Three-Dimensional Widgets

D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon,
Daniel C. Robbins, Robert C. Zeleznik,
Andries van Dam

Computer Science Department
Brown University
Providence, RI

ABSTRACT

The 3D components of today's user interfaces are still underdeveloped. Direct interaction with 3D objects has been limited thus far to gestural picking, manipulation with linear transformations, and simple camera motion. Further, there are no toolkits for building 3D user interfaces. We present a system which allows experimentation with 3D *widgets*, encapsulated 3D geometry and behavior. Our widgets are first-class objects in the same 3D environment used to develop the application. This integration of widgets and application objects provides a higher bandwidth between interface and application than exists in more traditional UI toolkit-based interfaces. We hope to allow user-interface designers to build highly interactive 3D environments more easily than is possible with today's tools.

Keywords

User Interface Design, Widgets, 3D Interaction, Virtual Reality

1 Introduction

Modern user-interface software is built using *widgets*, objects with geometry and behavior used to control the application and its objects. However, most of today's user interfaces for 3D applications take little advantage of the third dimension's added power, predominantly using 2D widgets. Commercial modeling and visualization systems typically present one or more 3D views surrounded by a large, hierarchical menu system, often with supporting dialog boxes and sliders. The menu system is sometimes replaced or augmented by another 2D interface widget such as a network or hierarchy editor. Direct interaction with the 3D world is limited primarily to interactive viewing, selection, translation, and rotation. 3D widgets used in these interactions include a 3D cursor, gestural translation, a virtual sphere, and direct manipulation of 3D spline points on paths or patches. While today's 3D applications clearly allow users to be productive with the current interface technology, we believe that they could be improved significantly by making greater use of 3D in the interface itself.

In virtual-reality systems, 3D interaction is especially crucial. However, the significant difficulties of 3D input and display have led research in virtual worlds to concentrate far more on the development of new devices and device-handling techniques than on higher-level techniques for 3D interaction [19]. Such interaction goes no further than a straightforward interpretation of device data, such as using a Polhemus for a head tracker or a DataGlove for simple gestural recognition of commands such as select, translate and rotate. Some virtual-reality systems make use of menus floating in 3-space with 3D icons instead of 2D pixmap icons [3]. Besides the additional options for its position, however, such a menu provides no more expressive power than its 2D equivalent.

There are many reasons for the underutilization of 3D. First, almost all interaction techniques must be created from scratch, since essentially no toolkits of 3D interaction techniques exist. Second, such toolkits are difficult to develop until metaphors for 3D interfaces grow beyond their current infancy. Finally, we believe such a toolkit is intrinsically more difficult to create than its 2D counterpart because of the inherent complexity of 3D interaction.

Widget toolkits are well known for 2D applications (e.g., the Macintosh Programmer's Toolbox, OSF/Motif, XView) [17]. However, 3D graphics libraries such as PHIGS+ and SGI's GL provide very little support for interaction beyond simple device handling. The industry standard PHIGS+ provides only six widgets (pick, locator, stroke, choice, valuator, and string). Further, the application programmer cannot change their look or feel, and all except 3D pick correlation are low-level, providing little functionality beyond that provided by a physical device. Thus, application developers are left to implement basic interactive techniques such as virtual sphere rotation themselves.

Most paradigms and metaphors for 3D interfaces are less developed than those for 2D interfaces. Some 3D metaphors are the natural analogs of those familiar in 2D, such as 3D menus and rooms [14] [4]. However, research in 3D interfaces must develop new metaphors and interaction techniques to take advantage of the greater possibilities of 3D. The cone tree and perspective wall, designed at Xerox PARC [22] [13], demonstrate the potential of 3D representation and interactive animation.

