But the ancient debate on emergence, whether indeed wholes may have properties not intrinsic to the parts, is besides the point. The fact is, that parts have properties that are characteristic of them only as they are parts of wholes; the properties come into existence in the interaction that makes the whole.

— Richard Levins and Richard Lewontin
The Dialectical Biologist, p 273

Chapter 7

A Graphics Toolkit

To this point, we have introduced the machinery of the differential approach, the methods for its realization, and the structure of a general purpose implementation. In this chapter, we consider how the approach can be encapsulated to support the construction of graphical editing applications. Graphical applications typically share a wide variety of functionality, so their construction is often facilitated by the creation of toolkits that encapsulate the common needs. This chapter discusses such a toolkit built on top of the differential approach. *Bramble* is an object-oriented graphics toolkit, built on the infrastructure of the previous chapters.

Bramble has much in common with other object-oriented graphics toolkits designed to support direct manipulation graphical editors, such as Garnet [MGD+90], Inventor [SC92], GROOP [KW93], and Alice [PT94]. Bramble's primary distinction is that it is designed on top of the differential approach. The major consequences of this are:

- The differential approach is used to create almost all graphical manipulations.
- Graphical objects have connectors that compute their various attributes, and often provide these standardized connectors in lieu of specific interaction code.
- Snap-Together Mathematics is used to connect objects together and to connect interactive controls to objects.
- Bramble provides support for application features such as geometric constraints that are easy to create with the differential approach, but difficult to implement in standard toolkits.
- Bramble's control flow is dictated by the differential approach. The model of an ODE solver with events interleaved between steps, as described in Section 6.2,

provides a centralized main loop that calls application code when needed. The differential approach allows continuous motion direct manipulation to fit nicely into such a scheme. callback style architecture as mechanisms exist that permit direct manipulation to fit in such a scheme.

Bramble not only shows how the machinery of the differential approach can be applied to the construction of graphical applications, but also how the approach's machinery can be encapsulated and hidden from the applications programmer. My goals in constructing Bramble were:

- To make it possible to quickly prototype a number of applications to illustrate and explore the differential approach. The emphasis is on speed of construction and extensibility, rather than on industrial strength tools. Much of this goal was pragmatic: I needed to construct enough examples to support the conjectures of this thesis in a reasonable amount of time.

- To support a variety of applications. It was important that Bramble could support both 2D and 3D applications. While the focus is on graphical editors and modelers, Bramble also supports other applications such as object viewers and visualization tools.

- To support the basic services of graphical applications typically provided by toolkits.

- To facilitate building applications that provide the features that the differential approach supports, such as geometric constraints.

- To facilitate experimenting with interaction techniques and evaluating them in context within applications.

- To show how the architectural features of the differential approach could impact applications architecture. In particular, the approach can enable increased modularity, since Snap-Together Math provides a common connection mechanism between parts, and separation of manipulation and representation aids encapsulation.

- To show that the machinery of the approach can be sufficiently encapsulated. The application programmer does not need to see the mathematics in order to make use of the approach's features. Bramble is designed to let developers program with the familiar abstractions of graphical applications. They see the abstractions of the approach only when defining new interaction techniques. Bramble programmers never need see the inner workings of the solver.

7.1 The Bramble Application Model

The flow of control in Bramble is dictated by the differential approach, as described in Section 6.2. Time flows forward as the ODE solver steps forward, continuously evolving the state of the objects. By continuously viewing the world as it evolves, we can see objects move according to their controls. At discrete instants, such as when an event occurs, changes to the objects in the world can be made. However, such events are instantaneous impulses: all changes to the configurations of objects must happen via the passage of time by the ODE solver.

The differential approach's model of time is significant in toolkit design for two related reasons:

1. It provides a uniform "main loop" for all applications.

2. It provides a way to incorporate simultaneous, continuous actions such as dragging into an event driven architecture. Events can asynchronously begin and end continuous motion actions, concurrency is handled by the solver: there is no need for multi-threading or other time sharing mechanisms to achieve concurrent, asynchronous actions.

These two elements make it practical for Bramble to use event callbacks as its sole application control mechanism. A Bramble application does not contain a "main" loop, but rather, after the application concludes its initial setup, it simply call a function defining a standard loop that runs the ODE solver, keeps the views of the world up to date, and calls appropriate fragments of application code when needed. Code to be called is specified in hooks, variables that can be bound to fragments of code that are accessed at defined times. Hooks are discussed in more detail in Section 7.6.

The overall structure of Bramble is shown in the schematic of Figure 7.1. Bramble is built on top of the Silicon Graphics GL graphics library [Sil91] and the Snap-Together Math toolkit, discussed in Chapter 5. Bramble itself contains no solver code and does not even provide access to the solver's internal structures. Bramble also includes the Whisper embedded interpreter, described in Appendix A. The symbiosis of Bramble and Whisper will be discussed in Section 7.1.1.

A Bramble application consists of two parts:

- Code implementing any custom classes.

- A program that is run to execute the application. This program typically sets up the application, doing tasks like opening windows, creating initial objects and widgets, and defining hooks. After initialization, the program starts the differential process' time flowing. Once time is started, application code is only executed when a hook is called.

