The Design of a Graphics Subsystem for Boxer

by

Jeremy Roschelle

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements
of the Degree of Bachelor of Science
in Computer Science

at the

Massachusetts Institute of Technology

MAY 1985

Signature of Author . . . . . .

Department of Electrical Engineering and Computer Science

Certified by . . . . . .

Thesis Supervisor

Accepted by . . . . . .

Chairman, Departmental Committee

The Design of a Graphics Subsystem for Boxer

by

Jeremy Roschelle

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements of the Degree of Bachelor of Science in Computer Science on May 17, 1985.

Abstract

This thesis discusses the design of a graphics subsystem which has been implemented as part of Boxer. This graphics subsystem, which is called Sprite Graphics, generalizes and improves upon Turtle Graphics. The primary goals of this new design are 1) compatibility with Boxer, and 2) utility for children and non-technical computer users. Sprite Graphics gives these users the ability to define object shapes and sizes, to make objects mouse sensitive, and to build composite objects out of simpler ones. Significant innovations of the design include making the state of all graphics objects transparent, and allowing the construction of drawings which can be displayed at different levels of detail. Examples include a clock, a calculator, a sketchpad, and icon programming system show the ease with which complex graphical objects can be created using Sprite Graphics. Suggestions for futher research are included.

Thesis Supervisor: Dr. Andrea diSessa

Acknowledgments

I would like to thank Andy diSessa for his ideas, his advice, and especially his enthusiasm for this project -- it has been not only a great learning experience but also a lot of fun.

I would also like to thank Hal Abelson for his help throughout the past year, both on this project and with other matters.

Special thanks are due to Ed Lay for teaching me about the Boxer implementation and for taking on some of the hardest bugs. Similiarly, Roland Ouellette deserves much thanks for supplying the answer to every question of the form, "On the Lisp Machine, how do I...?"

Finally thanks to my family, to Maria Daehler, and to the members of Boxer Group for being both great test subjects and the source of much support.

Dedicated to my parents

Table of Contents

Table of Figures

Chapter One

Introduction

This thesis presents the design of a graphics subsystem which has been implemented as part of Boxer, a new programming language/environment. This design will be integrated into the first official release of Boxer. Boxer is being developed by the Educational Computing Group at M.I.T.'s Laboratory for Computer Science.

1.1 Sprite Graphics and Turtle Graphics

The design philosophy for the Boxer language is firmly rooted in the design philosophy for Logo; that is, Boxer attempts to give the non-technical user in general, and children in particular, a powerful, expressive, understandable, and useful computational environment. The motivating force behind the development of Boxer is a desire to exploit recent technological advances -- primarily the bit-mapped screen -- in order to give these users a personal computing system which is more powerful, more understandable, more expressive and more generally useful than Logo. The designers of Boxer plan for Boxer to encompass the functionalities most commonly needed by non-technical computer users in one consistent, uniform, and relatively simple interface. Boxer currently supports a text editor, a mail system, and a programming language. Just as Boxer aims to improve upon Logo, so does the graphics subsystem described herein, which I shall call Sprite Graphics, aim to generalize upon and surpass the usefulness of Turtle Graphics.

The primary design goal behind Sprite Graphics is to make a good thing, Turtle Graphics, better. Logo's overwhelming success in schools and homes -- it is one of the most popular personal computer languages -- provides a strong indication of the value of Turtle Graphics, because Turtle Graphics is the centerpiece of every Logo implementation. Turtle Graphics succeeds for several reasons. Firstly, Turtle Graphics gives the novice computer

user an object, the turtle[1], which he or she can understand and relate to. As documented in Seymour Papert's book, Mindstorms[5], a turtle is an effective motivational tool: Because a learner can easily manipulate the turtle to produce aesthetically pleasing designs, he or she is naturally lead to explore the computational medium. In fact several Logo tutorials, for example Apple Logo[1] by Harold Abelson, use Turtle Graphics to introduce programming.

Moreover, Turtle Graphics is especially rich in mathematical and physical concepts, as well as many powerful ideas about problem solving. In fact, in Turtle Geometry[2], Harold Abelson and Andrea diSessa showed how the activity of playing with Turtle Graphics could help the learner understand a broad range of deep and complex ideas chosen from Mathematics and Physics. In these formal and abstract domains, the turtle acts as a bridge from familiar, concrete, and intuitive ideas to difficult new concepts. In short, Turtle Graphics succeeds because it gives the non-technical user an object to learn with, both at introductory and advanced levels.

Sprite Graphics expands upon Turtle Graphics by giving the non-technical user an more sophisticated object, the "sprite." Since simple sprites look and act just like turtles, Sprite Graphics inherits all of Turtle Graphics good properties as a learning tool. But Sprite Graphics also surpasses its predecessor in several ways.

1.2 Beyond Turtle Graphics Functionality

As later chapters will show, sprites are not only good learning tools, but also are useful as building material for new kinds of interfaces, objects, and programs. For example, the Symbolics 3600 supports a special device called a "mouse" which allows the users to point anywhere on the screen. The mouse also has three buttons which the user can press in order to cause the computer to do something. In Sprite Graphics one can create sprites which can carry out an user-defined action when the user points to them and presses a mouse button. This functionality is not difficult to use: in several cases, I have seen non-programmers learn to build simple mouse sensitive sprites in their first session with Boxer.

---

[1] A turtle is a triangularly-shaped object which draws of a computer screen in response to commands like "Forward" and "Right"

Sprite Graphics makes it possible to build many other kinds or useful graphics objects as well. Some examples which will be discussed later on include a calculator, a clock, and a sketchpad. In the case of the calculator, the Boxer version looks and operates just like a real hand-held calculator. However instead of pressing keys, the user simply points to the keys he'd like to press with the mouse. (On the touch screen version of Boxer, one can actually touch the keys!) The user can customize this calculator by adding to the Boxer code, so he or she can add extra keys and functions as needed. This calculator serves not only as a useful object, but also as a understandable model for the way real calculator works.

Sprite Graphics, furthermore, makes it possible to create programming languages in Boxer which look radically different from Boxer. Chapter Four will exhibit one alternative programming system, which is based on the manipulation of icons. In contrast to programming in ordinary languages, programming in this icon system consists of concretely arranging symbolic graphics objects into columns.

1.3 Understandability

One of Boxer's major design goals is to make the state of the system transparent -- that is, both visible and understandable. This goal leads directly to one primary accomplishment of the Sprite Graphics design: The state of the Sprite Graphics system is not only visible, but is also easily understood and modified by any user who is familiar with Boxer. Although Sprite Graphics allows users to manipulate sprites in the same way as they would as turtles in a Turtle Graphics, Sprite Graphics also allows the user to treat each sprite as a concrete Boxer-object, a Sprite Box. One can, for example, look inside a sprite's box, find its HEADING state variable, and directly edit the value of the variable, changing the orientation of the sprite on the screen. Moreover, the user can carry out many operations concretely and physically, instead of by using side-effecting commands. To add a sprite to a graphics screen, for example, the user simply uses the Boxer editor to move a Sprite Box, which is the logical representation of a sprite, into a Graphics Box. Thus Sprite Graphics aims to improve upon not only the functionality of Turtle Graphics but also upon the understandability.

1.4 Beyond Its Logo Roots

This above should give the reader general sense of goals of this thesis in terms of an improvement to Logo and Turtle Graphics. But Sprite Graphics subsystem also attempts to come to terms with the more general question "What should the graphics package of a future personal computer do?" While no attempt is made herein to produce a sound theory of what a graphics system should be able to do, the Sprite Graphics design incorporates many useful functionalities not found in Logo.

The design of Sprite Graphics draws inspiration from several of non-Logo environments. For example, the Apple MacIntosh shows the popularity of pull-down menu systems among personal computer users. Sprite Graphics aspires to become the agent of this functionality within Boxer by supplying suitable primitives to allow the construction and use of mouse sensitive menus. Another inspiration for Sprite Graphics is the "Illustrate" program found on the Symbolics Lisp Machine, which allows the user to compose drawings on the computer. In Sprite Graphics, one can construct special sprites which allow the user to place a geometric objects of any shape and size in a drawing, thus capturing the functionality of Illustrate. Interestingly, Sprite Graphics manages to incorporate both these seemingly different functionalities into one relatively simple object.

1.5 New Uses of Sprite Graphics

Sprite Graphics also attempts to provide graphics capabilities for several proposed experimental applications. For example, one persistent image which motivated the design of Sprite Graphics is the image of an interactive textbook written in Boxer. In this application, Sprite Graphics would allow graphics to be presented side-by-side with text, serving the same purpose that illustrations do in conventional textbooks. Sprite Graphics, however, has the potential to move beyond the value of an illustration by allowing the teacher to insert simulations and even Microworlds into the text. Moreover, with Sprite Graphics, the student could not only observe these simulations, but could also examine the logic which governs their operation. The student could even modify the workings of the simulation to customize it to his or her needs.

Another experimental functionality provided in Sprite Graphics is the ability to build graphics objects out of modular components. Like Boxer boxes, a composite graphics object can be constructed in Sprite Graphics so that it can be displayed either expanded, with all its components showing, or contracted, with only its abstract shape. Using this feature, one can imagine creating a drawing of an electronic circuit schematic in Sprite Graphics in which each component could be expanded into more detailed views. With this sort of schematic, a student could examine a circuit at different levels of detail without losing sight of the circuit as a whole.

1.6 Looking Ahead

So Sprite Graphics is many things: It is firstly a generalization of Turtle Graphics which aims not only to expand the functionality of Turtle Graphics, but also to improve the understandability and usability of the this style of drawing. Secondly, Sprite Graphics attempts to provide the opportunities to incorporate the useful qualities of non-Logo computer systems into Boxer. Finally Sprite Graphics will allow the exploration new ways of working with graphics on a computer.

The next chapter will discuss the goals which motivated the Sprite Graphics design. The following chapter will show how the design looks and works in current implementation. Chapter 4 presents several existing applications of Sprite Graphics, in order to give the reader an idea of the system's capabilities. The final chapter evaluates the Sprite Graphics design with respect to its goals and also suggests some areas for future research.

Chapter Two

Goals

This chapter discusses the goals which motivated the design of the Sprite Graphics both in terms of the Boxer Environment in which Sprite Graphics is embedded and in terms of general goals inherent in its primary purpose, to serve as a graphics system for school children.

2.1 The Boxer Environment

Because Sprite Graphics was developed specifically for Boxer, compatibility with Boxer is the system's major design goal. So that the reader may understand this basis for Sprite Graphics, I will briefly describe the features of Boxer in a few paragraphs below. For more detailed information about Boxer please see the Boxer Users Manual[4]. The reader who is familiar with Boxer can skip to the next section, which discusses the goals Sprite Graphics inherits from Boxer.

Boxer is a computer system for non-technical users. It is designed to be simple enough for children to use. Though Boxer now runs on a Symbolics 3600 Lisp Machine, the Educational Computing Group intends for it eventually for it to be inexpensive enough so that it can become the general-purpose personal computer language of the future. Boxer aims to encompass all the computing needs of the home or school computer user. The applications it supports include word processing, electronic mail and programming.

Boxer is able to satisfy a wide variety of functions without becoming incomprehensible for several reasons. Firstly, Boxer exploits one metaphor -- that of its namesake, the box -- extremely effectively in organizing its environment. In Boxer, the user works directly with boxes displayed in two dimensions on a bit-mapped screen. The box metaphor goes a long way in making the internal structure of the Boxer system both visible and understandable.

Secondly, the Boxer environment is "editor-top-level." This means that all Boxer objects can be directly edited with the same Emacs-style editor without ever changing modes. Because the user always interacts with Boxer through this editor, the user must learn only one way of working with the computer to gain access to all its functionality. Thirdly, Boxer obeys the principle of naive realism. Naive realism means that the user always works with the system as it is displayed on the screen, as if he or she were directly looking at and modifying the state of the system. Because of naive realism, the user need only to understand the information presented on the screen in order to understand the state of the Boxer system.

Boxes in Boxer deserve a little more explanation. Boxer supports two basic kinds of boxes, the Data Box and the Do-it Box. Data Boxes may contain all kinds of textual information. For example, when using Boxer as a word processor, the user types text into a Data Box. Data Boxes can also contain data for programs, like numbers and lists. Do-it Boxes contain code for programs.

Boxer boxes have several interesting properties. Firstly, they can be named. A named box in Boxer acts is considered to be a definition. A named Do-it Box, therefore acts like a procedure definition; it can be function-called by name. A named Data Box in Boxer, on the other hand acts like a variable and can be referenced by name. Secondly, Boxer boxes can appear expanded or shrunk. By shrinking boxes, the user can hide details he or she is no longer concerned with, while retaining the ability to expand the box later. Thirdly, boxes can be hierarchically nested in Boxer. Boxer applications can take advantage of this nesting in many ways. A file system, for example, can use nested boxes to show its hierarchical directory structure. A Boxer book could also use nesting, boxing its chapters and the sections within the chapters, to allow browsing through its contents. Boxer additionally uses the nesting of boxes in a more profound way -- to show the nesting of namespaces. Each box in Boxer is in fact an environment in which definitions can be locally bound. Therefore, the box structure displayed on the screen becomes in Boxer a spatial metaphor for the inheritance and modularity of namespaces.

Sprite Graphics also makes use of one other kind of Boxer box, the Port Box. Port Boxes are used in Boxer to implement sharing of data, in the same way pointers do in traditional computer languages. A Port Box never has contains data of its own. Instead the

port shares the contents of some other Data Box, called its target. Examining the contents of a port, therefore, is equivalent to examining the contents of its target. Moreover, if one changes a port's contents, the value in the target Data Box changes simultaneously.

2.2 Compatibility with Boxer

What does compatibility with Boxer require? Firstly, since the only objects in Boxer are boxes, any new objects introduced to support graphics should also be boxes. Boxer provides standard, uniform ways of interacting with boxes using both the editor and the mouse. To maintain consistency, any new kinds of boxes should work with the standard Boxer interface in the way that the user expects.

Secondly, any new constructs added to Boxer should preserve the viewpoint of naive realism. As mentioned previously, naive realism means that the user can treat the information on the screen as if he or she was directly looking at and modifying the state of the system. Naive realism requires that the user can see directly see and edit the parameters which affect the behavior of the system. Applied to Sprite Graphics, naive realism implies that the user must have access to the important state variables of all graphics objects.

Thirdly, Boxer incorporates a number of important presentational innovations. The user in Boxer has a great deal of control over the information presented on the screen. The user can, for example, shrink boxes to hide details and can arrange the placement of definitions within a box according to personal whim. Moreover, the presentation which the user sets up does not affect the operation of the language -- one can execute a shrunk or expanded Do-it Box equivalently, for example. Another design requirement for Sprite Graphics is to carry over this presentational flexibility.

Fourthly, the goal of compatibility with Boxer requires that Sprite Graphics fit into the hierarchical structure which pervades Boxer. Every Boxer object is uniquely located within one containing box, which in turn is uniquely located within a hierarchical nesting of boxes. Boxer shows the user this structure using a spatial metaphor: boxes which are logically nested appear spatially nested on the screen. Maintaining the consistency of Boxer's structure implies that sprites too should have a single location, and that this location should be displayed using spatial metaphor.

Fifthly, Boxer applications tend to have a characteristic which I shall call "explorability." Explorability comes into play whenever a user works with an application written by another person. Explorability means that the user can always "open up and look inside" the logic behind an application to get an understanding of how it works. This is made possible in Boxer firstly because the source code is always available, and secondly because the user can browse through the source code simply by expanding the boxes which contain it. Explorability makes Boxer more learnable because it allows the novice to learn Boxer programming from examples of Boxer programming. Explorability also makes Boxer programs easy to customize, because the user can modify any part of a Boxer program he or she is looking at. In order to carry over this kind of explorability, Sprite Graphics should allow the logic behind its applications to be opened up, looked at, and modified.

