Context-driven interaction in immersive virtual environments

Virtual Reality

ISSN 1359-4338
Volume 14
Number 4

Virtual Reality (2010)
14:277-290
DOI 10.1007/
s10055-010-0178-2

Springer

Virtual Reality (2010) 14:277–290
DOI 10.1007/s10055-010-0178-2

ORIGINAL ARTICLE

Context-driven interaction in immersive virtual environments

Scott Frees

Received: 6 August 2008 / Accepted: 28 October 2010 / Published online: 16 November 2010

Abstract There are many interaction tasks a user may wish to accomplish in an immersive virtual environment. A careful examination of these tasks reveals that they are often performed under different contexts. For each task and context, specialized interaction techniques can be developed. We present the context-driven interaction model: a design pattern that represents contextual information as a first-class, quantifiable component within a user interface and supports the development of context-sensitive applications by decoupling context recognition, context representation, and interaction technique development. As a primary contribution, this model provides an enumeration of important representations of contextual information gathered from across the literature and describes how these representations can effect the selection of an appropriate interaction technique. We also identify how several popular 3D interaction techniques adhere to this design pattern and describe how the pattern itself can lead to a more focused development of effective interfaces. We have constructed a formalized programming toolkit and runtime system that serves as a reference implementation of the context-driven model and a discussion is provided explaining how the toolkit can be used to implement a collection of representative 3D interaction interfaces.

Keywords Human–computer interaction · Context-sensitive interaction · Virtual reality · Virtual environments, 3DUI, interaction techniques

S. Frees (✉)
Ramapo College of New Jersey, Mahwah, NJ, USA
e-mail: sfrees@ramapo.edu

1 Introduction

A recognized interaction problem in immersive virtual environments is that users have difficulty manipulating virtual objects precisely. There are good solutions to this precision problem, many of which place constraints on the movement of objects (limiting the speed of motion, snapping to grid points, etc.). Of course, if the user does not desire precision, these same constraints can inhibit interaction and become a nuisance. Put simply, there are two *separate and competing* contexts in which users interact with virtual objects—one requiring free flowing, rapid manipulation, and another requiring a constraint to support precision.

This issue is not unique to object manipulation. Selection of proximate objects is typically straightforward; the user intersects the cursor or stylus with a virtual object and performs some action (perhaps a button press) to gain control of the object. Difficulties arise when the user wants to select objects outside arms' reach. Instead of forcing the user to navigate to distant objects, selection techniques exist that allow users to specify an object from a distance (Bowman and Hodges 1997; Forsberg et al. 1996; Pierce et al. 1997; Poupyrev et al. 1996). These techniques all share a common characteristic; they *add* to the interface a specialized method of selection suitable for selecting *distant* objects and, in most cases, provide the more natural direct-touch method when selecting near objects. Once again, there are two *separate and competing contexts* under which users select objects: one where objects are within arms' reach and one where objects are far from the user.

When confronted with this situation, developers are forced to make a decision; they can (1) implement an interaction technique that attempts to support both contexts or (2) implement two interaction techniques and force the

user to choose between them while interacting in the environment. Implementing an interaction technique that supports both contexts may seem ideal; however, this solution often leads to a monolithic technique, which supports neither context very well, degrading to a least common denominator solution. The second option has its own drawback, as explicit mode switching between interaction techniques disrupts natural interaction.

Missing from the current model of interaction is a formalized method of representing the current *context* under which interaction tasks are performed. Given a sufficient model, developers would be better equipped to create sophisticated context recognition (perhaps limiting the need for explicit system control) and specialized interaction techniques while maximizing code reuse and portability. This paper presents such a model. We present several context components as a means of fully specifying the current interaction context. The idea of using separate, independent context recognition mechanisms, both implicit and explicit, is introduced. Finally, a design pattern for developing context-sensitive user interfaces is defined and a programming toolkit built on top of SVE (Kessler et al. 2000) is presented. The result is a model that encourages developers to focus on creating highly effective and specialized interaction techniques for specific contexts rather than attempting to adequately support all contexts with one monolithic technique.

2 Related work

The notion that context, or the "current situation", affects the way we perform tasks (and consequently, our choice of interaction technique) is neither new, nor groundbreaking. In the real world, we constantly make these decisions—while both a bicycle and automobile can take you from place to place, contextual information (weather, time constraints, and distance to be traveled) usually makes the choice obvious. This principle holds in virtual worlds as well and has long been implicitly recognized by 3D user interface researchers (as well as for general UI design, as suggested by Nardi (1996). Early in virtual reality research, it was understood that selecting objects from a distance requires a fundamentally different approach than when selecting a nearby object—and thus the development of many interaction techniques to fill that void (Bowman and Hodges 1997; Forsberg et al. 1996; Pierce et al. 1997). Likewise, radically different interaction techniques have been developed to support rapid object manipulation (Poupyrev et al. 1996; Stoakley et al. 1995) versus precision techniques (Beir 1990; Frees et al. 2007; Albinsson and Zhai 2003; Ruddle and Jones 2001). A similar variety of interaction techniques have been created for navigation (Pierce and Pausch 2004; Tan et al. 2001; Usoh et al. 1999) and other interaction tasks. The key theme throughout the literature is that *specific* interaction techniques lend themselves well to *specific* situations. One of the principal objectives of this paper is to enumerate what those *specific* situations are, and how they can be modeled and acted upon.

Examples of contextual information that affects interaction techniques can be found throughout the literature. Perhaps the most studied is the idea of workspace. The workspace or "area of interest" is related to the idea of focus and nimbus (Greenhalgh and Benford 1995) used in the MASSIVE system. Poupyrev et al. (1998) have shown that (direct) virtual hand placement is the best manipulation technique when dealing with objects at close range and that ray casting outperforms Go–Go for selection at long distances. For manipulation at larger distances, Go–Go was shown to be significantly better than ray casting.

Other contextual attributes (or context components) found in the literature include object groupings (Bukowski and Sequin 1995; Stuerzlinger and Smith 2002), frame of reference (Ware and Arsenault 2004), and constraints (Mapes and Moshell 1995). Contextual information that does not fall into these categories may also exist, especially in highly specialized and domain-specific applications (Chen and Bowman 2006). Furthermore, other issues play a role in the selection of interaction techniques—such as hardware availability (Bowman et al. 2007).

Several interaction techniques have been developed that can be described as composite techniques, delivering different types of interaction depending on the current context. Go–Go (Poupyrev et al. 1998) is perhaps the most well known of the approaches. Go–Go delivers direct manipulation when the user is interested in objects within arms' reach and provides an amplification of the hand movement to allow the user to extend their reach as they interact with distant objects. Go–Go *recognizes* the interaction context using a very simple mechanism—the distance between the user's hand and body. As the user reaches further, Go–Go automatically delivers more amplification. PRISM (Frees et al. 2007) takes a similar approach; however, rather than adding amplification, it scales or dampens the user's hand movements to provide enhanced levels of precision when the user is working closely with an object. Just like Go–Go, it also includes an automatic *recognition* mechanism—inferring that the slowing down of the user's hand speed indicates they require more precision (and thus scaling). Go–Go and PRISM are specific examples of a generalized model, which uses context to switch between interaction techniques.