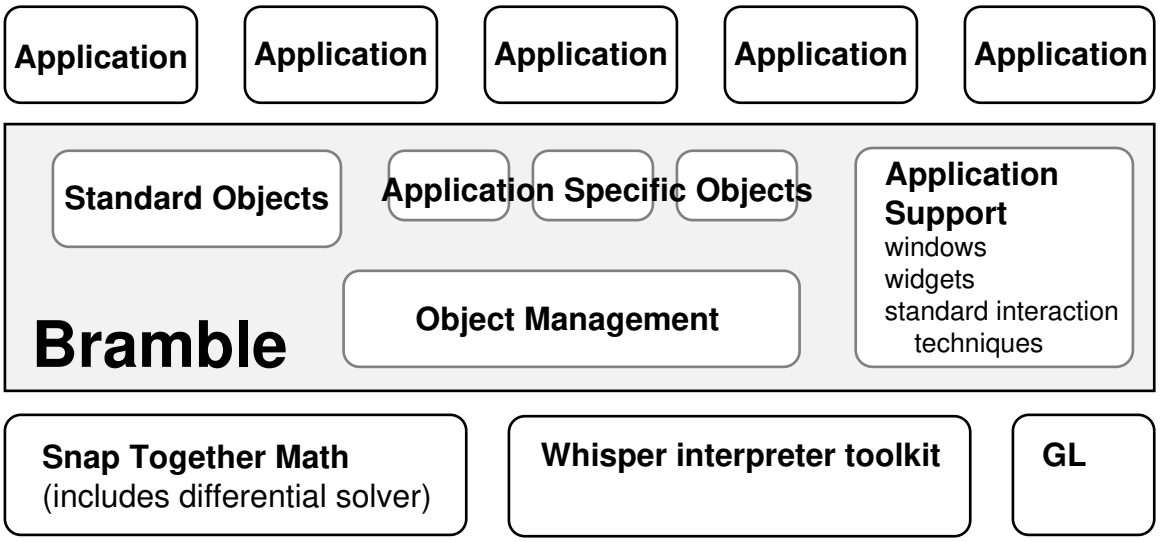

Figure 7.1: The pieces of the Bramble toolkit. Bramble is built on top of Snap-Together Math and the Whisper interpreter. It contains a variety of pre-defined object classes, support for managing sets of graphical objects, and other miscellaneous pieces needed by graphical applications. A Bramble application typically consists of a Whisper driver program and (optionally) C++ definitions of new object types.

The major pieces of Bramble are akin to other similar toolkits. Each is affected by the use of the differential approach:

Object Management – Bramble's mechanisms for managing sets of objects, discussed in Section 7.3, provide access to the features relevant to the differential approach, such as keeping track of state variables and connectors. All objects, even windows and widgets, can have state and connectors.

Standard Object Types – Bramble's standard object types, discussed in Section 7.5 provide a wide range of connectors to support graphical manipulation.

Hooks – Bramble provides a set of hooks that aim to be sufficient to specify a range of interaction techniques. These are chosen to provide a convenient set of opportunities to alter the controllers and objects of the differential approach. Bramble's hooks are discussed in Section 7.6.

Windows and Widgets – Bramble's windows can contain views of the world that are automatically kept consistent, and many of the supported widgets, such as sliders, are designed to manipulate their targets using the differential approach. These features are described in Section 7.7.

Standard Interaction Techniques – Bramble contains many basic graphical manipulation techniques in its library, all of which are defined with the differential approach. Picking and snapping services are tuned towards the selection of connectors for manipulation.

This chapter will discuss these various pieces of Bramble, following a discussion of an important part of Bramble's structure, and a simple example of how Bramble is used.

7.1.1 Whisper and Bramble

The Whisper embedded interpreter, described in Appendix A, is an important part of Bramble. An embedded interpreter is a useful feature for graphical applications as it provides support for features such as saving and loading user data files and user extensions. However, Whisper plays an even more significant role in Bramble. Internally, the Whisper interpreter's implementation is used by Bramble for support, and externally, much of the application programming in Bramble is done in Whisper.

As discussed in Section 2.3, using an embedded interpreter in a graphics toolkit is a common technique. However, the differential approach removes some of the drawbacks. The speed-critical computations occur as part of the mathematics; all standard pieces can be written in the compiled host language. The centralized main loop provided by the ODE solver also interacts nicely with the embedded interpreter. Most application behavior is defined by hooks. Bramble's hook mechanism, discussed in Section 7.6, permits pointers to C++ functions, text fragments of Whisper code, and Whisper closures to be dynamically assigned.

Like other similar languages, Whisper has a runtime support system for memory and object management. Bramble also uses these facilities for its needs. The advantages are twofold: first, it provides Bramble with a dynamic, memory managed object system; and second, it means that Bramble's objects are easily accessible from the interpreter.

The majority of Bramble application programming is done in Whisper. Bramble is designed such that applications are constructed by extending the generic default application. Constructing applications in the extension language is, therefore, sensible. Application programming in Whisper has many advantages:

- The interpretive nature of Whisper leads to faster turnaround than the statically compiled C++ environment. [1]
- The interactive loop of the interpreter provides a useful debugging environment, whereas the complexity of the entire C++ system makes using the standard C++ debuggers awkward.

[1] This is more a statement about the C++ programming environment at the present time than about Bramble.

- Whisper's abstraction mechanisms, notably first-class functions and dynamic objects, are useful in the design of interactive systems.
- Many of the details of the system are hidden from a Whisper programmer. For example, window management and refresh, and the solver internals are hidden. In fact, there are no mechanisms provided in Whisper to access internal solver data structures. The language does not even have vector or matrix constructs!

There is nothing that *must* be done in Whisper. However, almost all application programming *can* be done in whisper.

Bramble is actually wired into the Whisper interpreter as an extension. The C++ "main" of a Bramble application is a Whisper read–eval–print loop, typically that can select files to read from the command line. For my work, there is a single Whisper/Bramble executable and each application differs only by the Whisper program used to run the application. For many applications, however, a different version of the interpreter, with custom sets of extensions wired in, would be more desirable.

The interpreter-based organization of Bramble emphasizes that it meets its goals of insulating the application from the mathematics. The differential optimization is completely encapsulated inside the `ConstEngine` object as described in Section 5.4.5. Nowhere in the Bramble toolkit source code or applications is there mention of the internals of the solver process.

7.2 A Simple Example

In order to introduce the basic concepts of Bramble, a simple example will be presented. This example is designed to be similar to that used to introduce the Inventor toolkit. In the Inventor book [Wer94], a program called "hello cone" is given as a first introduction to Inventor. Here is the same program, one that displays a cone in a window, written with Bramble:

```
(1) (set my-view (make-view "Hello Cone"))
(2) (set my-cam (make-la-camera))
(3) (view-cam my-view my-cam)

(4) (set c (make-cone))
(5) (c (set material shiny-red-material))

(6) (go)
```

This simple example brings out the basic notions of Bramble. A program places graphical objects in the world, and does not worry about lower level details such as how they are drawn or how the windows are managed. The first line creates a window that

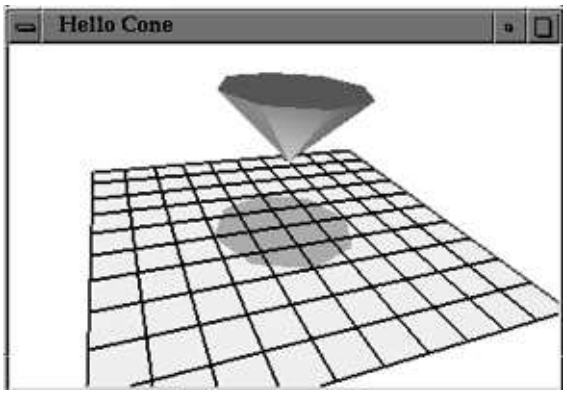

Figure 7.2: The simple "Hello Cone" program in Bramble.

views Bramble's world. Line 2 creates a default "lookat" camera, and line 3 specifies that the view created on line 1 should look at the world through the lens of this camera.

