A Visual Language for Non-WIMP User Interfaces

Robert J.K. Jacob
Department of Electrical Engineering and Computer Science
Tufts University
Medford, Mass. 02155

Abstract

Unlike current GUI or WIMP style interfaces, non-WIMP user interfaces, such as virtual environments, involve parallel, continuous interactions with the user. However, most current visual (and non-visual) languages for describing human-computer interaction are based on serial, discrete, token-based models. This paper introduces a visual language for describing and programming the fine-grained aspects of non-WIMP interaction. It is based on the notion that the essence of a non-WIMP dialogue is a set of continuous relationships, most of which are temporary. The underlying model combines a data-flow or constraint-like component for the continuous relationships with an event-based component for discrete interactions, which can enable or disable individual continuous relationships. The language thus separates non-WIMP interaction into two components, each based on existing visual language approaches, and then provides a framework for connecting the two.

Introduction

"Non-WIMP" user interfaces provide "non-command," parallel, continuous, multi-mode interaction—in contrast to current GUI or WIMP (Window, Icon, Menu, Pointer) style interfaces [11]. This interaction style can be seen most clearly in virtual reality interfaces, but its fundamental characteristics are common to a more general class of emerging user-computer environments, including new types of games, musical accompaniment systems, intelligent agent interfaces, interactive entertainment media, pen-based interfaces, eye movement-based interfaces, and ubiquitous computing [18,24]. They share a higher degree of interactivity than previous interfaces: continuous input/output exchanges occurring in parallel, rather than one single-thread dialogue.

Most current (WIMP) user interfaces are inherently serial, turn-taking ("ping-pong style") dialogues with a single input/output stream. Even where there are several devices, the input is treated conceptually as a single multiplexed stream, and interaction proceeds in half-duplex, alternating between user and computer. Users do not, for example, meaningfully move a mouse while typing characters; they do one at a time. Non-WIMP interfaces are instead characterized by *continuous* interaction between user and computer via several *parallel,* asynchronous channels or devices.

Because interaction with such systems can draw on the user's existing skills for interacting with the real world, they offer the promise of interfaces that are easier to learn and to use. However, they are currently making interfaces more difficult to build. Advances in user interface design and technology have outpaced the advances in languages and user interface management systems and tools. The result is that, today, *previous* generation command language interfaces can now be specified and implemented very effectively; *current* generation direct manipulation or WIMP interfaces are now moderately well served by user interface software tools; and the *emerging* concurrent, continuous, multi-mode non-WIMP interfaces are hardly handled at all. Most of today's examples of non-WIMP interfaces, such as virtual reality systems, have required considerable ad-hoc, low-level programming approaches, which, while very inventive, make these interfaces difficult to develop, share, and reuse.

What is needed are languages for describing and implementing these interfaces at a higher level, closer to the point of view of the user and the dialogue, rather than to the exigencies of the implementation. This paper outlines a visual language for describing and programming the fine-grained aspects of non-WIMP interaction. It is based on the notion that the essence of a non-WIMP dialogue is a set of continuous relationships, most of which are temporary. Its underlying model combines a data-flow or constraint-like component for the continuous relationships with an event-based component for discrete interactions, which can enable or disable individual continuous relationships. It separates non-WIMP interaction into two components, each of which can exploit existing visual language approaches, and provides a framework for connecting the two.

This work was supported in part by Grant No. N00014-95-1-G014 from the Naval Research Laboratory to Tufts and by Grant No. N00014-95-1-1099 from the Office of Naval Research to Tufts. We gratefully acknowledge their support.

Background

The main contribution of this approach will thus be its separation of non-WIMP interaction into two components, continuous and discrete, and its framework for communication between the two spheres, rather than the languages for describing the internals of the two components themselves. The discrete component can use a variety of existing techniques for describing discrete event handlers. The continuous component is similar to a data-flow graph or a set of one-way constraints between actual inputs and outputs. The model provides the ability to "re-wire" the graph from within the dialogue. Another goal is to keep the model simple enough to allow very fast run-time performance, with the ultimate purpose of supporting virtual reality interfaces directly.

A variety of specification languages for describing WIMP and other previous generations of user interfaces has been developed, and user interface management systems have been built based up on them [22], using approaches such as BNF or other grammar-based specifications [25], state transition diagrams [23], event handlers [12], declarative specifications [25], constraints [13], and others [10,26], including visual languages [16].

Several researchers are using constraints for describing the continuous aspect of graphical interfaces [8,9,19], and other recent work in 3D interfaces uses similar continuous approaches [3,27]. Also addressing the description of interfaces by continuous models, Mackinlay, Card, and Robertson allude to expressing interface syntax as connections between the ranges and domains of input devices and intermediate devices [21]. Lewis, Koved, and Ling have addressed non-WIMP interfaces with an elegant UIMS for virtual reality, using concurrent event-based dialogues [20].

