An Interactive 3D Toolkit for Constructing 3D Widgets

Robert C. Zeleznik, Kenneth P. Herndon, Daniel C. Robbins,
Nate Huang, Tom Meyer, Noah Parker and John F. Hughes

Brown University
Department of Computer Science
Providence, RI 02912
(401) 863-7693; {bcz,kph,dcr,nth,twm,nfp,jfh}@cs.brown.edu

CR Categories

I.3.6 [Computer Graphics]: Methodology and Techniques; Interaction Techniques D.1.7 [Programming Languages]: Programming Techniques; Visual Programming D.2.2 [Software Engineering]: Tools and Techniques; User Interfaces

1 Introduction

Today's user interfaces for most 3D graphics applications still depend heavily on 2D GUIs and keyboard input. There have been several recent attempts both to extend these user interfaces into 3D and to describe intermediary 3D widgets[1] that control application objects [3; 4; 5; 7; 13; 15]. Even though this style of interaction is a straightforward extension of interaction through intermediary 2D widgets such as dials or sliders, we know of no efforts to develop interactive 3D toolkits akin to UIMX or Garnet [11].

The Brown Graphics Group has had considerable experience using its Unified Graphics Architecture (UGA) system [16] to script 3D widgets such as deformation racks [14], interactive shadows [9], parameterized models, and other constrained 3D geometries. Using this experience, we have developed an interactive toolkit to facilitate the visual programming of the geometry and behavior of such interactive models. The toolkit provides both a core set of 3D widget primitives for constructing interactive behaviors based on constrained affine transformations, and an interactive 3D interface for combining these primitives into more complex widgets.

This video paper describes the fundamental concepts of the toolkit and its core set of primitives. In particular, we describe (i) the conceptual structure of the primitives, (ii) the criteria used to select a particular primitive widget set that would be expressive enough to let us construct a wide range of interactive 3D objects, and (iii) the constraint relationships among the primitives.

2 Overview of our 3D Toolkit

The traditional approach to designing user interface toolkits is to create a library of software objects and customize them through instantiation and specialization within standard programming languages [12; 15]. Although this approach is extremely powerful, exploring the full potential requires that programmers be able to visualize complex relationships among software objects (*e.g.*, constraint networks, data- and control-flow graphs). A second paradigm, based on graphically manipulating function networks [1; 8; 10], is more accessible to the non-programmer, but still suffers because inherently geometric relationships must be specified by wiring 2D boxes together.

Our toolkit uses direct manipulation of 3D widgets to model the construction of widgets and application objects whose geometric components are affinely constrained. This paradigm is more natural than scripting or dataflow programming because the process of constructing such objects is inherently geometric, and also enables non-programmers and designers to construct these objects visually. The scope of these constructions includes, for example, all of the widgets we have built in the last few years and standard joints such as slider, pin, and ball joints.

We introduce the notion of primitive 3D widgets that can be combined with other primitive 3D widgets, using a process called *linking*, to establish one or more constraint relations between them. In some cases, the resulting composite objects are still considered widgets; in others, they are thought of as the behavioral scaffolding to which the geometry of application objects can be attached. The fact that the interface and application objects exist in the same underlying system, UGA, allows us to blur the distinction between them. We feel that such blurring is natural for 3D applications in general, and especially for virtual reality applications.

Linking is related to snapping [3], but differs in requiring explicit interactive selection of source and destination objects, followed by explicit user confirmation. This protocol reduces clutter by eliminating alignment objects. In the interest of simplifying the user interface, all linking operations are unparameterized, although in future work, parameterized linking for more advanced users and more complicated widgets will be explored.

