BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-94-M6

"A Toolkit for the Construction of Three Dimensional Interfaces"

by

Marc Stevens

A Toolkit for the Construction of Three Dimensional Interfaces

Marc Stevens

Department Of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

February 1994

[illegible]

Professor John F. Hughes
Advisor

A Toolkit for the Construction of Three Dimensional Interfaces

Marc Stevens
mps@cs.brown.edu

Dept. of Computer Science
Brown University
Box 1910
Providence, RI 02912

Abstract

This paper presents a toolkit for the construction and prototyping of three dimensional interfaces, interactive illustrations, and three dimensional widgets. The toolkit provides methods for the direct manipulation of 3D primitives which can be linked together through a visual programming language to create complex constrained behavior. Features of the toolkit include the ability to encapsulate and parameterize complex models, exposure of the constraint behavior to the user, the ability to impose limits, and an intuitive user interface.

1.0 Introduction

In this paper, we present a toolkit for constructing and prototyping three dimensional widgets, interactive illustrations, and three dimensional interfaces [5]. We begin with a core set of 3D primitives that can be combined in various ways to create more complex constructions. We add a powerful visual language that includes the notion of classes containing typed variables, upon which relational operations, e.g. (<, >,...), can be performed, and parametrized subroutines. Interface issues for the specification of constrained geometry are also addressed. Finally, we present the toolkit's architecture which is easily extended to add more complex functionality, enhance the visual language, and add new primitives.

Interface issues involved in a constrained based 3D toolkit include presenting constraints that can be established on objects, visualizing which constraints have been established, and easily modifying constraints once they are established. If these issues are ignored, it forces the user to resort to trial and error to get the desired behavior. Over the past several years, there have been many other systems developed for specifying constrained geometry in both two [17] and three dimensions [3][7][10]. These systems have had varying degrees of success in providing functionality and flexibility. To a large measure, the degree to which these systems have succeeded or failed is determined by how well they have addressed the interface issues mentioned above.

Brown University developed one such 3D toolkit for constructing three dimensional widgets [5] such as deformation racks [14], interactive shadows [9], and other constrained three dimensional geometries using a visual programming language. The toolkit provides a set of 3D widget primitives for constructing interactive behaviors by constraining the affine transformations of objects, and an interactive 3D interface for combining these widget primitives into more complex widgets.

Although Brown's toolkit was successful in constructing a fixed set of three dimensional widgets, it has many shortcomings. In every attempt to build a new widget, either the visual language or the primitive set was not powerful enough to support the new construction. The architecture of the toolkit made the addition of new primitives or functionality a difficult and time consuming task. The users of the toolkit found the interface for specifying constraints hard to understand and difficult to use. We address these issues in our new toolkit.

In section 2.0 we give an overview our toolkit and its capabilities. In section 3.0 we discuss the visual language. In section 4.0 we discuss the interface design issues in specifying constraints. Section 5.0 details the toolkit architecture and implementation details.

2.0 Overview Of The Toolkit

The toolkit provides direct manipulation of 3D primitives through a visual language. These primitives are used to construct widgets, interface objects, and application objects whose geometry is affinely constrained. Constraints apply not only to the geometry but may also be applied to other non-geometric attributes. The visual programming paradigm of our toolkit has significant advantages over methods used by other toolkits, such as libraries [12][15] and graphical networks [1][8][10]. The traditional approach to designing user interface toolkits is to use libraries of software objects which are created using standard programming languages. This makes the task of visualizing the complex relationships between these objects difficult. It also rules out the possibility of non-programmers using them to do interface prototyping. The second paradigm is based on the graphical manipulation of function networks. In this paradigm, the developer wires together 2D boxes that have no direct relation to the application objects they represent. Our toolkit's direct manipulation paradigm has both the advantage of direct manipulation of application objects and a visual language that allows non programmers to use the toolkit for interface protoyping.

3.0 Visual Language

We introduce a visual language for constructing 3D interfaces. This language provides the framework for the construction of constraint relationships between toolkit primitives. The language consists of *classes* represented by 3D toolkit primitives. These classes contain typed variables called *slots*. Slots on classes can be *linked* to establish constraint relationships between primitives. We can then create new classes by encapsulating complex collections of constrained primitives. These encapsulations add new primitive classes to the toolkit and can be optionally parametrized and called (like a subroutine) to recreate complex constructions.

3.1 Classes

A class is an abstraction of the geometry and behavior of a toolkit primitive. A geometric object is associated with each class to provide a visual representation. The class behavior is defined by the slots on the class and the class's interaction technique. We define a constraint relationship between two classes by linking the slots on those classes.

3.2 Slots

Slots are typed variables in our visual language and represent the constrainable quantities of a class. Each primitive has a geometric representation and one or more slots which define its behavior. Primitives are constrained by their slots and are initially unconstrained. Constraints are defined by linking the slots on one primitive to the slots on another primitive.

A linking operation is effected by the selection of a destination primitive slot and a source primitive, followed by explicit user confirmation. This link sets up a bi-directional data flow between the slot on the destination primitive, and one or more slots on the source primitive.

Interaction techniques specify how to modify a slot during user interaction while maintaining the constraint relationships of the other slots. For example, if a point's position is constrained to a line and the point is then manipulated, the constraint can be resolved either by moving the point along the line or by moving the line with the point. The interaction technique chosen is one of, or a combination of, these solutions. When a link is established, a new interaction technique is installed on the primitive to reflect this new constraint. The toolkit supports translational and rotational interaction techniques. The selection of an interaction technique could have been an implementation choice, but this would prevent the user from selecting the interaction technique best suited for their particular problem. We therefore leave the choice to the user.

3.2.1 Type Conversion

Each slot also has a *type*. Types are used in the system to identify different geometric attributes of a primitive, such as position, direction, or length. Since our slots represent variables that can be linked together, we must address the problems of linking incompatible types.

Type checking is performed when slots are linked together to see if the data types are compatible. If the types are compatible, then data is passed through unchanged. To allow the linking of incompatible types, the toolkit supports a set of *cast* operators which convert data of one type to a compatible type. This is similar to the cast function in the "C" programming language. The toolkit links source primitives to destination primitive slots. A cast operation takes a primitive as an argument and produces a typed value as a result. This flexibility allows the casting functions the use of one or more of the slots on the source primitive when converting the source data to the destination slot type. This is why our cast functions take a primitive instead of a specific slot as input.

Since the data flow between slots is bi-directional, we also define *uncast* operators which converts data from the destination slot of a link back to the type of the source primitive. For every cast operation there is an uncast operation. Although this cast and uncast solution has the disadvantage of requiring the specification of both functions for all possible combinations of data types, we have eliminated the need to compute inverses. Relationships in numerical solvers are specified by mathematical equations, not all of which can be inverted (i.e, they cannot necessarily be rewritten to solve for each variable in the equation). A typical solution in numerical solvers is to write further equations to represent the non-invertible cases. This is similar to specifying the uncast operation.

3.2.2 Constraint Resolution

We provide a detailed example to illustrate the linking process and how constraints are resolved. In this example, we link a vector primitive's position slot to a point primitive. We first select a vector primitive as the destination primitive, and then a point primitive as the source primitive. An arrow is drawn between the two primitives to indicate the pending link (shown in Figure 1).

