LOG: Building 3D User Interface Widgets by Demonstration

Larisa Matejic

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-22
May 1993

LOG: Building 3D User Interface Widgets by Demonstration*

Larisa Matejic†

May 20, 1993

*©1993 Brown University Computer Graphics Group. This work was supported in part by NSF, DARPA, IBM, Sun Microsystems, NCR, Hewlett Packard and Digital Equipment Corporation.

†Department of Computer Science, Box 1910, Brown University, Providence, RI 02912

Abstract

Building 3D widgets was until now considered to be a costly task performed only by experts: programmers or people who understood complexities of the constraint networks found in the widgets they were constructing, as well as the mathematical details surrounding the operation their widgets were to perform. This paper addresses these problems and presents `LOG`, an interface for 3D widget construction. `LOG` is an inference-based 3D widget construction toolkit that facilitates quick and intuitive prototyping of widgets. When using `LOG`, designing widgets requires absolutely no programming on the user's part or understanding of the constraint networks and mathematical equations involved. Instead, all specification of constraints is done by example, where the users simply presents configurations of the widget they are constructing from defining the geometry of the widget to mapping the widget to an operation.

1 Introduction

Building 3D widgets was until now considered to be a costly task performed only by experts: programmers or people who understood complexities of the constraints in the widgets they were constructing, as well as the mathematical details surrounding the operations their widgets were to perform. This paper addresses these problems and presents `LOG`, an interface for 3D widget construction. The goal of our research was to build a widget construction toolkit in 3D that would be intuitive and simple to use. We also wanted our system to have a demonstrational interface, so that it would be accessible to non-programmers. Most of all, we wanted to design a system that would be efficient and practical, so that it could be used to solve real widget-design problems. `LOG` is an inference-based 3D widget construction toolkit that facilitates quick and intuitive prototyping of widgets. When using `LOG`, designing widgets requires absolutely no programming on the user's part or understanding of the constraint networks and mathematical equations involved. Instead, all specification of constraints is done by example, where the users simply presents configurations of the widget they are constructing from defining the geometry of the widget to mapping the widget to an operation.

2 Background and Related Work

LOG uses constraints to express the concepts it learns during the process of specifying geometry and behavior of widgets. Numerous systems have been built (especially in 2D) that use constraints to allow the user to build complex objects from simpler ones. Those systems can be classified according to the kind of user interface they employ: *explicit constraint specification* or *demonstrational constraint specification*. In explicit constraint specification systems the user is required to have a thorough understanding of the constraint networks involved, since he has to express all the constraints directly. On the other hand, in demonstrational constraint specification systems the user does not need to know much about constraints, since constraints are inferred by the system from the user's examples.

We only mention the landmark systems in the field for reference, because none of these systems with the exception of [KURL91] is particularly similar to LOG.

2.1 2D

The two landmark explicit constraint specification systems are Thinglab [BORN86] and Garnet [MYERS90]. Thinglab provides preconstrained parts which the user can assemble to create complex shapes. The user can also specify additional constraints on existing objects. Garnet helps the user interactively create user interfaces that let the user operate on graphic objects with the mouse and keyboard.

The three landmark demonstrational constraint specification systems are Peridot [MYERS88], Metamouse [MAUL89], and [KURL91]. Peridot is a system that infers graphical constraints automatically as objects are added to the scene. Peridot confirms all its inferences with the user, which gives the user hints about what the constraints present might be; however, the user still needs to understand thouroughly what constraints are. Peridot's inferring capabilities are limited, since it can only infer one-way constraints. Metamouse keeps track of the user's procedural editing actions and attempts to infer generalized procedures from them. For example, if the user aligns boxes along a line, `Metamouse` will infer the iteration and align the rest of the boxes. [KURL91] is meant to be used for doing geometric constructions. The system infers constraints by using the snapshot method, which requires it to draw information from still configurations, not dynamic environments.

