A Graphics Toolkit Based on Differential Constraints

Michael Gleicher
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
gleicher@cs.cmu.edu

ABSTRACT

This paper describes *Bramble*, a toolkit for constructing graphical editing applications. The primary focus of Bramble is improve support for graphical manipulation by employing differential constraint techniques. A constraint engine capable of managing non-linear equations maps interactive controls and constraints to object parameters. This allows objects to provide mathematical outputs that are easily composed, rather than exposing their internal structure or requiring special purpose interaction techniques. The model of interaction used with the differential approach has a continuous notion of time, which provides the continuous motion required for graphical manipulation. Bramble provides a LISP-like extension language and support for other application features such as windows and buttons. The paper concludes with examples of interaction techniques defined in Bramble and applications built with Bramble.

KEYWORDS: toolkits, graphical editors, constraints, interaction techniques

1. INTRODUCTION

The range of graphical editing applications continues to expand as hardware to support them becomes more widely available. In addition to common applications such as drawing, others such as architectural layout, scene composition, solid modeling, simulation and animation will also become available to users. As graphical applications become more available, they will need better interfaces. Advances in computer hardware offer opportunities for such improvements. For example, advanced graphics systems allow realistic images to be generated fast enough for use in interactive settings. Similarly, the numerical performance of new processors make the use of constraint techniques practical.

Constraints make a powerful addition to the interaction techniques available in graphical editors. They can also impact how interactive systems are constructed and interaction techniques developed. However when adding constraints to systems, the essential character of direct manipulation editing must be preserved. Systems must be smooth, fast, responsive and reactive.

This paper describes *Bramble,* a toolkit designed to support the development of graphical editing applications, with features such as geometric constraints, snap-dragging, differential controls, and animation. Bramble is designed to explore how numerical constraint techniques can effect the construction of graphical editors and the development of interaction techniques.

Figure 1 shows *MechToy,* a Bramble application. MechToy allows the user to sketch planar mechanisms and experiment with their behavior. Its direct manipulation interface is similar to that of a drawing program; however the graphical objects are designed to look like the symbols that appear in mechanics textbooks. The interface infers connections between pieces as they are sketched. Bramble's non-linear constraint techniques permit the user to play with and animate the gadgets once they are drawn.

Appeared in: Proceedings of the 1993 ACM Symposium on User Interface Technology (UIST '93), pages 109-120.

1.1. The Differential Approach

The core of graphical editing applications is the manipulation of graphical objects. The desired interface for this is typically direct manipulation: interactions where objects move with continuous motion that is coupled to the

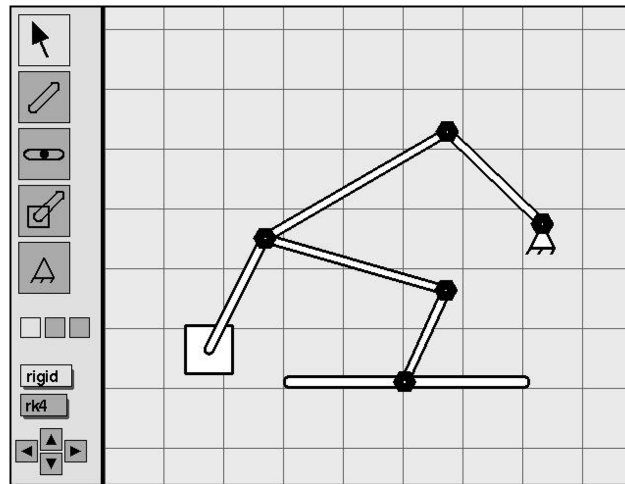

Figure 1: The *MechToy* application permits user to sketch a planar mechanism and experiment with its behavior.

user's actions, such as dragging. Bramble's focus is on supporting this graphical manipulation, although it does provide support for other parts of applications such as display lists, views, hierarchies, buttons, and windows. To support graphical manipulation, Bramble take a a *differential approach,* in which constraint techniques are used to support direct manipulation.

Graphical objects have many real-valued *aspects* in which a user may be interested. Even a simple line segment has a range of aspects which a user may want to specify, such as the positions of the endpoints or midpoint, or its length or orientation. Users may also be interested in aspects of groups of objects, for example the distance between the centers of two circles. *Controls* are the aspects which the user directly specifies or manipulates.

The configurations of each graphical object is represented by a vector of real-valued parameters called its *state vector.* There is typically a choice in how objects are represented. For example, we might represent the configuration of a line segment by the positions of its endpoints, by the location of its center, its length, and its angle. The choice of representation is an implementation concern, but ideally should make no difference to the user.

The basic idea of the differential approach is that objects are manipulated by constraining and controlling their aspects. Objects provide a variety of aspects as potential controls. The user can choose which to specify, either by constraining their values or by interactively dragging them. These constraints and controls can be combined arbitrarily. The differential approach gives the user more flexible and task specific controls, which can be mixed and matched as convenient for their problems.

The constraint engine used in the differential approach maps the constraints and controls into changes in the objects' parameters. The use of the engine allows all access to the objects' configuration to be accomplished via their aspects. Because the interface is in terms of aspects, rather than representation, programmers have the freedom to select object representations based on implementation concerns, yet still provide desired interfaces. System pieces are designed in terms of generic aspect types rather than specific objects.

Simply permitting the user to specify values for aspects is not a practical basis for interaction. Instead, we prefer that aspects are controlled so that objects move with continuous motion. This appears to be important for usability concerns[11]. It also permits implementations using simpler numerical techniques[14]. Therefore the differential approach takes a continuous view of time, unlike most approaches to graphical interaction. Discrete events alter the way objects move, rather than directly altering their configurations. Interaction techniques are defined by applying constraints and controls to aspects at appropriate instants, and permitting these actions to operate on objects over a duration of time.

