The world is your exercise book, the pages on which you do your sums. It is not reality, although you can express reality there if you wish.
You are also free to write nonsense, or lies, or to tear the pages.

— Richard Bach
Illusions, p. 127

Appendix A

The Whisper Programming Language

An important tool in the development of the differential approach has been a simple interpreter for a language called *Whisper.* Whisper serves four main purposes in this work:

1. An embedded interpreter is important for interactive systems to realize features such as saving and loading of models and configuration files.

2. The language runtime provides useful tools for system construction, such as a dynamic object system and dynamic function definition.

3. The language provides a convenient notation for describing interaction techniques in this thesis.

4. The interpreter permitted avoiding many of the problems with the programming environment available to develop this work, such as slow turnaround and bad debuggers.

Whisper is not directly connected to the differential approach. However, it is discussed here for several reasons. Foremost, the language is discussed enough so that a reader can understand the examples written in Whisper in various sections of the thesis.

The Whisper interpreter is specifically designed for being embedded into applications. The primary design goal were simplicity and extensibility, even at the expense of performance. Although performance of the interpreter is poor, the ease of extensibility allows adding speed critical code as new primitives written in the host language, C++. New data types can also be added easily to the interpreter. Once the core interpretive language was built, extensions were created to provide access to the Iris GL graphics library, Snap-Together Mathematics, an image processing library (that is not used in this thesis), and Bramble.

A.1 Whisper Basics

Whisper is a lexically scoped variant of LISP, like scheme. Whisper is very similar to other languages in this family. A basic introduction to programming in such a language is provided by Friedman and Felleisen [FF87]. Rather than providing yet another tutorial for such languages, we instead look at the differences between Whisper and more common dialects that are relevant to the code examples in the thesis.

Like other LISP dialects, Whisper has dynamic types. All data are tagged with their type, and the types can be determined for any object. Whisper's type system is extensible at compile time only. The basic set of types provided by Whisper includes integers, floating point numbers, strings, pairs, points (3 floating point numbers), closures, built-in primitive functions, clocks (special objects for timing) and environments. Types for basic Snap-Together Mathematics classes are provided, as well as for Bramble objects. Almost all Bramble objects have the Whisper type `IDObj`, but the subtypes can easily be determined.

Whisper's special forms for setting variable values are different from most LISP dialects. Whisper provides a `set` function that sets the value of a variable. If the variable is defined in the current scope, it is simply rebound to the new value. If the variable is not defined, it is created as a *global* variable. The `defun` function used to define a function is simply shorthand for `set`. The `bind` function acts like set, but it always sets a variable in the most local scope.

Whisper is lexically scoped. Variable bindings are determined by the code that surrounds them in the program text. For example,

```
(let ((a 5))
    (defun f (x) (+ x a)))
```

defines a function that adds 5 to its argument. An *environment* is the object that embodies a scope, that is, it is a pairing of variable names and values. Environments are hierarchical, that is, each environment refers to the environment it was defined in. Unbound references are deferred up the chain of environments until the global scope is reached.

One important feature of Whisper is that it permits treating environments as first class objects. Many dialects of Scheme, such as MIT Scheme [HtM93], do this as well, and the introductory programming text by Abelson and Sussman [AS85] provides an introduction to the use of first-class environments. Whisper's syntax and operations for first-class environments is different from Scheme's.

The `environment` special function returns a reference to the current environment that can be passed around as any other data element. For example,

```
(set e (let ((a 5)
             (b 6))
         (environment)))
```

sets the variable `e` to an environment that binds `a` and `b`. Various operations can be performed on environments, including adding additional bindings, inquiring as to their contents, combining two environments, and printing the contents of an environment.

The two most important operations that can be performed are to evaluate an expression within an environment, and to bind a value in an environment. To add a binding to an environment, a special version of the `bind` primitive function is used that takes an extra argument for environment, for example,

```
(bind-in e c 10)
```

which would add the binding $c = 10$ to the environment defined in the previous example. A similar version of `eval` is provided,

```
(eval-in e (+ a b))
```

which for the example would return 11. Whisper provides the syntax

(*env* $expr_1$ $expr_2$ ... $expr_n$)

to evaluate expressions $expr_1$ through $expr_n$ in succession inside of the scope of environment env. The value of the evaluation of the last expression is returned. This construction violates lexical scoping, for example,

```
(let ((a 1)
      (b 2))
  (e (+ a b)))
```

with `e` defined as in the above example evaluates to 11, not 3. This can cause an important distinction between the use of `bind` and `bind-in` . For example,

```
(let ((a 10))
  (bind-in e f a)
  (e (bind g a))
```

creates bindings for `f` and `g` inside of `e`, however, `f` will be bound to 10, while `g` will be bound to 5, because its binding statement was evaluated inside of `e`.