While their focus is on widgets found in current WIMP interfaces, Abowd [1] and Carr [4,5] both present specification languages that separate the discrete and continuous spheres along the same lines as this model. Both approaches support the separation of interaction into continuous and discrete as a natural and desirable model for specifying modern interactive interfaces. Carr provides an expressive visual language for specifying this type of behavior, with different types of connections for transmitting events or value changes. Abowd provides an elegant formal specification language for describing this type of behavior, and uses the specification of a slider as a key example. He strongly emphasizes the difference between discrete and continuous, which he calls event and status, and aptly refers to temporary, continuous relationships as *interstitial* behavior, i.e., occurring in the interstices between discrete events.

A question that arises is: Why can't the problem of user interface design be solved directly by visual programming techniques? For example, why is an interactive interface builder, such as the NeXT Interface Builder or Visual Basic not sufficient? Such a solution would handle the visual *layout* of the objects in the next-generation user interface, but it would not address the problem of describing new interactive *behaviors.*

Languages for visual programming can be divided into two categories. In the first, the object being designed is itself a static graphical object—a menu, a screen layout, an engineering drawing, a typeset report, a font of type. While such objects are frequently programmed in symbolic languages (for example, a picture might be programmed as a sequence of calls to Xlib graphics subroutines), they are obvious candidates for a "what you see is what you get" mode of visual programming. A programming environment for such a visual programming language need only simulate the appearance of the final object and provide direct graphical commands for manipulating it. When the designer is satisfied with its appearance, he or she saves it and has thereby written a visual program. Such systems can combine great power with ease of use, because the visual programming language employed is a natural way to describe the graphical object. It is so natural that the system is often not considered a programming language environment at all, but simply a "what you see is what you get" style of editor. Unfortunately, this approach is only possible where there can be a one-to-one correspondence between a visual programming language and the static visual object being programmed.

A more difficult problem arises with the second category of visual programming language. Here, visual programming is used to represent something abstract, which does not have a direct graphical image—time sequence, hierarchy, conditional statements, frame-based knowledge. To provide visual programming languages for these objects, it is necessary first to devise suitable graphical representations or visual metaphors for them. The powerful principle of "what you see is what you get" is not much help, since the objects are abstract. Applying the visual programming language paradigm to these situations depends critically on choosing good representations.

Sequence or interactive behavior of a user interface—as opposed to layout—is just such a problem; we need to invent appropriate visual representations for it in order to use visual programming. Current interface builders usually handle behavior by providing a fixed set of predefined interactive behaviors. They are the familiar screen buttons, sliders, scrollbars, and other commonly-used widgets. Their interactive behaviors have been defined by the programmer of the toolkit using a conventional programming language;

the user interface designer merely decides *where* these predefined objects should be placed on the screen. It is generally difficult or impossible for him or her to change their interactive behaviors or to create new objects with new behaviors within the toolkit interface. Next-generation interfaces will introduce new objects with new types of interactive behaviors. The designer needs a scheme for designing and programming the actual, internal interactive behaviors. Such languages may well be visual, and the designer may use visual programming techniques with them [16]. But they will require new visual languages, to describe sequence or behavior, not just a picture of the layout of the screen or virtual world.

Underlying Properties of Non-WIMP Interactions

To develop a visual language for non-WIMP interactive behavior, we therefore need to identify the basic structure of such interaction as the user sees it. What is the essence of the *sequence* of interactions in a non-WIMP interface? We posit that it is *a set of continuous relationships, most of which are temporary.*

For example, in a virtual environment, a user may be able to grasp, move, and release an object. The hand position and object position are thus related by a continuous function (say, an identity mapping between the two 3D positions)—but only while the user is grasping the object. Similarly, using a scrollbar in a conventional graphical user interface, the *y* coordinate of the mouse and the region of the file being displayed are related by a continuous function (a linear scaling function, from 1D to 1D), but only while the mouse button is held down (after having first been pressed within the scrollbar handle). The continuous relationship ceases when the user releases the mouse button.

Some continuous relationships are permanent. In a conventional physical control panel, the rotational position of each knob is permanently connected to some variable. In a flight simulator, the position of the throttle lever and the setting of the throttle parameter are permanently connected by a continuous function.

The essence of these interfaces is, then, a set of continuous relationships some of which are permanent and some of which are engaged and disengaged from time to time. These relationships accept continuous input from the user and typically produce continuous responses or inputs to the system. The actions that engage or disengage them are typically discrete inputs from the user (pressing a mouse button over a widget, grasping an object).