2.2 3D

In 3D, the situation is less well-developed. We know of only one explicit system [ZELE93] and *no* prior work on demonstrational constraint specification. [ZELE93] is a 3D widget construction toolkit that allows the user to construct widgets by bringing up primitives and clicking on appropriate icons to constrain objects in a specified way. There is absolutely no inferencing involved. In addition, the user does not have the freedom to custom map widgets to black box functions.

3 Widgets

Widgets encapsulate geometry and behavior and are used to change or display information and properties of application objects. Geometry refers to the appearance of a widget and the term behavior describes a widget's functionality.

Standard examples of 2D widgets are sliders, scroll bars, and dialog boxes.

Some examples of 3D widgets include the rack, virtual trackball, and the warp widget[CONN92].

Three major steps are involved in constructing 3D widgets:

- Specifying the geometry of widget components - involves declaring the 3D geometric primitives that the widget is composed of.

- Specifying geometric constraints among components of the widget - involves putting the 3D primitives together to define the geometry or appearance of the widget.

- Mapping widget manipulations to black box functions - involves giving functionality to the widget or specifying the effect different widget configurations have on application objects (We will say more about "black boxes" in Section 4.5.1).

4 LOG

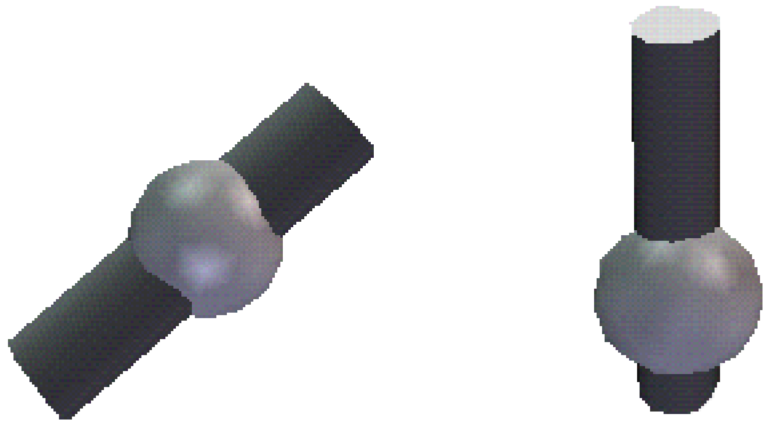
LOG allows the user to effortlessly construct 3D widgets. Its emphasis is on providing an easy way of accomplishing the latter two steps of widget construction: assembling the geometry of a widget and defining a relationship between a widget and the application objects the widget controls.

LOG infers both geometric and algebraic constraints. By assembling the geometry of a widget the user indirectly sets up constraints among the parts of the widget. Those are the *geometric constraints*. When the user associates the widget he built with an operation, he sets up *algebraic constraints* between the configuration of the widget and the numerical inputs to the operation.

Positioning objects in screen space precisely is a difficult task for any user. In 2D, various solutions have been introduced: grids, snapping and gravity[BIER86A]. All of these somewhat restrict the expressive freedom of the user in 2D and when generalized to 3D[BIER86B], the effect becomes even worse, partly because the user mostly uses a 2D input device to manipulate objects in three-space. In designing LOG, we wanted to remove the burden of accurately positioning objects in 3D from the user. We therefore designed LOG to be immune to noise found in the user's input, so that it would infer concepts the user intends without requiring precise data from the user. We say that LOG handles *approximate constraints*: it is sufficient for the user to specify approximately what he is trying to communicate to LOG.

4.1 The Snapshot Method

The user communicates with LOG by using the snapshot method[KURL91]. In doing so, the user draws a configuration that approximately complies with his design goals for the widget from which a number of possible constraints are inferred. As subsequent configurations are drawn, the system breaks (or eliminates) constraints that no longer hold, making each snapshot a valid solution of the constraint network. After a number of snapshots, we expect LOG to converge on a constraint set that when enforced exhibits behavior (geometric and/or functional) that is visually consistent with the user's examples.