Benefits of the differential approach are demonstrated by the example shown in Figure 2. Consider the task of aiming the light from the Luxo lamp. The user of a traditional 3D system would do this by adjusting the joint angles of the lamp. However, an application built with Bramble permits the user to directly manipulate the light direction, for example by dragging the place on the ground where the light points. As the user drags this control, Bramble's differential constraint engine adjusts the lamp's joints accordingly. The floor target is just one of the many aspects

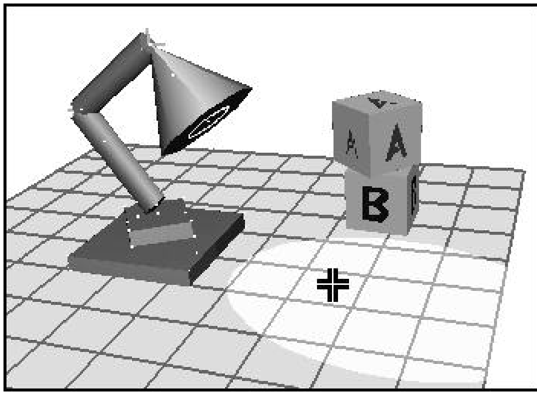

Figure 2: A lamp is manipulated by dragging the point at the center of where its light hits the floor. The joint angles are adjusted based on the manipulation of the light's target.

of a directional light source which can be used for manipulation. The developer of the light source object need only provide aspects, as Bramble's constraints and control techniques connect to these. In fact, since aspects can be defined in terms of other aspects, the directional light controls were defined in terms of the bulb's position and tip, so the light can be manipulated in any fixture in which it is placed.

The Bramble toolkit was built to explore applying the differential approach to interactive graphical applications. This paper examines this approach, and describes some relevant aspects of the toolkit. Following a brief review of related work, the differential approach will be introduced in more detail, and its notion of time will be discussed. Pieces of Bramble will be described, including its embedded interpreter and standard set of graphical objects. The paper concludes with examples of how interaction techniques can be defined with Bramble's differential approach and of applications built with Bramble.

2. RELATED WORK

Since the earliest interactive graphical editors, systems have attempted to provide sophisticated editing features. Sketchpad[33] introduced not only direct graphical interaction, but also constraints and snapping. Since this ground-breaking application, direct graphical interactions have been continually refined and have become commonplace in systems. Although constraint techniques have been limited to research systems such as [27], there is now a renewed interest in combining constraints with direct manipulation interfaces[1, 2, 9, 11, 23].

Several toolkits have been developed to support the development of graphical editors at a higher level than low level graphics toolkits such as GL[21] or PHIGS[8]. Tools such as Garnet[26], Coral[34], Unidraw[35], ArtKit[16] and Inventor[32] provide services for graphical applications. Like Bramble, such tools allow developers to work with abstractions which correspond directly with components of editors, such as graphical objects and views. Several of these toolkits contain support for advanced interface techniques, such as ArtKit's snapping or Inventor's 3D manipulators. However, unlike Bramble, none of these toolkits provide non-linear constraints or support for both 2D and 3D applications.

Previous toolkits have attempted to aid in the development of interaction techniques, and their incorporation into systems. For example, Garnet provides a basic set of interactors[25] from which more complex behaviors can be constructed. Inventor's manipulators[32] provide a method for coupling predefined interaction techniques with graphical objects. UGA[10] allows prototyping 3D interaction techniques procedurally. Like Bramble, GITS [28] defines interaction techniques with constraints; however it is limited to the design of 2D widgets and it precompiles constraint solutions. In [37], interaction techniques are interactively linked together in a constraint-like fashion to build more complex 3D widgets.

Graphics toolkits such as Garnet, Coral, MEL[18], and ThingLab II[24] have employed constraints to aid programmers, for example by maintaining consistency between program data structures. Bramble, in contrast, targets constraints as a user service. Therefore, the type of constraints Bramble provides is quite different than that of these other toolkits. Bramble provides support for general non-linear equations, whereas other toolkits have employed propagation solvers which solve restricted classes of equations, but with potentially better scalability and performance characteristics. The differential constraint methods used in Bramble are discussed in [13] and [14].

Providing an embedded interpreter in an interactive application is not an uncommon technique. Well known examples of such systems are Gnu-Emacs and AutoCAD, which provide LISP interpreters for extensions. Graphics toolkits which center around such interpreters include Tk[30] and ezd[3]. Like Tcl[29], the *Whisper* interpreter used in Bramble is designed as a language specifically to be embedded in interactive systems.

3. USING THE DIFFERENTIAL APPROACH

The differential approach aims to provide more flexible methods for the manipulation of graphical objects by per-

mitting constraints and controls on aspects of objects, rather than just directly on their parameters, and permitting these constraints and controls to be combined.

Aspects are determined by functions of the state vector. Sometimes these functions are defined directly, but often they are composed of other aspect functions. This composition affords an important modularity by allowing constraints and interaction techniques to be defined in terms standardized connectors, rather than individual objects. For example, graphical objects can provide the positions of points as outputs without knowing what will be connected, and interaction techniques can be defined in terms of point positions, without knowing what types of objects these points come from. An example is illustrated in Figure 3.

3.1. Interaction Via Constraint Switching

The differential approach permits users to control how combinations of aspects evolve over time. Interactions are defined by altering the set of controlled aspects, then letting this control alter the objects over a period of time. Two basic actions are used to describe the rate of change in the aspect. Aspects can either be driven towards a particular value, or forced to follow a moving target. These basic *differential interactors* serve as building blocks for interaction techniques. The *follow* differential interactor is used to tie an aspect to an input device or animated path. The *go–towards* interactor is used to set an aspect to a desired value and constrain it to stay there once it achieves that value.

