toolkits, freeing the application from obsolescence.

There are downsides to UIDLs but even these are lessened in 3DUI. UIDLs have high thresholds to learning. However, so too do 3DUI toolkits. UIDLs have limited expressiveness. However, most 3DUIs used in applications are simple, used to navigate an environment or perform the occasional object manipulation. UIDLs generate similar looking interfaces. 3DUIs follow a dialog management approach and as such are not as visual as direct manipulation interfaces would be. The visuals the 3DUI depends on are not usually icons but the content of the environment that are augmented with the interface’s functionality. UIDLs have been criticized for creating low quality interfaces. In 3DUIs, even low quality interfaces are time consuming and difficult to develop.

For these reasons, UIDLs are promising in 3DUI design and development.

2.2. 3DUI Requirements and Toolkits
The functionality requirements for 3DUIs are quite large and varied. 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 displays and realistic applications requiring realistic interaction, to lower fidelity graphics displayed in a web browser, with interactions as simple as navigating a scene. As such, no single tool has been found sufficient for all members of the 3DUI community and new toolkits are constantly being created. Toolkit requirements include: graphics, sound, hardware support, performance constraints, content loading, architecture scale, development support, collaboration requirements, local expertise and specialized functionality such as collision detection, virtual humans or physics support. Few toolkits come close to supporting all these varied requirements, as there is a tradeoff between: 1) supporting
more requirements or 2) supporting the existing requirements well. Because of this, most toolkits focus on supporting a subset of the functionality well and with time, becoming more robust. Unfortunately, due to rapid changes in the underlying technology, many robust toolkits are left supporting obsolete feature sets [Steed 2008].

There have been many attempts to improve the design and development of 3D environments through the use of toolkits. Some event-based tools are VR Juggler [Biebaum et al. 2001], DIVERSE [Kelso et al. 2003], Studierstube [Schmalstieg et al. 2002], Panda3D [Goslin and Mine 2004] and ARToolkit [Kato et al. 2000] to name a few. Event-based architectures are unfortunately inherently limited in the amount of complexity they can handle [Myers 1991]. Declarative approaches are found in the toolkits of Contigra [Dachselt 2001], X3D (http://www.web3d.org/x3d) and VRML (http://www.w3.org/MarkUp/VRML). A rise in game programming has lead to the creation of its own set of tools such as OGRE (http://www.ogre3d.org), Virtools (http://www.virtutools.com), Torque (http://www.garagegames.com) and XNA (http://www.xna.com). 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 [Taylor et al. 2001] has been successful at abstracting away device handling. More exhaustive lists of toolkits can be found here [Steed 2008][Ray 08]. As extant work has documented, a disturbing trend among 3D toolkits is their tendency to expire [Steed 2008].

Using the terms of [Myers, Hudson & Pausch 1999], we can evaluate each approach’s strengths and weaknesses. Event-based approaches offer high ceilings, or the ability to express many things with the tool but at the cost of high complexity. Declarative approaches have low thresholds to leaning but limited ceilings, you can only do what was predefined, and face a problem of moving targets, where a specification may allow declaration of no longer useful functionality. Game tools are useful but potentially have a path of least resistance that guides development towards game interfaces and not 3DUIs. Scenegraphs are useful for their predictability but only address a limited scope of the 3DUI.

2.3. Related Approaches

Several higher-level approaches to software development show promise when applied to 3DUI development. Dataflow programming originally focused on the potential for parallelism but more recently has focused on the software engineering benefits [Johnston et al. 2004]. Unfortunately, dataflow programming is a very different paradigm for
practitioners and this has limited its uptake. Flow-based programming [Morrison 1994] improves on dataflow programming by using blackboxes of code that run as dictated by the flow. However, flow decomposition has been noted as non-optimal for maintainability and development, as well as leading to confusing systems [Parnas 1972]. Many visual programming languages [Johnston et al. 2004] are similar to dataflow languages, attempting to increase the understandability of the representation through visualization. These are often domain specific and designed for end-users. Model-based approaches are related [Selic 2003; Kent 2002]. Generally, these involve the creation of a model and then either the translation or direct execution of that model with the benefit being that the model is more portable, understandable and faster to develop.

Formal language-based tools are another promising approach, especially regarding the management of the events in event-based architectures. For example, formal models can perform verification checks for higher reliability. Such approaches have been useful for Feature Interaction [Calder et al. 2003] and Statecharts [Harel and Politi 1998]. Statecharts shows promise in that the state machine representation is understandable to most practitioners, Statecharts is good at expressing flow, the hierarchical representation manages complexity and is has constructs to represent concurrency. However, Statecharts has many problems [von Der Beeck 1994], some specific to 3DUI development [Wingrave 2008].

2.4. Advanced 3DUI Tools

Many tools for 3DUI design and development are beginning to use the approaches found in UIDLs. The Interaction Framework For Innovation (IFFI) [Ray 2008] uses a higher-level API in which to write the 3DUI. The IFFI then executes the instructions for the intended toolkit, following the Adapter pattern. Interaction Technique Markup Language (InTml) [Figueroa et al. 2002] is a dataflow description of 3D interaction that generates code for X3D and VRML. FlowVR [Allard et al. 2004] is a dataflow architecture designed for the design and development of distributed virtual reality systems. PMIW [Jacob et al. 1999] is a UIMS 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 3DUIs. NiMMiT [Boeck et al. 2007] is a toolkit for the design and development of 3DUIs, which has both a state and dataflow representation. Marigold [Willans et al. 2001] is a 3DUI specification toolkit that has a split representation between discrete and continuous
behaviors, similar to PMIW and NiMMiT. Finally, Alice3D is a traditional 3DUI toolkit but was optimized to have a representation that naturally matched the thinking of developers [Conway et al. 2000].

2.5. Approaches to Decomposition

Decomposition of a software system is a process of: 1) selecting a piece of the problem to solve, 2) determining the components in this piece by a decomposition approach, 3) describing the component interactions and 4) repeating until decomposition is no longer needed. Three main decomposition approaches exist: functional, object and aspect.

A functional decomposition proceeds from a generic description, to a refinement of smaller and more specific steps [Wirth 1971]. The result is a description of the desired functionality. Unfortunately, functional decompositions have issues finding the main or top-most functionality in a system. Compounding this is that successful systems change, having new requirements added to them. As such, 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” [Meyer 1997]. Functional decomposition also has issues with premature commitment to ordering [Meyer 1997]. The specification of a system as a list of steps artificially locks the system into that order. When the system and requirements change, the built-in step ordering is too inflexible to change.

An object decomposition is focused on the things or nouns in the system over the relationships between them. The object motto has been stated as, “Ask not first what the system does: Ask what it does it to! [Meyer 1997].” The purpose of identifying the objects first is that objects are more stable in a system. As systems change 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.

An aspect decomposition [Kiczales et al. 1996] 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 non-primary 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 often-used example for an aspect decomposition is the placement of logging code, which is often repeated, 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, object-oriented
focuses on the things and nouns and aspect-oriented decomposition allows for non-primary functionality to have its own units of functionality.