2.3 General Philosophy and Goals

Many of the goals of the Sprite Graphics design have been mentioned in passing in the preceding section. Before describing the Sprite Graphics design, I would like to make these goals explicit.

The main purpose of Sprite Graphics is to allow children to draw graphics in Boxer. If the purpose of this design were just to allow children to draw graphics, the task would be easy: it would consist of supplying Boxer with a few of the Draw-line type commands typically found in Basic. Instead this design attempts to give children something that is more than just another set of commands to learn.

In Mindstorms[5], Seymour Papert describes a kind of computational environment he calls a "Microworld." A Microworld allows children to learn in a natural, intuitive way by allowing them to directly experiment with the subject matter. To illustrate how learning in a Microworld occurs, Papert gives an example of how a child, working in a Physics Microworld, learns Newton's Laws of Motion by playing with a turtle which acts in accord with the "Turtle Laws of Motion." In this style of learning, children acquire general intellectual skills and concepts while playing with the elements of the Microworld in open-ended projects. One purpose of Sprite Graphics is to support and encourage this type of activity in Boxer.

2.4 The Needs Of Children And Microworlds

The goal of developing a graphics system which allows children to work in this style carries with it a fairly stringent set of requirements. Firstly, the graphics system must be understandable. Since understanding how a computer works is a secondary goal for most children, the system must be simple enough for a person with no technical background to understand and use. Even a novice user should be able to make sense of the information presented on the screen and should, moreover, be able to manipulate it according to their own purposes without much instruction.

Secondly, the graphics system should be learnable. In comparison to a computer professional, a child will be less likely to invest large amounts of time and energy to learn the use of a computer system. Therefore, child's learning process for a computer system be incremental; that is, the learning should come in small chunks with rewards at each stage. In addition, a system will be easier to learn if it follows a consistent set of conventions. Such uniformity allows the user to apply previously acquired knowledge and expectations to each new area of the system which is encountered.

In addition to these two basic goals, a graphics system should embody several other characteristics which are common to any good computational environment.[2] For instance, the environment should allow a variety of workstyles. It should also support activities which have a strong constructive component. Furthermore, the environment should take advantage of things the student already knows. Finally a good graphics system should invite activities which are both stimulating and fun.

[2]The ideas in this paragraph are taken from a talk given by Dr. Sylvia Weir in May, 1985 at M.I.T.

2.5 Utility

One concept which embraces most the goals of this design, is the concept of Utility. In The Future of Programming[3], diSessa states that greater utility is the most important goal for future personal computer environments. The meaning of utility in case is close to its economic meaning: a graphics system should have a high ratio of perceived value to perceived cost. Therefore the most general concern behind the design of Sprite Graphics is to create a system in which the user can not only do wonderful things, but can do them with minimal effort.

Chapter Three

Design

This chapter will explain the Sprite Graphics design.

3.1 New Boxes

Sprite Graphics adds three new kinds of boxes to Boxer, Graphics Boxes, Sprite Boxes, and Graphics Data Boxes. Each box fulfills a necessary role in the graphics system. Graphics Boxes provide a place in Boxer for graphics to be drawn. Sprite Boxes serve as the Boxer representation for the objects which draw graphics. Graphics Data Boxes provide a means for connecting Sprite Boxes to Graphics Boxes.

In many ways, each of these new box types is very similar to a regular Boxer boxes. Sprite, Graphics, and Graphics Data Boxes can, for example, be shrunk or expanded. They can also be moved around using the standard Boxer editor. Like other boxes, they can be named and then referenced by name. In the case of Sprite and Graphics Data Boxes, both of which hold textual information, the contents of these boxes can be examined and changed in the standard Boxer way. In addition, one can type and execute code in them. One can also create inferior boxes of any kind within a Sprite or Graphics Data Box. Moreover each instance of these new boxes, just like other boxes, supports its own namespace in which procedure definitions and variable declarations can be bound.

However, each of these new box types is also necessarily different from all the other box types so that it can fulfill its role in the Sprite Graphics system. In the following sections each new box type will be discussed in more detail.

3.2 Sprite Boxes

As the section on Boxer explained, the only objects which exist in Boxer are boxes. Since a goal of Sprite Graphics is to support Turtle Graphics, Sprite Graphics must add new kind of object into Boxer, one which corresponds to a turtle. However, rather than simply introducing Logo turtles into Boxer, this design introduces a new kind of box, the Sprite Box which serves as a representation for turtle objects in Boxer. (To avoid confusion between Boxer and Logo, I call the object associated with a Sprite Box a "sprite" rather than a turtle.)

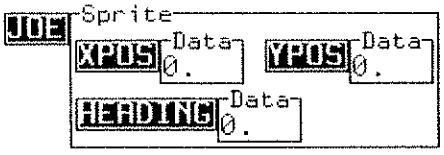

Figure 3-1: A Sprite Box

Figure 3-1 shows an example of a Sprite Box. To get a Sprite Box like this one in Boxer, the user presses a special key. In this case, the user has attached a name "JOE" to the Sprite Box using the standard Boxer naming procedure. The word "Sprite" which appears across the top edge of all Sprite Boxes differentiates these boxes from other boxes. Note that this Sprite Box contains three Data boxes named XPOS, YPOS, and HEADING. The numbers in these boxes correspond to the sprite's x and y position and its heading. The meaning and use of these boxes will be explained in more detail later.

3.2.1 Basic Turtle Graphics Functionality

A child using Boxer can interact with Boxer sprites using standard Turtle Graphics commands. Suppose a child has created a Sprite Box named "JOE" within a particular Graphics Box. Then to make this Sprite move, the child can type:

```
TELL JOE FORWARD 10
```

When the child executes this line the sprite JOE will move forward 10 sprite steps, drawing a line as he moves. This command really has two parts. The first half, TELL JOE, specifies which sprite will carry out the command. The second half can be any Boxer function. Alternatively, the child can specify a Do-it box containing a list of commands as the second half of a TELL command. Figure 3-2 shows a use of TELL in this format.

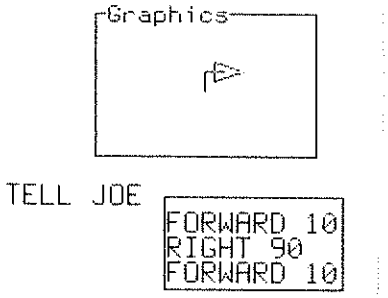

Figure 3-2: Using TELL with a Do-it Box

Other standard Turtle Graphics commands which sprites can carry out are:[3]

```
RIGHT [angle]
LEFT [angle]
BACK [steps]
CHANGE-XY [new xpos] [new ypos]
CLEARSCREEN
HIDE
SHOW
PENUP
PENDOWN
PENERASE
PENXOR
```

In a typical session a child would type these several of these commands in Boxer. Though the commands can be executed immediately after being typed, the child could also point at any command on the screen at any time and re-execute it. Moreover, the child could use Do-it boxes to build procedures out of these commands.

3.2.2 The Meaning of TELL

As the previous section illustrates, the TELL command is Boxer's way of directing a command to a particular object. In its simple use Boxer's TELL looks a lot like TELL in Logo. However the semantics are very different.

When inside a Sprite Box, the user need not type TELL because all sprite commands executed in a Sprite Box can be unambiguously directed to one sprite. However when outside a Sprite Box, the intended object of a command is unclear. Therefore, while the user can type sprite commands without prefacing them with a TELL from within a Sprite Box, the user must use TELL from outside a Sprite Box.

---

[3] The words in square brackets in these commands stand for numeric arguments which the user must supply to use the command.

The meaning of TELL in Boxer ties in perfectly with this distinction between being inside versus being outside a Sprite Box. Let's call the first input to the TELL command "WHO" and the second input "WHAT". TELL in Boxer means enter the environment of WHO and evaluate WHAT. Thus when the user executes the line "TELL JOE FORWARD 10", FORWARD 10 will be evaluated in the Sprite Box named JOE and will therefore be carried out by JOE's sprite.

3.2.3 State Variables

A Sprite Box stores all the state information required to display and manipulate its activity. You can think of this information as the logic behind the picture which would appear in a Graphics Box. A Sprite Box records its state information in a few specially named Data Boxes which it contains. Because Boxer represents all variables as named Data Boxes, the Data Boxes which hold state information may justly be called "state variables."

Recall Figure 3-1. In this figure the Data Boxes named XPOS, YPOS, and HEADING all are state variables. Sprite Graphics continually updates the values in these boxes, keeping them consistent with the sprite's actual position and heading. Because the user can directly examine any of these boxes, the current state of any sprite is completely transparent. Moreover, the user can reference any of these variables by name, in the same way he or she can reference any Boxer variable. The following line, for example, makes the JOE move forward a distance proportional to his heading:

```
TELL JOE FORWARD 10 * HEADING
```

The user can also modify any state variable using the standard Boxer command, CHANGE. The next line will set JOE's HEADING state variable to ten:

```
TELL JOE CHANGE HEADING 10
```

So state variables, like any Boxer variables, can be refered to by name. In addition, the user can directly change the value of any state variable with the Boxer editor. To change HEADING, for instance, the user need only to enter the HEADING box, then delete the old value and type a new one. As soon as the user exits the box, the pictured sprite will turn to the new heading.

One can even create a port to a state variable. In this case the port as well as the

variable will be continually updated to reflect the sprite's state. Moreover, changing the value in the port will change both the value of the state variable and the sprite's picture.

3.2.4 Additional State Variables

Sprite Boxes also can contain several other state variables. Boxer displays each of these state variables, like the first three, as a named Data Box within a Sprite Box.

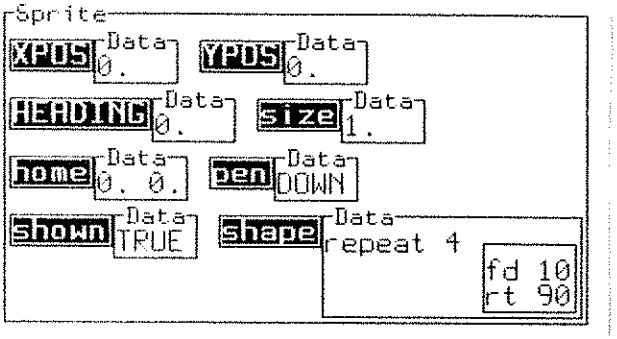

Figure 3-3: A Sprite Box Containing All The State Variables

The state variable PEN controls how the sprite draws as it moves. The pen Data Box can contain one of the four words UP, DOWN, XOR, or ERASE. DOWN is the default. With its pen down, a sprite will draw in black on the white screen. With PEN value of UP, a sprite does not draw at all. ERASE means the sprite will draw in white, erasing any black areas it passes over. When its value is XOR, a sprite draws in exclusive-or mode, switching black to white and white to black.

The state variable SHOWN determines whether the sprite is visible. A value of TRUE, the default, means the sprite is showing. A value of FALSE means the sprite will not be visible.

The state variable HOME contains two numbers. On encountering a GO-HOME command, the sprite will move to the point these two numbers describe. The default home in the center of the Graphics Box.

The state variable SIZE holds a positive number which sets the size of the sprite's shape on the screen. The default size is one.

The state variable SHAPE contains sprite commands which, when executed, will cause a sprite to draw a picture. A sprite's shape is the picture which the sprite would draw upon

executing the commands in its SHAPE Data Box. For example, the picture of sprite in figure 3-3 will be a square because the commands in that sprite's SHAPE box draw a square. The default shape for a sprite is the triangular shape used for a Logo turtle. Figure 3-4 shows what each of these shapes actually looks like.

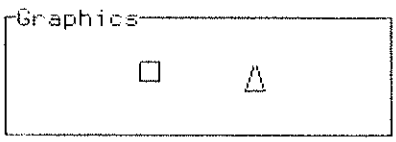

Figure 3-4: A Square Sprite and a Turtle-shaped Sprite

3.2.5 Controlling the Appearance of State Variables

Figure 3-3 shows a Sprite Box containing a full set of state variables. Needless to say, a box containing this many things is potentially very confusing to the novice user. Moreover, most beginning users will probably not need all the state variables. Of course, the user may shrink any of these boxes to hide the details, but this still leaves a crowded box. To combat this problem, Sprite Graphics makes the appearance of state variables optional.

In Sprite Graphics, the user may delete any state variable using the Boxer editor. When a state variable is deleted it reverts to its default value. The sprite in the case still maintains a state, however the state can no longer be changed or accessed by the user.[4]

One can also re-insert a state variable. To do this the user simply creates a new Data Box, enters an appropriate value in it, and then names the box with the name of the desired state variable. Once the user exits a state variable created in this manner, it will act exactly as if it had always been in place.

[4] This rule has an exception: All the sprite commands in Section 3.2.1 will work whether or not appropriate state variables are present.

3.3 Graphics Boxes

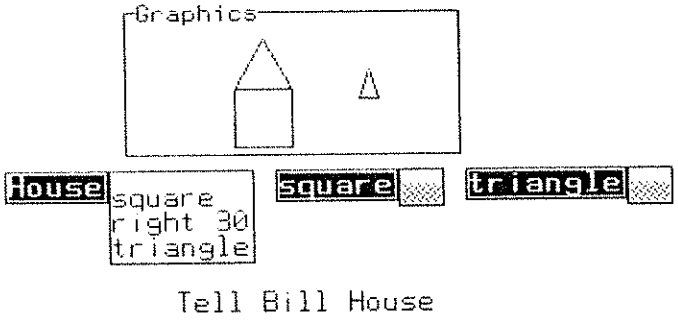

Figure 3-5: Graphics Box With House

3.3.1 Purpose

The purpose of Graphics Boxes is to delineate a region in Boxer which can hold graphical information. Many Graphics Boxes can be on the screen at the same time. To create a Graphics Box, the user presses a special key which inserts a Graphics Box at the current cursor location. Graphics Boxes can be treated like any other Boxer object. The user can shrink or expand one and can move them around with the Boxer editor. The role of Graphics Boxes is analogous to the role of Data Boxes: Just as a Data Box provides a place for the user to type text, a Graphics Box provides a place for the user to draw graphics. To draw on a Graphics Box, the user creates a sprite, attaches it to the Graphics Box and then gives the sprite commands which will cause it to move across the Graphics Box and draw.

Figure 3-5 shows a Graphics Box. The triangular object near the right side of the box is a sprite. I drew the house shown in this Graphics Box with this sprite, using the line of code shown below the Graphics Box, "TELL JOE HOUSE." This figure also shows the definition of the House procedure, which relies on two other definitions, Square and Triangle.

3.3.2 Commands

Graphics boxes accept three commands, CLEARSCREEN, WRAP, and WINDOW. The first of these, CLEARSCREEN erases the drawing in a Graphics Box, without erasing the pictures of any sprites contained in it. WRAP and WINDOW set the drawing mode for sprites in the Graphics Box. When a sprite goes past the edge of a Graphics Box in the former mode,

it will wrap around to the opposite edge. In the latter mode, sprites which go outside the edge of the box will simply disappear until they return inside the box.

To execute any of these commands, one uses TELL. Suppose the Boxer contains a Graphics Box named "PAD." Then the following lines will clear PAD and put it in WINDOW mode:

```
TELL PAD CLEARSCREEN
TELL PAD WINDOW
```

3.3.3 A Problem

To draw on a Graphics Box, a sprite must somehow get put in that Graphics Box. Since Boxer makes it possible to have many Graphics Boxes in any region, sprites can not unambiguously get attached to graphics areas automatically as they are in Logo. Consider two possible remedies to this problem.