Toward a Visual Language

Most current specification models are based on tokens or events. Their top-down, triggered quality makes them easy to program (and, in fact, everything in a typical digital computer ultimately gets translated into something with those properties). But we see in the above examples that events are the wrong model for describing some types of interactions; they are more perspicuously described by declarative relationships among continuous variables. Non-WIMP interface styles tend to have more of these kinds of interactions.

Therefore, we need to address the continuous aspect of the interface explicitly in our language. Continuous inputs have often been treated by quantizing them into a stream of “change-value” or “motion” events and then handling them as discrete tokens. Instead we want to describe continuous user interaction as a first-class element of our language. We describe these types of relationships with a data-flow graph, which connects continuous input variables to continuous application (semantic) data and, ultimately, to continuous outputs, through a network of functions and intermediate variables. The result resembles a plugboard or wiring diagram or a set of one-way constraints. It also supports parallel interaction implicitly, because it is simply a declarative specification of a set of relationships that are in principle maintained simultaneously. (Maintaining them all on a single processor within required time constraints is an important issue for the implementation, but should not appear at this level of the specification.)

This leads to a two-part description of user interaction. One part is a graph of functional relationships among continuous variables. Only a few of these relationships are typically active at one moment. The other part is a set of discrete event handlers. These event handlers can, among other actions, cause specific continuous relationships to be activated or deactivated. A key issue is how the continuous and discrete domains are connected, since a modern user interface will typically use both. The most important connection in our model is the way in which discrete events can activate or deactivate the continuous relationships. Purely discrete controls (such as pushbuttons, toggle switches, menu picks) also fit into this framework. They are described by traditional discrete techniques, such as state diagrams and are covered by the “discrete” part of our model. That part serves both to engage and disengage the continuous relationships and to handle the truly discrete interactions.

Our contribution, then, is a visual language that *combines* data-flow or constraint-like continuous relationships and token-based event handlers. Its goal is to integrate the two components and map closely to the user’s view of the fine-grained interaction in a non-WIMP interface. The basic model for it is:

- A set of continuous user interface **Variables**, some of which are directly connected to input devices, some to outputs, some to application semantics. Some variables are also used for communication within the user interface model (but possibly between the continuous and discrete components), and, finally, some variables are simply interior nodes of the graph containing intermediate results.

- A set of **Links**, which contain functions that map from continuous variables to other continuous variables. A link may be operative at all times or may be associated with a **Condition**, which allows it to be turned on and off in response to other user inputs. This ability to enable and disable portions of the data flow graph in response to user inputs is a key feature of the model.
- A set of **EventHandlers**, which respond to discrete input events. The responses may include producing outputs, setting syntactic-level variables, making procedure calls to the application semantics, and setting or clearing the Conditions, which are used to enable and disable groups of Links.

The model is cast in an object-oriented framework. **Link**, **Variable**, and **EventHandler** each have a separate class hierarchy. Their fundamental properties, along with the basic operation of the software framework (the user interface management system) are encapsulated into the three base classes; subclasses allow the specifier to define particular kinds of Links, Variables, and EventHandlers as needed. While Links and Variables are connected to each other in a graph for input and output, they comprise two disjoint trees for inheritance; this enhances the expressive power of the model.

The model provides for communication between its discrete (event handlers) and continuous (links and variables) portions in several ways:

- Communication from discrete to continuous occurs through the setting and clearing of **Conditions**, which effectively re-wire the data-flow graph.
- In some situations, there are analogue data coming in, being processed, recognized, then turned into a discrete event. This is handled by a communication path from continuous to discrete by allowing a link to generate tokens which are then processed by the event handlers. A link function might generate a token in response to one of its input variables crossing a threshold. Or it might generate a token when some complex function of its inputs becomes true. For example, if the inputs were all the parameters of the user’s fingers, a link function might attempt to recognize a particular hand posture and fire a token