User interfaces are inherently difficult to program [17]. 3D interfaces complicate interface design and implementation, since the interface must take into account such issues as a richer collection of primitives, attributes, and rendering styles, multiple coordinate systems, viewing projections, visibility determination, and lighting and shading. Further, 3D environments allow many more degrees of freedom than those easily specified with common interface hardware like mice. The interface can easily obscure itself, and 3D interaction tasks can require great agility and manual dexterity. Indeed, physical human factors are a central part of 3D interface design, whereas 2D interface designers can assume that hardware designers have handled the ergonomics of device interaction.

This paper reports some first steps towards the goal of creating a richly interactive 3D application development environment. After a more detailed discussion of the problems inherent in designing and implementing 3D widgets, we present a framework under development for their implementation, design, and use. By working with an object-oriented notion of a widget, we hope to provide a toolkit of modifiable and reusable 3D interaction techniques.

2 Extending Widgets

There are several points to consider when designing an environment for developing 3D widgets. Most fundamentally, what *is* a widget? How do existing notions of widgets derived from 2D environments extend to 3D environments? Secondly, how should a 3D application communicate with its 3D interface? Finally, what kinds of primitives are needed to build 3D widgets? 2D environments, like the X Window System, provide raster drawing primitives and event-based callback mechanisms. What sorts of primitives should a corresponding 3D environment provide?

2.1 Defining "widget"

We define a widget as an encapsulation of geometry and behavior used to control or display information about application objects. Although this definition is somewhat vague and general, it has the advantage of covering all the areas of the interface literature we have explored, from general constructs such as Garnet's Interaction Objects [16] and the Interactive Objects of Xerox's 3D Rooms [21] to very specific kinds of widgets such as those found in the X Toolkit or the Macintosh Toolkit.

The extent to which a 2D widget should be classified as consisting of behavior or of geometry varies widely. Some useful widgets are primarily geometric, such as the dividing lines and frames that serve to organize and partition an interface. Others, such as a gestural rotation widget in an object-oriented drawing program, have no inherent geometry. 3D widgets encompass a similar range of geometry and behavior. This makes our definition of the term "widget" useful for understanding interface problems that are not dimension-specific.

2.2 Comparing common 2D widgets and 3D widgets

Despite their often complex appearance, most 2D widgets have very simple behavior. They commonly have few degrees of freedom (usually only one) and support only a small range of values within a degree of freedom. Thus, while toggle buttons have bitmap icons to represent different states, they represent only a single bit of information, and similarly, sliders represent a single number within a range, usually only a small integer range.

3D space inherently has more degrees of freedom than 2D space: a rigid flying body has six degrees of freedom in 3D versus three in 2D. 3D graphics libraries are, in general, more capable of handling general transformations than their 2D counterparts. As noted, common 2D widgets rarely take advantage of all the degrees of freedom available to them. The use of multiple degrees of freedom to enhance interaction is thus largely unexplored potential, even in 2D [23], and 3D, with its greater degrees of freedom, has correspondingly greater potential. This potential must of course be handled with restraint: while we would like to be able to use several degrees of freedom simultaneously, using too many may make the widget too difficult to use. Rather, interface designers should be able to specify any subset.

The user interacts with most widgets, whether 2D or 3D, through manipulation involving motion and simple gestures that are interpreted directly, to produce, for example, a sliding button or a popup window. However, the user can gain more expressive power through interaction techniques that interpret and process movements and make possible more sophisticated interaction. For example, a calligraphic drawing program can attach a pen to a cursor by means of a simulated spring [9], a simple motion-control technique that makes possible a whole new range of drawings not easily created with a rigid pen-cursor linkage.

Both 2D and 3D widgets can benefit from more sophisticated reaction to user input. Interaction can potentially achieve substantial gains by using such techniques as dynamic constraints, inverse kinematics, and physical simulation as components of direct manipulation interfaces. These techniques currently appear only in systems designed explicitly to present or use them, such as demos or prototypes, but in the future, these techniques should be as accessible as any other component in the widget designer's repertoire [8].

