MASSACHUSETTS INSTITUTI
ARTIFICIAL INTELLIGENC

A.I. Memo No. 1325

Programmable Ap
Interpreter Meet

Michael Eise

Abstract

Current fashion in "user-friendly" softw
overreliance on direct manipulation inte
(and thus truly user-friendly), applicatio
faces and domain-enriched languages th
This paper discusses some of the desig
ation of such *programmable applications*
"SchemePaint," a graphics application t
interface with an interpreter for (a "grap

Copyright © Massachusetts Institu

This report describes research done at the Artificial Int
Institute of Technology. Support for the laboratory's
in part by the Advanced Research Projects Agency of t
Naval Research contract N00014-89-J-3202 and by the
number MIP-9001651.

Programmable Applications: Interpreter Meets Interface

Michael Eisenberg
MIT Laboratory for Computer Science

Abstract. Current fashion in "user-friendly" software design tends to place an over-reliance on direct manipulation interfaces. To be truly expressive (and thus truly user-friendly), applications need both learnable interfaces and domain-enriched languages that are accessible to the user. This paper discusses some of the design issues that arise in the creation of such *programmable applications.* As an example, we present "SchemePaint," a graphics application that combines a MacPaint-like interface with an interpreter for (a "graphics-enriched") Scheme.

Despite the fact that it is hard to write, hard to maintain, and hard to market, the fact of the matter is that good software exists. A visit to the local home-computer shop will prove it. The shelves are lined with beautifully packaged paint programs, music programs, statistics programs, CAD systems, games, databases, word processors. Many of these programs are best-sellers, and deservedly so. Their users will swear by them, and compare the features of one wonderful program against another, waiting for the next release of some favorite application to see what new amazing functionality has been added.

But underneath the well-earned praise of the best applications software, there is often a muted note of frustration. Somehow it seems that the software never does quite enough. The competitor's music program allows you to specify notes other than those in the twelve-tone scale; the word processor that your friend owns permits text regions of non-rectangular shape; the physics simulator at the school next door has a wider choice of integration algorithms. Or perhaps some straightforward task is impossible in *everybody's* version: the paint program can't be found that allows the user to draw a logarithmic spiral, or a sine wave, or a Lissajous figure.

After several years of running into barriers, users begin to write to the software company, asking for more functionality; often their wishes are granted. But like the legend of the Monkey's Paw, the granting of wishes only makes the problem worse. Now the paint program, version 5.0, includes spirals; and Lissajous figures (just specify two component frequencies); and sine waves (the dialog box asks for frequency, amplitude, and phase); and reflections in mirrored spheres; and color gradients; and non-regular hexagons; a hieroglyphics library; and much, much more. The menus have metastasized into nested submenus, each of which leads to a scrolling box whose list of choices branches off into multiple dialog boxes. It's impossible to keep straight which options are included in the software, which are included in somebody else's software, and which are still in the works for version 6.0. And the program *still* doesn't do enough.

The problem with version 5.0 of the paint program is actually a corollary of the problem with version 1.0. In fact, the problem with version 5.0 originated before version 1.0 ever existed. It originated on the drawing board, with a design decision that insisted on keeping the activity of programming sacrosanct. It began with the assumption that users never, ever want to write programs, no matter how small those programs might be or how much fun the user might have.

1. A Paradigm For Software Design

The potential exists for a new class of applications software, more expressive, more extensible, and more respectful of the user's imagination than that which preceded it. Instead of dividing the world into computer gurus and computerphobes, these applications would welcome the spread of a programming culture. And because these applications would make maximum expressive power their fundamental goal, they would truly be user-friendly, rather than—as is often the case—user-condescending.

The central design precept behind this software is easy to summarize, if tricky to realize. Applications should contain both an extensive, learnable direct manipulation interface *and* an interpreter for at least one well-supported programming language. In other words, we would like to combine the very real successes of the most popular home-computer applications (typified by

"Macintosh style" interfaces) with the unimaginably rich expressive range of programming languages.

Ideally these two aspects of program design can combine symbiotically. The goals of the interface include learnability; ease of understanding; explorability; and aesthetic elegance. Perhaps most important, the interface should allow the user to perform those tasks best suited to the human skills of hand-eye coordination. On the other side, the goals of the programming language should include a good match between language primitives and domain concepts; the ability to express important domain processes in short, simple programs; and pathways for incrementally increasing program complexity.

If these two "halves," interface and interpreter, were merely to coexist passively—if we were merely to slap together, e.g., the best paint-program interface with a Logo interpreter—we would already have a product arguably more exciting than most. But, again, these two halves can in fact do more than coexist in conceptual isolation. They can support one another, each building on the strengths of the other. Interface features can interact with programming constructs in a variety of imaginative ways. It is in such directions that many of the most interesting design possibilities await.

It is not easy to understand in retrospect why the two communities—those interested in learnable user interfaces and those interested in programming environments—have ever perceived themselves as being at odds with one another. The reasons may have more to do with sociology and researchers' personal style than with any kind of deep intrinsic philosophical division. To the user-interface designers, we might well ask, why not include programmability as one of the desiderata of a complete application? After all, if the point of an application is to provide the artist/musician/scientist with the most natural and attractive system possible, why not allow those who wish to program the opportunity to do so? And to the community of language designers and implementers, we might ask, why this fascination with the *general-purpose* programming environment? Why not include the development of domain-specific interfaces—maybe a whole collection of them—as part of the task of designing a useful language? The questions here go beyond adding menu- and window-objects to the language, but rather include issues such as the potentialities for building enriched language implementa-

tions, tailored to some particular domain.

