An interaction framework for VR and AR applications

M. Scarpa R.G. Belleman

Section Computational Science
University of Amsterdam
{mscarpa|robbel}@science.uva.nl

Keywords: Virtual Reality, Augmented Reality, UIMS, human computer interaction

Abstract

The implementation of interaction techniques in Virtual Reality (VR) and Augmented Reality (AR) applications still poses a considerable challenge for developers. There are few tools, conventions and standards to rely on. This paper presents the design of a framework aimed at the separation of interaction-specific and domain-specific functionality of VR/AR applications. The goal is to alleviate the development of interaction techniques by providing generic, re-usable components.

1 Introduction

Interaction paradigms for VR and AR have not undergone a comparable process of standardization that resulted in today's common WIMP paradigm and the standard user interface (UI) elements for the desktop [1]. One explanation for this is the highly heterogeneous nature of VR and AR setups.

As a consequence, there is a high degree of incompatibility between the various setups. VR/AR applications are therefore designed and developed with one particular setup in mind and are rarely ported to another. This is sometimes due to the high costs of adjusting the code to the differences between setups. Additionally, different environments present sometimes radical differences in the interaction possibilities offered to the end user, which introduces incompatibilities in the operation of the system next to incompatibilities in the application code.

This paper proposes a framework that aims at separating the domain-specific functionality of a VR/AR application from the interaction-specific functionality. It introduces an additional layer that handles all input data from the user. The obtained data is processed in a bipartite data-flow network made of components that together implement the interaction.

2 Related Work

The separation of interaction-specific and domain-specific functionality has been advocated in the work related to User Interface Management Systems (UIMS) [2]. A UIMS mediates the interaction between the user and an application by satisfying requests from both user and application. This is achieved through a three-tiered model: the lowest level isolates physical devices, which are then managed in the middle layer as logical devices. On the top level, an interaction management system contains descriptions of the dialogue sequences that describe the interaction between user and application [2].

Research on UIMS has been extensive. Among the more closely related work is that of Jacob et al. [3], in which state diagrams and Petri-nets are used to model interaction in VR environments. In [4] the development and adaptation of reality-based interfaces based on the token and constraint paradigm [5] is discussed. The author advocates dynamically changing execution environments and concurrent interaction across various devices.

Another direction is explored in [6], in which users of ubiquitous AR systems are allowed to adapt the system to their preferences using a concept called "interaction management". Similarly, Jacob addresses the runtime adjustment of the interface between UI and application to improve accessibility [7].

In the Alpha UIMS the separation between interaction-specific and domain-specific functionality is supported but not enforced by a component-based design, in which different "agents" communicate with each other, implementing various parts of the application [8]. These agents are organized through a "dialogue", specified using a separate language called "slang".

Many models have been developed for the design, prototyping and evaluation of user interfaces, both for common desktop environments, as well as more complex settings. In [9], the authors present an approach

wherein concrete interaction objects are abstracted into counterparts that capture only generic features. This approach allows no run-time adaptation and is aimed at the evaluation of virtual user interfaces.

Navarre et al. developed a formal description for interaction in virtual environments that addresses the variety of available input devices [10].

2.1 Scope of this work

This paper addresses several issues where the related work falls short, as described next.

Meyers et al. identify two reasons for the lack of success of UIMS [1]. First of all, the often strict separation between interaction-specific and domain-specific functionality has often been perceived as a hindrance rather than an aid to the design and implementation tasks. Secondly, UIMS were originally aimed at resolving the issues of user interface design for the emerging desktop systems, but through the standardization of the user interface elements in that environment, the need for UIMS disappeared. It is unlikely that a similar standardization of user interface elements will occur in VR/AR, which warrants the renewed investigation into UIMS for VR/AR applications.

Many interaction techniques exist in VR/AR today, both for generic as well as for very specific tasks [11]. Unfortunately, very little approaches exist that abstract them into generic, re-usable components. Most of the techniques available today are implemented as part of more complex applications that are tied to particular hardware configurations. The abstraction of interaction techniques into generic, re-usable components not only allows the creation of a whole library of interaction techniques [12, 13], but also a way to easily replace one technique with another. This makes it possible to easily replace techniques unsuitable for a given setup, as well as to compare and evaluate the use of different techniques in a particular scenario. Additionally, it allows the users to specify preferences about which techniques they want to use.

