A Software Model and Specification Language for Non-WIMP User Interfaces

Robert J.K. Jacob

Leonidas Deligiannidis

Stephen Morrison

Department of Electrical Engineering and Computer Science
Tufts University
Medford, Mass.

Abstract

We present a software model and language for describing and programming the fine-grained aspects of interaction in a non-WIMP user interface, such as a virtual environment. Our approach is based on our view that the essence of a non-WIMP dialogue is a set of continuous relationships—most of which are temporary. The 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. To demonstrate our approach, we present the PMIW user interface management system for non-WIMP interactions, a set of examples running under it, a visual editor for our user interface description language, and a discussion of our implementation and our restricted use of constraints for a performance-driven interactive situation. Our goal is to provide a model and language that captures the formal structure of non-WIMP interactions in the way that various previous techniques have captured command-based, textual, and event-based styles and to suggest that using it need not compromise real-time performance.

CR Categories and Subject Descriptors: D.2.2 **[Software Engineering]:** Tools and Techniques—*user interfaces*; H.1.2 **[Models and Principles]:** User/Machine Systems—*human*

factors; H.5.2 **[Information Interfaces and Presentation]:** User Interfaces; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism—*virtual reality*; F.3.1 **[Logics and Meanings of Programs]:** Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms: Human Factors, Languages, Design

Additional Key Words and Phrases: User interface management system (UIMS), interaction techniques, specification language, state transition diagram, virtual reality, non-WIMP interface, PMIW

1. INTRODUCTION

"Non-WIMP" user interfaces, such as virtual environments, are characterized by parallel, continuous interactions with the user. However, most current user interface description languages (UIDLs) and software systems are based on serial, discrete, token-based models. This paper proposes and tests a two-component model for describing and programming the fine-grained aspects of non-WIMP interaction. The 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. Its key ingredients are the separation of non-WIMP interaction into two components and the framework it provides for communication between the two.

As will be seen in the paper, our model abstracts away many of the details of specific input devices and treats them only in terms of the discrete events they produce and the continuous values they provide. It thus provides a high-level framework that can reduce the current difficulties of integrating novel devices into virtual reality applications, provided the

devices can be fit into our model of discrete events and/or continuous variables. Thus far we have not found one that does not fit. It will also be seen, particularly in our first example, that the model is applicable for describing elements of WIMP-style user interfaces that have continuous inputs and outputs, such as a scrollbar or slider.

This paper discusses our model and our implementation of it as follows:

- Our software model for capturing non-WIMP style interactions (Section 2), as first introduced in[35]
- A user interface description language that embodies it (Section 3)
- A programming environment we have developed for this language (Section 4)
- Some examples to illustrate the expressiveness or usefulness of the language for describing non-WIMP interactions (Section 5)
- Implementation issues and our use of constraints in virtual environments and similar high-performance interactive situations (Section 6).

1.1. Background

"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[19]. 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[33, 47]. They share a higher degree of interactivity than previous interfaces: continuous input/output exchanges occurring in parallel, rather than one single-thread, discrete-event dialogue.

Our goal is to develop and implement a model and abstraction that captures the formal structure of non-WIMP interaction in the way that existing techniques have captured command-based, textual, and event-based interaction. 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 the new 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 advances in models, languages, and user interface software 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 of necessity been designed and implemented with event-based models more suited to previous interface styles. Because those models fail to capture continuous, parallel interaction explicitly,

the interfaces have required considerable ad-hoc, low-level programming approaches. While some of these are very inventive, they have made such systems difficult to develop, share, and reuse. We seek techniques and abstractions 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.

1.2. Specifying the New Interfaces

It is not difficult to see why current specification languages and user interface management systems (UIMSs) have not been applicable to non-WIMP interaction. Consider the characteristics of current *vs.* non-WIMP interfaces:

- Single-thread input/output *vs.* parallel, asynchronous, but interrelated dialogues
- Discrete tokens *vs.* continuous and discrete inputs and responses
- Precise tokens *vs.* probabilistic input, which may be difficult to tokenize
- Sequence, not time, is meaningful *vs.* real-time requirements, deadline-based computations
- Explicit user commands *vs.* passive (“non command-based”) monitoring of the user.

On each of these counts, the characteristic of the traditional interaction styles corresponds to a characteristic of non-interactive programming languages processed by compilers. Indeed, much of current UIMS technology is built around compiler technology—processing of a single stream of discrete tokens via a single BNF (Backus-Naur Form) or ATN (Augmented Transition Network) syntax specification. Non-WIMP interfaces violate each of these assumptions and thus are not well served by compiler-based approaches.

For example, current UIMS technology typically handles multiple input devices by serializing all their inputs into one common stream. This is well suited to conventional dialogue styles but is less appropriate for styles where the inputs are logically parallel (that is, where the user thinks of what he or she is doing as two simultaneous actions). Parallel dialogues could still be programmed within the old model, but it would be preferable to be able to describe and program them in terms of logically concurrent (but sometimes interrelated) inputs, rather than a single serialized token stream. In a similar vein, it could be said that parallel processes *can* be programmed by explicitly time slicing each process. But it is unusual today to write parallel processes with explicit time slicing. Instead we write our parallel programs on top of the process abstraction, assuming parallelism. A separate layer then handles the transformation onto a single physical processor.

We seek similar kinds of abstractions for user interface software. Our model combines the applicable aspects of constraint-based and event-based user interface description languages into a framework intended to match the fine-grained properties of non-WIMP dialogues. Existing techniques could have been extended in various ad-hoc ways to describe the unusual aspects of non-WIMP dialogues. However, the real problem is not just to find *some* way to describe the user interface (since, after all, nearly any programming language could do that), but to find a language that captures the user's view of non-WIMP interaction as perspicuously as possible.

1.3. Underlying Properties of Non-WIMP Interactions

To proceed, we need to identify the basic structure of non-WIMP 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 3-D positions)—but only while the user is grasping the object. A scrollbar in a conventional graphical user interface can also be viewed this way. 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 1-D to 1-D), 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 by a continuous function (typically a linear function, mapping 1-D rotational position to 1-D). In a cockpit 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 (pressing a mouse button over a widget, grasping an object).

2. SOFTWARE MODEL

Most current specification models are based on tokens or events. Their top-down, triggered quality makes them easy to program. But we have seen that events are the wrong model for describing some of the interactions we need; they are more perspicuously described as

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 specification model. 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 model. 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. Such a model 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 issue for the implementation and is discussed below, but it should not arise at this level of the specification.)

Note that trying to describe the whole interface in purely continuous terms or purely discrete terms would be entirely possible, but inappropriate. For example:

- In the extreme, all physical actions can be viewed as continuous, but we quantize them in order to obtain discrete inputs. For example, the pressing of a keyboard key is a continuous action in space. We quantize it into two states (up and down), but there is a continuum of underlying states, we have simply grouped them so that those above some point are considered "up" and those below, "down." We could thus view a keyboard interface in continuous terms. However, we claim that the user model of keyboard input is as a discrete operation; the user thinks simply of pressing a key or not pressing it.

- Similarly, continuous actions could be viewed as discrete. All continuous inputs must ultimately be quantized in order to pass them to a digital computer. The dragging of a mouse is transmitted to the computer as a sequence of discrete moves over discrete pixel positions and, in typical window systems, processed as a sequence of individual discrete events. However, again, we claim that the user model of such input is a smooth, continuous action; the user does not think of generating individual “motion” events, but rather of making a continuous gesture.

Non-WIMP interactions (as well as some dragging interactions in WIMP interfaces, see Section 3.1) convey a sense of continuous interaction to the user, and our concern is capturing this continuous quality directly in the UIDL. At the implementation level, the hardware inputs and outputs are still realized as a series of discrete events. For example, grasping an object and moving it in 3-space appears to be a continuous interaction and ought to be specified that way in the UIDL; but, to the underlying software, it is ultimately implemented as a discrete series of input events.

This leads to a two-part model 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 main connection between the two 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 event handler part of our model. That part serves both to engage and disengage the continuous relationships as well as to handle the truly discrete interactions.

Our contribution, then, is a model for *combining* data-flow or constraint-like continuous relationships and token-based event handlers. Its goal is to provide a language that integrates the two components and maps closely to the user's view of the fine-grained interaction in a non-WIMP interface. Our model and language are intended to be independent of the choice of constraint solving mechanism used to implement it. In fact, we have built several different solvers and can use them interchangeably, as discussed in Section 6. The model, UIDL, and examples given from here through Section 5 are intended not to depend on the solver (but see Section 5.5 for discussion of one potential type of solver dependency at the UIDL level).

The basic model is:

- A set of continuous user interface **Variables**, some of which are directly connected to input devices, some to outputs, and some to application semantics. Some variables are also used for communication within the user interface model (within or between the continuous and discrete components); and 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:

- As described, 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 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. Such state diagram-based event handlers may be intermixed with other, arbitrary forms of event handlers. Using state diagram event handlers also leads to another method of integrating the continuous and discrete components. Imagine that each state in the state transition diagram had an entire data-flow graph associated with it. When the system enters that state, it begins executing that data-flow graph and continues until it changes to another state. The state diagram can then 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 associate such sets with states, we can automatically enable and disable the links belonging to a state whenever that state is entered or exited. This is simply a shorthand for setting and clearing the conditions with explicit actions, but it provides a particularly apt description of moded continuous operations (such as grab, drag, and release) and will be the basis for an alternate form of our user interface description language.

3. USER INTERFACE DESCRIPTION LANGUAGE

We have developed a language based on this model and are implementing it in several forms. The main form of the language is a visual one, for which we show an interactive graphical editor in Section 4. The language can also be used in an SGML-based text form (Section 4.2), which is intended both for user input and as an intermediate language for use by our graphical tools, or directly as a set of C++ classes.

3.1. Expository Example

To introduce the elements of the graphical version of our language, we begin by considering a simplified slider widget from a conventional WIMP interface. 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 in the form of a state transition diagram to process button-down and button-up events and enable and disable the continuous relationship.

Figure 1 shows the specification of this simple slider in our visual notation, running on our graphical editor, VRED. The upper portion of the screen shows the continuous portion of the specification using ovals to represent variables, rectangles for links, and arrows for data flows. The name of each variable is shown under its oval, and, below that, in upper case letters, its kind. The kind can be one of: **INPUT**, **OUTPUT**, **SEM**, **SYNT**, **CONST**, or **INT** to indicate its role in the user interface as, respectively: an actual device input, a variable that affects the display directly, semantic data shared with the application, syntactic data shared among components within the user interface, a constant, or a random interior node of the graph. The name of each link is shown under its rectangle and, below that, in upper case letters, the name of the condition under which it will be activated or else **ALWAYS**, meaning it is always active.

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 notation itself are found in[30, 31]. The state diagram shows, inside each state, in upper case letters, the name of a condition that is activated when this state is entered; names in lower case letters are just state names not associated with conditions. Each arc has a token and, optionally, a Boolean expression that must be true to take this transition and an action that will be executed if the transition is taken.

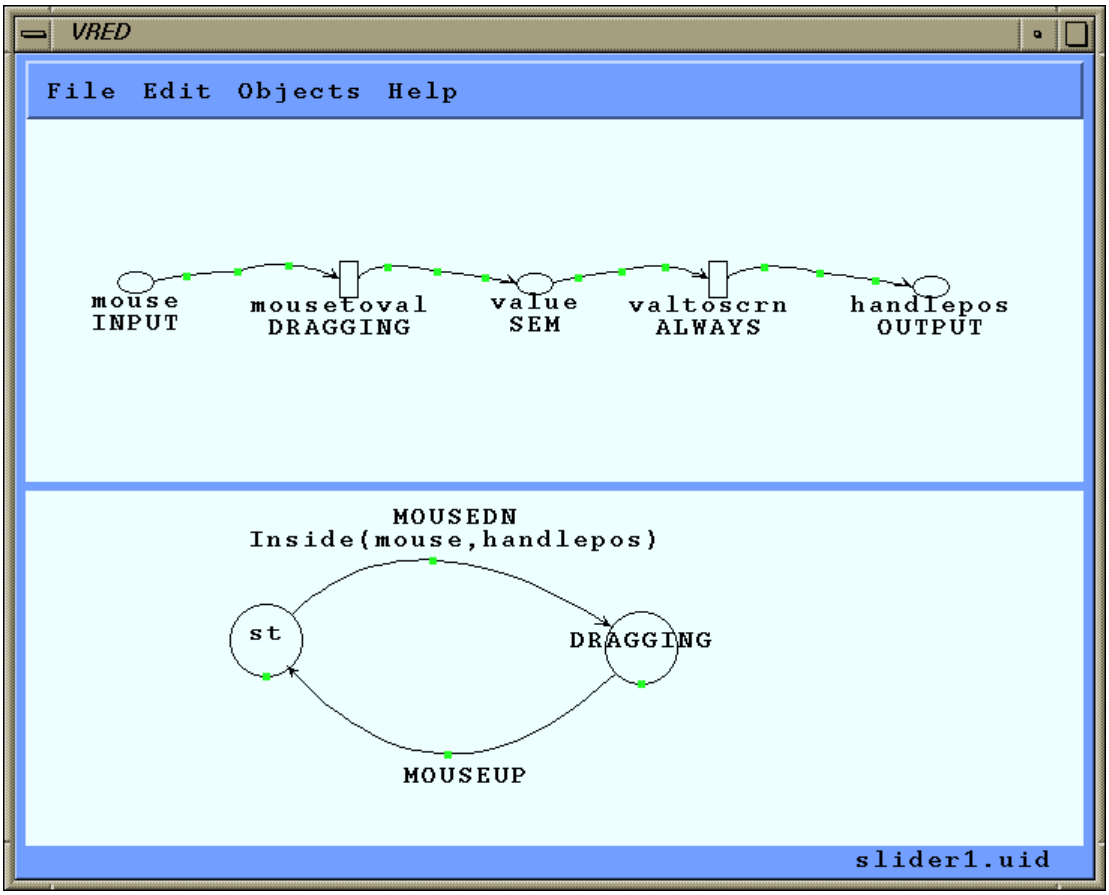

Figure 1. Specification of a simple slider, running in the VRED editor, to illustrate our graphical notation. The upper half of the screen shows the continuous portion of the specification, using ovals to represent variables, rectangles for links, and arrows for data flows. The lower portion shows the event handler in the form of a state diagram, with states represented as circles and transitions as arrows.

There is additional information, such as the types of each of the variables, the different input and output slots of each link in case it has more than one, and the body of the link itself (which is usually written as several lines of C++ code to be evaluated on demand). This information is entered and viewed through dialogue boxes for each link, variable, flow state, and transition. The layout of the elements on the screen is at the user's discretion, much like the arrangement of white space in a conventional programming language. Only the topology or connectivity of the elements in the diagram is meaningful to the run-time system.

The continuous relationship for this slider 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 might also be changed by the application or by function keys or other input, and the slider handle would still respond. The variable **mouse** is an input variable, which always gives the current location of the mouse; **handlepos** is an output variable, which controls where the slider handle is drawn on the display. The underlying user interface management system will automatically keep the **mouse** 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; the link 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 may be used interchangeably in the

underlying system. In the start state (**st**) it accepts a **MOUSEDN** token that occurs while the mouse is 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 maintained 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**. A link can also be associated with more than one condition; it would then be enabled when any of those conditions was enabled.) This very simple example illustrates the use of separate continuous and discrete specifications and the way in which the enabling and disabling of the continuous relationships by the state diagram provides the connection between the two. Abowd[1] and Carr[4, 5] also present specifications of sliders which separate their continuous and discrete aspects in different ways (see Section 7), and the Kaleidoscope constraint language[15] can support temporary constraints roughly similar to the one in this example.