In a very real sense, the two communities share and have always shared an important mutual interest—how to cultivate the use of programming as a means of expression. But despite this, despite the real gifts that the language designers and interface designers can offer one another, they often seem to view themselves in inexplicable opposition. Interface designers have taken it as an article of faith that users can't and won't write programs, and that the only way to sell an application is to trumpet that non-programmers can use it. (This without asking whether programmers *can* use it, or indeed whether non-programmers are happy in their condition.) Language implementers, in their turn, have treated interface and application writing as an afterthought to the main job of creating a general environment not particularly suited to anything; the resulting system is more of interest (really, *only* of interest) to professional programmers, and slights the myriad potential smaller-scale programming projects in which professionals of other stripes might indulge.

Language implementers are also not the only community that ignores the waiting audience of applications-users. In academic computer science, the study of programming languages is associated with issues such as denotational semantics, compilation techniques, and parallelism; and even "software engineering" typically focuses on specification, coding, and debugging of large systems software projects. These concerns are undoubtedly important; but at the same time, the design of personal tools and applications is unfortunately viewed as a "soft" issue, unworthy of rigorous scientific study. Why should this be the case? The creative potential of the individual, laboring to find a voice through the machine, is slighted. It is as if students of writing were to ignore novels, short stories, love letters, and folk songs, and to concentrate upon Webster's Unabridged Dictionary. We should rather look forward to the day when the subjects of "language design" and "software engineering" can also connote the creation of enjoyable, learnable customized languages and applications.

The remainder of this paper presents a fuller portrait of combined interface/interpreter systems, or what might be called "programmable applications." Sections 2 and 3 briefly review some of the problems inherent in providing only one or the other of the interface/interpreter dyad. Section 4 is a preliminary discussion of the design issues involved in creating pro-

grammable applications. In Section 5, a sample programmable application is presented: "SchemePaint," a program that combines the direct manipulation features of standard paint applications with a "graphics-enriched" Scheme interpreter. This example helps to make concrete some of the issues that were introduced in the previous sections. Section 6 presents some of the more interesting and provocative objections to the programmable application concept, and responses to these objections. Finally, Section 7 is a discussion of where programmable applications might lead: we present a number of possibilities for additional sample projects, and draw the (very skeletal) outlines of a research agenda for the future.

2. Direct Manipulation and Its Discontents

"Direct manipulation" is of course a broad term, and one whose definition is better indicated in practice than through abstract classification. Although there are many variations on the theme already in existence, the best collective example of direct manipulation is provided by the Apple Macintosh—its operating system and its most popular software packages. Specifically, we might regard programs whose complete interface is embodied in menu choices, icon selection, dialog boxes, and mouse clicks as good examples of direct manipulation systems.[1]

Direct manipulation at its best has a number of extremely attractive features, of which learnability and ease of use are among the most important. It is possible to learn the rudiments of the Macintosh operating system in an hour or two; and beginning projects can be undertaken on the best Mac software within an afternoon. Many programs show a wonderful attention to matching the techniques of manipulation to the user's intuitions. Just to cite one example: the ImageStudio drawing program {S6} includes an "index finger" icon which may be used as a kind of "smudging device"; by moving the finger back and forth over an already-drawn line, one can get the effect of "rubbed pastel" on the screen. This is a delightful feature, and the illusion of smudging a pastel line by hand is surprisingly strong; it is hard to imagine

[1] It should be mentioned in this context that "Macintosh-style programs" are now widely available on other home computers, and on high-end workstations. Thus, although our discussion uses the Mac as a source of examples, our observations should not be interpreted as applicable only to Apple machines and software.

even an occasional ImageStudio user either missing or misunderstanding this feature of the program.

Marvelous examples such as this abound in Mac-style software. But problems arise when these marvelous examples pique the user's interest and imagination, and suggest some new wrinkle not anticipated by the interface designer. Perhaps—just to pursue the "smudging finger" feature described in the previous paragraph—one would like to designate only certain colors as "smudgable"; that is, one would like some colors to be responsive to the smudging operation and others to be unaffected by it. This is a perfectly reasonable thing to want, but if it is not explicitly included in the interface the user has no hope of obtaining the desired effect.

It is worthwhile to underscore the point of the previous paragraph. The complaint is not that there are certain features missing from certain programs, or that those programs need to incorporate a new list of menu items. The complaint is rather that pure graphical interfaces are almost invariably severely limited in expressive power; the semantics of clicking/dragging/selection are simply too impoverished to accommodate the imagination of long-term, serious users. This statement is a matter of empirical fact rather than logical deduction; one might come up with the occasional circumscribed domain in which a graphical interface is close to ideal. Moreover, speaking precisely, it is of course possible to devise graphical mouse-driven programming languages that are Turing equivalent and hence have all the expressiveness one could wish. In pragmatic terms, however, these objections beg the issue. As of yet, there are no purely graphical programming languages in widespread use, and it remains unclear whether such languages will ever win converts away from textual programming. And—as to the first objection—there are still very few interesting applications that are sufficiently circumscribed so as to be both non-programmable *and* unimprovable by the incorporation of a language.