snapshot 1 snapshot 2

Figure 1: Example of using the snapshot method to define a 3D slider

The user is the one who indicates to LOG when the learning process is complete. He simply presses the `Infer Concept` button, after which he can opt to switch into the `Manipulate Tool` mode to seek visual feedback from LOG indicating which constraints were inferred during the learning process. When `Manipulate Tool` mode is selected, the constraints inferred during the learning process are enforced, so that the user can get a feel for what they are as he moves the tool around. For instance, if LOG inferred that two spheres are five units apart, the user will notice in `Manipulate Tool` mode that as he moves one of the spheres, the other one follows so as to make the distance between them always equal five units. Additionally, if the user is not happy with what `LOG` inferred, he can deselect the `Manipulate Tool` mode and take additional snapshots to eliminate certain constraints, or delete some snapshots that were already taken to add constraints that were broken by particular snapshots. Therefore, the user can temporarily suspend the learning process at any time to see how the snapshots he gave as input to `LOG` are furthering his design goals for the widget. The feedback he gets from `LOG` gives him direction for future snapshots.

There are numerous advantages to using the snapshot method for purposes of communication between the user and the system[KURL91, MYERS92]. Complex constraint systems can be specified with few operations, allowing novices to learn basic functionality quickly. Furthermore, experienced users can work rapidly to carry out a wide range of tasks. The snapshot method makes for a friendly system that is intuitive to use. Users do not need to be aware of all the constraints that must hold; they need only know the visual end result they expect. In addition, the snapshot method gives users a way to get rapid feedback; they can stop the teaching process at any point to see what the system has learned. Finally, the interface hides the underlying constraint set, and thus makes any changes or additions to the constraint set transparent to the user. Therefore, the user is not required to learn new functionality to use different constraint sets.

4.2 Example of Using LOG

We will illustrate each part of `LOG` by tracing the process of constructing a color-picker widget. Our color-picker widget will consist of three attached cylinders and three spheres. Each sphere will be able to move along the axis of one of the cylinders, and its position in space with respect to the cylinder it moves along will determine the value of one of the color channels of an application object.

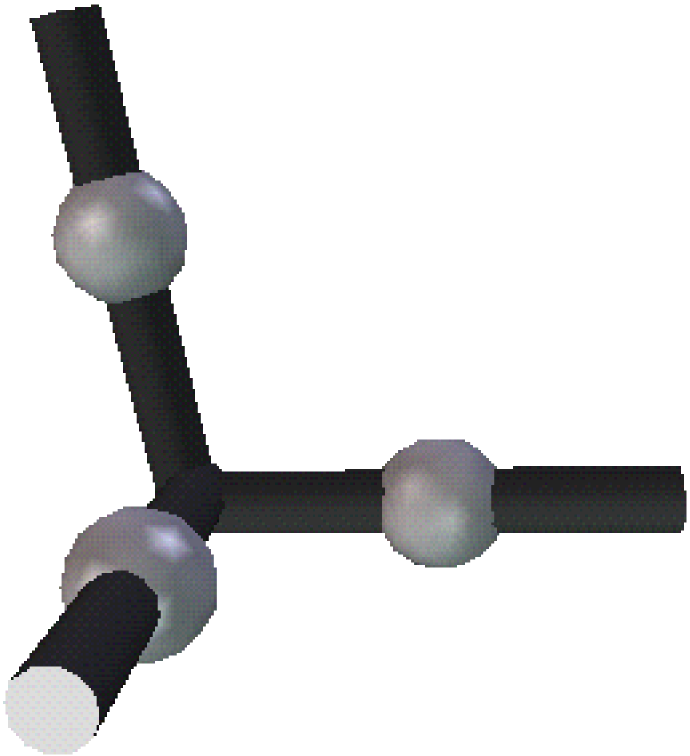


Figure 2: Color picker widget

4.3 Specifying the Geometry of Widget Components

The first stage of LOG requires the user to specify the 3D primitives that make up the widget. For the color-picker, the user would instantiate three spheres and three cylinders in the work area.

4.3.1 Objects

Objects in this phase of LOG are geometric 3D primitives, such as spheres, boxes, and cylinders. Each is internally represented by a list of points, vectors, and numbers. The representation of widget components exploits the inherent properties of each primitive. For instance, a sphere is represented by a point (its center) and a number (its radius) and a box is represented by 8 points (its vertices), 12 vectors (its sides), and 3 numbers (its three side lengths). This set of primitives can be enlarged, as long as each primitive is represented consistently with the rest; every primitive must be uniquely specified by its type (sphere, cube, etc.) and its list of points, vectors, and numbers.