Figure 2 shows an alternative form of the visual language. (Unlike Figure 1, a visual editor for this form is not implemented; Figures 2 and 3 are illustrations to describe this language, rather than editor screendumps.) This form of the language 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. When the system enters a state, it begins executing that data-flow graph and continues until it reaches another state. The state diagram can then be viewed as a set of transitions between whole data-flow graphs. As noted previously, this provides

a particularly apt description of moded continuous operations 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, a zoomable editor would address this problem, as would rapid continuous zooming, such as provided by the PAD++ system[2], or head-coupled zooming, as in the pre-screen projection technique[25]. Figure 3 shows the interface from Figure 2, zoomed in on the first state, with its enclosed data-flow diagram now visible for editing.

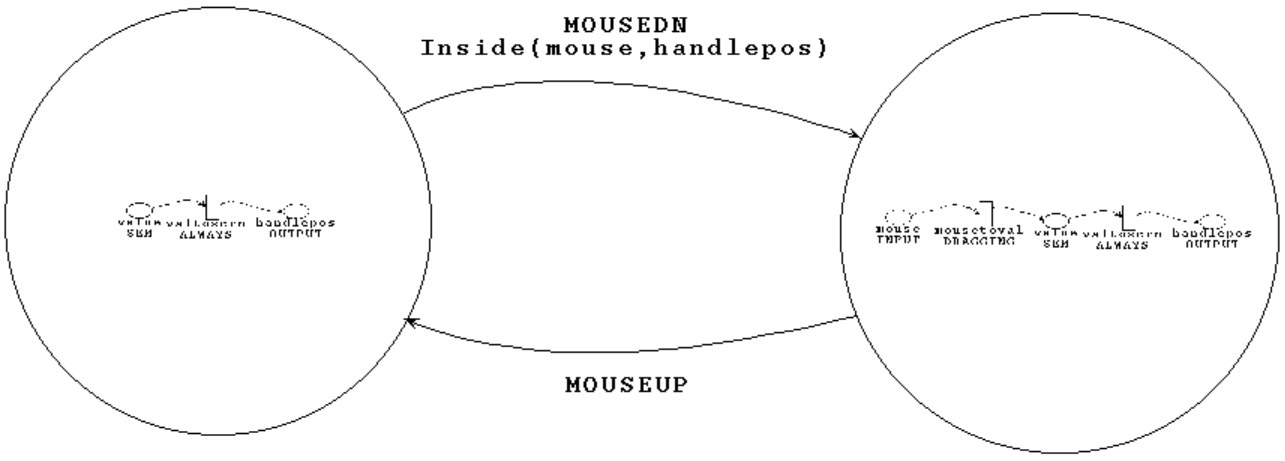

Figure 2. The same slider as in Figure 1, illustrating an alternate form of our graphical notation. Here, the large circles represent states, and the arrows between them represent transitions. Each state contains a data-flow graph showing the data flows that are operational while the system is in that state.

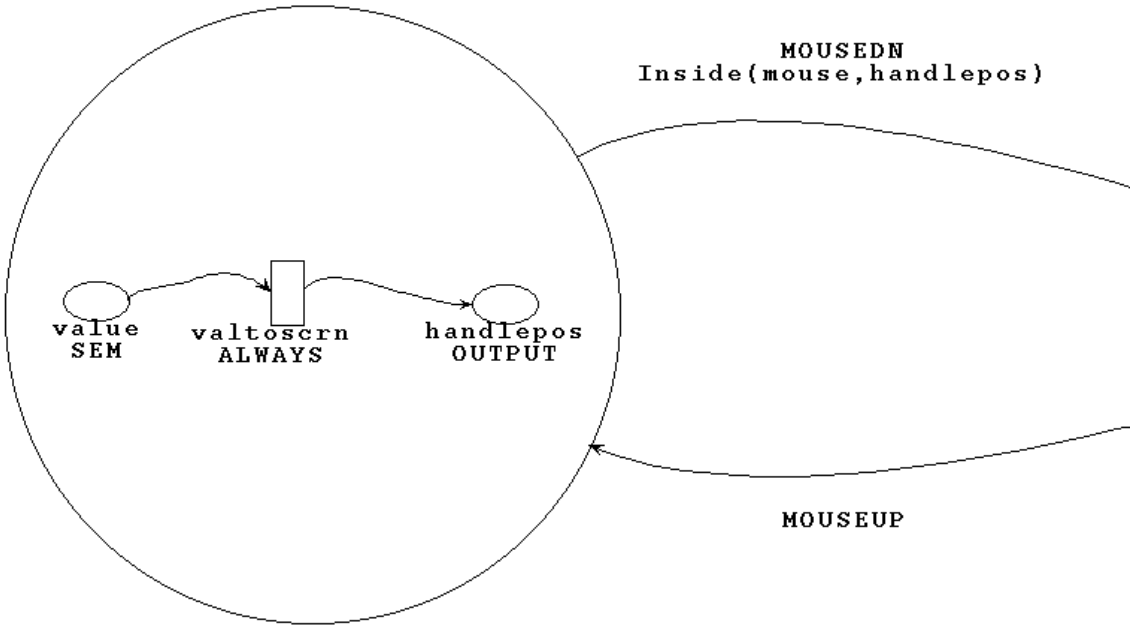

Figure 3. The example interface from Figure 2 after zooming in to edit the data-flow graph within one of the states.

4. PROGRAMMING ENVIRONMENT

4.1 The VRED Editor

The VRED editor provides a visual programming environment for our language. It implements both a data flow graph editor (also called a plugboard for its similarity to a digital logic prototyping bench) and a state diagram editor. Simple forms and menus are provided for the programmer to facilitate the entry of text based information such as element names and programmer annotations. Figure 1 as well as the UIDL figures below are all screendumps from the VRED editor, showing its base screen. The plugboard editor is the upper of the two drawing areas. Variables are represented by ellipses, links by rectangles, and data flows by arrows. Each flow also has three small rectangular handles for controlling the curvature of the arc. The first and third handle serve as smooth anchor points for splines while the middle handle is capable of

generating a cusp in the arc. The state diagram editor is the lower of the two drawing areas. States are represented by circles (with resizing handles), and transitions by arcs, labeled with their tokens, conditions, or actions.

A dialogue box can be brought up for a selected variable, link, flow, state, or transition. For a link, the dialogue box allows the user to enter a C++ code fragment as the link body; this becomes the body of the Evaluate() method. For a state transition, the dialogue box allows an optional Boolean condition (in the form of a C++ expression that returns true or false) and/or an optional action to be taken if the transition is made (as a C++ code fragment). New variables, links, flows, states, and transitions are added with menu commands and given placements (or routings, for arcs), which may be modified by dragging. When a new data flow is created, the user is prompted to select the start and end nodes node for the data flow. If one of these is a link with more than one input or output slot, a dialogue box will ask the user which of those slots the flow should be attached to. A smart delete facility automatically eliminates dangling elements.

4.2 Text Language

While the main form of our language is a visual one, Figure 4 shows its text-based form. We also use this form for dumping and restoring the information from the graphical editor and for interoperating with other tools. The language uses SGML for its meta-syntax in order to avoid introducing yet another incompatible meta-syntax into the world, since it is reasonably human-readable, and is increasingly supported by parsing and editing tools. Figure 4 illustrates the SGML-based intermediate language by showing the same example as Figure 1 in that form. It defines the variables, links, data flows between them, and state diagram(s) that make up the interface in a fairly straightforward way. Figure 4 also shows some of the information that is entered via dialogue boxes and therefore not visible in the other figures. If the **kind** attribute of a

link is “custom”, then the body of its Evaluate() routine is given in C++ directly, between the **<body>** and **</body>** delimiters. The **kind** field can also be the name of a predefined link, chosen from a library of links that perform common mathematical and geometric operations; in that case the **<body>** element would be absent. The actual contents inside a variable can be of various data types, as this example shows. The state transition diagram portion is expressed as a list of transitions from each state, similar to[30, 31]. This form of our language is translated into C++ code which is loaded along with base classes to implement the interface. The language also allows an optional **<render>** tag, not shown here, which can retain layout information generated and used only by the graphical editor; it does not affect the meaning of the non-visual language.

```
<def_system> slider1.UID

  <def_var type=Pos kind=INPUT> mouse
  </def_var>

  <def_var type=float kind=SEM> value
  </def_var>

  <def_var type=Area kind=OUTPUT> handlepos
  </def_var>

  <def_link kind=custom enableflag=DRAGGING> mousetoval
    <in type=Pos> src
    <out type=float> dst
    <body>
      dst->SetI (Scale (0.250 - src->GetI().y, 0., 0.050, 0., 100.));
    </body>
  </def_link>

  <def_link kind=custom enableflag=ALWAYS> valtoscrn
    <in type=float> src
    <out type=Area> dst
    <body>
      dst->SetI (Area ((dst->GetI()).x,
      0.250 - Scale (src->GetI(), 0., 100., 0., 0.050),
      (dst->GetI()).w, (dst->GetI()).h));
    </body>
  </def_link>

  <def_flow>
    <source> mouse
    <destination> mousetoval.src
  </def_flow>
```

```
  <def_flow>
    <source> mousetoval.dst
    <destination> value
  </def_flow>

  <def_flow>
    <source> value
    <destination> valtoscrn.src
  </def_flow>

  <def_flow>
    <source> valtoscrn.dst
    <destination> handlepos
  </def_flow>

  <def_state> st
    <transition token=MOUSEDN condition=Inside(mouse,handlepos)> DRAGGING
    </transition>
  </def_state>

  <def_state> DRAGGING
    <transition token=MOUSEUP> st
    </transition>
  </def_state>

</def_system>
```

Figure 4. Specification of the slider from Figure 1, illustrating the SGML-based intermediate language, which is used both for saving and restoring files and can be input by the user.

Our UIMS can also be used simply as a set of C++ classes (Link, Variable, etc.). This provides an object-oriented implementation of the underlying user interface management system. It can be used to build interfaces directly in C++ by subclassing to define the links needed for a particular interface and writing C++ code for their Evaluate() methods. Each form of the language, graphical, SGML, and C++, is ultimately translated into this C++ code, which runs on our PMIW user interface software testbed. The editor dumps the UIDL in its SGML-based text form. That form can be translated into C++ code that uses our UIMS classes and which can then be compiled and run directly. (Translation from SGML to C++ was formerly automatic, but at this writing it requires manual intervention, since we have updated the language but have not updated the translator to correspond; the examples below were generated, dumped, and then

manually edited before compiling.) Our overall UIMS design is intended not to preclude run-time editing of the UIDL, though the current implementation requires a compile cycle after a change in the UIDL. The run-time UIMS actually allows adding links, variables, states, and transitions at any point during execution (and deleting them provided they are not currently in use).

5. EXAMPLES

5.1. Grabbing an Object

Figure 5 shows the UIDL for a common, very simple interaction in VR: grabbing and dragging a (weird-looking) object with the hand in 3-D. Figure 6 shows the object and the hand cursor running under our PMIW system. The diamond-shaped cursor is permanently attached to the user's hand. The user can grab the object by holding Button 1 down (for simplicity, regardless of where the cursor is at the time; this is refined in the next example). While the button is held, the object position follows the cursor position; when the button is released, that relationship ceases, though the cursor continues following the user's hand.

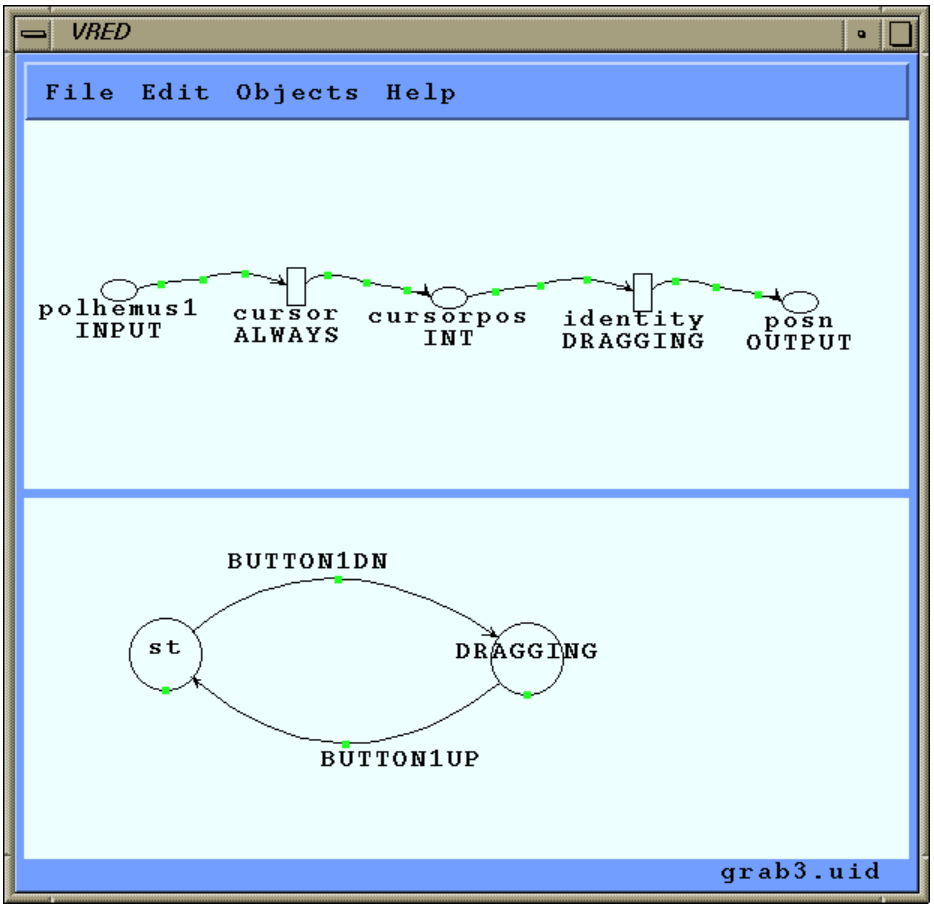

Figure 5. Grabbing and dragging an object with the hand in 3-D, a common, simple interaction in virtual reality. The user can grab the object by holding Button 1 down (for simplicity, regardless of where the cursor is at the time; see Figure 7). While the button is held, the object position follows the cursor position because the **DRAGGING** condition is enabled. When the button is released, that relationship ceases, though the cursor continues following the user's hand.