Once one starts examining the shelves and shelves full of applications software, the omission of programmability looms as a pervasive, near-universal, and nearly always major flaw in even the best systems. A music-composition program advertises that it allows crescendi and decrescendi to be notated directly on the screen: the composer selects one bar as *piano*, a later one as *forte*, and (via menu) notates the bars in between as a crescendo, and the

program will play the music with smoothly increasing amplitude. All well and good. But because amplitude is not programmable, the composer cannot express the notion of increasing it nonlinearly, varying it with a little bit of randomness, or making it dependent on the pitch or timbre of the note being played. We turn to the educational software aisle: here's an astronomical observatory program that allows the student to view a representation of the sky from any point on earth, at any time from centuries ago to centuries in the future. Simply click on a location (via a mouse-sensitive map of the earth), and click on a time (via a mousable year/month/day selector), and the stars appear on the screen. Mind-boggling effort and ingenuity has been lavished on the creation of this program, and the result is indeed beautiful. But because we cannot write programs in which time is a variable, we cannot see the path of Orion animated, night by night, over the course of a summer; the retrograde motion of Mars is likewise hard to see; nor can we chart a course to the North Pole and see how the night sky smoothly shifts as we change location. Well, perhaps this video-editing system in the brand-new multimedia aisle will satisfy us. Here, we can cut-and-paste video clips into sequences—just what an editor needs. But might not a video editor actually need something a bit different? Suppose she has retrieved two video clips and wants to find the *perfect* frame from which to cut from the first to the second; might she not wish to view the same two clips over and over, incrementing the final frame of the first clip until the effect was just right? This would be a crisp, half-page program to write in a language with "video-clip data objects"; but because the editor is limited to mouse clicks, she has to construct each viewing sequence by hand.

There are certain key elements—really, defining elements—of programming languages that are absent from virtually all existing "pure" direct manipulation interfaces. First, users of these interfaces rarely have access to the standard control structures and concepts of programming languages: conditionals, loops, recursion, and so on. The user of our sample musical-composition program, for instance, can specify where a crescendo starts and ends; but he cannot express the notion that *if* a note is higher in pitch, it should be played a bit louder. Likewise, our hypothetical video editor cannot express the concept of *repeatedly* viewing two clips in sequence, varying a frame number at each iteration. A user of any standard paint program cannot make a simple spiral, because he cannot write what amounts to a

two-line tail recursive procedure in Logo.

A second element missing from direct manipulation interfaces is the ability to create compound data objects—arrays, lists, and structures. Often one can select groups of objects on which to perform a particular operation (e.g., a set of graphical elements to move); but cannot re-use the same grouping repeatedly, append it to some other grouping, create an ordering among a set. We might have drawn several trees while working with a paint program, but we cannot designate that group as a stable entity: every time we want to re-color all the trees, resize them, whatever, we have to select the whole set all over again. We cannot recolor all pixels whose RGB color is within a certain range, both because we cannot express conditionals (as noted in the previous paragraph), and because we cannot create stable groupings based on those conditionals. To take one last example: in the astronomical observatory program, there is no straightforward way to take statistics—to ask, for instance, the mean number of hours per evening that Venus is visible during a year—since doing so would most likely require lists or arrays of numbers.

Finally, there is no way of *naming* procedures or objects; thus there is no way of using abstraction to build more complex operations from simple ones. Having created a graphic procedure to draw a fractal flower, we might want to draw a bunch of flowers, varying slightly by color, length, petal-number, and placement; but without a named `flower` procedure, and without the ability to parametrize that procedure by color, length, and so on, we are stuck. We cannot use our `flower` procedure to make a more complex `bouquet` procedure, or `window-box` procedure, or to make geometric designs incorporating fractal flowers. Again, without the presence of named procedures in the observatory program, we cannot express the general idea of looking for the closest approach between two objects in the sky over a given period as seen from a given location; and we cannot use a procedure for that purpose to find the closest approach between the moon and sun over a given year (as seen, e.g., from Boston); and we cannot use *that* procedure to write another that looks for the occurrences of solar eclipses in a given century; and we cannot use *that* procedure to write another that graphs the number of solar eclipses visible from the Northern Hemisphere by century. The ability to name composite objects and operations, and the ability to employ named parameters within operations, are crucial to the process of abstraction. Without these

abilities we cannot express higher-level concepts than those we began with. The concept of an "eclipse" is not built into the observatory program, but if we can write procedures we can express that concept nonetheless; we can write procedures that search for eclipses, record them, take statistics about them.

What is missing, then, from a "pure" direct manipulation interface is precisely what a programming language provides: the ability to express straightforward control constructs, the ability to create stable composite sets of data objects, the ability to build complexity through naming operations and objects. And because these things are missing, the boundless range of concepts expressible through them is likewise unavailable. As a result, the designers of direct manipulation interfaces are inevitably confronted by frustrated users. Typically, rather than redesign the system by incorporating programmability, the interface designers will instead add patchwork fixes—unrelated, *ad hoc* features tacked on to the original interface. The next-generation observatory program may include "eclipse searches"; the next-generation video-editing program may include a "repeated viewing" menu selection; and so on. Like the "version 5.0" graphics program mentioned in the opening paragraphs of this paper, these newer-generation programs do not address the core reasons for users' discontent. The problem in the observatory program, for instance, has nothing to do with the absence of "eclipse searches" *per se*; and no finite set of similar enhancements can possibly fix the problem. What is missing is a medium of expression in which concepts can be built, named, saved, re-used, extended, combined. What is missing is language.