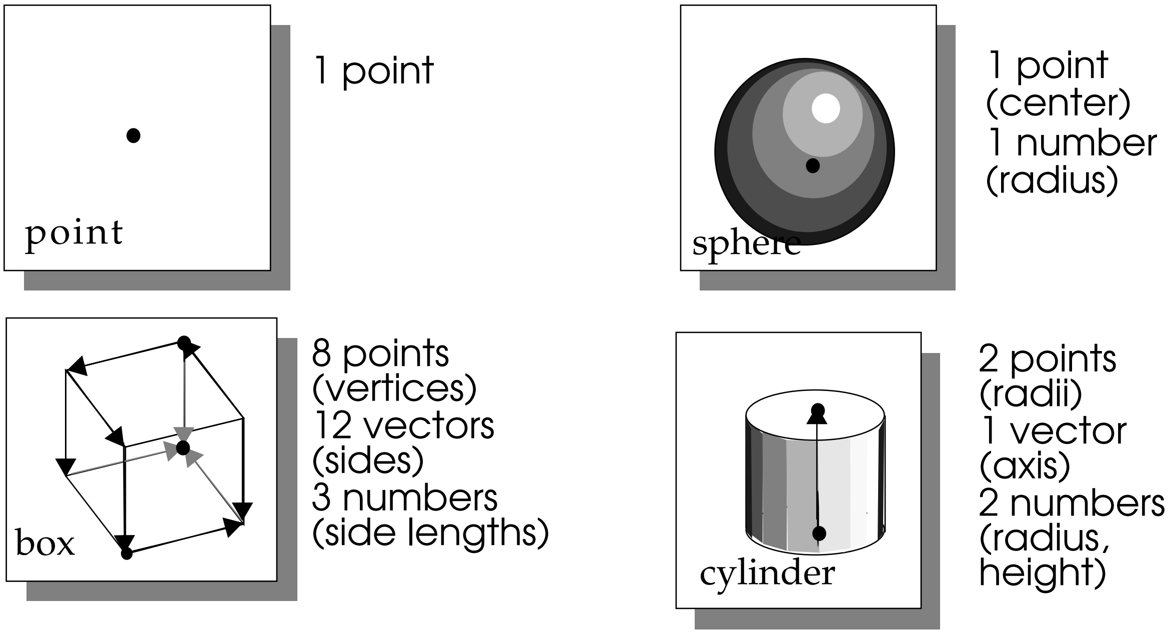


Figure 3: Representation of geometric primitives

4.4 Specifying Geometric Constraints Among Widget Components

In the second stage the user provides examples of legitimate configurations of the geometric components of the widget, from which his design intentions will be inferred. The user does so by using the snapshot method to set up a number of acceptable configurations of component objects, i.e. configurations that comply with desired geometric behavior for the widget. In our color-picker example, the user needs to communicate that the three cylinders are held together and that the spheres can each move along the axis of one of the cylinders. One snapshot could consist of the three cylinders all touching at one end and one of the three spheres placed somewhere along each of the cylinders. Another snapshot could show the three cylinders, again touching, and each of the three spheres placed at a new point along the axis of one of the cylinders. It is important to note that the order in which the snapshots are given will not affect what is learned in the end.

4.4.1 Constraint Set

We infer constraints among widget subcomponents, namely points, vectors, or numbers that charecterize the 3D primitives that comprise the widget. From widget subcomponents we derive objects that we can constrain, i.e. *constrainable objects*: from each point we extract the coordinates; we consider all pairs of points and compute the distance between the points in each pair; additionally, we compute dot products of each pair of vectors; finally, we calculate the sums of all pairs of numbers. It is these objects that we associate constraints with.