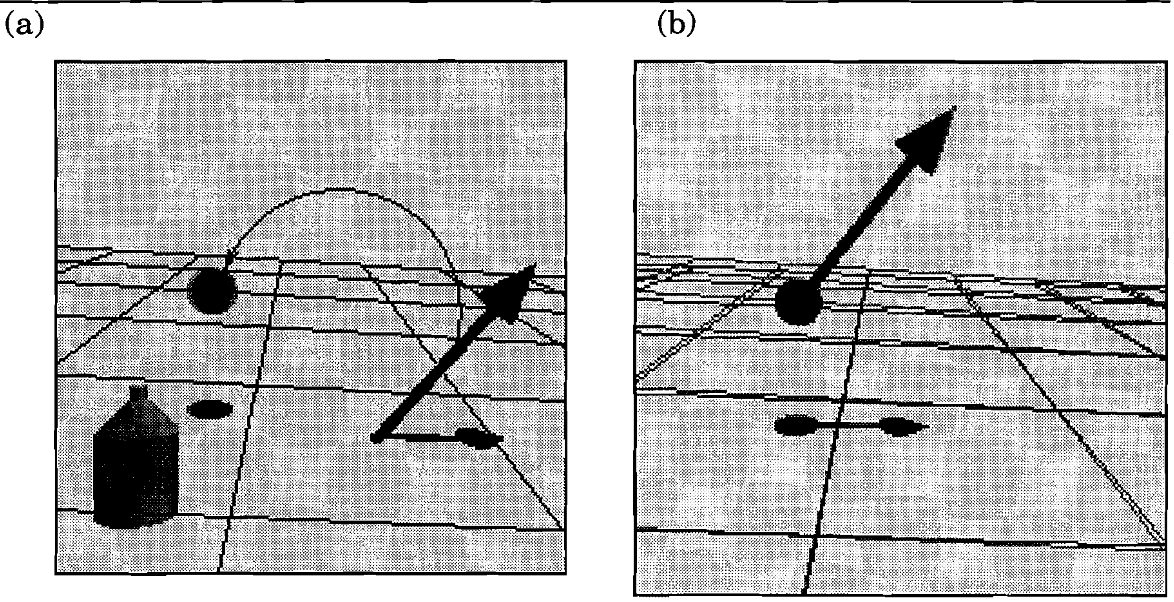
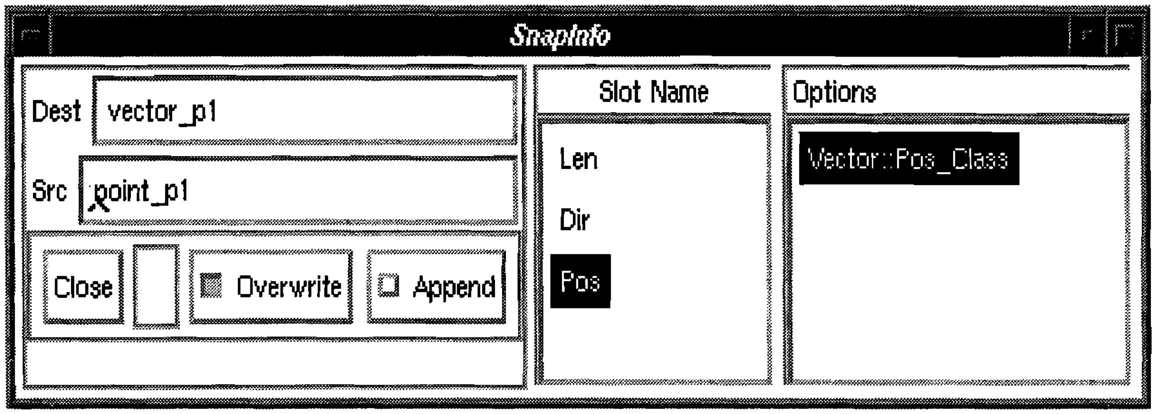


Figure 1 Linking Primitives (a) link pending, (b) link established.

We then select the slot of interest (in this case, the **Pos** slot) on the destination primitive through a MOTIF window (Figure 2). We are now presented with a list of possible options for establishing the constraint.


SnapInfo

Dest vector_p1

Src point_p1

Slot Name	Options
Len	Vector::Pos_Class
Dir	
Pos	

Figure 2 The Primitive's slots and the linking options.

These options combine the casting function, the interaction technique, and the methods for establishing the constraint. In this case we are presented with a single option which we select. Finally, we confirm the link by clicking on the glue bottle. The link is then established between the position slot on the vector primitive and the point primitive (as shown in Figure 3).

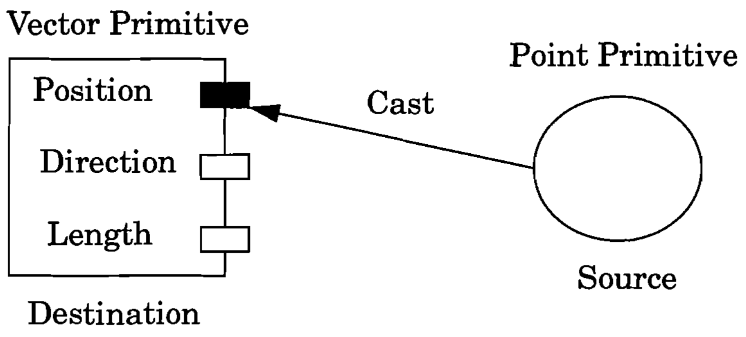


Figure 3 Data flow between the vector primitive and the point primitive

A link is created internally by storing a pointer to the source primitive in the linked slot of the destination primitive and by storing a pointer to the destination primitive in a list on the source primitive. When the user translates the point, the toolkit intercepts the mouse interaction and passes the mouse information to the point primitive's interaction technique. Since there are no constraints placed on the **Pos** slot of the point primitive, the interaction technique sets the value of the position slot to the new mouse location. The interaction technique then calls the point's *resolution method.* A resolution method is a method defined on each primitive that first resolves the slots of the primitive, then updates the graphical representation of the primitive, and finally calls the resolution methods of all of the primitives linked to it. If a slot is linked, then the slot is resolved by casting the primitive linked to the slot to the type of the slot being updated. If the slot is not linked, then its value remains unchanged.

In our example, the resolution method of the point primitive does not change the value of the position slot given by the point's interaction technique since the slot is not linked. The point is redrawn at its new position. Next, the point's resolution method calls the vector primitive's resolution method. When the vector primitive's resolution method resolves its **Pos** slot, it updates the slot by casting the point primitive to the type position, the type of the vector primitive's **Pos** slot. The return value of the cast is the value of the **Pos** slot of the point primitive (Figure 4).

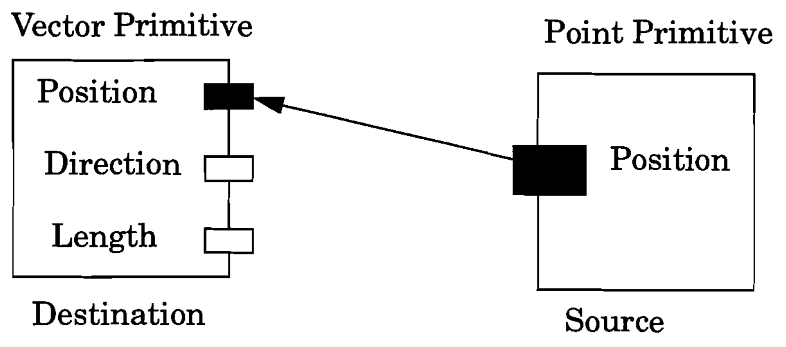


Figure 4 The Result of casting a point primitive to a position

Since the vector's **Dir** and **Len** slots are not linked they are not changed. The vector is now redrawn with its base at the position of the point primitive. A summary of the flow of control is depicted in Figure 5.

$$\begin{tikzcd}
\text{Mouse interaction with point} \arrow[d] \\
\text{Point's interaction technique} \arrow[d] \\
\text{Point's resolution method} \arrow[d] \\
\text{Vector's resolution method} \arrow[d] \\
\text{Resolution of vector's Pos slot}
\end{tikzcd}$$

Figure 5 Flow of control summary in constraint resolution

Now when the user translates the point, the base of the vector follows the point. Similarly, if the user attempts to translate the vector, the interaction technique of the vector allows the vector to freely translate and then assigns the new value of the vector's position to the point through an uncast function.