The implementation of interaction techniques (let alone context-driven techniques) can be difficult and is in need of

standardized toolkits similar to those previously developed for 2D windowing systems. This issue has long been recognized, but the goal has not yet been realized. Early 3D interaction toolkits, such as VRID (Tanriverdi and Jacob 2001) and SVIFT (Kessler 1999), have been followed by higher-level and portable implementations such as CHASM (Wingrave and Bowman 2008). A recent trend in 3D interaction toolkit development is the leveraging of task decomposition (Ray and Bowman 2007) (such as those found in (Bowman et al. 1997, 1999; Poupyrev et al. 1997)). Although our toolkit provides a similar approach toward interaction technique development, this paper focuses on the toolkit's support for contextual information as a first-class entity in a user interface toolkit—a feature we do not believe has been adequately addressed yet in the literature.

3 The context-driven interaction model

The goal of this research is to develop a programming and conceptual model supporting context-sensitive user interfaces with the following features:

a) A quantifiable model of the information that makes up the current "interaction context": information that can be used to decide which interaction technique will best suit the user's current needs.
b) A decoupling of the way in which context is recognized (either explicitly through modal commands or implicitly by observing user behavior) from the representation of context and implementation of the interaction techniques themselves.

The product of this research is not only a model; it is also a toolkit and runtime system that serves as a reference implementation. The Context-Driven Interaction (CDI) toolkit is built on top of the Simple Virtual Environment (SVE)[1] library (Kessler et al. 2000) and consists of a set of C++ abstract base classes that developers extend to implement interaction techniques, context types, and context recognition mechanisms (CRM). A large set of concrete implementations of common interaction techniques and CRMs are included. The toolkit can be broken into two conceptual layers, shown in Fig. 1. The first layer is the API, which allows developers to create, register, and activate context recognition mechanisms and context switch callbacks. The second layer is the internal runtime, which manages context components, updates context via registered CRMs, and invokes the context switch callbacks when necessary.

[1] The SVE toolkit is freely available for academic use at http://give.ramapo.edu/lab.html.

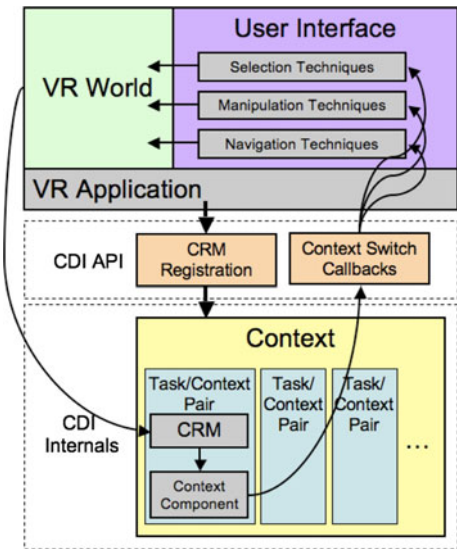

Fig. 1 CDI programming toolkit organization

3.1 Context components

Interaction context represents the current state of the user, including position, interaction history, and intentions or objectives while performing a task. This definition is imprecise by design; context in fact represents *all* knowledge the system has about the user's objectives—the sum of many small bits of information. These "bits" of information can be thought of as answers to questions—questions the system must "ask" in order to provide the most suitable interaction technique for the user's current task. Some information is easily acquired through simple observation of the current state of the virtual world. This includes the orientation and position of the viewpoint, speed of the user's hand movement, etc. Other information, such as the object the user's hand is "touching", can also be easily determined and is unambiguous. On the other hand, higher-level characterizations of context are also (and often more) valuable when deciding upon interaction techniques.

Part of the challenge of developing the CDI model was the identification of the most commonly used high-level characterizations of context—which we call *context components*. To establish a list of context components, we performed a literature review of the most commonly used and successful 3D interaction techniques. For each interaction technique, we asked the question: "Under what circumstance is this technique designed to be more effective than its competitors?" For example, when looking at a viewing technique like orbital viewing (Koller et al. 1996), its clear the technique attempts to add value in situations where the user is focused on a specific area/workspace, while a technique such as landmarks (Pierce and Pausch 2004) is for a broad overview (large workspaces). Looking

Table 1 Inferred context components

Context component	Representation	Description
Level of control	Value: 0–3 Range: Low/normal/high	The amount of precision or speed required. High levels of control indicate precision, low levels indicate preference for speed
Workspace	Volume: 0-INF Position: 3D point Range: small/medium/large, near/medium/far	Size and position of the portion of the world the user is most interested in
Frame of reference	Value: A virtual object Range: World, user, object	Object a particular task is being performed with respect to
Object groupings	Value: 0 or more groupings Range: No groups, groups	Object groups or relationships relevant to interaction task
Constraints	Value: 0 or more constraints Range: No constraints, constraints	Constraints upon the user or object relevant to interaction task

at the literature as a whole, five distinct categories of "context" began to emerge: level of control, workspace, frame of reference, constraints, and object groupings. Table 1 presents these context components and summarizes how our model quantifies them. For each of these components, non-trivial procedures must be developed to recognize the current context "value"; these procedures are referred to as CRMs.

While we believe these five context components cover most of the more common scenarios in a typical application, we do not claim that all contextual information can be distilled into five categories. There is little doubt that unique applications may or may not require specific context components. Our programming toolkit accommodates the creation of new context components by the application developer for use in the system.

3.1.1 Level of control

Level of control refers to the granularity or speed in which we manipulate controlled objects or navigate the world. Consider object translation: often users wish to move objects quickly over large distances. Go–Go (Poupyrev et al. 1996) translation provides this by amplifying the movement of the user's hand and thus the movement of the controlled object. This allows the user to move objects quickly over large distances, but sacrifices accuracy as small changes in the position of the user's hand results in even larger movements in the controlled object. This is an example of a translation technique useful in situations where a user requires a *low* level of control. In contrast, scaled manipulation can be used to enhance accuracy. Scaled translation moves a controlled object more slowly than the hand, dampening the effects of unintended movement. Scaled manipulation is most applicable when a user requires a *high* level of control. In between these extremes lies the normal 1:1 mapping of the physical and virtual hand.

The appropriate interaction technique can only be selected after the current level of control (LOC) is determined. The current LOC cannot be described by a set of discrete values, rather it can be any value lying along a continuum between "Low" and "High." In our model, LOC is defined by a real number between zero and three. A value of zero indicates an extremely low level of control is desired. More specifically, a value close to zero indicates an interaction technique providing rapid, broad reaching manipulation and navigation is most suitable. A value closer to three indicates the user requires precision to accomplish their task. The choice to represent LOC as a value between zero and three is arbitrary; any set of numerical values would have been sufficient.

