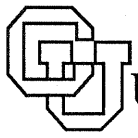
# SchemePaint: a Programmable Application for Graphics

Michael Eisenberg

CU-CS-587-92 April 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

# SchemePaint: a Programmable Application for Graphics

Michael Eisenberg
University of Colorado at Boulder

## Abstract

*Programmable applications* are software systems that seek to combine the best features of direct manipulation interfaces (accessibility, learnability, aesthetic appeal) with the best features of interactive programming environments (extensibility and expressive range). This paper describes such a programmable application for graphics. The program, named *SchemePaint*, combines a "Macintosh-style" paint program interface with a Scheme interpreter that has been specifically enriched for graphics programming. Besides describing the particular features of SchemePaint, this paper also touches upon some of the more important issues involved in creating programmable applications in general: the construction of "embedded sub-languages" within a programming environment, the extension of an application through language/interface libraries, and the development of methods by which a direct manipulation interface and a programming language can achieve some measure of symbiotic cooperation.

# SchemePaint: a Programmable Application for Graphics

Michael Eisenberg
University of Colorado at Boulder

## 1. A Scenario

Visit any home computer store and you will see a shelf or more of graphics applications—each one advertising that it is more featureful, more powerful, more usable than the others. Glancing over the applications, it is indeed hard to choose between them: they all present a cheerfully busy screen with a top row of menus, side panels of icons, multiple windows, perhaps a color wheel, and (the final bulleted item on every package) "much, much more."

Imagine, then, that an artist—wishing to exploit the artistic range of his newly-purchased home computer—walks into the store and picks out the first graphics application that catches his eye. This application, which we'll call *EZ-Paint*, advertises that it is user-friendly; and the artist (being after all a user) decides that this must be the right package for him. Proud of his purchase, the artist returns home and quickly becomes enthralled with his new program. Within an afternoon, he has placed all manner of rectangles and ellipses and spline curves on the screen, and by the end of the first evening has even drawn his first picture.

Several days later, the artist's daughter (who is in fourth grade) returns home from school and shows her father a picture that she made that afternoon in Logo class: a spiral. The artist is intrigued by the design and sets out to create a spiral in EZ-Paint. Surely, he thinks, this must be a standard project... but after hunting through the interface (and even as a last resort checking the manual) he discovers that there is no such thing as an EZ-Paint spiral. Well, perhaps there is some way to add to the interface?... Apparently not. "User-friendly" only goes so far, it seems.

As the weeks go by, the artist's catalog of frustration begins to expand. He can't create a maze; can't tile the plane with octagons and squares; can't generate a random walk; can't make a fractal, a sine wave, a Lissajous figure, a cycloid, a Moire pattern. The program that once seemed so enjoyable has—by virtue of its strengths and the high expectations they engendered—become that much more disappointing in the long run.

Perhaps one of those other programs at the store would be an improvement, the artist decides; and this time he purchases a larger package labelled *XTra-Power-Paint*. This program is much more imposing than the previous one; it includes more features, more (and longer) menus, submenus, sub-submenus, scrolling boxes, and even macros which allow the artist to give reusable names to sequences of mouse operations. But most of the projects that couldn't be done before with EZ-Paint still can't be done with XTra-Power-Paint. In some sense, in fact, the larger program is the inferior one: it's hard to find out just what it can and can't do. Days can be spent discovering that XTra-Power-Point isn't able to handle some particular project.

EZ-Paint, XTra-Power-Paint, and their non-fictional cousins are actually more alike in spirit than their designers might wish to believe. In particular, they all share the same fundamental design flaw: each is predicated on the notion that users cannot and will not write programs. Although it isn't clear why this should be true, it is nonetheless the fundamental tenet of the reigning philosophy of "user-friendly" application design. No matter that what the user wants to express could conceivably be written in a short, simple, expressive, interactive program; no matter that in fact fourth graders *do* write such programs to generate (among other patterns) spirals. It remains a matter of faith that users just can't attain the same level of sophistication as software designers. This condescending view toward the audience of applications users may or may not be mistaken; but it is certainly unsubstantiated by anything like hard evidence.

## 2. Programmable Applications: a Strategy for Software Design

This paper describes a program named *SchemePaint*. SchemePaint is designed as a working prototype of a *programmable application*: that is, it combines the learnability and accessibility of a direct manipulation interface (of the kind typified by "Macintosh-style" programs) with the power and expressive range provided by a domain-enriched interactive programming environment. By providing users with the ability to create graphical effects both "by hand" and "by program," SchemePaint—even as prototype—provides its users with a range of expression beyond the capabilities of even the most baroque non-programmable applications currently available.

The remainder of this section is devoted to a more thorough explanation of the design philosophy behind programmable applications in general (and SchemePaint in particular). We first describe some of the problems associated either with "pure" direct manipulation interfaces or with "pure" general-purpose programming environments; we then discuss some of the design issues that arise in trying to construct programmable applications and thereby steer a middle ground between these two extremes.

Section 3 is an overview of the SchemePaint system; this is followed by four sections describing specific aspects of the program. Specifically, Section 4 details the SchemePaint interface; Section 5 the associated "graphics-enriched" language; Section 6, some of the more experimental features of the system that attempt to achieve a measure of symbiotic cooperation between the language and interface portions of the program; and Section 7, several special-purpose "library" extensions to the program. We conclude in Section 8 with several illustrations of work done in SchemePaint; a discussion of related and future work; and some final speculations on potentially interesting research issues for developing programmable applications in general.

### *2.1 Direct Manipulation Interfaces (and Their Disadvantages)*

"Direct manipulation" is a broad term whose meaning is probably better indicated through examples than through any attempt at a formal definition. (Shneiderman{19} and Hutchins *et al*{13} provide informative discussions of the term.) For our own purposes, we can simply stipulate that an application has a "pure direct manipulation interface" if that interface is entirely

composed of (say) menus, icons, dialog boxes, buttons, sliders, and other relatively simple mouse-operated elements. The Apple Macintosh computer is a good source of illustrations: the Macintosh operating system and most of its more popular software packages may be comfortably classified as direct manipulation according to the informal criteria of the previous sentence.[1]

Direct manipulation interfaces have some important strengths, including most notably learnability and ease of use. Among graphics applications, *MacPaint*{S5} is a good example: in a matter of hours, one can pick up the rudiments of MacPaint usage and embark on reasonably sophisticated projects. Clearly this interface has been designed with a remarkable combination of care, ingenuity, and programming skill.

Problems arise with such interfaces, however, when they are subject to long-term, serious, creative use. Perhaps a MacPaint user has read a book on fractals and would now like to create pictures that freely combine fractals with other MacPaint features; or perhaps some slight variation of the standard MacPaint ellipse (say, a parabola) is desired; or perhaps one would like to draw a figure of a person turning a (correctly scaled) sequence of gears for a physics text. In every case, if the appropriate tools are not immediately available in the MacPaint interface, the user has no recourse: if the feature is absent now, it will be absent forever.

The point here is not merely that some specific features are absent from one specific paint program. The problem is rather a near-universal (and nearly always major) flaw in "pure" direct manipulation systems. The range of expression provided by clicking/dragging/icon-selection is simply too impoverished to accommodate the imagination of long-term users. Whenever a user, intrigued by the wonderful features of some program, wishes to invent some variant or extension of those features, he or she will inevitably run up against the limitations of an interface that sacrifices the creation of new concepts to selection among designer-specified concepts.

What are the aspects of programming environments that are missing from "pure" direct manipulation systems? Briefly, these fall under three headings:

- *Control Structures.* Virtually every programming language provides a "standard" set of control structures: conditionals, loops, recursion, sequencing. (More exotic languages might expand on this list and include (say) parallel-programming constructs such as "forking" and "joining.")

- *Compound Data Objects.* Programming languages include methods for combining data objects into larger groups: lists, structures, and arrays are typical examples.

- *Abstraction and parametrization mechanisms.* Perhaps most important of all, programming languages include techniques for *naming* data objects and procedures (and for likewise naming

[1] Of course, "Macintosh-style" programs are now available on other home computers and on high-end workstations as well. Thus, the discussion of "direct-manipulation" in this paper should not be interpreted as applying solely to Apple machines and software.

parameters for procedures). These named entities can then be used in the definition of other procedures and objects. Without this facility, we cannot create more complex concepts than those that we began with.

In the absence of these features—which may be said to operationally define the concept of programmability—users of direct manipulation applications find themselves hampered in one project after another.
The application designers, when confronted by those users' complaints, tend to respond by augmenting their programs willy-nilly, slapping on an ever-increasing collection of *ad hoc* icons and menu choices. Thus, the fictional EZ-Paint program (from the scenario described earlier) might eventually become *EZ-Paint 5.0*, including spirals, sine waves, reflections in mirrored surfaces, digital logic icons, international traffic symbols, and so on. These additions tend to render the program more and more confusing, and eventually they hurt the program rather than help it (one might think of the fictional XTra-Power-Paint as a reasonable analogue to EZ-Paint 5.0). More to the point, they don't really address the users' problems: no finite set of additions possibly could. Having noticed a (logarithmic) spiral choice on the menu, the user suddenly realizes that what he wants is an Archimedean spiral, or an Ionian column design, or a spider web. Another artist never uses spirals but she wonders why cycloids still haven't been included in the new release.

In short, users have a nagging habit of being creative; and they therefore need a medium in which new concepts can be defined, altered, saved, extended, and combined. They need programming languages.

### *2.2 General-Purpose Programming Environments (and Their Disadvantages)*

The discussion above might be read as an argument for the use of programming environments as opposed to direct manipulation applications. Why not, for example, provide our hypothetical artist with a Pascal or Lisp programming environment instead of a program like EZ-Paint? If direct manipulation interfaces are so limited, why shouldn't artists work in the far more powerful medium of a programming language?

There are two major arguments against the use of "pure" (general-purpose) programming environments in this context. The first is that direct manipulation interfaces, while insufficiently expressive for most applications, do have their characteristic strengths. Consider the problem, for instance, of trying to write a program to draw a horse (this type of problem is not unfamiliar to elementary-school Logo programmers). This is simply a task that is more easily accomplished by direct manipulation (using a mouse, or perhaps light-pen) than by writing code. In other words, there is a role in graphics applications (indeed, in virtually every type of application) for interface tools that take advantage of essentially extra-linguistic skills that users have; and these tools do not generally come pre-packaged with general-purpose programming environments.

The second problem with programming environments has to do with that phrase "general-purpose." An artist using a programming language wants to deal with data objects such as points, lines, polygons, colors, brush styles, and

so forth (and with procedures that operate on these data objects). Thus, there is a level of domain-specificity (or "domain-enrichment") that is required in a graphics language, and that is missing in standard general-purpose environments. Typically, programming environments are seen as tools for *programmers* (particularly, systems programmers), as opposed to professionals with other interests; the resulting language constructs are thus at a relatively low level with respect to other domains. Programming environments within applications need to be structured so that interesting tasks within the domain of the application can be expressed within two or three lines of code.

### *2.3 Design issues for Programmable Applications*

The previous two subsections argued for the incompleteness, within application design, of an approach that focuses either on "pure" direct manipulation or "pure" general-purpose programming environments. In this section, we briefly discuss some of the design issues that arise in attempting to combine the respective strengths of the two approaches within applications.[2]

#### *2.3.1 Choosing a Language*

One of the primary design choices in constructing a programmable application is deciding on the nature of the programming language that will accompany it. In most popular applications that achieve a measure of programmability (*Mathematica*{S7}, *Director*{S2}, and *4th Dimension*{S3} are good examples), the language provided is a brand-new application-specific language, complete with its own syntax and vocabulary. In contrast, one can imagine basing an application on a domain-enriched version of some existing general-purpose language: *AutoCAD*{S1}, which is based on Lisp, is perhaps the most famous example of this approach.

There are arguments to be made in favor of either approach; time and space do not permit a thorough airing out of the issues here. Briefly, one might argue that an application-specific language can be constructed with special attention to the syntactic and control structures appropriate to a given domain; on the other hand, using an existing language allows for a certain breadth in application design—users who know the rudiments of (say) Pascal can work with *any* Pascal-based application.

SchemePaint, as we will explain in the following section, may be viewed as fitting within the second camp: its primary programming language is the Scheme dialect of Lisp. The program does, however, include a rudimentary Basic interpreter for users who are more comfortable with this language; a Logo interpreter is also under development. The position thus argued for here is that the particular choice of language is a second-order issue in programmable application design; an application can indeed profitably include a choice of several languages. Providing the user with *some* language—whether application-specific or a domain-enriched dialect of any one of several existing languages—is much more important than the question of which particular language to include.

---

[2] Further discussion can be found in Eisenberg {8}.

### *2.3.2 Constructing a "Domain-Enriched Dialect": a Case Study*

Several times in the course of this paper we have referred to the notion of constructing a "domain-enriched" dialect of some general-purpose language. This simple phrase hides a wealth of delicate design considerations which can only be hinted at in this discussion.

The notion of language-building as a technique for application design is based on the software engineering philosophy described by Abelson and Sussman{2}. These authors present a portrait of the programming process in which the key engineering questions are essentially those used in language construction: i.e., what are the fundamental (primitive) data objects, procedures, means of combination, and means of abstraction necessary to build an interactive language in which a given domain may be fruitfully represented.

As a miniature example of this language-construction process, we can consider a portion of the actual SchemePaint language: namely, that portion directly related to the manipulation of colors. In a sense, this collection of object types and procedures embodies a "sub-language" within the larger SchemePaint language. The fundamental object type in this case is a *color-object*, represented as a list of three floating point numbers in the range 0-1 and corresponding to the R, G, and B components (respectively) of the given color. Thus, the color-object represented by the list (0. 0. 1.) would correspond to the color blue.[3]

Starting with this representation of a color object, our "color language" must now include a data-object constructor, and selectors that access parts of a given data-object:

> `make-schemepaint-color-object` *red green blue* **procedure**
> This procedure takes as arguments three floating point numbers between 0. and 1. and returns the appropriate color object.
>
> `get-schemepaint-color-object-red` *color-object* **procedure**
> `get-schemepaint-color-object-green` *color-object* **procedure**
> `get-schemepaint-color-object-blue` *color-object* **procedure**
> These three procedures each take a single (color-object) argument and return the appropriate component of that object (represented as a floating point number between 0. and 1.).

There is also a type-predicate for color objects:

> `color-object?` *object* **procedure**
> This procedure takes a single object of any type and returns `true` if the object is of type color-object (and `false` otherwise).

---

[3] Obviously, myriad other data representations are possible (e.g., we might have chosen a color representation based on hue-saturation-brightness). Although space considerations do not permit a discussion of this matter, such choices among alternative object representations are clearly fundamental aspects of the overall language construction process.

And finally, there are procedures that can reset the current foreground (pen) and background colors to a given color object.

`set-pen-color!` *color-object* **procedure**
This procedure, when called on a color-object, changes the current foreground color to the desired value. If the pen is now used to draw lines or points, they will appear on the screen in the specified color.

`set-background-color!` *color-object* **procedure**
This procedure, when called on a color-object, changes the default background color to the desired value. The next time the screen is cleared, it will appear as a solid background of the specified color.

This set of procedures constitutes the backbone of our color language. As it stands, this "language" is still quite sparse; and indeed, the actual "color sub-language" of SchemePaint is significantly larger. But there is a larger point to be made: because we can create our own (Scheme) procedures that build from the framework provided here, even this small core can serve as the basis for extensive experimentation. For example, we might begin by assigning names to two "typical" colors:

```
(define red
  (make-schemepaint-color-object 1. 0. 0.))

(define blue
  (make-schemepaint-color-object 0. 0. 1.))
```

Now suppose we would like to write express the idea of combining two colors by averaging together their respective R, G, B components. We could easily write a new procedure for this purpose:

```
(define (average-between-colors color1 color2)
  (make-schemepaint-color-object
    (average (get-schemepaint-color-object-red color1)
             (get-schemepaint-color-object-red color2))
    (average (get-schemepaint-color-object-green color1)
             (get-schemepaint-color-object-green color2))
    (average (get-schemepaint-color-object-blue color1)
            (get-schemepaint-color-object-blue color2)))))
```

We could now use this procedure to create a new color that is the "average" of red and blue:

```
(define purple (average-between-colors red blue))
```

And we could average the new color with some other one:

```
(define reddish-purple (average-between-colors red purple))
```

In point of fact, SchemePaint includes a more general-purpose procedure that subsumes the utility of `average-between-colors`; but again, the key issue is that we are able to use our overall Scheme language to build complexity where we need it from a simple set of "primitive" concepts. If the notion of "averaging

two colors" had not been included in our core set, we could still create it ourselves, by writing a relatively straightforward procedure; we needn't depend on some unseen omnipotent interface designer to include this notion in an advanced-level menu.