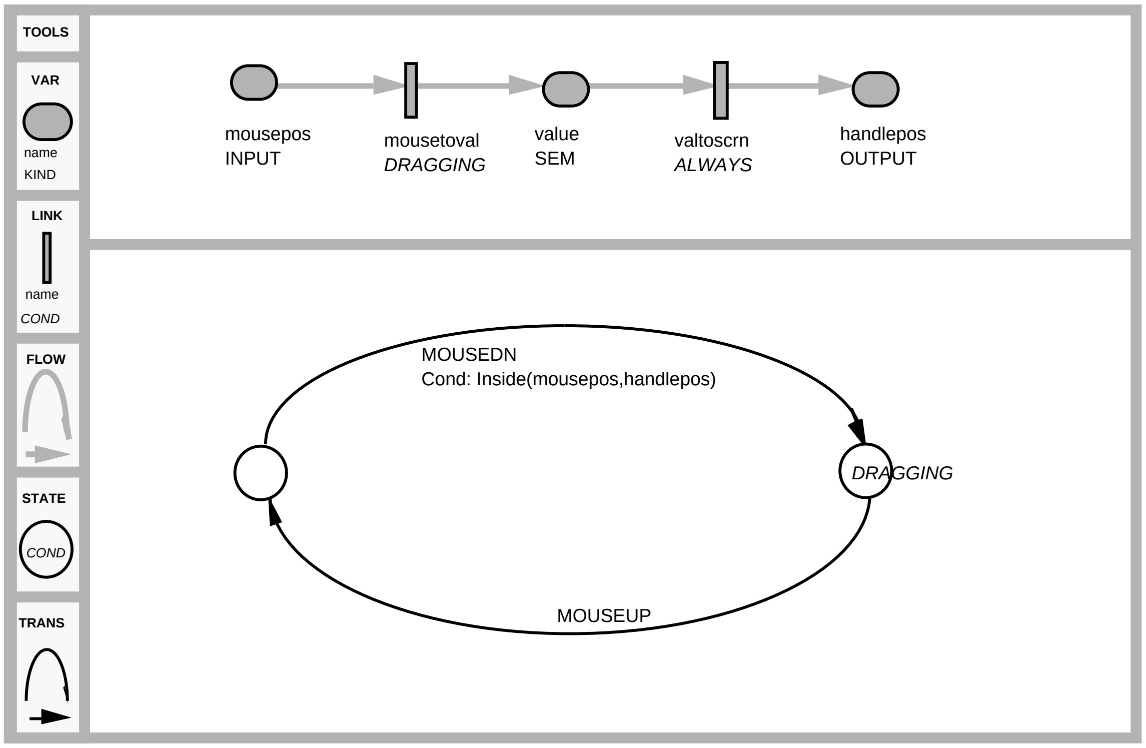

Figure 1. Specification of a simple slider, illustrating the first visual language. The upper portion of the figure shows the continuous part of the specification, using solid grey ovals to represent variables, solid grey rectangles for links, and grey arrows for data flows. The lower portion shows the event handler in the form of a state diagram, with states represented as circles and transitions as arrows.

when it was recognized.

- Finally, as with augmented transition networks and other similar schemes, we provide the ability for continuous and discrete components to set and test arbitrary user interface variables, which are accessible to both components.

A further refinement expresses the event handlers as individual state transition diagrams, which allows another method of bringing the continuous and discrete components closer together conceptually and leads to an alternate form of the visual language. Imagine that each state in the state transition diagram had an entire data-flow graph associated with it. When the system enters a state, it begins executing that data-flow graph and continues until it changes to another state. The state diagram can be viewed as a set of transitions between whole data-flow graphs. We have already provided the ability to enable and disable sets of links in a data-flow graph by explicit action. If we simply associate such sets with states, we can automatically enable and disable the links belonging to a state whenever that state is entered or exited, as if we set and cleared the conditions with explicit actions, yielding a particularly apt description of moded continuous operations (such as grab, drag, and release).

User Interface Description Language

We show the underlying model as two versions of a visual language, using a static mockup of an editor for it. To introduce them, we will use an example from a conventional WIMP interface. In this a simplified slider widget, if the user presses the mouse button down on the slider handle, the slider will begin following the *y* coordinate of the mouse, scaled appropriately. It will follow the mouse continuously, truncated to lie within the vertical range of the slider area, directly setting its associated semantic-level application variable as it moves.

We view this as a functional relationship between the *y* coordinate of the mouse and the position of the slider handle, two continuous variables (disregarding their ultimate realizations in pixel units). This relationship is temporary, however; it is only enabled while the user is dragging the slider with the mouse button down. Therefore, we provide event handlers to process the button-down and button-up events that initiate and terminate the relationship. Those events execute commands that enable and disable the continuous relationship.