2.3 Integrating the application and the user interface

User interfaces were originally designed by application programmers using the same tools they used to build applications. This produced interfaces that were tightly integrated with the application. Recently, however, interface design is more often done by specialists using UI development tools [17]. While this separation produces more consistent interfaces and more modular programs, it can also produce interfaces that are not as helpful as they could be if they were more specialized to the application — the interface designer is not only aided but also limited by the toolkit and its metaphors. In particular, as has been noted by those critiquing WIMP interfaces [8], today's toolkits are not oriented towards highly interactive applications.

Such highly interactive applications require a high bandwidth between the application and the user interface, particularly for semantic feedback [8]. Prior UI research indicates that this may be best accomplished if the application and the interface are part of the same development environment, with the same tools being used to build both [18]. An integrated environment has additional software engineering benefits. First, only a single paradigm must be learned, rather than one for the interface and another for the application. Also, separate paradigms can be hard to integrate at several levels: the conceptual level, the code implementation level, and the compile-debug level. Advocating integration is *not* a call to abolish modularity in application and interface design. Rather, it is a suggestion that the principles of modularity can be pushed too far. The reasons for separating the application from the user interface are valid, but the benefits of a single development environment may outweigh the benefits of using two, especially for 3D applications.

Consider the benefits of higher bandwidth between the application and the interface. A menu selection is a relatively small amount of input that specifies only an operation, operand, or attribute, leaving other parameters to be specified elsewhere (perhaps in another menu or a dialog box). Gestural interfaces, on the other hand, allow the user to specify operation, operand, and parameters in a single action [23], providing a faster interface and commands that do not depend on previous or further actions.

In addition to providing better input, a tighter integration between application and interface lets the application provide semantic feedback *while* the user is interacting. Structured program editors have provided this kind of functionality for many years through syntax checkers that check for or prevent syntactic errors as the user types. Similarly, some 2D graphical circuit design tools prevent the user from making physically impossible or illogical connections.

Existing UI toolkits do allow callbacks to alter a widget based on application feedback, but the mechanisms to do so are often clumsy and hard to use. Our interfaces are constructed in an environment called UGA [25] in which widgets can actively depend on the state of other widgets, in the same way that any other objects (e.g., the application's objects) in our system can depend on each other. Our widgets are not external to the application model. They are first class objects, indistinguishable from application objects. This provides the UI designer with all of our system's power for specifying behavior and geometry, and gives as high a bandwidth between application and interface as between application objects themselves, creating the possibility of interfaces that are tightly coupled with the application, both for input and for output.

We have advocated both 3D widgets and widgets that are tightly integrated with an application. The latter idea is the more powerful of the two, since it can apply to all areas of interface design. In the remainder of the paper, we consider tools applicable to integrated widgets and then examine some case studies of integrated 3D widgets.

3 Tools for Designing and Implementing Integrated Widgets

3D interfaces are presently too underdeveloped for us to specify a comprehensive library of tools for building useful interfaces. We have therefore devised an environment that provides a great degree of flexibility to design new 3D widgets. It is often pointed out that flexibility in a user-interface design environment is a double-edged sword, allowing novel and useful interfaces as well as novel and useless interfaces. Because of the undeveloped state of current 3D interfaces, however, we prefer to allow the possibility of some poorly conceived designs rather than rule out unexplored possibilities.

3.1 Dependencies and controllers

UGA supports the geometric components of widgets through its rich modeling environment. The system supports the behavioral aspects of widgets through one-way constraints called *dependencies* [25]. An object can be explicitly related to another object by using a dependency. Since widgets are first-class objects in UGA, they can use this dependency mechanism as easily as application objects can. For example, a cube can become a simple slider by constraining it to move only along its x axis, and a torus's inner radius can then depend on the x position of the cube.

To provide multi-way constraints and cyclical constraint networks [18], we use *controllers* [25], objects whose primary purpose is to control other objects. Thus, our dynamic constraint solver is encapsulated as a controller. Additionally, we encapsulate physical devices as controllers that filter and pass values to objects. Finally, we can use controllers to encapsulate simulation methods, such as inverse kinematics or collision detection. By employing controllers, widgets can make use of general constraints, hardware devices, and simulation techniques.