3.2.3 Dynamic Slot Creation

A link may not constrain all of the degrees of freedom of a slot. (The degrees of freedom of a slot are the ways, either translational or rotational, in which a constrained slot is free to vary.) These remaining degrees of freedom are represented by a dynamically created new slot on the primitive. For example, if we constrain a point primitive's position slot to a vector, then the point is projected onto the vector. The point interaction technique allows it to move along the line defined by the vector. A new slot, therefore, is created on the point primitive, called the ***T*** slot, which represents the parametric distance of the point along the vector. This slot is now constrainable and can be used to restrict the point's location along the line. The resolution method for the point resolves the **Pos** slot by first resolving the **T** slot, and then using the **T** value and the vector to cast into the type position, the type of the **Pos** slot. It should be noted that these dynamically created slots are defined on the primitives on a case-by-case basis.

3.3 Description of Toolkit Primitives

In this section, we describe the set of basic primitives defined in the toolkit. There is a fundamental difference between this toolkit's primitives and those in Brown's previous widget construction toolkit [18]. The difference stems from the definition of a primitive. The original toolkit had primitives consisting of multiple pieces of geometry and constraints between these pieces of geometry. In some ways, the name *primitive* was a misnomer. In our toolkit, all of the primitives have exactly one piece of geometry to represent them and all of the slots start out unconstrained. We designed our primitives to be simple, but powerful enough to build the more complex primitives in Brown's toolkit. This approach reduces the unnecessary and confusing clutter of complex primitives. The philosophy of this toolkit is based on the idea that more complex entities can be created from the simpler primitives using the encapsulation methods described in Section 5.0.

As in Brown's toolkit, we base the basic primitives on the Euclidean coordinate system metaphor (i.e., points, vectors, planes, and volumes). Previous experience has shown that this allows for the expression of a wide variety of constructions. It should be noted that this primitive set, although it has a fair amount of expressive power, is by no means adequate for all conceivable constructions. The toolkit was designed to allow for the easy addition of new primitives, or the extensions of existing primitives as the need for new behaviors arise.

The toolkit contains four basic primitives, the point primitive, the vector primitive, the plane primitive, and the graphical object primitive. Each primitive has an associated class which is an abstraction of the geometry and behavior (including interaction methods) of the primitive. Figure 6 shows the 4 basic primitives.

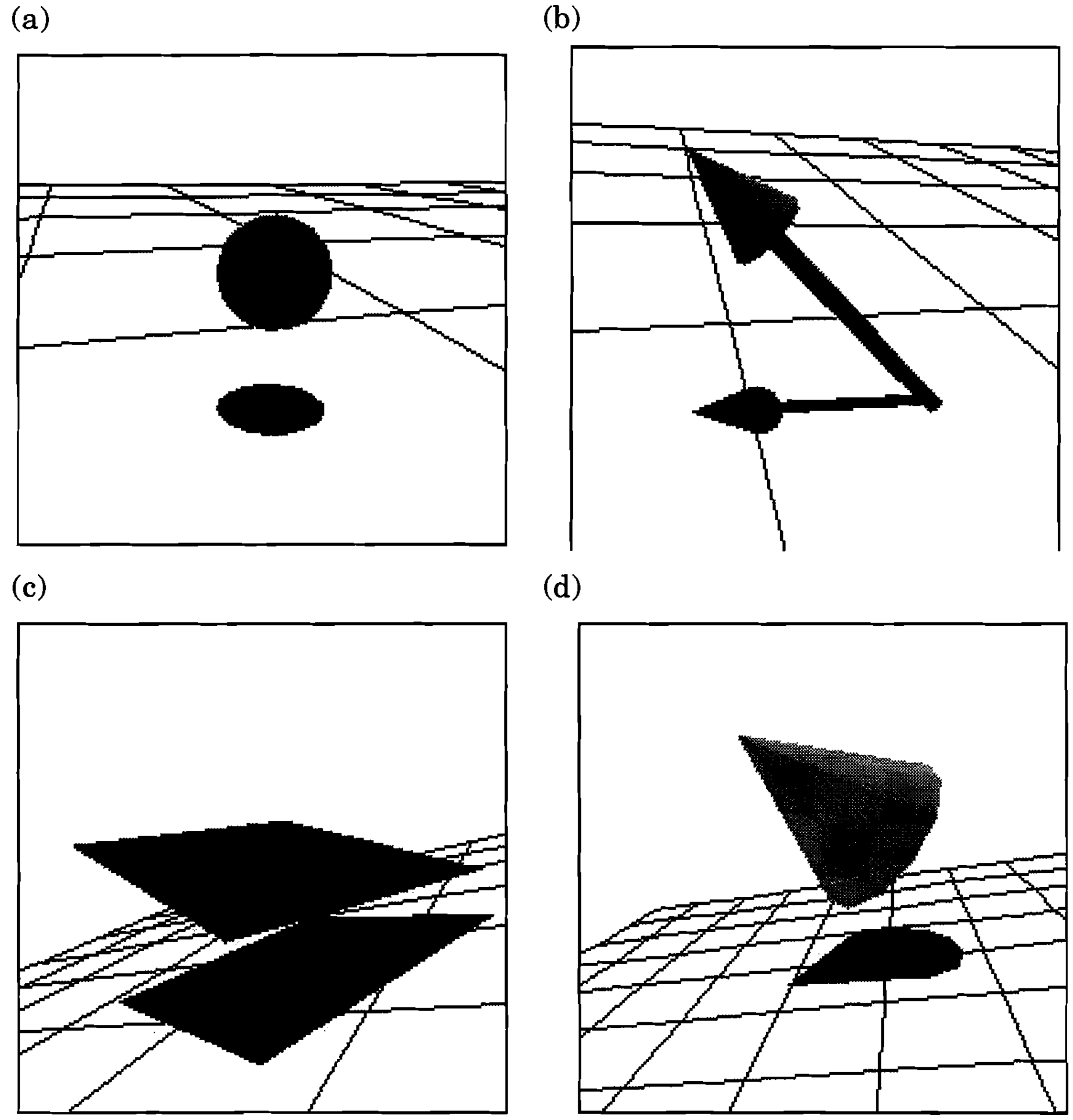


Figure 6 Toolkit primitive (a) point primitive, (b)vector primitive, (c) plane primitive, (d) geometric object primitive.

The point primitive, represented by a small sphere, has one constrainable slot, the **Pos** slot, which is of the type *position*. The **Pos** slot is an abstraction of a three space position. The default interaction technique on this primitive is the free translation of the point in 3-space.

The vector primitive, represented by a small arrow, has three slots, a **Dir** slot of type *direction*, a **Pos** slot of type *position*, and a **Len** slot of type *length*. The **Dir** slot represents the direction of the vector, the **Pos** slot represents where the base of the vector is in three space, and the **Len** slot represents the length of the vector. The default interaction technique of the vector primitive is the rotation of the vector, which sets the value of the **Dir** slot. By default the vector's **Pos** slot is located at the origin and the **Len** slot is set to 1.0.