First class environments function as Whisper's object system. Fields and methods are stored as bindings in the environment. The object creator makes an environment and binds variables to their initial configurations. While there is no explicit mechanism

to provide inheritance or delegation, such functionality can be provided in the creator functions.

A Bramble `IDObj` is not an environment, however each object carries an environment around. The `get-env` function returns the environment carried by an `IDObj`. Whisper syntax permits using the evaluation notation for Bramble objects as environments, so

(*IDObj* *expr*)

is a shorthand notation for

((get-env *IDObj*) *expr*).

A.2 Some Examples

In this section, we take a few examples of code fragments from Whisper programs to explain various features of the language, and how they are used with Snap-Together Mathematics and Bramble.

This first example defines a function that builds the function block graph for the line segments shown in Figure 5.1. The line segment is returned as a Whisper object, that is, as an environment. Notice that an environment is made first and then the intermediate values are defined in a different environment. This way the environment that will represent the line segment does not contain the intermediate values, only the connectors.

```
(defun make-line ()                                          ;
  (let* ((q (make-stobj 3))                                  ;state vector w/3 spaces
         (e (environment)))                                  ;empty "object" to put things in
    (let* ((c  (cos-block (signal q 2)))                     ;make blocks to compute
           (s  (sin-block (signal q 2)))                     ;   intermediate results
           (lc (times-block c (signal q 3)))
           (ls (times-block s (signal q 3))))
      (bind-in e left-x  (plus-block lc (signal q 0))) ;make connectors
      (bind-in e left-y  (plus-block ls (signal q 1))) ;   put the blocks directly into
      (bind-in e right-x (plus-block lc (signal q 0))) ;   the object
      (bind-in e right-y (plus-block ls (signal q 1))))
    e))
```

Calling this procedure returns an environment that contains 6 bindings: 4 for the connectors, 1 for the state vector and an extra binding that is a reference to itself. To use these objects to create the wiring of Figure 5.1, given a function to perform the

attach operation which would take 4 connectors as inputs and create the function block tree:

```
(set rod1 (make-line))
(set rod2 (make-line))
(set nail (attach (rod1 left-x) (rod1 left-y)
                  (rod1 right-x) (rod1 right-y)))
```

Attach would return an object (a Whisper Environment) that contained two connectors. It might be defined as follows:

```
(defun attach (x1 y1 x2 y2)
   (let ((c1 (minus-block x1 x2))
         (c2 (minus-block y2 y2)))
     (environment)))
```

The following code fragment is a more expanded example of constraint inferencing using Bramble's snapping than the one given in Section 7.7.3. It is a Bramble event handler that is used to draw lines with rubber banding and automatic inference of constraints.

```
(1)  (add-key dev-rightmouse k-none k-down              ; define button down handler
(2)    (lambda (v)                                      ;
(3)      (let* ((s (snapdp))                            ; get snap point
(4)             (p (if s (where s) (cursor-mapw view))) ; start drag at snap or mouse
(5)             (l (make-2d-line (p-x p) (p-y p)        ; create object at start
(6)                              (p-x p) (p-y p)))      ;
(7)             (d (pt-eq-2d (l end1) (v mouse-port)))) ; connect one end to mouse
(8)        (if s (pt-eq-2d (l end2) s))                 ; if snap, infer constraint
(9)        (add-key dev-rightmouse k-any k-up           ; define button up handler
(10)         (lambda (v)                                ;
(11)           (let ((s (snapdp)))                      ; get snapped point
(12)             (delete d)                             ; delete drag constraint
(13)             (if s (pt-eq-2d (l end1) s))))))))     ; if snap, infer constraint
```

This fragment is a call to `add-key` , the Bramble primitive for defining key handling events. The first three arguments specify that this call is to define the right mouse button press (down) event with no modifier keys. The last argument to the call is a closure that is to be called with one argument when the event occurs. The argument specifies the view that the event occurs in.

Lines 2-12 define the procedure that is called when the right mouse button is pressed. First, a number of local variables are bound to various quantities useful in the operation: `s` is bound to the current state of the snap-server; `p` is bound to either the position of `s`, if there is a point snapped to, or to the position of the cursor; `l` is bound to a newly

created line segment, created with both endpoints at `p`; and a constraint is created connecting one endpoint of `p` to the position of the mouse in the current view and stored in `d`.

Line 7 performs a constraint inference. If the cursor is snapped to a point when the line is created, the end of the line is connected to that point with a point equality constraint.