The possible constraints that might hold among these entities, the constrainable objects, are:

- **Distance** - We infer constraints that involve distances that are constant and distances that are equal to the sum of two numbers. For instance, in our color-picker example the distance between the two cylinder endpoints remains constant throughout the learning process.

- **Dot product** - We infer only constraints that involve constant dot products. In our color-picker example, the dot product of the unit vector pointing from one of the cylinder endpoints to the center of the sphere and the unit vector pointing along the cylinder axis stays approximately 1.0 throughout the learning process.

- **Same x, y, or z coordinate** - We infer constraints that assert that two points have the same x, y, or z coordinates. We infer constraints of this type mainly because people sometimes like to align objects with the major axes.

4.4.2 Taking Snapshots

Internally, each snapshot has a set of constraints associated with it. In the first snapshot we constrain all constrainable objects. Thus, for example, we generate a constraint that says that each number object is equal to its constant value. When the user takes the second snapshot, the set of constraints that were inferred in the first snapshot are passed along to the second snapshot to be validated. This simply means that for each constraint in the constraint set, the second snapshot locates the objects that are involved and checks to see if those objects still satisfy that constraint. For instance, we can consider a distance constraint present in the first snapshot that states that point A is five units away from point B. In analyzing the second snapshot, we would locate points A and B and compute their distance. If the distance computed is approximately five units, the constraint would remain present in the second snapshot. This is where the notion of approximate constraints comes into play in this stage of LOG. The constraints need only have sufficiently close, not identical, values in the two snapshots to be considered valid. If a constraint still holds in the second snapshot, it is added to the second snapshot's list of constraints; otherwise, it is discarded. Thus, the set of constraints associated with the second snapshot consists of constraints that hold in both snapshot one and two; it is equivalent to the intersection of the set of constraints present in the first snapshot and the set of constraints present in the second snapshot. Similarly, the third snapshot works with the set of constraints associated with the second snapshot to form its set of constraints, and so on.

```
if (first_snapshot) {
   /* compute all constrainable objects */
   compute all distances;
   compute all dot products;
   ...
   /* constrain all of them */
   each dist = const;
   each dot_prod = const;
```

```
    ...
}
else
    keep only the constraints that have
    not been violated from previous
    snapshot
```

4.4.3 Inference Algorithm

The user decides when to stop providing snapshots to LOG. Of course, the user can in the same manner temporarily suspend the learning process at any time to see how the snapshots he gave as input to LOG are furthering his design goals for the widget. What happens when the user indicates to LOG that he is done (or temporarily done) providing input is that LOG obtains the set of constraints associated with the last snapshot to be the representation of the concept the user is trying to communicate. The reason is that we know ahead of time that the concept the user is attempting to illustrate with his snapshots can describe any or all the snapshots he gave and thus must be comprised of the set of constraints that all the snapshots have in common. Since each constraint belonging to the set of constraints associated with the last snapshot taken has propagated through from the first snapshot, it is one that all the snapshots provided have in common, and should thus be incorporated into the set of constraints describing the user's concept.

If, for some reason, the user decides to delete one of the snapshots, the set of constraints associated with the snapshots that follow it is recomputed by propagating the constraint set associated with the snapshot that precedes the deleted one.

4.5 Mapping Widget Manipulations to Black Box Functions

4.5.1 Black Boxes

A black box is our abstraction of a collection of mathematical functions. For instance, the color-picker black box is comprised of three functions: one that changes the red channel, one that changes the green channel, and one that changes the blue channel of the color of an object. The inputs of a black box are either numbers or vectors (lists of three numbers). The color-picker balck box has three numbers as its inputs.

4.5.2 Description of the Mapping Stage

In this stage the user provides information about the widget's functionality. He selects a black box to map the widget manipulations to. An example, or a snapshot, in this stage consists of a widget configuration and a set of inputs to the black box. In essence, the user is stating that whenever the widget gets into a specific configuration, the input values to the black box should be the ones given in the snapshot. Snapshots of different widget configurations and different black box values are used to derive correspondences. This phase of `LOG` finds relationships between the changing numerical values that describe characteristics of the widget as it is manipulated and inputs to the black box function the widget maps to.