Even the tiny example of this section suggests some of the design issues that are involved in creating an embedded domain-specific language. We might wonder, for instance, if we have chosen an appropriate (elegant, efficient, learnable) representation for color-objects: perhaps a hue-lightness-saturation representation would be preferable on a number of different grounds. Or we might ask what additional procedures ought to be pre-supplied along with the very minimal core set shown thus far. These sorts of issues—the pragmatic, operational aspects of language construction—are crucial subjects for research in building programmable applications.[4]

### *2.3.3 Interface/Language Cooperation*

Programmable applications, as already noted, seek to combine the best features of direct manipulation interfaces and domain-enriched programming environments. The reason for attempting this combination is not that there is some law of software architecture that dictates that the marriage should always work; rather, it is simply that we wish to build maximally expressive applications, and in doing so we should feel empowered to employ the most appropriate methods available. If the user wishes to perform some tasks (e.g., drawing a Lissajous figure), our application should probably include a programming language; for other tasks (drawing a face) a direct-manipulation design is superior. If we now wish to construct an application that can accomplish a range of such tasks, then the application will quite reasonably contain elements of both design paradigms.

The simplest strategy for combining these two "halves" (interface and language) within one system would be to design them independently. In a graphics application, for instance, we might first imagine what sort of direct manipulation system would be concocted by our worst nightmare of a "user-friendly" designer (the hypothetical EZ-Paint program might be a starting point); then we might imagine what kind of graphics language would be constructed by a designer paying no attention to visual interface whatever. Joining these two sub-projects together would produce a "zeroth-order" programmable application.

In fact, the applications constructed by this methodology would quite likely be more expressive than virtually anything available in the current commercial software world. Even so, this "parallel-track" design strategy ignores the exciting possibilities of collaboration between language and interface. Rather than simply tack these two elements together, we can seek to promote a creative symbiosis between them.

As a very rough taxonomic scheme for discussing interface/language cooperation, we can divide the range of collaborative strategies into two broad categories according to the direction of influence. That is, we have (on the one hand) language constructs that affect interface behavior, and (on the other)

---

[4]We will return to this topic again in Section 8 of this paper.

interface operations that create or alter the objects or procedures of the language. The "macros" available in some graphics applications are reasonable illustrations of the latter strategy: these are sequences of interface operations that can be given names (and hence in some sense can "create new procedures").[5] An example of the former strategy can be seen in AutoCAD{S1}; here, AutoLisp procedures can be written that will add new command constructs to the system.[6] SchemePaint includes experimental constructs of both "directional" types; these are described in Section 6 later in this paper.

## 3. SchemePaint: an Overview

SchemePaint is a programmable graphics application written in the Lightship Software's MacScheme{S6} dialect of Lisp. The program is designed to run on Apple Macintosh computers equipped with color screens (such as the Macintosh II).[7]

When the application is started, our screen appears as in Figure 3.1. There are three windows visible: the **Pen** (or "palette") window, the **SchemePaint** (or "canvas") window, and the **transcript** (or "interpreter") window. Briefly, the first is used to select among various drawing modes (pen-width, color, text, and so forth); the second is the window in which graphics work appears; and the third is used to evaluate Scheme expressions. The interpreter window is a standard feature of the MacScheme environment, as are the first four menus (excluding the "apple" menu) in Figure 3.1; because SchemePaint is in fact not only written but embedded in MacScheme, it can take advantage of all the powerful features that accompany the MacScheme environment (including file system, editor, debugger, and byte code compiler).

---

[5] The classification is really only approximate in that the macros in question generally do not include either parameters or control constructs; thus applications with only macros of this type cannot truly be said to be programmable.

[6] A similar example would be the way in which new commands and behavior in the Emacs editor {20} can be created via Lisp procedures. In both this case and that of AutoCAD, the interface constructs affected are not of the (simple, manipulable) type generally classified as "direct manipulation." HyperTalk {} is perhaps another example of a language that is used to affect (really to create) interface constructs; more discussion of this program vis-a-vis the programmable application concept can be found in Eisenberg {8}.

[7] Currently, SchemePaint consists of about 30 source files, and runs in a minimum of approximately 1M of memory. The program is available from the author, along with documentation, for the cost of copying and postage.

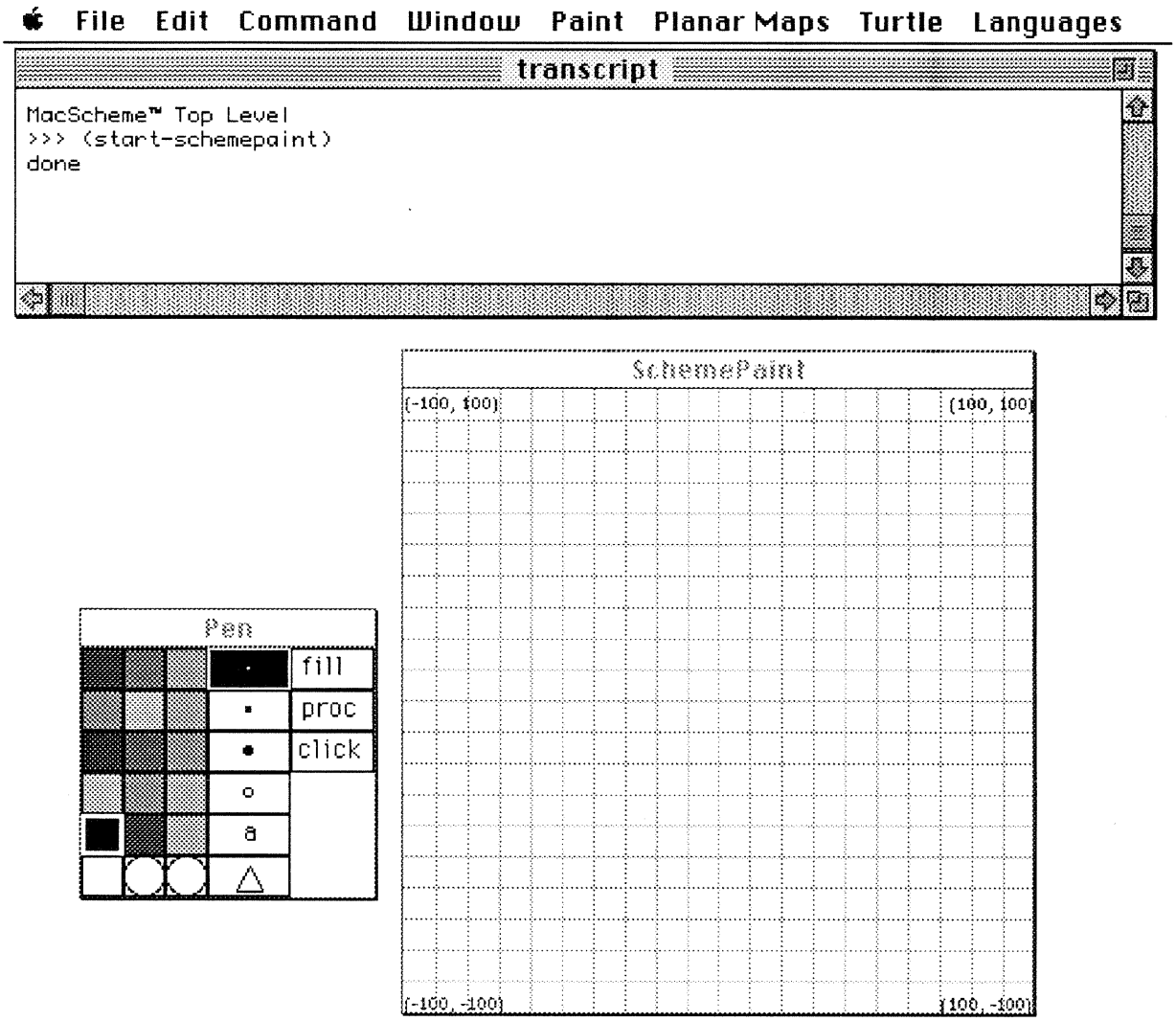



*Figure 3.1*: The initial SchemePaint screen.

The last four menus visible in Figure 3.1 are specific to SchemePaint. The **Paint** menu is mainly devoted to setting global variables (such as the x,y-coordinates associated with the corners of the canvas window, or whether a background grid will be displayed); the **Planar Maps** menu can be used to create and experiment with maps (both linear and nonlinear) from $\mathbf{R}^2$ to $\mathbf{R}^2$; the **Turtle** menu includes a few commonly-used turtle commands; and the **Languages** menu permits the user to select among several different language interpreters.[8] These last four menus and their associated commands

[8] Currently, only the Basic interpreter is implemented, and this is rather rudimentary. A (likewise simple) Logo interpreter is under development. These will be described in Section 8, but otherwise when programming issues are mentioned in this paper it will be assumed that language under discussion is Scheme.

will be described in subsequent sections of this paper; the four MacScheme menus will be mentioned occasionally, as needed.[9]

In the next four sections, we will focus on specific aspects of the SchemePaint system: its interface, language, special features, and library mechanism, respectively. Before proceeding to this more detailed examination, however, the reader is encouraged to peruse the Color Plates that accompany this paper. These pictures are discussed at some length in Section 8; but even in the absence of explanation, they should provide an intuitive portrait of the program's expressive range.

## 4. The SchemePaint Interface

This section introduces the SchemePaint program through its most straightforward interface features. Several more advanced features are best described in conjunction with elements of the SchemePaint language; in subsequent sections, we will return to examine these more language-oriented interface elements.

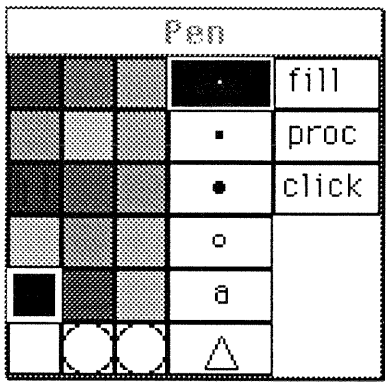



*Figure 4.1*: The palette (**Pen**) window in SchemePaint.

### *4.1 The Palette Window*

Figure 4.1 depicts SchemePaint's palette window. Loosely, the three columns on the left-hand side of the window are used to select a particular pen (foreground) color; the two columns at right present a selection of pen modes. The modes depicted in the fourth column correspond to (reading from the top down) three pen-widths for "normal drawing"; an "erase mode", a "text mode", and a "turtle-dragging mode"; the selections in the rightmost column correspond to "fill mode", "programmable-color-procedure mode", and "programmable-click mode". These last two will be described in Section 6, and the turtle mode in Section 5; the rest are explained below.

---

[9] A thorough description of the MacScheme interface can be found in the MacScheme documentation.{S5}

*4.1.1 Drawing and erasing lines; filling regions*

The most typical use of the pen is simply to draw lines on the canvas window by dragging the mouse. This is the "normal drawing mode", and it is depicted in Figure 4.2. The three upper selections in the fourth palette-window column correspond to three pen-widths (of one, three, or five pixels). The line drawn is in the currently selected foreground color.

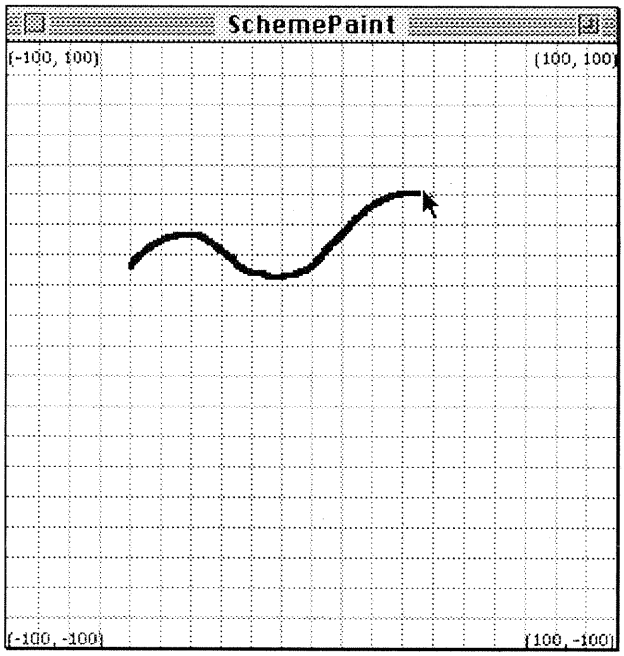



*Figure 4.2*: Drawing a line in "normal drawing mode."

Selecting "erase mode" effectively sets the pen color to the current background color and retains the current pen width.

When the pen is in "fill mode", then double-clicking on some canvas window point will cause the contiguous region with the same color as that point to be set to the current foreground color.

*4.1.2 Text mode*

When the pen is in "text mode", the mouse can be used to set a cursor position on the canvas window; starting at this position, the user may type text directly on the screen. (The text color is the currently selected foreground color.) There is a small selection of available fonts and type sizes; these are selected via SchemePaint expressions to be described later. Figure 4.3 depicts the use of the pen in text mode.

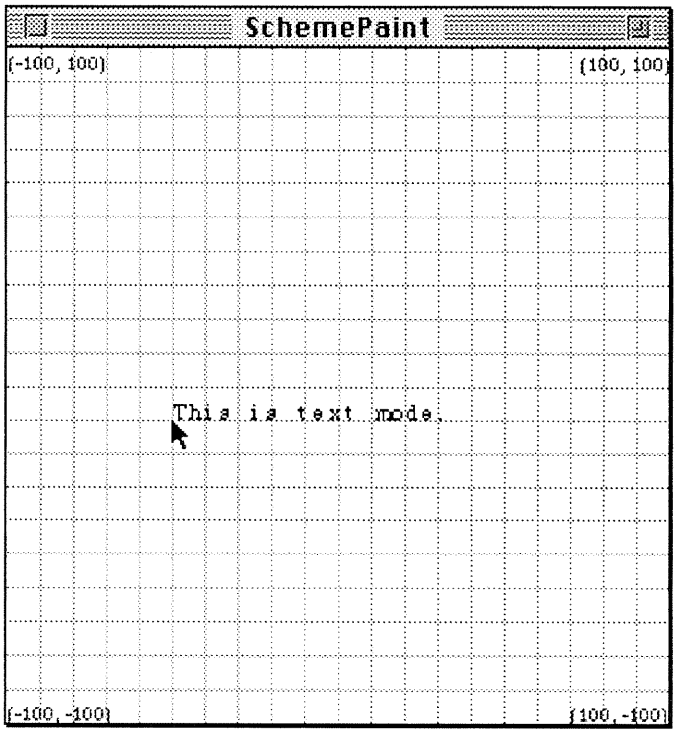



*Figure 4.3*: Typing text in text mode.

### *4.2 The Paint Menu*

SchemePaint's **Paint** menu is a sort of general-purpose parameter-setting menu for the program. The menu selections available (and their meanings) are listed in Table 4.4.

## 5. The SchemePaint Language

Thus far, we have examined SchemePaint through its interface features only. These might be regarded as constituting a useful if skeletal "Macintosh-style" application interface. In this section we consider the second major element of the system—namely, the graphics-enriched Scheme environment that adds programmability to the interface features already described.

The SchemePaint language consists of several major packages, or "sub-languages," which collectively augment the core MacScheme system. These sub-languages focus on the following topics:

- Turtle Graphics
- Planar Maps
- Colors
- Miscellaneous features

| Menu Selection | Notes |
|---|---|
| *Clear Canvas* | If "grid mode" is active, a gray background grid is also drawn when the canvas window is cleared. |
| *Toggle Polygon Fill* | When polygons are created (either by direct manipulation or by SchemePaint expressions) and displayed on the screen, they are shown by default either as filled or unfilled polygons. This selection toggles the current default setting. |
| *Toggle Grid Mode* | Determines whether a gray background grid is drawn when the canvas window is cleared. |
| *Reset Canvas Coords* | Allows the user to change (via dialog box) the region of the Cartesian plane to which the canvas window corresponds. |
| *Refresh Palette Window* | Redraws the Palette window. (Necessary because SchemePaint does not currently refresh window contents when they are hidden and later redisplayed.) |
| *Background Color* | Allows the user to change (via dialog box) the current background color. The change will not be seen until either the canvas is cleared or the pen is used in "erase" mode. |
| *Copy Region* | Allows the user to select a rectangular canvas window region with the mouse and copy it to some region of equal size elsewhere on the canvas window. |
| *Save Picture* | Allows the user to save the current canvas window contents as a Scheme file (whose name is chosen via dialog box). |

*Table 4.4*: **Paint** menu contents.

With few exceptions, each of these sub-languages adheres to the usual conventions of Scheme syntax and evaluation (to rephrase this in Scheme terminology: there are only a few special forms among the new additons). Thus, a user who knows the rudiments of Scheme's syntax and evaluation rules should be able to exploit the enhanced SchemePaint vocabulary without any difficulty.

In the remainder of this section, we will examine each of the four sub-languages in turn.