The plane primitive, represented graphically by a flat sheet, has five slots, the **Normal** slot of type *direction*, the **Center** slot of type *position*, two **Size** slots for the length and width of the sheet both of type *length*, and the **Up** slot of type *direction*. The **Normal** slot represent the normal to the sheet, the **Size** slots represent the scale for the length and width of the sheet, and the **Up** slot represents the orientation of the sheet (similar to PHIGS VUP).

The final primitive, the graphical object primitive, encompasses all of the 3D modeled objects (e.g., cubes, spheres, CSG's) available in UGA, Brown's modeling and animation system [16]. This primitive is similar to the plane primitive in that it has slots that represent the local 3D coordinate system of the modeled object, i.e., **Normal, Up, Center**, and **Size** in three dimensions. We extend these slots to include other non-geometric attributes of the modeled object; for example, we have added **Red, Blue**, and **Green** slots of type *real* to represent the object's color.

3.4 Encapsulation

Encapsulation is the process by which networks of linked primitives are stored and recreated by the toolkit. The toolkit supports two types of encapsulation, *structural encapsulation* and *class encapsulation,* both of which can be optionally parametrized using a technique called *parametrized encapsulation*.

3.4.1 Structural Encapsulation

Structural encapsulation aids the user by reproducing a network of linked primitives. For example, we might link a vector to two points so that the vector spans the two points (as shown in Figure 7).

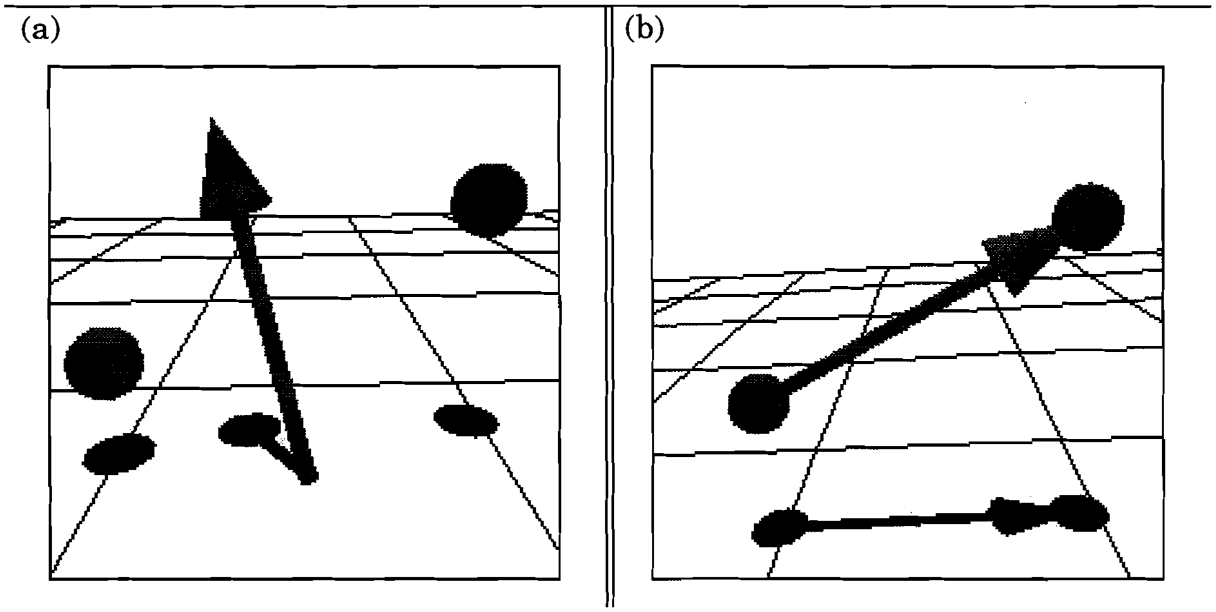
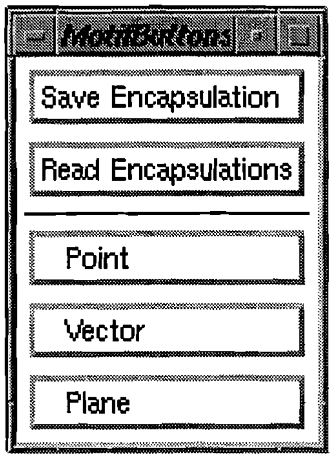


Figure 7 Vector primitive spanning two point primitives (a) before linking, (b) after linking

This is a construction we will use repeatedly, so we encapsulate it and name the encapsulation "line". The encapsulation process creates a new "line" menu item (as shown in Figure 8).

(a) 

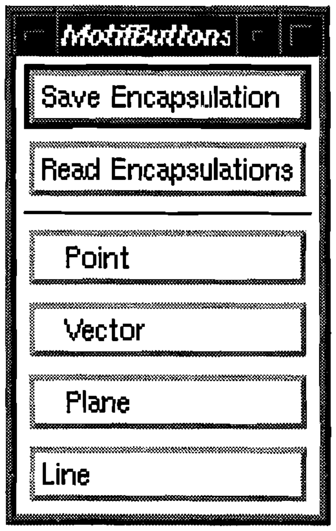
(b) 

Figure 8 Primitives menu (a) before "line" creation, (b) after "line" creation

A user who clicks on the "line" menu item gets a vector and two points with all the links of the original network that were encapsulated. The new network's behavior is identical to the original.

Structural encapsulation is implemented by traversing the constraint network of a construction and creating copies of all of the primitives and the links in the network. These copies are then stored by the toolkit but not drawn on the screen. When a structurally encapsulated object is requested, copies of all the primitives and links associated with network are created. These new primitives are then drawn.

3.4.2 Class Encapsulation

Class encapsulation is similar to structural encapsulation in that the constraint network is traversed and copies of all the primitives and links in the network are created. It differs in that we associate a new class with the collection of primitives in the encapsulated construction. We can think of this as creating a new primitive in the toolkit. Let's change our "line" example above to use class encapsulation. Now when we encapsulate the vector and points construction we also specify a class for the encapsulated construction, in this case the class "Line". The point and vector primitives' classes are changed to the class Line. This class modification changes the linking behavior of the primitives. When we link to a *structurally* encapsulated construction, we link directly to the primitive the user clicks on. For example, if we link a primitive to the vector primitive in the structurally encapsulated line, we are actually linking to the vector primitive, which is of the class vector. In *class* encapsulation, when we link to the vector primitive within the line, we actually link to the Line primitive. This allows new constraint behavior to be defined on the class encapsulated primitive. The new class encapsulated primitive behaves identically, during interaction, to a primitive created with structural encapsulation, but its behavior can be overridden as described below.

New classes are implemented by creating a new primitive which has slots for each primitive in the encapsulated network. These slots can optionally be named by the user when the object is encapsulated. In the line example, a

Line primitive is created with slots for the two points and the vector called, respectively, **start_point**, **end_point**, and **center_span** (see Figure 9).

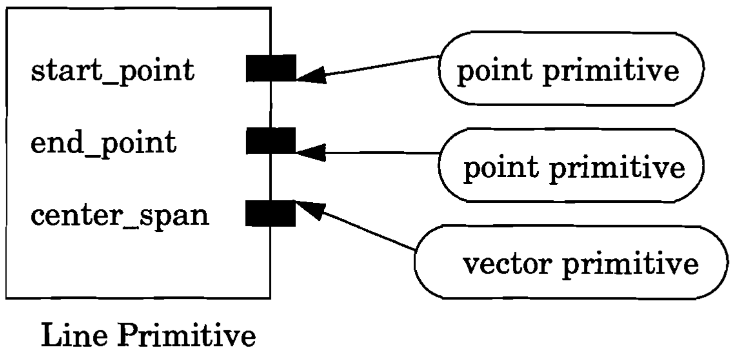


Figure 9 The slots on the line primitive created by class encapsulation

Assigning new classes to networks of primitives provides a mechanism for changing the interactive behavior associated with a network of primitives. For example, if we try to rotate an endpoint of the *structurally* encapsulated line, nothing happens because a point primitive has no rotational interaction technique. If we use *class* encapsulation, then when we refer to the point primitive, we are actually referring to the Line primitive. Perhaps we want the Line primitive to rotate around its center when we rotate an endpoint. Since the Line primitive knows about all of its components, we can install a rotational interaction technique on the Line primitive that rotates the line about its center, maintaining the constraints associated with the construction. Now a user who tries to rotate an endpoint of the Line gets the desired behavior.