Lines 4 and 5 of the program create the cone object, and changes its color to red. The color is changed by setting the property of the cone object that specifies the surface properties to be a predefined material that is shiny red. This unusual syntax for accessing object fields is described in Appendix A. Notice that when the cone is created, it is by default created in the world and is therefore visible to viewers of the world.

The "action" of the program takes place in the last line. The `go` routine begins the interactive loop which is in the toolkit proper. In differential terms, time is started. Steps are continually taken, and in between any events are handled. The interactive loop runs continuously until either the program is killed, or a flag is set.

The simple 3D example receives a lot of a 3D interface from Bramble's defaults, as shown in Figure 7.2. By default, Bramble creates a number of interface elements such as the ground plane, some lights, and the shadows dropped vertically onto the ground as if the sun were at high noon. These defaults can easily be overridden. In a sense, the Bramble program simply modifies the default application and runs it.

The most obvious difference between this program and the Inventor equivalent is that it is written in Whisper, not C++. This difference can be emphasized by adding the line

```
(add-key dev-esckey k-any k-down
    (lambda (v) (set bramble-going nil)))
```

which attaches a procedure that sets a flag to the escape key. This flag signals the `go` routine to stop the interactive loop and return. Since there is no code after the call to `go`, Bramble will return to an interactive Whisper prompt. This allows a user to interactively alter the program. The differential interactive loop can be restarted with another call to the `go` routine.

To get elements of a standard 3D interface, some standard key definitions can be read by replacing lines 1 through 3 with

```
(read std-world.wh)
(set v (make-standard-view "Hello Cone"))
```

at the top of the file. The Bramble standard 3D interface, described in Section 7.8 binds the mouse buttons to routines that permit objects to be grabbed and dragged and the viewpoint to be controlled.

With the 3D interface, we would notice that when we grab the cone in the simple program, it does not move. The primitive cone from the library is a rigid body that has no degrees of freedom. To make it movable, we place the cone in a transformation group with the lines

```
(set g (make-qs-group))
(add-to-group g c)
```

somewhere after the cone is created, but before `go.` The `make-qs-group` function creates an empty group that can rotate and scale (qs stands for quaternion and scale).

We might want to have the cone begin pointing upwards. Since this will require a non-differential change (i.e. we want it to instantaneously appear in the right orientation), we cannot use the differential machinery. This means we must actually directly access and modify the state variable of the object. This is done by

```
(set-qvar (get-q g) 'sy -1)
```

that sets the `sy` (scale in y) variable of the group to -1. We might then want to prevent the object from scaling as it is manipulated, which can be accomplished by

```
(freeze-scale g)
```

which is a special operation for groups.

The cone can now be grabbed and dragged using the standard Bramble mousepole manipulation technique, as described in Section 7.8. We will now add some other functionality that uses Bramble to operate the differential machinery. Suppose that we want to have the top of the cone fly towards the left. This is accomplished by taking a connector that computes the position of the top of the cone and attaching a controller to it. Since the cone object has a predefined "top" connector, this operation is as easy as

```
(controller (signal (c top) 'x) '= 5)
```

that finds the x coordinate output of the top connector of the cone and creates a con-

troller that drives it towards the value 5. We might want to instead bind this operation to a key so it happens only when we're ready

```
(add-key dev-gkey k-none k-down
    (lambda (v) (controller (signal (c top) 'x) '= 5))).
```

Notice how this example maps exactly to the abstractions of the differential approach. We manipulate an object by attaching controllers to its connectors. The actual change itself happens as time passes after the discrete instant when the controller is created. The manipulation is done completely in terms of the connectors of the objects, without regard for what parameters inside make them actually change.

Suppose we want to make a chain of cones. Here, we add a second cone and connect it to the first.

```
(set c2 (make-cone))
(set g2 (make-q-group))
(add-to-group g2 c2)
(pt-eq (c bottom) (c2 top))
```

The important thing to notice from this example is that we were able to create the connection by talking in terms of geometric or graphical objects, not mathematics. The constraint is created to connect the bottom of one cone and the top of the other.

The lack of mathematics in the last example is a result of us being fortunate that the cones had the connectors we desired. However, adding a new connector is easy enough. Suppose we wanted to add a new "horizontal" connector to the cone that measured if the cone was horizontal (i.e. that the y coordinate of its top and bottom are equal, as Bramble's coordinate systems defines the y axis to mean height). This can be done as

```
(c (bind horiz (minus-block (signal top 'y)
                            (signal bottom 'y))))
```

This code fragment uses the Whisper/Snap-Together Mathematics interface to create a Snap-Together Mathematics output that computes a new attribute called `horiz`. The attribute is computed by subtracting the height of the bottom of the cone from the top. A function block is used to perform the subtraction. Whisper scoping rules cause the code fragment that creates the block to be executed in the cone's environment, so that `top` and `bottom` refer to the cone. The code fragment stores stores the function block as a field in the cone object's environment permitting other objects to access it to use it as a connector. This code fragment could be written as a procedure that could be applied to other cones

```
(define add-horiz (a-cone)
 (a-cone (bind horiz (minus-block (signal top 'y)
                                  (signal bottom 'y)))))
```

These connectors could be used just as any others, for example

```
(add-horiz c2)
(controller (c2 horiz) '= 0)
```

7.3 Bramble's World

The focus of Bramble is on building object-oriented graphical applications. One of Bramble's central roles is to help maintain the set of objects that are being presented to the user. In Bramble, there is a single, implicit "world" in which all objects exist. By default, when objects are created they are placed in the world. The concept of the world is implicit in Bramble, unlike Inventor which requires explicit creation of "scene graphs." The advantage of this is simplicity, although the ability to have multiple simultaneous worlds is sacrificed.