The configurations of graphical objects are changed by the effects of the differential interactors over time, as discussed in section 3.3. At discrete instants, the set of acting differential interactors is altered, but not the values in the state vectors. Interactions are created by connecting differential interactors to aspects in response to particular cues, and permitting time to advance so that the interactors can affect the variables. For example, dragging is accomplished by creating a *follow* differential interactor between a point position aspect and the mouse when a button is pressed, and removing this interactor upon release. More examples are provided in section 8.

More complex differential interactors are created by rules which switch a basic interactor on and off as needed. For example, an inequality constraint on an aspect can be created by turning on a basic interactor which forces the aspect into the allowed region when the boundary is exceeded. Another compound interactor turns on a constraint which forces an aspect to a precise value when it is close, but releases the constraint when another interactor opposes it enough. This snapping interactor gives the feel of gravity seen in Snap-Dragging[5].

The set of differential interactors is small and somewhat fixed. The set used in Bramble is *go–towards, follow, bound, snap,* and *click.* The interactors may be attached to any aspect. The range of interaction techniques results not from extending this set, but rather from the aspects to which they are applied and how they are switched on and off.

3.2. Aspects and Snap-Together Math

Aspects are the subobjects which Bramble's graphical objects provide for other objects to refer to. They provide the numerical values of some feature of an object. The two types of connections to these ports are differential interactors and other aspect–producing objects. Aspects, therefore, must provide sufficient means for solving differential constraints and for composing other aspects. Methods providing the values and derivatives with respect to the state variables are sufficient.

Bramble is built with a mathematical toolkit called *Snap-Together Mathematics*[15] which permits the dynamic definition of mathematical functions and rapid evaluation of these functions and their derivatives. Aspects correspond exactly to the functional outputs of Snap-Together Math. Functions are built by composing other functions together. At a low level, one might consider building functions out of basic mathematical primitives such as addition; however, composition occurs at higher levels of abstraction when aspects are defined as functions of other aspects.

Briefly, Snap-Together Math permits functional elements to be connected together to form larger functions, effectively wiring the blocks into an expression graph. Individual functional elements need only compute their local functions and derivatives, and the global values are computed by a composition process which traverses the graph. Functions for local derivatives can be generated automatically by code generating tools. These smaller derivative matrices are assembled into the larger results by a process called automatic differentiation[22]. For performance, Snap-Together Math exploits sparsity in the derivatives and employs caching.

Snap-Together Math also provides support for managing sets of state variables. This is important since principles of object-oriented encapsulation suggest distributing the state among the objects to which it belongs, while the numerical algorithms require the state to be in a single vector. Snap-Together Math addresses this dichotomy by permitting objects to maintain their own local state vectors, but

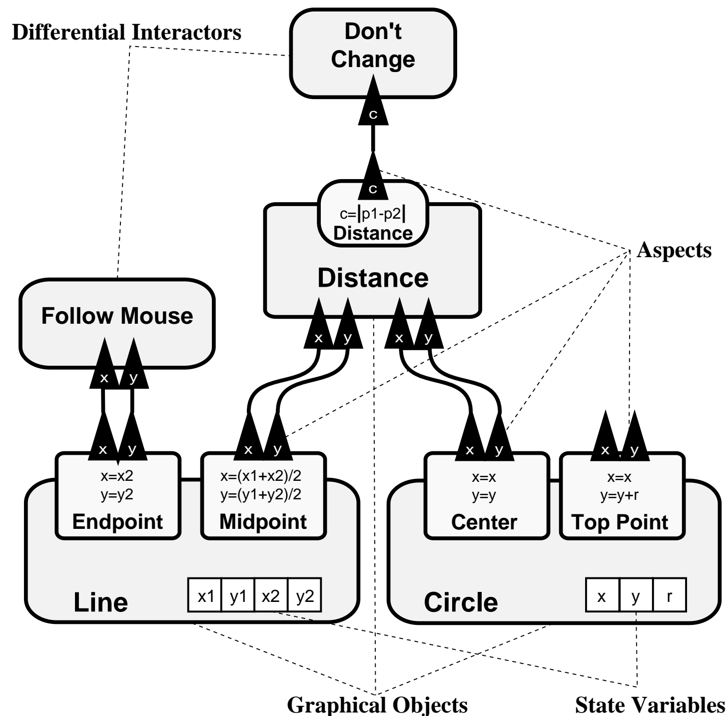

Figure 3: Aspects permit defining constraints independently of the objects they relate. In the example, the distance constraint graphical object is defined in terms of generic 2D points. The differential interactors are similarly defined to connect to a variety of aspects.

gathers these variables into a global vector for computations. Selectively gathering variables permits limiting the scope of computations for performance[12].

It is important to emphasize that aspects are the interface for how objects are controlled. Differential interactors and aspects only need to inquire the value and derivatives of aspects that they connect to. This is very important from the standpoint of modularity. Objects can provide aspects of various types, without concern for what will be connected to them. Interaction techniques and compound aspects can similarly be defined in generic terms. For example, a dragging interaction or distance constraint can be defined in terms of the positions of points on objects and can be connected to any aspect of any object that reports a position. Connections for such an arrangement are depicted in figure 3.

3.3. Bramble's Differential Engine

The Snap-Together Math library provides a differential constraint engine[1] which is used by Bramble. The engine keeps a list of active differential interactors and their associated aspects. It determines how the state variables should be altered to achieve the effects specified by the interactors. All of the variable dependencies are determined via the derivatives of the aspects. The mathematics for such calculations are described in [13] and [14].