Additionally, specific interaction techniques can be tightly related to a particular domain, rendering it much more difficult – and maybe even counter-intuitive – to generalize them. Such interaction techniques might take harm from a separation from the domain-specific functionality instead of profiting from it. The application developer should be able to decide which functionality is domain-specific and which is not, effectively allowing for the framework to adjust to the developer instead of forcing the reverse.

Interaction in VR and AR applications comprises both discrete and continuous, sequential and concurrent events [3]. Most approaches found in literature cope with this difference by using two separate models, such as Petri-nets for the continuous aspects and state diagrams for the discrete aspects [3]. The use of two different models introduces additional complexity in the description of interaction, which makes the adoption of such systems difficult.

In [14] the authors point out that there are indirectly related aspects that may be significant for the performance of interaction techniques. Especially the feedback of the system is important for the successful use of interaction techniques. This stands in contrast with the general concept of strictly separating input and output functionality, as advocated by many UIMS implementations.

3 Architecture

The proposed architecture consists of a framework that allows the application developer to "request" interaction capabilities for the end user, e.g. "the user must be able to change the position of this object". When the application is deployed, the framework must accommodate all the requests of the application. To achieve this, a data-flow network is constructed, such that the requested functionality can be obtained. The data-flow network consists of various objects and sub-nets that are taken (possibly automatically) from a predefined library or provided by the application developer – or even the maintainer of the system.

The following sections describe the elements of the proposed framework architecture in more detail.

3.1 Data Representation

All data processed by the components of the data-flow network is represented by subclasses of the abstract class *Value*. This class defines only the most basic requirements for a *Value* object:

name A human-readable string describing this value. The purpose is to add semantic information to the value, such that a user can understand what this value represents. For example, the object representing the vertical position of a mouse pointer would be called "vertical mouse position".

value The actual value. This is undefined in the abstract class.

compatibility operator Verifies if two *Value* objects are semantically compatible.

The compatibility of two *Value* objects is defined as the expressive equivalence of the two beyond simple type-compatibility. For instance, a float value with a range between 0.0 and 1.0 is not compatible with a value with a range between -1.0 and 1.0, since it cannot express values between -1.0 and 0.0. This same value is though compatible with a value of range [0.0,

0.5]. Furthermore, two lists of *Value* objects are defined as being compatible if every element with index i of the first list is compatible with the corresponding element at index i in the second list.

Concrete subclasses of the *Value* class are expected to extend the abstract base class adding any attributes necessary to properly characterize the value represented. This characterization is important to assert the compatibility of the values generated by one component and required by another component. The attributes of a value are stored together with the actual data in order to allow access to these attributes anytime during the processing of the data.

3.2 Data-flow Network

The whole process of interaction is modeled using a data-flow network that represents the flow of data from (logical) input devices to either interaction technique components or call-back functions of the application. This approach was chosen, because this concept is widely used in many tool kits and is therefore familiar to many developers. The data-flow is organized in a directed, bipartite graph consisting of two classes of objects: *processing objects* and *connection objects*.

Processing objects represent any kind of operation that uses data as input and/or output. Every logical input device or interaction component is represented as a special processing object in the data-flow graph. This class of objects is characterized by the following features:

identifier The identifier describes the functionality of the object. For components implementing interaction techniques, this identifier specifies what interaction is implemented. This is used when the requirements of an application are matched with the elements of the data-flow network. For input devices, this identifier specifies that the object is an input device.

input signature The input signature specifies a list of inputs required by this object. Every item in this list is semantically seen as one single input, which can either be a *Value* object or a list of *Value* objects. The order of the elements in the input signature describes the order in which the data has to be provided to the execute method (described shortly).

output signature The output signature specifies what outputs the technique provides. It is represented in the same fashion as the input signature.

execute method This method causes the execution of the functionality of this object. It takes as argument a list as defined by the input signature and computes a list consisting of the values as specified by the output signature. These values can then be accessed using the output retrieval method.

output retrieval method This method returns either a list of values matching the output signature specification or *NIL*, if no output is available. This distinction is crucial, because a connection object will only transfer data from its sources to its destinations if all the sources are able to provide data. For convenience reasons, a second output retrieval method could be added, that only returns specified outputs instead of returning all of them.