Being a real number, there are an infinite number of values for LOC; however, there is a finite set of interaction techniques available to the developer. Typically an interaction technique will be suited for a *range* of LOC values, not one value in particular. Our model explicitly supports three ranges: 0–1 is considered "low", 1–2 is "normal", and 2–3 is "high." These threshold values are again arbitrary; they are defined merely as a standard so values of the LOC can be interpreted regardless of the method used calculate it. As an example, PRISM interaction (Frees et al. 2007) uses hand speed to calculate the *level of control*. Slow hand speeds indicate high LOC; however, the definition of "slow" hand speeds can vary (depending on the user or implementation). The context recognition mechanism responsible for observing hand speed converts speed values into the standard LOC notation so the runtime/

developer need not be concerned about what hand speeds are considered "low" in order to decide when to provide scaled manipulation (for high LOC) instead of direct or amplified manipulation (for normal and low LOC).[2] This decoupling allows for alternative methods of determining LOC to be added to the system without modifying the implementation of the interaction techniques. For example, rather than simply observing hand speed, a CRM could be designed to detect physical or cognitive limitations of specific users and adjust LOC as necessary—all without changing the implementation of the scaled manipulation.

3.1.2 Workspace

Knowing which part of the world the user is currently interested in (their workspace) is also an important factor when deciding between interaction techniques. Although the workspace is almost always within view of the user, its exact location and size must be determined, possibly by observing the user's recent actions.[3]

The current workspace (WKSP) is defined as a volume, with a precise center position and dimensions. As with LOC, this definition implies that the current WKSP can be any sized volume—it is not a discrete value. When the current WKSP is large, rapid navigation techniques such as "flying" should be made available. A large or distant WKSP also suggests larger scale selection techniques such as ray casting and broad-reaching manipulation techniques such as Go–Go. In contrast, a small WKSP might indicate that viewpoint control techniques such as orbital viewing (Koller et al. 1996) should be used, as they allow the user to inspect smaller regions of the world rather than gain an overview. The *position* of the workspace provides additional information. For instance, a distant yet small WKSP might indicate scaled manipulation, but it also indicates that arm extension techniques should be used—perhaps a selection/manipulation technique such as HOMER (Bowman and Hodges 1997).

Our model defines three *size ranges* for the current WKSP: small, medium, and large. Like LOC, these ranges are described by using threshold values; however, the mapping is made in a slightly different manner. LOC is described by an abstraction (a real number between 0 and 3), and a context recognition mechanism is free to map any information to the level of control. In contrast, the WKSP is described by a very real value, its physical dimensions. In order to give WKSP recognition mechanisms the flexibility to define what sizes are to be considered "small" or "large", the CRM defines customizable threshold values to indicate small, normal, or large volumes. The distance between the user and the workspace is represented by a similar set of threshold values (near, medium, or far).

3.1.3 Frame of reference

Many interactions are implicitly made with the world or user as the reference point (e.g. navigating to a specific place in the world). Often however, interactions can be made with other objects as the *frame of reference* (FOR). For example, users might want to navigate relative to another object or to move an object such that it is at a specific position relative to another object in the world (Ware and Arsenault 2004).

Unlike LOC and WKSP, the FOR can indeed be defined by a discrete value, namely the object in the world that is currently the frame of reference. This object can be *any* object within the world; however, with respect to selecting the most appropriate interaction technique, the main concern is generally whether the reference object is the world itself, the user, or any other object. This suggests three ranges: world, user, and "other." Unlike the previous context components, the assignment of the range is trivial, as it is a straightforward classification of the object determined to be the frame of reference. As any object can be the current frame of reference, the developer may create arbitrary objects (such as lines or vectors) to serve as a frame of reference or extend the range to include pseudo-objects such as the viewport to aid in supporting interaction techniques utilizing view-fixed semantics as described in (Feiner et al. 1993).

3.1.4 Object grouping

It is common for an object to be part of a larger group or association of objects (Bukowski and Sequin 1995). Depending on the current situation, the user may intend to treat the object as an individual (independent) object or may want to operate on either a subgroup or the entire group of objects.

Object groups can be static, such as the association between a table and chairs. In other situations, object groupings can be dynamic, such as an association created when two objects have been placed in close proximity to each other. Typically the underlying VR system will contain a scene graph, defining parent/child relationships between objects. While a parent child relationship could be expressed through the scene graph, object associations are often dynamic and are semantic in nature. It is also possible that objects can be in more than one group at any given time. These properties make the scene graph a poor method

[2] The true value of the LOC is always available to the developer. The low, normal, and high ranges are defined simply for convenience and as a guideline.

[3] The user could also explicitly indicate their workspace—perhaps by entering coordinates into the application.

of representation. Our model provides the developer an additional construct, *group definition* (GD). The GD contains a *root* object and a set of one or more *child* objects. This is a purely semantic construct; creating and destroying a GD have no effect on the underlying structure of the VR system's scene graph. When any manipulation or selection is performed on the *root* of a *group definition*, the same, or similar, operation can be applied to the *children*. Note—to implement pure sets, where actions on any object within the set are replicated on all other members, one must define several GD relationships.

The *grouping* context type (GRP) is defined as the set of all GDs relevant to the current interaction task—as determined by a grouping context recognition mechanism. The presence of object groups may or may not be relevant for deciding which interaction technique to use, but it is very likely to affect how the interaction technique works. Consider manipulating an object within a group; the decision of which manipulation technique to activate does not normally depend on whether the object is part of a group—the grouping only determines the *number* of objects affected by the manipulation. Our toolkit allows interaction techniques to access group information and behave accordingly.

3.1.5 Constraints

Often either application semantics or the need to reduce complexity requires interaction to be constrained in some way. Constraints can be defined by the application, or they might be created when user behavior indicates they are appropriate. As with GRP, the *constraints* context type (CNSTR) is defined as the set of constraints *relevant* to the current interaction task. A constraints recognition mechanism is responsible for defining which constraints are relevant to the current interaction task. In addition, a constraint recognition mechanism may also create new constraints dynamically based on observation.

The presence of constraints may or may not affect the choice of an interaction technique. For manipulation and navigation tasks, constraints often limit the position of an object or the user. This constraint could be maintained by a specialized interaction technique; however, it is often sufficient to allow the interaction technique to perform the task and apply constraints separately. A constraint that requires the user to stay on the "ground" need not be supported by a special navigation technique; if stylus/cursor directed navigation is active then after the techniques moves the user by some distance the "ground constraint" can be applied independently.

From an implementation standpoint, constraints are supported in a very simple fashion. An abstract Constraint class is provided which defines one particular method—applyConstraints that must be implemented. When a constraint recognition mechanism provides a list of relevant constraint objects, the runtime can automatically invoke each of them by calling applyConstraints. Thus, to implement arbitrary constraints, a developer need only extend the base Constraint class.