3 Conceptual Structure of Widget Primitives

A primitive widget combines the geometries and behaviors of its ports and other more simple primitives. A *port* is an encapsulation of one or more constraint values and a geometric representation. It can be loosely considered a data type with the additional requirement that its visual appearance suggest the meaning of the data. Ports are related to one another within a single widget via a network of bi-directional constraints. In addition, specific interaction techniques are associated with each port. Each interaction technique tells how to modify a port while maintaining constraints on other ports. For example, if a user manipulates a point that is constrained to be on a line, the constraint could be resolved by moving the line with the point, by restricting the user's interaction so that the point never leaves the line, or by a combination of the two. We must choose one of these as we implement the toolkit. These interaction techniques can be thought of as hints to a constraint solver when the constraint network is underdetermined so it can provide real-time, precise interactions.

[1]That is, encapsulations of geometry and behavior.

In addition to having an internal constraint network, a primitive widget can be related to another primitive widget by linking a port of the former to a port of the latter. This establishes a constraint (bi- or uni-directional) between the two ports. Ports are already constrained by the internal constraint network of a primitive, and the new constraints must be consistent with the existing constraints. Therefore, associated with each port is a function that determines how to attach new constraints to that port and how to modify its interaction techniques so as to facilitate constraint maintenance.

4 Description of the Toolkit Primitives

Having selected this framework to build our toolkit, we designed a general set of primitives to allow the interactive construction of not only the various 3D widgets previously scripted, but also application objects such as parameterized geometric models. These primitives are intended to be general enough to allow exploration of a wide set of object designs without having to resort to hand-coding.

We chose a "coordinate system" metaphor as a basis for our primitives. Each primitive visually represents a 0D, 1D, 2D, or 3D coordinate system and each can be constrained by affine transformations to the coordinate systems of other primitives. This metaphor can be used to express a wide variety of user interactions, including those of our previous 3D widgets [5; 14; 9]. However, the coordinate-system metaphor is only a framework for conceptualizing the primitives, not a strict definition of them. That is, the primitives were designed with regard to the sometimes antagonistic desires both to represent the coordinate system metaphor faithfully and to provide the semantics most useful for geometric and behavioral constructions.

The toolkit has primitives that correspond to position, orientation, measure (linear and angular), 2D and 3D Cartesian coordinate systems, a general extension mechanism for importing an arbitrary relationship, and the full set of UGA's geometric models.

The two most basic primitives, *Point* and *Ray*, encapsulate position and orientation respectively. Points and Rays represent 0D coordinate system entities; *i.e.*, there is only one element[2] of a Point or a Ray and therefore 0 coordinates are required to specify it. (Contrast this with a line, which has an infinite number of elements, each specified by one coordinate.) The Point primitive, represented by a small sphere, is an abstraction of a single 3D point. The Ray primitive, represented by an arrow, corresponds to a based vector, although we often treat it as just a vector (its position being a display convenience). Both primitives can be freely translated in space, but only the Ray can be rotated.

The notion of distance (linear measure) is represented through a 1D coordinate system primitive, the *Length*, represented by two Points, a port for the 1D coordinate system (represented by a thin cylinder connecting the Points), and a port for the Length's measure (represented by a small marker at the middle of the thin cylinder). While the Length appears as a bounded line segment, it actually encapsulates the notion of an infinite 1D coordinate system whose origin is at the line's start point (indicated by a small disc) and whose unit length is equal to the distance between the two points measured in the world coordinate system. We reuse the Point for the endpoints to help define the user interaction with the Length. Each of the Length's endpoints can be directly translated while the other remains fixed. Translating the cylinder joining the two Points translates both endpoints by the same amount. An alternate formulation of the Length would have both endpoints move whenever either was translated. Choosing either formulation is difficult in the absence of an application, so we chose the technique that seemed most useful.

Angular measure is represented by a two-handed clock-like primitive, the *Angle*. Each hand of the clock represents a vector and the outer ring of the clock represents the angle between the two vectors.

[2] In the sense of sets.