3. The (Too-) General Purpose Programming Environment

The previous section argued the expressive inadequacy of direct manipulation unleavened by programmability. But it's possible, of course, to go to the other extreme and leave everything to programming—i.e., to provide the user with only the bare-bones interface that usually accompanies a programming language environment. For example, rather than provide a paint program to the artist/user, the designer might simply sit him in front of a Lisp machine and tell him that everything he needs is here, somewhere. This may be acceptable (barely) to the Lisp expert, but will almost certainly alienate anyone who views himself as an artist first and programmer

afterward.

There are really two major gaps in general-purpose programming environments vis-a-vis their use in applications. First, programming languages themselves, while tremendously expressive, are insufficient for many tasks that people do more naturally without linguistic mediation. For instance, it is possible to write a Logo program that will draw a horse; but it's much easier for most of us to draw a recognizable horse using a pencil than using a turtle. There is simply a kind of wisdom that people have in their hands, in their eyes, that defies representation in code (and often in English as well). It is easier to draw a face than to explain to someone else—computer or human—how to draw a face.

For many applications, then, a programming environment can benefit from the addition of direct-manipulation tools that take advantage of those extra-linguistic talents that users may have. But as it happens, programming environments rarely do come bundled with interfaces geared to some specific (non-programming) domain. For instance, no Lisp or Pascal environment of which I am aware comes with a loadable "paint interface" that augments the existing language system with mouse-driven tools for drawing polygons, splines, and so on.

There is of course a standard reply at this juncture for defenders of programming environments. They will say that these desirable interface features—mousable icons, menus, dialog boxes, *et al*—are, or ought to be, constructible within the environment. But why should expertise in building dialog boxes in Pascal be a prerequisite for using a paint application? To put it another way: even if the user has some mild programming experience, and even if he wishes to put that experience to use in working with a paint application, he should *not* be expected to have sufficient time or expertise to build that application from scratch. The artist/programmer is interested in language constructs that employ color, pen-size, texture, perspective; he is not therefore interested in constructs that build the interface itself.

The first problem, then, with "pure" programming language environments is that they are incomplete on their own in the absence of direct manipulation interfaces. Ideally, such an interface should provide the user with a quick, simple, engaging method for performing those functions that

are impossible (or maybe just too tiresome) to do by programming.

The second problem with general-purpose environments is that they are designed for the benefit of the professional programmer, and as a consequence are unsuited to other users. Consider what a musician might need in a programming language: wave-form objects, bar objects, sound libraries, chord libraries, a variety of others—along with primitive procedures to manipulate these object types. A standard programming environment will probably not provide any of these objects and procedures. To be sure, such "frills" could be built up within a sufficiently powerful environment; but, as noted a moment ago, this should be the *application programmer's* task, not the *musician's* (or other professional's) task. The musician should sit down at a fully-prepared musically enriched programming system—not a wilderness of tiny building blocks from which such a system might someday arise.

There is really a key requisite for a programming environment geared toward non-computer-science professionals: meaningful tasks, *within the domain of the user's interest*, should be expressible by a line or two of code in the user's very first program. This does not preclude, of course, the possibility that the environment could accommodate large programs; but it does constrain the environment to be much more "enriched" for a particular domain than its more general-purpose cousins.

To sum up, then, the second complaint about "pure" programming environments: they lack the linguistic tools that form the best possible match with domains outside of programming. (This is in contrast to the first problem mentioned above, which focuses on the lack of "extra-linguistic tools.") As such, the only people who feel intellectually comfortable within these environments are hackers.

4. Strategies for Integrating Mind-Work and Hand-Work

The previous two sections focused, Scylla-and-Charybdis-fashion, on the problematic aspects of direct manipulation (in isolation) and general purpose programming environments (likewise in isolation). In this section, we explore some of the key design issues that arise in steering a middle ground. The level of discussion here is necessarily speculative; many of the issues are unresolved, or allow (as design issues often will) for multiple answers. A few topics may represent embarkation points for empirical research. Nevertheless, it is worth at least introducing these issues here; and additionally, several of the topics first discussed in this section will be returned to and elaborated upon in the sections that follow.

4.1 Special-Purpose Language versus General Language

A bit earlier, we observed that general-purpose programming environments were unsuited to professional (non-computer-science) domains. This leads naturally to the question of just how domain-specific languages should be implemented. Granted that a "music language" or "mathematics language" should be written: how do we write it?

The most straightforward approach to this question is to invent a special-purpose language with syntax and vocabulary all invented from scratch. Such "little languages" do indeed currently exist in certain domains: the command languages supplied with many standard database programs comprise one type of example, as do the *Mathematica* language {S9} and the Lingo language associated with Macromind's *Director* program.{S3} Though in some individual cases, these applications occupy the "overemphasis-on-programming" end of the spectrum—i.e., they make too little use of direct manipulation—nevertheless, by and large these are highly successful products. Certainly the best of them do not suffer from the flaws of "pure" direct manipulation interfaces listed earlier. In the case of *Mathematica*, for example, an expressive and powerful application language has to date formed the basis of several textbooks and a quarterly journal.{21, 41, 42}

As we have noted, the designers of these applications have chosen to invent a completely new language—new syntax, new control structures, vo-

cabulary, everything—to go with each new program. There are admittedly certain advantages in this approach. Presumably, when creating a special-purpose language, designers can pursue the goals of learnability and domain-appropriateness. A hypothetical "database language," for instance, might be created in accordance with principles of language learnability, so that the database users could be expected to understand the language in a relatively short time. Or there could be extensive testing done with professionals in the user community to motivate particular choices of syntax and vocabulary.