3.2 Task/context pairs

The purpose of modeling contextual information is to allow a developer to select the best interaction technique for the user. Thus, the combination of an interaction task (e.g. "translate object") and a context component (e.g. high or low LOC) dictates the selection of an individual interaction technique (e.g. PRISM or Go–Go). This concept is referred to as "task/context pairs."

Combining common interaction tasks and the context components described in Table 1 leads us to a grid mapping out the possible task/context pairs, which is presented in Table 2. Common interaction *tasks* (based on those identified in Bowman et al. 1997, 1999) are listed as rows, and the context components defined above are listed as columns. Each cell in the table represents a *series* of interaction techniques; each specialized at providing an interface for the task and a specific value/range of the context component. For example, the cell in the table corresponding to Selection/Workspace might consist of

Table 2 Common task/context pairs

Each cell represents a series of specialized interaction techniques; each of which are designed to work well for the task under a specific context value/range

	Level of control	Workspace	Frame of reference	Grouping	Constraints
Selection	X	X		X	X
Attach object				X	
Perform translation	X	X	X	X	X
Perform rotation	X	X	X	X	X
Perform scaling	X	X	X	X	X
Object placement				X	X
Navigation	X	X	X		X
Viewpoint control	X	X	X		

"Touch Selection" when working in a small workspace and "Ray Casting" when working in a large workspace. The purpose of this table is to map where context-based decisions are often applicable in 3D interaction. For each marked cell, a UI designer must choose a method of recognizing the context value/range (using a CRM) and the best interaction technique for the current context value.

Of course, not all task/context pairs are as significant as others; while object translation is very dependent on the level of control desired, the user's frame of reference is not particularly relevant to the method in which the user places/releases the object after translation. The most relevant task/context pairs in Table 2 are marked with an X (this does not imply cells without a mark do not have interesting features of their own, however). This table is the product of in-depth studies of current interaction techniques and VR application domains. Section 4 provides several concrete examples where task/context pairs are implemented by specific interaction techniques. A broader discussion, along with a review of the different interaction techniques in the literature that support each cell, can be found in (Frees 2006).

3.3 Implementation of context model

As shown in Fig. 1, the Context object is the central repository for contextual information and logic within the runtime. This object is populated with a set of TaskContext objects—representing the task context pairs outlined in Sect. 3.2. A TaskContext object in turn holds a reference to two types of objects: (1) a CRM to implement the recognition mechanism for the context type, and (2) a ContextComponent, which defines the current context type. The developer indirectly deals with TaskContext objects by registering CRMs and context switch callbacks with the Context object. While the runtime is active, the Context object is updated before each frame drawn by the underlying VR system. The principal job of the update function is to use the registered CRMs (for each Task/Context pair) to determine whether a context switch has occurred and then to notify the application of such a switch via registered callbacks. The details of CRM objects will be discussed in Sect. 3.4. The remainder of this section is dedicated to the implementation of context components and context switch callbacks.

3.3.1 *Implementation of context components*

The base class of all context components is the ContextComponent object, an abstract class defining the interface that all components must implement. The interface class defines several utility methods, but the most important is the isContextSwitch method, called by the Context object to trigger context switches and callback functions.

```
bool virtual isContextSwitch(ContextComponent * oldContext) = 0;
```

This method should return "true" if the given ContextComponent (old context) is *fundamentally* different from itself. A context component is defined as *fundamentally* different from another when its "value" is in a different *range*. The predefined implementations of ContextComponent objects (the toolkit includes implementations of level of control, workspace, frame of reference, object groupings, and constraints) define context switches according to the ranges as described in Table 1. The application developer is free to extend the predefined context component objects to override this method.

3.3.2 *Context switch callbacks*

Context switches are fundamental in the development of context-driven user interfaces—they indicate a new interaction technique is likely necessary. Simply recognizing when context switches occur is not sufficient, the system must have a way to notify the application that this event has occurred. To facilitate this, the Context object allows developers to register callback functions. The developer can register a callback for a specific task context pair, all pairs associated with a specified task, or even all pairs.

```
void AddContextSwitchCallback(int task, int contextType, ContextSwitchCallback callback);

void AddContextSwitchCallback(int task, ContextSwitchCallback callback);

void AddContextSwitchCallback(ContextSwitchCallback callback);
```

For each of these methods, the developer specifies the task and/or context type that the callback will be registered to (each type of task and context component are enumerated by integer constants within the toolkit). The last parameter in each of these methods is of type ContextSwitchCallback, which is a typedef specifying the format of the function.

```
typedef void (*ContextSwitchCallback) (Context &context, ContextSwitch &contextSwitch);
```

The first parameter is simply a reference to the context object provided for convenience. The second parameter is of type ContextSwitch. As the name implies, this object describes the nature of the context switch that has occurred.

```
class ContextSwitch {
  public:
        int task;  // task id

        int contextType;  // context type id

        ContextComponent *oldContext; /* pointer the old ContextComponent associated with this
        task/context pair*/

        ContextComponent *newContext; /* pointer to new ContextComponent associated with this
        task/context pair*/
};
```

3.4 Context recognition

Using a combination of specialized and effective interaction techniques has been recognized as an important principle in the development of immersive user interfaces (Bowman et al. 2001). Providing many different interaction techniques for each task is not a panacea; however, the user/system must have some method of choosing the interaction technique depending on the current context—a CRM is needed.

The most common way of communicating a discrete decision in an immersive system is through an explicit modal command—such as a menu, button click, or more sophisticated interfaces as found in Bowman et al. (2001), Grosjean and Coquillart (2001), and Wesche (2003). When the number of interaction techniques available to the user is low, this form of *explicit* context recognition may be adequate—the user can explicitly indicate the context (and thus select the interaction technique they desire). The drawback of an *explicit* command is that system control widgets take up valuable screen space and often require the use of a keyboard, stylus button, or more specialized input device (scarce resources in an immersive environment).

Fortunately, context can often be recognized without an explicit modal command. Perhaps the most well-known example is Go–Go manipulation, which combines direct and amplified translation by selecting the active interaction technique based on the distance the arm has been extended. This can be thought of as a form of simple context recognition, the user signals to the system the need for low level of control by reaching out their arm.

3.4.1 Supporting the development of implicit CRMs

Implicit CRMs vary significantly with respect to their complexity, and developing quality techniques can be challenging. Worse yet, poorly developed CRMs can profoundly confuse the user and acting on faulty contextual information to select interaction techniques leads to an extremely frustrating experience. Ultimately, implicit context recognition requires extensive user studies in order to determine what contextual information can be gleaned from user behavior and how this information can be acted upon.[4]

With these challenges in mind, we set out to develop a programming model that promotes code reuse, modularity, and good software design principles while developing implicit CRMs. Our solution focuses on the decoupling of implementation of implicit CRMs, interaction techniques, and the mechanisms providing automatic interaction technique activation (based on context). The following text

[4] We do not claim that our model replaces the thorough usability analysis involved in CRM development—it only aids in their implementation.

explains, by demonstration, why this decoupling is advantageous and how it has been achieved.