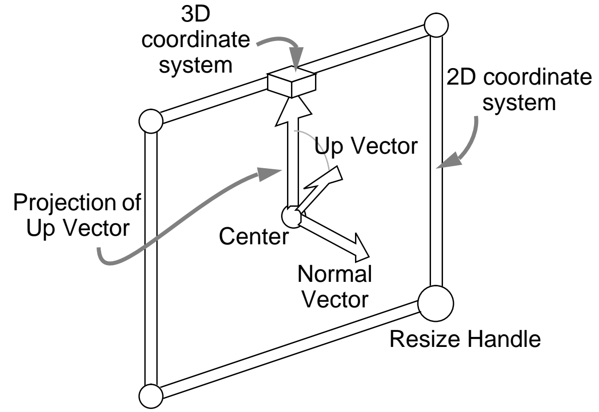

Figure 1: The ports of the Plane primitive.

The most complex primitive, the *Plane*, represents both a 2D Cartesian plane and a 3D Cartesian space. We opted to combine both concepts into a single primitive because users frequently use the two concepts in conjunction with one another and because the sets of ports are nearly identical, with a space being a superset of a plane. Visualizing an oriented plane requires ports for the plane's normal, center, and up-vector (similar to the PHIGS *VUP*), and for the size of a unit vector in each of the plane's axes. In addition, a port is required for the concept of the plane itself (as opposed to parameters that define the plane). A rectangle in the plane represents this port; its size determines the magnitude of each unit vector in the Plane's coordinate system. We also include a useful port for the projection of the plane's up-vector onto the plane, although this is not a required part of a Cartesian plane. To handle a 3D Cartesian space, the only additional port required is something to represent the concept of the space itself. The Plane reuses Points and Rays and introduces new geometry to represent the concept of the plane (a rectangle) and the space (a cube at the top of the up vector).

In order that the toolkit be extensible enough to handle new problem domains, there are also *Black-box* primitives, each representing a relationship with some number of ports that lacks a natural geometric representation. Ports on black-boxes are geometrically represented as labeled buttons. The accompanying video shows two Black-boxes: an interface to Barr's nonlinear deformation functions [2] and a PHIGS camera specification [6].

Finally, all the geometric objects in 3D modeling environments (cubes, spheres, CSGs, etc.) are considered collectively as a single primitive class called *Geometries*. In terms of the data it represents, each of the Geometries is essentially equivalent to a Cartesian space, although it is not annotated with additional geometry (as is the Plane primitive). In our system, each geometric object has an internal boundary representation relative to a local object coordinate system. This local coordinate system is used as a default coordinate system associated with a Geometry primitive to make it functionally equivalent to a Cartesian space. Since Geometries are not annotated with the ports of a Plane primitive, linking operations must infer from the context of the link operation which port of the implicit Cartesian space is intended. Linking operations usually apply to the origin of the Geometry's local coordinate system, though they can apply to the local coordinate system's normal and up-vector. When the default linking operation chooses the wrong port, the user can override the choice by making the object's local coordinate system explicit and choosing ports directly.

5 Linking the Toolkit Primitives

We now describe what occurs in the toolkit when a port of one primitive is linked to a port of another. Again, our choices for

the semantics of inter-primitive linking are guided by the desire to stay close to the coordinate-system metaphor and the desire to have reasonable behaviors when there is no obvious answer in the underlying metaphor.

A linking operation generally asserts one of two types of relations: it either establishes a bi-directional equality relationship between two similarly typed ports or projects one port into the coordinate system of the other port, using their common 3D embedding as the medium of projection.

Consider linking a Point to another Point: here, the first Point is set to be positionally equivalent to the second Point. However, linking a Ray to a Ray is slightly different in that the orientation of the first Ray is made equivalent to that of the second Ray, but the positions of the two Rays remain distinct. Rotating either Ray causes the other to change, but translating either Ray has no effect on the other. This choice of how to link two Rays together is ambiguous, because a Ray actually represents two geometric values, a position and an orientation. Thus the action is chosen by considering the context of the linking operation. In linking a Ray to a Ray, the user typically wants them both to have the same orientation, so only the orientation values are linked. If a user wishes to equate the positions of the Rays, then the position port of the Ray must be made explicit by linking each Ray to a common Point.