Figure 1 shows the specification of this simple slider in the first version of the visual language, with the upper portion of the screen showing the continuous portion of the specification, using solid grey ovals to represent variables, solid grey rectangles for links, and grey arrows for data flows. The lower portion shows the event handler in the form of a state diagram, with states represented as circles and transitions as arrows; further details of this state diagram

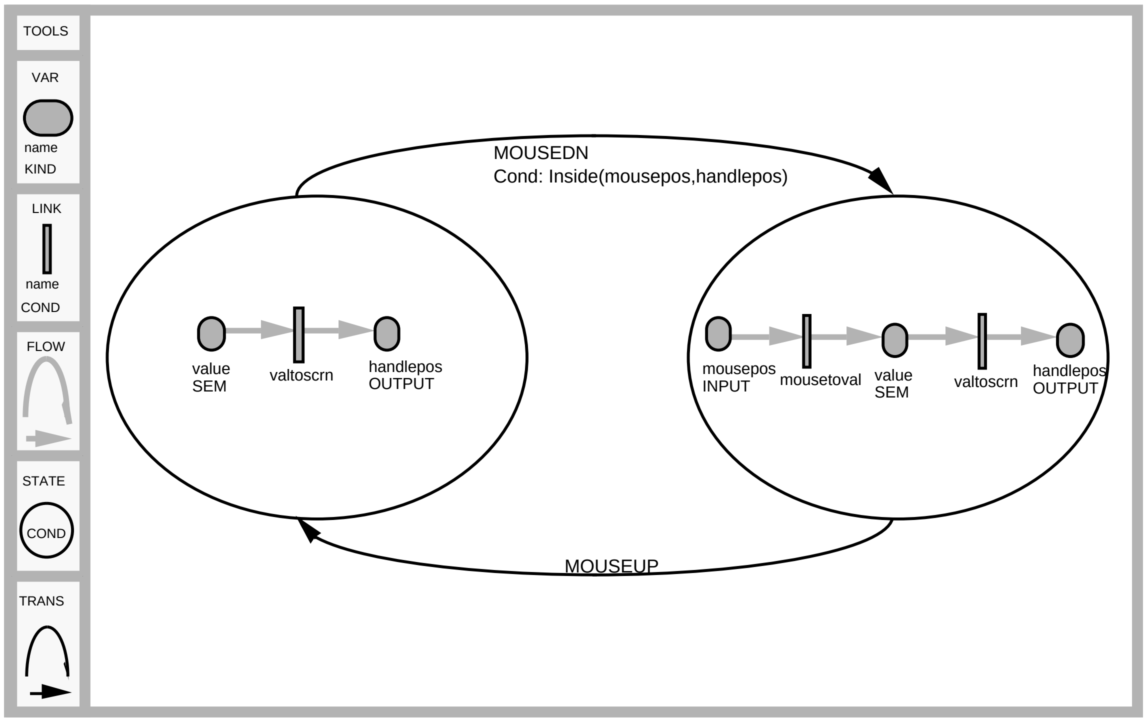

Figure 2. The same slider as in Figure 1, illustrating the second version of the visual language. Here, the large ovals represent states, and the arrows between them represent transitions. Each state contains a data-flow graph showing the data flows that are operational while the system is in that state.

notation itself are found in [15,16]. This very simple example illustrates the use of separate continuous and discrete specifications and the way in which enabling and disabling of the continuous relationships provides the connection between the two.The continuous relationship is divided into two parts. The relationship between the mouse position and the **value** variable in the application semantics is temporary, while dragging; the relationship between **value** and the displayed slider handle is permanent. Because **value** is a variable shared with the semantic level of the system, it could also be changed by the application or by function keys or other input, and the slider handle would respond. The variable **mousepos** is an input variable, which always gives the current position of the mouse; **handlepos** is an output variable, which controls the current position of the slider handle. The underlying user interface management system keeps the **mousepos** variable updated based on mouse inputs and the position of the slider handle updated based on changes in **handlepos**. The link **mousetoval** contains a simple scaling and truncating function that relates the mouse position to the value of the controlled variable; it is associated with the condition name **dragging**, so that it can be enabled and disabled by the state transition diagram. The link **valtoscrn** scales the variable **value** back to the screen position of the slider handle; it is always enabled.

The discrete portion of this specification is given in the form of a state transition diagram, although any other form of event handler specification could be used interchangeably in the underlying system. It accepts a **MOUSEDN** token that occurs within the slider handle and makes a transition to a new state, in which the **dragging** condition is enabled. As long as the state diagram remains in this state, the **mousetoval** link is enabled, and the mouse is connected to the slider handle, without the need for any further explicit specification. The **MOUSEUP** token will then trigger a transition to the initial state, causing the **dragging** condition to be disabled and hence the **mousetoval** relationship to cease being enforced automatically. (The condition names like **dragging** provide a layer of indirection that is useful when a single condition controls a set of links; in this example there is only one conditional link, **mousetoval**.)