It is dubious whether care of this kind has often (if ever) been taken in practice; but even so, the special-purpose language route does in principle have these points in its favor. There are strong advantages, though, to proceeding in a different way—namely, starting with a "standard" language like Lisp or Pascal as a base, and extending it for use with a particular domain.

Why make a "musical Pascal" or "musical Lisp" rather than a brand new "Musica" language? First, it is quite plausible that a number of applications might all be associated with (or accompanied by) the same base language. This would give users an important edge: it would mean that any user who learned the rudiments of (say) Pascal would be able to rapidly gain proficiency in *any* Pascal-based application. Someone who learned how to write Pascal procedures and functions could do so for a "music Pascal," a "graphics Pascal," and so on. (In contrast, most *ad hoc* application languages are so distinct that they represent a significant learning effort even for the experienced programmer.)

Conceivably, several applications based on the same programming language could be combined in such a way as to yield a more powerful "multi-domain" application. For instance, the set of additional procedures and objects that extend Pascal into "graphics Pascal" could be combined with the set that extend Pascal into "music Pascal"; and the net result would be a still-more-extended language that has capabilities for both graphics and music. We will return to this issue at greater length in the final section of this paper, in the discussion of domain-specific libraries; but for now, the point is simply that by using a standard base language, the common problem of "program integration" between separate applications is alleviated if not eliminated entirely. (In contrast, special effort must be expended to permit, e.g., a database language to be called from within some other application.)

The use of a general programming language (as opposed to an *ad hoc* language) as the basis for an application-specific dialect can also benefit from the continuing efforts of a large, long-term community of programmer/users. For instance, despite the criticism of general programming environments earlier, it is nevertheless true that considerable work has gone into the design of these environments (in the form of structure editors, debugging systems, incremental compilation, and so forth). Much of this effort can be appropriated into application programming environments as well. Moreover, in contrast to *ad hoc* languages, most existing general languages have a formalized semantics and an agreed-upon standard (at least for some "core" version of the language). This means that users of (say) a "graphics Pascal" can communicate the essential meanings of their programs even to Pascal users who have never worked with the same particular graphics application. (Compare the situation of *Mathematica* users, whose programs cannot be understood by anyone who has not worked with that one particular application.)

To sum up this topic, then: a strong case can be made for incorporating a well-supported language into a newly-designed application, as opposed to a unique *ad hoc* language. Even so, in the final analysis, this decision is far less important than the decision to allow programmability altogether. From the user's standpoint, the expressive power gained by using just about *any* language will outweigh the possible disadvantages of working with the "wrong" language. It should also be mentioned that—quite possibly—*multiple* languages could be incorporated into application interfaces. A graphics program (say) might be accompanied by programming environments for "graphics Pascal," "graphics Scheme," "graphics Basic," or whatever language the user finds congenial.[2] Again, though religious arguments might rage about the advantages of one language versus another, the key issue is the availability of *some* language.

4.2 Embedding a Domain-Specific Language Within a General Language

Throughout this paper so far, in describing domain-specific programming, the notion of an "enriched language" or "enhanced language" has come up. It is worth at least a brief digression at this juncture to elaborate on this notion.

[2]This list might include, of course, some new *ad hoc* language as well.

As a starting point, we might consider a typical programming environment for some language—for the sake of specificity, we can use Scheme as an example. A Scheme system must by definition contain a certain "core portion" of the language: perhaps a dozen special forms (listed under "essential syntax" in the Revised[3] Report on Scheme {34}), a hundred or so primitive procedures, and a variety of foundational object types (numbers, strings, symbols, pairs, procedures, vectors, and so on). Virtually every Scheme system contains much more than this central core: PC Scheme contains "window objects," MacScheme has "bytevector objects," MIT Scheme has "error condition objects." In every case, these object types (and the procedures and special forms associated with them) are not required by the Scheme standard, but they add to the range of expression of the programmer; and because these enhancements are embedded within the overall Scheme system, they obey the usual conventions of Scheme syntax. (For instance, one can write procedures in PC Scheme that take "window objects" as arguments, using the same syntax that would be employed for writing numeric procedures.)

In a sense, then, these Scheme implementations already come prepackaged with their own "miniature embedded languages" dealing with notions such as windows; that is, the core language has been enhanced with a number of new object types and primitive procedures designed for working with some domain not covered by the bare standard of the Scheme language. Admittedly, these particular "embedded languages" are rather skeletal, and deal with circumscribed domains mainly of interest to programmers; but they illustrate in principle how "enhanced languages" can be constructed.

Just to pursue a slightly more elaborate example; suppose we would like to add, as part of a larger graphics package, a collection of color-related procedures to Scheme. The key questions in doing so would be to note

- What new kinds of objects have to be added to the language, and
- What new kinds of object-specific procedures and special forms have to be added to the language.

To start with, we might choose to construct colors (in accordance with many graphics systems) out of three floating-point values ranging from 0 to 1 and standing for the red, green, and blue components of the color (with 1 corresponding to the maximal amount of some component, and 0 the mini-

mum). Thus, we start with a new Scheme constructor:

make-color-object *red green blue* procedure
This procedure takes three numeric arguments which should range between 0 and 1. It returns a new color object constructed from these values. For example, the expression

```
(make-color-object 0. 0. 1.)
```

returns a color-object corresponding to the color blue.