Class encapsulation also helps us to address the problem of visually linking our interface objects to our application objects. For example, the rack widget

[14] is an interactive tool that allows the user to deform geometric objects using the deformation operations described in [2]. The interactive behavior of the rack is easily constructed with the toolkit. When the rack is applied to geometric objects, we must somehow create a link between the interface object, in this case the rack, and the geometric object we are deforming. In Brown's previous toolkit this operation was done through a clumsy additional primitive called a *black box* [18]. In our toolkit, we can use the normal toolkit mechanisms of linking primitives (as shown in Figure 10).

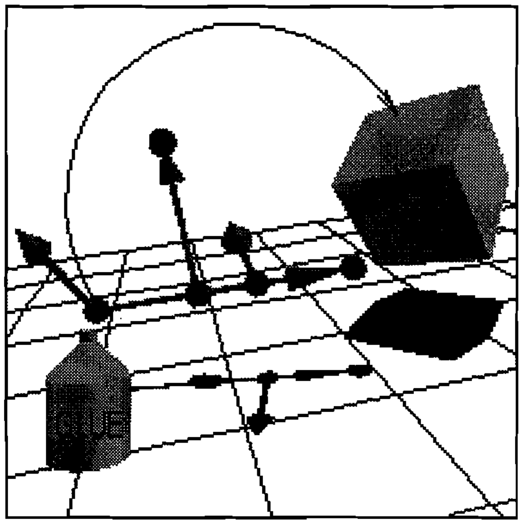


Figure 10 Linking the rack to a geometric object.

We interactively build the behavior of the rack with the toolkit and then use class encapsulation to create a new rack primitive. The toolkit can treat application objects as primitives if slots are defined on the objects. We can also define the type of the slots which allows us to define the behavior of the slot when it is linked to other primitives. In this example, we create a slot on our application object which we call the *deformation slot*. We also define a behavior for the deformation slot when a rack primitive is linked to it. The behavior passes the values from the rack primitive to the deformation operations defined on the application object. Now when we link a rack primitive to the deformation slot of the object and the rack is manipulated, the rack's resolution method calls the resolution method of the application object, which in turn deforms the object. When we are finished deforming the object, we can break the link (unlinking is described in section 5.4).

3.4.3 Parametrized Encapsulations

The ability to encapsulate and recreate constructions can be thought of as invoking a subroutine to create a complex primitive. We introduce the ability to parameterize these subroutines which we call *parametrized encapsulation*.

Often when we encapsulate a new class there are primitives in the network that we would like to keep as parameters. In our visual programming language, the primitives are the parameters to our subroutines. For example, if we create a line primitive as illustrated above, we might want to create it in such a way that we can create a line between any two points. In effect, we create a "line subroutine" that takes two endpoints as parameters. In our line subroutine, the endpoints can be any two points or any other primitives with the following restriction: the parameter passed must have a slot with the same slot type as the original primitive in the encapsulation. In this example, the parameters must each have a **Pos** slot.

To create a parametrized encapsulation, we specify the primitives which are to be parameters and then name them. When the class is created, the named primitives are marked as parameters. When users create a new primitive from a parametrized class, they are first prompted for primitives for the parameters. If no parameters are specified when the new primitive is created, then a duplicate of the primitive with which the encapsulation was originally created is used. One could say that the original parameters act as defaults for the parameters of the encapsulated object.

Parametrized encapsulations are created and stored in a manner similar to that described in structural and class encapsulation, but they differ in the method by which they are instantiated. As the toolkit begins to copy primitives and links from the stored copy of the primitive, it looks for primitives marked as parameters. When it encounters a primitive marked as a parameter, it prompts the user for the parameter. After the user specifies the parameter (or a default is provided), all link information from the primitive stored by the toolkit is copied into the parameter.

The ability to parametrize encapsulations greatly increases the usability of widgets like the interactive shadow widget [9]. The interactive shadow widget creates a shadow of an object by creating a scaled copy of the object and projecting it onto a shadow plane. These shadows are then used to interactively position objects. In Brown's toolkit, there was no way of creating a shadow widget on different objects without re-specifying all of the linking operations. In our toolkit, we might use a structurally encapsulated class, but this would create fixed primitives for the shadowed object and the shadow plane, which is of limited usefulness. By using parameterization we can specify that both the object to be shadowed and the shadow plane are parameters to the encapsulation. Now the user creates a shadow widget by specifying an object to be shadowed and a plane onto which to project the shadow. This makes it very easy for the user to create a shadow widget for any object in a scene.

3.5 Limits

There are many problems for which equality constraints are insufficient. As an example, consider a color picker made from three lines with points constrained to each of the lines. The three points slide up and down their respective lines, representing each point's distance from the start of its line as a fraction of the total line length. This fraction represents the percentage of red, blue, and green that are in the color of an object. In the standard RGB color model, color values are restricted to the interval between zero and one, but the toolkit allows the point to slide freely on the line. This produces values that can be greater than one or less than zero, which results in the color functions receiving invalid parameters.

There are two possible solutions to this problem. First, we could check the values being passed to the color functions and clamp them between zero and one. This solves the problem of passing invalid values to the color routines, but does not provide the user with any visual feedback on the valid color ranges. Second, we could impose limits on the constrained primitives.

We have taken the second approach in our visual language. We support comparison operators on slot variables, such as “greater than” and “less than”, through *limits*. Limits impose additional restrictions on the data that flows between slots in the toolkit.

Limits are imposed by associating limiting functions to slots. When a primitive’s resolve method is called, it in turn calls methods to resolve each slot on the primitive. After the slots have been resolved, the slots are then *limited*. The limiting methods checks the slot value to insure that it falls within a given range. If it does not, the value is adjusted to be within the desired range.

To continue with our color picker example, we can limit the point to lie on the line between the endpoints (as shown in Figure 11). This is accomplished by a limiting function which limits the value of the **T** slot of the point to the values between zero and the length of the vector. If the line segment is of unit length, then the color routines will receive valid values. An additional benefit to limits is that the point does not move past the ends of the line, providing the user with visual feedback on the valid point positions.