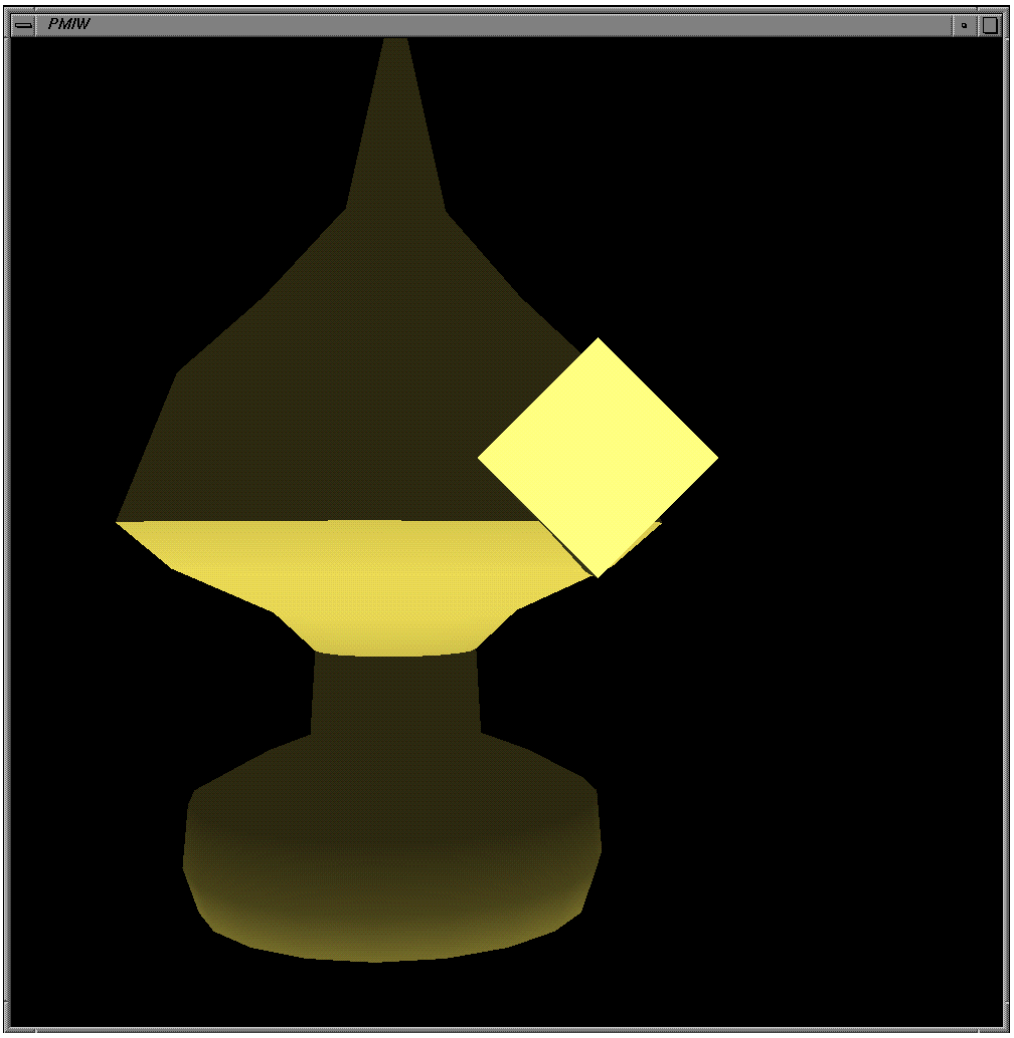

Figure 6. The object and hand cursor of Figure 5, running under our PMIW user interface management system. The diamond-shaped cursor is permanently attached to the user's hand; the other object can be grabbed and moved in 3-D.

The UIDL for this, in Figure 5, is quite straightforward. The hand (that is, the **INPUT** variable **polhemus1**, the first of the four Polhemus sensors) controls **cursorpos**, the position of the cursor, at all times; the link **cursor** is enabled **ALWAYS**. The cursor position, in turn, controls the position **posn** of the object with an identity function, but only when the condition **DRAGGING** is engaged. The condition is engaged by pressing Button 1, which causes a state transition and disengaged by releasing Button 1, which causes another transition.

It is worthwhile to note how this and subsequent examples do not require the user to write code for maintaining the relationships defined in the links nor for responding to change-value events. Because we use constraints, the user simply provides a declarative specification of the desired relationship and, if applicable, indicates how it will be turned on and off.

5.2. Arm

Figure 7 shows the UIDL for a simple movable arm, attached to a base column, and Figure 8 shows its appearance. The user can grab the arm and move it in three dimensions. The left end is always constrained to be fixed to the base column, as if it were attached by a doubly-hinged joint, while the rest of the arm can pivot to follow the user's hand cursor. **Linkc1** performs the calculations to relate the hand cursor position to the Performer rotation matrix for the movable portion of the arm. This link is active only while the user is grasping the arm; when the user lets go of the arm, the link ceases to operate and the arm remains where it was left. The state diagram shows the state change that occurs when the user grabs the arm (it activates the link) and releases the arm (deactivates the link). Unlike the simplified example in Figure 5, here the user must press the button while the hand cursor is touching the arm segment.

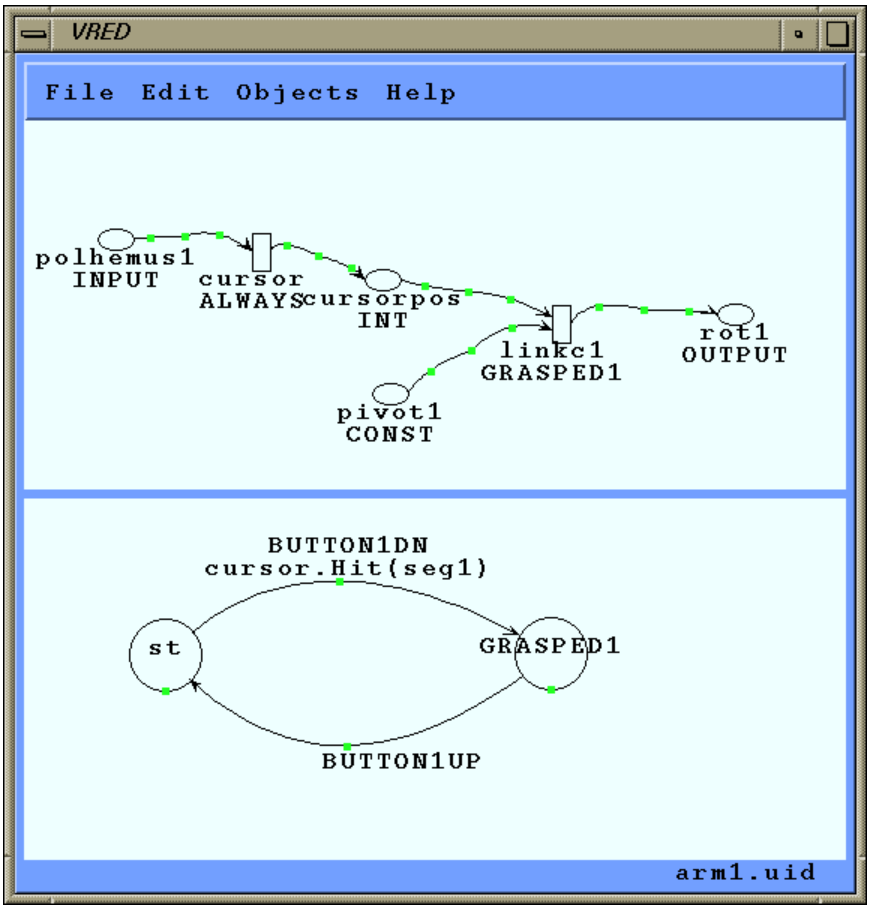

Figure 7. A simple movable arm, attached to a base column. The state diagram shows the state change that occurs when the user grabs the arm (it activates condition **GRASPED1**) and releases the arm (deactivates it). **Linkc1** relates the hand cursor position to the arm position continuously and is active only while the user is grasping the arm.

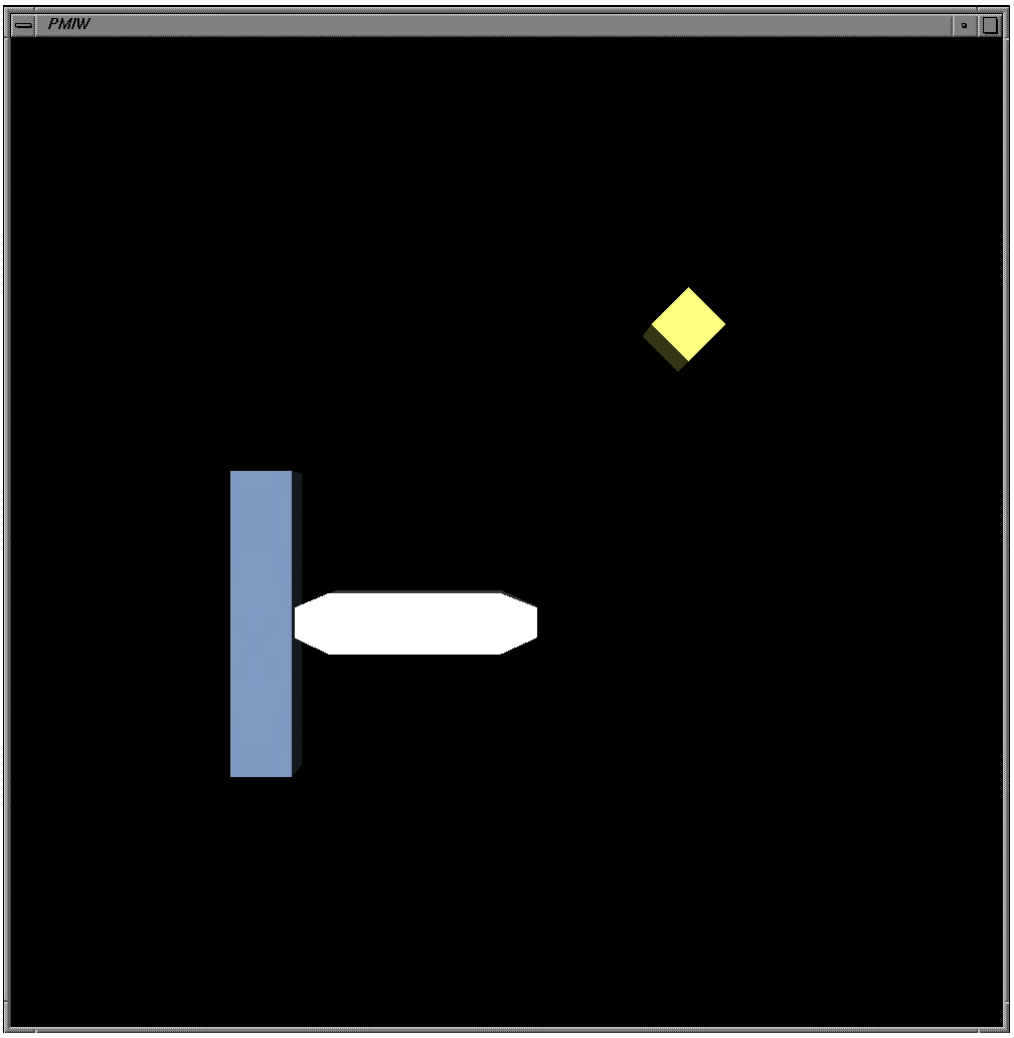

Figure 8. The arm specified in Figure 7, running on our system. The user can grab the arm and move it in three dimensions. The left end is always constrained to be fixed to the base column, as if it were attached by a doubly-hinged joint, while the rest of the arm pivots to follow the user's hand cursor.

The **pivot1** variable is a constant that contains information about the size and shape of the arm; it was generated along with the initial geometry specification for the arm. Variable **rot1** is the transform matrix that rotates the movable portion of the arm to follow the hand cursor. It is tagged as an **OUTPUT** variable because its value is directly reflected in the appearance of the screen—not necessarily because it is a sink in the data-flow graph (as will be seen in subsequent

examples). The UIMS automatically obtains updated values of all output variables and periodically redisplays the scene using their values. That relationship between an output variable and its screen display is not represented in the visual language, but it can only be a very simple, straightforward relationship (anything more complicated should first be calculated via appropriate links and variables and then fed to a simple output variable). In this example, **rot1** is simply the value of the actual 4 x 4 transform matrix contained in a DCS node of our Performer scene graph.

Although they are all shown as simple data flows, the variables in this diagram may be of different data types. For example, **polhemus1** and **cursorpos** are 3-D vectors giving *(x,y,z)* position as in Figure 5, and **rot1** and **pivot1** are 4 x 4 transform matrices. These types match the corresponding slots in the links to which they are connected. The variable type information and the names and types of the input and output slots in the links are entered and displayed in pop-up dialogue boxes in the editor, described in Section 4.1.

5.3. More Complex Arm

Figure 9 shows a two-jointed arm, to give a more interesting use of the links and variables; Figure 10 shows its appearance. To reduce clutter, **polhemus1** and **cursor** are not shown in Figure 9. As discussed below, they would be a candidate for encapsulation in a separate interaction object. The user can grab and move the first (proximal) segment of the arm as in the previous example. The second (distal) segment is attached by a hinge to the tip of the proximal segment. The user can grab the distal segment and rotate it with respect to its joint at the tip of the proximal segment. **Linkc1** is active when the hand cursor is controlling the rotation of the proximal segment of the arm (**GRASPED1** condition), and **linkc2** is active when the hand controls the distal segment (**GRASPED2**). The diagram should clearly show that, depending on

the state, the hand position controls **rot1** at some times and **rot2** at other times. Calculating the rotation of the distal segment of the arm requires knowledge of the rotation of the proximal portion of the arm. Note how **rot1** is therefore no longer a sink in the graph, but it is still an **OUTPUT** variable from our point of view because its value directly drives an element of the graphic display.

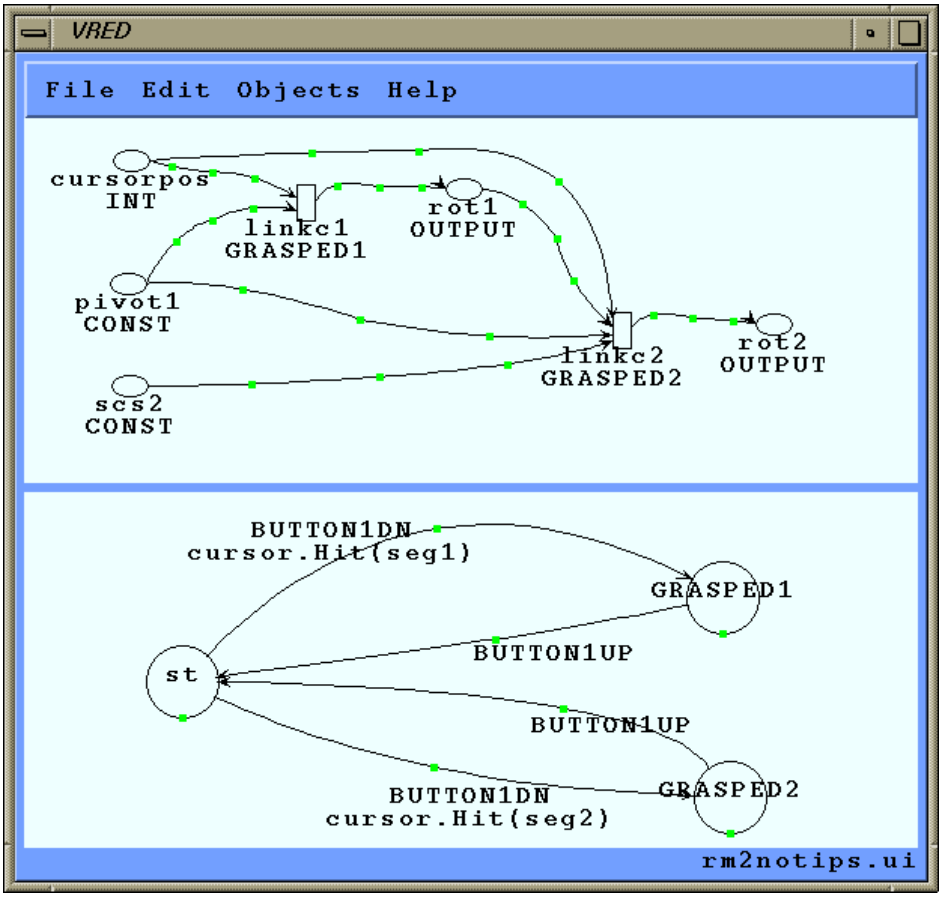

Figure 9. A two-jointed arm, using more links and variables. Here, **linkc1** is active when the hand cursor is controlling the rotation of the proximal segment of the arm (**GRASPED1** condition), and **linkc2** is active when the hand controls the distal segment (**GRASPED2**). The figure should show that, depending on the state, the hand position sometimes controls **rot1** and sometimes **rot2**. (To reduce clutter, **polhemus1** and **cursor** are not shown in this and subsequent figures; they would be the same as in Figure 7.)