The second form of the visual language, shown in Figure 2, unifies the two components into a single representation by considering each state in the state transition diagram to have an entire data-flow graph associated with it. As described above, this provides a particularly apt description of a moded continuous operation like engaging, dragging, and releasing the slider handle. The nested diagram approach follows that of Citrin [7], although in this case it is confined to two levels, and each level has a different syntax. One obvious drawback of this type of language is that it is difficult to scale the graphical representation to fit a more complex interface into a single static image. For interactive use, an editor that supports zooming will solve the problem. For example, Figure 3 shows the interface from Figure 2, zoomed in on the first state, with its enclosed data-flow diagram clearly visible and editable. Even better would be rapid continuous zooming, such as provided by the PAD++ system [2], or head-coupled zooming, as in the pre-screen projection technique [14].

As a further example, Figure 4 shows a two-mouse

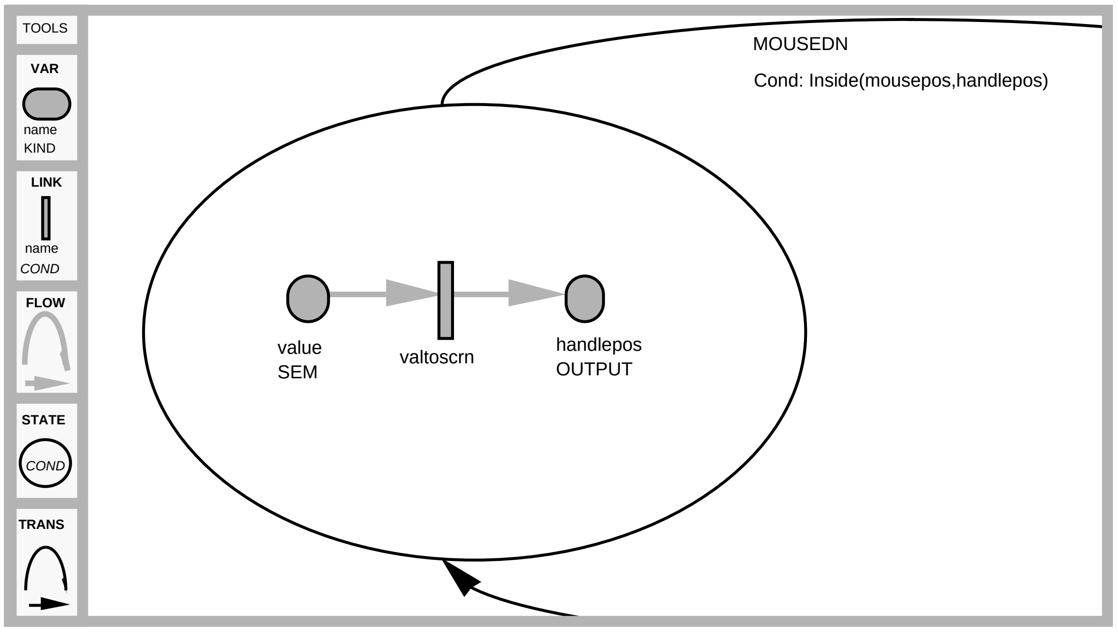

Figure 3. The example from Figure 2 after zooming in to edit the data-flow graph within one of the states.

scheme for graphical interaction. While we expect the second version of the visual language to be preferable for interfaces such as this, with few permanent links, we will use the first version here and below, because it is easier to read on a printed page. In this interface, devised by Chatty [6], dragging the right mouse normally moves a selected object, but dragging it while holding the left mouse button rotates the object around the location of the left mouse. Here, the **RIGHTMOUSEDN** token refers to pushing the button on the mouse in the user's right hand, and **LEFTMOUSEDN**, the left hand.

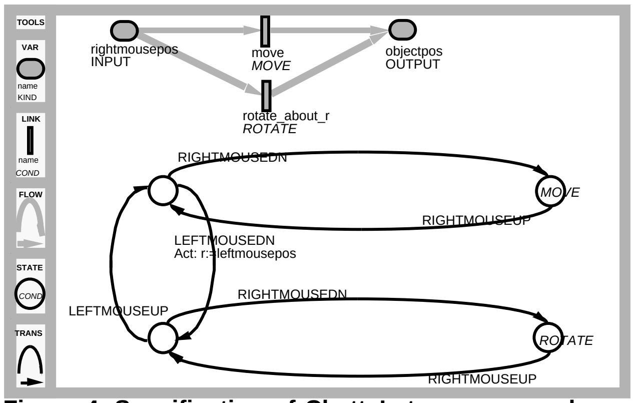

Figure 4. Specification of Chatty's two-mouse scheme for graphical interaction. Dragging the right mouse normally moves a selected object, but dragging it while holding the left mouse button rotates the object around the location of the left mouse.