The world provides a uniform 3D coordinate system for all objects. The world is always 3D, even for 2D applications. In such applications, 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 underlying GL graphics toolkit takes a similar approach.

In Bramble, the programming model is not of screens that are continuously redrawn. Instead, the model is of a time continuous world into which objects are placed and manipulated. Images on the screen provide views of this world.

7.3.1 The Bramble Object System

The mechanisms that support state variables, connectors, and other object management are implemented at a general level. This permits all major object types in Bramble, including windows and widgets, to support features that are usually associated with graphical objects, such as having state and connectors.

All major types of objects in Bramble are subclasses of the type `IDObj`. Having a common base allows common functionality to be shared among all types of objects. For example, each object has the ability to be named, and is assigned a unique ID number that serves as a soft pointer (hence the name `IDObj`). Each `IDObj` has a field that allows its major subtype to be determined dynamically. An object manager keeps track of all `IDObj`s created. The major subclasses of `IDObj` are:

`Drawable` – a graphical object in world space. This includes not only the user created scene objects, but lights, cameras, hierarchy elements, and world space interface elements such as 3D widgets.

`IDrawable` – a special object that exists in screen space, rather than in world space. Effectively, these objects are attached to lenses of cameras.

`View` – an object representing a view of the world on the screen. It is basically a pairing of a camera and a window, with added responsibilities for determining drawing parameters.

`DistinguishedPoint` – an important type of connector that represents a specific point on some object.

`Imagepoint` – a special type of connector that represents the position on the screen of a point.

`SubWin` – a subwindow of a screen window. Subclasses include widgets like buttons and sliders.

`Frame` – the "physical" GL window on the screen. May contain several `SubWins`, one of which may be a `View`.

`Material` – a set of surface properties.

`Demon` – a special object that performs its hooks when triggering events occur.

One of the key ideas behind Bramble's object management is that each IDObj contains a Whisper environment. In Whisper, environments serve as a dynamic object system (see Appendix A). Using Whisper environments as an object system for C++ has several advantages:

- It provides easy access to the object from Whisper.
- By defining methods as values in the Whisper object, they can be dynamically altered.
- Fields and methods can be added and removed dynamically as needed.
- Objects can inquire about which fields and methods other objects contain.
- Whisper environments, to a certain extent, are automatically memory-managed.

The dynamic nature of objects is important: fields and methods may be added in response to user actions. It also allows experimentation with application behavior: objects can be interactively modified while the application is running.

Any `IDObj` can have state variables and connectors. The base class handles management of these differential features. It also permits objects to create differential controllers or employ function blocks. These are automatically removed when the object is deleted. Object deletion is also handled by the base class using lists of hooks. This permits an object to define work that must be done when the object is removed. Dynamic definition of deleters, very much in contrast to the static C++ notion of a deleter method, is a very useful feature.

Memory management can be a difficult task in an interactive system. Bramble uses a variety of manual techniques to automatically manage memory without using a garbage collector. Bramble contains mechanisms to deal with the inverse garbage collection problem. An inverse garbage collector activates when an object must be deleted, for example in response to a user command, and insures other objects that refer to it are appropriately altered or deleted. A garbage collector does not provide this behavior. In fact, in a system that was simply garbage collected, the object would not be deleted until all references to it were removed. Interconnection of objects is pervasive in and central to the differential approach, so handling inverse garbage collection is important in Bramble. The mechanism used to handle this problem is to have each object maintain lists of other objects that should be notified or deleted when the object is deleted.

7.3.2 Graphical Objects

The graphical objects in Bramble's world are derived from the class `Drawable`. This class requires only a few methods. Subclasses of `Drawable` define a draw method that needs only to operate in the object's local coordinate frame. Objects may also provide other functionality. For example, most objects will define at least a few connectors so they can be manipulated, and will allocate a state vector to store their configuration. Other examples of optional `Drawable` functionality include ray intersection, bounding box computation and generation of external representations. Almost all methods apply in local coordinates. Hierarchy mechanisms perform required conversions automatically.

Bramble keeps all of the instances of `Drawable` on a list. The list of "the stuff" is Bramble's notion of a scene. Only one list is kept by the program. Selective drawing can effectively provide multiple scenes. Although the list is a flat representation, hierarchies are supported by the grouping mechanism, described in Section 7.5.3.

The graphical objects are not unlike their counterparts in other object-oriented graphics toolkits. One difference in Bramble is that each object class does not have to provide methods for interaction. An object need only provide connectors that the more general manipulation techniques can connect to. "Object-centric" styles of interaction are possible with Bramble. For example, an application can have each object define methods which are called by a dispatcher that receives events. Intermediate styles, for example

having an object use hooks to alter general manipulation techniques, are also possible.

Another distinction of Bramble's graphical objects is that they must also manage state and connectors in ways that permit the differential approach to manipulate the object. Bramble's object system provides a uniform place for objects to keep a state vector. Once a graphical object registers that it has a state vector, Bramble automatically insures that the variables are managed correctly.

Connectors are handled in a less uniform manner. In Bramble, a connector is simply a Snap-Together Mathematics `Port` that an object stores in a way that other objects can gain access to it. Standard mechanisms exist for certain important types of connectors, as discussed in Section 7.4.

7.4 Connectors in Bramble

In Bramble, a connector is simply a Snap-Together Mathematics `Port` that an object exposes in a way that other interested objects can gain access to it. The most common way to do this is for the object to place the `Port` in a field of its kept environment. Objects are free to create any types of connectors they want. Objects often provide special purpose connectors tuned to compute their specific types of attributes. Many specific examples are given in Section 8.1.

The most important type of connector in Bramble is the *distinguished point.* A `DistinguishedPoint` represents a particular point on a graphical object. The primary output of a `DistinguishedPoint` object is its position in space, but instances usually provide other attributes, such as surface normals and tangents, when they are known.