Connection objects manage the flow of data between processing objects. A connection object reads data from one or more processing objects and forwards this data to the execution method of one or more processing objects. The following attributes characterize a connection object:

sources A list of processing objects from which the connection object reads data using their output retrieval method. The processing objects that provide data to a connection object are called the “sources” of the connection object.

destinations A list of processing objects to which the connection object passes the read data calling their execution methods. The processing objects that receive data from a connection object are called the “destinations” of the connection object.

mapping The mapping defines how the outputs of the sources are mapped onto the inputs of the destinations. Every output of all sources is related to zero or more inputs of all destinations. A mapping is not required to be either injective nor surjective: inputs from any source can be relayed to one or more destinations, or none at all. This is useful e.g. for the conversion of discrete to continuous events (see Section 3.6).

Please note that the data passed through by a connection object is in no way modified by the connection object. Data is only processed by a processing object. If the output of a processing object is not compatible with the required input of another processing object, they cannot be connected directly through a connection object. Also, a connection object does not perform any kind of buffering. If such functionality is required, special processing objects that implement it must be used.

3.3 Logical Input Devices

There are various subclasses of processing objects, derived from the abstract class *PObject*. Physical input devices are abstracted into logical devices, which

are represented by subclasses of the *Device* class, an abstract subclass of the *PObject* class. This approach has been chosen to offer an easy means of extending the range of supported devices. To add support for a new device, it is sufficient to create a new subclass of the Device class that implements the necessary driver-calls. By creating implementations that support hardware-abstraction libraries like VRPN [15], a wide range of devices can easily be added to the list of supported hardware.

The abstract *Device* class does not extend the functionality of the abstract *PObject* class. Its existence is purely semantic, to group all logical device implementations in a separate branch of the class hierarchy. The input signature of such objects is usually empty, since instances of this class are supposed to provide data from the actual hardware they represent. The output signature, instead, provides the data. It is important to note that the way a device presents its data is ambiguous. A six degree of freedom tracking device can present its data as a list of six separate values, as only one tuple (consisting of six values), as two tuples (e.g. each containing three value), etc. It is though advisable to adopt a convention for the way devices present their data to the framework. Such a convention can be used to enforce certain aspects onto the networks created, allowing the quantitative distinction between analogous network configurations (see discussion in Section 5).

3.4 Interaction Techniques

Another group of subclasses of the *PObject* class are those implementing basic interaction techniques, like selection and manipulation.

Using a data-flow network to represent the interaction allows to represent complex interaction techniques as sub-nets of more basic techniques. Interaction techniques are often categorized using taxonomies, whereby tasks are divided into multiple subtasks [14, 11]. This structure can be entirely represented in the data-flow network, enhancing the reusability of the sub-components of a complex interaction technique and facilitating the design of new, complex techniques. The composition of complex techniques is discussed in more detail in Section 4.3.

3.5 Callback Functions

To give the application developer free choice about how to implement specific functionality, the presented framework offers the possibility to register callback functions that are called by special processing objects in the data-flow graph. In this case, the application specifies a callback function and an input signature. A special processing object is then instantiated, that takes over the specified input signature and passes the input data it receives to the given callback function.

3.6 Discrete and Continuous, Sequential and Concurrent Events

In [3] the authors highlight that interaction in VR/AR applications often comprises both discrete and continuous events, sequential or concurrent. The proposed model supports all these modalities through the special characteristics of the connection objects. Connection objects always pass their input data to their destination processing objects – if the data is available. This models continuous events, in the case that all the sources of the connection object are continuously providing data. Alternatively, discrete events are achieved, if at least one of the sources only provides data at discrete moments in time.

The modeling of interaction in a data-flow network also supports the description of concurrent events, as there is no limitation to how many separate paths the network can contain. One instance can be made of an arbitrary number of sub-graphs.

3.7 Feedback to the User

In order to ensure usability, interaction techniques must provide sufficient feedback to the user. The effects of user interaction should be reflected in the state of the application with as little delay as possible. To allow this, the presented architecture offers processing objects access to the application's scene graph. Special processing objects that provide lists of scene-graph objects can be made available e.g. by the application. This enables not only the complete implementation of an interaction technique outside the application itself, but it also allows interaction techniques to directly influence objects seen by the user, providing immediate feedback.

3.8 Graph Representation

The composition of interaction techniques and the arrangement of data-flow graphs can be expressed both programmatically using the provided API, but it can also be represented graphically as a directed, bipartite graph. This has the advantage that a developer can define the interaction-specific functionality that is not provided directly by the framework in the same fashion as the rest of the application. Alternatively, end users can adjust the interaction network easily to match their preferences using a visual editor to edit the data-flow graph.

The creation of a suitable data-flow network that satisfies the requirements of a given application can be done once for all users of the system by the system maintainer, but it is possible that there are multiple possible configurations. Therefore, a user might be able to choose between equal alternatives. These choices can consists of pre-defined configurations for the less experienced users, but more experi-

enced users can also edit the data-flow graph directly, using a visual editor.