3.2 A dialog model for sequencing

Some researchers choose to separate UI design into two broad categories: data-oriented UI design, usually supported through constraints, and dialog-oriented UI design [11]. We find both models useful. In addition to the data-oriented mechanisms of dependencies and controllers, we provide a dialog model that uses augmented transition networks (ATNs). We use ATNs because the sequencing of an interface is explicitly declared and is more easily visualized in a hierarchical ATN than in context-free grammars or event systems [7].

A simple transition network is a finite-state automaton (FSA). A complex interface can be described as an FSA but the complexity produces a combinatorial explosion of FSA states. Augmented transition networks handle some of the limitations of simple FSAs (allowing such behaviors as definite loops without specifying intermediate states) by adding variables and conditional transition along arcs based on the values in the variables. Recursive transition networks are used to provide hierarchy for ATNs, by allowing control in one ATN be suspended until a recursively invoked ATN reaches its final state.

Normally, an ATN, even a recursive one, has only one current state. Therefore, some events that can happen at any time, such as an "abort" or "help" request, are especially cumbersome to specify, requiring an additional arc from every state in the ATN. By contrast, event systems have greater expressiveness than ATNs [7], since they can easily handle an "abort" or "help" event by simply adding a new event handler to process this event. This would seem to make event systems a better choice. However, notions of current state, history, or context are more difficult to express in event systems. Consider a "help" event that should provide context-sensitive information. An event model must provide a different event for each context. On the other hand, an ATN can handle a uniform "help" event, with arcs corresponding to context-dependent actions looping back to each state or leading to one or more help states. We would like a dialog model that combines the best features of both ATNs and event handlers.

Thus, we modify the ATN model to allow possibly disconnected components of the state graph and more than one active state [12]. We can now represent a set of event handlers as a group of disconnected states in an ATN, one state per event handler, each with a single arc back to itself. The arc's input tokens represent the corresponding event handler's events, and the arc's action represents the handler routine. However, we can add explicit sequencing to this ATN. For example, in our model, it is easy to specify the sequence of events found in snap-dragging, described in Section 4.3, but relatively cumbersome to specify in an event model, because of the need to represent history.

Our dialog model also allows a clean separation of subparts of the interface (i.e., individual widgets or groups of widgets). The dialog specification of each subpart can be represented as a subgraph of the ATN that describes the specification of the entire interface. These subparts can run in parallel, corresponding to a situation in which several widgets are logically operating at the same time. This parallelism is very useful: we can, for example, use the mouse to control both a 3D cursor and a higher-level widget, such as the rack described in Section 4.5.

The components of this dialog model, such as the individual states in the ATN, are first-class objects in our system. Since the dialog model is embedded in the same environment as the application itself, dependencies can be used to establish the connections between the ATN and the application that allow each to modify the other.

3.3 Applying object construction techniques

The UGA system supports a rich set of modeling primitives and operations, including constructive solid geometry (CSG), volumetric sculpting, spline patch objects and deformations. Both geometric and non-geometric modeling techniques, such as hierarchical grouping, can be applied to widget creation. Geometric techniques are used to specify a widget's geometry. Correspondingly, since ATN states are first-class objects, they can be organized using non-geometric object grouping techniques. Thus, both a widget's geometry and behavior are specified in the same unified framework, the framework of the application objects it controls.

The underlying construction technique we use is *delegation*, where one object (the child) is created from a pre-existing object (the parent) [24] [10]. If the parent object is changed, the child changes as well. Since both the parent and its children are objects in the system, and any object can be a controller modifying other objects, one of the children can modify the parent object, and therefore modify itself and all of its siblings. Delegation provides the ability to change large portions of the interface at once. Furthermore, since delegation relationships are maintained at run time, we can modify the interface without recompiling. This allows rapid prototyping of interface designs.