In our color-picker example the black box function changes the color of an object and has three parameters, each controlling a single color channel. The user might want to have the distance between each sphere and the end of the cylinder it moves along control a color channel. Some of the snapshots he can provide to suggest that correspondence are the following: all spheres are at the midpoint of their cylinder axes and the black box values are set to 0.5, 0.5, and 0.5; the sphere controlling the red channel is placed halfway up its cylinder, the other spheres are at the points where the three cylinders intersect, and the black box values are set to 0.5, 0.0, 0.0.

4.5.3 Objects

Objects in this phase of `LOG` are divided into two categories: work area objects and black box objects. Work area objects are properties of the geometry of the widget. In particular, we are concerned with changes in numerical values of distances between points, displacement vectors between points, angles between vectors, and absolute positions of points and vectors. Black box objects are the inputs of the black box.

4.5.4 Taking Snapshots

When the user takes a snapshot, `LOG` computes the current value of each work area object and records the value of each black box item.

4.5.5 Mapping Algorithm

When the user has finished taking snapshots, `LOG` is left with two sets of arrays: the work area object array set and the black box array set. The work area object array set contains an array for every work area object. Similarly, the black box array set contains an array for every black box object. Each array in either set has a length equal to the number of snapshots taken and its elements are numerical values that were recorded during each snapshot.

We now need to find linear relations that hold between black box items and work area object items. We can look at this problem as a problem of fitting a line through the data points from our arrays. We can plot any black box item against any work area object by plotting (x,y)-pairs, where x is the black box array value and y is the work area object array value for the same snapshot. Then, we proceed by fitting a line (using a least-squares fit [STRANG88]) through the points. What we are looking for is the best such fit, where the error of fitting a line through the data points is minimized. This means that if we plot each work area object against a black box item, we would choose to map the black box item to the work area object that gives the best linear fit. We do this for all black box items.

The implementation of the mapping algorithm in `LOG` does not use the brute force method of computing a linear fit for every black box item and every work area object. We