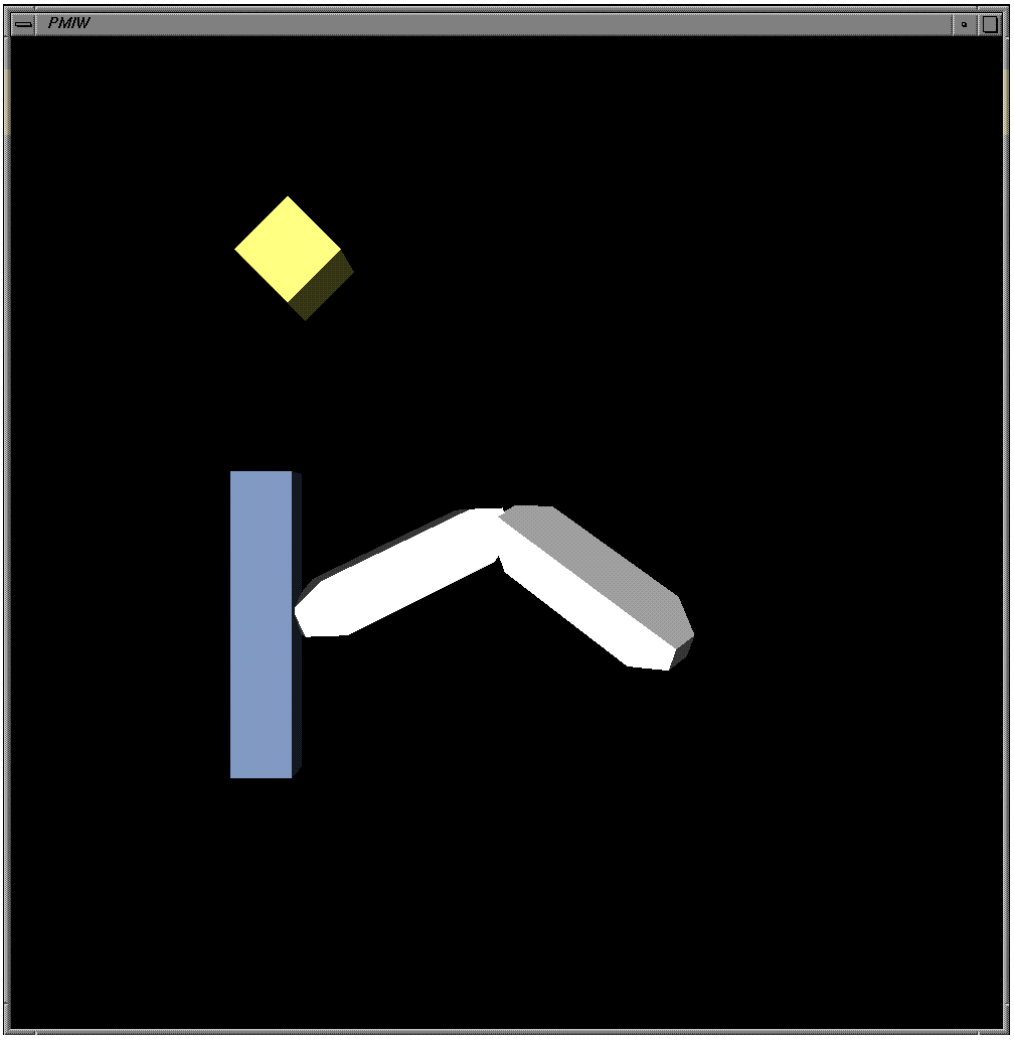

Figure 10. The arm specified in Figure 9. The user can grab and move the first (proximal) segment of the arm as in the previous example. The second (distal) segment is attached by a hinge to the tip of the proximal segment. The user can grab the distal segment and rotate it with respect to its joint at the tip of the proximal segment.

The UIDL in Figure 11 gives the same behavior as that of Figure 9 but adds two new unused variables to the graph (and stretches the limit of what can be seen on a single screen; the sub-assembly mechanism discussed below would be called for here). The new variables, **tip1** and **tip2**, are 3-D vectors that present the absolute location of the tips of this object's proximal and distal segments, respectively. This object makes them available for the use of other user

interface objects, though they have no direct bearing its own visual appearance or calculations. The links that calculate these variables are always active; that is, the variables are, conceptually, always kept up to date. However if no other links in the user interface use them, our UIMS will not devote any resources to recalculating them until they are requested. The next example makes use of these “exported” variables.

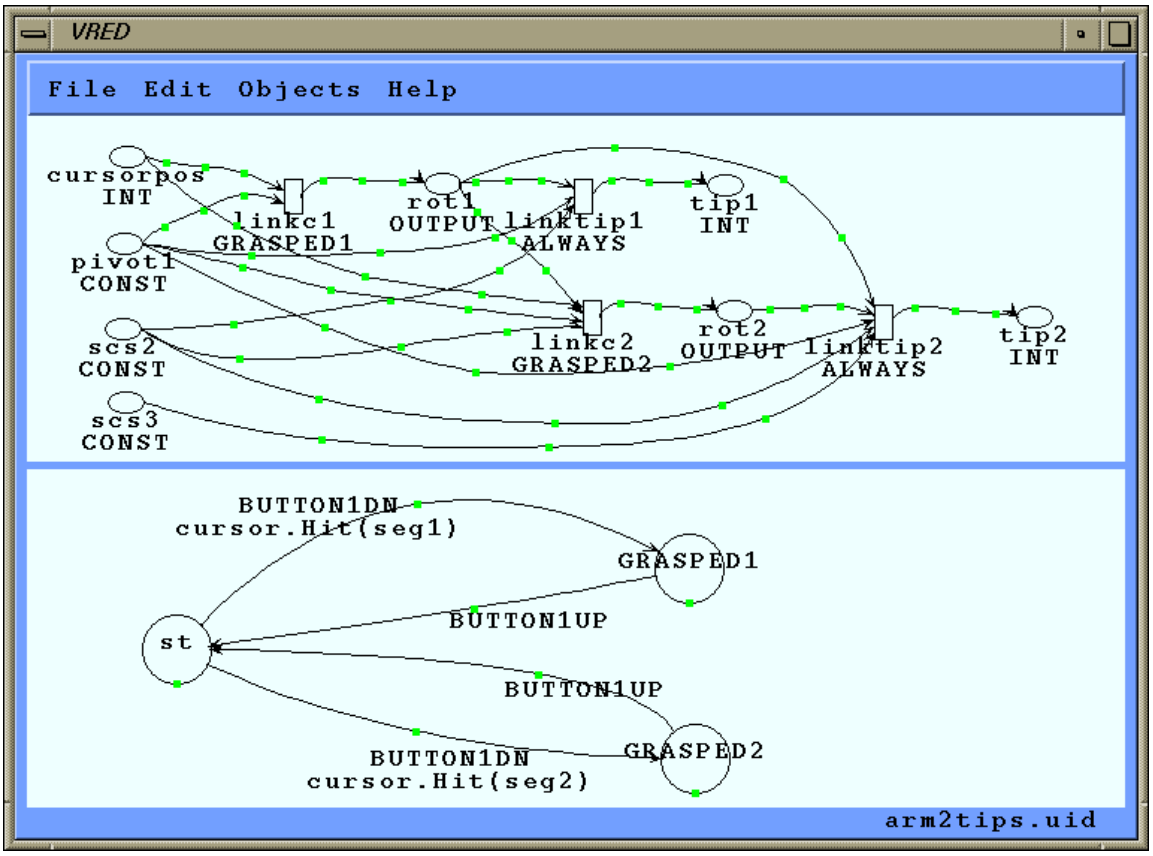

Figure 11. This object gives the same behavior as that of Figure 9 but also exports two additional variables, **tip1** and **tip2**, which are 3-D vectors giving the absolute location of the tips of this object's proximal and distal segments, respectively. They are made available for the use of other user interface objects, as seen in Figure 12, but have no direct bearing the visual appearance or calculations of this object. (This figure obviously stretches the limit of what can be seen on the screen; the sub-assembly mechanism discussed in Section 8 is called for here.)

5.4. World of Arms

Note that the position input to the simple arm of Figure 7, **cursorpos**, is simply a 3-D position vector in world coordinates. It need not be generated by the hand cursor. For example, it could be the tip position variable produced by yet another arm. In that way, we could create an arm that tries, within the constraint of the joint at its base, to point to the tip of a different arm. We have used this in order to create a more complex world, for investigating performance, as discussed in Section 6.6.

Figure 12 shows a virtual world with two instances of the two-jointed arms, each exactly like the UIDL in Figure 11. Each can be grabbed and released separately, using the hand cursor, and each generates its own rotations and its own **tip1** and **tip2** variables. We then provide a collection of 24 single-jointed arms in the background. Each of them has a link that takes one of the tips of the foreground arms as its input position variable instead of **cursorpos**. Half of the background arms point to tips of the rightmost foreground arm; the others point to the arm in the lower center foreground. Alternating background arms point to the proximal tip of their foreground arm, or to its distal tip. This could be seen by minute examination of Figure 12, but it is immediately apparent when one of the arm segments moves in the live application.

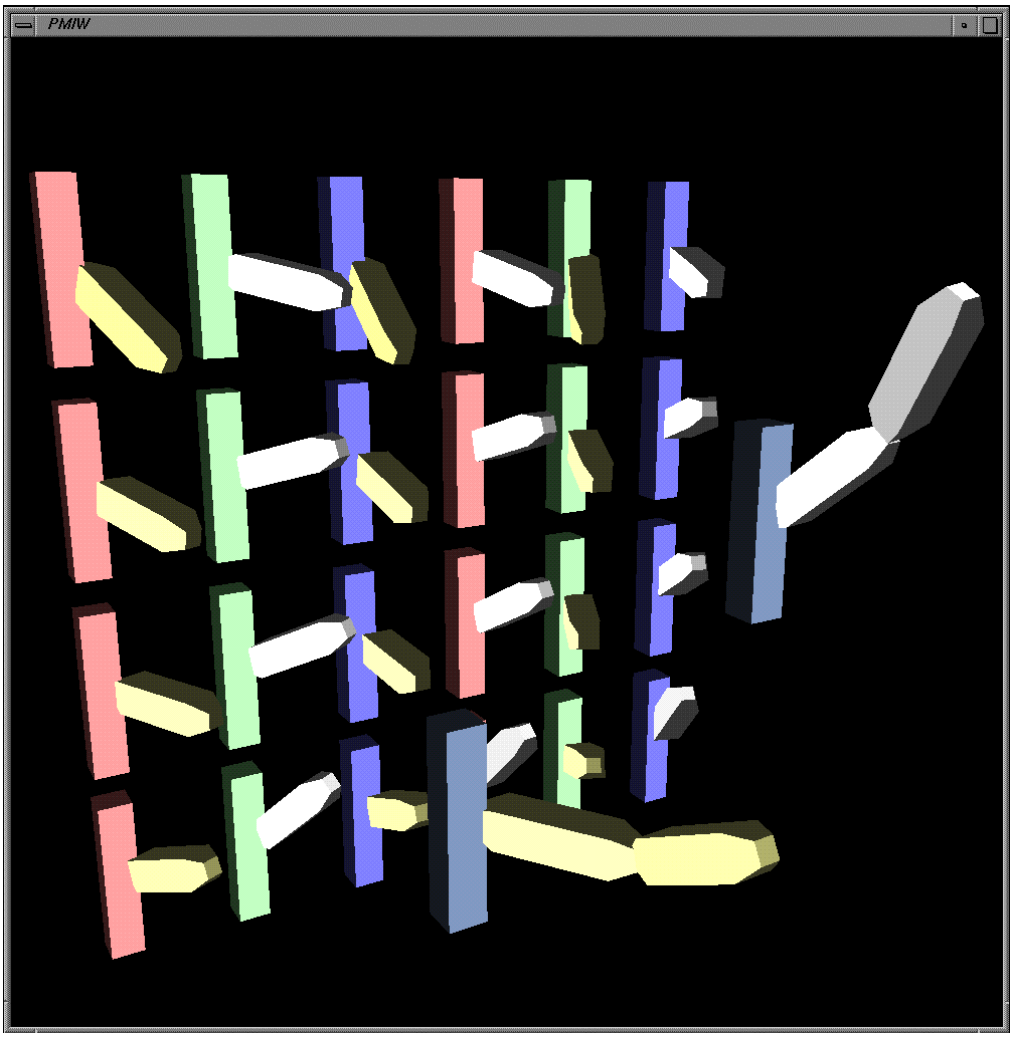

Figure 12. A world containing two instances of Figure 11 and 24 of Figure 7, with the input position variable slots of the latter set to point to the tips of the former. The two foreground arms can be grabbed and released separately, using the hand cursor, and each generates its own rotations and its own **tip1** and **tip2** variables. Each of the 24 single-jointed arms in the background takes one of the tips of the foreground arms as its input position variable instead of **cursorpos**. Some of the background arms point to tips of one of the foreground arms, some to the other. Alternating background arms point to the proximal tip of their foreground arm, or to its distal tip.

The world shown in Figure 12 was created simply by instantiating two of Figure 11 and 24 of Figure 7 and changing the input position variable pointers of the latter. In addition, the background arms were modified to add an extra feature that is not apparent from the

screendump. Each of the background arms has a state diagram that allows its **linkc1** to be turned on or off with a different keyboard key, color-coded to the colors of the base columns of the arms. This is helpful for demonstrating the performance of our system, since turning the arms on and off can be done while a foreground arm segment is moving, and it substantially changes the number of constraints to be solved without changing the amount of rendering to be done.

5.5. Virtual Environment

Figure 13 shows a more complex example of a small virtual world, containing various objects and widgets that the user can manipulate. Each object was specified individually in our UIDL and then loaded all together into the main program. Some of these objects have been described above: the cube cursor for the hand position at the upper right (Figure 5); a single-jointed arm in the foreground, far left (Figure 7); a two-jointed arm in the foreground (Figure 11); two more single-jointed arms (Figure 7) that point to two different parts of the two-jointed arm; and various grabbable objects, including the cylinders, cones, arrows, and most of the spheres visible on the display (Figure 5, with different geometry but identical UIDL). Some additional objects are introduced and discussed here: the large throwable ball at the center right of the display; the orbiting pyramid near the top of the display; and the 3-D slider, seen head-on in the lower foreground center. As we saw with the interconnected arms, objects can share constraint graph variables so that, for example, one object can control another. In this example, two of the single-jointed arms are connected to points on the two-jointed arm, and the slider is connected to control the speed of the orbiting pyramid.