Firstly, one could add a command to Boxer called "ATTACH." This proposed command would take two arguments, a Sprite Box and Graphics Box and would link them together. Functionally this command would be satisfactory. However, the link in this scheme would be internal. This crucial information about the state of the system, would therefore be hidden from the user. Because it hides this important information from a user, this scheme violates the principle of naive realism.

Secondly, one could add a G-BOX state variable to each sprite. G-BOX could contain the name of the Graphics Box to which the sprite is attached. This scheme improves on the previous one by making the connection between boxes explicit and directly changeable. But consider the consequence of this type of pointer: In this scheme a sprite would exist in two places at once! A sprite's representation, its Sprite Box could exist in one area, while its actual drawing object could appear in a Graphics Box in an unrelated area. Thus this scheme would violate the strict hierarchical spatial scheme of Boxer in which each object has one unique location.

To overcome this problem, Sprite Graphics introduces one additional kind of box, the Graphics Data Box.

3.4 Graphics Data Boxes

A Graphics Data Box contains the logic behind a Graphics Box. To get a Graphics Data Box, the user points to a Graphics Box and presses a special key, the toggle-box key. (This is the same key used to convert a Do-it Box to a Data Box.) When this key is pressed, Boxer will change the Graphics Box on the screen into a Graphics Data Box. The user can think of a Graphics Data Box as the flip-side of a Graphics Box. Pressing the toggle-box key again toggles the a Graphics Data Box back into a Graphics Box, restoring the picture the Graphics Box contained before the flip.

The inside of a Graphics Data Box acts almost exactly like a regular Data Box. For example, the user can type and edit text inside a Graphics Data Box. However, the main purpose of this box type is to allow the user to link Sprite Boxes to Graphics Boxes and thereby to place a sprite on a drawing region.

3.4.1 Connecting Sprites to Graphics Boxes

In this design, the user attaches a particular sprite to a Graphics Box by placing the Sprite Box which represents that sprite inside the Graphics Data Box which holds the logic of the Graphics Box. In other words, a Sprite Box which appears contained in particular Graphics Data Box will show its picture in the Graphics Box which is the flip-side of that Graphics Data Box. Therefore, by toggling back and forth, the user can see the one-to-one correspondence between sprites in a Graphics Box and their Sprite Box representation in the associated Graphics Data Box. Since one can directly edit the contents of a Graphics Data Box, this design allows the user to concretely create, attach, and remove sprites from Graphics Boxes with the standard Boxer editor.

Figure 3-6 contains two named Sprite Boxes. The shape of one of these sprites is a square while the other has the default shape, but a bigger size. Figure 3-7 shows what will appear when the user toggles this Graphics Data Box.

3.4.2 Names

Graphics Data Boxes are special in one additional way -- a Graphics Data Box exports

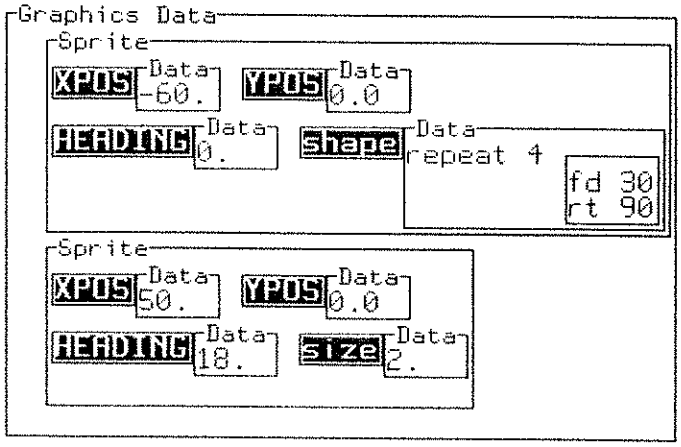

Figure 3-6: A Graphics Data Box with Two Sprite Boxes

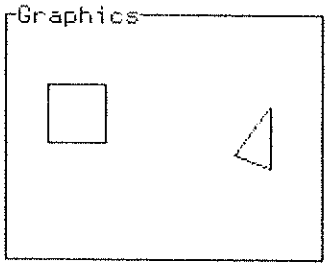

Figure 3-7: Toggling to a Graphics Box

the name of each box it contains to the surrounding box. This allows the user to reference the name of a Sprite Box from the environment which contains the Graphics Box in which the Sprite appears. Without this export mechanism, the TELL command would not work properly from outside a Graphics Box. In the figure 3-7, the lines below the Graphics Box show that the user can send commands to sprites from outside a Graphics Box.

Also note that when the user toggles a named Graphics Box, the resulting Graphics Data Box will have the same name.

3.4.3 Commands

The commands CLEARSCREEN, WRAP and WINDOW can be executed inside Graphics Data Boxes. In this case, the commands will affect the Graphics Box on the flip-side of the Graphics Data Box.

3.5 Advanced Features

In addition to the basic features described in the preceding sections, Sprite Graphics gives the user several other useful capabilities.

3.5.1 Additional Commands

In addition to the regular turtle commands discussed above, Sprite Graphics supplies five special-purpose commands -- GO-HOME, STAMP, TYPE, FLASH-NAME, and COPY-SELF. Like other sprite commands, these can either be executed directly from within a sprite, or can be directed to a sprite using TELL.

The GO-HOME command returns a sprite to the point specified by the sprite's HOME state variable, and sets its HEADING so that it is pointing straight up. This command is useful for establishing a defined starting location for a sprite.

The STAMP command causes a sprite to draw its own shape on the Graphics Box. A child could use stamp, for example, to draw a forest of trees by repeatedly stamping and moving a tree-shaped sprite.

TYPE takes one input, a data box filled with text. TYPE causes a sprite to type the text in the data box near its location. This purpose of this command is to allow the user to get get text to appear inside a Graphics Box. However, because the TYPE command can appear within a sprite's SHAPE state variable, the user can also use this command to incorporate words into a sprite's picture. As later examples will show, by making a sprite's shape a word, the user can easily create mouse sensitive menus.

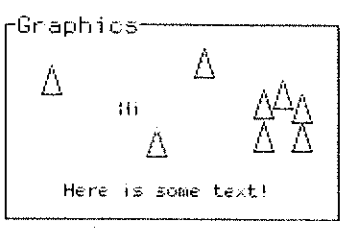

Figure 3-8: A Picture Made with STAMP and TYPE

FLASH-NAME causes Boxer to display a sprite's name next to its picture for two seconds. In the case where a user forgets which Sprite Box corresponds to a particular sprite, the user can use this command to identify a sprite.

COPY-SELF makes a copy of a Sprite Box. COPY-SELF allows the user to get Sprite Boxes without having to build each one by hand. This command encourages users to build shared libraries of sprites, and also allows novice users to acquire Sprite Boxes which they could not themselves build. In addition, this command permits the user to repeatedly copy a Sprite Box in order to build up a large collection of identical sprites.

3.5.2 Ports

As explained previously, a user can, in Sprite Graphics, create port a sprite's state variable in order to gain access to the variable from a remote place. The user can moreover create a port to a Sprite Box, in order to access all of the sprite's state information at once. (To create such a port, one need only type the name of the sprite and press the Do-it Key.) This capability is particular useful because it allows a user to simultaneously display a sprite's picture and its logic. By keeping a port to a Sprite Box, the user can watch the changes in a sprite's picture and its logic occur in parallel.

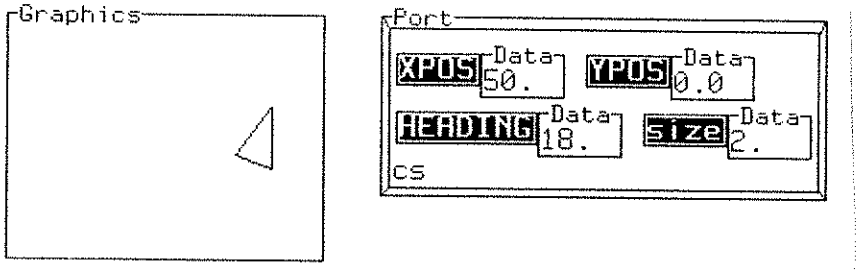

Figure 3-9: Using a Port to
Simultaneously Display A Sprite's Picture and Logic

3.5.3 TELL-ALL

It is sometimes useful to be able to direct a command to many sprites in one step. For example, a user might want to tell every sprite GO-HOME in order to get a Graphics Box back to its starting configuration. The TELL-ALL command provides this capability.

TELL-ALL, like TELL, needs two inputs, the second of which is either a command to execute or a Do-it Box containing commands to execute. TELL-ALL takes as its first input a Data Box which contains a list of names and/or ports. TELL-ALL sends the command half of its form to every box referenced by this Data Box.

3.5.4 Mouse Sensitivity

In Sprite Graphics, every sprite is mouse sensitive. As the user moves the mouse pointer across a Graphics Box, each individual sprite becomes highlighted as the mouse points to it. As figure 3-10 below shows, the highlighting is accomplished enclosing the sprite in a rectangle. This is one way in which the mouse acts differently in Graphics Boxes from the way it acts in the rest of Boxer.

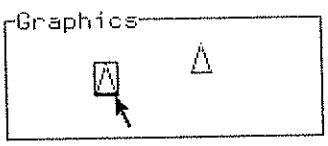

Figure 3-10: Pointing to a Sprite

In addition, clicks on the mouse buttons have a special meaning in Graphics Boxes. When the mouse is clicked on a highlighted sprite, Boxer looks in the sprite's Sprite Box for a Data Box named either L-CLICK, M-CLICK, or R-CLICK, depending on which mouse button was pressed. If the Sprite Box has an appropriate click box, Boxer will execute the code in the box. Otherwise nothing will happen.

Using click boxes the user can program sprites to perform arbitrary actions on mouse clicks. For example one might want a sprite which moved FORWARD 30 on a left click, turned RIGHT 90 on a middle click, and did CLEARSCREEN on a right click. Figure 3-11 below shows a sprite with these three click actions.

One command designed especially for click boxes is the command FOLLOW-MOUSE. FOLLOW-MOUSE causes the mouse to pick up and drag the sprite around the Graphics Box. The mouse will continue to drag the sprite until a mouse button is clicked. By using this command on a sprite which has its pen down, one can scribble in a Graphics Box. See the Sketchpad discussed in Chapter Four for an interesting example of FOLLOW-MOUSE.

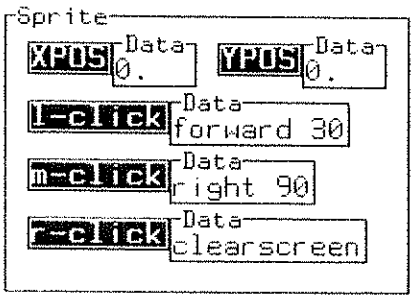

Figure 3-11: A Sprite Box with Click Boxes

3.5.5 Sprite-To-Sprite Sensitivity

Sprites know several commands regarding their interaction with other sprites. TOUCHING? will determine if a sprite is in contact with another sprite another sprite. TOUCHING? takes one argument, which can be either a name of a sprite box or a port to a sprite box. Here's an example:

```
TELL JOE TOUCHING? BILL
```

This will return TRUE if JOE is in contact with BILL and FALSE otherwise.

Another useful command is SINGLE-TOUCHING-SPRITE. This will return a Data Box containing a port to the one sprite that is directly under a sprite's center. If more than one sprite are under a sprite's center the one with the smallest area will be returned. (This is the same function as the one the mouse uses to decide which it will highlight.)

The complementary function of SINGLE-TOUCHING-SPRITE is the command ALL-TOUCHING-SPRITES. This command will return a Data Box with ports to all the sprites in contact with the sprite which the command is directed to.

Both the above are most useful in conjunction with TELL-ALL. For example, the line under this will cause JOE to send all the sprites in contact with him to their homes:

```
TELL JOE [TELL-ALL ALL-TOUCHING-SPRITES GO-HOME]
```

One other useful command is ENCLOSING-RECTANGLE. This returns a Data Box containing four numbers. The first two are the x and y coordinates of the upper left hand corner of the rectangle which encloses the sprite. The second two numbers give the lower right hand corner.

3.5.6 Composite Sprites

Sprite Graphics gives the user the ability to build sprites out of other sprites. This feature is useful for creating sprites which react to commands in a complex way. For example, suppose a child wants to create a clock sprite with two hands. On each tick of such a clock, the small hand must rotate 1/12 as much as the big hand. To accomplish this rotation with a simple sprite would require recalculating the sprite's shape on every tick. With subsprites, the user can take simpler approach. Recognizing that a clock face is a compound object made of two sub-objects, the hands, the child could define a constant shape for each hand, and change only the angle at which each hand points.

Sprite Graphics displays the connections between a sprite and its subsprites using Boxer's spatial metaphor: Whenever one Sprite Box contains a second Sprite Box, the second Sprite Box is a subsprite of first. The user therefore can build a composite sprite simply by inserting Sprite Boxes within other Sprite Boxes. Moreover, because Sprite Boxes can be nested to any depth, sprites may have many layers of subsprites.

In keeping with Boxer's structure, sprites in Sprite Graphics can only be connected in hierarchical arrangements. Each sprite therefore can have at most one superior sprite, but may possibly have many inferior sprites. For convenience, I will call an inferior sprite a "subsprite" and a sprite which has subsprites a "composite sprite."

Figures 3-12 and 3-13 show the logic and picture of a composite sprite side by side. This composite sprite illustrates one design for a clock. Note that the shape of the outermost sprite ("CLOCK") is a 12-sided polygon, while the shape of each subsprite is an straight line. Note furthermore, that the lengths of the two subsprites differ: one represents the big hand, while the other represents the little hand. The shape of the composite sprite, as you can see, is the composition of its own shape and the shape of its subsprites.