### *5.1 Turtle Graphics*

SchemePaint includes a full set of the standard turtle-graphics procedures usually associated with Logo systems. (The most important of these are listed

in Table 5.1.) In addition, the **Turtle** menu presents a selection of commonly-used turtle-related commands (including `home`, `pen-up/down`, and `show/hide-turtle`).

As an example of how turtle commands can be used, suppose we wish to have the turtle draw an octagon. To accomplish this we could evaluate the following Scheme expression:

```
(repeat 8 (fd 25) (rt 45))
```

The shape drawn is shown in Figure 5.2. We might now wish to define a more general octagon-drawing procedure:

```
(define (octagon side)
  (repeat 8 (fd side) (rt 45)))
```

This new procedure can be used to draw an octagon of any side-length desired.

Because SchemePaint is embedded in Scheme, it can take advantage of some of the more powerful semantic features of that language—notably including the use of procedures as first class objects. For instance, we can edit our original `octagon` procedure so that it now takes two arguments—a side argument as before, and a "turtle-move-procedure" argument:

```
(define (octagon turtle-move side)
 (repeat 8 (turtle-move side) (rt 45)))
```

| Command | Description |
|---|---|
| (fd *steps*)<br>(bk *steps*) | Moves the turtle forward/back the distance specified by *steps*. |
| (rt *angle*)<br>(lt *angle*) | Turns the turtle right/left by *angle* (measured in degrees). |
| (turtle-penup)<br>(turtle-pendown) | Sets the state of the turtle pen. |
| (setpos *x y*)<br>(seth *heading*) | Sets the position (heading) of the turtle to the desired value. |
| (getpos)<br>(geth) | Returns the current position (heading) of the turtle. |
| (show-turtle)<br>(hide-turtle) | Renders the turtle visible (invisible). |

*Table 5.1*: Turtle-Related Commands

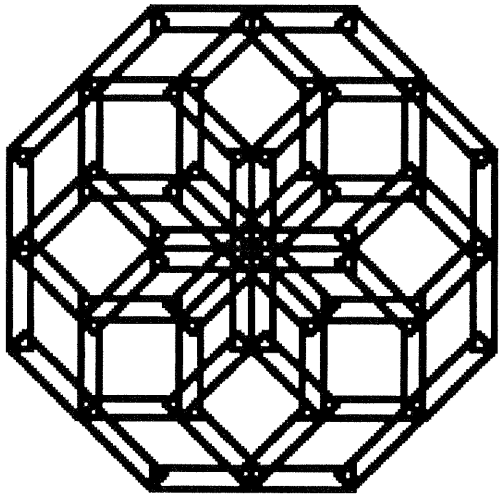


*Figure 5.3*: A turtle-drawn figure.

### *5.1.1 The Turtle Menu; Turtle-dragging mode*

Before leaving the topic of turtle graphics, two elements of the SchemePaint interface deserve mention. The first is the Turtle menu; this menu presents a selection of turtle operations that are so common that they constitute "cliches": moving the turtle to the origin, picking up (or putting down) the turtle's pen, rendering the turtle visible (or invisible). Because these operations are so often used, they are reasonable candidates for the simple selection operation provided by a menu: that is, rather than evaluate the expression

```
(turtle-penup)
```

the user need merely select the appropriate menu choice for this operation. (Note, however, that we have not eliminated the language construct for this operation; the use of the menu is an optional feature.)

The second SchemePaint interface feature worth noting here is "turtle-dragging mode," denoted by the box with the turtle-icon in the palette window. When the mouse is placed in turtle-dragging mode, it can be used to drag the turtle interactively about the canvas window: by pressing the mouse button on the turtle's center and dragging the mouse to a given point, the turtle will follow the mouse pointer (without drawing lines). In this mode, the mouse may also be used to turn the turtle: by pressing the mouse button on the turtle's tip, the turtle will point toward the mouse position (without moving its own position) until the mouse button is finally released.

Turtle-dragging mode is an extremely useful feature in combination with turtle-graphics procedures. Suppose, for instance, we have written a procedure such as the one for drawing octagons above; we can now position

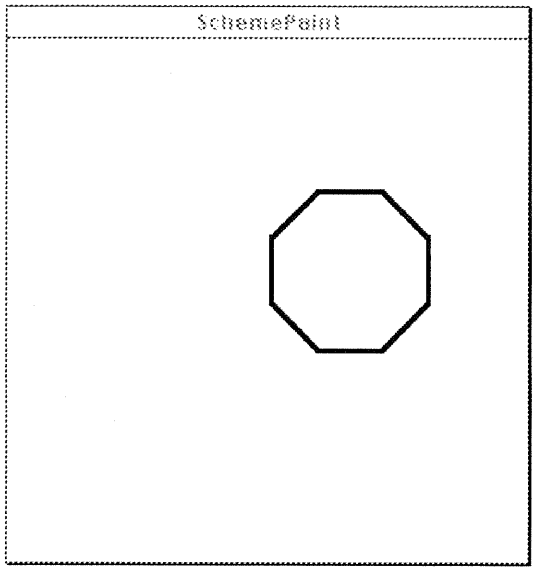



*Figure 5.2*: A turtle-drawn octagon.

Now our standard octagon could be generated as follows:

```
(octagon fd 25)
```

We could also use the new procedure to express a variety of "ocatgon variations." As a start, we could write a procedure that makes a tall, thin `rectangle`:

```
(define (thin-rectangle side)
  (repeat 2 (fd side) (rt 90) (fd (/ side 5)) (rt 90))
  (fd side))
```

We now use the `thin-rectangle` procedure as the `turtle-move` argument in the `octagon` procedure:

```
(repeat 8 (octagon thin-rectangle 25) (rt 45))
```

The result of evaluating the expression above is shown in Figure 5.3.
These examples barely begin to suggest the wealth of marvelous patterns that can be created with turtle procedures. Abelson and diSessa{1} and Papert{17} provide more description of this topic; and some additional illustrations of turtle graphics are introduced in Section 8.

the turtle interactively, by eye (using turtle-dragging mode) and then call the `octagon` procedure to place an octagon in some desired location on the screen.

## *5.2 Planar Maps*

In addition to its turtle-related graphics features, SchemePaint includes a package of procedures and objects for experimenting with planar maps (maps from $\mathbf{R}^2$ to $\mathbf{R}^2$). The graphic effects that can be produced with this sub-language form an interesting complement to those produced via turtle graphics.

A typical scenario for using planar maps in SchemePaint is as follows:

- Create a graphics object (point, line, or polygon);
- Create a map to apply to this object;
- Show (in the canvas window) the result of applying the map (perhaps iteratively) to the given object.

Loosely following this scenario, we will first illustrate how to create graphics objects and maps in SchemePaint, and will then discuss how objects (and the results of applying maps to objects) can be displayed in the canvas window.

### *5.2.1 Creating Graphics Objects and Maps*

SchemePaint includes procedures for constructing point objects (given x- and y-coordinates as arguments), line segment objects (given two point objects as endpoints), and polygon objects (given a list of point objects as vertices). These are summarized in Table 5.4, along with procedures for accessing parts of the various object data structures.

As an illustration of how to construct a SchemePaint graphics object, the following expression creates a point object (corresponding to the origin) and binds the name `origin` to it:

```
(define origin (make-point 0 0))
```

This expression creates a polygon object (corresponding to a right triangle) and binds the name `triangle-1` to it:

```
(define triangle-1
 (make-closed-polygon
   (list origin
         (make-point 80 0)
         (make-point 0 80))))
```

There are numerous ways of creating maps in SchemePaint; the most general-purpose method employs a special form named `make-map`. The following `make-map` expression, as an example, creates a map which, when applied to a point (x, y), returns the point (0.5x + 2, 0.6y):

```
(make-map (x y)
  (+ 2 (* 0.5 x))
  (* 0.6 y))
```

| Procedure | Description |
|---|---|
| (make-point *ptx pty*) | Takes two numeric arguments and returns a point object. |
| (point-xcor *pt*) | Takes a point-object argument and returns its x-coordinate. |
| (point-ycor *pt*) | Takes a point-object argument and returns its y-coordinate. |
| (make-line *pt1 pt2*) | Takes two point-object arguments and returns the line-segment object with these two points as endpoints. |
| (line-endpoint-1 *line*) | Takes a line-segment object and returns its first endpoint. |
| (line-endpoint-2 *line*) | Takes a line-segment object and returns its second endpoint. |
| (make-closed-polygon *ptlist*) | Takes a list of (at least three) point-objects and returns the polygon object with these points as vertices. |
| (polygon-vertices *poly*) | Takes a polygon object as argument and returns its vertices (as a list of point-objects). |

*Table 5.4*: Constructor and selector procedures for graphical objects.

Maps can be affine (like the one above) or nonlinear. The following expression creates the well-known nonlinear "Henon map"{22} and binds the name `henon` to it:[10]

```
(define henon
  (make-map (x y)
    (+ 1 y (* -1.4 x x))
    (* 0.3 x)))
```

It should also be noted that maps may be passed as arguments to Scheme procedures and returned as results of procedure calls (i.e., maps are "first-class objects"{21}). The following sample procedure implements a parametrized version of the Henon map above: it takes two numeric arguments and returns the appropriately parametrized map:

```
(define (henon-maker param1 param2)
  (make-map (x y)
   (+ 1 y (* param1 x x))
   (* param2 x)))
```

[10] In typical algebraic notation, this map would be written:

$$x' \leftarrow 1 + y - 1.4x^2$$
$$y' \leftarrow 0.3x$$

Thus, the following expression creates a map identical to our already-created Henon map:

```
(define henon
  (henon-maker -1.4 0.3))
```

SchemePaint maps may be composed together to yield some new map which, when applied to an object, will simply apply each of the original component maps in sequence. Another method of map composition is "superposition" (or applying several component maps independently to some given object, thus yielding a set of objects); this will be illustrated a bit later. There are also a

| Procedure | Description |
|---|---|
| (make-map (*var1 var2*) *next-var1-expression next-var2-expression*) | A special form. Followed by a list of two variable names *var1* and *var2* corresponding to x- and y-coordinates (typically the names are just x and y). The next two expressions indicate how to compute the following x- and y-values from the current ones. |
| (make-scale-map *scale*) | Takes a numeric argument and returns a map which, when applied to some point, uniformly scales both coordinates by the given value. |
| (make-translate-map *xtrans ytrans*) | Takes two numeric arguments and returns a map which, when applied to some point, translates the x- and y-coordinates by the specified values. |
| (make-rotate-map *theta*) | Takes a numeric (radian) argument and returns a map which, when applied to some point, rotates the point by *theta* radians about the origin. |
| (make-affine-map *m11 m12 m21 m22 off1 off2*) | Takes six numeric arguments and returns the affine map corresponding to the 2x2 matrix specified by the first four values followed by the x- and y-offsets specified by the last two values. |
| (compose-maps *m1 m2 ... mn*) | Takes two or more maps and returns the map which applies in sequence *mn,..., m2, m1*. (That is, the final argument map *mn* is the first to be applied in the composed map.) |
| (superpose-maps *m1 m2 ... mn*) | Takes two or more maps and returns the map which, when applied to a given point, returns a list of all the points returned by each of the individual maps. |
| (iterated-map *map n*) | Takes a map and a positive integer argument and returns the *n*-fold iterated map. (That is, the given map is composed with itself *n* times.) |
| (make-next-point-map *diffeq-map dt*) | Takes a "differential equation map" of the form (x', y') = (dx/dt, dy/dt), and a *dt* value, and returns a map which, when applied to a given point (x, y) will integrate the differential equation (using a fourth-order Runge-Kutta integrator) to obtain the point (x+dx, y+dy). |

*Table 5.5*: Map constructors.

variety of "simple" map constructors to create commonly-used types of affine maps: for instance, the following expression uses the procedure `make-rotate-map` to construct a map which, when applied to a point, will rotate that point $\pi$ radians (180 degrees) about the origin:

```
(make-rotate-map 3.14159)
```

Some of the more important SchemePaint map constructors are summarized in Table 5.5.

### *5.2.2 Applying Maps to Objects; Displaying Objects*

The previous paragraphs illustrated how both graphics objects and maps may be constructed in SchemePaint. Thus far, however, we have said nothing about how objects may be displayed on the screen; and we have not seen how to apply maps to graphics objects. The SchemePaint procedure `show` is used for the first purpose: this procedure takes a graphics object (or set of objects) and displays the object(s) on the screen. Thus, if we evaluate the following expression:

```
(show triangle-1)
```

we will see displayed on the screen the polygon object that we created earlier (see Figure 5.6).[11]

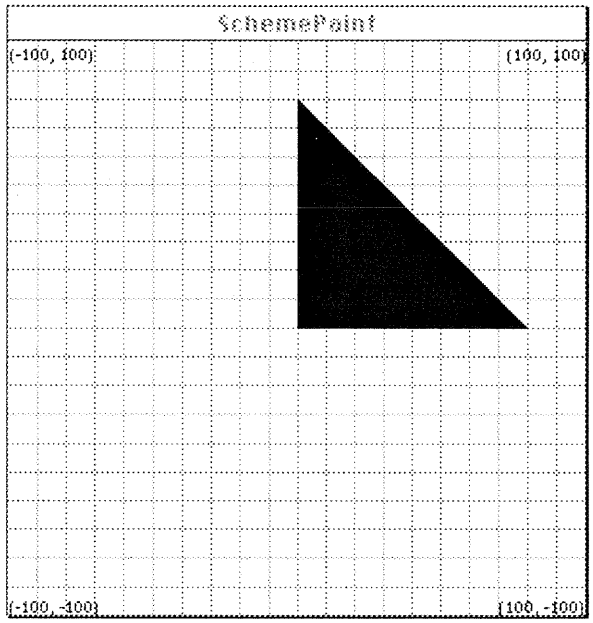



*Figure 5.6*: Displaying a polygon object.

[11] Note that the SchemePaint screen corresponds to the rectangular region of the plane bounded by the point (-100, -100) at bottom left and (100, 100) at upper right. These coordinates may be changed by the user via menu commands or SchemePaint expressions.

The most general SchemePaint procedure for applying maps to objects is named `apply-map`; this procedure takes two arguments—a map and an object (or set of objects)—and applies the given map to the object(s). The result returned is itself a new graphical object which may be displayed on the screen (via the `show` procedure) or to which we may apply still other maps.

As an example of how maps may be applied to objects, consider the following expression; this applies the rotate-180-degrees map that we created earlier to the polygon object `triangle-1` (and displays the result on the screen):

```
(show (apply-map (make-rotate-map 3.14159) triangle-1))
```

The resulting picture is shown in Figure 5.7 (compare the original triangle of Figure 5.6).

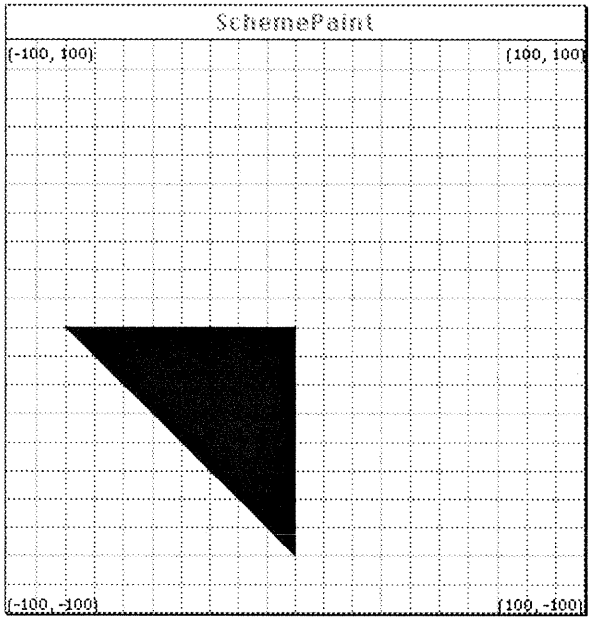



*Figure 5.7*: Applying a rotation map to our triangle object.

Maps may be applied iteratively to a given starting object. The SchemePaint primitive `show-n-iterations` is used for this purpose; this procedure takes three arguments (a graphics object, a map, and a number *n* of iterations), and shows the result of applying the map 0, 1, 2,...*n* times to the given object.[12] As an example, we can take our already-created `henon` map and apply it several thousand times to a particular starting point, showing the result of each successive application:

```
(show-n-iterations (make-point 1 1) henon 3000)
```

---

[12]It would not in fact be all that difficult to write this procedure ourselves, using rudimentary knowledge of Scheme syntax and the existing primitives `show` and `apply-map`; but because applying maps iteratively is such a common task, the `show-n-iterations` procedure is included as a primitive in its own right.

The resulting figure is seen in Figure 5.8.