Distinguished points are first class Bramble objects unto themselves, that also happen to be `Port` objects as well. By being a real `IDObj` a distinguished point can:

- be accessed by standard object naming and reference techniques;
- store state, permitting the point to be moved around on the object;
- define hooks to specify its behavior;
- have other connectors associated with it, for example, to compute the position of its shadow on the floor or wall.

To standardize access, each object keeps a list of its distinguished points. Similarly, a global list of all distinguished points is kept to aid in such tasks as picking and snapping. Registry in this global list is automatically handled by the `DistinguishedPoint` creation process, and permits points to be found by name or by location.

The position and orientation of a distinguished point are not simply a function of the graphical object the point is on, but also the transformations applied to the object.

However, the hierarchy mechanisms automatically handle these transformations. A point must provide only methods for computing its connectors in the object's local coordinate frame. Jacobians are also composed for points in hierarchies.

Standard subclasses of `DistinguishedPoint` include:

Fixed Points that are a fixed position in the object's local coordinate system.

Point on Point that connect directly to a set of object variables that represent a position.

Free Points that store their position in the object's local coordinate space as their own state vector. Constraints are often used to keep such points in the volume of or on the surface of an object.

Different types of objects may also define special types of `DistinguishedPoint`. For example, a parametric surface might define a type of point that stores its parameter values as state. Such a point can slide along the surface, or the parameters can be frozen to create a fixed point.

7.5 Graphical Objects

Bramble supports a wide range of graphical objects. Many standard subtypes of `Drawable` are provided, and new application-specific types can be defined.

7.5.1 Standard Object Types

Bramble predefines a variety of basic object types, along with standard connectors with which to manipulate them. For example, the 2D set includes lines, circles, rectangles, ellipses, and polygons. A general parametric curve class allows the definition of new object types by simply providing the parametric function. Connectors and a drawing function are defined automatically from this as described in Section 8.1.1.

Most of the standard 3D objects are defined as rigid bodies that must be placed inside of groups to be moved. Most standard shapes, such as cones, tori, cylinders, and spheres are provided. Polyhedra can be defined in several ways, including readers for several data file formats.

A planar mirror is provided as a 3D rigid object. In terms of its geometry, the mirror is simply a rigid square. However, the mirror's draw function calls other objects' draw function with a new transformation that draws the object on the mirror's surface to simulate reflection. Mirrors define special connectors that permit the reflections to be directly manipulated.

Geometric constraints are represented by `Drawable` objects which create associated controllers on some of their connectors. For example, a connection constraint

would take two `DistinguishedPoint` position connectors, and compute the difference as a connector. When created, it would also create a `GoTowards` controller on the distance connector to maintain the constraint. The objects that represent constraints may provide drawing methods that provide feedback to the user. 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 object connectors so they can be attached to many types of objects.

7.5.2 Defining New Object Types

Many applications will need special purpose object types not included in the standard set. For example, the planar mechanisms simulator of Section 9.2 defines special objects for mechanical parts such as motors, linkage rods, and sliders. Often these objects are simply slight variations on the standard toolkit objects. For example, a linkage rod is simply a line segment that is drawn in a different way and has its length constrained to be fixed at creation.

Most of the standard functionality of `Drawable` objects can be implemented in Whisper, allowing new types of objects to be dynamically defined. For example, the following code defines a new type of `Drawable` that is a simple 2D line segment.

```
(1)  (defun make-line (x1 y1 x2 y2)                    ; procedure to create a line
(2)    (let* ((obj (wh-drawable 'line))                ; create an empty object
(3)           (q   (make-stobj 4)))                    ; make a 4 place state vector
(4)     (set-qvar q 0 x1 y1 x2 y2)                     ; put initial values into state
(5)     (caste-vars q vc-sceneobj)                     ; declare type of variables
(6)     (brobj-add-vars obj q)                         ; install variables in object

(7)     (point-on-vars-2d obj 0 1)                     ; create point connectors
(8)     (point-on-vars-2d obj 2 3)                     ; one for each endpoint

(9)     (bind-in (get-env obj) length                  ; define a length connector
(10)     (dist2d-block (signal q 0) (signal q 1)       ; compute with distance block
(11)                   (signal q 2) (signal q 3)))

(12)    (bind-in (get-env obj) drawf                   ; define draw method
(13)     (lambda (draw-flags)                          ; function of draw params
(14)       (prog (move (val q 0) (val q 1))            ; use GL move/draw commands
(15)             (draw (val q 2) (val q 3)))))         ; to draw line
(16)   obj))                                           ; return created object
```

This defines a procedure that creates an instance of the new type, given initial positions for its endpoints. It first creates an "empty" object (line 2) and a state vector

(line 3) that contains space for 4 variables. The object represents the line by the positions of the endpoints, so the initial values can be placed directly into the variables (line 4). Lines 5 and 6 declare the type of the variables and install them into the object. Lines 7 and 8 create `DistinguishedPoint` connectors for the endpoints. These connectors can obtain their values directly from variables in the state vector. Lines 9 through 11 create a length connector by computing the distance between the two endpoints. Lines 12 through 15 define a draw method for the line segment. The method takes a single argument, the drawing mode, that it ignores since it uses the defaults.

To facilitate the creation of new 2D objects types, Bramble provides a special `define-shape` function, detailed in Section A.2.1. It provides a concise syntax for specifying how the shape is drawn and placing connectors on it. The syntax permits intermediate variables to be specified for convenience and internal constraints on the object to be defined.

7.5.3 Groups

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 supports only linear transformations. Each `Group` has a connector that computes its transformation matrix from its state variables. Different `Group` types define different functions, for example to employ a quaternion or an Euler angle representations for rotation.

The `Group` class automates the process of building graphical hierarchies. `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 before the object draws itself, permitting the graphics hardware to handle the transformation hierarchy. Making sure that the correct results occur when position or surface orientation connectors are computed is more complicated. The `DistinguishedPoint` class handles this automatically.

Bramble's grouping mechanisms make it simple to build hierarchies. Adding an object to a group requires simply

```
(add-to-group group obj)
```

and everything is configured correctly. Similarly the ungrouping operation

```
(remove-from-group group obj)
```

does all of the required hierarchy management. One complication with ungrouping is that when the transformation is removed, the child object might jump in space. To prevent this, each object keeps a matrix that contains any previously applied transfor-

mation. This matrix is called `leftStuff` since it represents what was to "the left" of the object when it was drawn.[2] When an object is removed from a group, the group's transformation is premultiplied into the object's `leftStuff` matrix. This way the object does not jump when ungrouped. The `leftStuff` matrix is inserted into the matrix stack as each object is drawn.