The differential interactors specify how values should be changing at a given moment, their time derivatives. This is integrated over time by an ordinary differential equation (ODE) solver. Snap-Together Math makes a variety of solvers available.

Bramble uses a single differential engine for controlling all objects in the system. Viewing parameters, lighting, object configurations, material surface properties, and animation motion paths are all treated uniformly. The engine merely sees mathematical functions of state variables, without regard for what they do.

To Bramble, the differential solving engine is a black box. The developer of an application need not understand the mathematics inside it. In fact, application developers rarely need to concern themselves with derivatives. In cases where new functions need to be defined to compute new types of object aspects, the derivatives can be generated automatically.

[1] We dislike using the term solver, as the engine does not simply solve systems of non-linear equations.

4. BRAMBLE'S NOTION OF TIME

Bramble differs from most other interactive graphical toolkits in that it has a continuous notion of time. Just like physical objects in the real world, graphical objects in Bramble exist continuously and move with continuous motion. Of course, digital computers can only approximate continuous things. Bramble uses machine precision floating point to represent continuous values, and uses ODE solving techniques to discretize time. Such solvers work by computing steps which move time forward in small increments. The rich and varied literature on ODE solving (see [31] for a practical introduction) provides insight on the effects of time discretization.

Bramble's main loop consists of executing a number of solver steps to advance time forward some amount, and then processing any discrete events which have occurred during the step. Each of these iterations is called a *tick*. Since graphical objects exist continuously, there is no notion of redrawing objects when they are altered. Views always display the current state of the model; however, they are only updated at the end of each tick. Previous systems, such as Coral[34] have used constraint mechanisms to achieve this same effect.

4.1. Event Model

Reactive systems, like the graphical editors that Bramble is designed to support, are typically described and built in an event driven fashion. With such an approach, systems are designed by specifying what happens in response to various stimuli (events) such as input device clicks. All effects in the system happen in response to events. The event driven model has been extended to support direct manipulation. Events are generated for each small bit of input device motion or clock tick. At each of these events, the positions of dragged objects can be updated. This can be extended further to support snapping and gestures[16].

Rather than augmenting the event model to support continuous operations, Bramble instead uses events only for discrete actions. Discrete events do not affect the configurations of objects. Instead, discrete events only create and destroy objects and differential interactors. Motion occurs as the interactors affect the state over time.

When an event, such as a button click, occurs, Bramble queues it, but does not process the queued events until the end of the tick. Normally this is not a problem, as this time lag is very short. In order to maintain the illusion of continuous motion, Bramble must process many ticks per second. When this rate slows down too much, we experience interactive breakdown. Below approximately 4 ticks per second, not only does the illusion of continuous motion break, but also the lags between events and when they are processed grow noticeable.

In Bramble, cursor snapping is handled separately from the event process. The model for cursor snapping is also continuous, but in practice it is only updated at the end of each tick. Event actions can inquire about the state of snapping at any time. Facilities are provided to control the scope of snapping, permitting techniques such as semantic snapping[20]. Presently, the Bramble snap server is limited in the types of targets it permits, but it is being extended to support the range of snapping seen in Snap-Dragging[4, 5] in a general way.

5. GRAPHICAL OBJECTS IN BRAMBLE

Bramble uses the drawing model provided by the GL library on the Silicon Graphics Iris workstation[21]. As with GL, all drawing is actually done in 3D, although many objects ignore the third dimension. This has a small cost, for example in doing transformations. The benefits include avoiding redundant code and the ability to place 2D objects in 3D worlds.

The base class for graphical objects in Bramble is the class `Drawable`.This class only requires a few methods. Subclasses of `Drawable` define a draw method and a bounding volume method, both of which need only operate in the object's local coordinate frame. Objects may also provide other functionality. For example, most objects will define at least a few aspects so they can be manipulated, and will allocate a state vector to store their configuration. Other examples of optional `Drawable` functionality include ray intersection, creation of differential interactors, and generation of external representations.

Bramble supports object hierarchies with its `Group` class. A `Group` is a subclass of `Drawable` which contains a list of other objects and a transformation to apply to them. Like the hardware and graphics library it has been built on, Bramble presently only supports linear transformations. Each `Group` has an aspect which computes its transformation matrix from its state variables. Different `Group` types define different functions, for example to employ quaternion or Euler angle representations for rotation. `Group` objects manage the hierarchy for their member objects. For example, the members need only draw in their local coordinates as the group ensures that the proper transformations are applied.

Drawable objects can provide a variety of aspects. One of the most common is a distinguished point; for example,

the center of the object, a vertex, or a point on its surface. In addition to providing its position, a `DistinguishedPoint` object may provide many other aspects, such as the surface normal and tangents at the point. Although these quantities are affected by the transformation hierarchy, `Drawable` objects need only define aspects of `DistinguishedPoint` in local coordinates as hierarchy mechanisms automatically perform the required transformations. Some less obvious aspects for points are discussed in section 5.1. Interaction techniques and constraints are typically defined in terms of `DistinguishedPoints,` so they can be applied to any object. Rather than their own manipulation techniques, objects merely need to provide these aspects.

Bramble predefines a variety of basic object types, along with standard aspects with which to manipulate them. For example, the 2D set includes lines, circles, rectangles, ellipses, and polygons, as well as a general parametric curve class which only requires providing the parametric function (see [13]). Standard 3D objects include cones, tori, cylinders, etc. Polyhedra can be defined in several ways, including readers for several data file formats.