Lines 8-12 redefine the event handler that is called when the right mouse button is released. Because of Whisper's lexical scoping, this code is executed in the environment created by the key down handler and has access to those local variables. It first deletes the constraint that was connecting the line segment to the cursor. If the cursor was snapped to a point when the button release occurs, the line segment is attached to this point in line 12.

A.2.1 The Define-Shape Syntax

A special facility is provided in Bramble to facilitate the creation of 2D shapes. The process is inspired by the shape spreadsheets in the Visio drawing program [Sha93], but has greater utility because it permits placing constraints inside objects and using any point as a handle. The `define-shape` special function takes a description of a shape and automatically generates two functions: one that creates instances of the object, and another that draws a prototype version of the shape suitable for displaying in an icon. The `define-shape` primitive is special because it does not evaluate its arguments in the standard way.

The syntax of `define-shape` are as follows:

(define-shape $name$ $variables$ $defaults$ $lets$ $command_1$ $command_2$ ...)

where:

name is a string that names the type;

variables is a list of state variable names;

defaults is a list of initial values for each variable;

let is a list of let pairs to define internal variables;

command is a pairing of commands and data.

For example, a simple rectangle can be defined by:

```
(define-shape rectangle (w h) (.5 .25)
    ((w2 (/ w 2))
     (h2 (/ h 2))
     (mw (- 0 w2))
     (mh (- 0 h2)))
    (spath ( (w2 h2) (mw h2) (mw mh) (w2 mh) )))
```

All shapes are defined in their local coordinates, so the rectangle only has 2 state variables, width and height. Default values for these are provided. The let list defines 4 local variables. The `spath` command defines a list of vertices that are connected to draw a polygon, with handle points placed at each vertex. The command is equivalent to separate commands to define a drawing function and handle points. Literally, the definition used for drawing is also used for manipulation.

The `define-shape` mechanism makes extensive use of Whisper first-class environments to make the concise specification possible. Each variable declaration and let list clause defines a new symbol in the objects local environment. Each of these is bound to a signal: when the `define-shape` is executed, it is run in a special environment that shadows all of the basic arithmetic operations with their Snap-Together Mathematics block generating counterparts. Because the drawing routines do not execute during the `define-shape` , they can do normal arithmetic.

Other commands permit more general drawing commands, explicit specification of handle points, and generation of constraints on the object. The fuel-gauge widget of page 162 demonstrates many of the features of `define-shape` . Here we present a more complete version:

```
(1)  (define-shape gas-gauge (sz theta) (.35 0)            ; 2 params
                                                           ; local variables
(2)    ((l (* (one-way sz) .9))                            ; size
(3)     (t (+ .2 (* 2.7 theta)))                           ; angle (in radians)
(4)     (x (- 0 (* l (cos t))))                            ; positon of needle
(5)     (y (* l (sin t))))
                                                           ; draw function
(6)    (drawf (prog (color gl-white) (arcf 0 0 sz 0 1800)  ; white arc for back
(7)                 (color gl-black) (linewidth 3)         ; black border
(8)                 (arc 0 0 sz 0 1800)                    ; arc border
(9)                 (move (- 0 sz) 0) (draw sz 0)          ; bottom border
(10)                (icon-font)                            ; set text font
(11)                (cmov (* l -.9) (* l .2)) (fmprstr 'E) ; place labels
(12)                (cmov (* l .8) (* l .2)) (fmprstr 'F)  ;  for E and F
(13)                (move 0 0) (draw x y) ))               ; draw needle
                                                           ;
(14)   (> theta 0)                                         ; limit to legal values
(15)   (< theta 1)                                         ;
(16)   (handle x y))                                       ; create handle on needle
```

The gas gauge has 2 parameters, one for the size of the gauge, and one for its current value. The let clause of the `define-shape` defines a number of intermediate variables using arithmetic operations. One special operation is `one-way` that defines that controls on its output should not effect its input. The `one-way` function block returns the value of its input, but always returns a zero derivative. For the gas gauge, the one-way is used so that dragging the needle of the gas gauge does not change the size of the gauge. The `drawf` command defines a code fragment that is used to draw the gauge. The list of statements are calls to the GL graphics library. The > and < commands define constraints, and the `handle` command creates a `DistinguishedPoint` connector at the specified position on the object, which in the example is the tip of the gauge's needle.

The `install-shape` function is defined by certain applications to automatically install an icon for creating shapes defined with `define-shape` . It simplifies using `define-shape` in an application. The `install-shape` function takes the values returned by `define-shape` and creates an icon and the proper handlers for creating new instances of the shape.