3.9 Network Execution Model

For the execution of a given data-flow network, a notification mechanism is introduced. A processing object sends a notification to all connection objects reading from it, signaling the availability of data. Every connection object maintains a status array describing the availability of data for each of its sources. Whenever a notification arrives, the connection object marks the corresponding field in the status array as true. Only if all fields are true, it resets the array and passes on the data.

4 Implementing User Interaction

This section discusses the implementation of user interaction techniques using the proposed framework.

4.1 Notation Conventions and Basic Constructs

The easiest way to represent the described data-flow networks is through diagrams. In the following sections, the data-flow network examples discussed will be represented graphically as bipartite, directed graphs. Processing objects will be represented by ovals, while connection objects will be represented as thin rectangles. The text inside an oval describes the type (between pointy brackets) and/or the name of the object (between double quotes). An exception to this rule is made for "conditional filters" (see Section 4.1.1). The text inside ovals depicting these objects represents the condition under which the received data is forwarded. A connection object's mapping is represented graphically through the vertical correspondence of inbound and outbound arrows.

The following sections discuss some basic constructs often used in data-flow networks as presented here.

4.1.1 Conditional Connections

It can be useful to create connections between processing objects such that they are only active under certain conditions. To achieve this, special processing objects called "conditional filters" are used in combination with the particular characteristics of connection objects.

Conditional filters are special processing objects that take a given input and test it on a given condition. Only if the condition is met, the input is passed on as output. Adding such an object to the sources of a given connection object results in stalling the connection as long as the condition of the conditional filter is not met – even if all other inputs are available.

4.1.2 State Transitions

Interaction techniques often distinguish different states. Depending of the current state, the actions of the user have different meanings. In terms of the presented data-flow networks, states are used to activate and deactivate certain connections. To model this, another special processing object type is used: "buffers".

Buffers are special processing objects that remember the last received input and provide it continuously as output. Using a buffer object as the state variable, it is possible to implement state-persistence. Furthermore, connecting a buffer to a conditional filter can be used to render connections active only for certain states.

4.1.3 Event Conversion

Discrete events can easily be converted to continuous events using a buffering objects to remember the last data obtained and output it continuously. In similar fashion, continuous events can be converted into discrete events in two ways. The first option is to use conditional filters to restrict the continuous output. The second option is to use the characteristic of a connection object to only forward data once all its sources have data available. This way, the continuous output is blocked until another, discrete output is available.

4.2 Examples

The following sections present some examples for the implementation of common interaction techniques using the proposed framework.

4.2.1 Pointing Selection with a Wand

A wand consists of a 6DoF tracker with some buttons attached. One of the most popular class of selection techniques used in conjunction with a wand is "pointing selection": a virtual ray extends from the front tip of the wand and objects intersecting with this ray are selected when the user presses a button.

To model this behaviour in a data-flow network, the wand is represented as a logical device from a subclass of the *Device* class. This object must provide the position and orientation of the wand, as well as the state of all the attached buttons.

Another component must implement the selection mechanism itself. This component requires three inputs: a position from where the ray is cast, a direction into which the ray is projected and a list of objects that can be selected. Using this data, it is possible to compute ray-intersection with all selectable objects and thus produce a new list of selected objects.

In this example, the wand is a continuous source of data, which means that it constantly emits the current

position and orientation of the tracker, as well as the state of all the buttons. Yet, the actual "act" of selecting an object is a discrete event that happens when the user presses a certain button. To model this, the continuous status-report of the button are converted into discrete events. This is achieved using a conditional filter. In this case, the criterion is that the status of the button must be "pressed", syntactically defined through a symbolic constant.

Finally, a selection should be persistent, until the user changes the selection (e.g. selecting another object, or "de-selecting" through a selection of "nothing"). A buffering object is used to store the results of the selection. Figure 1 displays the resulting dataflow network.

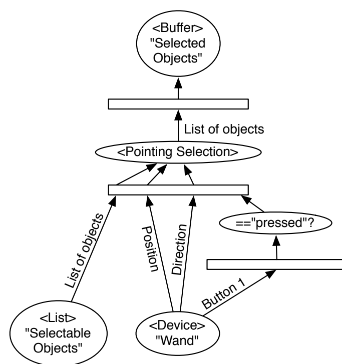

Figure 1: Pointing selection with a wand.