Geometric constraints are represented by `Drawable` objects which create associated differential interactors on some of their aspects. For example, a connection constraint would take two `DistinguishedPoint` position aspects, and compute the difference as an aspect. When created, it would also create a go–towards interactor on the distance aspect to maintain the constraint, as well as providing a method for drawing itself. Bramble includes many basic constraint objects in two and three dimensions, including point connection, distance, collinearity, normal or tangent alignment, and parallel. Inequality constraints in the basic set include constraints to keep points inside many of the basic shapes. All of these constraints operate generically on `DistinguishedPoint` aspects so they can be attached to many types of objects.

Cameras are a special subclass of `Drawable.`They provide special aspects which define a viewing transformation matrix. For perspective transformations, the position of the camera in worldspace can be determined from this matrix, so the camera can be drawn and manipulated as a scene object. Cameras can also provide a variety of aspects for controls, many of which are examined in [14]. Bramble's basic object set includes many varieties of cameras, including orthographic transformations (which are required for viewing in 2D), and many of the standard perspective representations.

5.1. Objects for 3D interaction

Bramble provides a standard set of objects and aspects to support basic 3D interaction. The goal is to provide a fast and easy interface for 3D experimentation. The basic interface has a particular style, which derives from systems we have built in our lab over the past few years and is shown in Figure 4. Applications can use or ignore these interface elements.

An important part of Bramble's 3D interface is a floor called the *groundplane* and an optional back wall. This reference object defines the coordinate system and gives the user a reference frame called a stage[17]. To further aid the user's perception of 3D objects, Bramble can draw shadows on these reference objects. These plane shadows can be easily generated with the available hardware using techniques described in [6]. Bramble's shadows can either be simple drop shadows, or shadows computed from the positions of the light sources. The positions of shadows are computed as aspects, so they can be directly constrained and manipulated, an interaction technique shown in [17]. In Bramble, shadow manipulation can be used to control light sources as well as objects.

The follow interactor permits the user to couple the motion of an aspect to the motion of an input device. An aspect which represents a position in 3–space could be connected to a 3D input device, although we typically only have a mouse. Bramble's standard 3D input technique uses the mouse along with a special graphical object called the *mousepole.* The mousepole is a vertical line which extends from the floor to the mouse position. The line is used to provide feedback of the 3D location. As the mouse is moved, the pole tracks it, with the top point moving parallel to the groundplane. When a designated elevator button is held down, the pole top moves in a vertical plane instead of front-to–back. The tip of a mouse pole can be tracked by a differential interactor.

Another method for controlling 3D objects with the mouse is via Through-the-Lens (TTL) controls. Through-the-Lens controls are special aspects of `DistinguishedPoint` objects which compute where the point appears on the screen in a given view. In [14], these controls are discussed for manipulating cameras. However, by constraining the camera's configuration, the controls can also be used to manipulate objects. Because they only specify 2 dimensions, TTL controls are not sufficient to form a 3D interaction technique. However, they serve as a building block, which when combined with other constraints, can be used to describe 3D interactions. An example is the axis translation handle described in Section 8.

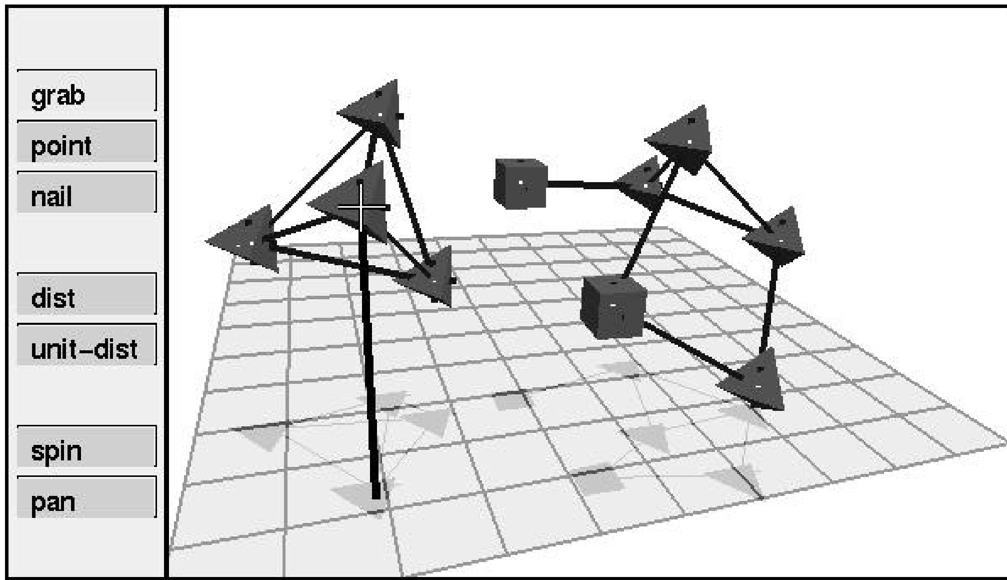

Figure 4: The *Tinkertoys* application demonstrates Bramble's standard 3D interface. The user manipulates the point objects using the mousepole. A groundplane and shadows are provided to help depth perception.

Lights are special `Drawable` objects. Like cameras, lights often have positions in the world and can be depicted in the scene (exceptions being distant and ambient light sources). Light objects provide aspects for their position, orientation, and intensity which Bramble uses to configure the hardware lighting support for drawing. We have been exploring different aspects to provide useful lighting controls. For example, spotlights can be manipulated by dragging the ellipse of their umbra on the ground plane, as shown in Section 1.1 and Figure 2. Another experimental control is providing the result of the lighting calculation at points as an aspect, allowing a user to directly control the color that a point appears.