In order to support composite sprites, the meaning of state variable in subsprites must be refined. The refined semantics for sprite state variables, however, is completely compatible with the earlier explanation in the case of simple sprites. Firstly, as the above example shows, a sprite's displayed shape is actually the combination of its own SHAPE state variable with the shapes of all its subsprites. Secondly, a subsprite's position and heading state variables are relative to the position and heading of its superior sprite. Each subsprite's

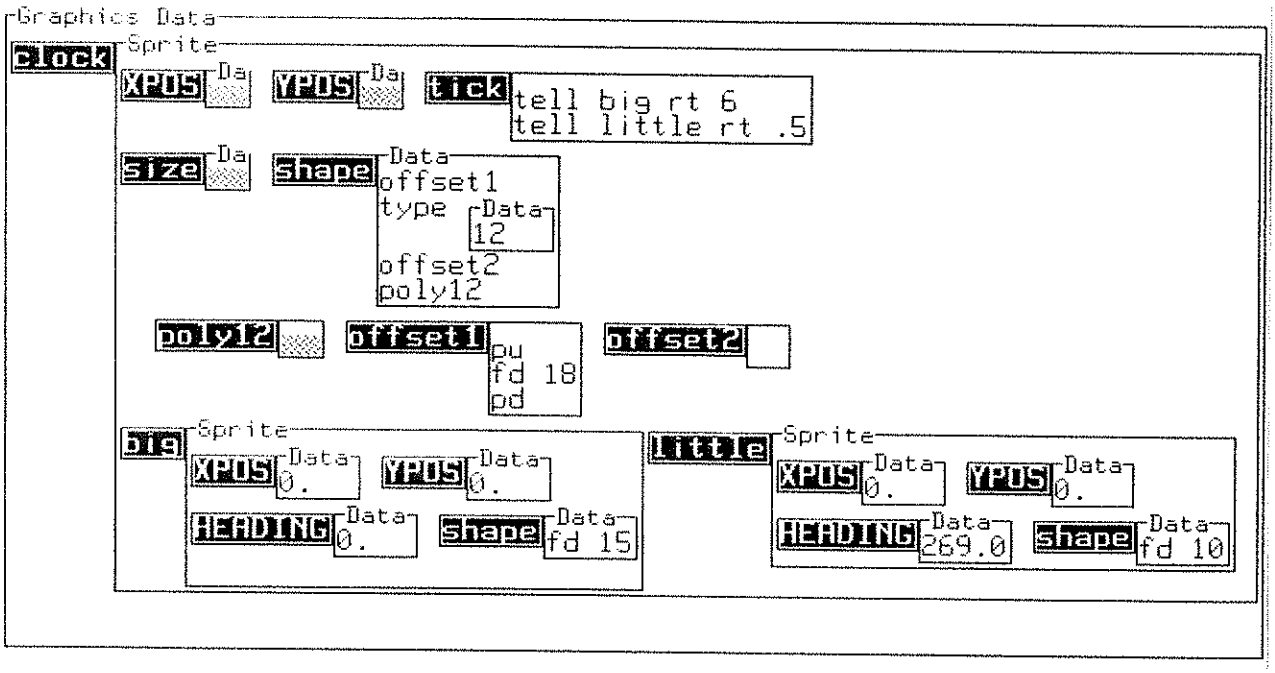

Figure 3-12: The Logic of a Sprite Graphics Clock

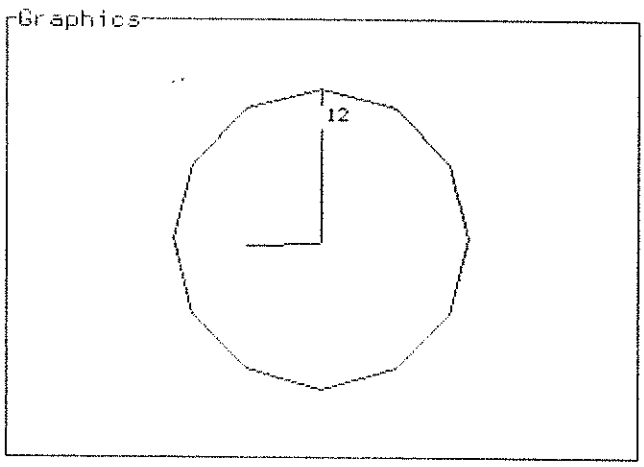

Figure 3-13: The Picture of a Sprite Graphics Clock

state variables thus determine its positioning within its superior sprite's combined shape. Thirdly, a subsprite's SHOWN state variable is relative to its superior sprite's SHOWN variable. If the value in a composite sprite's SHOWN box is false, none of the composite sprite's inferiors will appear on the screen. Finally, the value in a subsprite's SIZE variable is also relative. When the user changes the SIZE box inside the CLOCK sprite, for example, the hands as well as the clock face will change size.

Under most sprite commands, a composite sprite acts exactly like a simple sprite with an equivalent shape would. For example, the FORWARD command makes the entire shape of a composite sprite move forward. The RIGHT and LEFT commands, moreover, rotate the composite sprite's picture rigidly -- the parts of a shape maintain their relationships under rotation. In addition, these commands can be directed to subsprites in order change their state variables and thus change the shape of their superior sprite's composite shape. In the clock example, the Do-it Box named TICK illustrates the use of sprite commands in this way. This procedure, as you can see, rotates the big hand 6 degrees (i.e., one minute) and the small hand 1/12 of this angle.

Sprite Graphics supports one new command specifically for composite sprites, ROTATE. ROTATE works like RIGHT, but does not rotate a composite sprite rigidly. Though this command maintains a subsprite's relative position in the composite sprite's shape, it changes the subsprite's relative heading, to keep the subsprite's absolute orientation constant. ROTATE is useful in cases where a composite shape's parts change their heading independently.

Sprite Graphics also allows two special values in the SHOWN state variable of a composite sprite, SUBSPRITES and NO-SUBSPRITES. With the former value, the composite shape include only the shapes of the subsprites. With the latter value, a composite sprite effectively reverts to being a simple sprite by suppressing the shapes of its subsprites.

It is difficult to communicate the semantics of subsprites concisely in words even though they have been carefully designed to work intuitively "right." The example in the next chapter should help the reader get an idea of how composite sprites work. To learn more about composite sprites, I recommend that the interested reader experiment with them through a working Boxer.

Chapter Four

Examples

This chapter will present several sample applications of Sprite Graphics. These examples should help the reader to better understand the capabilities of this design.

4.1 Turtle Graphics

As the previous chapters have explained, the major purpose of Sprite Graphics is to give Boxer the functionality of Turtle Graphics. At this point, I would like to give an example to illustrate this use of Sprite Graphics. However, the dynamic aspects of a graphics system are difficult to capture in prose. To overcome this problem, a Turtle Graphics tutorial typically provides a series of snapshots of a computer session to illustrate how the system should be used. Because Sprite Graphics is compatible with Turtle Graphics except for syntactic differences, it seems pointless to provide this sort of example here; the reader can consult any book on Logo or Turtle Graphics to get an adequate idea of the capabilities of this style of graphics system. This chapter will instead concentrate on the capabilities of Sprite Graphics above and beyond Turtle Graphics.

4.2 A Thought Experiment

One potentially rewarding use for Sprite Graphics results from Boxer's capability for combining text and graphics. This capability opens up the possibility for a new communicational medium, the Boxer "book." In such a book, which readers would examine through a Boxer computer, the author could use Sprite Graphics to include illustrations along with the text.

Sprite Graphics illustrations, however, could communicate information much more

vividly then is possible with conventional printed books because the illustrations could be interactive. To get a feel for the value of Sprite Graphics illustrations, consider the question: "How could this thesis better describe Sprite Graphics if it was written in Boxer rather than on paper?"

In a Boxer version of my thesis, I would probably include many of the same illustrations that are in this version. However, in Boxer, these illustrations could fill a stronger role. Firstly, because all the Graphics Boxes in Boxer are "live," a reader could learn about Sprite Graphics by experimenting with the sample Graphics Boxes provided in the text. Secondly, because Boxer allows the user to point at previously typed lines and execute them, I could easily lay out some interesting examples for the reader to try. Thirdly, the examples in a Boxer thesis could be more open-ended -- the reader could use the examples as building blocks for projects of his or her own design. Figure 4-1 shows how a page in a Boxer thesis on Sprite Graphics might look.

This more direct style, in which the illustrations are primary and the text secondary, is especially appropriate for a thesis on Sprite Graphics. Still, Sprite Graphics could enhance the effectiveness with which many other topics could be communicated as well. For example, a recent project of the Educational Computing Group was to produce interactive computer simulations of concepts taught in Freshman Physics. These simulations could be re-written in Boxer with text accompanying graphics side-by-side on the computer screen. A student reading this text could immediately confirm his understanding of each new concept introduced in the written material by using the concept in an interactive simulation. If seeing is believing, classroom material written in Boxer is likely not only to be easier to understand, but also more convincing.

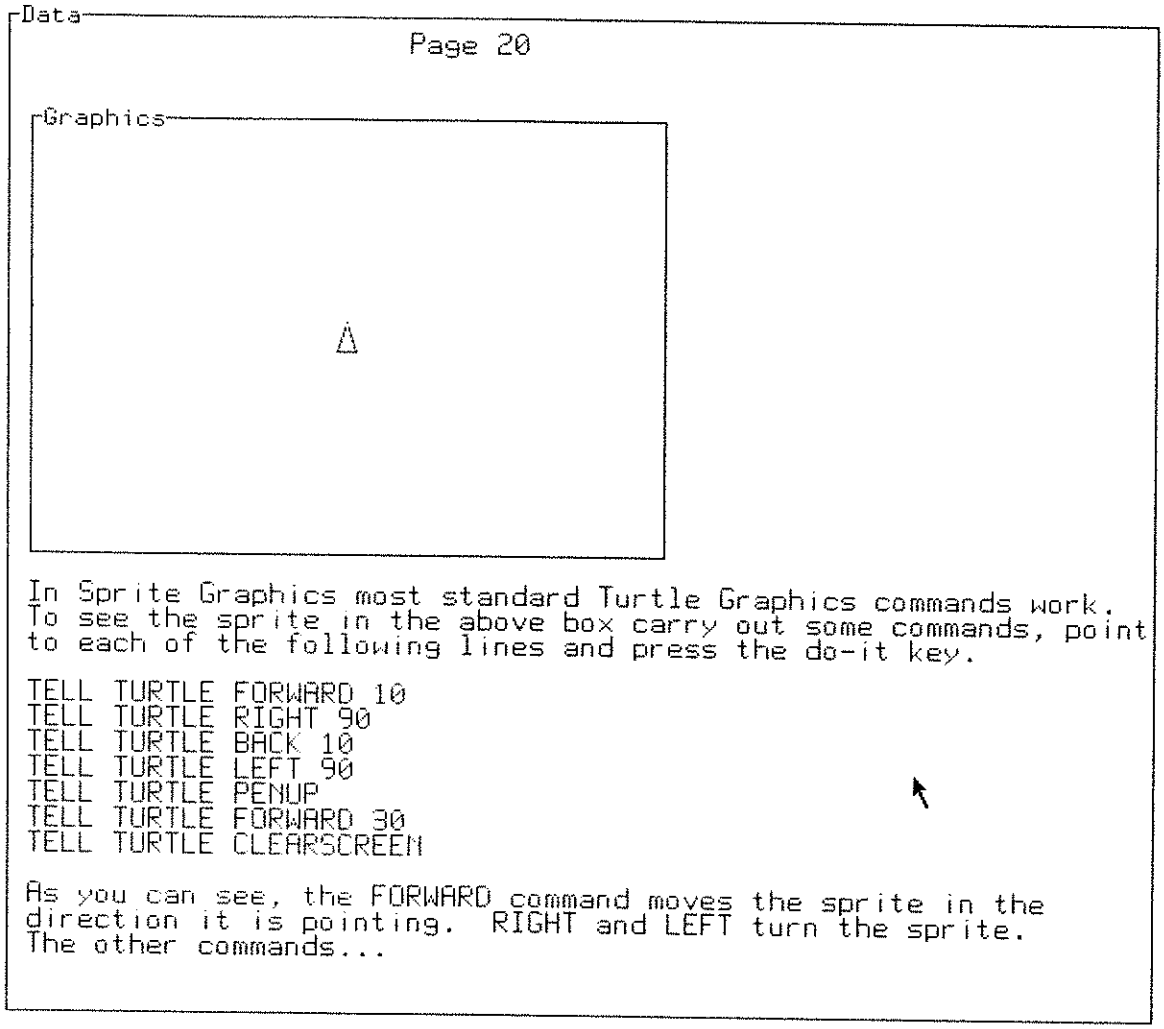

Figure 4-1: A Page From A Boxer Book

4.3 Sketchpad

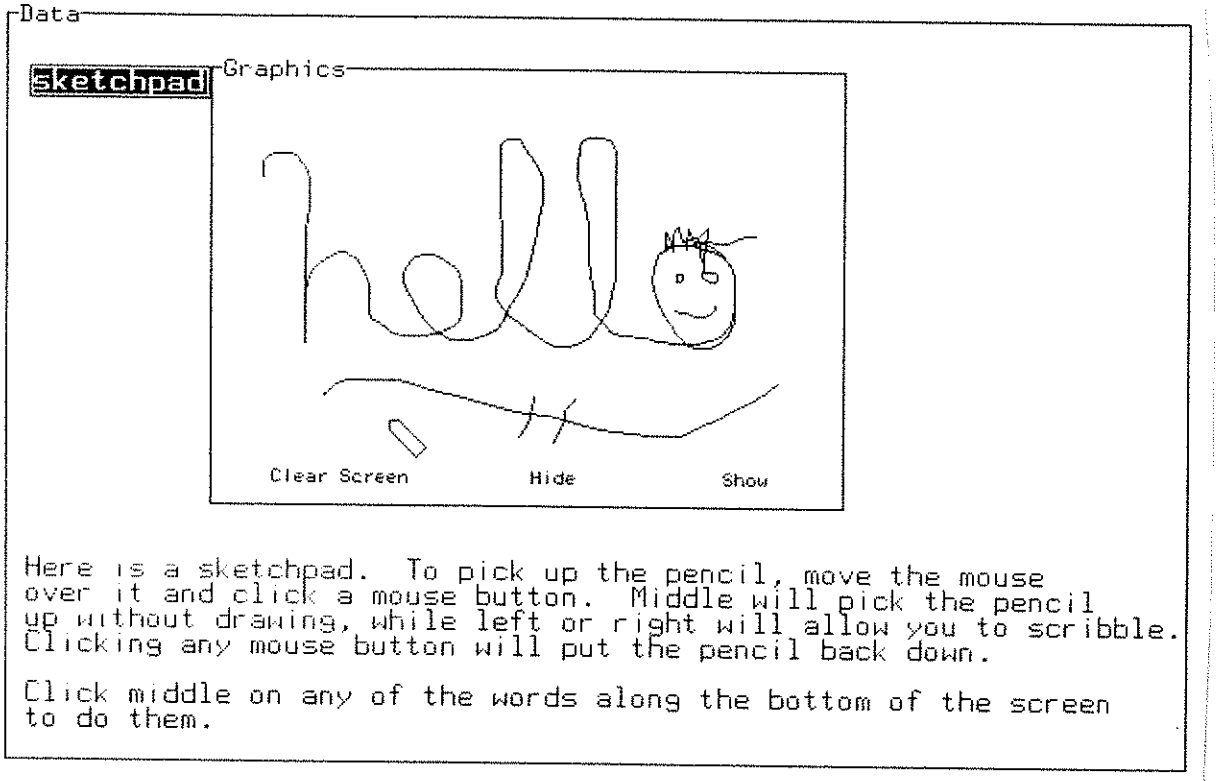

Figure 4-2: A Sprite Graphics Sketchpad

This example shows a simple, yet fun and useful interface which can be built with Sprite Graphics -- a mouse sensitive sketchpad. Figure 4-2 shows the Sketchpad Graphics Box, complete with a free-hand drawing done with the mouse. There are four mouse-sensitive objects in this Graphics Box, the pen (which appears just above the word "Screen"), and the menu choices "Clear Screen," "Hide," and "Show." The user can pick up the pen and drag it along with the mouse by pointing to it and clicking a mouse button. Depending on the mouse button pressed, the pen may or may not draw as it moves. The user may select menu choices by pointing and clicking the mouse. The first choice clears the screen, while the other two control whether the pen appears in the box.

Figures 4-3 and 4-4 show the flip-side of this Graphics Box, revealing its implementation. The Graphics Data Box contains four sprites, one for each mouse sensitive object. Figure 4-3 displays a view of the Graphics Data Box with the Pen sprite expanded, while the other figure shows a view with the three menu sprites expanded.