The 3-D slider highlights (by changing color) when the user's 3-D cursor touches it, and it further highlights (by extending its handle outward) when the user engages it and drags the handle. The UIDL for this is a straightforward extension of Figure 1. Its state diagram keeps

track of the four possible highlighting states as well as turning the dragging on and off; the actual dragging is handled by the data flow graph, just as in Figure 1.

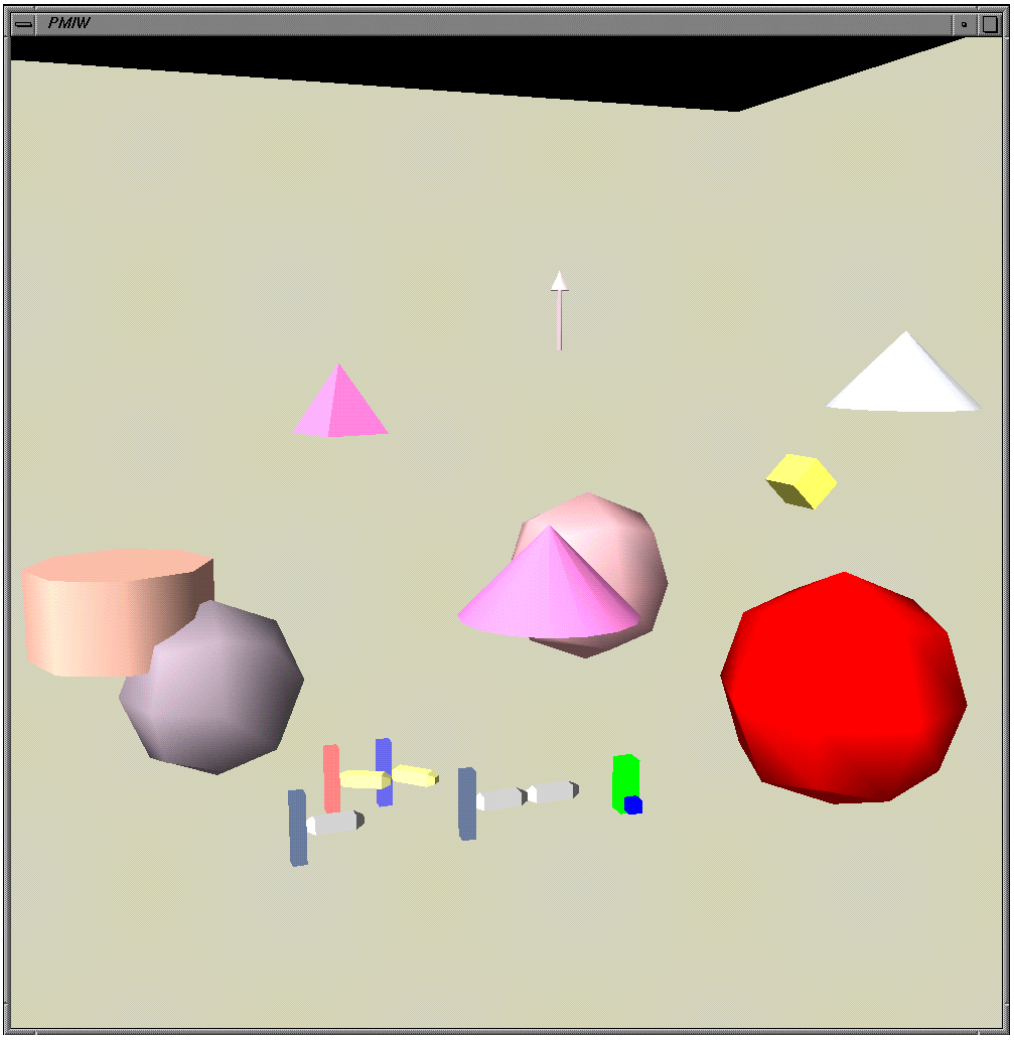

Figure 13. Screendump from a more complex virtual world, including cube cursor for the hand position at the upper right; single-jointed arm in the foreground, far left; two-jointed arm in the foreground; two single-jointed arms that point to different parts of the two-jointed arm; grabbable cylinders, cones, arrows, and spheres; a throwable ball, center right; 3-D Slider, lower foreground center, viewed head-on; and an orbiting pyramid near the top.

The large sphere is a throwable ball; the user can not only drag it but also toss it into the air. Once thrown, the object maintains its course and speed, without friction or gravity; the user can catch it and throw it again. The pyramid near the top of the frame orbits around the user. Its

speed is continuously adjustable, controlled by the slider in the foreground. The orbiting pyramid and the throwable ball are simple examples of physical simulations, introduced here in order to discuss this topic. In both cases, the output variable we send to the graphics system must be the location of the object, not its speed or direction, but our user controls are in terms of speed and direction. Our links must thus perform the calculations to convert the available data into a location, in one of two ways. The Toss object illustrates the easier case, where the location of the object is a simple function of the current clock time and the position, velocity, and time at the moment it was tossed. From those inputs, we can always calculate the current location anew, without any need for integrating over other history data. If CPU time becomes scarce and we can only evaluate this link occasionally, the motion of the ball will become jerky, but its speed will be correct; that is, it will always jump to the correct position for the current time, as is usually the preferred degradation strategy in a virtual environment[17, 52].

The orbiting object is an example of the more difficult case, because its current location depends on integrating over the entire history of the previous settings of the slider that controls its speed. Unlike the ball, we cannot calculate its correct location from time plus initial position and velocity without this history. This provides us an example of one potential dependency on the choice of constraint solver used. A data-driven or forward-chaining solver, run at frequent intervals, will handle this situation without special provision. A demand-driven or lazy evaluation solver may not, particularly if the user looks away from the pyramid, so its output is not requested for some time period. At the UIDL level, we handle this by tagging those constraints that need this special forward-chaining treatment as instances of the subclass **LinkStep**. We discuss LinkStep further in Section 6.4, but note that it is a no-op unless demand-driven evaluation is being used.

Figure 14 shows the UIDL for the orbiting object. The link **linkrot** is a subclass of **LinkStep** (though the graphical notation does not reveal this). It reads the current value of **speed**, the **SYNT** variable that is exported by the slider object, along with the current time, obtained from the **timer** input variable, which is connected to the system clock (see Section 6.2.5). The **linkrot** link contains internal variables that accumulate the inputs and integrate them over time, continuously generating an up-to-date rotation matrix, **rot**, which gives the transform to the correct current location for the orbiting pyramid.

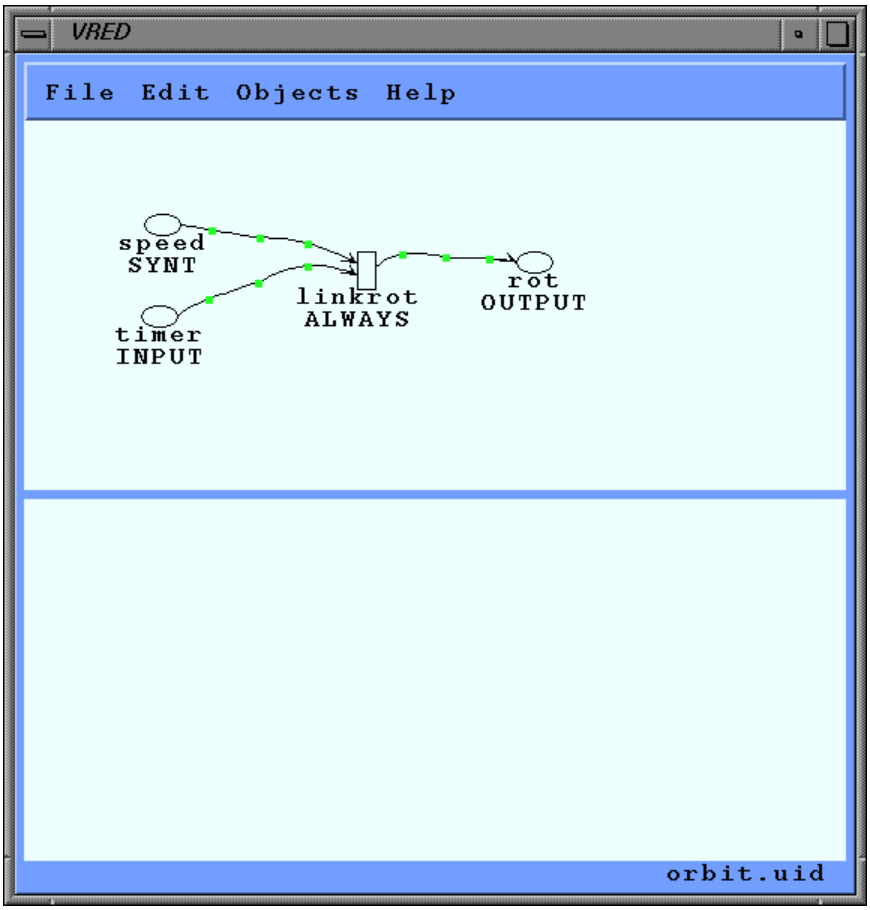

Figure 14. Specification for the orbiting pyramid, visible near the top of Figure 13. The **speed** variable is exported by the slider; the **timer** variable is connected to the system clock; and the **rot** variable contains a transform to the current location for the pyramid. This object has no discrete component.

Figure 15 shows the UIDL for the Toss example; it also shows an additional use of LinkStep. This example shows two distinct modes, one (**TOSSING**) while the ball is flying through the air and the other (**DRAGGING**) while the user is grasping it with the hand cursor. While TOSSING, we can simply calculate the position (**posn**) of the ball anew at any time, from the current time (**timer**) and the initial conditions when it was last tossed (**initPos, initTime,** and **velocity**); forward chaining with LinkStep is thus not necessary during TOSSING. However, we also need to measure the velocity of the ball at the moment it is tossed, just before we make the transition from DRAGGING to TOSSING. The Polhemus sensor measures the hand position not its velocity. Therefore, whenever the user is DRAGGING the ball, we use a LinkStep (called **savelast**) to save the last two positions of the user's hand and their timestamps (in the four variables, **last1Pos, last2Pos, last1Time,** and **last2Time,** which are shared between the plugboard and the state transitions, and used to communicate between them). At the moment the user tosses the object, the state transition calls the **StartToss()** action, which uses those most recent two saved positions and times to determine and set the initial conditions for the next toss (in **initPos, initTime,** and **velocity,** which are also shared between the plugboard and the state diagram).

Finally, not explicitly visible is the PMIW object that takes head position and orientation from the Polhemus sensor and controls viewpoint. It uses a simple set of links and variables to do this job, in the obvious way. This object is normally **Update()**d out of the usual sequence, at the last moment before rendering, as described in Section 6.3. It can also be interchanged with a mouse-driven viewpoint controller, simply by instantiating that object instead. Both produce the same output variables for viewpoint.

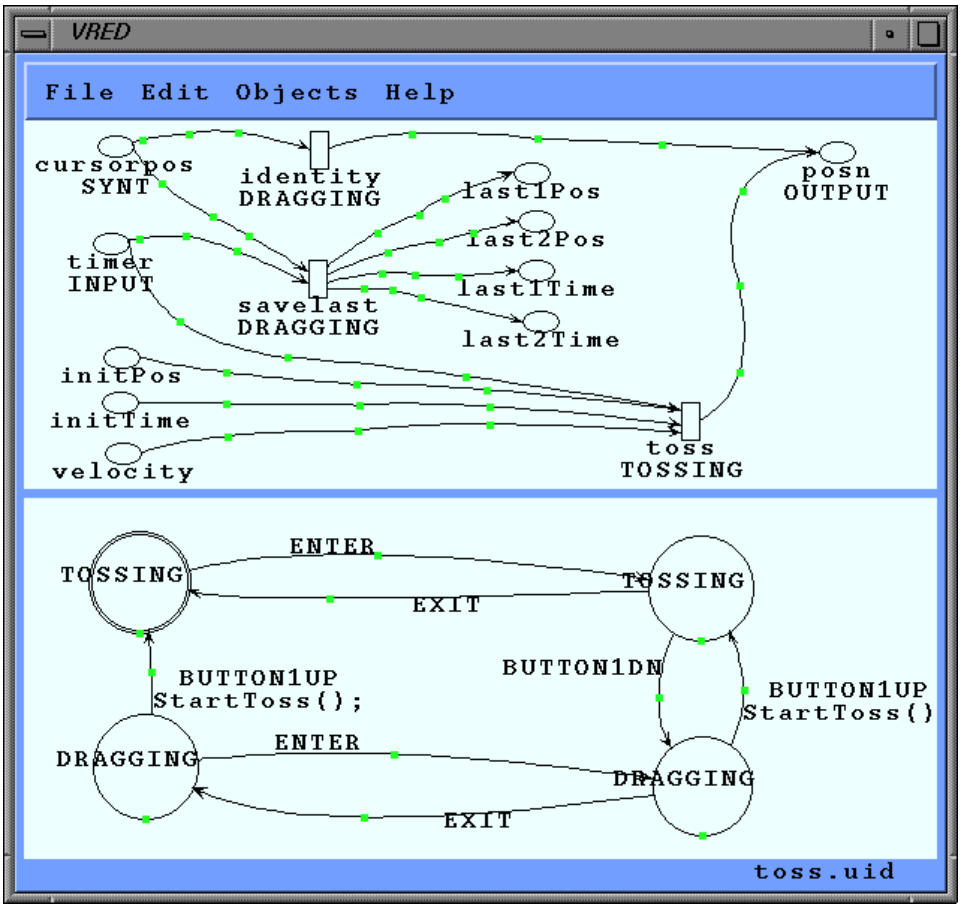

Figure 15. Specification for the throwable ball, visible in the center right of Figure 13. This interaction reveals two distinct modes, one (**TOSSING**) while the ball is flying through the air and the other (**DRAGGING**) while the user is grasping it with the hand cursor. While TOSSING, we continuously calculate the position (**posn**) from the current time (**timer**) and the initial conditions when it was last tossed (**initPos, initTime,** and **velocity**). While DRAGGING, we update the position of the ball based on the hand cursor. We also need to measure the velocity of the ball at the moment it is tossed. To do this, the **savelast** link saves the last two positions of the user's hand and their timestamps (in variables, **last1Pos, last2Pos, last1Time,** and **last2Time**); the state transition action, **StartToss()**, then uses them to determine the initial conditions for the next toss.

5.6. Interaction Techniques in Virtual Reality

In addition to those presented above, we have experimented by sketching UIDL specifications for other interaction techniques in virtual reality and, thus far, they have turned out to be surprisingly straightforward in our UIDL, like the examples shown above. The geometrical

calculations themselves within a link may be complex, but the interaction sequence or syntax, that is, what input actions cause what results, are thus far easy to express. Most grabbing or dragging interactions resemble the examples above. Some menu or selection interactions are almost entirely discrete, represented by the state transition diagram portion of the UIDL.