Bramble also has special mirror objects. Using a technique similar to that used for shadows, Bramble can draw reflections in a planar mirror. Like shadows, the positions of objects in the mirror are aspects which can be directly controlled and constrained. Reflection aspects affect the mirror, the reflected object, and the camera. Although Bramble cannot draw reflections on curved surfaces, it can generate input for renderers which can. Bramble therefore contains some special aspects to aid in positioning reflections. `DistinguishedPoints` can compute an aspect which, based on camera position, computes the reflection ray. By constraining an object to lie along this ray, what appears in the reflection can be controlled. By attaching a light source to the ray, a specular highlight can be positioned.

Bramble's facilities to manipulate reflections are unlikely to provide a standard 3D interaction technique. They are described here to give an example of how aspects can be created to provide unusual, experimental interactions. New aspects can be defined which permit the user to directly control some quantity of interest. Similarly, if new methods for position points are developed, they can not only be applied to points on objects, but also to other points such as those on reflections and shadows.

6. AN EMBEDDED INTERPRETER

Bramble contains an embedded interpreter for a LISP-like programming language called *Whisper.* Applications can use the interpreter to provide services to the user, such as customization scripts and as a save and load format. The interpreter also facilitates the process of constructing the applications, helping to alleviate some of the drawbacks of the environment in which Bramble was constructed[2].

Whisper is a programming language akin to Scheme and LISP. The Whisper interpreter is specifically designed to be embedded in interactive applications, and is constructed as a toolkit to facilitate incorporation. Like Scheme, Whisper supports dynamic types and lexical closures. It has an extensible type system which allows it to manipulate pointers to C++ objects without the need to access their internals.

[2] Bramble was constructed in C++ instead of a more dynamic environment, such as Common LISP or Scheme, because available compilers failed to provide the needed numerical performance.

Whisper supports first class environments as an object system and a module mechanism. An environment in Whisper is a name space, or a pairing of names and values. In order to support Scheme-like lexical closures, environments are hierarchical, shared, and automatically memory managed. Whisper allows programmers to directly refer to these objects, and to pass them around. Environments are used as objects in Whisper, providing a flexible communication mechanism with C++ data structures. The C++ objects in Bramble keep an associated environment with them. Often, rather than using C++ fields, objects will store elements, including parameters, aspects, and methods, inside of their carried environment. This permits these elements to be accessed from Whisper code and dynamically altered. It also provides a concise and clean protocol for objects to inquire about one another. For example, it is possible to determine if an object has defined a particular aspect.

Hooks in Bramble are pointers to either a C++ function or a Whisper closure. Many parts of Bramble's behavior are defined by hooks, allowing them to be altered dynamically. This permits, for example, a configuration file to alter the behavior of objects so that new interaction techniques can be prototyped.

7. WINDOWING SUPPORT

Most of Bramble is concerned with providing support for the graphical objects which the user will actually place into models. However, there are other supporting objects, such as windows and buttons, which applications require and Bramble supports. Many other toolkits provide similar support for these things; relevant aspects of how they are handled in Bramble will be mentioned briefly. The emphasis in the development of these objects is in providing tools which will allow these parts of applications to be built quickly, so that more effort can be concentrated on the development of graphical manipulation techniques with the differential approach.

Bramble contains an object type called a `Frame` which represents a window system window object. A frame can contain many subwindows. One of these subwindows may be a `View` which is where Bramble draws a depiction of the current state of the graphical objects. Each `View` has an associated camera and attributes which determine how the image is rendered. There can be many views at once, but each must have its own camera and frame.

A `Frame` can contain other subwindows besides a `View`. These are used to provide buttons and other widgets around the edges. Examples can be seen in the applications depicted in the figures. Like the graphical objects, the widgets are designed to automatically update themselves to maintain a consistent view of the values that they access. For example, a button can be created to "watch" a whisper variable, or a slider can be placed on any aspect to track its value. The slider interactor can also control the aspect to which it is connected by creating differential interactors in response to mouse clicks. These sliders permit not only dragging an aspect's value, but also nailing or bounding it.

8. INTERACTION EXAMPLES

With the differential approach, interaction techniques are defined by attaching differential interactors to object aspects at appropriate times. This section introduces some of Bramble's mechanisms for defining when to perform these actions by providing examples of how interaction techniques are defined.

A Distance Constraint: Graphical objects can create associated differential interactors when they are created. For example, the distance constraint used as an example throughout this paper would be implemented as a graphical object which creates a *go–towards* interactor attached to its distance aspect.

Dragging: Differential interactors can be created in direct response to user events. For example, to implement dragging, a *follow* interactor is created between the mouse and the grabbed point when the mouse button is pressed. When the button is released, the interactor is destroyed.

Constrained Dragging: Differential interactors are used in combination to create more complex behaviors. For example, to rotate an object, the same technique as dragging would be used, except that a differential interactor would be used to fix the center of rotation. These additional interactors could be created and destroyed with the follow, or at the time when rotation mode is selected, or at some other time depending on the desired interface.