In addition to our new constructor, we create selectors for the color components, and an object-type predicate:

get-color-object-red *color-object* procedure
get-color-object-green *color-object* procedure
get-color-object-blue *color-object* procedure
These three procedures each take a color object as argument and return the appropriate component value (expressed as a number between 0 and 1) as result.

color-object? *object* procedure
This predicate returns *true* if the object is of type color-object, and *false* otherwise.

Finally, we would presumably wish to include a procedure that will set the current graphics "pen color" and "background color" to some particular value:

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.

These procedures constitute the bare bones of a "color-manipulation language" that could now be employed within our graphics package. For instance, we now have sufficient means to write procedures that vary colors

at random, that produce colors by combining other colors, that check for "closeness" between colors, and so forth. Very likely, some of these procedures would themselves be supplied as additional primitives in our color package (rather than assuming that the user should write them); but the rudimentary set presented here at least lays the conceptual groundwork for a more extensive language. As an example—just to indicate how we might create higher-level constructs—we present a Scheme procedure that averages two colors:

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

Now we could use this new procedure to construct new colors from existing ones. In the following expressions, we first construct "basic" red and blue color objects, and then use these to construct other shades:

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

(define blue (make-color-object 0 0 1))

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

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

This still-very-brief example should provide some illustration of the way in which a few new basic elements, once embedded within a larger language, can be used to build "compound thoughts" from simpler ones. We now have the ability to interpolate between colors, and could consider, e.g., writing a procedure that paints a region with colors smoothly varying from blue (at left) to red (at right). In other words, "interpolating between colors" has become an operation that we can work with, even though it was not *explicitly* contained in our original set of objects and procedures.

In microcosm, this example also illustrates a principle of software design articulated by Abelson and Sussman in their book *Structure and Interpre-*

tation of Computer Programs. These authors write of program-building as an effort of language construction: i.e., rather than writing software whose main focus is the decomposition of problems into subproblems and sub-subproblems and so forth, we can instead write software that provides us with a language in which the entire problem domain can be expressed. As an example, Abelson and Sussman present a digital circuit-simulation language (embedded in Scheme) in which the primitive objects consist of wires; there are procedures that act like AND/OR gates combining these wires; and there are a few additional procedures that examine the states (logical 1 or 0) of wires, that propagate wire values through a circuit, and so forth. Again, because the system is written as a language, with its own primitives and means of combination, we can build greater complexity from the original foundation: we can construct flip-flops, counters, full-adders, and all sorts of digital devices, and—more importantly—these new devices now become part of our conceptual vocabulary. That is, having built a procedure that constructs flip-flops, we can now treat that procedure as a new primitive with which to express still more complex ideas (e.g., we could now create a procedure that creates shift-registers by combining together a series of flip-flops).

There are still many issues left unresolved in this discussion—indeed, in Abelson and Sussman's treatment as well. For instance, we might ask if there are broad design principles in the embedded-language-construction process: given that we wish to construct, say, a "music Scheme," how do we start? We might ask if there are methods for comparing two alternative sets of language primitives—whether it is possible, e.g., to construct a "better" (more learnable, more compact, more elegant) set of primitives for color manipulation than those suggested above. Or again, coming back to the main issue of this paper, we might ask if there are strategies for integrating such constructed languages with powerful interfaces—for instance, we might want to combine the digital simulation language with an interface that allows us to construct circuits by dragging gate-icons about on the screen.

We will return to these issues in subsequent sections of this paper; but, as a final note, it is worth mentioning that the "embedded language" idea has been used with notable success in at least one instance—the Logo language. Logo was originally devised as a dialect of Lisp with a learnable syntax suitable for children. As such, it is a perfectly good general-purpose language. In

practice, however, the overwhelmingly popular aspect of Logo is not so much its syntax or Lisp-like origins, but rather its characteristic association with turtle graphics. It is this embedded "turtle language" within Logo that has helped the language truly make its mark; one can no more think of a "Logo system without a turtle" than of a Lisp system without lists. Thus, Logo is in many ways a prototype of the domain-specific embedded-language idea: it is a general-purpose programming environment enriched with a collection of objects and procedures around some particular domain. What is missing in a pure Logo environment, arguably, are the direct manipulation interface features that would make it a much more powerful medium for graphics of all kinds; but on the linguistic side, Logo provides a good working example of the "enhanced language" concept. We will come back to this issue in Section 5 of this paper.

4.3 Interface-Level Programming versus Application-Level Programming

The previous discussion was devoted to elaborating upon the notion of developing a "domain-enriched" version of an established general-purpose language such as Scheme. It is worth contrasting the activity of programming in this type of domain-specific dialect with a different sort of programming—what might be called "interface-level programming." The distinction has its grey areas; but since this issue often arises in talking about programmable applications, we might as well face up to the topic now.

Interface-level programming is what users do when they are allowed "under the hood" of an application—most likely by obtaining the source code for the application. A user who does interface-level programming can go mucking about in the code of the application itself: he may change the window arrangement, or the appearance of icons, or the default meanings of particular keystrokes. This type of activity is often associated with the term "tailorability": an application is said to be tailorable if it allows the user to alter the presentation or behavior of that application.

Tailorability is certainly desirable in applications, and reflects a laudably democratic spirit on the designer's part. By permitting users to indulge in interface-level programming, tailorable applications encourage the user to think about and participate in the design issues that underlie the construction

of the application.