Let us first consider Go–Go's arm extension CRM from an implementation perspective. To determine the current distance between the user's hand and body, a trivial calculation is performed based on tracking data. This data is unambiguous and can be mapped directly to a level of control (LOC) value. This value can then be used to control a scaling (or amplification) coefficient to provide 1:1 or 1:N mappings between physical and virtual hand movements. These mappings are so trivial that one may argue that a formal separation of the context recognition mechanism (mapping arm extension to LOC) and interaction technique (LOC to scaling coefficient) is in fact unnecessary; a monolithic approach that directly maps arm distance to the scaling coefficient is adequate.

Now let us consider a more complex scenario, one that focuses on using context to choose between two view control techniques: egocentric view control (analogous to the real world) and object-centered orbital viewing (Koller et al. 1996). An implicit CRM for deciding between egocentric and orbital viewing might require the system to determine if a user is focused on a particular object or volume in the world (*frame of reference*). This could be achieved by observing the user's interaction history and developing heuristics to identify their current frame of reference. The implementation of such a CRM would be complex and no doubt requires significant development time and testing. A monolithic interaction technique incorporating the frame of reference heuristics and both types of view control techniques could be developed nonetheless.

The software engineering problem with the monolithic approach occurs when the developer later wishes to create another user interface that also considers frame of reference, but perhaps for some other interaction task (e.g. object manipulation) and with other interaction techniques. In this situation, a developer has two options, (1) reimplement the heuristics for implicitly determining frame of reference or (2) devise a way to share the information already produced by the existing code.

This shared contextual information is precisely what our model provides. The model isolates the implementation of the CRM from the implementation of the interaction techniques by representing contextual information in a standard notation. CRMs communicate the current context through this standard (the Context object), and any interaction techniques can then make use of the information. Different CRMs can be swapped within an application, while the interaction techniques can remain the same—and vice versa.

This decoupling aids in the implementation of interaction techniques as well, as they can be implemented with the assumption that a specific context holds. Instead of embedding conditional code within interaction techniques to support changes to the context, the developer can focus on creating *individual and specialized* interaction techniques. The fact that context is represented in a standard form allows a generic runtime to be built that can simply activate the appropriate specialized interaction technique whenever a change in context occurs.

3.4.2 CRM implementation

CRMs are implemented through an abstract CRM class. CRM objects are registered to a specific task/context pair through the Context object. Once registered, the Context object uses the CRM to determine the current ContextComponent corresponding to the task/context pair. Below is the public interface for the CRM class:

```
class CRM {
  public:
   virtual ContextComponent * GetCurrentContext(Context & context) = 0;
   virtual void Update(Context &context) = 0;
   virtual void Reset() = 0;
};
```

The most important method in a CRM is the GetCurrentContext method. The runtime calls this method before each frame to obtain a current ContextComponent for the task/context pair. A CRM can produce any type of ContextComponent (typically Level of Control, Grouping, Frame of Reference, Workspace, or Constraints). The method in which the CRM determines the context is independent from the runtime (GetCurrentContext is the runtime's only way of obtaining contextual information).

The runtime does help a CRM decide the current context by ensuring that its Update method is called before each frame is drawn (as long as the CRM has been registered for a task/context pair). This method allows the CRM to monitor all activity in the world on a regular basis. Special caching facilities are also implemented in the base CRM implementation to make the sharing of a single CRM instance between multiple task/context pairs more efficient. All CRM objects must also implement the Reset method,

which clears (historical) data being collected to determine context. This method is not called by the runtime, rather by the user/application.

A developer creates new context recognition mechanisms by extending the CRM class. When creating a CRM to handle one of the predefined context components, a developer rarely inherits CRM directly. Instead, the developer would extend LevelOfControlCRM, WorkspaceCRM, FrameOfReferenceCRM, GroupingCRM, or ConstraintsCRM, respectively. These objects contain functionality more specific to the types of context components they produce, especially concerning the caching of context. Much more detail concerning the use of these objects is found in Frees (2006).

3.4.3 Example CRM object

As an example, this section describes the implementation of a CRM designed to implicitly determine the LOC based on hand speed. This CRM is used to implement PRISM. HandSpeedCRM extends the LevelOfControlCRM. Below is the (simplified) class definition:

```
class HandSpeedCRM : public LevelOfControlCRM {
    public:
        HandSpeedCRM();
        ContextComponent * virtual GetCurrentContext(Context & context);
        void virtual Update(Context &context);
        void virtual Reset();
        void SetMinThreshold(float min);
        void SetMaxThreshold(float max);
        void SetPrecisionCuttoff(float cutoff);
    private:
        double currentLOC;
};
```

HandSpeedCRM's Update method is called by the runtime before each frame is drawn. This method records the position of an object (the hand) and calculates its speed. The speed is mapped into a value between 0 and 3 (level of control) and stored it in a private member called currentLOC. This translation is done through linear interpolation between the threshold values set by the SetMinThreshold, SetMaxTreshold, and SetPrecisionCuttoff.

After calling Update, the Context object will call GetCurrentContext, to determine if a context switch has occurred. This method simply wraps the currentLOC in a LevelOfControl context component object, which can then be used by the Context object to determine if a switch has indeed occurred. Note, while the Context object always calls the CRM's Update and GetCurrentContext in sequence, they are implemented separately so external application code can obtain the current context from the CRM at any time with less overhead.

The toolkit provides other CRM objects that use hand position or speed to determine context, including CRMs to implement Go–Go's hand distance methods and another to calculate level of control based on the hand's rotational speed. In addition, CRM objects may use a more explicit method of determining context such as a menu system or widget (see examples in Sect. 4).

3.5 Modeling interaction tasks and techniques

The preceding sections describe how our model achieves the two research goals outlined in the beginning of this section: (a) a quantifiable model of interaction context (Context and ContextComponent objects) and (b) a decoupling of context recognition from interaction technique (and application) development—achieved through CRM objects and context switch callbacks. A secondary goal of the reference implementation was a model and API for developing interaction techniques. Although the details of the API are beyond the scope of this paper, we elaborate some of the design features here in order to make the examples in Sect. 4 more clear.

All interaction techniques implemented with the toolkit are managed by the runtime itself. Each interaction technique object is composed of a number of other object/components; decomposed according to the task decomposition outlined in Table 2. These components contain logic (and any necessary virtual objects/visualizations) that implements specific techniques for accomplishing an interaction task. These include methods in which a user indicates an action (such as "select", or "stop manipulating"), the method in which a user indicates an object to select, the way in which a user manipulates an object, and the method used to indicate direction of navigation, etc. Interaction technique objects serve as containers for these components and for managing communication between them (ex. passing a selected object to a manipulation technique). Each of these components can be swapped in and out at runtime by calling the appropriate methods on the interaction technique object. It is by switching these

components a developer can respond to context changes. Further detail of our interaction technique API can be found in Frees (2006), and http://give.ramapo.edu/lab.html.