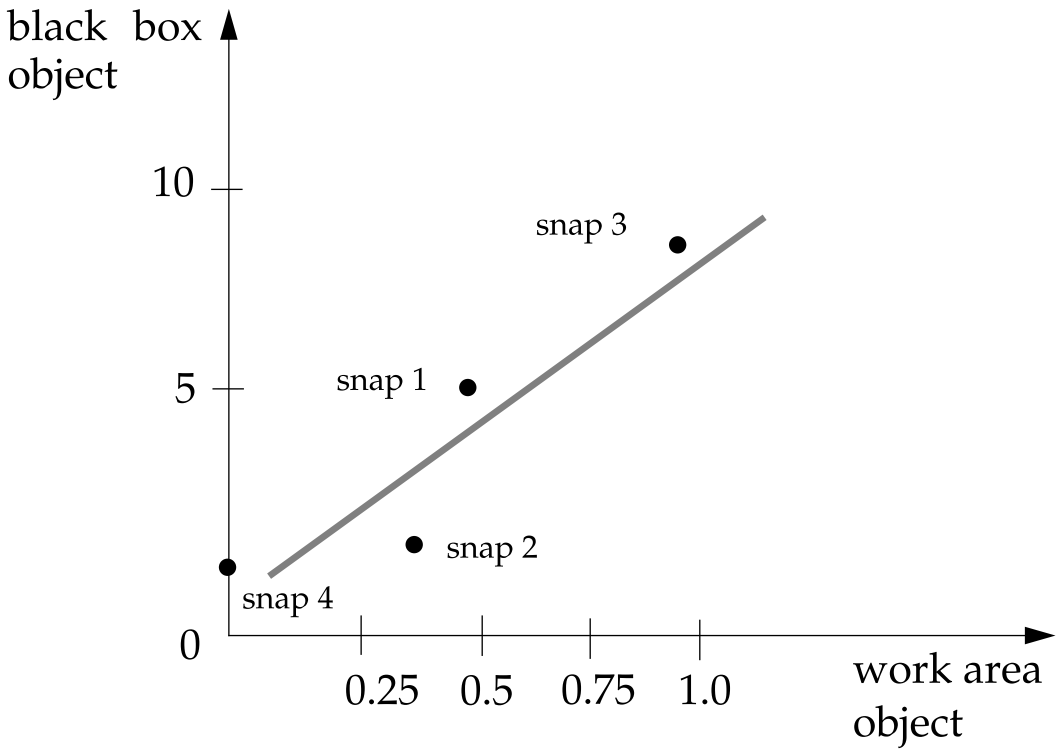


Figure 4: Example of fitting a line through snapshot data

use what we call a *preferred fit*. We divide all work area objects into two categories: numbers and vectors (the same categories that exist for black box items). Distances between points and angles fall into the number category, whereas displacement vectors between points, absolute positions of points, and absolute positions of vectors are all classed as vectors. Our experience in widget design has shown that numbers are more likely to map to numbers than vectors, and vectors are more likely to map to vectors than numbers. Therefore, if we are trying to find a mapping for a black box item that is a number, we will compute linear fits for all data points in the number array set of work area objects. If that does not give us a satisfactory fit, we search through the set of work area vectors. A fit is satisfactory if the error of the fit is not "too large", i.e. does not exceed some cut-off value set by the programmer.

When the mapping process is completed, every black box item is associated with an equation of the form `black_box_item[i]=a*work_area_object[j] + b` that describes the linear mapping from properties of objects in the work area to black box parameters. Now, as the user manipulates the widget into different configurations, the numerical values describing the properties of the widget are changed, thus affecting input values to the black box and thereby changing the attribute of the application object controlled by the black box.

5 Future Work

The code for mapping angles from the work area to sliders will have to be developed. Interpreting angle information from the user's actions introduces ambiguity, since there is no easy way to tell how many times the user rotated a vector by 360 degrees and whether that information is important in learning the concept.

Also, we will implement the multiple concept feature, which makes it easy for the user to construct large widgets incrementally. The feature allows the user to highlight objects of current interest. LOG only infers constraints among the highlighted objects. When the user is done assembling the widget, all concepts that were inferred separately will be unified into a single concept.

Additionally, it will be interesting to explore the possibility of adding inequality constraints to LOG, so that a concept like "object A is to the left of object B" can be inferred. In our color-picker example, LOG did not infer that the spheres should not go past the end of the cylinders they were moving along. In such a situation it would be useful to have inequality constraints.

Most of all, it will be challenging to add negative examples. Negative examples describe configurations that are not in accordance with the user's design goals for the widget, or invalid configurations. Some concepts, like the ones involving inequality constraints, can not be easily communicated without the use of negative examples. Of course, one could take positive examples as indicating extrema in the design space. However, that would complicate specification of many other concepts, since the user would be required to input additional examples to break the unnecessary extrema constraints. For instance, in assembling the geometry of the color-picker widget, the user could give a couple of positive examples with the spheres lying on the cylinders and a couple of negative examples: where the spheres are moved just beyond the cylinders on one side and then the other.

6 Conclusions

LOG is the first 3D inference-based widget construction toolkit. It uses a constraint inference algorithm rather than direct specification of constraints by the user. This feature allows non-programmers to gain access to domains usually thought to be exclusively reserved for programmers. LOG handles approximate constraints, thereby doing away with the need for grids or other positioning aids. The approximate constraint feature provides a natural way for users to place objects in space: it gives users the freedom to position objects while eliminating the clutter of additional positioning tools.

LOG introduces a notion of inferring algebraic, as well as geometric constraints, so that it becomes possible to establish a relationship between a widget and an operation it performs. In other words, LOG not only infers constraints present in geometric constructions, but also constraints encountered in widget behavior.

Most important of all is probably the fact that LOG has practical efficiences. It runs in O(n^2) time, where n is the number of widget subcomponents. Because the Brown Graphics Group research on 3D widgets has shown that it often takes only five or six objects to construct many 3D widgets, we expect n to be small. Even if 3D widgets in the next generation are composed of a large number of objects, we can use the multiple concept method described in Section 5 to partition the tool-building task into smaller parts and thus reduce the local time complexity.