Finally, to illustrate some of the kinds of interactions in a virtual environment, Figure 5 shows three simple, very common interactions. First, the position and orientation of the user's head control the position and orientation of the viewpoint at all times. Second, the user can "fly" in the direction his or her head is pointing by holding a pushbutton. Third, the user can grasp an object, move it, and release it. All of these interactions can be performed simultaneously. The two state transition diagrams shown are both active, as coroutines, using the approach described in [17]; and all data flows that are enabled execute conceptually in parallel.The user can drag an object while simultaneously flying without any changes to Figure 5.

We are testing the language by attempting to use it to program a variety of WIMP and non-WIMP interactions, as well as some non-computer interactions (such as automobile controls), which we feel future non-WIMP interfaces are likely to emulate. The language will evolve as we continue these efforts, and then our prototypes will be refined into a full-scale user interface software testbed. We have also developed a user interface management system to provide a run-time implementation of user interfaces that are described by our model, processing the data-flow graphs and state diagrams as required. Our long-term goal is to introduce higher level, cleaner user interface description languages into the non-WIMP arena, particularly for virtual environments, where performance requirements are severe. Using the testbed, we intend to demonstrate that the new languages need not compromise performance; the underlying model is free of restrictions that might prevent it from being transformed and compiled into fast runtime algorithms.

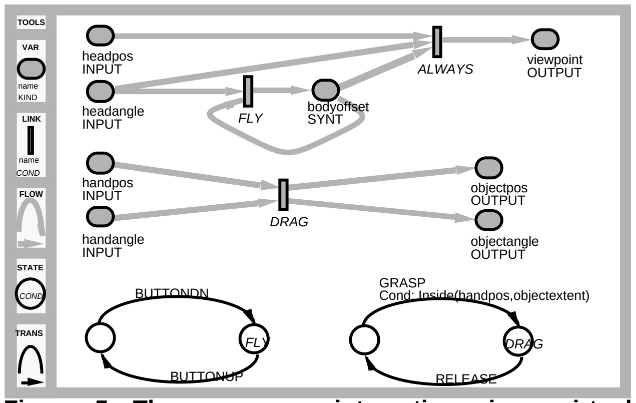

Figure 5. Three common interactions in a virtual environment: the position and orientation of the user's head control the position and orientation of the viewpoint at all times; the user can "fly" in the direction his or her head is pointing by holding a pushbutton; and the user can grasp an object, move it, and release it. The three interactions can be performed simultaneously. The two state transition diagrams shown are both active, operating as coroutines.

Acknowledgments

I want to thank my graduate students, Leonidas Deligiannidis, Quan Lin, and Stephen Morrison, for their help with this work and Linda Sibert and James Templeman for valuable discussions about these issues.

References

[1] G.D. Abowd and A.J. Dix, "Integrating Status and Event Phenomena in Formal Specifications of Interactive Systems," *Proc. ACM SIGSOFT'94 Symposium on Foundations of Software Engineering*, Addison-Wesley/ ACM Press, New Orleans, La. (1994).

[2] B.B. Bederson, L. Stead, and J.D. Hollan, "Pad++: Advances in Multiscale Interfaces," *Proc. ACM CHI'94 Human Factors in Computing Systems Conference Companion* pp. 315-316 (1994).

[3] L.D. Bergman, J.S. Richardson, D.C. Richardson, and F.P. Brooks, "VIEW - An Exploratory Molecular Visualization System with User-Definable Interaction Sequences," *Proc. ACM SIGGRAPH'93 Conference* pp. 117-126, Addison-Wesley/ACM Press (1993).

[4] D. Carr, "Specification of Interface Interaction Objects," *Proc. ACM CHI'94 Human Factors in Computing Systems Conference* pp. 372-378, Addison-Wesley/ACM Press (1994).

[5] D.A. Carr, N. Jog, H.P. Kumar, M. Teittinen, and C. Ahlberg, "Using Interaction Object Graphs to Specify and

Develop Graphical Widgets," Technical Report ISR-TR-94-69, Institute For Systems Research, University of Maryland (1994).

[6] S. Chatty, "Extending a Graphical Toolkit for Two-Handed Interaction," *Proc. ACM UIST'94 Symposium on User Interface Software and Technology* pp. 195-204, Addison-Wesley/ACM Press, Marina del Rey, Calif. (1994).

[7] W. Citrin, M. Doherty, and B. Zorn, "Design of a Completely Visual Object-Oriented Programming Language," in *Visual Object-Oriented Programming*, ed. M. Burnett, A. Goldberg, and T. Lewis, Prentice-Hall, New York (1995).