3. NATURALNESS OF COD
Natural Programming [Myers et al. 2004] is focused on creating development representations that map directly to how developers think, in order to speed development and increase understandability. As such, a natural UIDL better scales with practitioner understanding as it maps to how practitioners think about 3DUIs. In Wingrave [2008], multiple methods were used to gather data about how practitioners thought about 3DUIs. These methods included interviews of multiple 3DUI practitioners from around the globe, a collection and study of development artifacts from the 3DUI community, source code analysis of different toolkits and an experiment that asked developers to describe in pseudocode 3DUIs they observed in movie clips to elicit their internal models of 3DUIs. A Grounded Theory approach [Glaser and Strauss 1967] was employed to extract themes and from this the principles of COD were distilled.

In this section, we discuss the nature of a concept, the principles of COD and the naturalness of COD through an example of the WIM technique. Later, in section 4, we discuss how COD is implemented as a system Chasm.

3.1. Concepts and Component Concepts
Practitioners were able to use loaded terms to describe 3DUIs, even defining them on the fly. As such, a concept was chosen as a unit of development, representing a reusable chunk of functionality – a single software concern [Dijkstra 1982] - an encapsulated cohesive practitioner term. In this way, a concept is designed to be composable and composed with other concepts; to be understood similarly to how communities develop and share a domain vocabulary. Concepts are natural, extracted from language, becoming the unit of discussion and composition in a 3DUI and are ultimately its unit of development.

A Concept-Oriented decomposition unifies different decompositions in a concept. Object-oriented decomposition breaks functionality down by the “nouns” in a system that is readily stored in a single concept. Aspect-oriented decomposition addresses the need to encapsulate non-primary functionality that is spread throughout a system but so too does a concept which inserts its functionality at runtime with the other concepts in a system to maintain the practitioner’s original intent. Functional decomposition can break a system down into methods that can be placed in a concept. While functional decomposition is
susceptible to changing requirements and premature commitment to ideas, concepts improve upon this by having multiple tiers to ease the reordering of functionality (see section 3.2.3 below). Lastly, callbacks and event-based decompositions focus on reacting to events and defining events in a system, a close match to how practitioners think about a system. Addressing this, a concept has 1st class objects to represent causality and event creation which matches to internal flow. Concepts even improve on callbacks and event-based decompositions by maintaining internal state, addressing the issue where a system should respond differently as the system state changes.

3.2. Principles of Concept-Oriented Design

3.2.1. Principle 1: Artifact and Language Focus

Developers are able to conceive of 3D interaction despite the lack of tool support and the growth in complexity. A UIDL built upon practitioner artifacts and language will allow a UIDL to grow in complexity at a rate comparable to developer understanding. Practitioners were able to easily create storyboards or discuss new 3DUIs even though they expressed difficulties developing them. Practitioner journal entries, an example of which can be seen in Figure 1, with diagrams and language were created which described problems and hypothesized solutions to them. Additionally, these language descriptions were seen scaling from very abstract ideas to low-level pseudo code. Despite this, these artifacts quickly lost utility as development progressed to machine representations and designs were iterated on, making the development look less like the early artifact.

Journal Entry Text: “On raycasting click, send to the planet – update ray from freight[er] to each [illegible] planet – dump stack all down”

Figure 1. In this journal entry, a diagram and causal text are used to describe the practitioner’s envisioned behavior of the control of a freighter in an envisioned 3D space transportation game.
3.2.2. Principle 2: Envisioned Behavior

The envisioned behavior attributed to a concept is the “what is to occur”, in a concept. This needs to be represented naturally for the practitioner’s work of decomposing their envisioned behavior into its component concepts. In the investigation, practitioners had fleeting thoughts about their 3DUI that were captured in language or loaded terms. Often, they present this in a journal with an overview statement followed by causal statements that describe how to achieve the envisioned behavior. Committing their envisioned behavior to a representation, such as an artifact or language representation, enabled them to begin the process of development.

3.2.3. Principle 3: Tiered Representation

No single representation is optimal for all practitioner needs but a UIDL with tiered representations enables an optimal representation to be chosen for each information type. Multiple tiers is a means of avoiding the “tyranny of the dominant decomposition” [Tarr 1999], allowing multiple means to separate the concerns of a software system. Practitioners commonly represent their 3DUIs as storyboards, state machines and pseudo code and can discuss their envisioned behavior in conversational language. These representations allow for different levels of design exploration, specificity and change. Unfortunately, these representations are not easily updatable once development has begun, failing the desired quality of flexibility in software. Many times, high-level representations aid development by generating a low-level representation such as code stubs. However useful this is, it is problematic as changes to the high-level representation often change the semantics of the original stubs. Since 3DUI development is iterative, no section of code is ever final and thus code reuse is limited. A round-trip solution is required [Selic 2003].

For a tiered representation to be effective, each tier must be composed of first-class object used for design, development and execution of that concept and other concepts in the system. 3DUI usability is chaotic, where even short informal usage reveals major problems, and thus iterative development is required and must be supported. With a tiered representation, changes are possible at multiple abstractions in the representation and when usability issues are found, the practitioner has intermediate tiers to work through and multiple levels of abstraction to aid their understanding and traceability of changes. Through tiers, a concept’s representation reaches a desirable stratified representation [Abelson and Sussmen 1996].
3.2.4. Principle 4: Longevity

Legacy systems continue to have value to practitioners but over time, legacy systems deteriorate or even fail. Several reasons for this have been mentioned [Steed 2008] such as improved graphics capabilities, lack of documentation for a system, hard coded assumptions, abandoned software toolkits, etc. As such, successful 3DUIs are short lived and constantly reinvented. Despite this, domain knowledge, conventions and terms evolve towards a consensus of meaning. In journals and development artifacts, practitioners are able to easily convey and abstract the details of 3DUIs using loaded terms, small diagrams and short descriptions. A representation based off a community’s domain knowledge and conventions should reuse concepts naturally and create concepts naturally prone to reuse. In this way, in addition to being easy to express 3DUIs, concepts exist to easily develop them, these systems continue to function over time, remain understandable to many practitioners, and through reuse concepts are constantly improved and tested.

3.2.5. Principle 5: Community

A community performs many important functions for a UIDL including maintenance, building robustness, documentation, testing and bug patches, generating new concepts, tool creation and creating a consensus of meaning. In current practice, communities form around tools with practitioners of related work unable to share solutions and successes across tools even though they can readily communicate these ideas in conversation. Concepts should function across tools so large communities can form around ideas, unfettered by tools, and the benefits of a large community can be applied to 3DUIs.

3.3. WIM Example: Natural UIDL

Through an analysis of the WIM technique, we begin to see how COD’s principles can improve design and development. The first step of the WIM’s decomposition is gaining an understanding of what is to be developed. The WIM described by Stoakley and Pausch [1995] was, “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.” However, when an expert was asked to describe it, the description was a flow, “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.” Both descriptions have advantages that Chasm uses in its design and development.