4 Examples of 3D Widgets in Our System

Our user interface group has developed several simple 3D widgets in our framework. Some of these, such as the virtual sphere and the cone tree, duplicate other researchers' widgets; others are experiments with new paradigms for a 3D user interface. We present these widgets below, explaining the design process we used in creating them, and stress the progress made possible by rapid prototyping.

4.1 A virtual sphere

A virtual sphere rotation widget can be handled by a simple two-state ATN (Figure 1). The ATN processes mouse motion, passing the mouse positions to a function that maps the 2D mouse coordinates into another object's space, in this case producing a point on the surface of a sphere. The deltas between a series of these projections produce rotations. We can easily change the kind of object that mouse coordinates are mapped to, so as to produce a "virtual cube" or "virtual donut." This sort of modification of the interface can be done at run time.

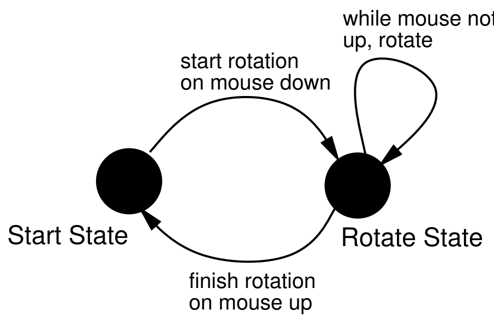

Figure 1: A two-state ATN for virtual sphere rotation

4.2 Handles

Object handles [6] are a 3D widget that contains more visual geometry than the virtual sphere widget. We can build handles with an arbitrarily complex appearance. Once they are built, we are free to establish dependencies on them or use them as a controller. Color Plate I shows various handles being used to translate, rotate and scale an object.

The same kind of constrained motion can be produced by holding down various modifier keys or different combinations of buttons [20]. However, a user presented with such an interface has no easy way to determine what the possible actions are. Handles allow constrained motion through intuitive direct manipulation: when a particular handle is selected, motion is constrained along or around the axis it describes. For example, clicking on an object-space translation handle located along an object's x axis limits translation to the x axis.

The visual feedback of a widget can range from the direct movement of the selected object to more complex widgets, such as handles that include numerical output and other quantitative indicators. Because our system provides rich support for geometry, the same set of primitives used for the application can be used to assemble widgets and their visual feedback. The behavior of handles can be produced without the corresponding geometry. An example is the creation of "hot spots" on an object that may or may not have a visual indication. The behavior of a virtual sphere can in turn be augmented with geometry — for instance, a semi-transparent sphere can be placed around the object during rotation to convey the behavior of the widget to the user more effectively. The flexibility of the system allows the widget designer and user to explore a wide range of options.

4.3 Snapping

With a more intricate ATN (Figure 2) we can perform simple snap-dragging [2]. A mouse's coordinates are used to generate a ray from the camera through the projection of the mouse's position onto the viewplane. If this ray intersects an object, the ATN lets the user choose a point on an object to snap to a point on another object. Since this is done with ray intersection, the point to snap includes a complete Frenet frame [15] defined by the surface normal and tangents. When the user releases the mouse button and clicks again, the ATN begins checking to see if the ray specified by the mouse intersects another object. If so, this new object becomes the object to snap to. Again, the user can choose exactly which point to use, including the entire Frenet frame. When the user has chosen both points, the widget produces a transformation to align the two frames, and applies it to the first object.