In summary, `LOG` makes it feasible for non-programmers to more easily design 3D widgets. The prototyping of widgets can be accomplished quickly by designers and may prove to be an essential tool for designing 3D widgets in the future.

References

[BIER86A] Bier, Eric A., and Stone, Maureen C. “Snap Dragging”, SIGGRAPH ’86 Proceedings, 1986.

[BIER86B] Bier, Eric A. “Skitters and Jacks: Interactive 3D Positioning Tools”, 1986 Symposium on Interactive 3D Graphics, pp. 183-278, October, 1986.

[BORN86] Borning, A. and Duisberg, R. “Constraint-Based Tools for Building User Interfaces”, ACM Trans. Graphics, Vol. 5, No. 4, Oct. 1986, pp. 345-374.

[CONN92] Conner, D. Brookshire, Snibbe, Scott S., Herndon, Kenneth P., Robbins, Daniel C., Zeleznik, Robert C. and van Dam, Andries. “Three-Dimensional Widgets”, 1992 SIGGRAPH Symposium on Interactive 3D Graphics, pp. 197-208, March, 1992.

[FALK86] Falkenhainer, B. C. and Michalski, R. S. “Integrating Quantitative and Qualitative Discovery: The ABACUS System”, Machine Learning I, pp. 367-402, 1986.

[KOKAR86] Kokar, M. M. “Determining Arguments of Invariant Functional Descriptions”, Machine Learning I, pp. 403-422, 1986.

[KURL91] Kurlander, David and Feiner, Steven. “Inferring Constraints from Multiple Snapshots”, 1991.

[LANG90] Langley, Pat and Shrager, Jeff, ed. Computational Models of Scientific Discovery and Theory Formation. Morgan Kaufmann Publishers; San Mateo, CA, 1990.

[MAUL89] Maulsby, David L., Witten, Ian H., and Kittlitz, Kenneth A. “Metamouse: Specifying Graphical Procedures by Example”, SIGGRAPH ’89 Proceedings, pp. 127-136, Aug. 1989.

[MYERS86] Myers, Brad A. and Buxton, William. “Creating Highly-Interactive and Graphical User Interfaces by Demonstration”, SIGGRAPH ’86 Proceedings, pp. 249-258, 1986.

[MYERS88] Myers, Brad A. Creating User Interfaces by Demonstration, Academic Press, San Diego, CA, 1988.

[MYERS89A] Myers, Brad A., Vander Zanden, Brad, and Dannenberg, Roger B. “Creating Graphical Interactive Application Objects by Demonstration”, Proceedings ACM SIGGRAPH Symposium on user Interface Software and Technology, ACM, 1989, pp. 95-104.

[MYERS89B] Myers, Brad A. "User-Interface Tools: Introduction and Survey", IEEE Interface Systems, pp. 15-23, January, 1989.

[MYERS90] Myers, Brad A., Guise, Dario A., Dannenberg, Roger B., Zanden, Brad Vander, Kosbie, David S., Pervin, Edward, Mickish, Andrew, and Marchal, Philippe. "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces", IEEE Computer, Nov. 1990.

[MYERS92] Myers, Brad A. "Demonstrational Interfaces: A Step Beyond Direct Manipulation", IEEE Computer, pp. 61-73, Aug. 1992.

[SNIBBE92] Snibbe, Scott S., Herndon, Kenneth P., Robbins, Daniel C., Conner, D. Brookshire and van Dam, Andries. "Using Deformations to Explore 3D Widget Design", SIGGRAPH '92 Proceedings, pp. 351-352, July, 1992.

[STRANG88] Strang, Gilbert. Linera Algebra and its Applications, 3rd Edition, Harcourt Brace Jovanovich, Orlando, Fla., 1988.

[ZELE93] Zeleznik, Robert C., Herndon, Kenneth P., Robbins, Daniel C., Huang, Nathan H., Parker, Noah F., Meyer, Thomas W. and Hughes, John F. "3D Widget Construction Toolkit",To appear in SIGGRAPH '93 Proceedings.