A different form of linking occurs when a lower-dimensional primitive is linked to a higher-dimensional one. Such a link causes the lower-dimensional primitive to be geometrically projected onto the implied *span*[3] of the higher-dimensional primitive. After this projection, the lower-dimensional primitive is associated with a coordinate in the higher-dimensional primitive based on the location of the lower-dimensional primitive in the span of the higher-dimensional primitive. This association is then enforced during subsequent manipulation. Typically, higher-dimensional primitives are composed of a number of lower-dimensional primitives, each of which can still be linked to higher-level primitives (*e.g.*, the center point of a Plane primitive is a Point primitive and can be linked to other higher-dimensional primitives.)

To illustrate, consider linking a Point to a Plane. This link operation causes the Point's position to be projected onto the Plane. The Point is then constrained to be at the coordinate associated with that projection point, unless it is moved directly. Whenever the Plane is manipulated, the Point will remain at the same position relative to the origin and orientation of the Plane. Yet, if the Point is manipulated, it will move in the span of the Plane, and thereby change its associated coordinate in the Plane's span.

Some link operations do not fall directly into either category. When this occurs, we chose what we considered the most reasonable solution. For example, we defined the linking of a Geometry primitive to a Length's measure port as a scale operation on the Geometry primitive along the axis of the Length. If the Length's orientation is linked to a principal axis of the Geometry primitive (or vice versa), then the Length acts as a standard 1D scale operation along that axis; otherwise it is a shear.

Figure 2 displays the link behavior that applies to the toolkit primitives when neither primitive has been linked to anything else. In cases where one primitive has already been linked, very different behavior may result; space prevents us from defining all these possibilities. Consider a Point linked to a Plane. The Point becomes constrained to move only in the Plane. If the Point is subsequently linked to a second Point, a different table takes into account the pre-existing constraints on the first Point. In this case, the first Point is constrained to lie at the position of the projection of the second Point onto the Plane.

[3] In the linear algebra sense; a Length's span is the line defined by the endpoints, a Plane's span is the plane defined by the Plane's center point and normal vector.

6 Implementation details

The toolkit is implemented in UGA's scripting language, with geometry provided by UGA's interactive solids modeler. The linking constraints between primitives are established using UGA's object-dependency network.

User feedback is provided in the course of a linking operation to aid in link specification. When the user picks a primitive to be linked, it is highlighted and the cursor changes to indicate that the system is waiting for the user to pick the object to link to. After the user picks the object to link to, the system indicates its "ready" state through a cursor change that prompts for a mouse click to confirm the link.

Other highlighting methods indicate a primitive's degrees of freedom. For example, a Ray, like other primitive widgets, is green when it is created, indicating that it is unconstrained. If it is linked to another Ray, its orientation is linked but not its translation, and it turns yellow to indicate a partial constraint. When it is linked again to a Point, it turns red, indicating that all of its degrees of freedom are constrained. Another possibility would have been to change the primitive geometries after linking (*e.g.*, a spherical Point primitive could become a thin cylinder when it is linked to a Length, and could become a disc when linked to a Plane, although this strategy can result in a overly large collection of shapes).

7 Future Work

The toolkit as described lacks techniques for specifying range limits on a primitive's degrees of freedom. These would be especially useful when modeling the behavior of real-world objects, or when creating interface objects such as bounded sliders, joints, and dials. We intend to add this functionality (and perhaps other inequality constraints too), and also extend the range of our toolkit to deal with other graphics concerns, such as surface and volumetric modeling, scientific data exploration of scalar and vector fields, and behavior modeling including dynamic simulations.

When two primitives are linked together, a single constraint based on Figure 2 is installed. However, it would often be useful to have a set of possible link behaviors that the user can select from. Advanced users would be able directly select the desired behavior with only a single link operation.

Once a complex widget has been constructed from primitives, it is useful to interactively encapsulate it, along with appropriate parameters, for reuse in a tool library. For example, having constructed a shadow widget, the user should be able to easily apply the same process to any other object. This amounts to interactively defining a function and embodying it in a new, higher-level primitive.