Each different type of `Group` needs only to define its function that computes its matrix from its state variables. Some auxiliary information, denoting which variables, if any, correspond to rotation, translation, scaling, or center may also be provided. Bramble's standard set of `Group` types include rigid body transformations (translate and rotate) with both quaternions and Euler angles, rotate/translate/scale transformations, and a slider transformation that restricts objects to translate along a vertical line from the normal of a given point.

7.5.4 Cameras and Lights

A camera in Bramble is not just a 3D element. It is used for all transformations between the world coordinate system and screen coordinates. Like a `Group`, a Bramble `Camera's` primary distinction is a function that maps from its state variables to a transformation matrix. In the case of a `Camera,` however, this transformation is from world space to screen space. The inverse of this matrix is used to draw the camera, for example when another camera is looking at it.

The computer graphics literature is filled with many camera models. However, Bramble does not need to support a huge variety because the differential approach makes them unnecessary. As best exemplified by Blinn's work on spacecraft fly-bys [Bli88b], alternate formulations of a camera model are used to provide parameters which serve as convenient controls for the user. With the differential approach, a single camera model can be used with a variety of interactive controls. By mixing and matching controls, the user can specify camera positions as conveniently as with special purpose formulations. The through-the-lens camera controls of Section 8.1.4 provide a building block from which many camera manipulations can be created, including those of Blinn's paper as well as the more common LOOKFROM/LOOKAT model.

Bramble provides a small number of standard camera transforms. Orthographic transformations are provided to produce 2D images, as well as front, top, or side views of 3D scenes. The most common models in computer graphics define camera position by a rotation about the eye point. Models differ in how they represent the rigid body configuration of the camera. Bramble provides camera representations based on quaternions and Euler Angles. Usually, the quaternion representation is used because of its attractive mathematical properties, but an Euler angle representation is also available, mainly for expository purposes.

[2] I use the postmultiply or function application notation, where the point to be transformed is written to the right of the list of matrices.

A common representation of the orientation of a camera in computer graphics is to store a "look at" point that is to appear in the center of the image. A camera parameterized by LOOKFROM/LOOKAT is provided by Bramble. However, the functionality is typically obtained by employing a Quaternion camera and a through-the-lens control, because the representation has better mathematical properties. The primary use of the `LookAtCam` is when the configuration of the camera is going to be statically configured in a program, rather than manipulated. In such a case, the programmer will type the numbers that configure the camera, and therefore will require an easy to type representation.

Like cameras, lights are represented by graphical objects within the scene. A `Light` object is a special type of `Drawable` that has an extra method which initializes the graphics hardware to use the light. Lights provide information for optional shadow generation as a special connector which denotes the "bulb position" in homogeneous coordinates to allow for distant light sources.

Bramble supports several types of light sources, corresponding with the capabilities of the GL graphics library and some of the renderers used. The Bramble light classes are: point lights, distant lights, directional spot lights, and ambient light. As graphical objects in the scene, point and spot light sources can be manipulated and transformed as other graphical objects. This allows placing lights in "fixtures," like the Luxo lamp of Section 1.1. Special connectors for manipulating lights will be described in Section 8.1.5.

7.6 Hooks

Bramble's interaction model most closely resembles a dispatcher or notifier model [FvDFH90] where an application registers callback procedures with a centralized dispatcher that calls the appropriate procedures at the appropriate times. Bramble uses this model for most application behavior, permitting an application to specify functionality and then delegate the flow of control to the main ODE solver loop.

The callback mechanism in Bramble is through Whisper *hooks*. Like their LISP namesakes, Whisper hooks are variables that can be given procedural values that are called at appropriate times. In Bramble, the hook mechanism allows either a pointer to a C++ function or a whisper closure. Often a variable stores a list of hooks to be called, rather than just a single hook.

By defining behavior with hooks, rather than hard-coded routines, the behavior of an application can be dynamically altered. A goal in the design of Bramble was to provide a sufficient set of hooks such that all post-setup application behavior can be defined with hooks. This is more practical with the differential approach because time continuous activities are taken care of by the passage of time, so manipulation actions are discrete events that can be tied to hooks. Only in extremely rare cases do hook

functions not act instantaneously. Because the main loop only cedes control for short instants, processes such as screen update, window maintenance, and object animation can be maintained.

Many of Bramble's hooks are similar to those provided by systems not built with the differential approach. Having the right set of hooks to attach and detach controllers at the appropriate moments is essential to creating graphical manipulation. In this section, we summarize the most important of Bramble's hooks, and provide simple examples of how they can be used with the differential approach. These mechanisms will be used in Chapter 8 to define specific interaction techniques.

7.6.1 Events

An event in Bramble corresponds to a GL input device event, such as a key press or mouse click. Separate events are generated for both button up and down, allowing each to be handled differently. Bramble keeps a dispatch table that assigns a hook to each event. The dispatcher permits defining modifier selectors, allowing different sets of modifier keys to cause events to be interpreted differently. Event hooks are called with the current view as an argument.

The following code implements a 2D dragging behavior. The code fragment attaches a hook that handles left mouse-button down events. The `add-key` function takes an event name, a set of modifier keys, an event type, and a function to call when this event occurs. When it gets the currently selected point from the snap server (Section 7.7.3) and creates a constraint that equates this point with the position of the mouse, which is provided by the view as a connector.

```
(1) (add-key k-leftmouse k-none k-down            ; left-mouse down definition
(2)   (lambda (view)                              ; attach a procedure, called with view
(3)     (if (snapdp)                              ; if there is a selected point
(4)       (let ((c (pt-eq-2d (snapdp)             ; create a constraint attaching
(5)                          (view mouse-port))));    selected point to mouse
(6)         (add-key k-leftmouse k-any k-up       ; redefine left-mouse button up
(7)           (lambda (v) (delete c)))))))        ;    remove the attachment constraint
```

When the left mouse button is pressed, the procedure uses the `snapdp` function to get the currently snapped-to point (line 3). If there is a selected point, a constraint is created that attaches the point to the position of the mouse (line 4 and 5). The procedure also redefines the button up event to delete the constraint (line 6 and 7). Notice how the use of Whisper's lexical scoping rules keeps the internal data of this operation, the constraint, localized and how the code for the dragging action is only executed at the beginning and end of the operation.