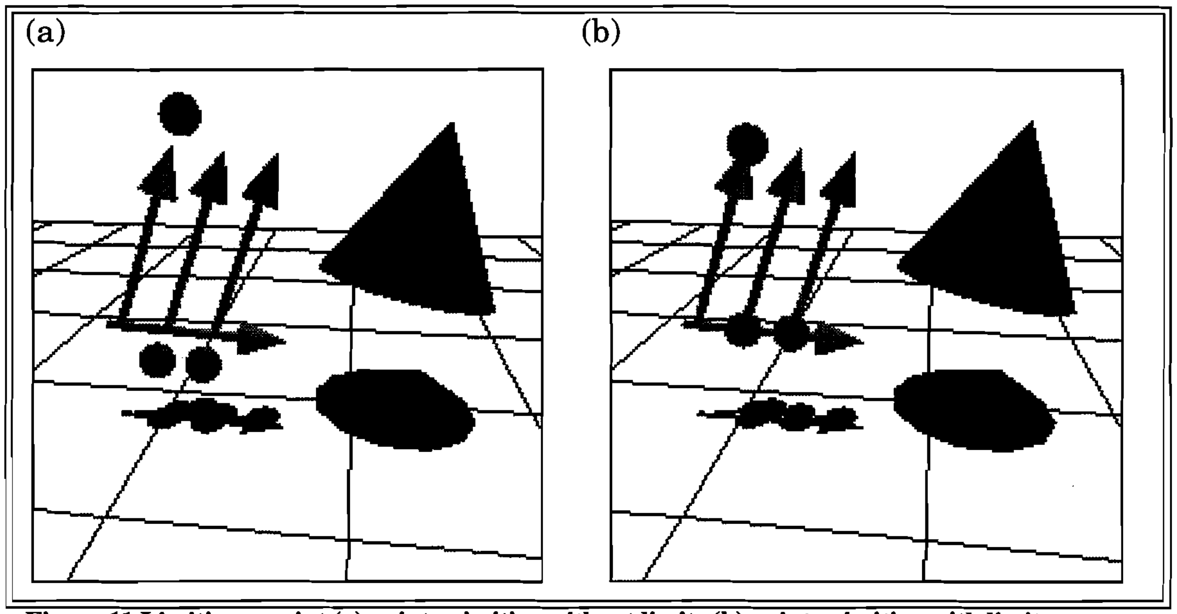


Figure 11 Limiting a point (a) point primitive without limits (b) point primitive with limits imposed.

With this ability to limit values we can enhance the representation of our color picker by making an RGB cube and limiting a point to lie within that cube, or making an HSV cone with a point limited to lie within the cone. This greatly extends our flexibility for interactive widget design.

4.0 Interface Design

User interfaces for specifying constraints [3][7][10][17][18] are often based on gestural pointing and clicking on the objects to be constrained. In the design of an interface for specifying geometric constraints the following problems must be addressed: What constraints can be applied to these objects? How can the constraints on these objects be visualized? What is the behavior of constrained objects under interaction? Is this behavior modifiable? Can we remove constraints?

In most of the current systems the interface for specifying constraints fails to address one or more of these issues. Usually the behavior is implicitly defined by the implementation of the constraint solver.

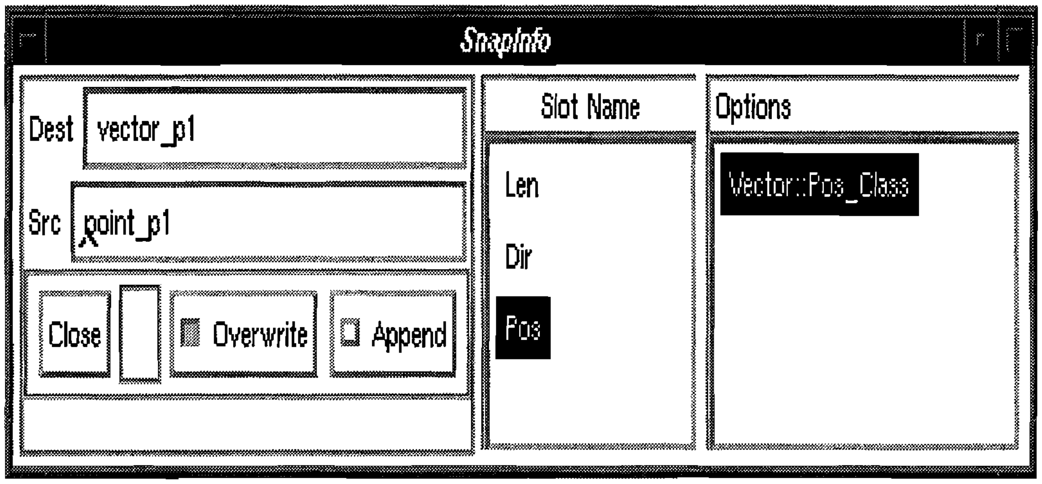
4.1 Linking Options

It is often desirable to know what the linking behavior will be when two objects are constrained. For example, if we link a vector to a point, does this implicitly mean the vector is now based at the point or that the vector's direction points in the direction of the point? There are many potential possibilities, all of which can be the "correct" behavior depending upon what the user is trying to do.

In commercial applications such as Intellidraw [17] on the Macintosh, the user is provided with hints in the form of icons that represent the constraint operations. Although this provides some help, the meaning of the icons is often difficult to decipher because the icons are too abstract. Additionally, when a user places multiple constraints on objects, the icon's meaning may become ambiguous.

In our toolkit, all the slots that represent a primitive are formally defined. For example, the point primitive is defined entirely by its position in three space. A more complex primitive, such as the vector primitive, is defined by a position, direction, and a length. These attributes, or slots, on primitives are what is presented to the user through the interface as constrainable quantities. This allows the user to choose which of the slots of a primitive are to be constrained.

The slot abstraction provides the user with a method for breaking a component down into its primitive behaviors. For example, the user who wants to constrain the location of a vector constrains the position slot of the vector. The user who wants to constrain the vector's direction uses the direction slot. In our toolkit, the slots are presented to the user in a Motif [12] window (see Figure 12).



Dest: vector_p1
Src: point_p1
Slot Name: Len, Dir, Pos
Options: Vector::Pos_Class

Figure 12 Motif Window for presenting the slots to the user.

The toolkit provides a default selection of slots to be used in the linking operation. For example, if we link a vector to a point we normally want to link their respective positions. A user may, however, want to link the vector's direction, not its position, to the point. In this case the defaults can be overridden.

Currently the slots have no visual representation on the primitive. A possible future enhancement is the addition of a visual representation for the slots. This would allow the user to directly wire slots together and reduce the amount of movement between windows during slot selection.

4.2 Interactive Behavior of Constrained Objects

When objects are constrained, the resulting interactive behavior is often ambiguous. For example, if a point is first constrained to a line and then to a plane, there are several interaction behaviors possible when the point is moved. The point could stay fixed, the point could move along the line and the plane move with it, the plane and the line could move with the point, or the point could move in the plane and the line move with the point. All of these behaviors are acceptable, but the correct choice depends upon the expectations of the user. In most systems, the behavior is chosen by the underlying implementation of the constraint solver.

The user should be able to choose from any of these options, and switch between them at will. In our toolkit, networks of primitives can be encapsulated into higher level classes which then have knowledge of the entire constraint network. Interaction techniques for these classes can be developed to exhibit behavior different from the behavior of the individual networked primitives while maintaining the constraint relationships. In the example given above we can build interaction techniques for each of the behaviors the user might want.

Currently these interaction techniques are developed by the toolkit author as new classes are created. It would be desirable in the future to automatically analyze the network and determine the degrees of freedom so that these interaction techniques could be automatically generated.

4.3 Visualization of Constraints

Once a network of constrained objects has been constructed, we would like to be able to examine the links in the network. In most constraint systems, the only way to accomplish this is to interactively move objects and deduce the constraints from their behavior. There are some systems such as Intellidraw in which a second window can be popped up to display the constraint network. This is helpful, but it still has the drawback of requiring a secondary window that is detached from the application objects. The user must still visually match the objects in the application window to the objects in the secondary window. Usually the entire constraint network is displayed, adding to the visual clutter on the screen and making it even more difficult to find the objects of interest.