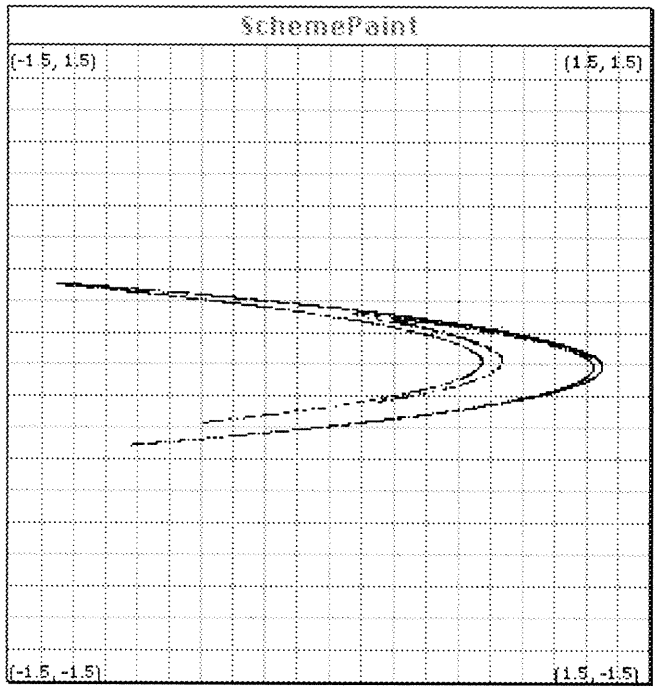



*Figure 5.8*: The "Henon attractor", produced by iterating a nonlinear map. (Cf. {22})

### *5.2.3 A Brief Example: Creating a Fractal*

Just to demonstrate how the various elements of SchemePaint's dynamical systems sub-language may be combined within an overall project, this subsection illustrates how we can create a well-known fractal. We begin by creating a polygon—in this case, the same right triangle (`triangle-1`) that we have used for previous examples. We now create three simple affine maps:

```
(define map1
  (make-scale-map 0.5))

(define map2
  (compose-maps (make-translate-map 40 0)
                map1))

(define map3
  (compose-maps (make-translate-map 0 40)
                map1))
```

In Figure 5.9 we have shown (in gray) the original triangle object and (in black) three smaller triangles that result from applying each of our three new maps to that triangle. (The smaller triangles, displayed afterward, block most of the gray triangle from our view.)

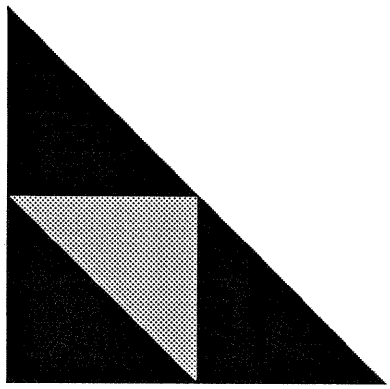


*Figure 5.9*: The original triangle (in gray), mostly blocked by three smaller triangles (in black).

We can now create a composite "superposition map" that is the combination of all three of our individual maps:

```
(define sierpinski (superpose-maps map1 map2 map3))
```

If we were to apply our new superposed map to the original triangle, as follows:

```
(show (apply-map sierpinski triangle-1))
```

SchemePaint would display the same three smaller triangles of Figure 5.9.

Having come this far, we can now apply our new `sierpinski` map iteratively. Figure 5.10 shows the result of evaluating the following expression:

```
(show (apply-map sierpinski (apply-map sierpinski triangle-1)))
```

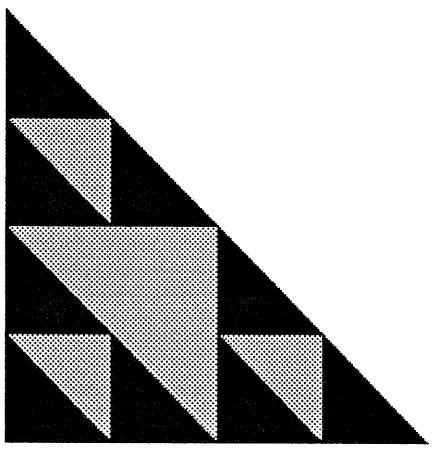


*Figure 5.10*: Applying the `sierpinski` map twice in succession. Nine smaller (black) triangles again block much of the original (gray) triangle.

In Figure 5.10, each of the three triangles from Figure 5.9 has had a (three-fold) superposed map applied to it, and the result is a set of nine triangles.

We could continue this process by applying the `sierpinski` map five successive times; a relatively straightforward method for achieving this uses the map constructor `iterated-map`, as follows:

```
(show (apply-map (iterated-map sierpinski 5) triangle-1))
```

The result, depicted in Figure 5.11, is an approximation to the "Sierpinski triangle", a well-known fractal set.{4}

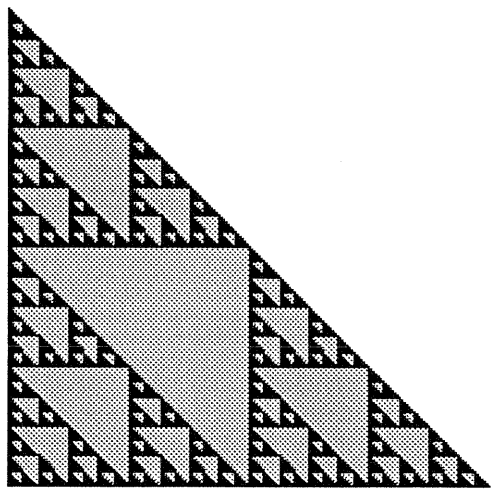


*Figure 5.11*: Iterating the `sierpinski` map five times. (The original triangle is again shown in gray behind the 243 smaller black triangles.)

### *5.2.4 The **Planar Maps** Menu*

Most of SchemePaint's map-related functionality is based in its language component—in the procedures and objects with which the core Scheme language has been enhanced. There are, however, a few interface features (realized as selections within the **Planar Maps** menu) that facilitate novice-level experimentation with maps.

The selections presented on the **Planar Maps** menu are summarized in Table 5.12. These can essentially be read, in order, as the component operations of certain simple types of experiments. That is, we first create a map which will act as our "default map" (using the *New Default Map* selection); we then choose how many iterations of that map we want (using the *Number of Iterations* selection); we create an initial object (using *Make an Object*); optionally, we specify how the color of the object will depend on iteration count (using *Color Procedure*); and finally, we display the operation of the map on the given starting object for n successive iterations. With the exception of the color-procedure setting, these individual steps are basically the same choices

| Menu Selection | Notes |
|---|---|
| *New Default Map* | Presents dialog boxes which allow the user to type in expressions for computing next-x and next-y (or, in polar coordinates, next-r and next-theta) values. |
| *Number of Iterations* | Prompts the user to specify how many times the default map should be applied (and if some initial number of iterations should not be displayed). |
| *Make an Object* | Asks the user to create (via mouse or by typing in coordinates) a point, line, rectangle, polygon, or circle on the screen. (This will be the object to which the new given map should be applied.) |
| *Color Procedure* | Allows the user to create a procedure which determines (based, among other things, on iteration count) the color of objects displayed when the map is applied. |
| *Do the Map* | When a map, iteration count, starting object, and (optionally) color procedure are known, this selection applies the map for the specified number of times and displays the mapped objects (at appropriate iteration counts and in the appropriate colors). |

*Table 5.12* **Planar Maps** menu contents.

reflected in the arguments to the `show-n-iterations` procedure mentioned earlier.[13]

Because there is little conceptually new in the **Planar Maps** menu, we will not present a detailed discussion here. One might consider this menu a kind of "training wheel" for helping the user find out the kind of patterns one can create with planar maps. There is one feature of the menu, however, which has more general utility and will be referred to a bit later on in this paper: namely, the *Make an Object* selection. Using this selection, the user can create a point, line, or polygon object (as well as certain types of polygons such as rectangles and circles) directly on the screen, via mouse operations. In this sense, the *Make an Object* choice is similar in function to the typical polygon-creating modes supplied with virtually all interactive paint programs.

[13]Actually, yet another SchemePaint primitive, `experiment-with-iterations`, is an "enhanced" version of `show-n-iterations` which includes a color procedure parameter.

### *5.3 Color-Related Procedures*

The bulk of SchemePaint's package of color-related procedures and objects was introduced earlier in this paper, in Section 2.3.2; to summarize that discussion, color objects are represented as lists of three floating point numbers in the range 0.—1. (corresponding to the R, G, and B components of the desired color) and there are a variety of primitive procedures that create, access, test, combine, and use these color objects. SchemePaint includes a "starter's kit" of sixteen named color objects (including `red`, `blue`, `green`, `black`, and `white`), corresponding to those visible in the palette window; and of course the user can create as many new color objects as desired.

There are several color-related features in SchemePaint not mentioned in the earlier discussion. As an example of one such feature, we present the color-combining primitive `interpolate-between-colors`. This procedure takes three arguments *color1*, *color2*, and *n* (two color objects and a number in the range 0-1), and returns a new color object somewhere on a spectrum "between" *color1* and *color2* (with *n* = 0 corresponding to *color1* and *n* = 1 corresponding to *color2*). Thus, the color object returned by the following expression corresponds to a bluish color with just a hint of purple:[14]

```
(interpolate-between-colors red blue 0.9)
```

This color corresponds to a deep purple:

```
(interpolate-between-colors red blue 0.5)
```

Even though `interpolate-between-colors` is supplied as a primitive procedure, it would not be difficult to write; the enterprising reader familiar with Scheme might try this as an exercise, using the earlier example of `average-between-colors` as a hint of how to proceed.

Another feature worth mentioning is SchemePaint's use of "custom colors." The palette window shown in Figure 4.1 includes two "empty circles" in its bottom row; these two circles can be filled with any color of the user's choosing, after which the newly-added color can be accessed like any other in the palette window. The procedure that adds (or changes) an existing custom color is `set-custom-color!` (which takes two arguments: the first is the integer 1 or 2—depending on which of the two circles we want to access—and the second is the new color object). Thus, if we wished to add a yellow color to the first empty circle of the palette window, we could evaluate the following SchemePaint expression:

```
(set-custom-color! 1 (make-color-object 1. 1. 0.))
```

---

[14] In this expression we are using the color objects named `red` and `blue` defined in the earlier section.

### *5.4 Miscellaneous Features*

Besides the language features already described—features which, as noted, may be grouped into several "sub-languages"—there are a few additional handy SchemePaint procedures that deserve mention here. Two file-related procedures (`save-schemepaint-canvas` and `restore-schemepaint-canvas`) save the contents of the canvas window to a file and restore the canvas contents from a file, respectively. There are also several text-related procedures: one, `draw-string-at-point`, takes two arguments (a string and a point object). When called on these arguments, the procedure causes the string to be displayed starting at the given point on the canvas window. Another procedure, `setup-canvas-text-size-and-font`, takes two arguments (a size and font-number); when called on these arguments the procedure changes the current font and size for future text displayed in the canvas window. As an example, we might evaluate the following expression:

```
(setup-canvas-text-size-and-font 18 1)
```

Having evaluated this expression, subsequent text written either via direct manipulation (using the text mode palette choice mentioned earlier) or via SchemePaint expressions (e.g., calls to `draw-string-at-point`) will appear in 18-point Geneva font.

## 6. Features for Interface/Language Cooperation

In designing programmable applications we seek to go beyond a mere passive combination of direct manipulation interfaces and programming environments. Rather—as discussed in Section 2.3.3—we would like to find ways in which these two portions of the application can play off one another in new and creative ways, thereby enhancing the expressiveness of the overall system. In this section, we examine several features of SchemePaint that represent experiments in this direction—features in which the "language half" and the "interface half" of the program join forces to produce novel types of functionality for the user.

### *6.1 Programmable colors and mouse-clicks*

Two simple (but powerful) techniques for interface/language cooperation involve the as-yet-undescribed "proc" and "click" boxes of the palette window in Figure 4.1. These boxes are the interface elements that access SchemePaint's mechanisms for "programmable colors" and "programmable mouse-clicks." The following two sub-sections explain these features in more detail.

#### *6.1.1 Programmable colors*

The purpose of SchemePaint's "programmable color" feature is to allow the mouse to draw lines in procedure-determined colors. The typical scenario for using this feature is as follows:

- The user writes a new SchemePaint procedure which takes two arguments (corresponding to the x- and y-coordinates of a point on the canvas window) and returns a color object.

- The user now calls the SchemePaint primitive `set-mouse-function!` with the new procedure as argument.

- Having completed these steps, the user selects "proc" mode on the palette window. The mouse will now draw lines on the canvas window in colors specified by the given procedure.

As an example of this scenario, suppose that the user would like the mouse to draw lines whose color is red if the x-coordinate of the mouse point is less than zero, and green if the x-coordinate is greater than or equal to zero. The following procedure expresses this idea:

```
(define (negative-x-red-positive-x-green x y)
  (if (< x 0)
      red
      green))
```

Now, the user calls the SchemePaint primitive `set-mouse-function!` with this new procedure as argument:

```
(set-mouse-function! negative-x-red-positive-x-green)
```

Finally, by selecting *proc* (for "mouse-procedure") mode in the palette window, the user can now draw lines with the mouse which will obey the given procedure: that is, if the mouse is dragged along in the canvas window from left to right past the vertical line x = 0, the color drawn by the mouse will change from red to green, as shown in Figure 6.1.[15]

A little imagination at this juncture will suggest the expressive range of the programmable color mechanism. Suppose, for instance, we would like to have the mouse draw in a shade that varies smoothly from red at the left of the canvas window (for an x-coordinate of -100) to blue at the right of the window (for an x-coordinate of 100). The following procedure expresses this idea:

```
(define (red-to-blue-horizontally x y)
  (interpolate-between-colors
   red blue (/ (abs (+ x 100)) 200)))
```

Again we can call the `set-mouse-function!` primitive:

```
(set-mouse-function! red-to-blue-horizontally)
```

Having done this (and again selecting *proc* mode) we have captured the functionality that is usually associated with the notion of "color gradients" in conventional graphics applications. Note, however, that our programmable color mechanism provides a much richer range of features than color gradients alone. We can create non-linear gradients; gradients with randomized elements; "two-dimensional" gradients (e.g., colors whose red

---

[15] Note, in this figure, that the default canvas coordinates range from (-100, -100) at the bottom left corner to (100, 100) at the upper right coordinates.

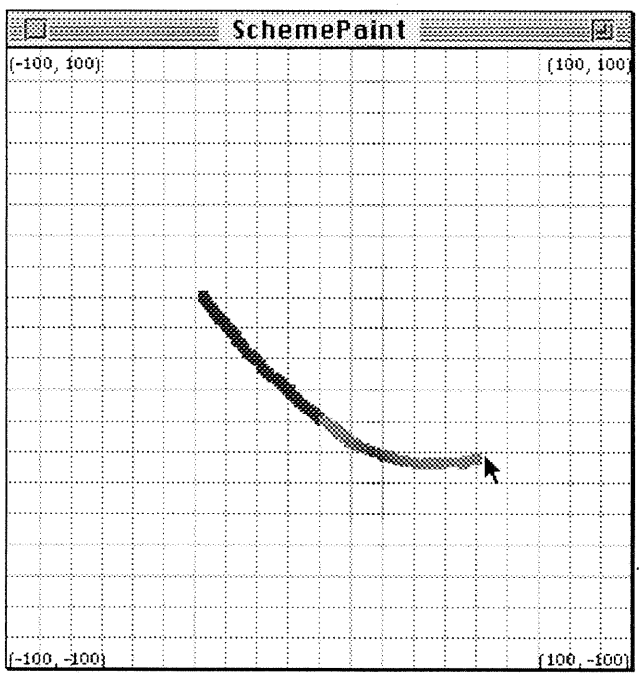



*Figure 6.1*: An example of programmable colors in action. The pen color changes from red to green as the mouse is dragged past the vertical line x = 0 while in "proc" (or "mouse-procedure") mode.

component varies horizontally and whose blue component varies vertically). Indeed, gradients are only a tiny fraction of what we can express: we could have the mouse draw in stripes or plaids; or have the mouse draw colors according to the values of an electrical potential in the plane (e.g., a high potential corresponds to red, a low potential to blue); or have the mouse draw colors that vary sinusoidally. Because we can express colors through procedures—procedures that we, as users, can write for ourselves—we can give our creative powers free rein in a way that would be unthinkable in a "pure" direct manipulation setting.

### *6.1.2 Programmable mouse-clicks*

The programmable mouse-click mechanism in SchemePaint is similar in operation to the programmable color mechanism. Here, the idea is that we can write a procedure that will determine an action that is taken every time the mouse button is pressed and released in the canvas window. The typical scenario is as follows:

- The user writes a procedure which takes two arguments (again corresponding to the x- and y-coordinates of the current mouse point) and which performs a series of actions when called on those arguments.

- Having done this, the user calls the SchemePaint primitive `set-mouse-click!` with the new procedure as argument.

- Now, by selecting click mode in the palette window, the mouse behavior is set so that every time the mouse button is pressed and released in the canvas window, the given procedure is called on the current mouse x- and y-coordinates as arguments.