That said, tailorability is oversold as a design philosophy. Consider once more the examples of frustrating direct manipulation interfaces that we listed earlier: we wanted the paint program to include Lissajous figures, and the observatory program to include eclipse-searches, and the video program to allow repetitive playing of video-clips. These aren't presentational issues to be solved by allowing the user under the hood of the original applications: we don't want to add a "Lissajous figure" choice to the paint program menu, even if such an alteration were straightforward. The problem was that the application was missing a language in which Lissajous figures, and billions of other ideas, could be realized as they were needed.

To put it another way: users, by and large, don't need or want to confront the problems of a designer. They do not want to deal with interface issues, but rather with domain issues. A student working with the observatory program does not care about what fonts and icons and window arrangements the program uses; he cares about astronomy. Thus he needs a language in which to talk about astronomy, as opposed to talking about astronomy programs. The objects of interest are "star objects," "planet objects," values for latitude and longitude, and so on; and it is the ability to write powerful programs with these objects that constitutes real progress for the user.

Of course, there is no reason that interface-level and application-level programming can't coexist: we should allow users under the hood whenever they want to go there. But, as software designers, we flatter ourselves to think that users are interested in our job. Languages are best used not to tailor applications but to tailor the way that we conceptualize a domain.

4.4 Where Does the User Begin?

Suppose, as a consequence of frustration, that a user actually goes out to the computer store and purchases his first programmable application: a "graphics Scheme" system perhaps similar to the one described later in this paper. Could any user—particularly one new to programming—possibly take advantage of the power of such an application? How would he begin?

This is an especially thorny question. Conventional wisdom in much of

the computer industry is that the "average user" does not want to deal with a programming language under any circumstances. Any user foolish enough to purchase a "graphics Scheme" would never bother to learn programming, according to this view; the overhead in time and mental effort is simply too great.

Conventional wisdom has been wrong before and may be wrong now; only actual case histories, based on experience, will eventually decide this particular issue. Nevertheless, if programmable applications are to attract new users, they had better be structured in such a way as to make the entry into programming as gentle and useful as possible. Users may well *begin* their work with the application by focusing all their attention on the interface; but at some point they will wish to put their toe into the water of programming, and when that occurs the system should encourage their efforts.

There are several "learnability-enhancement" strategies that should all be investigated in concert in the design of programmable applications. First, sample programs with extensive documentation (and with suggestions for experimental changes) might be provided along with the application; whenever the user wished to, he could load and peruse these educational files. Thus, our hypothetical graphics-Scheme user might begin his foray into programming with the kind of simple turtle programming that enchants many elementary-school-age Logo users.[3]

Second, there could be "learning-mode" programs that, when run, actually step the user through the creation of code; this type of embedded tutoring would follow in the tradition of programming tutors created in the past for Logo {30}, Basic {7}, Pascal {10, 23}, and Lisp {6}, among others. Unlike most programming tutors, however, this type of system would focus not on abstract problems to teach language syntax, but rather would concentrate on tasks that are centered in the application domain. Thus, the graphics-Scheme user might learn recursion in the context of creating a spiral—a graphics task that is both interesting and yet at the same time unachievable in the interface alone.

Finally, it is conceivable that sequences of interface actions could be trans-

[3]Compare the discussion of "programming by modification" in Lewis and Olson {26}.

lated into readable (and editable) programs. Thus, the graphics-Scheme user might draw straight lines on the screen, and the program could respond by saving sequences of Scheme expressions into a file:

```
(show (make-line (make-point 15 20) (make-point 18 40)))
(show (make-line (make-point 18 40) (make-point 50 40)))
```

and so forth. By now editing this code, naming it, parametrizing it, the user could begin to get some feeling for the flexibility of programming. This kind of "save-a-program" feature could be associated with particularly simple operations (to avoid the difficulty of trying to represent *every* interface operation in some canonical textual form). It might also constitute an element of the programming tutor mentioned above.

4.5 Other Kinds of "Direct Manipulation"

Earlier in this paper, we noted that "direct manipulation" is a broad concept, and that attempting a formal definition of it is tangential to our main argument. For our purposes, a "mouse-based" interpretation of direct manipulation will suffice—i.e., we will identify the concept with the "point-and-click" interfaces typical of Macintosh applications.

Nevertheless, it is worth leaving a reminder to ourselves that mice are not the last frontier in human-computer interfaces. Wondrous innovations—"DataGloves," tactile-feedback devices, 3-D viewing devices—are on the near horizon.{19, 27} It is important to realize in this regard that direct-manipulation interfaces based on these marvelous inventions will be every bit as frustrating as their mouse-based ancestors, as long as the interfaces are not combined with the expressiveness of programming. Merely substituting a DataGlove for a mouse will not cure the observatory program of its inability to represent new concepts, nor will a 3-D viewer add spirals (or perhaps helices) to a paint program. To be sure, these devices are great to have, and they offer new ranges of expression to the computer user; but they do so primarily by expanding only the "extra-linguistic" side of applications. The need for language remains untouched—indeed the potential for creative collaboration between language and interface becomes even more complex and exciting as the tools for interface construction evolve.

5. SchemePaint: a Programmable Application

In this section, opinionated generalities are finally dispensed with, and we present an actual working prototype for a programmable application. "SchemePaint" is a program that combines the essential core of a direct-manipulation paint system (similar in spirit to MacPaint {S7}) with an interpreter for a graphics-enriched Scheme (similar in spirit to Logo). The program allows users to create hand-drawn (or perhaps more accurately, mouse-drawn) pictures and to combine these pictures with complex computer-drawn graphics. SchemePaint also includes a few embryonic (but interesting) attempts at integrating its direct manipulation and interactive language "personalities." At present the program runs on Macintosh computers with color graphics (such as the Macintosh IIci).