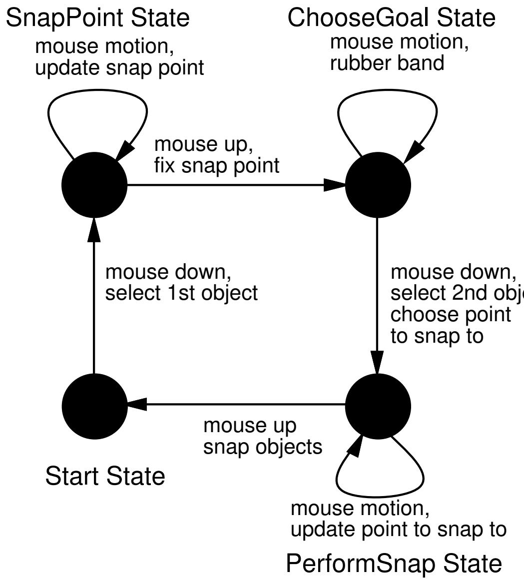

Figure 2: A four-state ATN for interactive snapping

By changing the states in the ATN, the user can experiment with different ways of specifying snap-dragging. Several different ATNs for different snapping techniques can be concurrently developed and experimented with, even at run time. For example, a user could develop a more complex ATN to allow the specification of the distance between the surfaces as well as the relative orientation of the Frenet frames.

4.4 A color picker

Color spaces are inherently multidimensional. To illustrate these spaces we can build a color picker in three dimensions and show how changes in the values affect the output color. Color plate II shows two interactive views of RGB color space and one interactive view of HSV color space. One view of RGB space is built with three sliders, each of which was specified using dependencies. Another view is built using a cubical marker that can translate within the bounds of a unit cube. Here, each axis of the cube's position represents a component of the color value. Thus, all three components can be specified simultaneously using 3D gestural translation. The third view is of HSV space. As in the RGB cube, the position of the spherical marker in the center represents the three components of the HSV color. The constraints on the sphere permit it to move around in the cone that represents valid HSV color values.

All of the spaces are different visualizations of the same data, kept consistent through the use of dependencies. Thus, a user can choose a color in one view and see how that color is represented in the other two. As the user interactively chooses a color, the other two color representations update accordingly. Users familiar with the RGB space can learn about the nature of HSV space by watching the motion of the HSV indicator as they move the RGB indicator.

4.5 The rack

Recall that the ATN states are first-class objects and that our system provides hierarchical grouping of objects. An ATN can pass control to another ATN through dependencies and controller mechanisms. Thus, pre-existing ATN's can be grouped together to form a more complex, hierarchical ATN (see Figure 3) that controls the sequencing of the lower-level ATNs. In other words, we can build more complex widgets out of pre-existing widgets.

To construct a more complex widget, we start with the simple rotation and translation handle widgets discussed in Section 4.2. By rearranging them and changing their connections, we combine them to form a "rack" for specifying high-level deformations such as twists, tapers and bends [1], shown in Color Plate IV.

Different handles specify the parameters to three deformations. The distance between the two upright handles specifies the range over which the deformation applies. The angle of the red handle on the end indicates the amount of bend, and the angle of the pink handle indicates the amount of twist, while the height of the blue handle indicates the amount of taper. By reconfiguring the rack, changing the number of handles and their respective behaviors, the user can control how the deformation is specified. Specialized racks that only bend, taper, or twist can be easily built. A new rack can be designed to apply wave deformations, or to allow both geometric transformations and nonlinear deformations at the same time.

Textual specification of a bend deformation requires four floating-point values and two vectors. The rack specifies all of these visually. The major axis of the rack specifies one vector, and the red handle specifies another vector, determining the angle and direction in which the object should bend. The floating-point values are all specified by how much particular handles are moved.

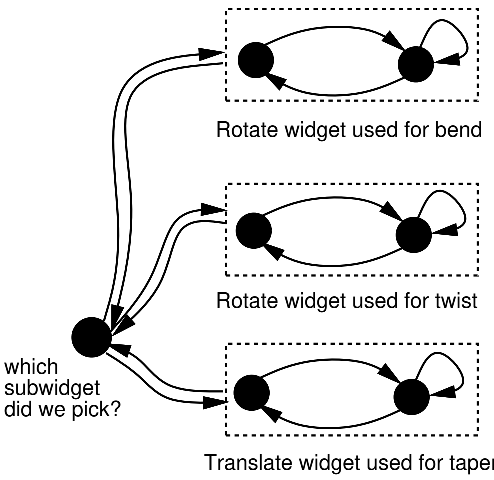

Figure 3: Several ATNs can be combined to form a more complex widget. This widget specifies high-level deformations.