Highly complex widgets linked together from dozens of primitives may present efficiency problems, especially for real-time interaction. It may be necessary to optimize the constraint network after the widget has been completed in order to maximize the toolkit's evaluation speed. It would also be useful to display graphically the constraint relations between primitives to provide feedback on the links established on any widget.

8 Conclusions

This toolkit provides a methodology for interactively constructing the geometric behavior of a variety of 3D widgets and parameterized 3D application objects, so that non-technical users can rapidly and interactively generate constrained 3D objects. Previously, such widget construction required programming in C or our scripting language. Even for experienced programmers, graphical construction is a more suitable and efficient environment to conceive, prototype, and implement many types of interactive 3D objects.

destination (columns) / source (rows)

Linking constrains	Point	Ray	Length Body	Length Measure	Angle Measure	Plane Frame	Plane Space	Geometry
Point	positions are equated	Point to lie on Ray	Point to lie on Length body	×	×	Point to lie in Plane	Point to be in Plane's 3D coordinate system	position of Point to position of Geometry
Ray	Ray to position of Point	orientations are equated	orientation of Ray to orientation of Length	orientation of Ray to be orientation of length measure	×	orientation and position of Ray to lie in Plane	Ray to be in Plane's 3D coordinate system	orientation of Ray to orientation of Geometry
Length Body	×	End Points of Length to lie on Ray	×	×	×	End Points of Length to lie in Plane	End Points of Length to be in Plane's 3D coordinate system	×
Length Measure	×	End Points of Length to lie on Ray	×	length of first Length to be length of second Length	length of Length to map to Angle's measure	×	×	×
Angle Measure	×	×	×	Angle's measure to map to length of Length	first Angle's measure to be second Angle's measure	×	×	×
Geometry	position of Geometry to position of second Point	orientation of Geometry to Ray's orientation	Geometry to lie on Length body	scale of Geometry to length of Length	×	Geometry to lie in Plane	Geometry to be in Plane's 3D coordinate system	positions are equated

Figure 2: Linking behaviors for unconstrained primitives.

Acknowledgments

This work was supported in part by the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization and by ONR Contract N00014-91-J-4052, ARPA Order 8225. We also gratefully acknowledge the sponsorship of IBM, NCR, Sun Microsystems, Hewlett Packard, Digital Equipment Corporation, and NASA. We thank Andries van Dam and the members of the Brown University Graphics Group for their help and support. Please contact the authors for a copy of the accompanying videotape.

References

[1] AVS, Inc. *AVS Developer's Guide, v. 3.0*, 1991.

[2] A. H. Barr. Global and local deformations of solid primitives. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):21–30, July 1984.

[3] Eric A. Bier. Snap-dragging in three dimensions. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):193–204, March 1990.

[4] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 181–188, 1991.

[5] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-dimensional widgets. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2):183–188, March 1992.

[6] James D. Foley, Andries van Dam, Steven Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd edition, 1990.

[7] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):331–340, July 1992.

[8] Paul E. Haeberli. Conman: A visual programming language for interactive graphics. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):103–111, August 1988.

[9] Kenneth P. Herndon, Robert C. Zeleznik, Daniel C. Robbins, D. Brookshire Conner, Scott S. Snibbe, and Andries van Dam. Interactive shadows. *1992 UIST Proceedings*, pages 1–6, November 1992.

[10] Michael Kass. CONDOR: Constraint-based dataflow. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):321–330, July 1992.

[11] 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 magazine*, pages 71–85, November 1990.

[12] Open Software Foundation. *OSF/Motif Reference Guide*.

[13] Steve Sistare. Graphical interaction techniques in constraint-based geometric modeling. In Steve MacKay and Evelyn M. Kidd, editors, *Graphics Interface '91 Proceedings*, pages 161–164. Canadian Man-Computer Communications Society, March 1991.

[14] Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, D. Brookshire Conner, and Andries van Dam. Using deformations to explore 3d widget design. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):351–352, July 1992.

[15] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):341–349, July 1992.

[16] 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. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):105–112, July 1991.