Handles: Aspects can provide hooks to be called when they are grabbed or released for dragging. This permits, for example, different points to have different behaviors. This can be applied to create special handle objects. As an example, consider creating an axis translation handle like those seen in [32] and [10]. These handles are special objects which attach to scene elements and drag them along a particular direction when grabbed. To create such a handle, a graphical object is created which has an aspect that is some offset from the point it is meant to control. When it is grabbed, it creates constraints which force the

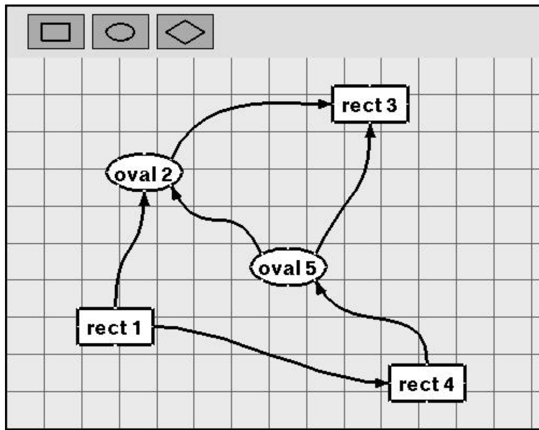

Figure 5: This application allows users to edit box and arrow diagrams. Constraints not only keep connections, but also prevent boxes from overlapping.

point to move only along the desired axis. Because the motion is restricted, the two dimensions of the mouse are sufficient to specify the translation, so a through-the-lens control can be used. When the point is released, the axis constraint is removed.

9. APPLICATIONS IN BRAMBLE

This section describes some of the initial applications which have been constructed with Bramble. These applications consist of two parts: C++ definitions of new object types and aspects, and a Whisper driver. The driver creates some windows and objects, binds actions to events, and begins the interactive loop.

Boxes and Arrows: Figure 5 depicts a box and arrow diagram editor. This is a popular demonstration for constraint-based toolkits because constraints can keep the diagram together as pieces are moved. Because the `DistinguishedPoint` aspects to which the arrows are connected provide curve normal information, the arrows can connect correctly. Unlike systems created with most other constraint-based toolkits, Bramble permits non-linear constraints, such as distance and orientation, in addition to simple connections. A special differential interactor uses constraints to prevent boxes from overlapping, instead causing them to collide and push one another away as they move.

Curve Modeling: The program depicted in Figure 6 is a new version of the modeler described in [13]. This tool aims to demonstrate how differential methods permit a uniform interface to a wide variety of object types. All objects can be grabbed by their points and dragged. The application provides a range of objects including not only the typical ones like lines and circles, but also things like spirals. New object types can be defined simply by providing their parametric functions and a small amount of auxiliary information. Special interfaces do not need to be defined for each object. This applications also provides a variety of constraint types.

A Mechanism Simulator: The *MechToy* application, described in Section 1 and seen in Figure 1 is a variant on basic drawing programs. Variants of basic objects have been defined so they appear as in textbook illustrations; for example connectors appear as bolts, lines appear as linkage rods, and nail constraints appear using the standard grounded point symbol from textbooks. The interface is tuned to infer connections between pieces, and to provide motor objects which animate when enabled. These features are simple to create with Bramble's abstractions.

3D Tinkertoys: Figure 4 depicts a simple 3D construction toy which permits users to connect point objects together with distance constraints. The intent was to recreate the functionality shown in the system presented in [36]. Tinkertoys uses the standard Bramble 3D interface to place and adjust points, and to create constraints.

For this application, we attempted to design an interface with a minimum set of command keys. In fact, the entire application can be used with only the left mouse button and the elevator button for the mousepole. We use a simple view control. When the selector button is set to "spin," the left mouse button becomes a world rotator. Upon mouse down, the "center of the universe" is pinned down, and a tracking interactor is created between the through-the-lens position of the grabbed point and the mouse. This provides an interaction similar to the virtual sphere of [7]. A similar control is used for panning.

Differential Control Demos: To demonstrate experimental interaction techniques, such as reflection and lighting control, we often need to build small dedicated demonstrations. *Showoff* is a Bramble application designed to aid in this process. Unlike other applications, Showoff does not create windows or fill the scene with graphical objects. Instead, it defines Whisper functions which make it easy for scripts to do these tasks. For instance, it defines a function which automatically creates a framed view, complete with a standard camera and buttons along the side to set various viewing modes. Showoff creates commands for finding and manipulating object aspects.

An example use of Showoff is depicted in Figure 2. The script merely creates a window, the pieces of the Luxo

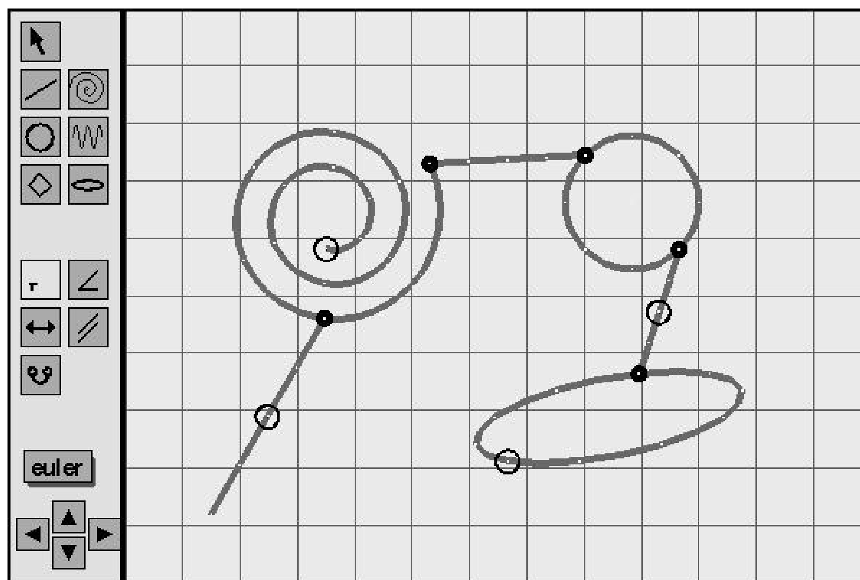

Figure 6: This curve modeler allows a variety of objects and constraint types.

lamp, and the background blocks. Showoff's interface permits the user to control any of the aspects provided by the objects. For example, it can create sliders for the bulb's color, or allow the umbra circle to be dragged along the floor. In contrast to Tinkertoys, Showoff's interface was designed to maximize the number of functions it provides, even though this was done at the expense of usability.