It is important to note that while CRMs and the firing of context switch events are completely managed by the runtime, what occurs in response to a context switch is entirely up to the application developer. Typically a context switch indicates that the interaction technique should be changed to something more applicable. This change could be automatic, especially in circumstances where the interaction techniques blend together well. For example, direct manipulation can transition naturally into scaled interaction without creating an abrupt change in the interface. In other circumstances the two interaction techniques may be quite different—and automatically transitioning between them could lead to a jarring experience for the user. In this situation a context switch may simply be used to recommend a different technique to the user (with explicit confirmation). Often an application developer may choose the set of interaction techniques to be provided based on how well they can be blended together and how seamlessly the user can be switched between them; however, our framework does not require this.

4 Example implementations

The decoupling of context, CRM, and interaction technique allows developers to reuse and combine components to create new user interfaces techniques quickly. Below we present an outline of how the model supports a PRISM hybrid manipulation technique using hand speed (level of control) or workspace to determine the appropriate translation technique. The toolkit's reliance on well-defined abstract classes allows significant code reuse between the two interfaces, with the only difference being the implementation of the CRM and registering to different task/context pairs. Later in the section we also show how the toolkit is reconfigured to support other interaction techniques from the literature. While by no means a formal proof of the toolkit, each of the examples that follow have been successfully implemented and have served as an initial evaluation of the effectiveness and flexibility of the model.

Figure 2 illustrates a default implementation of PRISM translation. An object translation technique consists of three distinct parts—a way in which the user translates the object, a way in which the user indicates that the translation is complete (e.g. a button press, voice command, etc.), and finally a way in which the object is placed (perhaps gravity is applied, etc.). We focus here on the way in which an object is translated, and how level of control dictates the proper technique. In PRISM, scaled translation (implemented by the ScaledTranslator object) is active when the level of

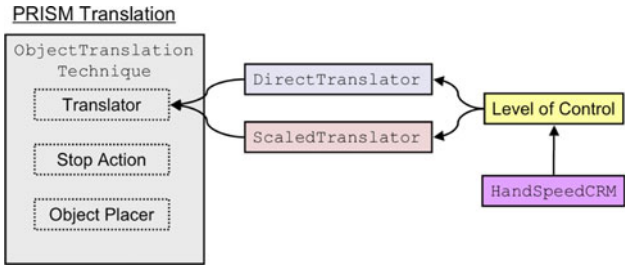

Fig. 2 PRISM translation implementation

control is *high* and direct translation (implemented by the DirectTranslator object) is active when the LOC value is normal or low. Using context switch callbacks, PRISM can easily be implemented by switching between the component translator objects based on LOC. The current LOC value is determined by the same HandSpeedCRM object (registered to the translation/level of control task/context pair) described in Sect. 3.4.3.

Figure 3 illustrates a significantly different user interface, but one that reuses most of the same programming code. In this example, PRISM translation is being implemented using all of the same translation code (DirectTranslator and ScaledTranslator), however the techniques are being activated based on the current *workspace*—not level of control. In addition, a new CRM is introduced (ExplicitWorkspaceCRM), which calculates the current workspace using a simple widget (shown in Fig. 4). In this case, the DirectTranslator could be activated when the workspace is large and the ScaledTranslator would be activated when the workspace is small. More detailed user analysis may also reveal a more implicit method of recognizing the current workspace size, at which time the new CRM could be substituted into the interface.

The example above demonstrates how different CRMs can be plugged into the runtime to fundamentally change the interface without changing the code/implementation of the manipulation technique itself. Similarly, one can reuse the same CRM across different tasks. For example, a HandRotationSpeedCRM has been developed that maps the rotational speed of a user's hand to the level of control context component (slow hand speed maps to high level of control). This determination can be used across interaction

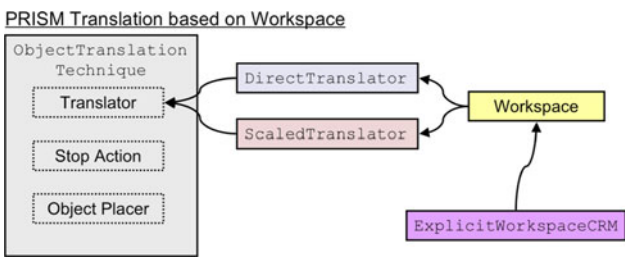

Fig. 3 PRISM translation based on workspace

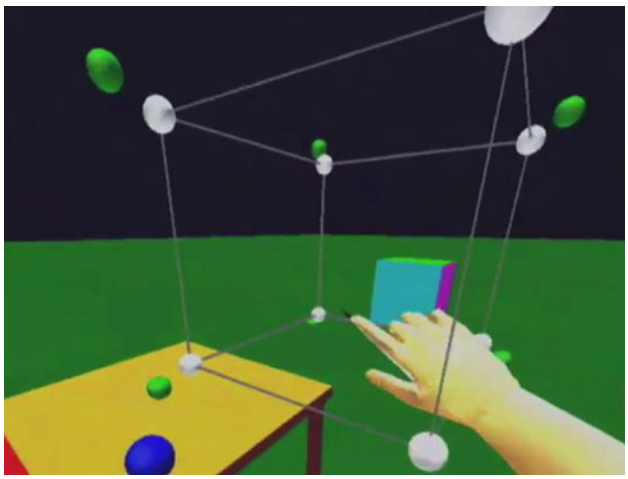

Fig. 4 Explicit workspace CRM visualization—the user explicitly resizes and moves the outlined volume to indicate their current workspace

tasks to choose between object rotation techniques or selection techniques. The effectiveness of this CRM was demonstrated with PRISM scaled rotation and ray casting (Frees et al. 2007). As shown in Figs. 5 and 6, the implementation of these techniques differ only in the interaction task and techniques activated—the CRM is completely reused.