As an example, suppose we want the mouse behavior to be as follows: every time the button is pressed, the turtle will move to the current mouse location and draw an octagon (of side-length 10) at that spot. As a first step, we write a new procedure:

```
(define (draw-octagon-here x y)
  (turtle-penup)
  (setpos x y)
  (turtle-pendown)
  (octagon 10))
```

We now call the SchemePaint primitive `set-mouse-click!` with our new procedure as argument:

```
(set-mouse-click! draw-octagon-here)
```

Finally, we select click mode in the palette window; and henceforth, every mouse-click in the canvas window will cause an octagon to be drawn at the current mouse location.

Just as with programmable colors, the range of expression provided by even this relatively simple programmable mouse-click mechanism is extremely wide. We could, for instance, simply reset the current color in a fashion specified by the current mouse position:

```
(define (set-color-according-to-point x y)
  (set-foreground-color!
   (make-color-object (/ (abs x) 100)
                      (/ (abs y) 100)
                      1.)))

(set-mouse-click! set-color-according-to-point)
```

Or we could write a procedure that simply points the turtle toward the current mouse point:

```
(define (point-turtle-here x y)
  (seth (toward (make-point x y))))

(set-mouse-click! point-turtle-here)
```

### *6.2 General Constructs for Programmable Mouse Behavior*

The two features described in the previous section are actually special cases of more advanced constructs in SchemePaint. In particular, SchemePaint includes procedures named `set-mouse-procedure-1!` and `set-mouse-procedure-2!` which can be used to denote an even wider range of mouse-dragging and

mouse-clicking behavior. Here, we will describe the operation of the former procedure; the behavior of `set-mouse-procedure-2!` is analogous.

The SchemePaint primitive `set-mouse-procedure-1!` takes three arguments, each of which is itself a procedure. The three arguments—*buttondownproc*, *buttonstilldownproc*, and *buttonuppproc*—dictate (respectively) mouse behavior when the button is first depressed in the canvas window in *proc* mode; mouse behavior as the mouse is dragged in the canvas window; and mouse behavior when the button is finally released. The form of each of the three argument procedures is as follows:

- *Buttondownproc* should be a procedure of four arguments: *mousex*, *mousey*, *this-point*, and *this-color*. These refer to the x- and y-coordinates (in window—i.e., pixel—coordinates) of the mouse when the button is first pressed; the point (in SchemePaint coordinates) where the button has been pressed; and the color of the pixel at which the mouse is positioned.

- *Buttonstilldownproc* is a procedure of 10 arguments, called repeatedly as the mouse is dragged along. Its arguments are: *mousex*, *mousey*, *this-point*, *this-color*, *previous-mousex*, *previous-mousey*, *previous-point*, *previous-color*, *counter*, and *down-time*. The first four of these are analogous to the four arguments for *buttondownproc*; the next four denote similar values for the previous recorded mouse point during the current dragging operation; the ninth argument simply counts how many times *buttonstilldownproc* has been applied; and the final argument refers to the amount of time (in 0.01-second units) since the button was first pressed in this dragging operation.

- *Buttonuppproc* is a procedure of eight arguments, called when the mouse button is finally released (i.e., when the dragging operation is concluded). The eight arguments for this procedure are *mousex*, *mousey*, *this-point*, *this-color*, *previous-mousex*, *previous-mousey*, *previous-point*, and *previous-color*. The meaning of these arguments is identical to that of the first eight arguments to *buttonstilldownproc*.

As an example of how this more general construct may be used, imagine that we would like the behavior of the mouse to be as follows: when the button is first pressed (in procedure mode) the color is blue, but over the course of twenty seconds the color should change smoothly to red. Finally, when the mouse button is released, the color should reset back to blue.

To accomplish this, we need to write the three procedures which will eventually be used as arguments for `set-mouse-procedure-1!`. The first (the eventual value of the *buttondownproc* argument) should simply set the current foreground color to blue:

```
(define (start-at-blue mousex mousey this-point this-color)
  (set-pen-color! blue))
```

The dragging procedure (the eventual value of *buttonstilldownproc*) should change the color toward red over the course of twenty seconds, making sure to display the current point:

```
(define (toward-red-over-20-seconds
         mousex mousey this-point this-color
         previous-mousex previous-mousey
         previous-point previous-color
         counter down-time)
  (set-pen-color!
   (interpolate-between-colors
    blue red (min 1 (/ down-time 2000))))
  (show this-point))
```

The third (button-releasing) procedure should reset the pen color to blue:

```
(define (finish-at-blue mousex mousey this-point this-color
         previous-mousex previous-mousey previous-point
         previous-color)
  (set-pen-color! blue))
```

Finally, we pass the three new procedures as arguments to `set-mouse-procedure-1!` as follows:

```
(set-mouse-procedure-1!
  start-at-blue
  toward-red-over-20-seconds
  finish-at-blue)
```

Now, when the mouse is in *proc* mode, it will (as specified) paint in a color that gradually changes from blue to red over the course of 20 seconds per dragging operation.

### *6.3 Naming Objects Created by Direct Manipulation*

In our earlier discussion of interface/language cooperation (Section 2.3.3), we noted that there were, broadly speaking, two effective "directions" of cooperation: we could design language constructs that specify interface behavior, or interface operations that send information back to a running program. Thus far, the operations that we have examined in this section fall within the former category: we have presented language constructs which can denote rather wide-ranging and sophisticated interface behavior. In this sub-section, we look at SchemePaint features which work in the opposite direction—features which allow interface operations to have linguistic consequences.

The first such feature is associated with the object-creation techniques provided by the Make an Object selection of the Planar Maps menu. When a new object (point, line, rectangle, circle, or general polygon) is created through this menu, it is automatically given a default name (`*last-point-created*`, `*last-line-created*`, `*last-rectangle-created*`, and so forth). Thus, if we choose *Make an Object* from the **Planar Maps** menu and (for instance) choose to create a rectangle using the mouse, then this rectangle will

afterward be the polygon object denoted by the name `*last-rectangle-created*`; and the name will continue to be bound to this object until such time as another rectangle is created via the *Make an Object* choice.

The net result of this feature is that the objects we create via direct manipulation automatically have a name that allows us to access them in a running program. For instance, suppose we have written a procedure that takes as argument a polygon and finds the geometric center of that polygon:[16]

```
(define (find-geometric-center polygon)
  (let ((vertices (polygon-vertices polygon)))
   (make-point
    (average-of-list (map point-xcor vertices))
    (average-of-list (map point-ycor vertices)))))
```

If we have at some time created a rectangle using the *Make an Object* selection, then we can evaluate the following expression:

```
(show (find-geometric-center *last-rectangle-created*))
```

and the center point of the previously-created rectangle will be displayed on the screen. With little additional effort we could (e.g.) show the rectangle itself with lines from each vertex to the geometric center; or we could display another polygon translated so that its center point is now at the geometric center of the rectangle; or we could rotate the rectangle about its geometric center and display the result.

The second SchemePaint feature to be mentioned here is similar in spirit to the first; this feature allows graphics objects to be "read" by Scheme expressions. Specifically, SchemePaint includes primitive procedures named `read-a-point`, `read-a-line`, `read-a-rectangle`, and so forth; when one of these procedures is called (on no arguments), the user is prompted to create a graphics object of the appropriate sort in the canvas window; and this object then becomes the returned value of the procedure call. Suppose, for instance, we would like to find the distance between two points on the screen. The following procedure will work for this purpose:

```
(define (find-distance-between-screen-points)
  (let* ((pt1 (read-a-point))
         (pt2 (read-a-point)))
   (point-distance pt1 pt2)))
```

When this procedure is called on no arguments, the user will be prompted to enter two particular points on the canvas window; the returned result of the procedure call will be the distance (in SchemePaint coordinates) between these two points.

---

[16] Technically, this procedure computes the center of mass of equal masses placed at the vertices of the polygon.

## 7. Extending SchemePaint via Libraries

In the earlier discussion of the SchemePaint language, we noted that the program's linguistic features could be divided, roughly, into portions dealing with colors, dynamical systems, turtle graphics, and so forth. Indeed, we referred to these portions as "sub-languages" embedded in Scheme—collections of interrelated procedures, object types and (occasionally) special forms designed around some particular topic.

This organization of the SchemePaint language suggests a natural strategy for expansion of the system. If we would like to extend SchemePaint to draw (say) flowcharts, or geometric diagrams, or tiling designs, we need only create files that constitute a new sublanguage, or *library*, that can be added in to the set already present. In other words, we create a "flowchart-design library"—some new collection of object types and procedures for creating flowcharts—and we load this library into SchemePaint. Once this is done, our programming environment can be regarded as a "core" Scheme system enhanced by the union of all the procedures and object types of the various sublanguages.

There are many software design issues raised by this rather oversimplified scenario; some of these will be mentioned toward the end of this section. Before discussing these more general matters, however, we will first examine two sample libraries that have been implemented for SchemePaint.

### *7.1 Escher (Tiling) Library*

The first sample library to be discussed is a "tiling library" based on a program first developed by Henderson{12} and later implemented in Scheme by Abelson and Sussman{3}. This library is particularly useful for drawing recursive and repetitive designs of the type made popular by the late Dutch artist M.C.Escher.

Our tiling library may be loaded into SchemePaint by evaluating the following expression:

```
(load-schemepaint-library "escher")
```

Once this is done, a collection of new procedures and object types is added to our language; in addition, a new (fourth) window appears on the SchemePaint screen. This window, labelled **Escher**, is shown in Figure 7.1; its use will be described a bit later on in this section.

#### *7.1.1 The Tiling Language*

The basic object type of our new tiling language is the *picture* type. A picture is an object associated with a collection of lines and polygons drawn within the unit square (i.e., the square of side-length 1 whose bottom left-hand corner is the origin). We can think of these graphic objects as the "canonical picture"—the picture as it would appear if it were drawn within the unit square. In fact,

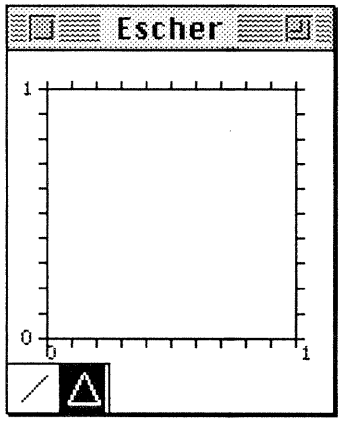



*Figure 7.1*: The **Escher** window accompanying the tiling library.

however, a picture may be drawn within any rectangular region on the screen; and when this is done, the "canonical picture" is scaled to the dimensions of the particular rectangle in which the drawing occurs. Figure 7.2 illustrates the idea: here, the very same picture object has been drawn in three rectangles of different dimensions. In the first rectangle (the unit square), we see the canonical picture; in the second, the picture has been stretched into a short, fat version; in the third, the picture has been stretched lengthwise into a tall, thin version.

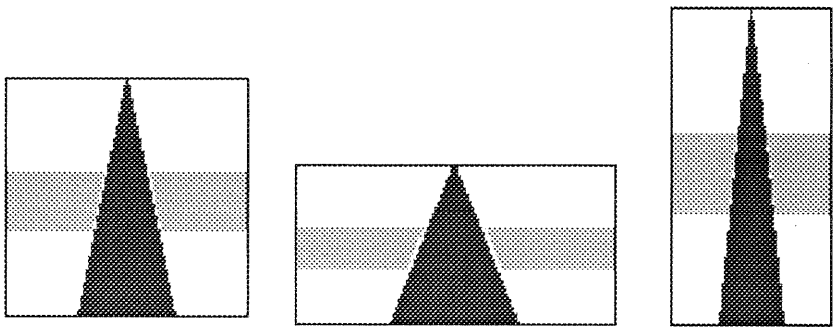


*Figure 7.2*: The same picture-object, displayed within three different rectangles. (Here, the boundaries of the rectangles are shown; the actual picture-object itself consists of the triangle and "stripe.")

By drawing pictures within rectangles at various sizes and orientations, we can generate a wide range of patterns. To start with a simple example: by drawing a picture within a rotated rectangle, we obtain a rotated version of the Figure 7.2 picture, as shown in Figure 7.3.

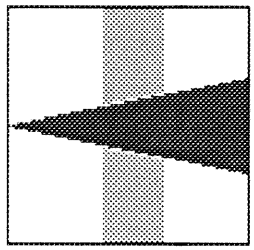


*Figure 7.3*: The same picture object as in Figure 7.2, now displayed within a rectangle that has been "turned on its side."

We can also combine pictures together by drawing them within smaller rectangular portions of a larger rectangle. Figure 7.4 depicts the result of drawing two pictures side-by-side in the left and right halves of a long rectangle.

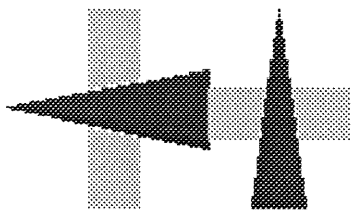


*Figure 7.4*: The original picture and its rotated version have been placed beside one another within a short, wide rectangle.

All of these concepts—creating picture objects, drawing them within rectangles, combining them together—have linguistic representation in our library. Table 7.5 summarizes a few of the more important SchemePaint procedures; the crucial point, however, is not so much the specific choices of procedures and object types, but the overall power provided by having a programming language at all. In particular, we are able to build complexity by giving names to procedures that create compound pictures from simpler ones. For instance, the following procedure takes three pictures as arguments and (using the `beside` procedure from Table 7.5) creates a new picture which, when drawn within some rectangle, will draw each of the three component pictures in one third of the rectangle:

```
(define (make-three-fold-picture pict1 pict2 pict3)
  (beside pict1
          (beside pict2 pict3 0.5)
          0.333))
```

We could now make a three-fold version of the picture shown in Figure 7.2:

```
(define three-fold-pict
  (make-three-fold-picture
    figure-7-2-pict figure-7-2-pict figure-7-2-pict))
```

| Procedure | Description |
|---|---|
| `make-primitive-picture` *lines-and-polygons* | Takes as argument a list of line and polygon objects and creates the primitive picture object which would draw these lines and polygons within the unit square. |
| `together` *pict1 pict2* | Takes two picture objects as arguments and returns a new picture object that, given a rectangle, draws both *pict1* and *pict2*. |
| `beside` *pict1 pict2 ratio* | Takes two picture objects and a ratio and returns a new picture object that, given a rectangle, draws *pict1* in the left portion of the rectangle and *pict2* in the right portion. The division between left and right portions is made so that the left portion has (ratio * 100) percent of the overall rectangle. |
| `above` *pict1 pict2 ratio* | Takes two picture objects and a ratio and returns a new picture object that, given a rectangle, draws *pict1* in the upper portion and *pict2* in the lower portion. The division between upper and lower portions is made so that the upper portion has (ratio * 100) percent of the overall rectangle. |
| `flip` *pict* | Takes a picture object and returns a new picture object which, given a rectangle, draws the "flipped" version (a horizontal mirror image) of *pict*. |
| `rotate90` *pict* | Takes a picture object and returns a new picture object which, when given a rectangle, will draw *pict* rotated by 90 degrees. |

*Table 7.5*: Procedures for creating and combining picture objects in the tiling library.

The result of drawing `three-fold-pict` in a given rectangle is shown in Figure 7.6. This is all well and good, but again the more interesting point is that `three-fold-pict` is now *itself* a picture object that can be combined with others in still larger compound forms. We could, for instance, create a nine-fold picture as follows:

```
(make-three-fold-picture
 (rotate90 three-fold-pict)
 three-fold-pict
 (rotate270 three-fold-pict))
```

By building complexity in this way—by combining compound pictures into "second-level" compound pictures, and combining those into "third-level" compound pictures, and so on—we can quickly create the ingredients for a rich

catalog of intricate designs. Figure 7.7 and Color Plate 9 illustrate a very tiny subset of the easily available possibilities.

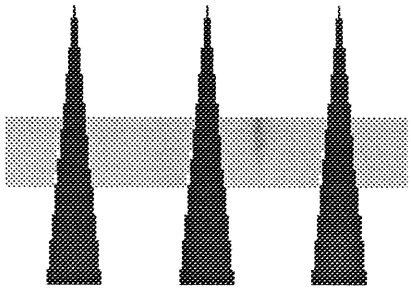


*Figure 7.6*: A three-fold copy of our original picture (from Figure 7.2). This is now a new, compound picture object.

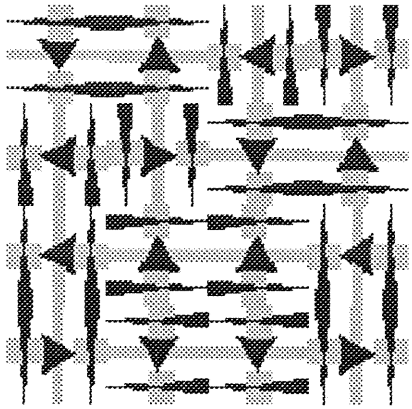


*Figure 7.7*: A design made by using the original picture of Figure 7.2, repeated with different scales and rotation values.