4.2.2 Complex Example: Positioning using a Data-glove

This example discusses a common interaction technique used in combination with a data-glove input device. The user selects a virtual object by "grabbing" it. The selected object will remain selected as long as the gesture is retained. As long as the object is selected, its position relative to the data-glove is preserved: the user moves the object by moving the hand wearing the glove. Please note that for the sake of brevity, rotation is left out in this example, although it could easily be added, as described at the end of this section.

The described interaction can be decomposed into three parts: first, the selection is determined; second, the relative position of the object in respect to the data-glove is calculated upon selection; finally, the object is moved as long as the object is selected.

An object is selected if it is within a given threshold from the glove's position when the user performs the gesture "grabbing". It would be a mistake to simply couple the proximity calculation to this gesture recognition, because the selection should only occur once. As soon as an object is selected, the interaction paradigms change: movement of the glove is reflected in movement of the selected object, while approaching another object must not result in the selection of that object as well.

To achieve this, a state-transition is introduced. When the gesture performed by the user changes from "not grabbing" to "grabbing", the state changes from "nothing selected" to "selection made". This reflects the conflictual nature of continuous and discrete events: a gesture is a continuous characteristic of the data-glove, while the transition from one particular gesture to another is a discrete event.

In addition to the discrete event of selection, there is another discrete event caused by the user "releasing" the selected object. For the sake of argument we shall say that the user signifies the end of the operation with a particular "release" gesture. If and only if the "release" gesture is performed after the "grab" gesture, the state changes again from "selection made" to "nothing selected". In this case, the coupling between the object's and the glove's position must be interrupted.

Since the state is described by the currently performed gesture, it is sufficient to store the gesture in a buffer object to maintain the current state of the interaction.

The selection is computed by a dedicated filtering object that takes as inputs a list of selectable objects, a position and a proximity threshold.

The second step is achieved using another filtering object. This object takes a list of objects and a position in space and returns a list containing the distance vectors of each object to the given position. Note that the implementation of the previously mentioned proximity selection object can make use of another instance of this same class to perform its calculations. The details of this are left out for brevity.

The distance vectors are then added to the glove's position to determine the new position of each object. This is then applied in the final part, using another processing object that takes a list of objects and a list of positions and moves the objects to the given positions.

Note the use of an object representing an empty list in Figure 2. This object is needed to "reset" the buffer containing the list of selected objects when the user "releases" the selection. If this was omitted, the processing object that performs the positioning could receive the previous selection as input parameter while the proximity selection is still busy calculating a newly performed selection.

Support for orienting the selected object can be added by supplying a processing object capable of applying a given orientation to a given object, as well as means to calculate and store the relative orientation of the selected object – in respect to the glove – upon

Figure 2: Object positioning using a data-glove.

selection. All these operations are analogous to those performed for the relative positioning of the object.

4.3 Composite Interaction Techniques

The composition of interaction techniques to achieve more complex functionality can be approached from two opposite directions. The application developer can either create the complex functionality inside the application, requiring the underlying framework to provide the simpler components used. Alternatively, the application can request the complex interaction functionality entirely from the framework. In the latter case, appropriate data-flow networks must be put in place to accommodate the requirements of the application. A new processing object can implement the complex technique by relaying parts (or all) of its inputs to the other objects, thereby encapsulating a sub-graph. This allows to organize complex data-flow graphs into smaller sub-graphs, facilitating their maintenance.

For example, consider the scaling of an object. To keep this example simple, we will only consider uniform scaling. This functionality can be implemented by means of a widget: a simple sphere is drawn at a certain position and distance relative to the center of the object. By manipulating the position of this sphere, the user determines the scaling factor: if the distance between the sphere and the center is increased, the object becomes bigger; if the distance is decreased, the object becomes smaller. Obviously, the movement of the sphere must be constrained to one axis.

The implementation of this functionality can be achieved through the application itself. The application specifies the sphere as an object that can be moved (constrained to a certain axis relative to the object). It further needs to register a callback function to be notified whenever the user moves the sphere. This callback function then queries the new position of the sphere and calculates the distance to the object's center and consequently the scaling factor, finally applying the scaling.

Alternatively, the application can simply require the functionality for an object to be scaled. In this case, the framework mimics the same functionality as described above: a scaling component scales a given object by a given factor. One component calculates the distance between the default position of the sphere and the current one, in order to determine the scaling factor. Another component transforms the position of the sphere from world-coordinates into coordinates relative to the object to be scaled. Finally, a constrained positioning component allows the sphere to be moved along a given axis. The sphere object is part of the functionality and is therefore defined and created in the framework, relieving the application programmer of this task.