Figure 16 sketches the UIDL for one of the more innovative interaction techniques for VR, the daisy menu, a new type of 3-D menu developed by Jiandong Liang and Mark Green[39]. (We have not implemented the graphics for this object, just the UIDL shown in Figure 16.) This menu pops up a sphere containing the command icons around the position of the Polhemus sensor held in the user's hand. “These primitives [i.e., the menu command items] can be chosen from the shell of a 3-D spherical menu, called a *daisy*, using the bat [a Polhemus sensor with 3 buttons attached].... Primitives are selected by rotating the menu until the desired primitive enters a selection cone that always faces the user.”[20] The state transition diagram shows how the menu is activated and deactivated by Button 3. While it is activated, the links shown in the figure are all enabled, causing the sphere and the highlighted selection cone to move. The data-flow diagram shows clearly how the menu sphere and the selection cone both follow the position of the bat in the user's hand (**identity1** and **identity2**). However sphere also rotates with the bat (**identity3**), but the cone does not, it just keeps pointing toward the eye (**calcvector**).

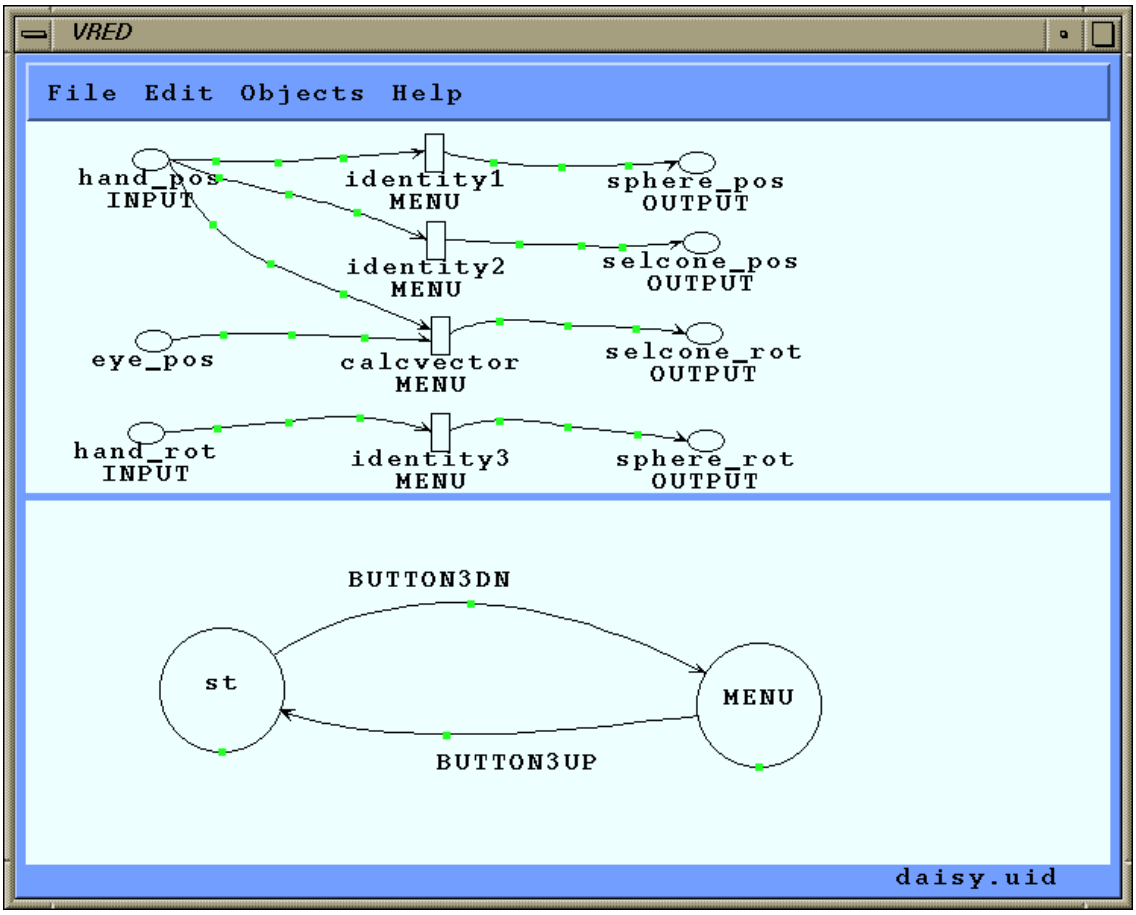

Figure 16. Specification of a 3-D "daisy menu" developed by Jiandong Liang and Mark Green for selecting commands in VR. The menu pops up a sphere containing the command icons around the position of the Polhemus sensor held in the user's hand. The state transition diagram shows how the links are all activated and deactivated by Button 3. The data-flow diagram shows that the menu sphere and the selection cone both follow the position of the user's hand (**identity1** and **identity2**). However sphere also rotates with the hand (**identity3**), but the cone does not, it just keeps pointing toward the eye (**calcvector**).

The action for the BUTTON3DN transition would be:

```
Show daisy and selection cone;
```

For BUTTON3UP transition, it is:

```
If (intersection of cone and daisy covers a menu item) {
     Select that item;
}
Hide daisy and selection cone;
```

Even the daisy menu is fairly straightforward in our UIDL. This may reflect the power of our approach, but it also reflects the relative infancy of this area of non-WIMP interaction techniques. While WIMP interfaces seem to have stabilized around a set of standard widgets, to the point where developing new interaction techniques or widgets is a rare activity, non-WIMP interfaces are a long way from this type of codification. They provide a far richer range of mechanisms to engage and communicate with the user, and the search for the right set of interaction techniques in this realm has just begun[24, 60]. For this reason, languages to enhance programming of WIMP widgets are not a major practical concern, since so few genuinely new widgets are developed. (This may be a cause-and-effect problem: Programming new widgets with current tools is much more difficult than plugging in existing widgets, creating an incentive to make do with existing ones.) We believe there will be a growing need for usable languages to help facilitate inventing and experimenting with new interaction techniques for virtual environments.

6. IMPLEMENTATION ISSUES AND THE USE OF CONSTRAINTS

6.1. UIMS Implementation

The software for our PMIW user interface management system is written in C++ for Unix. We use Silicon Graphics Performer software[51] for graphics and rendering (but not for input). PMIW is separated into portions dependent on and independent of the SGI Performer 3-D graphics system; the latter can be run on other flavors of Unix, by providing different means of

drawing output on the screen. For example, the simple slider in Figure 1 uses X graphics only and runs on Sun Unix. We use the MR toolkit[52] for communicating with the Polhemus tracker.

Our main loop reads X window input (from a conventional X window or a Performer GLX window) and then input from our own additional devices, dispatches the inputs (either to the plugboard as changes in its input variables or to the event handler as tokens), allows the plugboard to recalculate as needed, propagates changes in output variables to the Performer scene graph data, reads the head tracker, and then calls Performer to render the scene graph. The state diagram interpreter is based on one developed in previous work[31, 32]. The constraint solvers are discussed in this section.

The constraint solver is required because the system is implemented on a conventional digital computer with serial input devices, where all inputs, processing, and outputs are ultimately discrete, not continuous. Our goal is to provide a model that allows the user interface designer to think of continuous variables and a language that allows the designer to program them as though they were continuous—because we posit that the user and designer will think of some aspects of the interaction as continuous. Our runtime system ultimately merges the continuous and event-based portions of the specification and runs them in discrete steps in a single thread main loop, which includes the constraint solver. The point is that the user interface specifier need not be concerned with this level (as long as we can provide sufficiently good performance).

6.2. Fundamental Classes

Our runtime UIMS functionality is encapsulated in a set of base classes. These are described here, along with lists of the principal methods of their public interfaces. The discrete event handler is contained in the **EventHandler** class; the plugboard or constraint solver is

contained in the **Variable**, **Link**, and **Condition** classes; the connection from input devices to events and plugboard variables is contained in **DeviceBase**; and the connection from output variables to the screen is provided (by the user) in the **IO** (interaction object) class.

6.2.1. Variable

```
Variable (kind)

GetE ()
SetE (value)

GetI ()
SetI (value)
```

These are the nodes in the plugboard; they represent continuous user interface Variables, some of which are directly connected to input devices or outputs. The Variable class provides a container for holding a value, with code for determining when it needs recalculation.

Access to the value of a Variable is different depending on whether you are calling from inside a constraint solver (that is, from within the **Evaluate()** routine of a Link) or not. **GetE()** returns the value of the variable, and **SetE(value)** sets it. Inside the solver (that is, within the body of the **Evaluate()** routines), **GetI** and **SetI** are used instead. The external routine (GetE) always returns an up to date value, recalculated if necessary (but subject to the time management degradation features). The internal routine (GetI) simply accesses the value field without triggering further calculation; it is only used within a recalculation cycle.

Internally, the implementation of **Variable** is divided into a **VariableBase** class containing common routines and a template for instantiating classes to hold different data types,

such as **Variable<int>** and **Variable<float>** or aggregate data types such as **Variable<pfVec3>** and **Variable<pfMatrix>**

The **kind** tag in the constructor indicates how this variable is used. It is currently mainly for documentation; the system makes limited use of this information. The choices are:

- INPUT: Input coming directly from a device
- OUTPUT: Output destined for a device (or for the Performer scene graph)
- SEM: Semantic data, that is, data shared with the application or semantic layer, outside of the UIMS
- SYNT: Syntactic data, that is data used within the UIMS to keep track of state or other user interface information
- INT: Other intermediate node in plugboard
- CONST: Constant data (could be semantic, syntactic, or intermediate).

6.2.2. Link

```
Enable ()
Disable ()


static Recalc ()
abstract Evaluate ()
```

A Link is a part of a plugboard; it connects the nodes (Variables) and contains a function that maps from one or more variables to one or more other variables. The base constructor adds

the Link to the plugboard by adding each instance of any of its subclasses to a (static, class variable) list of all Links to be evaluated when the solver is run. The base destructor removes it from this list.

A library of common, specific kinds of Links is provided. For others, the user subclasses Link and overloads **Evaluate()**. The Evaluate routine accesses Variables in the plugboard via GetI() and SetI(). This syntactic sourness can be avoided by using the overloadings we provide for "=" and "*" for Variables or by using the SHADOW precompiler, mentioned in Section 6.7. A Link may be **Enable()**d or **Disable()**d by calling the respective methods; this is usually done via a Condition, described in the next section.

Implementation is divided into the **LinkBase** class, which contains the basic functionality of Links, irrespective of the solver used, and **Link**, which adds a specific solver implementation. The solver is triggered by calling **GetE** on a Variable. For a forward-chaining solver, this causes a complete recalculation of all dirty nodes in the graph (via the static method **Link::Recalc**). For a backward-chaining solver, only those Links needed to satisfy the particular **GetE** request are recalculated, and there would be no **Link::Recalc** method.

6.2.3. Condition

```
Add (link)
Enable ()
Disable ()
```

Condition is one of the principal communication paths from the discrete to the continuous portion of the UIDL, it effectively re-wires the data-flow graph in response to user inputs. A Link may be associated with one or more Boolean flags or **Condition**s, which allow

groups of Links to be turned on and off (usually in response to state transitions). A link may belong to any number of such groups, or none. Links may be added to a Condition with the **Add** method (or optionally in the Condition constructor). The Condition may then be **Enable()**d or **Disable()**d via the corresponding methods. **Enable** and **Disable** are not normally called by the user. Instead, a state in a state transition diagram may be associated with a **Condition**. Whenever the state diagram interpreter enters or exits that state, it will make the corresponding Enable() or Disable() call automatically.

6.2.4. EventHandler

```
static IH (token)
SendTok (token)
IhIo (token)
```

Each object that can receive tokens from discrete inputs inherits from this class. The UIMS sends each token to **EventHandler::IH**, which dispatches it to the **IhIo** methods of its individual instances. The user supplies an **IhIo** method for each instance, to receive tokens and respond to them. The responses may include making state transitions, setting syntactic-level variables, or making procedure calls to the application semantics. If the EventHandler is defined by a state machine, a precompiler generates the body of the **IhIo** method from the state transition diagram[30, 31]. Each EventHandler object remembers its state in its state transition diagram as an instance variable. Finally, EventHandlers, Links, and other objects can generate their own tokens by calling **SendTok**.

6.2.5. DeviceBase

```
Read ()
```

```
Dispatch ()
```

Subclasses of this class receive inputs and dispatch them to EventHandlers and/or input Variables, as discussed in more detail in Section 6.3. Subclasses are provided for **DeviceXWindow** (reads mouse and keyboard events for regular X windows or Performer GLX windows, which share the same input mechanism), **DevicePolhemus**, and **DeviceEye** (ISCAN eye tracker over serial port). A **DeviceTimer** is also provided to feed the current time (read from the system clock) into the plugboard.

6.2.5. Interaction Object (IO)

```
static UpdateAll ()
abstract Update ()
```

Objects that produce output Variables that must be propagated to the scene graph inherit from **IO** and supply the body of the **Update** method. The **IO** constructor maintains a list of all instances of its subclasses created; the **IO::UpdateAll()** static method then **Update()**s each of them.

6.3. Connecting with Inputs and Outputs

Our constraint solvers receive input variables, process the constraints as needed, and ultimately generate output variables. We discuss here how the input variables are connected to the actual input devices and how the output variables are connected to outputs to the screen. These aspects are outside the UIDL and hence not visible to the user interface designer; they

appear as basic capabilities hard-coded into the UIMS. There is also provision for extending the UIMS to include additional devices, by subclassing **DeviceBase**.

Each input device (including input half of a window) is encapsulated in a subclass of **DeviceBase**. While our current input devices all communicate serially, some of them are fed into the UIMS as discrete events and others as continuous variables. Each of the subclasses of **DeviceBase** provides a **Read()** method and a **Dispatch()** method; the separation is to accommodate unusual devices that might require scheduling reading and dispatching separately. **Read()** is optional; it does any required periodic servicing of the device that might be needed. For example, if device streams data continuously, this routine could drain the serial port input queue and stash the data away for us to use later. **Dispatch()** then processes the input. It may read all the device data, just the latest value of the device data, or use data saved by **Read()**. If the input generates an event (keyboard key, mouse button), **Dispatch** simply converts it to a Token and calls **EventHandler::IH(token)**, which processes the event synchronously as it is called. If, however, the input is destined for a continuous variable (mouse position, Polhemus position, eye point of regard), **Dispatch** takes the input and sends it to the **SetE()** method of the corresponding input variable. It follows that the mapping of input device data to our input Variables is coded in these **Dispatch** methods. Performing the **SetE** sets dirty flags as needed, but does not cause any other calculation until later, when the constraint solver is run. As noted in Section 6.4, there is also a provision for designated Links to be reevaluated for every new input value.

The other half of the interface to the outside world connects our continuous variables to outputs on the screen or head-mounted display (these are our only output devices at present). Under Performer, this means each of our output variables ultimately controls some data element

in the Performer scene graph. Again, since the underlying output hardware is digital, a continuous variable must ultimately be read out at a discrete time and sent to the frame buffer. Unlike the input variables, which are permanently connected to their corresponding hardware devices, the output variables differ for each application. For example, the Grab object (Figure 5) outputs the 3-D position of the grabbed object in world coordinates; Arm2 (Figure 9) produces a 4x4 rotation matrix for each of its two movable joints. For each object with output variables, the user must thus provide an **Update()** routine, which takes the values of its variables and sets the corresponding pieces of data in the Performer scene graph. Under our approach, this code should simply **GetE()** the variable and copy its value into the appropriate element of the scene graph (to which it has retained a pointer). If more substantial calculation were needed, it ought to be done in a link, where we can manage and schedule execution; the **Update()** simply copies the data to its final destination. Objects that need to be **Update()**d are all subclasses of **IO**, as described above.