### *7.1.2 The **Escher** Window*

From the user's standpoint, probably the most time-consuming and difficult aspect of experimenting with SchemePaint's tiling language is in the creation of "primitive pictures" via the make-primitive-picture procedure. In order to create a primitive picture this way, the user has to specify each of the component line and polygon objects that go to make up the canonical picture (i.e., the picture as it would appear within the unit square).

The Escher window, shown earlier in Figure 7.1, permits the user to create primitive pictures via direct manipulation. This window depicts (by default) the unit square of the Cartesian plane; by using the mouse to draw lines and polygons within this square, the user can essentially construct by hand the various shapes that can later be incorporated into primitive picture objects.

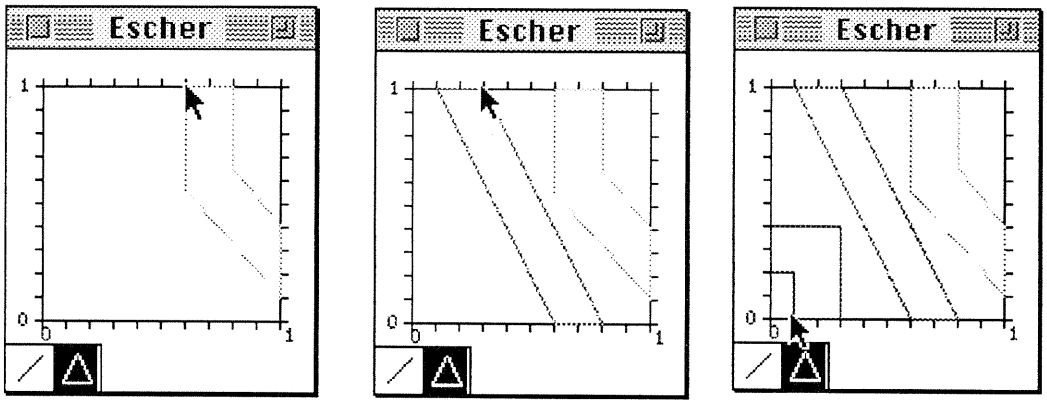



*Figure 7.8:* Creating a primitive picture. The figure shows three "snapshots" as three polygons are created, all constituents of a single primitive picture.

Figure 7.8 depicts several stages in a typical picture-drawing scenario using the Escher window. Here, the user has selected the "polygon" icon at the bottom of the window, indicating that she wishes to draw polygons (as opposed to lines). The user now creates several polygons in different colors [17] and decides to make a primitive picture object for which these polygons constitute the canonical picture. To perform this final step, the user evaluates the following expression:

```
(define my-primitive-picture
 (snap-escher-picture))
```

The `snap-escher-picture` procedure is the central SchemePaint mechanism for translating Escher window constructions into primitive picture objects: when called on no arguments, the procedure returns a new primitive picture whose component canonical objects are those currently appearing within the Escher window. Figure 7.9 depicts a design created by combining the new primitive picture with itself as part of a larger, compound picture.

Before leaving the subject of the tiling library, it is worth mentioning two themes that will reappear in the discussion at the end of this section. First, our library does not merely consist of new linguistic elements (procedures and data objects), as suggested by the earlier overly telegraphic description of the library concept. Rather, our tiling library includes both linguistic and interface elements: we have primitive picture objects and procedural methods for combining them, but we also have a new window that allows us to create certain types of pictures more easily, via direct manipulation. Thus, programmable application libraries—like their "parent" applications—can again be designed to incorporate the best features of both programming environments and direct manipulation interfaces.

---

[17] The operative color in the Escher window is simply the current SchemePaint foreground color, and may be altered via selection within the palette window or by using the `set-foreground-color!` primitive.

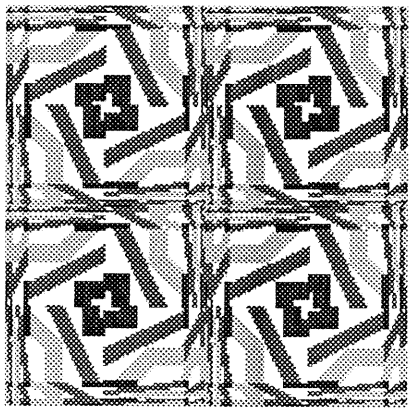


*Figure 7.9*: A compound picture created with multiple copies of the newly-created primitive picture.

The second point worth noting is that the "line and polygon objects" used to create primitive pictures are the same line and polygon objects used in the planar maps package. Thus, our tiling library is designed with an eye toward cooperation (or—viewing the matter more pessimistically—interference) with other sublanguages within the application.

## *7.2 A Library of "Specialty Fill" Procedures*

The second sample library to be discussed is somewhat less elaborate than the Escher library, and is at a comparatively earlier stage of implementation. Nevertheless, the features provided by even this embryonic library are quite powerful and useful; and the artistic effects made possible by these features will perhaps seem less exotic than those suggested by the Escher library.

This second library is devoted to performing "specialty fills" of various types; in essence, its purpose is to expand the semantics of "filling" as provided by most graphics applications. The library is initialized by evaluating the expression:

```
(load-schemepaint-library "fills")
```

Unlike the Escher library, this second library makes minimal additions to the basic SchemePaint interface: one new menu choice is added to the **Paint** menu, and the behavior of the "fill mode" selector in the **Pen** window is slightly altered. Specifically, the new **Paint** menu choice is labelled *Toggle Specialty Fill Mode*; and when "specialty fill mode" is in effect, the meaning of a fill operation is dictated by several new procedures to be described below. (As might be expected, when "specialty fill mode" is not in operation the behavior of a fill operation is unchanged from that provided by the standard SchemePaint interface.)

The abstract notion of "filling" supported by our library is indicated by the following skeletal algorithm:

*Step 1*. Given some starting pixel $p$, find a set of "pixels-to-fill."
*Step 2*. For each of the pixels in the set found in Step 1, perform a "fill action" on that pixel.

Expressed in this form, the "normal" filling operation could be written:

*Step 1*: Find the set of pixels of the same color as $p$, and connected to $p$ by an unbroken path of pixels.
*Step 2*: Re-color each of these pixels to some new specified color.

In this sense, our abstract notion of filling is capable of recapturing the usual operation. However, it is of course capable of expressing much more. We might, for instance, wish to re-color all pixels connected to some starting pixel up to some specified boundary color:

*Step 1*: Find the set of pixels connected to $p$ by a path that does not include a pixel of color *boundary-color*.
*Step 2*: Re-color each of these pixels to some new specified color.

The idea here is that we might have a multicolored region surrounded by (say) a black boundary; and we would like our fill operation to re-color all the pixels within this boundary (rather than just those of some one specified color).

Yet another possible fill operation could be written as follows:

*Step 1*: Find the set of pixels of a given color $c$ within some radius (expressed in pixels) of the original pixel $p$.
*Step 2*: For each of these pixels, "flip a coin" and re-color the pixel to some new color $c'$ if the choice is heads. (That is, for each pixel in the set returned by step 1, we have a 0.5 probability of changing that pixel's color.)

Our new library uses two global procedures to specify the "variable slots" of the abstract algorithm given above. One procedure, `*find-fill-pixels*`, takes a single argument (a starting pixel), and should return a set of all pixels on which the "fill action" will be performed; a second procedure, `*fill-action*` takes two arguments (the original starting pixel and the "current pixel") and performs some specified action on the current pixel. When a given pixel is now selected via mouse in "specialty fill mode," SchemePaint first finds all the pixels dictated by the `*find-fill-pixels*` procedure, and performs the `*fill-action*` operation on each. Thus, we might specify the normal filling operation by evaluating the following expressions:

```
(define (*find-fill-pixels* start-pixel)
  (find-all-connected-pixels-of-same-color start-pixel))

(define (*fill-action* start-pixel current-pixel)
  (set-pixel-color! current-pixel blue))
```

Evaluating these expressions would cause our specialty filling mode to behave like a normal filling operation; in this case, selected regions are filled with the color blue. An important point to note is that we have assumed for this example that procedures such as find-all-connected-pixels-of-same-color have already been written; indeed, a selection of "pre-supplied" procedures of this kind are

included in our library. Some particularly handy "pixel-finder" and "pixel-action" procedures are summarized in Table 7.10 below.[18]

| **Pixel-Finder Procedure** | **Description** |
|---|---|
| `(find-all-connected-pixels-of-same-color` *start-pixel*`)` | Returns the set of all pixels of the same color as the start pixel and connected to it by an unbroken path of pixels. (This is the pixel-finder for "standard" fill operations.) |
| `(find-all-connected-pixels-up-to-boundary-color` *start-pixel boundary-color*`)` | Returns the set of all pixels not of the given boundary color, and connected to the start-pixel by an unbroken path. |
| `(find-pixels-within-radius` *start-pixel radius*`)` | Returns the set of all pixels within a given radius (measured in pixels) of the start pixel. |
| `(find-boundary-pixels-of-same-color` *start-pixel*`)` | Returns the set of pixels of the same color as the start; connected to it by an unbroken path; and adjacent to a pixel of a different color. |
| **Pixel-Action Procedure** | **Description** |
| `(set-pixel-color!` *pixel color*`)` | Sets *pixel* to the specified color. |
| `(set-pixel-color-with-probability!` *pixel color prob*`)` | Sets *pixel* to the specified color with probability *prob* (where *prob* is between 0 and 1); thus, with probability 1 - *prob* the pixel is left unchanged. |
| `(set-pixel-according-to-distance!` *pixel start-pixel distance-to-color-function*`)` | Sets *pixel* to a color specified by calling *distance-to-color-function* on the distance (in pixels) between *pixel* and *start-pixel*. |

*Table 7.10*: Procedures for finding pixel-sets and for creating fill-actions in the "specialty fill" library.

[18] Our library makes use of various low-level procedures that construct and access "pixel objects" (expressed in integer coordinates according to the x- and y-distances of the given pixel from the upper left corner of the canvas window). The procedures in Table 7.10 are in turn implemented in terms of fairly general procedure-constructors, so that a wide range of variations may be expressed with comparative brevity.

Just to illustrate how some additional specialty filling operations may be created, we provide the SchemePaint recipe for the latter two fill-algorithms mentioned earlier. The first, which fills all pixels up to some boundary color, could be written as follows (here, the boundary is black, and the fill color is red):

```
(define (*find-fill-pixels* start-pixel)
  (find-all-connected-pixels-up-to-boundary-color start-pixel black))

(define (*fill-action* start-pixel current-pixel)
  (set-pixel-color! current-pixel red))
```

The second example, a "randomized" fill up to a given radius, could be written as follows (here, the radius is 10, and the "random fill color" is red):

```
(define (*find-fill-pixels* start-pixel)
  (find-pixels-within-radius start-pixel 10))

(define (*fill-action* start-pixel current-pixel)
  (if (= (random 2) 0)
      (set-pixel-color! current-pixel red)
      '()))
```

### *7.3 Modularizing Programmable Applications Through Libraries*

The two sample libraries discussed in this section illustrate a more general software-engineering strategy for developing and expanding programmable applications. By loading in a personalized collection of libraries, we can build our own enhanced language as the union of sub-languages, each tailored to some domain of interest to us. Thus, we might imagine (say) a graphics application for designing charts and graphs, and whose language is built from a variety of distinct cooperating chart-drawing libraries (one for drawing, e.g., bar, line, and pie graphs; another for drawing time-lines; another for drawing mathematical functions; and so forth).

It is worth considering, in contrast, the dilemma encountered by most users of graphics applications. Typically, someone who begins by owning a single graphics application will eventually own many: a paint program, a draw program, a flowchart-drawing program, a business-chart-drawing program, a geometric diagram-drawing program, and more. The reason for this plethora of programs (which are inevitably less than completely compatible) is that the user needs various special-purpose graphics features at different times; and no one program is ever satisfactory for this range of concerns.

Programmable applications—particularly those built by "enriching" a general-purpose language environment—offer an alternative to this scenario through the use of libraries. One can imagine, for instance, a SchemePaint user loading in libraries for the various specialized domains mentioned in the previous paragraph; and not only would the user have relatively little new material to learn (besides vocabulary—the core Scheme syntax remains constant), but the usually horrendous complications of multiprogram integration are alleviated if not eliminated entirely.

The notion of modularizing programmable applications through libraries brings with it a variety of attendant complications and software engineering issues. First, we note that modularization takes place both at the language and interface level simultaneously: our libraries must be designed to achieve some measure of cooperation in both realms. For instance, in the tiling library, we noted that the foreground color of the Escher window was determined in the same way as for the canvas window—i.e., either by selecting a color directly (in the palette window) or by evaluating a set-foreground-color! expression. Similarly, the polygon objects of which primitive tiling pictures are constructed are the same polygon objects created both by direct manipulation and by SchemePaint expressions. In designing a library for a programmable application, then, we need to take account of (and take advantage of) both the linguistic and interface-related contexts into which the library will be loaded. Going a step further, we might also wish to create "second-order" libraries which can be thought of as enhancements to our original libraries, and which are expressly designed to foster collaboration between libraries (e.g., by translating data structures from the form used by one library into that used by the other).

A second issue raised by the introduction of libraries is the question of how one achieves modularity in language design. This is similar in spirit to the overall task of programmable application design, which depends crucially on the creation of learnable and expressive embedded languages—but now there are added difficulties. Typically, a programmable application designer would like to think of an embedded language as a conceptual "layer" in a hierarchy of languages: the new embedded language makes use of more "primitive" data object types from the underlying language, and communicates with that underlying language only through the medium of procedures that construct and access abstract data types. When a collection of libraries is added to this picture, new patterns of inter-library communication are introduced; we can no longer think of the various libraries as occupying a strict ordering or hierarchy of complexity. Thus, we may want a solid geometry library to work with a tiling library (to place, say, a planar tiling design on the surface of a tetrahedron); and we may want the tiling library to work with solid figures (say, to tile three-dimensional space with cubes); and we may want both libraries to make use of procedures for incorporating perspective elements such as "vanishing points" in three-dimensional scenes. In short, the conceptual pecking order between various libraries is not as clear-cut and amenable to abstraction barriers as is that between an "embedded" and "host" language (as exemplified in the "core" SchemePaint system); and finding strategies for constructing and maintaining these more complicated language collections will be a recurring problem in programmable application design.

## 8. SchemePaint Pictures; Related Work; Future Research

### *8.1 SchemePaint Pictures*

By way of summing up the features of SchemePaint discussed so far, Color Plates 1-9 illustrate graphical work done with the program. Plates 1 and 2 were generated using the planar maps package; the first depicts three "basins of attraction" of a complex map. The fixed points of this map are the three cube roots of 1, and each point in the complex plane is shaded according to which root it approaches (and how quickly that approach is made) under Newton's

root-finding method. The second color plate depicts the result of applying a particular set of superposed affine maps (the notion is described in Section 5.2.3); here the maps are actually constructed with small random factors to provide a more naturalistic "botanical" appearance.

Plates 3-8 all illustrate the same basic idea: namely, SchemePaint's ability to combine hand-drawn and computer-drawn figures to good effect. Plate 3 (the amateurish one, drawn by the author) depicts a bee in a honeycomb. The bee is of course easily created by hand but would be hard to create via code; whereas the honeycomb is produced by a relatively straightforward turtle program but would be tedious (though admittedly not impossible) to create via direct manipulation in most commercial graphics programs. Plates 4-8 make a similar point (but with higher-quality artwork, by Orca Starbuck). Plate 4 incorporates a simple "rotated octagon" figure, similar to the pattern shown in Figure 5.3 earlier; note that the amount of programming required to generate this pattern amounts to three lines of SchemePaint code. Plates 5-7 include patterns described in Abelson and diSessa's book *Turtle Geometry*{1}: recursive tree-like shapes (Plate 5), the "dragon curve" (Plate 6), and an "inspi" variant of a simple spiral pattern (Plate 7). The feathers in Plate 8 are in fact created by a procedure whose structure is nearly identical to the tree-generating code of Plate 5.

Finally, Plate 9 was created by combining the features of the Escher tiling library (to make the fish-and-butterfly design) and the planar maps package (to "wrap" the tiled plane around a sphere).[19]

### *8.2 Related Work*

SchemePaint may be fruitfully compared to other work along two separate dimensions: as a graphics application *per se*, and (more generally and perhaps more interestingly) as a programmable application.