![Diagram](image)

Figure 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 expert’s flow in Figure 2 is useful for its complete understanding of the technique, it has limited utility in development:

- The flow separates related events into distant steps. For example, steps 2 and 6 deal with the button.
- There is no separation between discrete and continuous behaviors, important because they are implemented differently. Steps 2 and 6 are discrete steps dealing with a button while steps 1 and 5 operate over time.
- Alternate paths are not easily identified, important as missing paths create unexpected behaviors. In step 3, the expert did not identify the alternate path that would handle the case if an object were not selected.
- The flow does not decompose the requirements of the hand, button, selection technique and object-to-object relationships. The practitioner’s mental model is never simplified during development.
- Flow is the first to change during the lifetime of a system [Parnas 1972]. Building a system along a flow will most likely result in major changes in the future.

In contrast, a more natural representation allows us to use the abstractions developers understand in language for better decompositions. Stoakley and Pausch’s [1995] original description depended on domain terms such as “3D map” and “hand-held”. The first sentence described the world and implies a 3D map containing proxy objects for the full-scale objects. The second sentence assumed the notion of a virtual hand technique to select and manipulate objects as well as a causal relationship between proxy and full-
scale objects. This description is used in section 4 to create cohesive ideas and detail causal relationships using loaded terms and importantly, it is not a flow description. Later, in section 5, it is shown how execution recreates the flow description and how this is used for understanding and debugging of the system.

4. TIERS OF THE CHASM UIDL

COD was realized in a prototype UIDL toolkit Chasm. Chasm’s tiered UIDL is represented in a simple XML parsable file ChasmXML as well as in code, currently C++. This split representation is achieved through a special preprocessor step and a few special-purpose Chasm command-line tools. During development, both XML and C++ files are modifiable and are combined at compile time into an executable (see section 5 below). As each concept is ultimately a C++ object, Chasm incorporates the full power of object-oriented programming such as encapsulation and inheritance. Inheritance is achieved through a <parent> tag and states, actions, transitions, causal statements, state functions and envisioned behaviors are inherited from the parent concept. As such, Chasm works alongside existing development approaches, tools, environments and methodologies to reduce the barriers of uptake that many UIDLs and other approaches often create [Myers et al. 1999]. This is similar to the design decisions in the toolkit SUIT, a 2D GUI toolkit designed to simplify development for novices by exploiting “knowledge and skills the user already has” [Pausch et al. 1992]. Because of this, the design of Chasm for 3DUIs is like SUIT’s design of 2D GUIs that involved “making dozens of independent design decisions which, when taken in combination, produce a system which is easy for novice GUI programmers to learn” [Pausch et al. 1992].

An example of ChasmXML is shown in Table 1 for three concepts. There are nine tags that represent the tiers of Chasm. Mixed in the representation is also display information for use in visualizing the concept in the ChasmGUI tool [Wingrave 2008].
Table 1. Shown is the example ChasmXML representation for the concept AbstractPositionRelationship, used to implement PositionRelationship in Figure 3. Manipulation of this is by a text editor or the ChasmGUI tool [Wingrave 2008].

```xml
<CHASMConcept name="AbstractPositionRelationship">
  <parent name="vocab/Relationship" />
  <description>This is a relationship between objects based upon their position.</description>
  <component name="origin" concept="vocab/Position" access="PUBLIC" cif="PASSED" x="-225.000000" y="184.000000" />
  <when state="origin.changed" action="isoperating.checkevent" />
</CHASMConcept>

<CHASMConcept name="vocab/Relationship">
  <parent name="vocab/function" />
  <description>This a function which can be turned on and off.</description>
  <component name="isoperating" concept="Switch" access="PUBLIC" cif="CREATED" />
  <when state="isoperating.eventpassed" action="undefine" />
</CHASMConcept>

<CHASMConcept name="vocab/Function">
  <description>This Concept is designed to be extended. It maintains a value and enters the incorrect state when its dependencies change. It is recomputed when explicitly asked for its value. The purpose is to minimize the computation as it only recomputes when needed, not when its dependencies change</description>
  <state name="correct" x="147" y="250" height="30" width="50"/>
  <state name="incorrect" x="50" y="156" height="30" width="57"/>
  <action name="undefine"/>
  <action name="define"/>
  ...
  <transition from="correct" to="incorrect" by="undefine"></transition>
  <transition from="correct" to="correct" by="define"></transition>
  <transition from="incorrect" to="incorrect" by="undefine"></transition>
  <transition from="incorrect" to="correct" by="define"></transition>
</CHASMConcept>
```

4.1. External Concept Tiers

There are four tiers in Chasm: envisioned behavior, causality, automata and code. Of the four tiers of Chasm, the first two represent the external, loose coupling connections between concepts and component concepts while the deeper tiers represent the cohesive internals of a concept. This loose coupling and tight cohesion is a sought after quality in good software [Yourdon and Constantine 1979]. The first tier stores practitioner’s expressed envisioned behavior for a concept as conversational domain language. In this language, loaded terms suggest component concepts. As practitioners explore their ideas about a concept, they begin to understand causal relationships that are expressed naturally in conversational language [Pane et al. 2002; Wingrave 2008], which forms the second tier, the causality tier.
As an example of the naturalness of these tiers, the two-sentence description of the WIM technique [Stoakley and Pausch 1995] follows this division and effectively describes the 3DUI technique. The first sentence is a general description of the envisioned behavior, “A World-in-Miniature is a hand-held miniature 3D map.” The second sentence is a causal statement, “When the user manipulates objects in the map, the corresponding objects simultaneously update in the full scale virtual reality.” The full process of transforming the WIM’s description into something executable in Chasm is covered in section 4.3.

4.1.1. Envisioned Behavior Tier

Chasm’s envisioned behavior tier stores the practitioner’s understanding of a concept as conversational language without having to transform their thoughts to another representation. In ChasmXML, the language is stored in the <description> tag as pure text. In this way, development begins by practitioners thinking through their ideas and writing them down as a means of eliciting the desired envisioned behavior. As the first tier, the closeness of the mapping to developer thinking allows mistakes to be made early, often and in a representation easily correctible. This is similar to CRC Cards that are designed 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” [Horstmann 2005]. The tier’s representation in conversational language also allows concepts to be easily read and understood, upholding the ideals of Literate Programming [Knuth 1992].

Language helps practitioners decompose systems into object and flows. The utility of flow decomposition is shown in practitioner’s use of storyboards, use cases [Cockburn 2001] and scenarios [Rosson and Carroll 2002], which elicit practitioner understanding as a flow of behavior. However, decomposition by flow has been considered a poor choice for system maintainability and extensibility [Parnas 1972]. Decomposition by objects has been shown to produce better systems [Meyer 1997], with nouns often indicating objects in a software system. This has led to further research in understand systems through the use of natural language analysis [Witte, Li, Zhang and Rilling 2008; Gall et al. 2008; Fry, Shephard, Hill, Pollack and Vijay-Shanker 2008]. Chasm relies on the natural ability of practitioners to use loaded terms, nouns and verbs, to express ideas. In this way, language functions as an object decomposition. As communities create common 3DUIs, creating their own specific interaction metaphors and language used to describe it, their
envisioned behavior will use the language, suggesting concepts to create the interaction and in the process encourage concept reuse.