The rack is a widget that provides a more meaningful interface to complex deformations than a conventional widget such as a panel of independent sliders. Such a panel provides no semantic correlation: the user must extrapolate a single deformation from multiple independent slider positions. Thus, the rack serves to abstract out the essential characteristics of a deformation. When handles are used to translate an object in its own object space, the handles themselves give the user feedback on the orientation of that space, which might not be apparent from the object itself. Similarly, an object being deformed with the rack may be so geometrically complex that it has no clear axis around which to twist, bend or taper. The rack provides this axis, along with immediate and understandable feedback about the magnitude and effects of the deformations.

4.6 The cone tree

More complicated metaphors for 3D interfaces can be constructed and experimented with in our system. A large number of rotation widgets can be assembled into a Xerox PARC-style cone tree. Here, we use the cone tree to display the hierarchy of a 3D model (Color Plate III). The cone tree is itself an object in the system and can be freely manipulated as a whole.

The nature of this widget inherently requires motion control to animate the rotation of the subtrees. When we modify the cone tree, we can affect the underlying geometric hierarchy it represents. Moving subtrees of the cone tree to other nodes in the tree affects the hierarchy of the model that the cone tree represents. If we use other tools to modify the hierarchy, the cone tree's structure is also updated.

Since the cone tree is itself a widget, we can combine it with other widgets to make more intricate information browsers, much as simple rotation and translation widgets were composed above to make a deformation editor. We plan to explore using cone trees to represent portions of a hypermedia graph that are primarily hierarchical but have some cross-links, e.g., a multimedia technical paper with its various sections, subsections, references, and see-also's.

5 Conclusions

5.1 Accomplishments

We have presented a concept of 3D widgets as first-class objects encapsulating behavior and geometry that can be treated as any other objects in a 3D world. Their behaviors may be defined using complex control methods and user input techniques. We have provided a first implementation of these widgets within the UGA system. Widgets can be rapidly prototyped, modified, and combined into more complicated systems of widgets. Close integration with the application allows rich forms of interaction and feedback in our 3D applications.

5.2 Future work

Constructing 3D widgets is reasonably fast with our system. However, widget designers at present must be experts in the use of UGA. We hope to make specifying 3D widgets even more natural and intuitive than it is now, so that a far less technically expert designer can implement 3D widgets. Part of the complexity stems from limitations of dependencies. We might address these limitations with a more generic constraint model at the basic system level, making it easier to specify some of the complex relationships of 3D widgets. In addition, our system does not run as fast as we would like, even on today's high-end platforms. A large portion of time is spent evaluating dependencies. Unfortunately, the addition of a more generic constraint model is not likely to help performance. Thus, dependencies merit a close look, at both the conceptual and the implementation level.

We would like to continue developing individual widgets and exploring the potential of various techniques from the world of 3D graphics in interface design. We want to investigate the use of more sophisticated motion control, modeling and rendering techniques for 3D widgets. We can foresee widgets that will use dynamic constraints, physical simulation, volumetric techniques, particle systems, and even radiosity. Our application framework already includes many of these techniques, so it is simply a matter of their imaginative application in our system to make use of such techniques in 3D interfaces.

In addition, we are in the process of constructing full 3D applications and interfaces with the system presented. We believe the unusual nature of our widgets will provide some interesting avenues of exploration. Since the widgets are as much a part of the application as the application itself, it is straightforward to manipulate widgets with widgets. In other words, a user interface can be built by starting with simple widgets and using them to bootstrap more complex ones.