All that now remains is for the main loop to perform the following steps repeatedly. Each step is described in English and also shown in C++ below:

- Call the **Read()** and then **Dispatch()** methods for each of the input devices you are using. Three devices are shown here: XWindow, for mouse and keyboard events; eye tracker, which includes the Polhemus data; and timer, a pseudo-device that reads the system clock. (See Section 6.4 for special callback argument to **deviceEye->Dispatch**):

```
pfwindow-> GetDeviceXWindow()-> Read ()
deviceEye-> Read ()
deviceTimer-> Read ()
```

```
pfwindow-> GetDeviceXWindow()-> Dispatch ()
deviceEye-> Dispatch (&LinkStep::Step)
```

```
deviceTimer-> Dispatch ()
```

- Process any **LinkStep**s specially (see Section 6.4). This iterates over all instances of **LinkStep** and allows them to recalculate as needed:

```
LinkStep::Step ()
```

- **Update()** all the **IO**s. This iterates over all instances of **IO** and calls the **Update** method. These in turn trigger recalculation as a side effect:

```
IO::UpdateAll ()
```

- Update the head position just before rendering the frame:

```
pfSync ()
headCoupler-> UpdateManual ()
pfFrame ()
```

Observe that the **Update()** routines will call **GetE()** for any variables they need to output; this triggers the constraint solver to do its work. The main loop is written to run no more than once per video frame (via the **pfSync()** and **pfFrame()** calls), but may well run less frequently. As discussed below, there is also a special provision for **LinkStep**s that should be evaluated more frequently than once per frame (those that accumulate input history from streaming input devices).

6.4. Constraints

We find thus far that constraints are indeed a good element of a UIDL for virtual reality and other non-WIMP interfaces—with the following modifications:

- They are restricted to simple one-way, non-ambiguous constraints, that is, the constraint graph is simultaneously solvable. (We will consider cases where, for execution speed, we do not solve it all; but given enough execution time the constraints are written so that they could all be satisfied simultaneously.)
- Constraints are permitted to be temporary, that is, there is an efficient mechanism for turning constraints on and off, and it can be triggered by the discrete user interface.
- The constraint specification is combined with an additional mechanism for discrete interactions and incorporated into our overall UIDL.

While we introduced the plugboard constraint graph for its expressive power in describing parallel, continuous interaction, we found that, although constraints are often viewed as introducing performance penalties compared to conventional coding, our approach provides leverage for improving performance or interactive responsiveness. This is because our constraint-based formalism allows a separation of concerns between the desired interactive behavior and the implementation mechanism. The user interface designer can thereby concentrate on and express the former, in a high-level, declarative, continuous-oriented way, while the underlying runtime system can perform optimization, tradeoffs, and conversion into discrete steps independently, beneath the level of the UIDL. One could thus tailor the response speeds of different elements of the user interface (specifically, the individual constraints in the

constraint graph) within the available computing resources from moment to moment[58]. All this would be specified separately from the user interface description; the UIDL need only describe the desired behavior (i.e., the behavior if infinite computing resources were available). By the same token, because the Links and Variables that comprise the continuous portion of our system are only a declarative specification of the desired user interface behavior, a run-time constraint solver is required to implement the interface.

The language described up to this point is intended to be independent of the particular constraint solver used, and we see our principal contribution as this model and language, rather than as introducing a new constraint solver into the world. The semantics of our UIDL make relatively modest demands on a constraint solver: a set of non-ambiguous one-way constraints, executed once per video frame, or less frequently if necessary. We have therefore implemented several quite different constraint solvers. We will discuss some of them briefly here, but since we intend the language to work with other solvers as well, the details of our current solvers are not an integral ingredient of our approach. They implement the same semantics, but have different real-time performance characteristics. They are each implemented as alternate definitions for the **Link** and **Variable** subclasses, derived from the common **LinkBase** and **VariableBase** classes. The examples here can run on any of them, with no changes to the UIDL or to any other code shown here, except for actually loading the subclasses that contain the desired solver. Note that the tagging of certain links as **LinkStep** (Section 5.5) provides extra runtime information for backward chaining solvers, and is a no-op for forward chaining ones.

Our first implementation is a correct but simple-minded forward-chaining constraint solver. Whenever the value of a variable is requested, it performs a complete recalculation of all dirty nodes in the graph. Setting a variable to a new value sets its **dirty** flag, indicating that links

that depend on this variable must be recalculated. Recalculation iterates until all flags are cleared (or a maximum count is reached, to break infinite cycles). Of course, if several variables are requested in succession, and the inputs have not changed, only one full recalculation would be executed, because it would have cleared the dirty flags.

Our next solver is LoVe a backward-chaining system, using incremental, lazy evaluation. Its algorithm is based on that of Hudson's Eval/Vite system[26, 28]. This is an optimal algorithm, which evaluates the minimal set of links for each request. LoVe accepts cyclic graphs, i.e., one can define a relationship $x = x' + 4$ where x' is the previous value of x. When the value of a variable is requested, LoVe recursively finds all the links that need to be re-evaluated and evaluates them in the correct order (so that each link is evaluated at most once). Because the algorithm is incremental, there may be some links that don't need to be re-evaluated because their dependencies are up-to-date. As LoVe finds links to be re-evaluated, it also marks them as **visited** so that it will know when a cycle exists in the graph, and cease recursion. The algorithm clears the **dirty** flag of the variables that depend on each link that is re-evaluated, to record the fact that these variables are now up to date. LoVe adds some new aspects to the algorithm of Eval/Vite, which are specific to our application, such as the ability to turn Conditions on or off without reinitializing and a set of hooks to accommodate time management features. LoVe is set up to allow its user to change the constraint graph at run time via Conditions, thereby redefining the relationships of the variables and modifying the behavior of the system dynamically. When it is done, the algorithm and the data structure remain intact; there is thus no penalty for changing the graph as often as desired at run-time.

Despite the performance benefits of a demand-driven solver, there are a few cases where a formula is easier, or simply becomes possible, to express if a forward-chaining solver could be

assumed. With lazy evaluation, we cannot be sure that a particular output variable will be calculated on every frame, and, therefore, that those constraints that feed it directly or indirectly will be evaluated every frame. We have seen examples of cases where it is more straightforward to express an algorithm if we can assume that certain constraints are called fairly regularly, regardless of the state of the demand-driven solver. As mentioned in Section 5.5, we support this by defining the **LinkStep** subclass; instances of this subclass are called more regularly than ordinary links, that is, regardless of whether the solver needs their output. There are three situations where this is necessary:

- Input history-dependent calculations, such as the Orbit example in Section 5.5.
- Simulations. For example, a simple physical simulation involving colliding objects is often easier to write if you do not have to predict ahead when the moving objects are going to touch, but just write code to step through the simulation at a fixed time step.
- Recognizer links, which scan the input stream for patterns (gestures, eye fixations) and fire tokens when they are recognized.

As seen in the main loop above, **LinkStep**s are evaluated explicitly once on each cycle.

Finally, we allow for LinkStep processing that should be done more often than once per frame. Links that process streaming input and search it for patterns (such as gesture or eye fixation recognizers) or accumulate input history should be executed for every new input value that is handled. They are handled by the optional callback argument to **Dispatch** (note the call to **deviceEye->Dispatch(&LinkStep::Step)** above. Dispatch then calls the given routine on every new input value it handles; **LinkStep::Step()** evaluates the LinkStep links as needed for it.

We are also experimenting with DLoVe, a solver that distributes the constraint solving workload over several computers to improve speed. As with the other solvers, it is designed to provide identical semantics but use parallel processing to improve performance.

Finally, the SHADOW system[41, 42] incorporates its own solver along similar lines, along with features for time management and runtime decimation of link bodies.

6.5. Multi-user Interfaces

The principal use of the distributed constraint solver, DLoVe, is to exploit the computational power of additional workstations to solve the Links and Variables faster. However, it can also be used to support multi-user, collaborative virtual worlds. Using DLoVe, the dataflow graph (that is, all Links, Variables, and Conditions) is automatically shared among all workstations. Updates made on one workstation will (eventually) be seen by all of them. This makes it easy to implement a multi-user interface. The UIDL consists of a single set of state diagrams and dataflow graphs, which together describe what happens in response to inputs from users on all of the devices and workstations.

From the interface designer's point of view, the only difference is that the UIDL for a meaningful multi-user interface will typically need to refer to inputs from the different users individually. A mouse or Polhemus on one workstation must thus be named differently from those on another; the UIDL may refer to either or both as required to describe a two-person interaction. The UIDL code for multi-user interfaces should refer to the mouse of each user specifically. In some cases, the users may have different roles in the collaborative interaction (pitcher vs. catcher). To accommodate this, the (shared) dataflow graph thus contains an array of Variables, mice[0..N], one for the mouse of each workstation, and the tokens for mouse buttons on the different mice are named LEFTDN_1, LEFTDN_2, etc. For example, we have developed

two-person versions of both Grab and Toss. In the Grab example, the state transition diagram is written so that while one user is dragging the object, the other user can grab it away. The state diagram keeps track of whose turn it is. For variety, in the two-person Toss example, when one user is dragging the object, the other user cannot grab it away. One user can throw the object and either can then catch it. These were simple extensions of the UIDL shown in Figures 5 and 15. No other special coding was required, other than handling the multiple input devices as described above and launching the distributed version of the system on at least two workstations. (Additional workstations may participate in the calculation workload, but this UIDL calls for only two mice, so any other mice would be ignored.)

6.6. Performance

Many non-WIMP interfaces must meet severe performance requirements in order to maintain their perceptual illusions. For virtual reality, in particular, these requirements are the driving force behind the design of most current implementations[51]. We want to introduce higher level, cleaner user interface description languages into this field, but we must not compromise performance. We claim that our underlying model contains nothing that adversely impacts run-time performance—all penalties are paid at compile-time—because the links or constraints are one-way and the event handler technology is straightforward. Keeping the model conceptually simple leaves some degrees of freedom available to our run-time system to manage CPU resources in a specialized way tuned to the peculiarities of a video-driven VR system. The simple homogeneous system of links also allows us to build optimized constraint systems underneath it, providing the same semantics and incorporating optimizations invisible at the UIDL level, as seen here. We thus try to separate the concerns of interface modeling and run-time optimization. We obtained a rough subjective impression of performance under LoVe by

modifying the world in Figure 12 to increase the complexity of its constraint graph, by making each arm point to its immediate neighbor in a chain instead of to one of the foreground arms. Running on an SGI Indigo2 Extreme workstation with a single 200 MHZ IP22 CPU, running IRIX 5.3 and Performer 1.2 and our non-distributed LoVe solver, we grasp one of the foreground arms with the cursor (attached to the mouse) and move it rapidly. We found that the frame rate (reported by **pfDrawChanStats()** call to Performer) varied between 24 and 36 Hz., and the subjective effect was of very rapid response. This rate was maintained with the input cursor stationary or moving rapidly and with some Conditions turned off or all turned on. The response to turning a Condition on or off seems subjectively instantaneous.

6.7. SHADOW: Scaling Up to Larger Worlds

We are developing a more ambitious system based on this approach[41, 42]. SHADOW uses the same two-part model, combining one-way constraints for its continuous part with state transition diagrams for its discrete part. It adds further enhancements to the UIDL, particularly in the area of specifying a hierarchy of objects within other objects. It allows entire subsystems defined in their own UIDL specifications to be plugged into a larger constraint graph and activated or deactivated by its event handler. We have demonstrated its scalability by developing a nontrivial “rookery” VR world with it. This fairly complex world of penguins and ice floes required 24 visual program diagrams and a total of 4000 lines of C++ code in our language. Each diagram is a relatively simple, self-contained specification, containing an average of 2.3 states, 1.9 state transitions, and 4.1 constraints. Of the 4000 lines of code, 2000 were array initialization for the vertices of the graphic objects (which would normally have been generated with a 3-D modeling tool) and 1000 were library modules for device and window handling (which would not be written anew for another world). The remaining 1000 lines of “real” code were contained

in relatively small, self-contained modules (average procedure body = 24.5 lines, average constraint body = 11.7 lines), and had an average Cyclomatic Complexity measure of 3.0 (where a lower value indicates better code maintainability, and values under 10 to 20 are generally considered good).

7. RELATED WORK

The contribution our model makes for non-WIMP interaction is its separation of the interaction into two components, continuous and discrete, and its framework for communication between the two spheres—more than the internals of the two components themselves, which draw on existing techniques. We first separate non-WIMP interaction into continuous and discrete components, then, within each of the two spheres, we build on different threads of previous user interface software research. The discrete component draws on research in event-driven user interface management systems[48]. The continuous component is similar to a data-flow graph or a set of one-way constraints between actual inputs and outputs and draws on research in constraint systems[26, 28]. The model provides the ability to “re-wire” the graph from within the dialogue.

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[45], using approaches such as BNF or other grammar-based specifications[48-50, 53], state transition diagrams[29, 46], event handlers[13, 21], declarative specifications[48], frames[57], and others[14, 18, 36, 43, 54, 61]. For example, although BNF had been a good match for programming languages or batch command interfaces, interactive, moded graphical interfaces were perhaps better captured by state transition diagram-based approaches[30]; and modern modeless WIMP interfaces fit a coroutine-based model[32].

In the continuous domain, several researchers are using constraints for 2-D graphical interfaces[22, 23, 27, 44, 59]. Kaleidoscope[15] is a constraint-based language motivated by 2-D WIMP interfaces, and it explicitly supports temporary constraints. The CONDOR system uses a constraint or data-flow model to describe interactive 3-D graphics[37]. TBAG also uses constraints effectively for graphics and animation in the interface[11]. Gleicher provides constraints that are turned on and off by events[16]. Other recent work in 3-D interfaces uses a continuous approach[55] or a discrete, but data-driven approach[3]. Mackinlay, Card, and Robertson address the description of interfaces by continuous models by discussing interface syntax as a set of connections between the ranges and domains of input devices and intermediate devices[40].

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 graphical syntax 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. Kearney and Cremer[9, 10] also use an approach that combines discrete events with continuous data flows to program a sophisticated virtual environment automobile driving simulator.

Other work from the formal specifications area is also relevant, such as that of Sufrin and He[56]; and Zave and Jackson[62], who provide a formal basis for combining multiple specification techniques to describe a single system in a coherent way. Myers' Interactors[34] also combine discrete state changes with what might be viewed as continuous actions. The continuous actions are handled as sequences of state transitions by providing a transition from a state back to itself, which accepts a “mouse-motion” or “value-changed” input token. Hill[21] and Chatty[6] have both provided UIDLs that handle parallel input from multiple input devices, particularly suited to two-handed input; their approaches are both based on discrete events.

Software architectures for virtual reality interfaces have been developed by Feiner and colleagues[12] and by Pausch and colleagues[8]. Green and colleagues developed a toolkit for building virtual reality systems[52]. Most of this work has thus far concentrated on the architecture or toolkit level, rather the user interface description language. Lewis, Koved, and Ling, addressed non-WIMP interfaces with one of the first UIMSs for virtual reality, using concurrent event-based dialogues[38].