4.1.2. Causality Tier
Chasm’s causality tier stores the next level of understanding during development, causality between concepts. This is stored as causal relationships in statements similar to the statements naturally made by practitioners. These statements are of the form of when/then or while/then. In ChasmXML, the <when> and <while> tags, as shown in Table 1, store this information by defining a context X and response Y to a context being entered. For example, “<when state='X' action='Y'>”. Both the context and response can remain abstract and represented only as a text description or instantiated to a state in one concept and an action in another (see Automata Tier below). Additionally, the causal expressiveness is improved with a Conjunction concept that allows the practitioner to say, “When X₁ and X₂, then Y” or “When X₁ or X₂ then Y.” Changes to the causal statements are readily understandable and easily made, giving the practitioner the flexibility to: 1) respond to a different context for different functionality as the developer understanding changes (ex. a button press versus a voice command), 2) have the concept fulfill a new role in the system, and 3) be reused in new systems with new contexts. Defining this causality assists the practitioner in identifying independent behaviors and concurrently operating flows.

4.2. Internal Concept Tiers
With the external causal relationships defined, practitioners continue into the lower tiers with a now simplified mental model. This internal flow of behavior is represented in the third tier, automata and the fourth tier, code. The automata tier is the internal flow, 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 represent, as well as document and support, the practitioner’s understanding of a concept.

4.2.1. Automata Tier
In Chasm, a state machine was chosen for the automata tier’s representation because it is familiar to many practitioners and it can explicitly define the computation of a system. The state machine of Chasm is a modified Moore machine [Moore 1956] with states, actions and transitions between its states. The automata tier consists of an alphabet of
actions it accepts, $\Sigma$, a set of states it can exist in, $Q$, 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 does not use a separate output alphabet, output function or multiple end states. Instead, each state in Chasm has a state entry and exit function, $\omega$, which is evaluated when the state is entered or exited. State functions are implemented in the code tier and have the option of returning an action in $\Sigma$, 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).

The automata tier represents the internal flow of a concept’s behavior. As the causality tier maps external stimuli to internal actions, the automata tier is simplified and free to focus only on internal behavior. In this way, the flow of the single concern can be explored independent to the system’s flow. This greatly reduces the complexity of a concept and exhaustive consideration of every action in each state becomes tractable. In this way, unexpected behaviors at runtime are reduced.

### 4.2.1.1. States

A state in the automata tier is a named step in the internal flow of a concept as well as serving as the context of causal statements. In ChasmXML a `<state>` tag defines a state and is named by a name attribute and contains a description. Generally, a state’s meaning seldom changes in a concept, making the state a good context for a causal statement. Additionally, as a state is the representation for the internal flow, no additional development and maintenance time needs to be spent creating or maintaining “hooks” [Polys and Ray 2006] to externalize the concept’s functionality for reuse. In this way, a concept can be extended by directly attaching to its internal flow. Additionally, a state can exist as a decision state, executing a state function when the state is entered and returning an action to the concept.

The granularity of the states used in a concept depends upon the needs of the developer. While a state machine is able to represent many fine-grained details of a flow, only those states that are useful and helpful should be in the automata tier. Other information that is easily represented in code as a variable should be represented as code in the code tier.

### 4.2.1.2. Actions

An action drives the flow in the concept from its existing state to a new state. As with the `<state>` tag, an `<action>` tag has a name attribute and a description. One of the most
important aspects of an action is that an action only has local scope. This greatly reduces a concept’s complexity as it is sealed from outside influences. Additionally, an action is atomic, having no additional information associated with it such as a float or string. This reduces data structure dependencies between concepts in addition to simplifying Chasm’s representation. Such information is easily attainable in the code tier through method calls where existing software engineering abstractions can encapsulate data.

Actions can occur in three ways. First, an action can occur through a causal statement that maps a context in the system to an action in this concept. Second, an action can be returned by a state function, acting as a decision state discussed above. Finally, the code tier can inject an action with a method call and in this way the automata and code tiers are bridged.

4.2.1.3. Transitions

Transitions are valid paths out of a state to another or the same state caused by an action. They are defined in ChasmXML as a <transition> tag, containing attributes for from, to and by, referring to the transition’s start state, a destination state and the action of the transition. 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. A transition for ignoring an action in a state can also be defined.

4.2.2. Code Tier

The code tier is a single file containing ten different sections of code. Two of these sections deal with the builder (discussed below), one to define debugging output of a concept, two for C++ header include files, one for header defines, one for constructor parameters, two for constructor and destructor code and a final section for general code such as state functions and other C++ methods. At compile time, these sections are merged with the ChasmXML. Preprocessing checks, compile-time checks and code generated for runtime checks look for missing state entry functions, missing passed parameters, non-initialized concepts and other errors that can occur that would otherwise generate abnormal runtime behavior.

Code was used as the final tier since procedural programming is very effective at listing instructions to execute and developers are familiar with it. Additionally, practitioners have existing coding practices and many tools and toolkits to support their
development. In this way, the strengths of procedural programming are maintained inside Chasm while the higher tiers of Chasm address the weaknesses.

There are two important issues in the code tier: builders and CIF.

4.2.2.1. Builders

A builder is a useful code tier abstraction named after the Builder pattern [Gamma et. al. 1994] that separates the concept from toolkit-specific or project-specific code. This is achieved by the original concept forwarding method calls to a separate builder object. In this way, the concept is portable to new toolkits by only requiring the reimplementation of the builder for the new toolkit without also managing the code tier’s functionality as well. It also allows the concept to be simpler, focusing on its core functionality, and systems implemented using builders of one toolkit are swappable for another toolkit’s builders. For example, a button concept could be used in a system implemented on top of the DIVERSE [Kelso et al. 2003] toolkit but its builder could be swapped so as to function in VR Juggler [Bierbaum et al. 2001]. In this way, a system is ported to a new toolkit, extending the longevity of the system. In addition, builders support the Community principle, as toolkits no longer divide practitioners. An observation of this approach is that as higher-level concepts are created, less toolkit specific code is required implying that higher level functionality might not even require builders.

4.2.2.2. Components and CIF

Component Initialization Form (CIF) is specified for each component concept to indicate how it is initialized and in some cases deleted. The four types of CIF 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 created by the concept during the initialization process, be it in the builder or concept constructor. As such, the lifetime of the component expires when the concept’s lifetime expires. In potential CIF, the component is external and potentially passed in while in vacant CIF, the component is external and starts as null. In both potential and vacant CIF, these components will be assigned and managed by the practitioner’s code and not automatically by Chasm.