[8] C. Elliot, G. Schechter, R. Yeung, and S. Abi-Ezzi, "TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications," *Proc. ACM SIGGRAPH'94 Conference* pp. 421-434, Addison-Wesley/ACM Press (1994).

[9] M. Gleicher, "A Graphics Toolkit Based on Differential Constraints," *Proc. ACM UIST'93 Symposium on User Interface Software and Technology* pp. 109-120, Addison-Wesley/ACM Press, Atlanta, Ga. (1993).

[10] M. Green, "The University of Alberta User Interface Management System," *Computer Graphics* **19**(3) pp. 205-213 (1985).

[11] M. Green and R.J.K. Jacob, "Software Architectures and Metaphors for Non-WIMP User Interfaces," *Computer Graphics* **25**(3) pp. 229-235 (July 1991).

[12] R.D. Hill, "Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction-The Sassafras UIMS," *ACM Transactions on Graphics* **5**(3) pp. 179-210 (1986).

[13] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, and W. Wilner, "The Rendezvous Architecture and Language for Constructing Multiuser Applications," *ACM Transactions on Computer-Human Interaction* **1**(2) pp. 81-125 (June 1994).

[14] D. Hix, J.N. Templeman, and R.J.K. Jacob, "Pre-Screen Projection: From Concept to Testing of a New Interaction Technique," *Proc. ACM CHI'95 Human Factors in Computing Systems Conference* pp. 226-233, Addison-Wesley/ACM Press (1995). http://www.acm.org/sigchi/chi95/Electronic/documnts/papers/dh_bdy.htm [HTML]; http://www.cs.tufts.edu/~jacob/papers/chi95.txt [ASCII].

[15] R.J.K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Communications of the ACM* **26**(4) pp. 259-264 (1983). Also reprinted in Software Specification Techniques, ed. N. Gehani and A.D. McGettrick, Addison-Wesley, Reading, Mass, 1986, pp. 209-222.

[16] R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming," *IEEE Computer* **18**(8) pp. 51-59 (1985).

[17] R.J.K. Jacob, "A Specification Language for Direct Manipulation User Interfaces," *ACM Transactions on Graphics* **5**(4) pp. 283-317 (1986). http://www.cs.tufts.edu/~jacob/papers/tog.txt [ASCII]; http://www.cs.tufts.edu/~jacob/papers/tog.ps [Postscript].

[18] R.J.K. Jacob, "Eye Movement-Based Human-Computer Interaction Techniques: Toward Non-Command Interfaces," pp. 151-190 in *Advances in Human-Computer Interaction, Vol. 4*, ed. H.R. Hartson and D. Hix, Ablex Publishing Co., Norwood, N.J. (1993). http://www.cs.tufts.edu/~jacob/papers/hartson.txt [ASCII]; http://www.cs.tufts.edu/~jacob/papers/hartson.ps [Postscript].

[19] M. Kass, "CONDOR: Constraint-Based Dataflow," *Proc. ACM SIGGRAPH'92 Conference* pp. 321-330, Addison-Wesley/ACM Press (1992).

[20] J.B. Lewis, L. Koved, and D.T. Ling, "Dialogue Structures for Virtual Worlds," *Proc. ACM CHI'91 Human Factors in Computing Systems Conference* pp. 131-136, Addison-Wesley/ACM Press (1991).

[21] J.D. Mackinlay, S.K. Card, and G.G. Robertson, "A Semantic Analysis of the Design Space of Input Devices," *Human-Computer Interaction* **5** pp. 145-190 (1990).

[22] B.A. Myers, "User Interface Software Tools," *ACM Transactions on Computer-Human Interaction* **2**(1) pp. 64-103 (March 1995).

[23] W.M. Newman, "A System for Interactive Graphical Programming," *Proc. Spring Joint Computer Conference* pp. 47-54, AFIPS (1968).

[24] J. Nielsen, "Noncommand User Interfaces," *Comm. ACM* **36**(4) pp. 83-99 (April 1993).

[25] D.R. Olsen, *User Interface Management Systems: Models and Algorithms*, Morgan Kaufmann, San Mateo, Calif. (1992).

[26] J.L. Sibert, W.D. Hurley, and T.W. Bleser, "An Object-Oriented User Interface Management System," *Computer Graphics* **20**(4) pp. 259-268 (1986).

[27] M.P. Stevens, R.C. Zeleznik, and J.F. Hughes, "An Architecture for an Extensible 3D Interface Toolkit," *Proc. ACM UIST'94 Symposium on User Interface Software and Technology* pp. 59-67, Addison-Wesley/ACM Press, Marina del Rey, Calif. (1994).