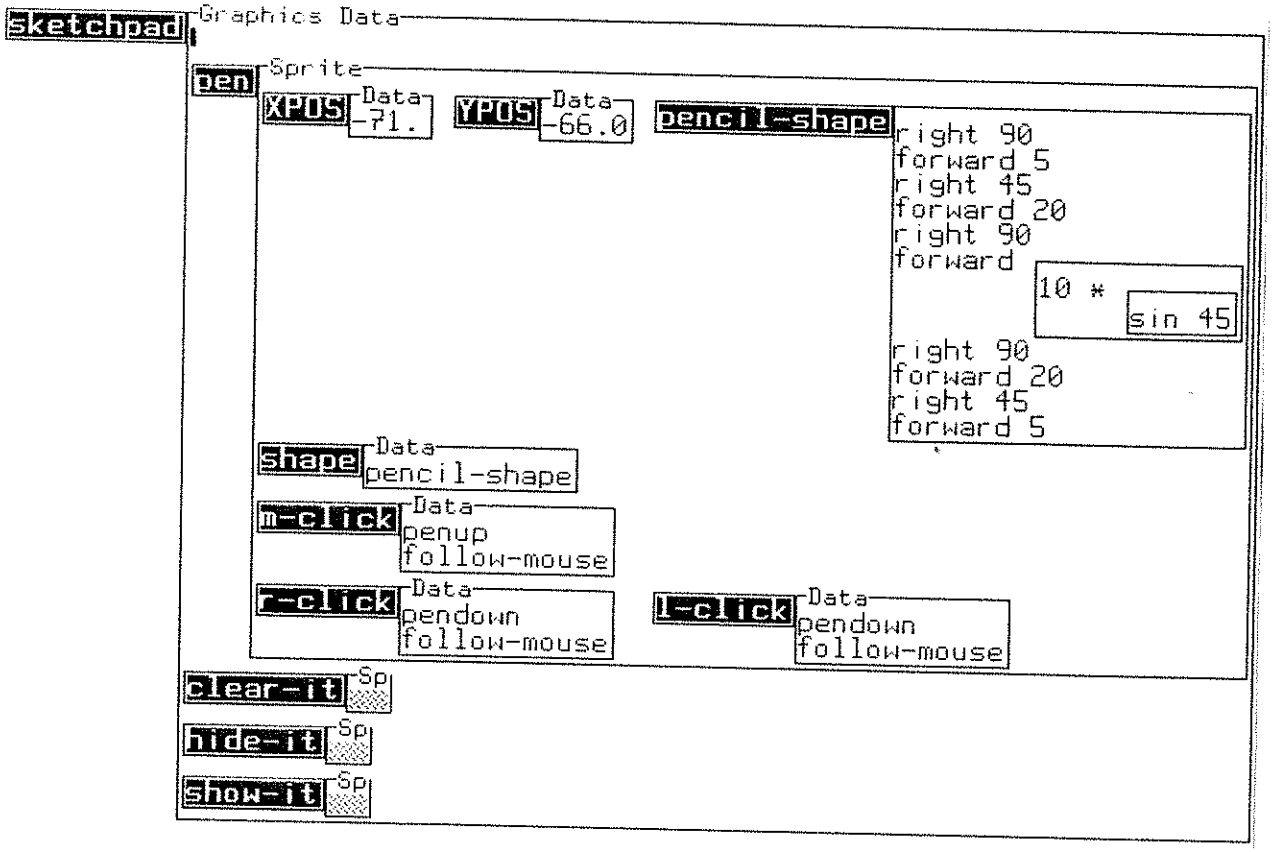

Figure 4-3: The Logic of the Sketchpad's Pen

The Pen sprite differs a default sprite in only two ways, its shape and its click boxes. The sprite's SHAPE state variable refers to a procedure, PENCIL-SHAPE, which draws the appropriate picture with sprite commands. The click boxes specify only two commands each. The first is a new status for the pen, either up or down. The second is the command FOLLOW-MOUSE which causes the sprite to be dragged along with the mouse. So although this sprite exhibits an interesting behavior which is not supplied by default, its definition is still relatively simple.

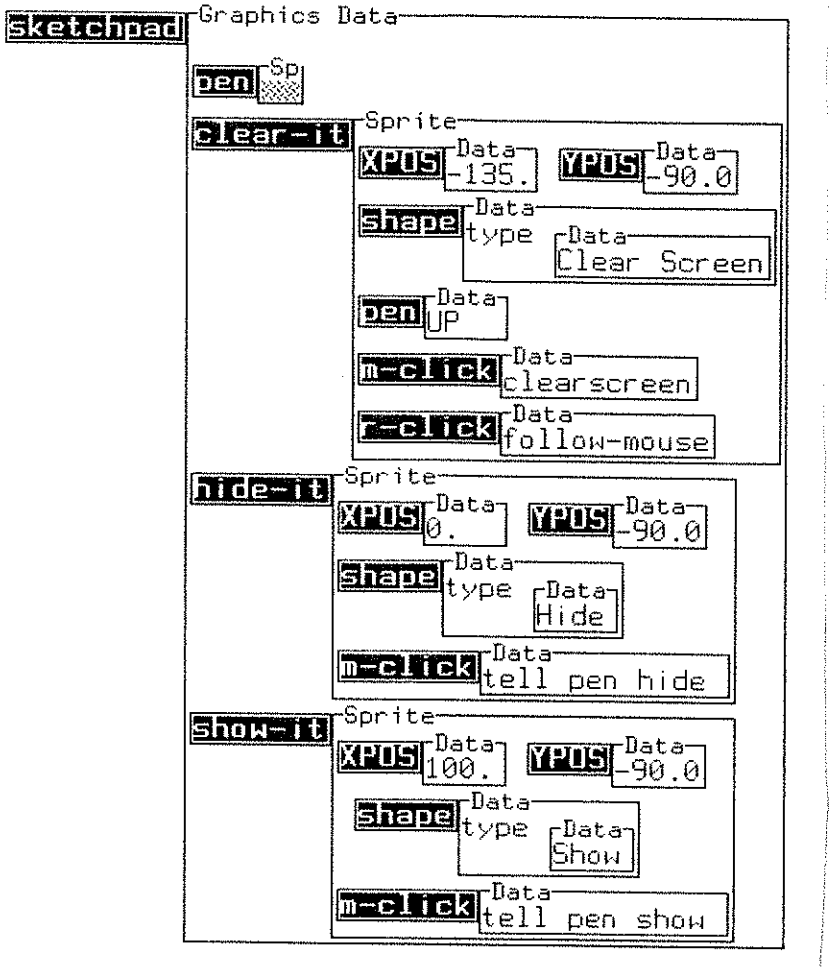

Figure 4-4: The Logic of the Sketchpad's Menu Choices

The three other sprites are likewise fairly simple. Each uses the TYPE command within its SHAPE state variable in order to get text to appear in the Graphics Box. Moreover, each specifies a click box which will cause an appropriate action. The first, Clear-it, does CLEARSCREEN on a middle click. The other two tell the pen either to hide or show itself. Though this information alone is enough to cause the described behavior, the Clear-it sprite includes one additional click box, R-CLICK, which contains the command FOLLOW-MOUSE. This sprite, therefore, can be picked up and moved to a different part of the screen. Note that since Clear-it's pen is up, it will not draw as it is moved.

From this example, it should be clear how a user could build more sophistication into this Graphics Box by adding more sprites. For instance, one might want to add a eraser-shaped sprite which erased instead of drawing. Or one might want to add new menu choices

to the Graphics Box. Therefore, this example shows not only how easily interesting mouse-sensitive interfaces can be built using Sprite Graphics, but also how the explorability of Sprite Graphics encourages the user to extend and personalize graphics applications.

4.4 A Calculator

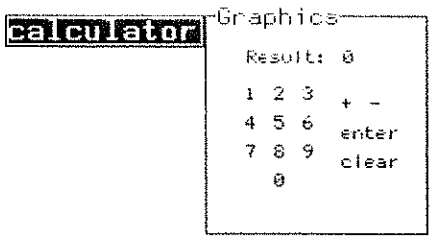

Figure 4-5: A Sprite Graphics Calculator

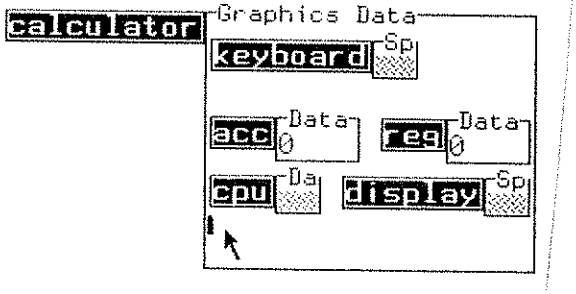

Figure 4-6: Top-level View of the Inside of the Calculator

This application demonstrates how Sprite Graphics can be used to model a real world object, in this case, a simple calculator. Figure 4-5 shows what the Sprite Graphics calculator looks like when displayed in a Graphics Box. The top line of the calculator, which contains the word "Result," is its display. The rest of the calculator is a mouse-sensitive keypad. To use this calculator, one presses the "keys" by pointing the mouse. The calculator works in reverse Polish notation, like a Hewlett-Packard.

Figure 4-6 shows a top-level view of the Graphics Data Box which contains the logic for the calculator. This model, as is clear from the structure of the Graphics Data Box, contains five top level objects, a keyboard, two registers ("ACC" and "REG"), a central processing unit ("CPU") and a display. In this model of a calculator, the objects interact as follows: The keyboard translates key presses into CPU commands. The CPU commands then modify the

contents of the two registers. Finally, each time it changes a register the CPU asks the display to update itself.

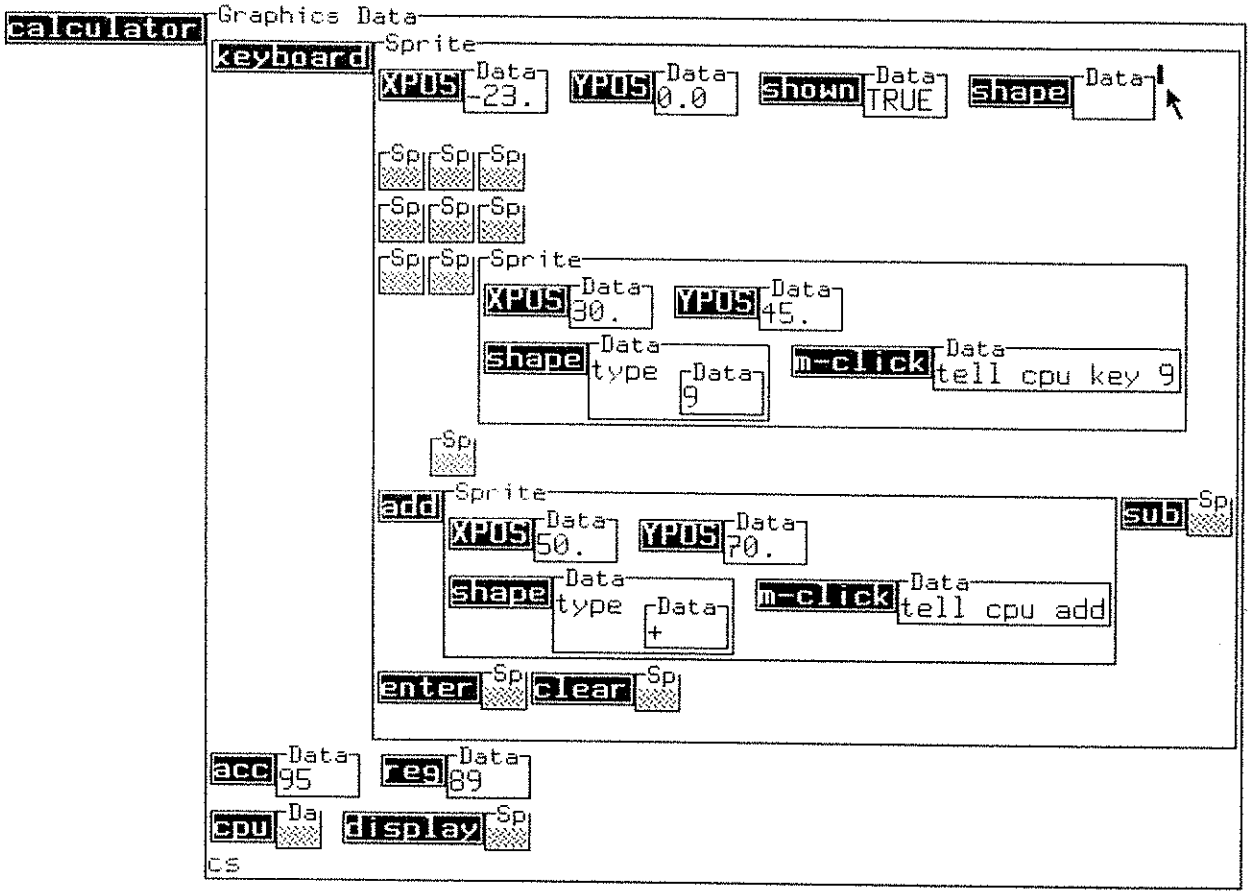

Figure 4-7: The Keyboard Sprite

Figure 4-7 shows an expanded view of the keyboard module. The keyboard is built out of fourteen sprites, one for each key. The Keyboard sprite itself serves only as a container for these subsprites. Note that the subsprites inside the Keyboard sprite are conveniently arranged to reflect their position on the keypad. The first four rows of Sprite Boxes represent the number keys; the three boxes in the first row represent keys 1 through 3, the second row holds keys 4 through 6, etc. The four Sprite Boxes named "add," "sub," "enter," and "clear" hold the plus, minus, enter and clear keys respectively.

Figure 4-7 moreover exhibits two expanded Sprite Boxes to illustrate how these subsprites work. Note that each subsprite contains three important pieces of information, its position, its shape, and its mouse click operation. The position of each sprite is, of course, its

relative position in the keyboard layout. Each sprite's SHAPE box contains a TYPE command with the appropriate text symbol. In its click box, each sprite sends a message to the CPU. There are only five possible messages, "KEY n", "ADD", "SUBTRACT", "ENTER", and "CLEAR". The first message signifies that a number key has been pressed. The number supplied with this command is the number of the key (The number 9 key sends the message "KEY 9," for example). The other messages signify that the key of the same name has been pressed.

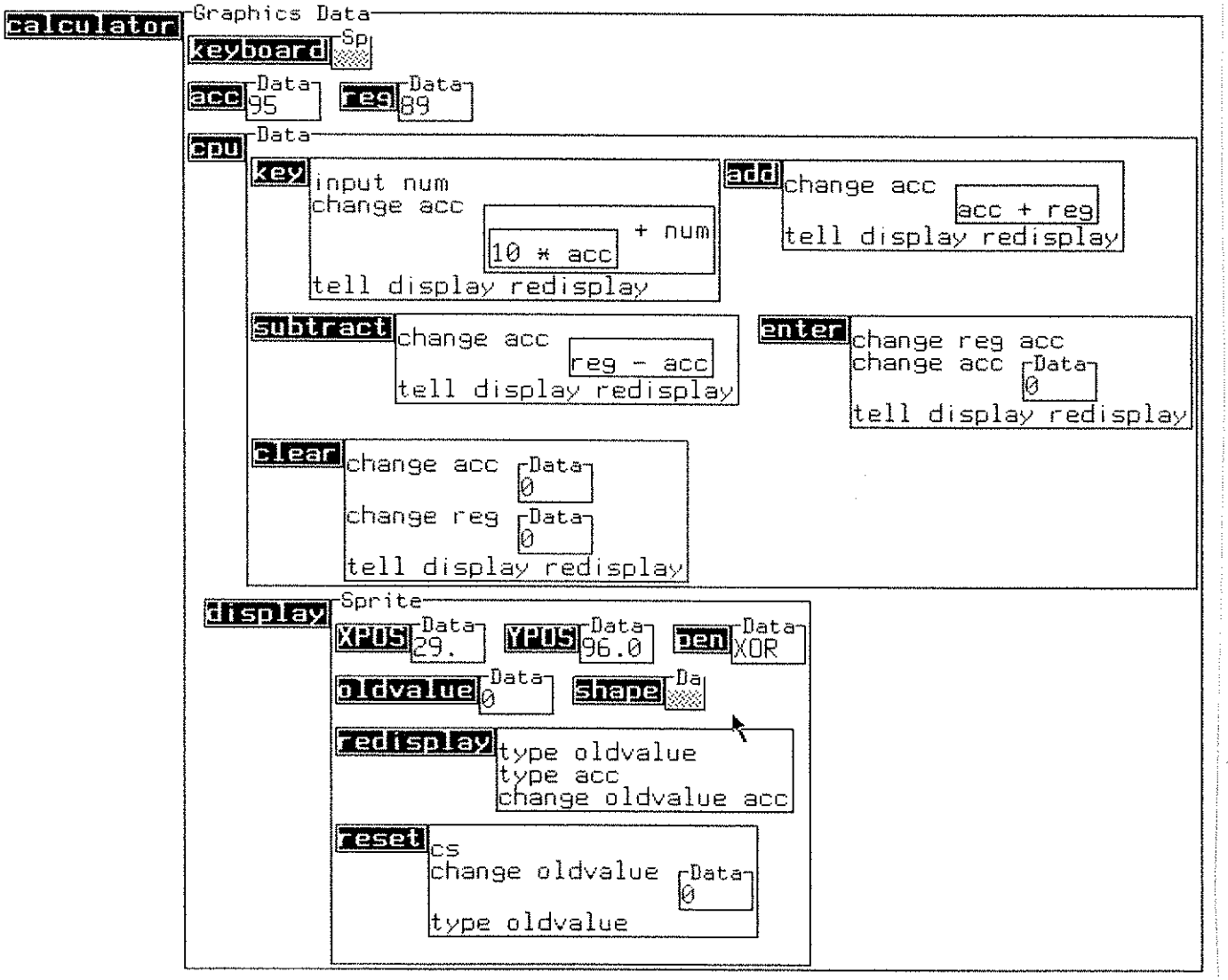

Figure 4-8: The CPU and Display Modules

The CPU, as figure 4-8 shows, is just a Data Box which contains five Do-it Boxes, one to handle each message. Each Do-it box transforms the registers ACC and REG appropriately and then sends the Display sprite a redisplay request. For example, the "Add" Do-it Box adds the contents of the two registers and puts the result into the ACC register. The Clear Do-it Box just changes the value in each register to zero. The only non-obvious procedure is in the Key Do-it Box. This procedure first multiples the contents of the ACC register by ten, which effectively shifts each digit one place to the left. Then it adds the number of the new key to this register, which puts the new number into the rightmost place.