CIF has several useful functions. First, it is useful for performing runtime checks to catch common development errors. This includes the errors of forgetting to pass a component concept in the constructor as well as forgetting to initialize the component. Second, CIF is used to manage the lifetime of a concept, expiring when the parent
concept’s lifetime expires unless the component is used by other concepts. Without CIF, abnormal runtime behavior could be created with errors difficult to fix as the error is created but only realized later.

4.3. WIM Example: Tiers
The four tiers of COD in Chasm for the WIM are shown in Figure 3. In this concept, the natural description of [Stoakley and Pausch 1995] is placed in the envisioned behavior tier. This informs the deeper tiers with a high-level, easily understood, language description. From this description, the first realization 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 in the World In Miniature (WIM) concept as the concepts of virtual hand (VH) and position relationship (PR). Because these are common ideas in the domain, they have existing concepts that can be reused.
Figure 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 as shown in Figure 2.

The next tier is causality with the causal statements shown in Table 2. 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 concept and then represents that understanding as short causal statements. These statements are then directly connected to states and actions of concepts (see Table 2). At this point, the developer can focus solely on the internal flow of the WIM.
Table 2. In the WIM example, the natural causal statements are connected to states and actions in concepts. The language description of the causal statement is on the left and on the right, the causal statements is shown connected to states and actions.

<table>
<thead>
<tr>
<th>Natural Causal Statement</th>
<th>Conceptual Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>When &quot;grab&quot;, then &quot;manipulate&quot;.</td>
<td>When VH.newselected, then WIM.create_relationship.</td>
</tr>
<tr>
<td>When &quot;release&quot;, then &quot;drop&quot;.</td>
<td>When VH.unselected, then WIM.end_relationship.</td>
</tr>
</tbody>
</table>

The automata tier represents the internal flow of the WIM technique. Where the decomposition by flow in Figure 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 concurrently are handled by the virtual hand and the position relationship concepts. This greatly simplifies the WIM concept.

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 clearer. 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. If more functionality is required of the VH or PR concept, a new concept can be created that extends or modifies the existing concept. As such, no concept is every sealed-off from modification or improvement.

5. EXECUTION OF THE CHASM UIDL

The UIDL of Chasm, in addition to being natural and tiered, was designed around three tenets in its execution:

- Directly executable
- Preserved practitioner intent
- Integrated with existing tools and practice

By directly executable, it is meant that Chasm’s UIDL has no intermediate representation that requires further work by the practitioner. For example, there is no generation of stub files for code to be placed in or a compilation step from one tier to the next. Because of this, the practitioner’s model is not radically changed or reorganized by Chasm as they work such that the model remains consistent and understandable. In this
way, high-level changes necessitated by iterative development or design exploration can be made with minimal effort and minimal disorientation on the part of the practitioner.

By preserving practitioner intent, it is meant that concepts continue to function according to the original intent of the practitioner even as additional concepts are added to a system or the concept is reused in new systems. Systems like Statecharts and its variants have been criticized for problems with its syntax that led to event ordering issues [von der Beeck 1994] as well as not being sealed due to action global scope [Wingrave 2008]. To address this, Chasm orders events at runtime according to the action-processing dictum and the ordering rules that follow from it (see Sections 5.2 and 5.3). While the tiers function to extract and then represent developer understanding, much of the utility would be lost if unexpected behaviors and system errors result that necessitate the modification of the existing concept or requiring the practitioner to tailor the concept for each system in which it is used. Concept invariance supports the longevity principles of COD.

By integrating with existing tools and practice, it is meant that as much as possible, Chasm should make use of the tools and practices that currently support practitioners as well as lowering the barriers of entry to using Chasm. To achieve this, Chasm executes actions in event cascades and discretizes continuous behaviors so that processing can return to the system. An event cascade refers to an event being generated in a Chasm system, the processing of that event and all the events that occur as a direct response to first event and are processed in it. These cascades can begin by a procedural call to Chasm, in a Chasm event loop or by integration into an existing event loop of another architecture, for example an OpenGL loop. In this way, Chasm remains architecture agnostic and functions with a wide array of potential programming paradigms. The principle of community is supported by this integration.

Below, the action-processing dictum is discussed as well as the ordering rules that follow from the action-processing dictum. Additionally, how Chasm implements these rules is covered. Following this, the special cases that arise and must be handled by Chasm to maintain the tenets above are discussed.

5.1. Action-Processing Dictum
The action-processing dictum is designed to manage the potential change to stateful data when an action is processed. If stateful data is allowed to change before the intended actions of the original developer are satisfied, the concept’s behavior may change. In Chasm, stateful data refers to both automata tier states and code tier variables. As such,
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 rules and the data structures and algorithms enforcing them. Successful enforcement removes the burden of ordering events from the developer. However, not all cases can be automatically handled so in cases where two or more actions are possible, the potential for error should occur in the more recently added concept, so as to keep errors near the practitioner’s focus.

Consider this trivial example: a button increments a counter when pressed and when the counter updates, it notifies the display of the counter. In this example, the counter’s value must update before the display redraws so the display knows what to draw. If new functionality updates the counter without notifying the display or causes the display to update before the counter increments, the original developer intent is broken.

### 5.2. Event Types and Ordering Rules

The four ordering rules of the action-processing dictum maintain the proper order of the five event types in Chasm. An action can be in one of the following Chasm event types: 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 event type is the normal state machines event.
- An *invoked event* occurs when the code tier causes a change that requires notifying the automata tier with an action. The event must be handled immediately as keeping with the developer’s expectation.
- *Continuous events* 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 event cascade. In this way, continuous behaviors such as animations are discretized and gain the appearance of continuous operation. A future event is an event set to occur at a point in the future, delaying its function, and executes at a specific time, for example, in three seconds.

The ordering rules for these events as well as examples of their handling are covered below. For these examples, the notation for a state or action of a concept is the concept’s name followed by a period followed by the name of the state or action.
5.2.1. Rule 1. Partially Ordered Responses

In the action-processing dictum, causal statement responses for a single state happen in a partially specified order. Chasm assumes this order to be that of the initialization order. 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 events will be executed in the partial order they were requested: B.b, C.c and D.d (see Figure 4). Chasm also has functionality to reorder responses but these are on a per concept basis and have only been used in the creation of the specialized Gather concept (see section 5.4.3). Chasm has no means at this time to specify the order to be non-deterministic.

![Figure 4. The order of events on entry to A.state is (B.b, C.c, D.d).](image)

5.2.2. Rule 2. Responses Before Response

A response's responses have higher priority than previous responses, ensuring that a flow of behavior completes before returning to another flow. 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.

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 rationale for this rule 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.
Figure 5. The order of events on entry to A.state is (B.b, D.d, C.c).

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

- **Case 1:** Consider a fully tested and released system existing only as A, B and D. 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. after B.b and D.d), any changes C makes to the system are irrelevant to the existing functionality as they were 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. 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 process before D.d. However, D can change state in C before C.c is handled (because D.d 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.