A more effective approach is to move the visualization of the constraints back into the same scene as the application objects. We accomplish this by drawing arcs between primitives to show constraint relationships. The user specifies the primitive of interest by clicking upon it with the mouse (see Figure 13). Outgoing arcs are drawn from the primitive to the primitives to which it is constrained. Incoming arcs represent primitives constrained to the specified primitive. Incoming and outgoing arcs are colored differently for visual clarity. The information display could be enhanced by adding text to the arc which clarifies the constrained slot. The advantage of this approach is that the user can focus directly on the object of interest without needing a second window.

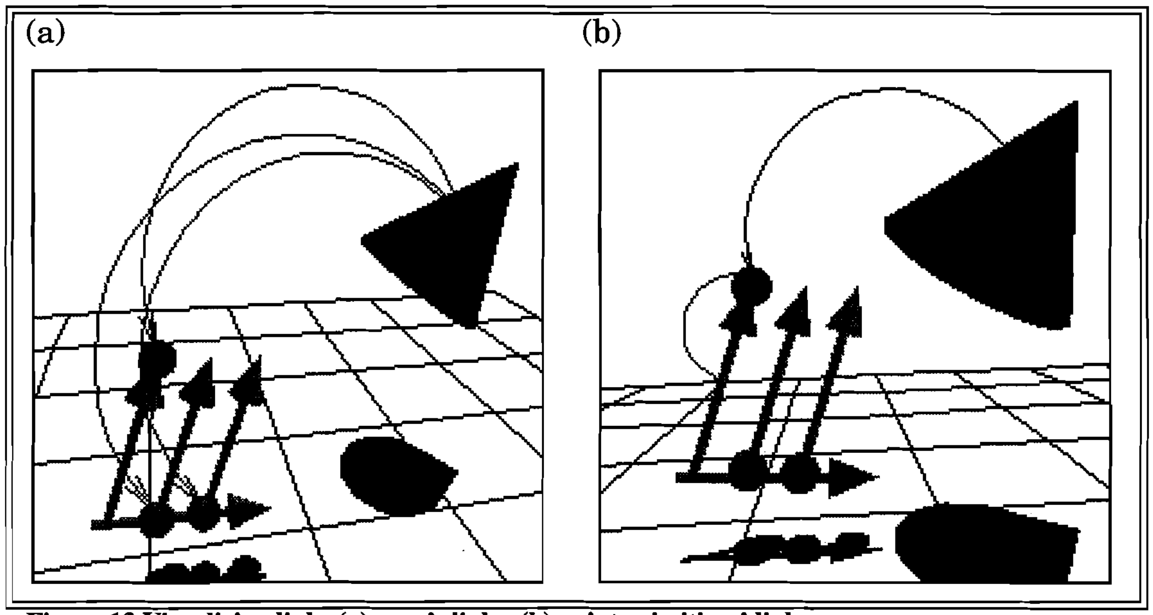


Figure 13 Visualizing links (a) cone's links, (b) point primitives' links.

One visualization issue we have not addressed is that of indicating how a constrained object may move. For example, when we constrain a point to a line and then try to move the point, does the point move along the line or does the line move with the point keeping its same relative location? At this time we do not know of a toolkit which is capable of visually representing what the constrained interactive behavior is, nor do we have any ideas for a good visual representation for this.

4.4 Unlinking

In Brown's toolkit, there is no mechanism for unlinking primitives. The widget designer is forced to restart after any mistake. This deficiency makes the toolkit very frustrating and difficult to use.

In a system designed to allow the user to rapidly prototype different ideas, the user must easily be able to add and remove constraints from the primitives. There are several useful mechanisms for removing constraints from a system. A history mechanism is useful for undoing several past actions, an arbitrary removal of constraints allows the user to pick a specific constraint of interest to remove from a primitive, and a removal of all of the constraints frees a primitive from all its constraint relations.

In our toolkit, we provide an unlinking history mechanism on a per primitive basis, as well as the other two unlinking methods listed above. With the history mechanism, each primitive keeps an ordered list of constraints involving itself as they are added. A global history mechanism is easily added by maintaining a list of primitive-slot constraint pairs. To implement the second undo mechanism, we note that primitives are constrained by slots, so it is easy to present the user with a list of constrained slots. For the final mechanism we can free all of the constrained slots on a primitive by removing all of the constraints on each slot.

4.4.1 Slot deletion

As discussed earlier, when slots are linked together, new slots might be created to represent the remaining degrees of freedom on the slot. Care must be taken when primitives are unlinked to remove any slots created by that link. For example, when we link a point's position to a vector, we create a **T** slot. If we unlink the position slot of the point, the **T** slot no longer has any meaning for the point and so is removed.

5.0 Implementation Details

The toolkit is implemented in Brown's modeling and animation system, UGA [16]. UGA supports an object oriented scripting language called FLESH. We use an object oriented approach to constraint solving which takes advantage of UGA's delegation based sharing and FLESH's multiple inheritance and dynamic parenting.

5.1 Object Inheritance

Here we describe the object model for the toolkit primitives (Figure 14). The *primitive object* inherits from a *geometric object* and a *behavioral object.* The *geometric object* is modeled in UGA and defines the primitive's geometry. The *behavioral object* defines the class of the primitive. The class defines the default slots, resolution and limiting methods, and interaction techniques for the primitive. Behavioral objects in turn inherit from a *constraint* object which defines any additional fields and methods needed to establish and solve the constraints.

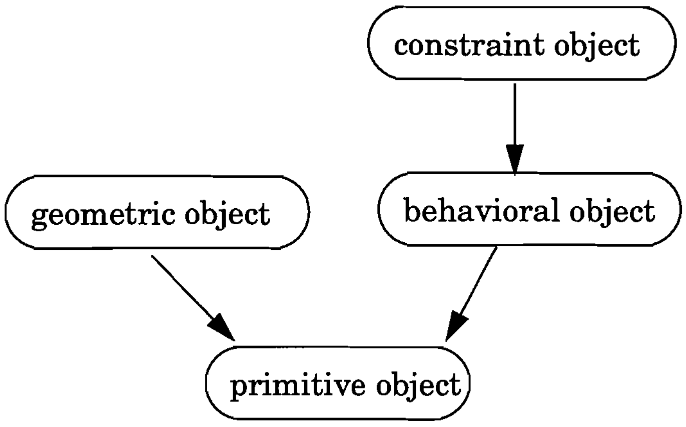


Figure 14 The toolkit primitive's object model.

5.2 Constraining Slots

A slot is constrained by the dynamic addition of a *slot constraint object* to the end of the parent list of a primitive (Figure 14).

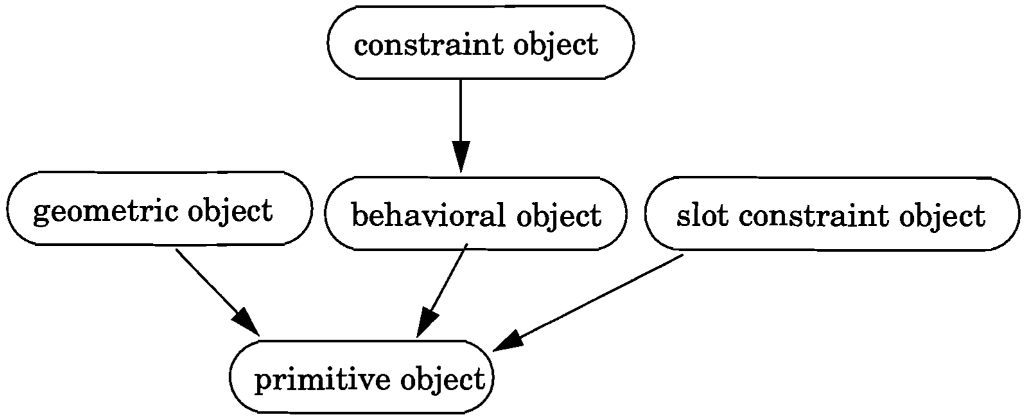


Figure 15 The object model after a constraint has been added.