Note that the CPU is not a Sprite Box, because it does not need appear as part of the calculator. On the other hand, the Display necessarily is a Sprite Box as its job is to maintain the graphic display of the ACC register. The Display sprite uses the TYPE command to print the value of the accumulator onto the Graphics Box. But before it prints a new value, the Display must erase the old number. In order to erase the old number from the screen, the display must know what the current value shown on the screen is. The display sprite therefore holds a private Data Box called "OLDVALUE" which contains the number last printed on the screen. Then to erase this value, because it uses XOR mode, Display needs only re-type the value. After erasing the old value, the redisplay procedure types the new value of the ACC Data Box. Finally, redisplay stores the value of the ACC register into OLDVALUE so it will be able to erase this value the next time around.

This calculator has several interesting properties. Firstly, like the real object, a graphical calculator can be used to solve arithmetic problems. Secondly, because this calculator's internal structure reflects a model which can explain how a real calculator might work, a student can learn about real calculators by exploring this one. Thirdly, because this calculator is readily changeable a student could learn about building calculators by attempting to expand upon this model. For example, a teacher could give students the project of adding a multiplication function to the calculator (or a division key for extra credit). While adding this function is not a terribly complex problem[5], the student could gain a great

[5]The user must do only two things to do to add a new key. First, the user must add a Sprite Box to the Keyboard to represent the key. Secondly, the user must add a new Do-it box to the CPU to carry out the calculation. The Keyboard and CPU modules supply plenty of prototype boxes to guide the construction of each of these new boxes

sense of accomplishment and understanding at the end of such a project, because a calculator seems to be such an impossibly complex thing to build. Finally because this calculator is so easy to expand upon, it can customized to suit one's individual needs -- an business student might add a net present value key, for instance.

4.5 Icon Programming System

Because of its flexibility in defining the interaction of graphics objects, Sprite Graphics makes it possible to define a different kind of programming language on top of Boxer. This section will describe a Icon Programming System written in Boxer. In this programming system, the user can build procedures concretely by arranging columns of icons in a Graphics Box.

Figure 4-9 shows what the Icon Programming System looks like. This system uses two Graphics Boxes. The one on the top contains the sprite which will carry out all the icon programs. The bottom Graphics Box contains the icons, which appear as two letter tokens (though they could just as easily be representative shapes). Each icon stands for a specific command -- FD stands for FORWARD, RT for RIGHT, PU for PENUP, etc. The two other objects in the bottom Graphics Box are columns, which play a role in the Icon System analogous to the role of Do-it Boxes in normal Boxer programming.

The Icon System supports three different levels of interaction with the sprite. At the lowest level, the user can cause actions to happen immediately. For example, to cause the sprite to go forward, the user clicks the mouse on the FD icon. At the second level, the user can pick up icons and drop them into column slots. Then the user can execute the column in immediate mode, by clicking the mouse on it. When a column is clicked on, it executes the icon in each of slots in top-down order. The second level of interaction with the Icon System therefore is similar to the activity of grouping Boxer code within a Do-it Box and executing the Do-it Box.

The third level or interaction allows the user to use the columns of icons abstractly. To do this the user would first shrink the column labeled "Prog" by clicking the left mouse button on it. When it shrinks, this column hides its column shape and instead appears as the two-

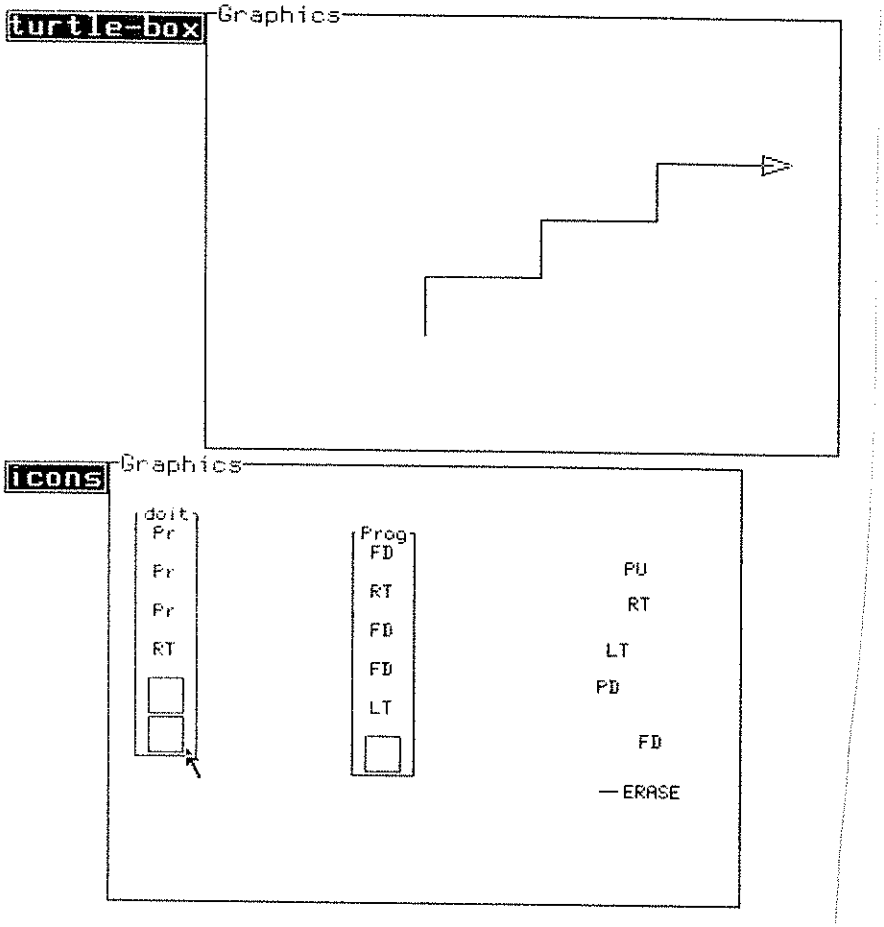

Figure 4-9: The Icon Programming System

letter token "Pr." Having shrunk this column, the user can then pick it up just like any other icon, and can insert it into the column labeled "Doit." When the latter column is executed, it will call the Prog column as a subroutine.

Figure 4-9 shows the system after the user has executed the Doit column. In this case, the first three slots in the Doit column are "Pr", so the sprite first follows the instructions in the Prog column three times. Then since the fourth icon in the Doit column is "RT," the sprite turns right ninety degrees. The final two slots in the Doit column are empty, so nothing else happens. The top Graphics Box shows the final state of the sprite after carrying out these instructions.

Building an Icon Programming System like this one is by no means a trivial project in most computer languages. Consider the sub-problems which are involved. Firstly, the

programmer must design a data structure to represent an icon. Second, he or she needs to specify specify the interaction between the mouse and the icons. Thirdly, the programmer must develop procedures for hooking up icons with columns. In addition, the programmer must arrange for all this information to be graphically displayed. Yet using Boxer and Sprite Graphics, I was able to create this system in about four hours. Moreover, the code behind this system is simple enough that anyone with a working knowledge of Boxer and Sprite Graphics should be able to understand its operation.

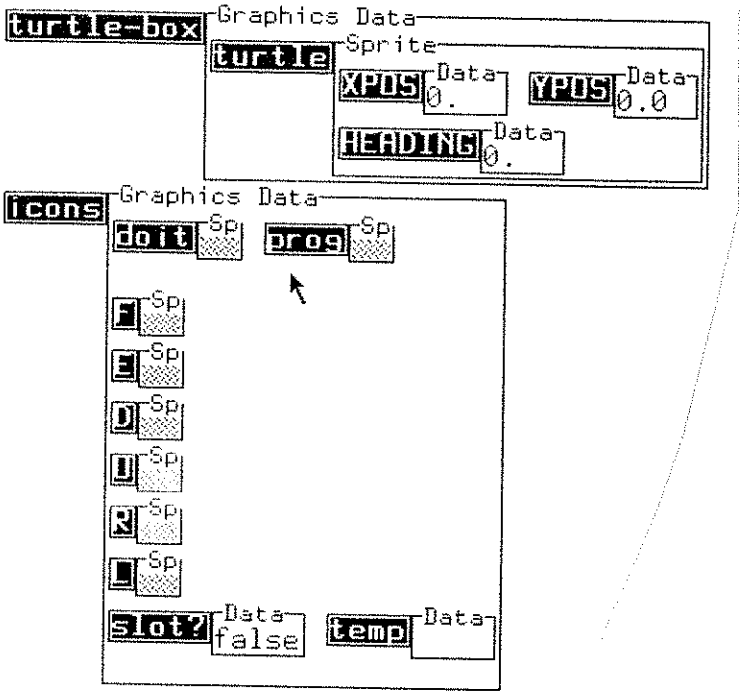

Figure 4-10: A Top-level Look at the Graphics Data Boxes for the Icon Programming system

Figure 4-10 above shows the objects which make up the system. The top box contains only one object, a simple sprite named "Turtle." This sprite does all the drawing in response to commands sent to it from the icons via the TELL command. The bottom box contains eight Sprite Boxes. The first two sprites, "Prog" and "Doit" represent the two columns in the system. The other sprites each represent an Icon -- the sprite "F" is the "FD" icon for example. The following paragraphs will explain each component of the system in more detail.

Figure 4-11 shows the contents of the FD icon's Sprite Box. As in previous examples,