### 5.2.3. Rule 3. Responses Before Continuation

A state entry function can return an action to itself as a continuation event that occurs after all existing 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 is a decision state that 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 6.
Figure 6. 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 in the expected state. Consider a system A, B, D where C is added (as in Figure 6). 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 rule 2, responses before response, C.c could incorrectly run before B.b or D.d. This would potentially allow changes made by C to affect B and D. Therefore, processing the response events (B.b and D.d) before the continuation event A.a means that the system will continue to function as it did before C was added to the system.

5.2.4. Rule 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 7 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. Additionally, this is the expectation of developers working in procedural programming.
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 parentheses 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 any existing response events that are yet to be handled. In Chasm, this is addressed by using a new event cascade for the invoked event. In this way, the state of the current event cascade is untouched and can be returned to. In Figure 7, 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.state was entered.

5.2.5. Rule 5. 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 and given a time at which to mature. If continuous events or future events were not processed on a separate event cascade, the system would behave without regard for other system functionality, never sharing time to allow other actions to process.
5.2.6. Final Ordering

A final example in Figure 8 shows how the different events 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).

![Diagram showing event ordering](image)

Figure 8. 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. Implementing the Order

Ordering the events in the discrete and linear timelines is managed in Chasm through three stacks, an ordered list and two algorithms. In the discrete timeline, the cascade stack maintains the current system’s cascades. Each entry in the cascade stack contains a response and continuation stack to hold response and continuation events waiting to process in that cascade. The algorithm to select the event to process is shown in Table 3 as GetNextEvent(). The temporal timeline uses the future list, a list sorted by each event’s maturation time, to store events waiting to be processed. When the future list is checked, all mature events are removed and added to the continuation stack in a new cascade entry. Processing of an event, regardless of the event type, is handled in a similar fashion as shown by the ProcessEvent(…) algorithm in Table 3.
Table 3. The algorithms in pseudocode below: left, select the next event to process in the discrete timeline and right, execute the action in an event.

<table>
<thead>
<tr>
<th>GetNextEvent() : Event</th>
<th>ProcessEvent(event, currentcascade)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>flag = false</td>
<td></td>
</tr>
<tr>
<td>do { # get a cascade with events</td>
<td></td>
</tr>
<tr>
<td>cascade =</td>
<td></td>
</tr>
<tr>
<td>CascadeStack.TopEntry()</td>
<td></td>
</tr>
<tr>
<td>if cascade == null</td>
<td></td>
</tr>
<tr>
<td>return null</td>
<td></td>
</tr>
<tr>
<td>else if cascade.HasNoEvents()</td>
<td></td>
</tr>
<tr>
<td>CascadeStack.Remove(c)</td>
<td></td>
</tr>
<tr>
<td>else flag = true</td>
<td></td>
</tr>
<tr>
<td>} while flag == false</td>
<td></td>
</tr>
<tr>
<td>if !cascade.ResponseStack.IsEmpty()</td>
<td></td>
</tr>
<tr>
<td>retval = cascade.</td>
<td></td>
</tr>
<tr>
<td>ResponseStack.TopEvent()</td>
<td></td>
</tr>
<tr>
<td>else # assume not empty</td>
<td></td>
</tr>
<tr>
<td>retval = cascade.</td>
<td></td>
</tr>
<tr>
<td>ContinuationStack.TopEvent()</td>
<td></td>
</tr>
<tr>
<td>return retval</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>newstate = event.Concept.</td>
<td></td>
</tr>
<tr>
<td>Transition(event.Action)</td>
<td></td>
</tr>
<tr>
<td>if newstate == null,</td>
<td></td>
</tr>
<tr>
<td>generate an error message</td>
<td></td>
</tr>
<tr>
<td>and quit</td>
<td></td>
</tr>
<tr>
<td>continuationaction =</td>
<td></td>
</tr>
<tr>
<td>newstate.StateEntryFunction.</td>
<td></td>
</tr>
<tr>
<td>Execute()</td>
<td></td>
</tr>
<tr>
<td>if continuationaction != null,</td>
<td></td>
</tr>
<tr>
<td>currentcascade.</td>
<td></td>
</tr>
<tr>
<td>ContinuationStack.</td>
<td></td>
</tr>
<tr>
<td>Add(continuationaction)</td>
<td></td>
</tr>
<tr>
<td>currentcascade.</td>
<td></td>
</tr>
<tr>
<td>ResponseStack.Add(</td>
<td></td>
</tr>
<tr>
<td>newstate.</td>
<td></td>
</tr>
<tr>
<td>CausalResponseActions )</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

5.4. Special Cases

Special cases still exist in the execution of events. These can potentially lead to practitioner difficulties and an invalidation of a concept's behavior. Where possible in Chasm, these cases are handled and moderated.

5.4.1. Cycles

Chasm problematically creates new ways to create non-halting cycles that affect it and the systems in which it integrates. Examples are failing to properly discretize a behavior over time [Blom and Beckhaus 2007] or creating mutual data dependencies though causal statements. Identifying and correcting these cycles would be difficult without Chasm's support, especially when the cycle occurs across reused concepts or even cascades. Therefore, to be developer friendly, Chasm assists developers with the four cycles below.

In the discrete timeline, three cycles exist. The continuation and response cycles are an infinite loop of events in a single event cascade. Continuation cycles are created when state functions return actions that eventually lead back to the original state. Response cycles are similar to continuation cycles but response events, along with continuation events, create loops involving multiple concepts. Cascade cycles occur when a state function invokes an event causing a new cascade which, itself or during the processing of the cascade, recreates the same cascade. These cycles are commonly caused by mutually dependencies or improper implementation of dynamic behaviors. Identification of these
cycles is achieved by entering each processing state into a cascade's state stack. If during the processing of a cascade, a state is re-entered, then a cycle is identified. However, cascade cycles are not identifiable in this way, as the state is re-entered in a different cascade. While not implemented, checking if the current state is the current state in a lower cascade identifies this. Because decision states can return different actions due to state data in the code tier, all of these checks are susceptible to false positives.

In the temporal timeline, maturation cycles can 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, never returning to other events in the system. Chasm addresses the maturation cycle by maturing events as an atomic action in a cascade. As such, all mature events finish processing and execution returns to the system before another check for mature events can occur. Developers can still mistakenly schedule events to process before the current event is finished but it does not delay execution of the system. In many systems, timing is critical and plenty of research exists on achieving real-time behavior [Alur and Dill, 1994].

Due to these cycles, concepts and notations were added to support the developer. A Resource concept was created to enforce mutual exclusion of data. Additionally, the ability for state functions to return actions that occur “at the next possible time” was developed using continuous events. This was incorporated into the Animation concept that is used to discretize behavior over time. Runtime tracing was also added. Though cycles were at first common and difficult to identify, these steps reduced the prevalence of cycles.

**5.4.2. Encapsulation Assumption**

Much of the invariance of a concept's function stems from the assumption that a concept fully encapsulates its stateful data. Though this is clear in the automata tier, where the current state fully describes the automata’s state and the system’s state is the combination of every concept’s state, the assumption fails in the code tier. This is because the code tier is implemented in an imperative language with variables and global data. The effect being that stateful data outside the concept, and not accounted for in the representation, could affect the functioning of a concept and enter it into a state for which it was not intended or tested. If changes to state in the code tier should change the state of the automata tier, the appropriate action should be inserted into the concept.