8. CONCLUSIONS

We have presented a software model for describing and programming the fine-grained aspects of non-WIMP style interactions (Section 2) and a UIDL that embodies it (3). 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 and provides a framework for connecting the two. To exercise our new model, we have then presented:

- The VRED visual editor for this language (Section 4)
- Some simple examples, running under our PMIW UIMS, to illustrate the applicability of the language for describing non-WIMP interactions (5)
- Our use of constraints in a performance-driven interactive situation and our LoVe and DLoVe constraint systems (6).

Software for virtual environments usually uses one or both of two models: event queues or device polling. Our constraint- or plugboard-based model is slightly different from both of these. Writing a constraint or data-flow graph is different in a small but important way from writing code that says “Whenever variable X changes, do the following calculations.” Instead, the constraint expresses something closer to: “Try to maintain the following relationship to input X, whenever you have time, doing the best you can.” The distinction becomes meaningful when the system is short on time, which is often the case in a virtual environment. We believe this is closer to the user's view of the situation and closer to what the programmer would like to express. Experience with our examples showed the straightforwardness of this declarative specification. The interface designer writes no code to maintain the relationship or handle the change-value events, but simply declares a relationship and then turns it on or off as desired. Our restricted use of such declarative specification does not hurt performance; in fact, it makes possible introducing time management and optimization designed specifically for the needs of a video-driven virtual environment.

Finally, we note some areas of non-WIMP interfaces that our language specifically does not address, and how they would be connected to this work:

- *3-D Modeling:* We treat this as a separate issue, to be done offline from our system and imported into it (specifically, we can import into Performer from DXF, Open Inventor which is similar to VRML, and a variety of other formats). Figure 5 shows an example of this process using an object that we modeled in IRIS Inventor 1.0 and then imported. Figure 13 makes clear that our research focuses on interactive behavior, rather than attractive graphical appearance!

- *Animation:* Our UIDL is not an animation language; it is a language for interaction with the user. These two aspects of the interface would be integrated by having the underlying animation system provide parameters or "levers" that the user's inputs can control. Our system then connects user actions to changes in these animation parameters or levers via variables in our graph or actions taken on state transitions. A user interaction might thereby start or stop an animation, change its acceleration, path, or destination. PMIW provides the user interface to the "levers;" the animation software provides the levers.

- *Physical Modeling:* Many VR systems include extensive geometric and physical models and simulations, in contrast to typical WIMP applications. Most systems also require some capabilities that do not mimic the real world and cannot be described simply by their physical properties. There are usually ways to fly or teleport, to issue commands, create and delete objects, search and navigate, or other facilities beyond the physical world analogy. These behaviors are where our UIDL will be most useful, since they cannot be described by relying only on the real-world analogy. We therefore view physical modeling in a similar way to animation. That is, there might be an underlying physics engine or simulation engine with controllable parameters, and we provide the means for the user's inputs to control parameters of the physical simulation. (Our data-

flow language might also be good for the sort of ad-hoc, purpose-built physical models as are found in many VR systems today.) In the future, we envisage a more general, separate system for handling physical simulation in much the same way that rendering 3-D geometry is now usually handled by a separate graphics system. Our UIDL will specify the aspects of the interface that do not follow directly from physical laws (fly, teleport, delete, pop-up a menu) and turn them into inputs to the simulation system; the normal action of the simulation will handle those aspects that follow directly from physical laws.

- *Sub-Assemblies:* Our current system will obviously explode with complexity as we build bigger worlds; Figure 11 is already too cluttered. To address this, PMIW is designed to allow higher-level sub-assemblies to be defined in a straightforward way. Any set of links and variables can be packaged into a single component, with some of its inputs and outputs exposed and others encapsulated internally, much as an electronic sub-assembly encapsulates a set of components and provides a reduced set of input and output pins. This is currently available in our system, but not yet accessible from the editor. Note that such encapsulation has no bearing on run-time operation. At run time, the components are exploded back into their individual links and variables and executed by the constraint system; the higher level encapsulations have no performance impact. The Polhemus-to-cursor function that appears in some of the figures is an example of a simple candidate for such an assembly. In fact, we have implemented it as a plug-compatible module which can be replaced by one that lets a moded mouse control the 3-D position of the cursor instead; both export the same **cursorpos** variable and thus can be used interchangeably

Our goal is to provide a model and abstraction that captures the formal structure of non-WIMP dialogues in the way that various previous techniques have captured command-based, textual, and event-based dialogues. We seek to bring higher level, cleaner user interface description language constructs to the problem of building of non-WIMP interfaces. We have demonstrated how such a language can be used and implemented and shown that it need not compromise real-time performance.

ACKNOWLEDGMENTS

We thank Alan Bryant, David Copithorne, Ying Huang, Konstantina Kakavouli, Hansung Kim, Quan Lin, George Mills, Eric Reuss, Vildan Tanriverdi, and Daniel Xi, who worked with us; they are students and recent alumni of the Electrical Engineering and Computer Science Department at Tufts. And we thank Linda Sibert and James Templeman of the Naval Research Laboratory for valuable discussions about these issues.

This work was supported by National Science Foundation Grant IRI-9625573, Office of Naval Research Grant N00014-95-1-1099, and Naval Research Laboratory Grant N00014-95-1-G014. We gratefully acknowledge their support.

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. by M. Burnett, A. Goldberg, and T. Lewis, Prentice-Hall, New York, 1995.

8. M. Conway, R. Pausch, R. Gossweiler, and T. Burnette, “Alice: A Rapid Prototyping System for Building Virtual Environments,” *Adjunct Proceedings of ACM CHI'94 Human Factors in Computing Systems Conference*, vol. 2, pp. 295-296, 1994.

9. J. Cremer, J.K. Kearney, and Y. Papelis, “HCSM: a Framework for Behavior and Scenario Control in Virtual Environments,” *ACM Transactions on Modeling and Computer Simulation*, vol. 5, no. 3, pp. 242-267, July 1995.

10. J. Cremer, J.K. Kearney, and Y. Papelis, “Driving Simulation: Challenges for VR Technology,” *IEEE Computer Graphics and Applications*, pp. 16-20, September 1996.

11. 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.

12. S.K. Feiner and C.M. Beshers, “Worlds within Worlds: Metaphors for Exploring n-Dimensional Virtual Worlds,” *Proc. ACM UIST'90 Symposium on User Interface Software and Technology*, pp. 76-83, Addison-Wesley/ACM Press, Snowbird, Utah, 1990.

13. M.A. Flecchia and R.D. Bergeron, “Specifying Complex Dialogs in ALGAE,” *Proc. ACM CHI+GI'87 Human Factors in Computing Systems Conference*, pp. 229-234, 1987.

14. J. Foley, W.C. Kim, S. Kovacevic, and K. Murray, “Defining Interfaces at a High Level of Abstraction,” *IEEE Software*, vol. 6, no. 1, pp. 25-32, January 1989.

15. B. Freeman-Benson and A. Borning, “The Design and Implementation of Kaleidoscope'90, a Constraint Imperative Programming Language,” *Proc. IEEE Computer Society International Conference on Computer Languages*, pp. 174-180, April 1992.

16. 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.

17. R. Gossweiler, C. Long, S. Koga, and R. Pausch, “DIVER: A Distributed Virtual Environment Research Platform,” *IEEE Symposium on Research Frontiers in Virtual Reality*, San Jose, Calif., October 25-26, 1993.

18. M. Green, “The University of Alberta User Interface Management System,” *Computer Graphics*, vol. 19, no. 3, pp. 205-213, 1985.

19. M. Green and R.J.K. Jacob, “Software Architectures and Metaphors for Non-WIMP User Interfaces,” *Computer Graphics*, vol. 25, no. 3, pp. 229-235, July 1991.

20. M. Green and S. Halliday, “Geometric Modeling and Animation System for Virtual Reality,” *Comm. ACM*, vol. 39, no. 5, pp. 46-53, May 1996.

21. R.D. Hill, “Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction-The Sassafras UIMS,” *ACM Transactions on Graphics*, vol. 5, no. 3, pp. 179-210, 1986.

22. R.D. Hill, “The Rendezvous Constraint Maintenance System,” *Proc. ACM UIST'93 Symposium on User Interface Software and Technology*, pp. 225-234, Atlanta, Ga., 1993.

23. 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*, vol. 1, no. 2, pp. 81-125, June 1994.

24. K. Hinckley, R. Pausch, J.C. Goble, and N.F. Kassell, “A Survey of Design Issues in Spatial Input,” *Proc. ACM UIST'94 Symposium on User Interface Software and Technology*, pp. 213-222, Marina del Rey, Calif., 1994.

25. 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.eecs.tufts.edu/~jacob/papers/chi95.txt [ASCII].

26. S. Hudson and I. Smith, “Practical System for Compiling One-Way Constraint into C++ Objects,” Technical Report, Georgia Tech Graphics, Visualization, and Usability Center, 1994.

27. S.E. Hudson, “Graphical Specification of Flexible User Interface Displays,” *Proc. ACM UIST'89 Symposium on User Interface Software and Technology*, pp. 105-114, Addison-Wesley/ACM Press, Williamsburg, Va., 1989.

28. S.E. Hudson, “Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update,” *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 3, pp. 315-341, July 1991.

29. R.J.K. Jacob, “Executable Specifications for a Human-Computer Interface,” *Proc. ACM CHI'83 Human Factors in Computing Systems Conference*, pp. 28-34, 1983.

30. R.J.K. Jacob, “Using Formal Specifications in the Design of a Human-Computer Interface,” *Communications of the ACM*, vol. 26, no. 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.
http://www.eecs.tufts.edu/~jacob/papers/cacm.txt [ASCII];
http://www.eecs.tufts.edu/~jacob/papers/cacm.ps [Postscript].

31. R.J.K. Jacob, “An Executable Specification Technique for Describing Human-Computer Interaction,” in *Advances in Human-Computer Interaction, Vol. 1*, ed. by H.R. Hartson, pp. 211-242, Ablex Publishing Co., Norwood, N.J., 1985.

32. R.J.K. Jacob, “A Specification Language for Direct Manipulation User Interfaces,” *ACM Transactions on Graphics*, vol. 5, no. 4, pp. 283-317, 1986.

http://www.eecs.tufts.edu/~jacob/papers/tog.txt [ASCII];
http://www.eecs.tufts.edu/~jacob/papers/tog.ps [Postscript].

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

34. R.J.K. Jacob, J.J. Leggett, B.A. Myers, and R. Pausch, "Interaction Styles and Input/Output Devices," *Behaviour and Information Technology*, vol. 12, no. 2, pp. 69-79, 1993.
http://www.eecs.tufts.edu/~jacob/papers/bit.txt [ASCII];
http://www.eecs.tufts.edu/~jacob/papers/bit.ps [Postscript].

35. R.J.K. Jacob, "A Visual Language for Non-WIMP User Interfaces," *Proc. IEEE Symposium on Visual Languages*, pp. 231-238, IEEE Computer Society Press, 1996.
http://www.eecs.tufts.edu/~jacob/papers/vl.txt [ASCII];
http://www.eecs.tufts.edu/~jacob/papers/vl.ps [Postscript].

36. D.J. Kasik, "A User Interface Management System," *Computer Graphics*, vol. 16, no. 3, pp. 99-106, 1982.

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

38. 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.

39. J. Liang and M. Green, “JDCAD: A Highly Interactive 3D Modeling System,” *Computers and Graphics*, vol. 18, no. 4, pp. 499-506, Pergamon Press, London, 1994.

40. J.D. Mackinlay, S.K. Card, and G.G. Robertson, “A Semantic Analysis of the Design Space of Input Devices,” *Human-Computer Interaction*, vol. 5, pp. 145-190, 1990.

41. S.A. Morrison and R.J.K. Jacob, “A Specification Paradigm for Design and Implementation of Non-WIMP User Interfaces,” *ACM CHI'98 Human Factors in Computing Systems Conference*, pp. 357-358, Addison-Wesley/ACM Press, Poster paper, 1998.

42. S.A. Morrison, “A Specification Paradigm for Design and Implementation of non-WIMP Human-Computer Interactions,” Doctoral dissertation, Tufts University, 1998.

43. B.A. Myers, *Creating User Interfaces by Demonstration*, Academic Press, Boston, 1988.

44. B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Vander Zanden, D.S. Kosbie, E. Pervin, A. Mickish, and P. Marchal, “Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces,” *IEEE Computer*, vol. 23, no. 11, pp. 71-85, November 1990.

45. B.A. Myers, “User Interface Software Tools,” *ACM Transactions on Computer-Human Interaction*, vol. 2, no. 1, pp. 64-103, March 1995.

46. W.M. Newman, “A System for Interactive Graphical Programming,” *Proc. Spring Joint Computer Conference*, pp. 47-54, AFIPS, 1968.

47. J. Nielsen, “Noncommand User Interfaces,” *Comm. ACM*, vol. 36, no. 4, pp. 83-99, April 1993.

48. D.R. Olsen, *User Interface Management Systems: Models and Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1992.

49. P. Reisner, “Formal Grammar and Human Factors Design of an Interactive Graphics System,” *IEEE Transactions on Software Engineering*, vol. SE-7, no. 2, pp. 229-240, 1981.

50. J.R. Rhyne, “Extensions to C for Interface Programming,” *Proc. ACM SIGGRAPH Symposium on User Interface Software*, pp. 30-45, 1988.

51. J. Rohlf and J. Helman, “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics,” *Proc. ACM SIGGRAPH'94 Conference*, pp. 381-394, Addison-Wesley/ACM Press, 1994.

52. C. Shaw, M. Green, J. Liang, and Y. Sun, “Decoupled Simulation in Virtual Reality with the MR Toolkit,” *ACM Transactions on Information Systems*, vol. 11, no. 3, pp. 287-317, 1993.

53. B. Shneiderman, “Multi-party Grammars and Related Features for Defining Interactive Systems,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-12, no. 2, pp. 148-154, 1982.

54. J.L. Sibert, W.D. Hurley, and T.W. Bleser, “An Object-Oriented User Interface Management System,” *Computer Graphics*, vol. 20, no. 4, pp. 259-268, 1986.

55. 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.

56. B.A. Sufrin and J. He, “Specification, Analysis and Refinement of Interactive Processes,” in *Formal Methods in Human-Computer Interaction*, ed. by M. Harrison and H. Thimbleby, pp. 153-200, Cambridge University Press, Cambridge, UK, 1990.

57. P. Szekely, “Modular Implementation of Presentations,” *Proc. ACM CHI+GI'87 Human Factors in Computing Systems Conference*, pp. 235-240, 1987.

58. S.L. Tanimoto, “VIVA: A Visual Language for Image Processing,” *Journal of Visual Languages and Computing*, vol. 1, no. 2, pp. 127-139, June 1990.

59. B. Vander Zanden, B.A. Myers, D.A. Giuse, and P. Szekely, “Integrating Pointer Variables into One-Way Constraint Models,” *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 2, pp. 161-213, June 1994.

60. C. Ware and R. Balakrishnan, “Reaching for Objects in VR Displays: Lag and Frame Rate,” *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 4, pp. 331-356, December 1994.

61. T. Yunten and H.R. Hartson, “A Supervisory Methodology and Notation (SUPERMAN) for Human-Computer System Development,” in *Advances in Human-Computer Interaction, Vol. 1*, ed. by H.R. Hartson, pp. 243-281, Ablex Publishing Co., Norwood, N.J., 1985.

62. P. Zave and M. Jackson, “Conjunction as Composition,” *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 4, pp. 379-411, October 1993.