The interface in Fig. 5 implements object rotation in much the same way as the PRISM translation example in Fig. 2. Using the same HandRotationSpeedCRM, one can also build PRISM ray casting as illustrated in Fig. 6. This user interface activates normal ray casting for normal and low LOC values and scaled ray casting for high LOC values (i.e. when a high degree of control/accuracy is

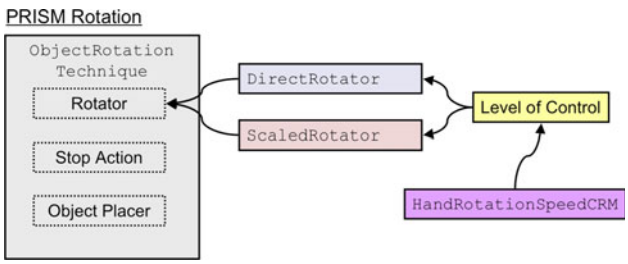

Fig. 5 PRISM rotation implementation

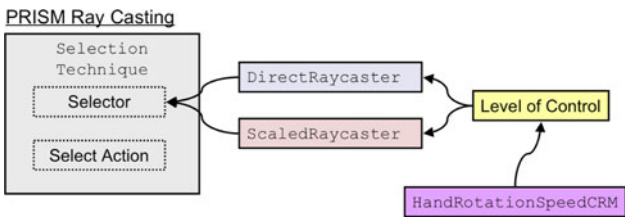

Fig. 6 PRISM ray-casting implementation

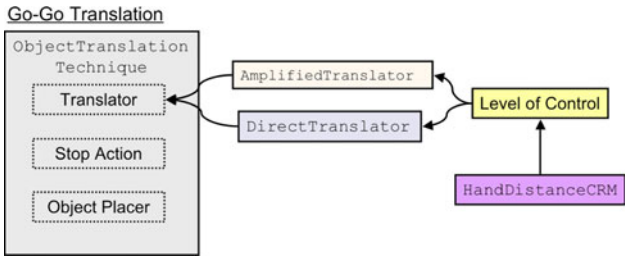

Fig. 7 Go–go interaction implementation

needed). Each of these techniques are again implemented in their own objects (DirectRayCaster and ScaledRayCaster) and extend the base Selector class, which dictates basic functionality required in order to allow the runtime system (and SelectionTechnique object) to work with them.

Other established user interface techniques are also supported by our model and programming toolkit. Go–Go object manipulation can be implemented in a similar fashion as the PRISM manipulation technique shown in Fig. 2; however, in this implementation a hand position CRM could be used to drive the level of control, and an amplified translator could be used for low levels of control (Fig. 7).

Moving out of the realm of object selection/manipulation, the framework can also be used to work with viewpoint control. One implementation might offer the user the ability to switch between normal viewing and orbital viewing when the workspace becomes focused and small (as orbital viewing may be more helpful when concentrating on a small area). In this situation, one could reuse the explicit workspace constraint from Fig. 3 but now use the workspace value to switch between viewpoint control techniques (Fig. 8).[5]

Of course, more than one context component can work to drive the user interface. Our final example combines Go–Go manipulation with an alignment constraint that is automatically created as the object being manipulated comes to rest near another object. This type of user interface resembles the object associations described in Bukowski and Sequin (1995), where a virtual object such as a picture frame may be constrained to snap against a virtual wall depending on its proximity. In this implementation, an implicit constraint recognition mechanism is driving the creation of an alignment constraint and the object placer is using this constraint to move the object to its final resting location (Fig. 9).

This example is of particular note, as the addition of such a constraint is being applied at the time the object is released, meaning the actual selection of the interaction technique is

[5] Switching viewpoint control techniques certainly has the possibility of disorienting the user, and thus this is a prime example of a situation where explicit confirmation might be used so the user is not "surprised" by the change.

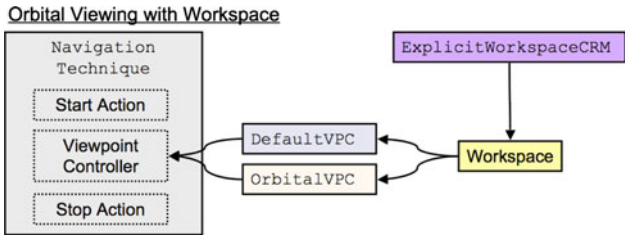

Fig. 8 Egocentric and orbital viewing based on workspace

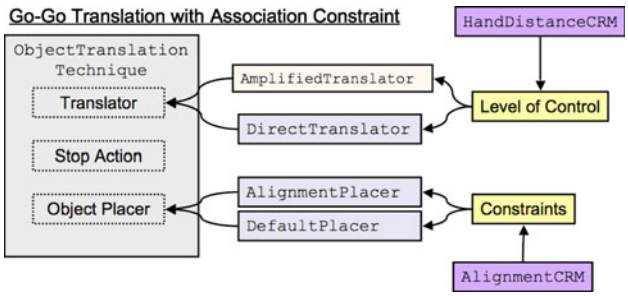

Fig. 9 Go–Go with association constraints

unaffected and somewhat irrelevant. Alternatively, if the translator object implementing the object translation technique were to also take into account current constraints, the user could be provided with instant feedback by constraining the object while it was being manipulated.

5 Conclusion

This research consists of both abstract and concrete contributions. We first outline the important context types involved in interaction in immersive virtual environments and describe the importance of maintaining a separation between the implementation of the context recognition mechanism and interaction technique. We have shown how several successful interaction techniques fit within this model (Go–Go, PRISM). We believe this model also suggests a general strategy toward the development of context-sensitive user interfaces.

1. Select relevant Task/Context pairs for the particular application: *Each application requires users to perform some set of interaction tasks. In addition, most applications incorporate a number of interaction contexts. Identifying these pairs is the first step towards developing a context-sensitive user interface*.
2. Select a Context Recognition Mechanism (CRM) to calculate each Task/Context pair: *The selected CRM may be implicit (context is deduced from observed behavior) or explicit (such as a menu system). Extensive domain knowledge cannot be substituted—for each task/context pair, the development and selection of appropriate CRMs requires significant thought, development time, and evaluation*.
3. Select interaction techniques to support each range associated with each Task/Context pair: *The selection of which technique to use for each context should be informed by the literature and usability studies.*

To aid in the development of context-sensitive user interfaces, we have developed a programming toolkit that follows the context model. We have presented the most important aspects of the toolkit, specifically its treatment of context and the method in which developers can reuse and combine interaction techniques and context recognition techniques to implement user interfaces. We have used the toolkit to implement user interfaces dealing with each of the context components identified in this paper, and the toolkit as proven to be flexible and fairly easy to work with—as demonstrated by the fact that undergraduate students have been able to use the toolkit to implement the examples provided in the previous section of this paper. We have also found that using the toolkit works well for rapid prototype development. This is due to the decoupling of interaction techniques and CRMs, which allows the developer to create interaction techniques with a simple menu-based CRM and then once the interaction techniques have been perfected, turn their attention to creating the implicit or more sophisticated CRM.

6 Future work

To a large extent, the goal of this research was the facilitation of future work. Table 2 identifies task/context pairs that should be considered when designing a user interface and serves as a roadmap for the future development of interaction techniques; each task/context pair represents a relatively independent path of research. The future development of effective and modular context recognition mechanisms also provides a type of feedback loop. As more reusable CRMs and interaction techniques are developed, these components can be combined in different ways without requiring extensive development work. It is hoped that this multiplicative effect can eventually make developing context-sensitive user interface less of a programming task and more a careful decision process focusing on assigning the best CRMs and interaction techniques for the task/context pairs relevant to the target application.