Chasm places the burden on developers to properly encapsulate their concepts. This is a typical burden for developers and Chasm supports C++’s encapsulation and inheritance
mechanisms. Additionally, Chasm’s tiers should help encapsulation by creating simpler concepts, designed using the envisioned behavior and causality to create loose coupling and automata and code for tight cohesion. Finally, through reuse, concepts become robust through more testing and under differing conditions of use.

An additional state change that can break the intended functioning of a concept is premature destruction. For example, because a concept can be reused in new situations, its original lifetime as determined by its CIF might be extended due to new requirements. For example, if a concept A’s original lifetime ends, it is marked for destruction and placed in its end state. If concept A is used by concept B and concept A’s C++ destructor is called, concept A would no longer provide the needed functionality for B. Therefore, Chasm partially manages the deletion of concepts and only destroys concept A, i.e. calls the C++ destructor, when no further concepts, such as concept B, use the concept. By partially managing deletions, additional concepts to a system will be assured the concepts they rely on will remain available.

5.4.3. Gathers

It is possible that a state can serve as the context for multiple causal statements that lead to multiple actions entering another concept. Unmediated, this diamond relationship\(^2\) in the best case wastes computation by handling multiple actions and in the worst case produces unexpected and difficult to debug system behavior, improper transitions or even system termination. Unfortunately, the problem is quite common.

An example can be taken in the VirtualHand concept in Figure 3. Typically, feedback during selection is used to indicate the potentially selectable object such as highlighting the object or drawing a bounding box around it. This feedback needs to be checked for whenever a selectable object moves or the user’s hand moves. Without a Gather, if both the object and hand move, the highlighting is applied twice and can crash the system.

A Gather concept (see Figure 9) addresses this problem and is automatically entered by Chasm when identified at runtime. A Gather functions by routing the offending causal relationships into itself, waiting for the causal statements to finish and then sending the action once, as in Figure 9. As these checks occur at runtime when causal statements are connected, and scale according to the depth of the causal statement hierarchy, there is a potential performance bottleneck. To optimize this, a secondary data structure stores every state that leads to an action, reducing the overhead checks to only one level of depth.

\(^2\) Not to be confused with the diamond problem in multiple inheritance.
By using a Gather, the problem of the action D.d being sent by both B.state and C.state when A.state is entered is resolved. This is achieved by wrapping the causal statements and a Gather sending D.d once when both B and C are finished.

5.5. WIM Example: Execution

In Figure 10, we see how the WIM concept from Figure 3 is executed to reestablish the flow of behavior in Figure 2. In this way, COD in Chasm assists the practitioner creating the WIM concept in a natural and tiered manner and Chasm’s execution recreates the originally understood flow. Recreating this flow is useful for tracing and debugging purposes.

COD in Chasm is also useful in the extension and reuse of the WIM. One modification of the WIM concept would be to add the scaling and scrolling functionality of the SSWIM technique [Wingrave et al. 2006]. The WIM concept requires no modification as the SSWIM concept can respond directly to the WIM’s manipulating state, to know when to check for scrolling, and a mouse concept can be added to control the scaling of the WIM.
Figure 10. The processing of events in Chasm recreates the expert’s description of the flow as in Figure 2 (the numbers on the right relate to the steps in the flow) and allows development as a cohesive concept as in Figure 3.

6. EVALUATION AND CURRENT USE

Evaluation of software systems is difficult and often subjective, relying on criteria sets or trivial examples for evaluation. Direct comparisons to existing systems are made difficult by the effect of user experience and tool maturity found in existing approaches as well as costs involving expert’s time. An analysis of evaluation approaches in software engineering systems and their limitations can be found in Shaw [2001]. As such, several evaluation approaches were used for Concept-Oriented Design in Chasm. These evaluations made heavy use of the Cognitive Dimensions of Notations [Green and Petre 1996] as it presents a standardized criterion set to judge Chasm. The first evaluation was a domain expert evaluation. The second evaluation was an analysis of ten case studies created by long-term use by multiple practitioners. Finally, due to a lack of rigorous quantitative analysis of Chasm’s model, we created a form of analysis termed analysis of spaces. Through analysis, the case studies are quantitatively compared as if implemented in naïve, hierarchical and Chasm’s models. The full evaluation of Chasm is in [Wingrave 2008].

6.1. 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. The five experts were all associated with the 3D Interaction Lab at Virginia Tech, but not this project, and had a
wide variety of experience regarding 3DUI toolkits. In the evaluation, they were given instructions on Chasm and then followed a step-by-step script in the implementation of a Chasm system. The entire process took roughly two hours for each domain expert. Their comments were collected throughout the evaluation and at the end as a semi-structured interview using the Cognitive Dimensions Questionnaire [Blackwell and Green 2000]. They were instructed to use their domain experience to compare Chasm to the tools and methods in which they were familiar. Comments were generally positive, including, “You can put everything out there and figure out how you want to string it together,” “You capture [the implementation] while you think of it,” and regarding Chasm development, “You think sequentially. You think, ‘When I do this, the system does that.’” The experts also made several comments regarding Chasm’s steep learning curve but offered one expert, “I don't know if I can pick out anything that was stranger or harder than other [notations].”

6.2. Long-Term Evaluation
A long-term evaluation by four practitioners created ten case study Chasm systems. Two students used Chasm for a semester project in a graduate course on Virtual Environments, a third used Chasm to implement their thesis project over a period of one year and the fourth was the primary author. Some of these case studies involve development of the more complicated 3DUI techniques in the literature such as HOMER [Bowman and Hodges 1997] and Voodoo Dolls [Pierce et al. 1999]. The developers were supported by the primary author and observed on multiple occasions in their use of Chasm. Additionally, a final interview using the Cognitive Dimensions Questionnaire [Blackwell and Green 2000] was used.

In general, the practitioners were happy with Chasm. Some of their comments included, “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,” and “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.” Several occurrences mentioned by practitioners in the case studies also support Chasm’s approach to 3DUIs. In one example, a long-term practitioner commented about their adding a new requirement to a concept late in their development due to a design iteration. They added a state and several transitions to a concept and to their surprise, all existing functionality continued to work. In another example, a long-term practitioner was asked about Chasm’s event ordering. They didn’t understand it but
were curious because they realized it could be problematic. They were unconcerned however; saying it just did what they wanted. Other long-term practitioners didn’t understand the issue regarding event ordering, as they had never had difficulties with it. A final example of Chasm’s utility was through the observation that practitioners used Chasm’s approach, diagramming style and representations in the planning of other projects outside the evaluation. While they did not use Chasm’s tools in these projects, it can be seen that their use of Chasm fundamentally changed how they approached design and development of software. This implies a wider applicability for Chasm outside only 3DUIs.