Certainly many commercial graphics applications include features that SchemePaint does not (mouse-selectable brush-types, fill-patterns, and more elaborate text-manipulation facilities are typical examples). Indeed, many of these features do represent desirable potential enhancements to SchemePaint. On the other hand, by virtue of SchemePaint's programmability—because it allows the user to express control constructs such as recursion and to build complexity by naming and parametrizing procedures and data structures—it is capable of a far wider range of expression than any of the better-known commercial applications. (Compare, for instance, the functionality of the systems reviewed in {16}.) Moreover, it is not entirely unfair to point out that there is no structural reason preventing the inclusion of additional features such as "brush types" within SchemePaint—that is, it would in no way represent a rethinking of SchemePaint's architecture to include these features either in the core application or as a library. In contrast, "pure" direct manipulation programs cannot approach the expressive range already present in SchemePaint without undergoing a philosophical conversion and incorporating a full-fledged programming environment.

---

[19] Plates 1-8 all appear as well in Eisenberg {8}; Plate 9 in this paper is a variant of the (planar) Plate 9 of the earlier paper.

Several interactive graphics programs do incorporate programmability. Beckman{5} describes a Scheme graphics package (but apparently this system does not include direct manipulation interface features). Sherin{18} has constructed a Boxer program that combines direct manipulation features (e.g., for drawing lines using the mouse) with a full-featured Boxer programming environment{7} ; his system also includes an extremely interesting option that allows direct manipulation operations to be translated automatically into equivalent Boxer statements. The graphics language of Sherin's system, however, is less extensive than that of SchemePaint (it does not include, e.g., planar maps, color-manipulation features, or the interface-language cooperation features described in Section 6). Lieberman{14} describes an interesting variation on the theme of programmability: his graphics program works to induce procedural descriptions of the user's direct-manipulation operations, which are rendered into Lisp and made accessible to the user for editing.

In domains other than graphics, there are instances of commercial or widely-known applications that incorporate some degree of programmability. Examples include Mathematica{S7} (for mathematical programming and symbolic algebra) , 4th Dimension{S3} (a database system), Director{S2} (for constructing animations), Stella{S8} (for simulating dynamical systems). Each of these programs includes a new, *ad hoc* programming language (as opposed to an "enhanced" dialect of some existing language). In contrast, AutoCAD{S1} is a programmable application based on AutoLisp, a "design-enriched" Lisp dialect; similarly, one might describe the venerable Emacs editor{20} as an early example of a programmable application (in its current instantiation, Emacs is, like AutoCAD, based on its own "enhanced" Lisp dialect). All these applications have experienced longevity and success, though each in some measure deviates from the programmable application "ideal": the applications based on *ad hoc* languages exhibit problems deriving from that choice (such as the absence of a clear language semantics or a powerful programming environment), while the Lisp-based applications are not really designed to lead novice users into programming by tight integration of interface and language features.

### *8.3 Continuing SchemePaint Development*

SchemePaint is expressly intended as a prototype of a programmable application in one particular sample domain; as such, it is the first of a projected suite of sample applications in different domains. In practical terms, this means that continuing work on SchemePaint will proceed on only a part-time basis, in tandem with development of other (conceivably related) applications. Nevertheless, there is an extensive slate of enhancements planned for the program; some of these are motivated by research issues mentioned later in Section 8.4.

In the near term, several new library files are currently being created for the program. These include a package of 3D-mapping procedures (and a 3D-turtle); a geometric-diagram package; an enhanced text manipulation package; and a function-graphing package. As noted earlier, a rudimentary Basic interpreter has been implemented (to be viewed as an alternative language environment for those users unfamiliar with Scheme); and a Logo interpreter, created in a

similar spirit, is at an earlier stage of development. Other possible enhancements include multiple-window features (e.g., primitives that combine the contents of windows according to the dictates of user-defined procedures); brush-styles (an exciting, though as always non-programmable, feature of the best current commercial packages); and procedures for manipulating scanned-in artwork and "clip art" files.

Longer-term development is also planned for SchemePaint—experimental enhancements of the program that go the beyond "quantitative" expansion projected in the previous paragraph. One planned project (in collaboration with G. Fischer) would involve incorporating knowledge bases into the basic SchemePaint structure; these would form the basis of *critics*{9} to assist users in the design of information charts and displays. (Cf. {6}, {23}.) The resulting system should also include a browsable database (or "catalog") of exemplary instances of chart design that can be employed as starting points for new user-created charts. A second major avenue of development focuses on the use of SchemePaint as a venue in which novice users can learn to program; this direction of work would likely include the creation of embedded "tutoring" programs, files of illustrative SchemePaint examples, and perhaps enhancements that allow direct manipulation operations to be translated into editable programs.{8}

### *8.4 Research Directions*

The creation and use of programmable applications present a wealth of fascinating questions for research. Some of these have been alluded to in previous sections in the context of the SchemePaint program in particular—embedding tutors or knowledge bases within programmable applications; modular design of domain-specific languages; the use of multiple programming language environments in a single application; finding avenues of creative symbiosis between the "interface" and "language" portions of a given application. Naturally, all of these questions transcend the particular example of SchemePaint; ideally, research into these issues should inform the design of programmable applications in a wide range of domains.{8}

There are still other major questions in programmable application design that deserve mention here. We would like to know, for instance, what type of "software life cycle" to expect from these applications—how they are debugged, maintained, and extended, and what sorts of user communities grow around them. We would also want to investigate what sorts of programmable applications might be developed for use by children: assuming that one wished to develop a version of SchemePaint accessible to (say) fifth-graders, which of the original design decisions in interface and (especially) language would have to be rethought or abandoned? Finally—and perhaps more futuristically—we would like to extend the programmable application concept to incorporate the exciting new developments in interface hardware (3D-viewing devices, "DataGloves," and so forth) that are likely to appear in commercially available systems in the near future.{15} All these areas of research could have direct application to SchemePaint and its future instantiations; but again, all have more general consequences for programmable application design. In pursuing these questions, we as application designers can work toward a world

of tools in which language and interface, word and hand, are both given their due; and we can look forward to a world in which programming languages are not treated as arcane cryptograms hidden deep beneath a smoothly polished graphical interface, but rather as means of creative expression available to all.

## Acknowledgments

The ideas of Hal Abelson, Andy diSessa, and Gerald Jay Sussman have been the primary motivating influences behind this work. Thanks are also due to Barry Dworkin, Wally Feurzeig, Gerhard Fischer, Mark Friedman, Matthew Halfant, Paul Horwitz, Roy Pea, Mitchel Resnick, and Franklyn Turbak for advice, erudition, encouragement, and criticism. Orca Starbuck generously contributed her time and artistic talent to create five SchemePaint pictures.

## References

{1} Abelson, H. and diSessa, A. *Turtle Geometry*. MIT Press, Cambridge, MA 1980.

{2} Abelson, H. and Sussman, G. with Sussman, J. *Structure and Interpretation of Computer Programs*. McGraw-Hill, New York; MIT Press, Cambridge, MA 1985.

{3} Abelson, H. and Sussman, G. "Computation: an Introduction to Engineering Design." MIT Artificial Intelligence Memo 848a, 1986.

{4} Barnsley, M. Fractals Everywhere. *Fractals Everywhere*. Academic Press, Inc. Boston, MA 1988.

{5} Beckman, B. "A Scheme for Little Languages in Interactive Graphics." *Software—Practice and Experience*, 21:2, 1991.

{6} Bentley, J. L. "Document Design." *Communications of the ACM*, 29:9, 1986.

{7} diSessa, A. and Abelson, H. "Boxer: a Reconstructible Computational Medium." *Communications of the ACM*, 29:9, 1986.

{8} Eisenberg, M. "Programmable Applications: Interpreter Meets Interface." MIT Artificial Intelligence Lab Memo No. 1325, October 1991.

{9} Fischer, G. et al. "The Role of Critiquing in Cooperative Problem-Solving." *ACM Transactions on Information Systems*, 9:2 , 1991.

{10} Foley, J. "Interfaces for Advanced Computing." *Scientific American*, 257:7, 1987.

{11} Grabowski, R. with Huddleston, D. *Using AutoCAD*. QUE, Carmel, IN 1991.

{12} Henderson, P. "Functional Geometry." *1982 ACM Symposium on Lisp and Functional Programming.*

{13} Hutchins, E.; Hollan, J.; and Norman, D. "Direct Manipulation Interfaces." In *User Centered System Design*, Norman, D. and Draper, S., eds. Lawrence Erlbaum Associates, Hillsdale NJ 1986.

{14} Lieberman, H. "Mondrian: A Teachable Graphical Editor." MIT Media Lab Technical Report, 1991.

{15} Marcus, A. and van Dam, A. "User-Interface Developments for the Nineties." *IEEE Computer*, 24:9, 1991.

{16} McClelland, D. "Paint Imitates Life." *MacWorld*, March 1992.

{17} Papert, S. *Mindstorms*. Basic Books, New York 1980.

{18} Sherin, B. Personal communication.

{19} Shneiderman, B. "Direct Manipulation: a Step Beyond Programming Languages." *IEEE Computer*, 16:8, 1983.

{20} Stallman, R. GNU Emacs Manual (Fifth Edition, Version 18). Free Software Foundation, Cambridge MA 1986.

{21} Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA 1977.

{22} Thompson, J.M.T. and Stewart, H.B. *Nonlinear Dynamics and Chaos*. John Wiley and Sons, Chichester 1986.

{23} Tufte, E. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT 1983.

*Software*

{S1} *AutoCAD*. Autodesk, Inc. Sausalito, CA.

{S2} *Director*. MacroMind, Inc. San Francisco, CA.

{S3} *4th Dimension*. Acius, Inc. Cupertino, CA.

{S4} *Hypercard*. Apple Computer, Inc. Cupertino, CA.

{S4} *MacPaint*. Claris Corporation. Santa Clara, CA.

{S5} *MacScheme*. Lightship Software, Inc. Beaverton, OR.

{S6} *Mathematica*. Wolfram Research, Inc. Champaign, IL.

{S7} *Stella*. High Performance Systems, Inc. Hanover, NH.

## Color Plate Key for SchemePaint Pictures

**Plate 1.** A "basins-of-attraction" map created using the embedded planar maps package in SchemePaint. The colors red, green, and blue correspond to the three complex roots of 1; points are shaded according to which root they approach under Newton's root-finding method.

**Plate 2.** Tree-like figures created by iterating "superposition maps" in the planar maps package.

**Plate 3.** A hand-drawn bee in a computer-drawn honeycomb.

**Plate 4.** A snake slithers through a simple rotated-octagon figure.

**Plate 5.** A hand-drawn zebra munches on fractal (turtle-drawn) trees.

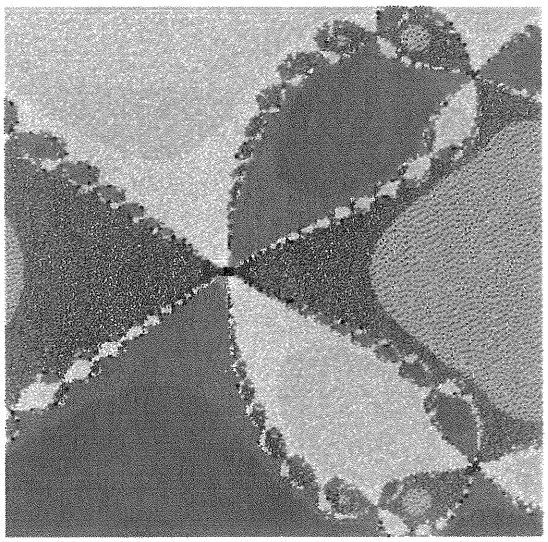
**Plate 6.** The "dragon curve" decorates a hand-drawn seahorse.

**Plate 7.** The waves are generated by the inspi procedure from Abelson and diSessa's book Turtle Geometry.

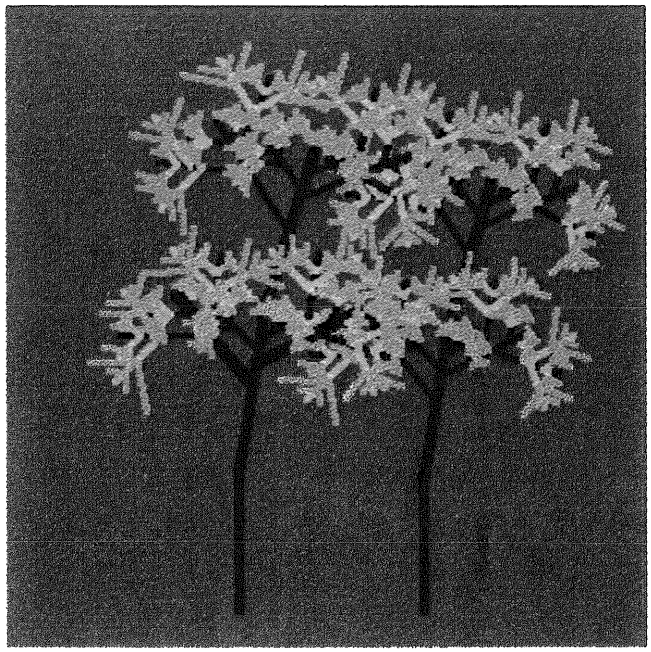
**Plate 8.** A peacock displays fractal feathers.

**Plate 9.** An Escher-esque tiling of fish and butterflies wrapped around a sphere.

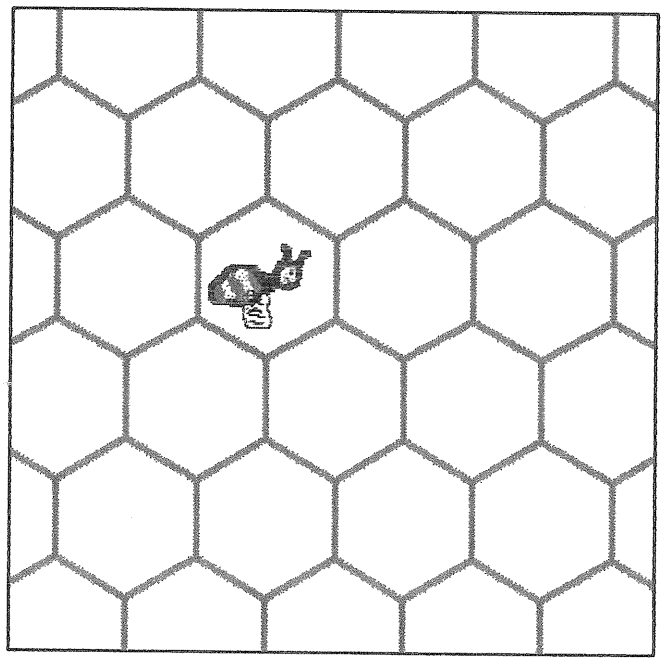
Artwork for Plates 4-8 by Orca Starbuck.