5 Conclusions and Future Work

The presented work is aimed at solving the issues described in Section 2.1. The abstraction of interaction techniques into a network of components allows for re-usability of these components to implement new techniques. The separation between interaction-specific and domain-specific functionality is left open, allowing the application developer to choose how to implement which functionality. Continuous and discrete events are described using the same concept of a bipartite data-flow graph. Concurrent event handling is possible through parallel paths in the graph. System feedback is achieved directly through the data-flow graph, allowing for shorter delays and more complete implementation of interaction-specific functionality within the concept of the graph. An additional, graphical language is provided next to an API, allowing for both a programmatic as well as a graphical interface to the description of user interaction.

A first prototype of the presented framework has been implemented. Various hardware setups are in place to test the adaptability of the framework to varying configurations. A number of test cases are being developed to verify the applicability of the presented abstraction to various application scenarios.

Furthermore, the performance of the architecture must comply with the requirements of typical VR/AR applications. Existing interaction techniques will be implemented using these paradigms to both assess the feasibility of the presented approach, as well as to create a library of components to aid the development of new applications and interaction techniques.

The possibility to objectively compare different data-flow networks using defined criteria will be researched. A measure to qualitatively distinguish multiple solutions for the accommodation of the requests of an application with a given set of input devices represents the first step towards the automatic generation of such data-flow networks.

6 Acknowledgments

This work was carried out in the context of the Virtual Laboratory for e-Science (VL-e) project[1]. This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT Innovation program of the Ministry of Economic Affairs (EZ).

References

[1] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

[2] James J. Thomas and Griffith Hamlin. Graphical input interaction technique (GIIT). *SIGGRAPH Comput. Graph.*, 17(1):5–30, 1983.

[3] Robert J. K. Jacob, Leonidas Deligiannidis, and Stephen Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Trans. Comput.-Hum. Interact.*, 6(1):1–46, 1999.

[4] Orit Shaer. A framework for building reality-based interfaces for wireless-grid applications. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1128–1129, New York, NY, USA, 2005. ACM Press.

[5] Brygg Ullmer, Hiroshi Ishii, and Robert J. K. Jacob. Token+constraint systems for tangible interaction with digital information. *ACM Trans. Comput.-Hum. Interact.*, 12(1):81–118, 2005.

[6] Otmar Hilliges, Christian Sandor, and Gudrun Klinker. Interactive prototyping for ubiquitous augmented reality user interfaces. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 285–287, New York, NY, USA, 2006. ACM Press.

[7] Robert J. K. Jacob. Open syntax: improving access for all users. In *WUAUC'01: Proceedings of the 2001 EC/NSF workshop on Universal accessibility of ubiquitous computing*, pages 84–89, New York, NY, USA, 2001. ACM Press.

[8] Daniel Klein. Developing applications with the Alpha UIMS. *interactions*, 2(4):48–65, 1995.

[9] Jean Vanderdonckt, Chow Kwok Chieu, Laurent Bouillon, and Daniela Trevisan. Model-based design, generation, and evaluation of virtual user interfaces. In *Web3D '04: Proceedings of the ninth international conference on 3D Web technology*, pages 51–60, New York, NY, USA, 2004. ACM Press.

[10] David Navarre, Philippe Palanque, Rèmi Bastide, Amèlie Schyn, Marco Winckler, Luciana Porcher Nedel, and Carla Maria Dal Sasso Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In *Human-Computer Interaction - INTERACT 2005: IFIP TC13 International Conference*, pages 170–183. Springer-Verlag GmbH, 2005.

[11] Doug A. Bowman, Ernst Kruijff, Joseph J. LaViola, and Ivan Poupyrev. *3D User Interfaces: Theory and Practice*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[12] Mark Green and Joe Lo. The Grappl 3D interaction technique library. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 16–23, New York, NY, USA, 2004. ACM Press.

[13] Wai Leng Lee and Mark Green. Automatic layout for 3D user interfaces construction. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 113–120, New York, NY, USA, 2006. ACM Press.

[14] Doug A. Bowman and Larry F. Hodges. Formalizing the design, evaluation, and application of interaction techniques for immersive virtual environments. *Journal of Visual Languages and Computing*, 10:37–53, 1999.

[15] Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61, New York, NY, USA, 2001. ACM Press.

[1] http://www.vl-e.nl/