The slot constraint object defines the new resolution and limiting methods for the slot, a method for establishing the constraint, and the new interaction techniques for the slot. These objects are presented to the user as options when primitives are linked (section 3.2.2). As other slots on a primitive are constrained, additional slot constraint objects are added to the parent list of the primitive. As slot constraint objects are added to a primitive, new slot resolution, limiting, and interaction methods occlude the previously defined methods. This is due to the multiple inheritance method lookup mechanism of UGA.

These slot constraint objects combine the resolution methods, limiting methods, and interaction techniques. This a convenient way to bundle the various objects of constraint behavior together. In the future, it may be more flexible to break these apart into separate objects called *slot resolution objects*, *slot limiting objects*, and *slot interaction objects*.

5.3 Interface details

The user interface is managed by both the *Snapper* and the *Solver* objects. The Snapper object sets up all of the mouse mappings and tracks the user's choices of source primitives and destination slots. After the user has selected a source primitive and a destination slot, the Snapper object passes this information on to the solver object. The Solver object finds the appropriate constraint object to enforce the constraint requested by the user, and adds this information to the source and destination primitives. It then calls the *establish* and *resolve* method for the constrained primitive.

5.4 Unlinking

Unlinking is implemented through the use of dynamic parenting by removing the necessary constraint object from the parent list of the constrained primitive. An unlink method is also called before the object is removed. This method removes any slots created by the addition of this constraint.

5.5 Adding new constraint behavior

This section describes how to add new constraint behavior to a toolkit primitive. The addition of a new behavior to the toolkit involves creating a new slot constraint object (slot constraint objects are described in section 5.2.). Slot constraint objects have the following naming convention: "**class**::**slot**[::::**class_extension**]_Class". Everything in brackets is optional. The first **class** is the class of the primitive we are developing behavior for. The **slot** is the slot on the primitive which we are constraining. Optionally, we can specify a second class. This **class** is the class of the primitive the slot is constrained to. This optional second class is used to develop a behavior for a link in a specific source-destination class combination. If a second **class** is explicitly specified, the resolve method for the slot can use this information to create behavior for a specific type of link. Without this second **class** the resolve method for the slot must call a cast function to convert the data since it can accept any class. The **extension** field is used to describe the behavior of the class. For example, we may want to develop two different behaviors when linking a point's **Pos** slot to a vector (Point::Pos::::Vector_Class). This would be ambiguous, so we use the extension field to uniquely identify the behaviors (Point::Pos::::Vector_XXX_Class and Point::Pos::::Vector_YYY_Class).

There are three types of methods which can be defined on the slot constraint object. The first method is a resolution for the slot named **Resolve*Slot***, where ***Slot*** identifies the slot being resolved. This method updates the value of the slot. An example of this type of method is the **ResolvePos** method of the **Point::Pos_Class**. The **ResolvePos** method calls a casting function which takes the object the slot is constrained to and returns a value of type *position* (the type of the **Pos** slot), and then assigns it to the **Pos** slot.

The second type of method is the **Limit*Slot*** method, where ***Slot*** identifies the name of the slot. This method is called after the **Resolve*Slot*** method and is used to restrict the value of a slot to a certain range. In our example in Section #3.5, we limited a point's position to lie between the endpoints of the line. We accomplished this by writing a **LimitT** method which limits the value of the **T** slot of the point primitive. The **T** slot in turn is used to determine where the point lies on the line.

The final method is the interaction technique for the slot; these methods are called the **TransMouse** and the **RotMouse** methods. **TransMouse** and **RotMouse** both take the mouse information and update the slot maintaining the constraint on the slot. For example, the **TransMouse** method on the **Point::Pos::::Vector_Class** constraint object computes a new value for the **T** slot for the point based on the closest point on the line to the mouse. The interaction technique then calls the resolve method of the primitive. The resolve method of the point resolves the **Pos** slot by using the vector and the new **T** value assigned in the interaction technique. The point is then updated to its new position on the line.

5.6 Adding new primitives

To add a new primitive to the toolkit, an object must be created to represent the primitive. This object must adhere to the object model described above in Section 5.1. This means we must create a geometric object and a behavioral object for the primitive. The geometric object can be any arbitrary piece of geometry created in UGA. The behavioral object must inherit from the constraint object, define the slots and their types for the primitive, and give a class to the primitive. If these slots have new slot types, then new casting functions must be written from each class in the toolkit to the new slot type. We must also create cast functions from the new primitive's class back to each existing slot type in the toolkit. We can then start writing slot constraint objects for the primitive (described above in section 5.5) to define its constrained behavior.

6.0 Conclusions

Our toolkit provides a framework for the construction of a wide variety of interactive 3D models used in modeling, animation, and scientific visualization. The visual language provides an environment for both non-programmers and programmers to conceive and rapidly prototype these 3D models. Encapsulation and limiting mechanisms allow interface designer to build more complex models than were possible in the past. The visual interface also provides a method for linking and unlinking interface object to application objects. Constraints are specified in an interface that is understandable and easy to use for both non-programmers and programmers.

7.0 Future Work

Currently, when the user requests an encapsulation the entire network is encapsulated. A user may wish to encapsulate a subcomponent of a complex network. An interface for specifying the encapsulation of a sub component of the network needs to be developed.

At this point we have no method for the visualizing how a constrained primitive can be interacted with. For example, when a point is constrained to lie on a line there is no visual feedback to indicate that the point can only translate in the direction of the line or that the point cannot rotate. Possibilities for this visualization are arrows indicating in which direction the primitive may move, or altering the geometric shape of a primitive when it is constrained (e.g., changing the point's sphere to a flat disk when the point is constrained to lie in a plane).

To date, the slots on a primitive can only be linked to a single primitive. It would be useful to be able to constrain a slot on one primitive to a slot on another primitive or a slot on a primitive to more than one primitive.

In our toolkit constraints are solved procedurally. Although this has advantages, such as the elimination of problems related to numerical error that numerical solvers suffer, its main disadvantage is that it leads to a combinatorial explosion in the number of procedures as we add more primitives and types to the toolkit. To date we have not been able to find a numerical solver that is stable or interactive enough to use for our toolkit. If one is found in the future, the current procedural method for solving constraints should be changed.

All of our primitives constrain geometric attributes of objects. Primitives that represent abstract non-geometric quantities could be useful. For example, temporal primitives might be useful in specifying how primitives change over time, allowing us to build animations visually.

Slots on primitives are presented to the user via a Motif window. Slots should be visual quantities on the primitive (perhaps depicted as a little socket). The user could then wire the slots together directly instead of going through the auxiliary window.

8.0 Acknowledgments

I would like to thank my advisor John Hughes for all his comments and suggestions during this project, Andries van Dam and the Brown Computer Graphics group for providing such an incredible environment to learn and grow, a special thanks to Robert Zeleznik for the discussions we had during the design phase of this project, Nate Huang who was always willing to help whenever I had any problem at all, to Cindy Grimm for anytime I had a tough math question and for helping review this document, and finally to my family who have been so supportive throughout this experience.

9.0 Bibliography

[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, and information workspace. In *Proceeding* 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. Graphics 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.

[17] Aldus Inc. *IntelliDraw.* User Manual v. 1.0, 1992.

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

Interactive 3D Toolkit for Constructing 3D Widgets. *Computer Graphics* (SIGGRAPH '93 *Proceedings*), 27(4):81-84, July 1993.