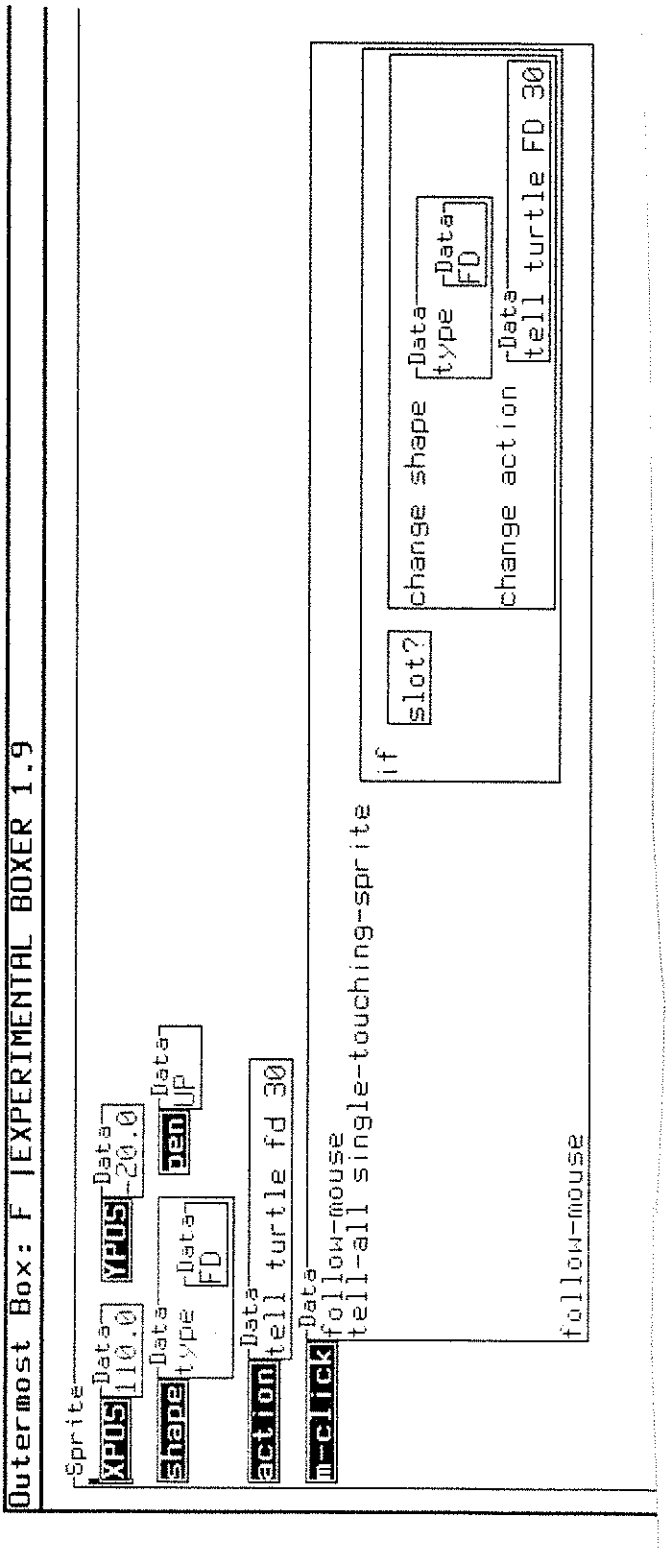

Figure 4-11: Inside the FD icon's Sprite Box

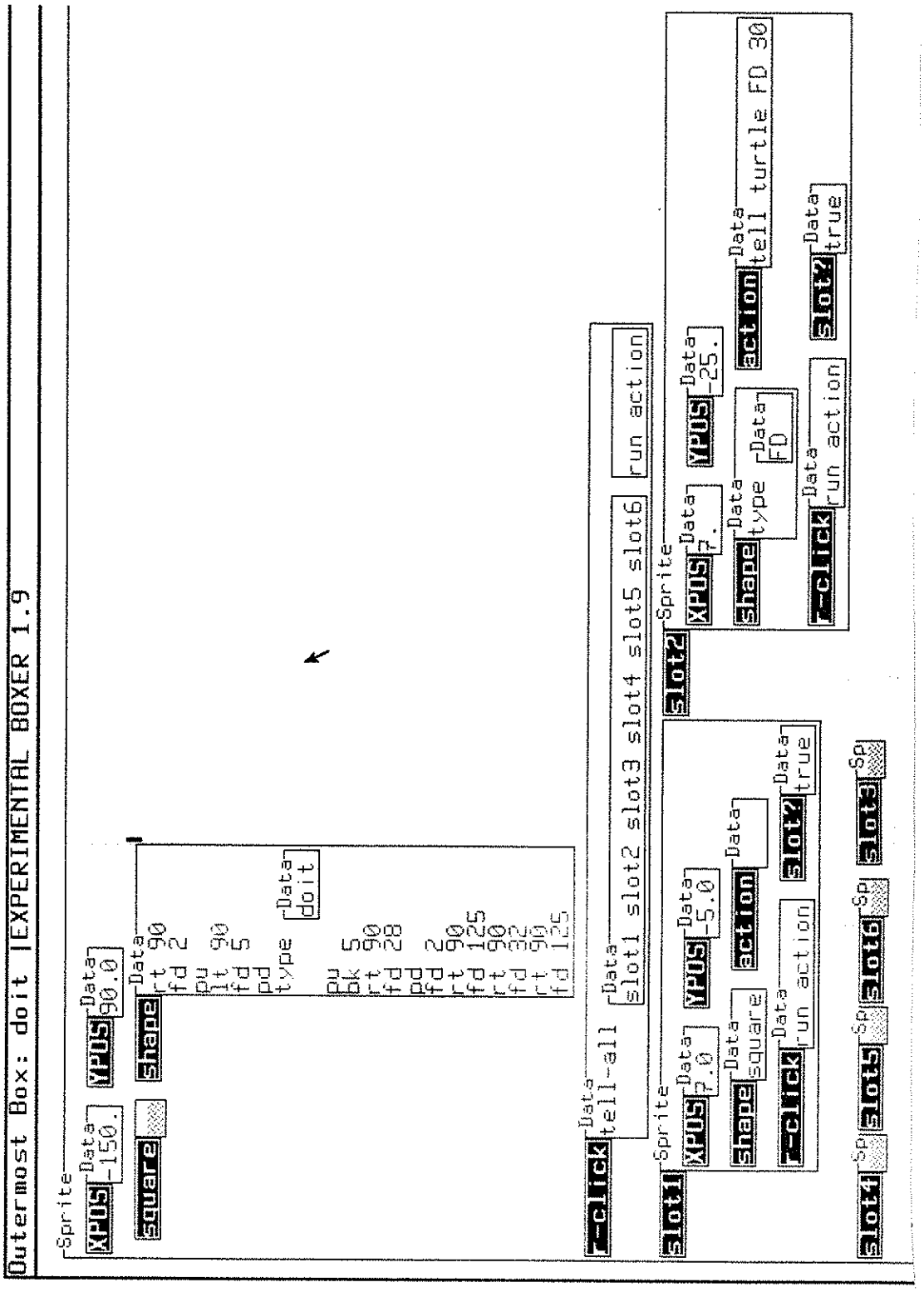

Figure 4-12: Inside the Doit Column's Sprite Box

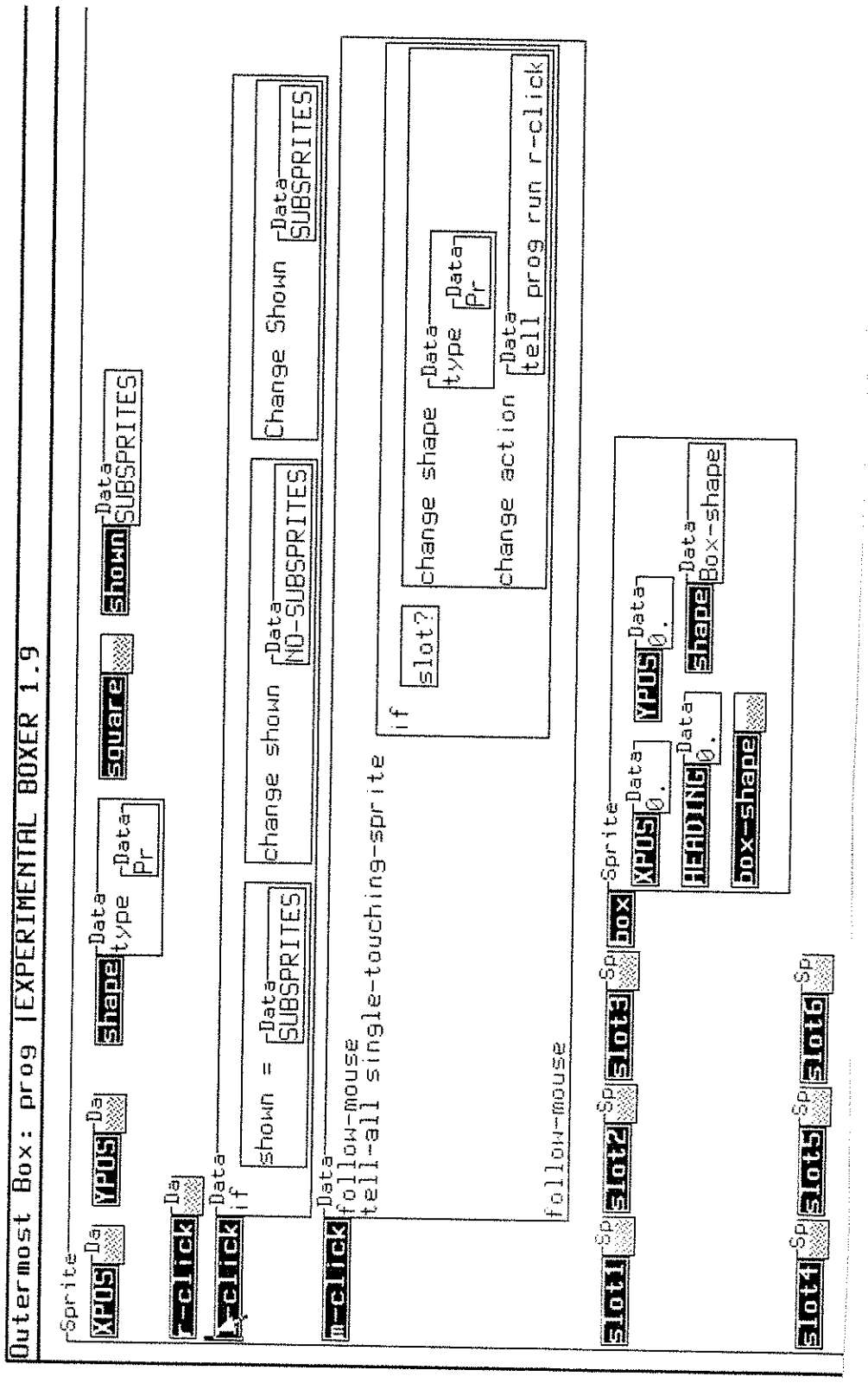

Figure 4-13: A Look Inside the Prog Sprite

the icon's SHAPE state variable uses the TYPE command in order to get the text "FD" to be the icon's picture. Every icon's shape contains a similar line. In addition, each icon contains a box labeled "ACTION". This code in this box cause the TURTLE sprite to carry out an appropriate action, going forward in this case. Moreover, since the R-CLICK box of each icon specifies the code "RUN ACTION", each icon executes its action on a right click of the mouse. The M-CLICK box contains the code which causes the icon to insert itself into a slot in a column. This code will be explained in more detail later.

Figure 4-12 shows the contents of the less complicated column sprite, the Doit column. Interestingly enough, though this sprite's picture looks like a Boxer Do-it Box, its shape is just a list of regular sprite commands which when executed will draw a box. Both column sprites contain six subsprites named "SLOT1" though "SLOT6." Each of these slot sprites has two Data Boxes which contain the two crucial pieces of information about an icon, its shape and its action. The column sprite, in order to execute itself, simply executes each slot's action in sequence, as the outer sprite's R-CLICK box shows.

To make any of these slots look like an icon, therefore, one must only set the slot's shape and action appropriately. In fact, this is exactly what the M-CLICK box of each icon does. Look back to the M-CLICK box in figure 4-11. The FOLLOW-MOUSE command in the first and last lines of the code allow the icon to be carried to and from the column sprite. The middle line sends a Do-it Box to the sprite which the icon is touching. If the icon is actually touching a slot, this Do-it Box changes the slot's shape and action to be its own. Therefore after an icon has touched a slot, the slot will look and act just like the icon.[6]

The Prog column sprite (figure 4-13) must combine the features on a column sprite and of an icon sprite, since it acts like both. The most interesting feature of this sprite is the way it uses Sprite Graphics mechanism for handling subsprites to approximate Boxer's ability to shrink boxes. Compared the Doit sprite, the Prog sprite contains one extra subsprite. This sprite, which is called "Box", has a box shape. The SHAPE variable of the Prog box itself specifies the a two-letter token "Pr". With this arrangement, the combined shape of all the subsprites looks like a column, while the shape of the top-level sprite alone looks like an icon.

---

[6]This style of programming, in which one object injects its own code into another object, might be called "Virus Programming" in analogy to the way a virus attacks a cell.

To toggle back and forth between these two pictures, the Prog sprite must only toggle the word in its SHOWN state variable between SUBSPRITES and NO-SUBSPRITES. Figure 4-14 below shows the Prog sprite in each of these states.

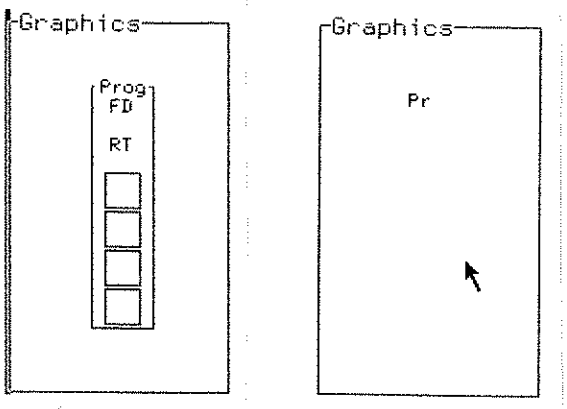

Figure 4-14: Expanding and Shrinking the Prog Sprite

In column mode, the Prog sprite looks and acts identically to the Doit column. Like the Doit column, it has six slot sprites, each of which maintains a shape and an action. Moreover, the Prog sprite also executes itself by running the actions of each of its subsprites. On the other hand, this sprite's M-CLICK operation is very similar to an icon sprite's: Prog is able to insert code into slot which will make the slot look like its token "Pr" and will also make the slot run the its action. The latter is accomplished by making the slot's action be "TELL PROG R-CLICK." In other words the slot's action is to call Prog's action.

This example makes use of the most advanced features of Sprite Graphics, including mouse sensitivity, sprite-to-sprite sensitivity, message-passing, and composite sprites in order to produce a complex and interesting interface. Of course, a more sophisticated icon system using Sprite Graphics can be built which incorporates more programming language constructs, like loops for example. But the main point of this example is not to show the functionality which Sprite Graphics brings to Boxer, but rather is to show the ease with which complicated graphics systems can be understood, constructed, and modified using Sprite Graphics.

4.6 Ideas

The potential uses for Sprite Graphics certainly go beyond the few applications given here. For example, I can think of several projects based on the Sprite Graphics STAMP command. Imagine a set of sprites which look like the schematic shapes of electrical components. With such a set and the commands FOLLOW-MOUSE and STAMP, one could easily produce drawings of electrical circuits in Boxer. Another possibility would be a flowchart drawing tool based on a similar concept. One could even make a flowchart in which each element of the flowchart could be opened up (using the subsprite mechanism described in the Icon example) and examined in more detail.

Another useful kind of interface which could be built with Sprite Graphics would be a mouse sensitive system menu. Such menus are common in many computer systems, for example the Apple MacIntosh and the Symbolics 3600. A Boxer system menu might take over functions which are now bound to hard-to-find keys, like name-box, make-box, exit-box, shrink-box, etc. Because the user can always flip over a Graphics Box and add new sprites or modify the ones already there, a Boxer system menu would be more easily customized than either the Apple or Symbolics systems.

Of course, Sprite Graphics is also the natural place for many sorts of Graphic utilities which a school child could use. The calculator is one example. Another would be a Plotter which could quickly draw line graphs of lab results, which then could be included directly into a student's lab report via the Boxer editor.

Chapter Five

Evaluation

This chapter will evaluate the achievements of Sprite Graphics design relative to its goals. It will also suggest some topics for future research.

5.1 Compatibility with Boxer

Several features make Sprite Graphics especially well suited to Boxer. Firstly, Sprite Graphics adapts all graphics objects to box representations. This allows the user to manipulate Sprite Boxes, Graphics Boxes, and Graphics Data Boxes with standard Boxer commands. In the case of Sprite Boxes, the box representation extends further: Sprite state variables, like all Boxer variables, are represented as Data Boxes.

Secondly, unlike Turtle Graphics, Sprite Graphics allows standard Boxer commands to operate on graphics objects whenever possible. To update a state variable, XPOS for example, in Sprite Graphics one uses the standard Boxer CHANGE command. In Turtle Graphics, one needs a special SET-XPOS command instead. The TELL command, too, is a generic Boxer command. In some cases, the availability of Boxer features even makes common Turtle Graphics commands unnecessary. For instance, Sprite Graphics makes the operation of attaching Sprite Boxes to Graphics Boxes a physical one: to add sprites the user makes a new Sprite Box, picks it up and drops it into the Graphics Data Box.

Thirdly, Sprite Graphics carefully preserves the viewpoint of naive realism. The state of each part of the Sprite Graphics system is open to the user. One can directly see every sprite state variable maintained by the underlying implementation as well as the connections between sprites and Graphics Boxes. Sprite Graphics moreover supports the illusion that a sprite's picture and the logic in its Sprite Box are simply alternate representation of one thing. This is done by keeping each sprite's logic and its picture consistent and by allowing the user

to flip freely from one representation to the other (or even see both at once via a port). The fact that the user can change a sprite's state variable in any number of ways and immediately see the resulting change to the sprite's picture makes the illusion of naive realism especially believable.

Sprite Graphics however stays away from the conceptual difficulties which might result if the system were too realistic. Consider what the user's impression would be if sprite state variables were Port Boxes rather than Data Boxes. Port Boxes would, of course, be more realistic because the values in the Sprite Box really share the values in the Lisp implementation of sprites. However, if state variables were ports, the user would likely want to know where the targets of these ports were. Any convincing explanation would have to rely on the fact that the targets of these ports are objects hidden in the state of the system. Such an explanation however, destroys the viewpoint of naive realism and the sense of understanding and control that comes with it.

Sprite Graphics also carries over Boxer's presentational flexibility. For example, Sprite Graphics allows the user to arrange the state variables inside a Sprite Box in any order. Unused state variables can even be deleted. Another example of presentational flexibility results from the decision to make Graphics Boxes toggle into Graphics Data Boxes, rather than keeping both boxes always visible. With this arrangement, one can use a Graphics Box with being distracted by its logic. On the other hand one can always toggle the Graphics Box to see the logic or even use ports to see the logic and picture simultaneously.

Fifthly, Sprite Graphics maintains Boxer's hierarchical structure and its accompanying spatial metaphor. As mentioned in the Chapter 3, each object in the graphics system has one unique location within the Boxer world, though ports allow the object to be shared elsewhere. Because Graphics Boxes toggle into a Graphics Data Box in the same position, the connection between a sprite and a Graphics Box can be represented by the containment of a Sprite Box inside a Graphics Data Box without violating the one object/one location requirement. Sprite Graphics further uses Boxer's spatial metaphor to show the connection between a sprite and its subsprites, making the hierarchy of subsprites within a sprite explicit.