The case studies also showed the ability of developers to reuse concepts and to create new concepts from scratch and through inheritance. In the case studies, twenty-five concepts were created by the long-term developers and twenty-three by the primary author. With one main concept for each case study, this totaled eighty distinct concepts. Of these eighty concepts, thirty-three used inheritance and sixty-one held component concepts. There was an average of 38.0 concepts, with 18.5 of them unique, in each case study. Removing the ten case study concepts and seven false-start concepts (those not used in any final implementation), sixty-three 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 four or more case studies (15 of 63) with five concepts used in all case studies (7.9%).

6.3. Analysis of Chasm by “Spaces”

Lacking a rigorous quantitative claim of improvement over other models, we measured aspects of each concept’s automata tier to analyze the “space” of each case study. In this way, we verify our claim of a reduction of complexity using Chasm’s model. 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, and thus more work, needs to be performed to exhaust all possible states/actions interactions in the model.

The three spaces created are the state space ($S_{sp}$), action space ($A_{sp}$) and interaction space ($I_{sp}$) of the automata tier. 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. Of the three spaces, we posit that the interaction space is a better representation.

---

3 These numbers vary slightly from [Wingrave and Bowman 2008] due to the collection issues discussed in [Wingrave 2008].
of development complexity as it is a closer measure of the number of decisions a practitioner must make during development.

These spaces are in turn representable by different models: Chasm’s model (C), a hierarchical model (H) such as Statecharts [Harel and Peliti 1998] and a naïve model (N) such as a single state machine. Though the ten case studies discussed above were implemented in a Chasm model, the other models can be computed from Chasm’s model (see below). In this way, the case studies are comparable as if modeled in each of the three listed models.

Computing these models from a Chasm model is generally performed as follows (a full discussion can be found in [Wingrave 2008]). Both Chasm and hierarchical models collapse the \( S_{sp} \) in a similar fashion, retaining the original representable state space by allowing the state machine to exist in multiple states simultaneously, sometimes called and-states [Harel and Politi, 1998]. As such, \( HS_{sp} \) and \( CS_{sp} \) are the sum of all the states in concepts in a system. Computing \( NS_{sp} \) from a Chasm model means expanding the and-states. For \( A_{sp} \), computing \( NA_{sp} \) or \( HA_{sp} \) is achieved by counting all actions in the Chasm system. Because Chasm actions have limited scope, \( CA_{sp} \) is only the number of actions internal to a concept. For the \( I_{sp} \), \( NI_{sp} \) is then the product of the actions and states in a system, \( HI_{sp} \) is the product of each concept’s state and the system’s actions and \( CI_{sp} \) is the sum of the product of each concept’s state and action. In practice, hierarchical models have only a small number of actions with impact outside of their level, due to the cohesion in the hierarchy. This only reduces, and does not exclude, the possibility for an interaction and thus when interactions occur, it is difficult to identify [von der Beeck, 1994] in a way similar to problems with variables of global scope [Wulf and Shaw 1973].

Table 4. The ten case studies showing their interaction spaces in naïve, hierarchical and Chasm models.

<table>
<thead>
<tr>
<th></th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
<th>#9</th>
<th>#10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( NI_{sp} )</td>
<td>6.57 ( \cdot 10^{24} )</td>
<td>2.90 ( \cdot 10^{17} )</td>
<td>8.64 ( \cdot 10^{7} )</td>
<td>7.72 ( \cdot 10^{8} )</td>
<td>4.61 ( \cdot 10^{8} )</td>
<td>1.09 ( \cdot 10^{8} )</td>
<td>6.30 ( \cdot 10^{4} )</td>
<td>8.10 ( \cdot 10^{3} )</td>
<td>2.36 ( \cdot 10^{6} )</td>
<td>2.27 ( \cdot 10^{8} )</td>
</tr>
<tr>
<td>( HI_{sp} )</td>
<td>40200</td>
<td>17030</td>
<td>2550</td>
<td>4422</td>
<td>3480</td>
<td>41808</td>
<td>812</td>
<td>1332</td>
<td>1806</td>
<td>3248</td>
</tr>
<tr>
<td>( CI_{sp} )</td>
<td>734</td>
<td>399</td>
<td>144</td>
<td>219</td>
<td>189</td>
<td>772</td>
<td>100</td>
<td>130</td>
<td>154</td>
<td>191</td>
</tr>
</tbody>
</table>

Using this analysis on the case studies, we find hierarchical models to greatly reduce the \( I_{sp} \), but Chasm reduces it further still. Comparing the hierarchical and Chasm reduction in the case studies of the naïve interaction space shows this (see Table 4). The
median fraction of the hierarchical model \((HI_{sp}/NI_{sp})\) is \(1.09\times10^{-5}\) versus the still smaller median fraction of the Chasm model \((CI_{sp}/NI_{sp})\) that is \(6.26\times10^{-7}\). To demonstrate the size of these numbers, a team of 100 developers exhaustively considering the \(NI_{sp}\) of the largest case study would require, if each interaction took 1 second, \(3.17\times10^{19}\) years working every hour of every day. Even with a hierarchical model, it would require roughly 12 tedious hours for a single developer. \(CI_{sp}\) however would require thirteen minutes of consideration. At this point, it becomes a tractable option, especially considering Chasm’s case studies showed 45% concept reuse.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have discussed Concept-Oriented Design’s implementation in Chasm as a natural, tiered and executable UIDL. The use of a UIDL as an approach to 3DUI design and development, an approach that has failed in the past for traditional interfaces, is now showing promise and Chasm is maturing as a UIDL tool for 3DUI practitioners. The naturalness of COD in Chasm was achieved through an investigation of existing 3DUI practitioners and distilled into the five principles of COD: artifact and language focus, envisioned behavior, tiered representation, longevity and community. The tiers of Chasm assist developers in developing concepts, with the first two tiers documenting and representing external influences in the concept and the final two tiers its internal flow and executable C++ code. These tiers are represented in ChasmXML and code files, making use of existing developer practice and tools. The execution of the UIDL then proceeds by following the rules stemming from the action-processing dictum while remaining directly executable, preserving practitioner intent and integrating with existing tools and practice. COD in Chasm has shown great promise both in the ability to assist design and development, concept reuse and in containing the complexity of the system. This was supported by multiple evaluations and an analysis of spaces.

Guided by the principles of COD, 3DUI development in Chasm is continuing to be improved. In its current form, Chasm successfully assisted developers in the creation of non-trivial 3DUIs in classes, research [Wingrave and Bowman 2005; Bowman et al. 2007] and in academic achievement [Badillo 2007; Wingrave 2008]. As such, it has moved beyond a trivial academic implementation and future work will be to refine its use, provide guidelines to assist practitioners during development, improve its tools, add to the vocabulary and expand its applicability to other domains and tasks. There is also a great potential for automated parallel and distributed execution in Chasm that will be
explored as well as means to automate the interpretation of practitioner’s envisioned behavior.

ACKNOWLEDGMENTS

Many thanks go to the long-term developers, the usability specialists, the donators of development artifacts, the interviewees, the reviewers and the domain experts.

REFERENCES


Received January 2009; revised; accepted.