10. CONCLUSION

Graphical editing applications have much common functionality. With Bramble, it is possible to easily create variants of applications. For example, a drawing program can become a mechanism simulator by adding a few new object types and making some minor alterations to the interface. Once a basic application is in place, new functionalities can be explored. We believe that there are advantages to exploring interactive functionalities in the context of applications, rather than in isolation as is typically done in user studies such as [7, 19].

Bramble is good enough to build applications which push the limits of the differential approach. Scalability and performance are two of the greatest concerns. Integrating the differential approach with more common techniques also poses some new issues. However, our main concern now is how to use the differential approach. Good interfaces for 3D and constraint-based graphical editors are challenging to design. Tools like Bramble make it easier to develop and test interaction ideas, but ultimately are merely aids in addressing the issues of application interfaces.

The differential approach employed in Bramble facilitates experimentation with new interaction ideas. Aspects can be defined easily. Because they can be almost arbitrary non-linear functions of one another, there is substantial flexibility to define aspects which directly control quantities of interest. These aspects can be constrained and controlled in combination. Bramble provides mechanisms for employing these controls in the design of interaction techniques. It also makes it possible to create frameworks for their evaluation, or to place them in realistic settings so they can be used.

References

[1] Aldus Corporation. Intellidraw. Computer Program, 1992.

[2] Sherman R. Alpert. Graceful interaction with graphical constraints. *IEEE Computer Graphics and Applications*, pages 82–91, March 1993.

[3] Joel Bartlett. Don't fidget with widgets, draw! Technical report, DEC Western Research Laboratory, May 1991.

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

[5] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986. Proceedings SIGGRAPH '86.

[6] James Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, pages 82–86, January 1988.

[7] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3d rotation using 2d input devices. *Computer Graphics*, 22(4):121–130, August 1988. Proceedings SIGGRAPH '88.

[8] PHIGS+ Committee. Phigs+ functional description, revision 3.0. *Computer Graphics*, 22(3):125–215, 1988.

[9] Computervision Corporation. DesignView. Computer Program, 1992.

[10] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, and Andries van Dam. Three-dimensional widgets. In *Proceedings of the 1992 Workshop on Interactive 3d Graphics*, pages 183–188, March 1992.

[11] Michael Gleicher. Integrating constraints and direct manipulation. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pages 171–174, March 1992.

[12] Michael Gleicher. Practical issues in graphical constraints. In *PPCP-93: Workshop on the Principles and Practice of Constraint Programming*, April 1993.

[13] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

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

[15] Michael Gleicher and Andrew Witkin. Supporting numerical computations in interactive contexts. In Tom Calvert, editor, *Graphics Interface*, pages 138–145, May 1993.

[16] Tyson R. Henry, Scott E. Hudson, and Gary L. Newell. Integrating gesture and snapping into a user interface toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 112–121, October 1990.

[17] Kenneth P. Herndon, Robert C. Zelenik, Daniel C. Robbins, D. Brookshire Conner, Scott S. Snibbe, and Andries van Dam. Interactive shadows. In *Proceedings of the 1992 ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 1–6, November 1992.

[18] Ralph D. Hill. A 2-d graphics system for multi-user interactive graphics based on objects and constraints. In E. Blake and P. Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*, pages 67–92. Springer Verlag, 1991.

[19] Stephanie Houde. Iterative design of an interface for easy 3-d direct manipulation. In *Proceedings CHI '92*, pages 135–142, May 1992.

[20] Scott E. Hudson. Adaptive semantic snapping – a technique for semantic feedback at the lexical level. In *Proceedings CHI '90*, pages 65–70, April 1990.

[21] Silicon Graphics Inc. *Graphics Library Programming Guide*, 1991.

[22] Masao Iri. History of automatic differentiation and rounding error estimation. In Andreas Griewank and George Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 3–16. SIAM, January 1991.

[23] Solange Karsenty, James A. Landay, and Chris Weikart. Inferring graphical constraints with Rockit. In *HCI'92 Conference on People and Computers VII*, pages 137–153. British Computer Society, September 1992.

[24] John Harold Maloney. *Using Constraints for User Interface Construction*. PhD thesis, University of Washington, 1991. Appears as Computer Science Technical Report 91-08-12.

[25] Brad Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[26] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Phillipe Marchal. Comprehensive support for graphical, highly-interactive user interfaces: The garnet user interface development environment. *IEEE Computer*, November 1990.

[27] Greg Nelson. Juno, a constraint based graphics system. *Computer Graphics*, 19(3):235–243, 1985. Proceedings SIGGRAPH '85.

[28] Dan R. Olsen and Kirk Allan. Creating interactive techniques by symbolically solving geometric constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 102–107, 1990.

[29] John K. Osterhout. Tcl: An embeddable command language. In *1990 Winter Usenix Conference Proceedings*, 1990.

[30] John K. Osterhout. An X11 toolkit based on the Tcl language. In *1991 Winter Usenix Conference Proceedings*, 1991.

[31] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1986.

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

[33] Ivan Sutherland. *Sketchpad: A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[34] Pedro A. Szekely and Brad A. Myers. A user interface toolkit based on graphical objects and constraints. In *OOPSLA '88 Proceedings*, pages 36–45, September 1988.

[35] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain specific graphical editors. In *Proceedings of the 1989 ACM SIGGRAPH Symposium on User Interface Software and Technology*, November 1989.

[36] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24(2):11–21, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[37] 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*, 27:81–84, August 1993. SIGGRAPH '93 video paper.