Part of our future efforts will also be to enhance and extend our programming toolkit. The toolkit itself is merely a reference implementation and as more user interfaces are implemented with it there will undoubtedly be a need for more functionality to be added. It is also currently closely integrated with the SVE virtual reality

library; although SVE is freely available and works with a large variety of VR platforms, we are interested in creating an abstract layer between our toolkit and the underlying VR library (scene graph, tracking, etc.) so our implementation can be useful to a wider audience.

References

Albinsson PA, Zhai S (2003) High precision touch screen interaction. In: Proceedings of SIGCHI conference on human factors in computing systems, pp 105–112

Beir EA (1990) Snap-dragging in three dimensions. In: Proceedings of the ACM symposium on interactive 3D graphics, 24(2): 193–204

Bowman DA, Hodges LF (1997) An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In: Proceedings of the ACM symposium on interactive 3D graphics, pp 35–38

Bowman DA, Koller D, Hodges LF (1997) Travel in immersive virtual environments: an evaluation of viewpoint motion control techniques. In: Proceedings of the virtual reality annual international symposium, pp 45–52

Bowman DA, Johnson D, Hodges LF (1999) Testbed evaluation of V.E. interaction techniques. In: Proceedings of the ACM symposium on virtual reality software and technology, pp 26–33

Bowman D, Wingrave C, Campbell J, Ly V (2001) Using pinch gloves for both natural and abstract interaction techniques in virtual environments. Proc HCI Int:629–633

Bowman D, Kruijff E, LaViola J, Poupyrev I (2001) An introduction to 3D user interface design. Presence Teleoper Virtual Environ 10(1):96–108

Bowman D, Badillo B, Manek D (2007) Evaluating the need for display-specific and device-specific 3D interaction techniques. In: Proceedings of virtual reality international conference (in Lecture notes in computer science, vol 4563, pp 195–204)

Bukowski RW, Sequin CH (1995) Object associations: a simple and practical approach to virtual 3D manipulation. In: Proceedings of the ACM symposium on interactive 3D graphics, pp 131–138

CDI Toolkit Web Site: http://give.ramapo.edu/lab.html

Chen J, Bowman D (2006) Evaluation of the effectiveness of cloning techniques for architectural virtual environment. In: Proceedings of IEEE virtual reality. Alexandria, VA, pp 103–110

Feiner S, Macintyre B, Haupt M, Solomon E (1993) Windows on the world: 2D windows for 3D augmented reality. In: Proceedings of the ACM symposium on user interface software and technology, pp 145–155

Forsberg A, Herndon K, Zelesnik R (1996) Aperture based selection for immersive virtual environments. In: Proceedings of the ACM symposium on user interface software and technology, pp 95–96

Frees S (2006) Intent driven interaction in immersive virtual environments (Doctoral Dissertation). Lehigh University, Department of Computer Science, 266 p. Available from Dissertations and Theses database. (UMI/AAT No. 3215837)

Frees S, Kessler GD, Kay E (2007) PRISM interaction for enhancing control in immersive virtual environments. ACM Trans Comput Hum Interact 14(1)

Greenhalgh C, Benford S (1995) MASSIVE: a collaborative virtual environment for teleconferencing. ACM Trans Comput Hum Interact 2(3):239–261

Grosjean J, Coquillart S (2001) Command & control cube: a shortcut paradigm for virtual environments. In: Immersive projection technology and virtual environments 2001 proceedings, pp 1–12

Kessler GD (1999) A framework for interactors in immersive virtual environments. Proc IEEE Virtual Real, pp 190–197

Kessler GD, Bowman DA, Hodges LF (2000) The simple virtual environment library, and extensible framework for building VE applications. Presence Teleoper Virtual Environ 9(2):187–208

Koller D, Mine M, Hudson S (1996) Head-tracked orbital viewing: an interaction technique for immersive virtual environments. In: Proceedings of the ACM symposium on user interface software and technology, pp 81–82

Mapes DP, Moshell JM (1995) A two-handed interface for object manipulation in virtual environments. Presence: Teleoper Virtual Environ 4(4):403–416

Nardi BA (ed) (1996) Context and consciousness—activity theory and human computer interaction. MIT Press, Cambridge

Pierce J, Pausch R (2004) Navigation with place representations and visible landmarks. In: Proc IEEE Virtual Real, pp 173–180

Pierce J, Forsberg A, Conway M, Hong S, Zeleznik R, Mine M (1997) Image plane interaction techniques in 3D immersive environments. In: Proceedings of the ACM symposium on interactive 3D graphics, pp 39–40

Poupyrev I, Billinghurst M, Weghorst S, Ichikawa T (1996) The go-go interaction technique: non-linear mapping for direct manipulation in VR. In: Proceedings of the ACM symposium on user interface software and technology, pp 79–80

Poupyrev I, Weghorst S, Billinghurst M, Ichikawa T (1997) A framework and testbed for studying manipulation techniques for immersive VR. In: Proceedings of the ACM symposium on virtual reality software and technology, pp 21–28

Poupyrev I, Weghorst S, Billinghurst M, Ichikawa T (1998) Egocentric object manipulation in virtual environments: empirical evaluation of interaction techniques. Comput Graph Forum 17(3):41–52

Ray A, Bowman D (2007) Towards a system for reusable 3D interaction techniques. In: Proceedings of the ACM symposium on virtual reality software and technology, pp 187–190

Ruddle RA, Jones DM (2001) Movement in cluttered virtual environments. Presence: Teleoper Virtual Environ 10:511–524

Stoakley R, Conway M, Paush R (1995) Virtual reality on a WIM: interactive worlds in miniature. In: Proceedings of the ACM conference on human factors in computing systems, pp 265–272

Stuerzlinger W, Smith G (2002) Efficient manipulation of object groups in virtual environments. Proc IEEE Virtual Real, pp 251–258

Tan DS, Robertson GG, Czerwinski M (2001) Exploring 3D navigation: combining speed-coupled flying with orbiting. In: Proceedings of the ACM conference on human factors in computing systems, pp 418–424

Tanriverdi V, Jacob RJ (2001) VRID: a design model and methodology for developing virtual reality interfaces. Proceedings of the ACM Symposium on Virtual Reality Software and Technology, pp 175–182

Usoh M, Arthur K, Whitton MC, Bastos R, Steed A, Slater M (1999) Walking > walking-in-place > flying, in Virtual Environments. In: Proceedings of the 26th annual conference on computer graphics and interactive techniques, pp 259–264

Ware C, Arsenault R (2004) Frames of reference in virtual object rotation. In: Proceedings of the 1st symposium on applied perception in graphics and visualization. Los Angeles, California, 7–8 Aug 2004

Wesche G (2003) The toolfinger: supporting complex direct manipulation in virtual environments. ACM international conference proceedings series—proceedings of the workshop on virtual environments, pp 39–45

Wingrave C, Bowman D (2008) Tiered developer-centric representations for 3D interfaces: concept-oriented design in chasm. IEEE Virtual Real