Bramble automatically handles GL window events, such as raises, focus changes, and window removal.

7.6.2 Object Hooks

Event hooks are defined on a per-application basis. By default, an event handler does not provide different behavior based on the object being referenced. However, it is often useful to provide behavior on a per-object basis. Some toolkits, such as Inventor [SC92], dispatch events directly to objects to facilitate the diversity of object behavior.

To support diversity of object behavior, Bramble permits objects to specify hooks that are called when certain operations are performed on them. Most of these, such as the save hook, the draw hook, or the deleter hook are primarily useful for dynamically creating a new class of graphical object, as shown in Section 7.5.2. Certain applications permit objects to define specific event hooks and perform the dispatching themselves. For example, the Showoff application, described in Section 9.7, permits each object to define a method to be called when the right mouse button is clicked on it. The application defines the right mouse event handler to determine which object should be notified of the event, and to call the appropriate hook.

One set of standard event handling hooks that is particularly useful for defining interactive behaviors with the differential approach is the grab/ungrab pair. Unlike the example code of the previous section, the standard built-in drag handlers permit objects and specific points to specify hooks that are called before a dragging operation is begun and after it has ended. This is often used to apply constraints that should act during the dragging. Some examples of how this is used are provided in Section 8.3.6.

Object hooks can be defined on a per instance basis, permitting individual instances to each have their own behavior. Because hooks can be defined dynamically, the behavior of an object can be altered while the program runs.

7.6.3 Demons

A `Demon` is a special class of object that causes hooks to be called when certain conditions occur. The common condition for a `Demon` is that a signal is within a threshold of a desired value. Each `Demon` can define three main hooks:

do which is called when the condition the demon is looking for occurs. The demon removes itself after calling this hook.

fails which is called if the demon is removed before its condition is satisfied.

done which is called when the demon is removed.

`Demons` have an optional timeout which causes them to be removed after a specified number of steps, whether or not their conditions have been achieved.

`Demons` are useful for creating a variety of behaviors. One use of a `Demon` is to create a non-persistent constraint by removing a `GoTowards` controller after it achieves its goal. For example, to make example of Section 7.2 non-persistent so that

the cone can be freely manipulated after its top reaches the left edge of the table, we could

```
(set cl (controller (signal (c top) 'x) '= 5))
((demon) (bind done (lambda (x) (delete cl)))
         (bind tolerance .2)
         (bind lifetime 50)
         (bind possessed cl))
```

In this example, after creating the controller, a demon is created that "possesses" it. When the controller gets within tolerance of its target, or when the demon's specified lifetime of 50 steps times out, the demon's done hook is called, deleting the controller. Demons are also often used to remove constraints that are unable to be satisfied, permitting the application to give up.

7.6.4 Other Hooks

Much of the behavior of an application is defined by hooks. Most of these hooks are used for tasks not particular to the differential approach. For example, each `View` defines a hook that draws the background of the image when its screen is cleared. Any of these hooks *could* contain code that alters the set of controllers. However, a small set seems useful for defining interaction techniques. The set includes:

sub-step-hook is called before each call to the differential solver (e.g. each substep of the ODE solver). This hook could be used to create active set methods, although hooks do not have access to solver internals to access the Lagrange multipliers.

step-hook is called after each step. One common use of the step-hook is for periodically altering a `GoTowards` controller to achieve the effect of a `Follow` controller.

add-obj-hook is called each time a new graphical object is added to the system. This can be used to permit automatic registering of new connectors.

redraw-hook is called each time a view is redrawn. This is called once per view per step as all views are updated each step in Bramble. This is often used to provide feedback to the user by drawing an overlay.

7.7 Other Application Components

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.

7.7.1 Windows, Views, and Widgets

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` 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.

`Frames` and `Views` are first class objects in Bramble. In fact, `Views` always have connectors that provide the position of the mouse relative to the coordinate frame of the window. This simplifies attaching other connectors to the mouse.

A `Frame` can contain other subwindows besides a `View`. These are used to provide buttons and other widgets around the edges of the `View`. All behavior of the widgets is defined by hooks, including the widget's appearance. These subwindow hooks include methods for drawing, handling mouse clicks, and a "tick" function that is called periodically. Like the graphical objects, the predefined subwindow widgets are designed to automatically update themselves to maintain a consistent view of the values that they access. The provided set includes:

Buttons that watch the value of a Whisper variable, and update automatically.

Radio Buttons that watch a variable and allow its value to be selected from a set.

Differential Sliders that can attach to connectors and create controllers in response to mouse clicks. These sliders permit not only dragging a connector's value, but also nailing or bounding it, as discussed in Section 8.2.1.

Color Sliders that create a set of differential sliders for RGB values along with feedback for what the color is.

Text Elements that allow static display of information.

A special type of widget is provided to help watch the status of an object's state vector. The `VarWatcher` is a separate window that contains sliders for each variable in a particular state vector. The special sliders depict the gather state of the variables as well as permitting their values to be displayed and controlled. `VarWarchers` are particularly useful for debugging. A programmer can monitor an object's state with a single function call.

7.7.2 External Representations

Bramble provides rudimentary features for reading and writing representations of models outside of Bramble. For example, save and load is handled using Whisper, the saved representations are Whisper programs that recreate the model. Applications augment objects to write Whisper code to recreate themselves when reloaded. Objects have hooks that are called during the save process, allowing applications to specify this behavior.

Other external file representations are handled by Bramble. Scene descriptions for renderers, including Renderman [Ups89] and Rayshade [Kol91], can be generated, as well as PostScript pictures. In each case, objects define a hook that is called whenever the object needs to be written to a file. By defining this behavior as a hook, applications can alter the way that objects are written on either a per class, or even a per instance basis. Bramble also has support for writing bitmapped images of views to disk.