<--- Color Plate 1

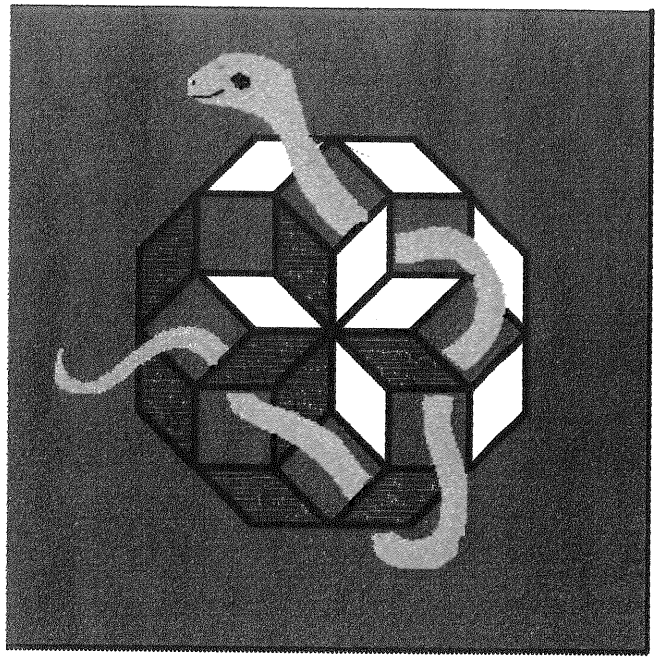


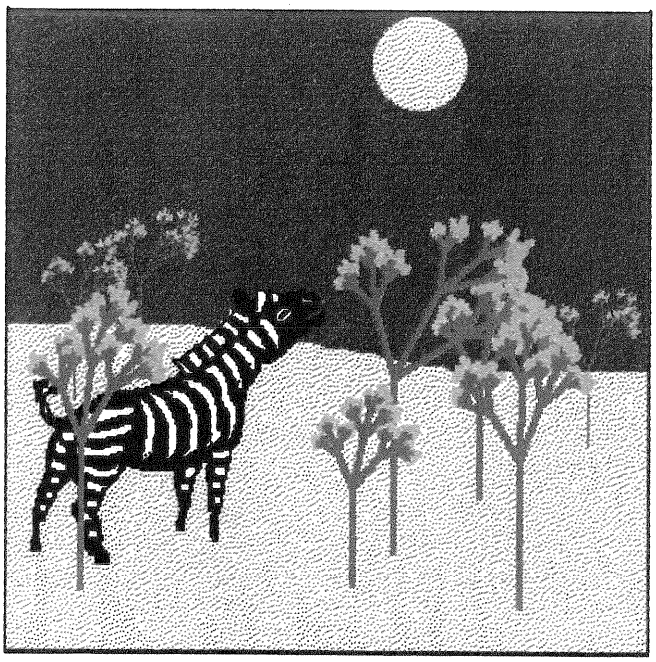
Color Plate 2 --->



<--- Color Plate 3

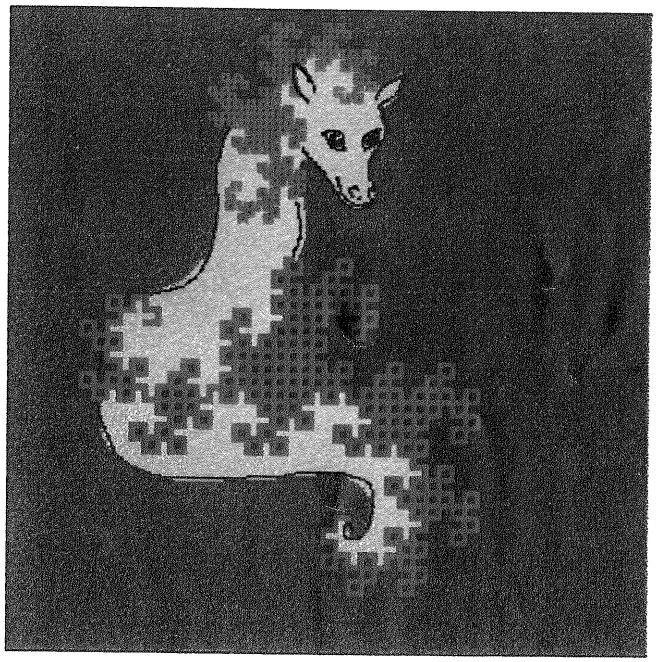
Color Plate 4 --->

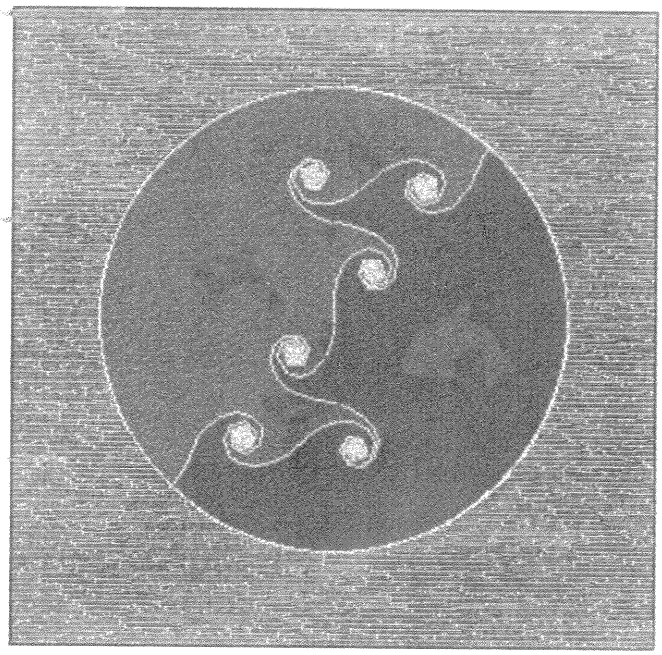




<--- Color Plate 5

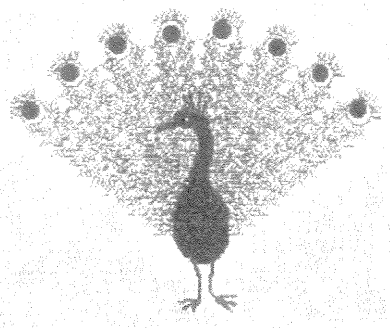
Color Plate 6 --->

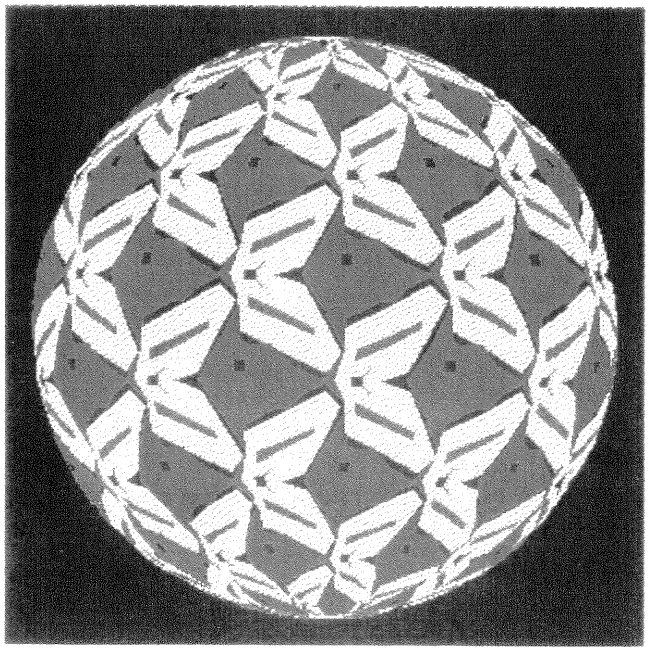




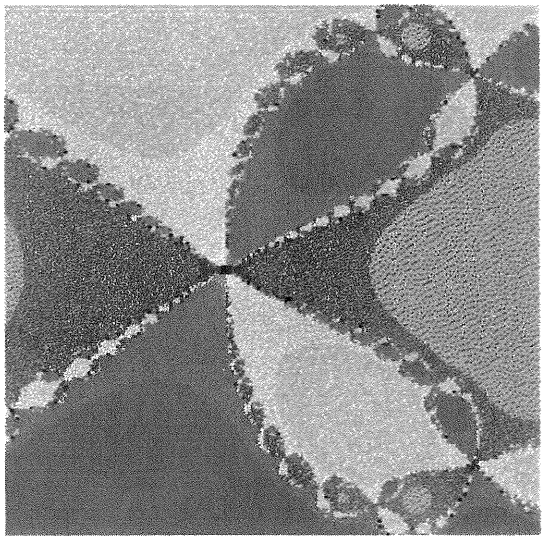
<--- Color Plate 7

Color Plate 8 --->



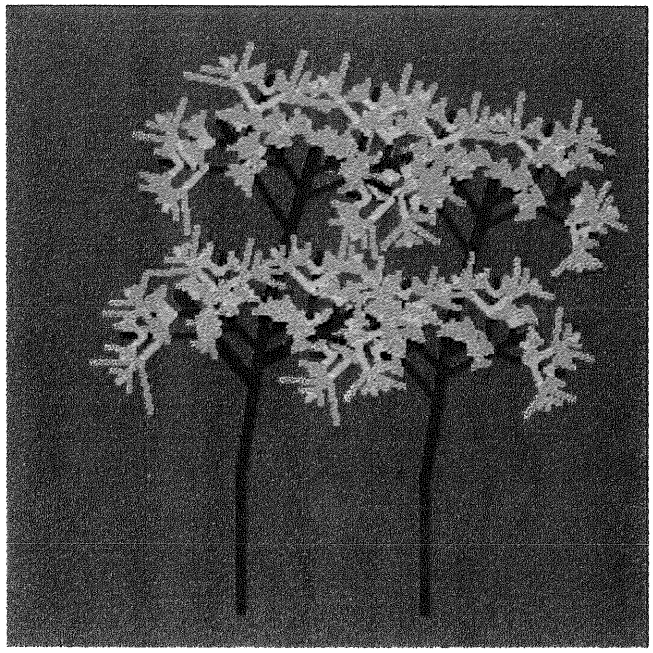


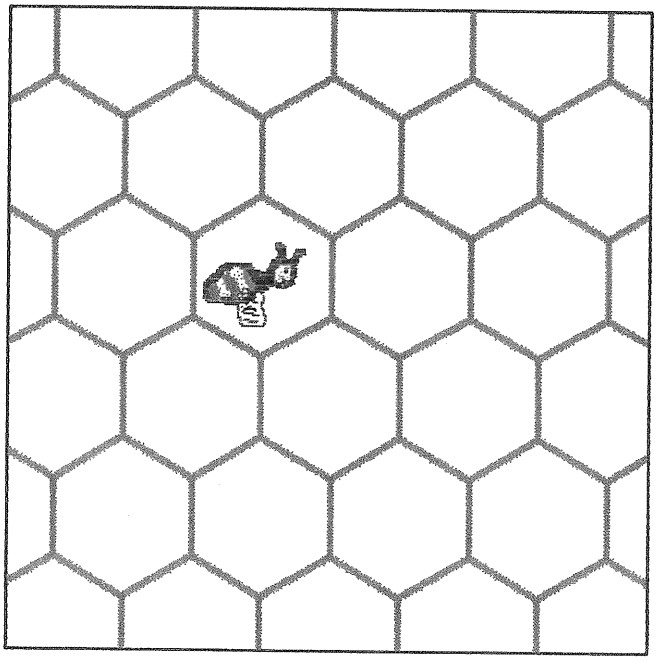
<--- Color Plate 9



<--- Color Plate 1

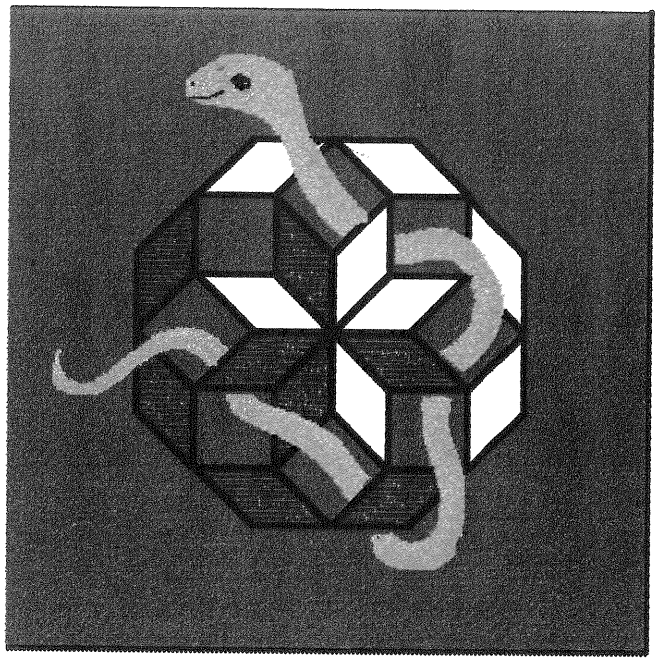
Color Plate 2 --->

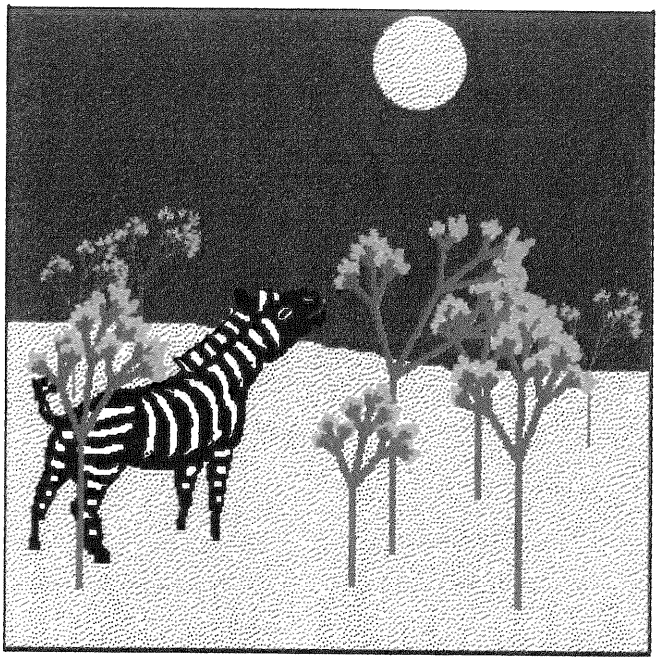




<--- Color Plate 3

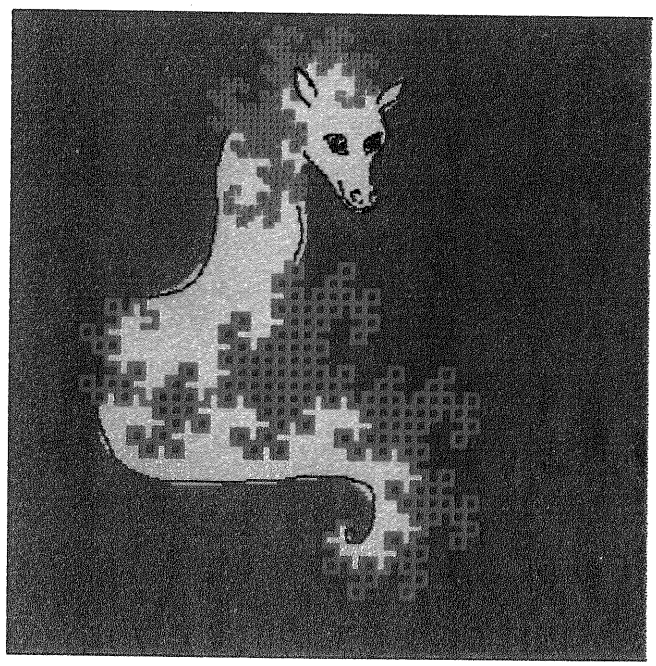
Color Plate 4 --->

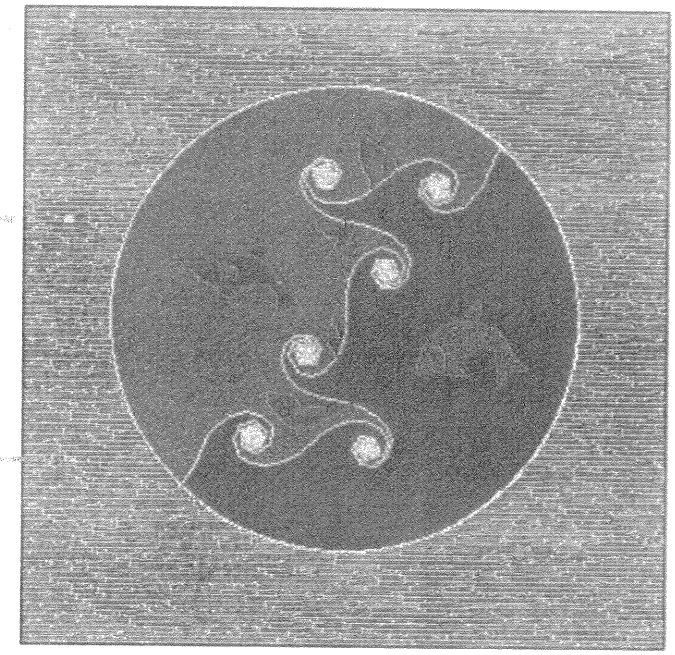




<--- Color Plate 5

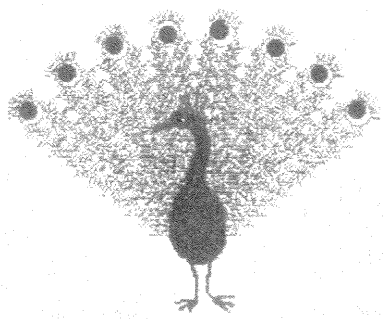
Color Plate 6 --->

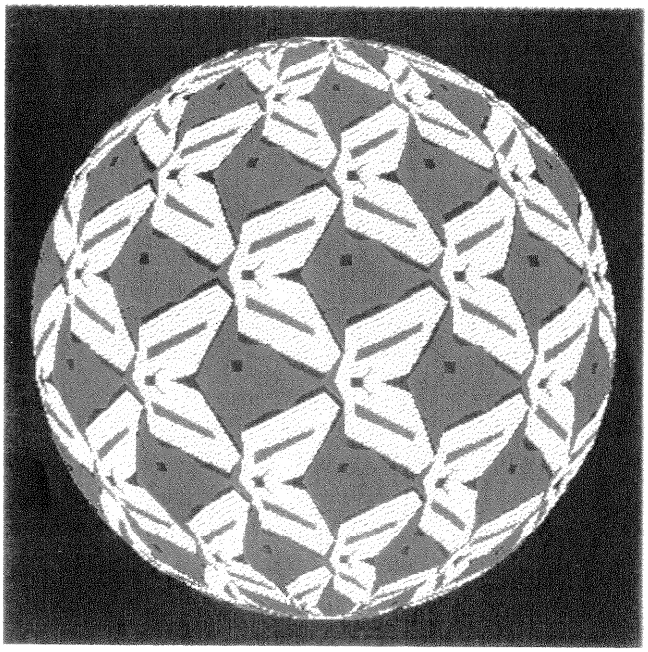




<--- Color Plate 7

Color Plate 8 --->





<--- Color Plate 9