Finally, we hope to develop a high-level UIDS (user interface design system) [5] for our system. As previously noted, our system currently has no tools for making high-level specifications of an interface. Most commercial UIMSs, having been built on top of a widget toolkit, focus on appearance and geometry of widgets. Some research-level UIDSs handle behavior and sequencing. A UIDS suitable for our system would clearly have to be able to handle full application behavior and would perhaps be an Application Design System, a full-fledged programming environment for 3D interactive applications.

5.3 Acknowledgments

This work was supported in part by NSF, DARPA, IBM, Sun Microsystems, NCR, Hewlett Packard and Digital Equipment Corporation. Software contributions by Bitstream Inc., Pixar and Visual Edge Software Ltd. are gratefully acknowledged. We also thank Frank Graf for his help in implementing and extending the rack widget and Nate Huang for technical support.

References

[1] Alan H. Barr. Global and local deformations of solid primitives. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, volume 18(3), pages 21–30, July 1984.

[2] Eric A. Bier. Snap-dragging in three dimensions. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, volume 24(4), pages 193–204, March 1990.

[3] Jeff Butterworth, Andrew Davidson, Stephen Hench, and T. Marc Olano. 3DM: A three dimensional modeler using a head-mounted display. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, 1992.

[4] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Human Factors in Computing Systems, Proceedings of the ACM SIGCHI*, pages 181–188. Addison Wesley, 1991.

[5] James Foley, Won Chui Kim, Srdjan Kovačević, and Kevin Murray. Designing interfaces at a high level of abstraction. *IEEE Software*, pages 25–32, January 1989.

[6] Tinsley Galyean, Melissa Gold, William Hsu, Henry Kaufman, and Mark Stern. Manipulation of virtual three-dimensional objects using two-dimensional input devices. Class project, Brown University, December 1989.

[7] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–375, 1986.

[8] Mark Green and Robert Jacob. SIGGRAPH '90 workshop report: Software architectures and metaphors for non-WIMP user interfaces. *Computer Graphics*, 25(3):229–235, July 1991.

[9] Paul Haeberli. `dynadraw`. posted to `comp.graphics`, 1990. GL program.

[10] Brent Halperin and Van Nguyen. A model for object-based inheritance. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[11] Scott H. Hudson. Graphical specification of flexible user interface displays. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 105–114, 1989.

[12] Robert J. K. Jacob. A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

[13] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The perspective wall: Detail and context smoothly integrated. In *Human Factors in Computing Systems, Proceedings of the ACM SIGCHI*. ACM SIGCHI, 1991.

[14] Microelectronics and Computer Technology Corporation. An introduction to the visual metaphors team's software releases. Video tape, 1986. TR# HI-344-86.

[15] Richard S. Millman and George D. Parker. *Elements of Differential Geometry*. Prentice-Hall, 1977.

[16] Brad A. Myers. Encapsulating interactive behaviors. In *Proceedings of CHI '89* (Austin, TX, April 30–May 4, 1989), pages 319–324. ACM, New York, May 1989.

[17] Brad A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, pages 15–23, January 1989.

[18] Brad A. Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[19] Randy Pausch. personal communication, 1991.

[20] Cary B. Phillips, Jianmin Zhao, and Norman I. Badler. Interactive real-time articulated figure manipulation using multiple kinematic constraints. In *Special Issue on the 1990 Symposium on Interactive 3D Graphics, Computer Graphics*, pages 245–250. ACM SIGGRAPH, ACM Press, March 1990.

[21] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. The cognitive coprocessor architecture for interactive user interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 10–18, 1989.

[22] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Human Factors in Computing Systems, Proceedings of the ACM SIGCHI*. ACM SIGCHI, 1991.

[23] Dean Rubine. Specifying gestures by example. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, pages 329–337. ACM SIGGRAPH, Addison-Wesley, July 1991.

[24] Peter Wegner. The object-oriented classification paradigm. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[25] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, pages 105–112. ACM SIGGRAPH, Addison-Wesley, July 1991.