7.7.3 Picking and Snapping

Picking is a fundamental operation in any graphical editor. Bramble contains methods for easily using the standard Iris GL pick by redraw mechanism. Picking by ray casting is also supported for object classes that provide a ray intersection routine.

The most commonly used picking mechanism is cursor snapping. Cursor snapping continuously locates the cursor near important points in the view. It is used to aid in object selection and precision cursor positioning. Cursor snapping was first introduced in Sketchpad [Sut63], and has since been further extended to better support precision manipulation (as in Snap-Dragging [Bie89]), infer constraints (as in Briar [GW94]), or provide feedback to the user as to the range of available options [Hud90].

In Bramble, cursor snapping is handled separately from the event process. The model for cursor snapping is also continuous, but in practice it is updated only at the end of each step. Event actions can inquire about the state of snapping at any time. A single Whisper call returns the object currently snapped to. The snap server actually keeps a list of targets that are near the mouse, to permit hysteresis and cycling between objects that are close to one another. At present, the Bramble snap server snaps only to existing `DistinguishedPoint` connectors. It has been designed to be extended to support the range of snapping seen in Snap-Dragging [Bie90, BS86].

Facilities are provided to control the scope of snapping, permitting techniques such as Semantic Snapping [Hud90]. By using either a hook or status bits in each potential target, the snap server is able to cull objects based on criteria other than being close to the mouse. One common use of this is to limit snap-targets to legal choices for the current operation. Bramble automatically provides the user with feedback of the snapping state. Applications can provide more than just the cursor feedback, for example highlighting objects that are snapped to.

Constraint inferencing from snapping is supported by Bramble. By using snap-server accesses in object creation and manipulation event handler hooks, augmented snapping [GW94] can be implemented easily. Augmented snapping, discussed in Section 9.1, automatically generates constraints from snapping operations in order to make the positioning operation of the snap persistent. For example, the dragging handler of Section 7.6.1, might be extended to

```
(1)  (add-key k-leftmouse k-none k-down            ; left-mouse down definition
(2)    (lambda (view)                              ; attach a procedure, called with view
(3)      (if (snapdp)                              ; if there is a selected point
(4)        (let* ((p (snapdp))                     ; store selected point
(5)               (c (pt-eq-2d p (view mouse-port)))) ; attach point to mouse
(6)          (add-key k-leftmouse k-any k-up       ; redefine left-mouse button up
(7)            (lambda (v)                         ; now also infers constraint
(8)             (prog (delete c)                   ;     remove the attachment constraint
(9)                   (if (snapdp)                 ;   if there's a point to connect to
(10)                    (pt-eq-2d p (snapdp))))))))))) ; infer a constraint
```

The code has two differences from the earlier example. First, the point that is being dragged is stored in line 4. Secondly, when the mouse button is released, line 9 checks to see if the cursor is snapped to a point. If it is, a constraint is created in line 10 that connects it to the point that was dragged. In Section A.2, a code fragment demonstrates the addition of augmented snapping to the creation of a line segment by rubber banding.

7.8 The Bramble Standard 3D Interface

Bramble provides a standard set of objects and connectors to support basic 3D interaction. The goal is to provide a fast and easy interface for 3D experimentation. The basic interface, shown in Figure 7.3, has a particular style derived from systems built in the CMU Animation Lab over the past few years. 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[HZR+92]. 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 [Bli88a]. Bramble's shadows can either be simple drop shadows that are directly below the objects as if the light was the sun at high noon, or shadows computed from the positions of the light sources.

A `Follow` controller couples the motion of a connector to the motion of an input device. Any connector which represents a position in 3–space could be connected to a 3D input device. Bramble's standard 3D input device is a virtual device that uses the

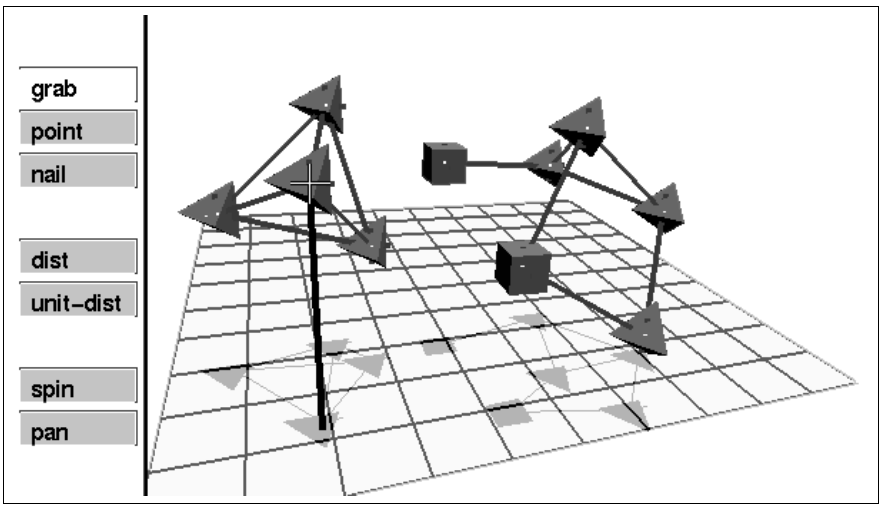

Figure 7.3: The *Point Tinkertoys* application demonstrates Bramble's standard 3D interface. The user manipulates the point objects using the mousepole. A groundplane and shadows are provided to help depth perception. This application is discussed in Section 9.6.

mouse along with a special graphical object called the *mousepole*.[3] 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 cursor location. As the mouse is moved, the pole tracks it, with the top point moving parallel to the groundplane. When a mouse button that has been designated as the "elevator" button is held down, the pole top moves in a vertical plane instead of a horizontal one. The tip of a mouse pole can be tracked by a controller.

A standard 3D interface package can be loaded from any Bramble application. It provides a standard set of key bindings and object manipulation techniques including mousepole manipulation of scene objects, virtual trackball-like manipulation of the viewpoint by grabbing the corners of the groundplane, and viewpoint panning by grabbing the center of the groundplane.

In addition to its standard 3D interface, Bramble provides support for developing other types of 3D interactions. Objects provide a variety of connectors which serve as building blocks for creating interaction techniques. Many of these will be described in Chapter 8.

[3] Although the mousepole is unpublished to date, I credit it to Andy Witkin.