Finally, this design carries Boxer's characteristic explorability into the domain of graphics. Each of the Sprite Graphics applications discussed in this thesis can be interacted

with on a abstract basis. The user need not, for example, know anything about the underlying implementation of the calculator sprite in order to use it as a calculator. The interested user can however easily get inside the logic of the calculator by toggling its Graphics Box. Moreover, because the structure of the calculator is both visible and open to modification, this application gives users the opportunity to learn about the calculator by experimenting with changes to it. The explorability of this application, moreover, is not unique: the logic behind every Graphics Box, no matter how complex, is always available for inspection and modification.

5.2 Suitability for Children

Because the target audience for Sprite Graphics and Boxer is children, this design tries hard to accommodate the inexperienced non-technical user. The characteristics which make Sprite Graphics especially suitable for these users can be loosely grouped into three sets -- Ease of Understanding, Ease of Learning, and Utility.

5.2.1 Ease of Understanding

The most important feature of Sprite Graphics with respect to ease of understanding is the illusion of naive realism which the system maintains. Because of this feature, the user need only understand the information presented on the screen in order to understand the state of the graphics system. Of course, one might say the same of every computer system: if the user could only understand the information on the screen he would understand everything. So the real contribution of Sprite Graphics lies not in making the state of the system visible, but in making the presentation of the information understandable.

One feature of Sprite Graphics which helps make the presentation of graphics objects understandable is its intuitively sensible use of box structure to show the relationship between objects. For example, the link between sprites and graphics regions in Sprite Graphics is shown by containment of a Sprite Box within a Graphics Data Box. The intuitive meaning of "being inside" therefore explains the connection between a sprite and a particular Graphics Box.

The use of standard Data Boxes to represent state variables also contributes to ease of understanding. Because of this choice of representation, the user can easily transfer his understanding of standard Boxer variables to these slightly special variables. Of course, the fact that each state variable bears a suggestive name helps too.

Making the state variables visible also makes the meaning of each sprite command easier to understand. For example, many first-time users of Turtle Graphics are surprised that the command RIGHT does not move the turtle to the right. With Sprite Graphics, users can see that RIGHT changes a sprite's heading rather than its position, and thus quickly correct their conceptual bug.

Moreover, because each state variable can be directly changed, Sprite Graphics users have the opportunity to re-write each primitive command in Boxer to gain a fuller understanding of what it does. One could, for example, make a RIGHT procedure which acted equivalently to the primitive. This procedure would have only the two following lines:

```
INPUT ANGLE
CHANGE HEADING HEADING + ANGLE
```

In fact, even the most sophisticated features of Sprite Graphics can be explained in terms of primitives available to the user. Consider the way the mouse highlights each sprite it moves over. With Sprite Graphics, one can build a sprite which would exhibit similar behavior. This sprite would use FOLLOW-MOUSE to move around until it was pointing to a sprite, which could be determined by SINGLE-TOUCHING-SPRITE. It would then use ENCLOSING-RECTANGLE to get the size of that sprite, and use FORWARD and RIGHT to draw a highlighting rectangle around it. Finally, the mouse-sprite could cause the highlighted sprite to act out a mouse click by using TELL to send the sprite the message "RUN M-CLICK."

5.2.2 Ease of Learning

Four main factors make Sprite Graphics easy to learn:

1. Knowledge about the system can be divided into to small chunks, which can be learned one at time.

2. The Sprite Graphics design attempts to consistently fulfill the user's expectations so he or she can adapt previously learned concepts to each new area.

3. Sprite Graphics has a rich base of default values, so that the beginner can use the system without understanding all its features.

4. A teacher can construct applications for students to learn from such that inconsequential details are hidden from the student.

Each of these factors deserves some more explanation. Consider the first factor. The idea of knowledge coming in small chunks relates to principle of learning in incremental steps. In Sprite Graphics, the user can get started knowing as few as two commands, FORWARD and RIGHT. Given these commands and a Graphics Box with a Turtle inside, one can begin drawing interesting pictures. Of course, with each new Boxer or Sprite Graphics primitive learned, the user can make a more intricate picture. Note, however, that the complete list of basic sprite commands (from section 3.2.1) is only twelve items long.

It is also significant that each of the advanced features of Sprite Graphics can be learned independently. One can learn about sprite-to-sprite sensitivity without knowing about mouse-sensitivity, or about STAMP without knowing TYPE, etc. In the same way, one can learn about each state variable one at a time, as determined by the user needs. This flexibility in the order of learning allows each user to discover the features of Sprite Graphics at his own pace, according to his own interests.

The second factor, which concerns bringing old knowledge to bear on new problems can be seen throughout Sprite Graphics. For example, notice the way sprite shapes are defined. Rather than requiring that the user supply a list of vectors as the shape (which is the underlying representation in this implementation), Sprite Graphics allows the user to define a shape using the sprite commands he or she already knows. Therefore, once a user figures out how to draw an airplane, for example, he or she can easily make an airplane-shaped sprite. Similarly, the design of click boxes allows one define a sprite's mouse operations using any Boxer commands one knows. Note also that each state variable can be created, deleted, accessed and modified in the same ways. So once a child learns to manipulate one state variable, he or she can easily extend that knowledge to the rest of the state variables.

Because Sprite Graphics is well integrated with the rest of Boxer, a child also can make use of everything he or she knows about Boxer. All standard Boxer operations on boxes, like naming, shrinking, and entering, work on Sprite Graphics boxes. Suppose a child knows that

the CHANGE command can alter the contents of a Data Box. Since state variables are Data Boxes, the child would expect that CHANGE should work on state variables as well. And indeed it does. Similarly, Sprite Graphics supports natural expectations about Port Boxes.

The importance of Sprite Graphics defaults is most apparent when working with composite sprites. Imagine if one had to define the meaning of each sprite command for every new arrangement of sprites and subsprites. The complexity of creating a composite sprite under these circumstances would be forbidding. To get around this problem, Sprite Graphics assumes that the user wants composite sprites to be rigid bodies. This assumption makes a great deal of intuitive sense because it makes a composite sprite act mostly like simple sprite with the same shape would. With this system of defaults in place, the user can immediately begin using subsprites without worrying about the details and still have most things work as expected. Of course, the option to define more complicated motions is always available.

Defaults appear in other places as well. Each state variable, for example, has a default value. These default values allow the user to work with sprites without knowing about all the state variables. For instance, the beginner need not know about the state variable SHOWN because the default value is TRUE. The sprite will therefore appear in the Graphics Box even if this variable is missing.

Finally consider the last factor, that teachers can construct examples for students which hide unimportant details from them. Admittedly, it takes more steps to start using Sprite Graphics in Boxer than it does to begin using Turtle Graphics in Logo. However, the teacher can reverse this situation by constructing a Boxer environment for the students to use. For example, a standard introductory set-up might supply a Graphics Box with a single sprite named "Turtle" and a few lines of code to try. With such a set-up, the beginner needs only to point to the lines in order to try them; no typing is necessary. The teacher can also shrink the less important boxes in the logic of an more complicated example, so that when the student explores the logic of the example, the student will naturally be lead to its most important features first.

5.3 Utility

The examples in the preceding chapter should give the reader a good feel for the utility of the Sprite Graphics design. The ratio of value gained to effort expended is particularly high in Sprite Graphics for two reasons. Firstly, as the previous sections explained, the system is easy both to understand and learn. Secondly the knowledge one learns can be used over and over again to make worthwhile things. Look back to the examples in Chapter Four. Note that while the appearance and functionality of each example was very different, the logic behind each application was in many ways similar. The basic ingredients of each of these applications are: Using the click boxes (especially with the FOLLOW-MOUSE command), changing the sprite's shape, and using the TYPE command. After mastering only these three advanced topics, and the basics of the Sprite Graphics system, the student can begin developing his or her own equally interesting applications using mouse sensitivity, icons, and menus. I believe this is the most significant contribution of Sprite Graphics: that it allows the user to construct a wide variety of useful things out of a few simple building blocks.

5.4 Problems

Sprite Graphics, alas, is not perfect. I can foresee several elements of the design which could potentially trouble future users. For example, the fact that user can delete state variables potentially introduces inconsistencies into the system. Suppose a user, for example, deletes the PEN state variable and then uses PENUP to raise the pen. A second user, looking at the system, would expect the sprite's pen to be down, because this is the default. However, the pen in this case would not be down.

There are several potential solutions to this problem. One is to make state variables undeletable. Another is to remove all commands like PENUP or disable them when the state variable is missing. Both these solutions seem too harsh, considering that the problem is not likely to be extremely common. Perhaps the most reasonable alternative is to place deleted state variables in the Local Library Box of the Sprite Box, though I have not had time to work out this possibility in detail.

Another potential inconsistency results from the fact that a sprite's shape is only updated when the SHAPE box is directly changed. Suppose someone refers to a procedure named "Airplane" inside a sprite's SHAPE state variable. At the time the user exits the state variable, the information from Airplane becomes incorporated into the sprite's shape. Now suppose the user changes the definition of Airplane. The sprite's shape will not in this case change until the user again enters and exits the shape state variable.

There is no obvious solution to this problem. Sprite shapes must be converted into another representation before use so that the system will draw reasonably quickly. Moreover, no mechanism currently exists in Boxer for noticing when a procedure definition changes, because the Boxer is designed to be an interpreted language.

Another problem with Sprite shapes is that certain shapes are difficult to produce using sprite commands. For example, it would be very difficult to get the shape of a filled-in triangle with the current set-up. Here's one solution to this problem which I have not had time to implement: Firstly, add a new sprite command, let's call it "INCLUDE," which would work like TYPE except that its argument would be a Graphics Box. INCLUDE would cause the sprite to copy its argument's bit-map into the sprite's shape. With this command the user could specify complicated shapes as bit-maps rather than as line drawings. Secondly it would be useful to have a command which would convert a Graphics Box to a two-dimensional array (Data Box) of binary numbers and visa versa. This command would allow the user to specify bit-map shapes more directly. In addition, it would be useful to have a FILL command which could fill in any closed contour in a Graphics Box.

The fact that Graphics Data Boxes export all their variable bindings could also be a problem. Because Graphics Data Boxes are often not visible, the user may not be aware of all the names which a particular Graphics Data Box contains. Thus the unsuspecting user might encounter a name conflict between a binding in a Graphics Data Box and a binding in its surrounding box. This problem might be lessened by making the exporting of bindings from Graphics Data Boxes more discretionary, though this might also make these boxes more complicated to use.

Graphics Data Boxes also fail in a relatively minor way to maintain naive realism because these boxes maintain an internal state variable to remember whether the box is in

WRAP or WINDOW mode. This state variable currently is not made explicit, though perhaps it should be.

Another parameter of Graphics Boxes which is difficult to change is their size. Maybe these boxes should be mouse sensitive in a way that allows their borders to be easily re-shaped.

One final problem with the current design is worth noting. The user should have a better way to find the a particular sprite's Sprite Box. One would like to be able to point to a sprite in a Graphics Box and say "Show me your logic." In the current design the only tool the user has to make connection between a sprite and its logic is the FLASH-NAME primitive. A possible fix for this situation would be to allow Sprite Boxes to pop-up in temporary windows on demand, though Boxer does not presently support any temporary window capability.

5.5 Suggestions for Future Work

In the course of developing Sprite Graphics several general issues have come to light which are worthy of further investigation, but are outside the scope of this thesis project. I will briefly outline each of these issues and my thoughts on them in remainder of this chapter.

5.5.1 Animation

One capability that a good graphics system should have, that Sprite Graphics is noticeably lacking, is the capability to create animated pictures. Two problems must be tackled in order to give Boxer the capability displaying interesting animations. Firstly, one must develop a good way of describing the velocity of sprites -- making sprites move by repeated use of FORWARD is much to obscure to describe interesting trajectories. Perhaps the answer to this need could take the form of a velocity state variable along with sprite commands which alter a sprite's velocity. Secondly, one must develop a way of introducing multi-tasking into Boxer so that several objects can move simultaneously. One might do this either with special programming constructs, special state variables, or perhaps with when-demons.

5.5.2 One-of Boxes

In several places in the Sprite Graphics design there are boxes which can contain only a certain range of values. For example, a sprite's PEN state variable can only have the value UP, DOWN, ERASE, or XOR. Yet Boxer offers no way of making the possible choices explicit. One would like a new functionality which I will call a "One-of Box" although it does not necessarily have to be implemented with a new box type. A One-of Box would contain a list of possible values of which exactly one would be marked at a time. The Boxer interpreter would see only the marked value. The functionality provided by such a box seems general enough to be included as a feature of Boxer, rather than just a feature of Sprite Graphics.

5.5.3 Expandable Graphics Objects

The Icon Programming System briefly touched on a feature of Sprite Graphics that could potentially become its most significant contribution: the ability to build graphics objects which can be displayed at many different levels of detail. Because Sprite Graphics can alternately display only a composite sprite's top-level shape or only the shapes of its subsprites, the user can easily build a graphical object with hierarchically nested levels of detail. In this way, sprites can act like transparent overlays, but with much more flexibility. This feature might allow teachers to draw diagrams which communicate complicated ideas more clearly and concisely than is possible with any other medium. An future inquiry might test how well this new communicational strategy works with an appropriate type of drawing, for example a flowchart.

5.5.4 The Status of New Boxes

Although this design has attempted to fully integrate Sprite, Graphics, and Graphics Data Boxes into Boxer, some grey areas still exist. For example, should all Data Box commands work on Sprite Boxes? Currently, they do, though there is no good reason why they should. In fact, allowing users to take apart Sprite Boxes is probably a bad idea because it leads them to depend on the state variables occurring in a standard order. Ordering the state variables contradicts the presentational flexibility which Sprite Boxes are designed to offer and subverts the purpose of naming the boxes. Other related issues are: How does one

build Sprite and Graphics Data Boxes under program control? Can procedures return or pass around Sprite Boxes? How does one connect Sprite Boxes to Graphics Data Boxes or to other Sprite Boxes under program control?

5.5.5 Better Pointing Devices

In my experience, most beginning users find the mouse very difficult to use. Not only do they have trouble controlling the mouse, but they also have trouble remembering which button does what. In addition, the duality of having a mouse and a cursor is very confusing. Although the magnitude of each of these problems can be reduced, it would be interesting to see a study made comparing the effectiveness of various alternative pointing devices.

5.6 Conclusion

In my opinion, the Sprite Graphics design is quite worthy of becoming a permanent addition to Boxer. Sprite Graphics brings to Boxer not only the functionality of Turtle Graphics, but also the ability to build interesting graphic interfaces. In addition, Sprite Graphics allows the user to graphically model many interesting real world objects. Sprite Graphics moreover gives the user these functionalities in such a way that a school child or any similar non-technical user can understand and learn to build complicated graphics systems. Therefore Sprite Graphics offers these users substantial utility. Finally, note that Sprite Graphics is compatible throughout its design with the principals behind the design of Boxer.

Sprite Graphics, however, is by no means the complete solution to the expected needs with respect to graphics of future Boxer users. Further research is needed to correct some of Sprite Graphics problems, to evaluate the usefulness of Sprite Graphics innovations, and to build other desirable features, like animation, into the design.

References

1. Abelson, Harold, Apple Logo, Byte/McGraw-Hill, Petersborough, NH, 1982.

2. Abelson, H., and diSessa, A., Turtle Geometry, MIT Press, Cambridge, MA, 1980.

3. diSessa, Andrea, Notes on the Future of Programming: Breaking the Utility Barrier, Draft for D.A. Norman and S.W. Draper (eds.) User Centered Design: New Perspectives on Human-Computer Interaction, date of draft 1985.

4. Lay, E., Klotz, L., and Neves, D. ,Boxer Users Manual, Version 5.3, Available from the Educational Computing Group at M.I.T., 1985.

5. Papert, Seymour, Mindstorms, Basic Books, New York , 1980.