5.1 The SchemePaint Interface

The current SchemePaint interface consists of three windows: a graphics window (in which drawings will be created), a "palette" window (which is used to alter properties like pen-color interactively), and a Scheme interpreter window. The first two of these windows may be thought of loosely as the "direct manipulation" side of the program, and the third as the "language" side. In addition, there are seven menus presented by the program: the first four of these are general Scheme-system menus[4], and the last three are SchemePaint-specific. Figure 1 shows the initial SchemePaint screen.

In getting started with the program, SchemePaint may be thought of as a small (and rather spare) direct-manipulation paint program. By selecting a pen-size and pen-color (from the palette window), the user can draw lines directly within the graphics window. Figure 2 depicts a hand-drawn line created with this portion of the SchemePaint interface.

The palette window enables the user to select among twelve suggested colors and three pen-sizes. It also includes selections for filling screen regions (with a given color), for using the pen as an "eraser," for typing text within the graphics window, and for dragging the Logo turtle (to be described

[4]The program was written in MacScheme, published by Lightship Software {S8}; and these first four menus are provided with the language system itself.

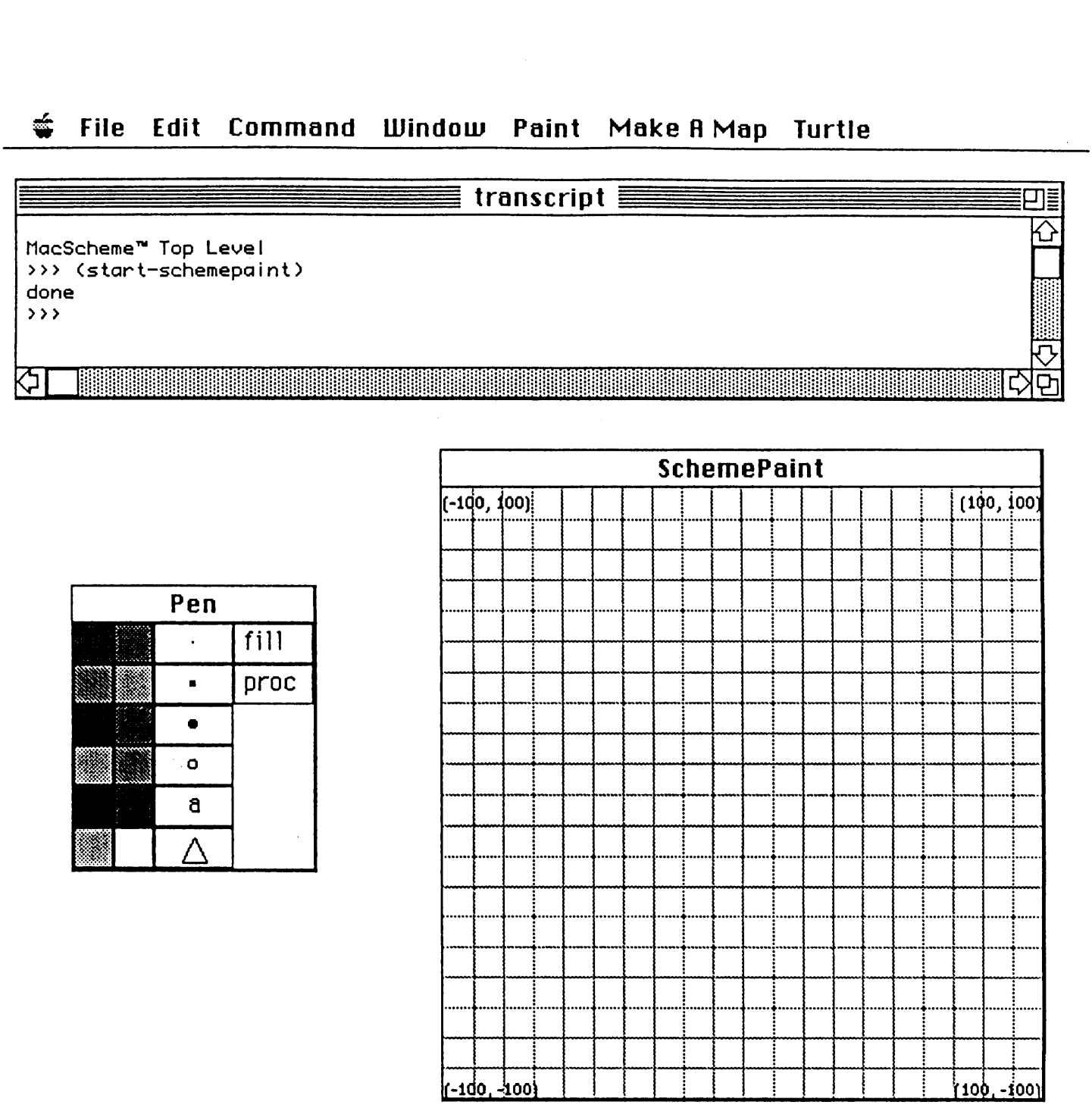


Figure 1: The initial SchemePaint screen. The interpreter window is labelled "transcript," and the graphics window is labelled "SchemePaint."

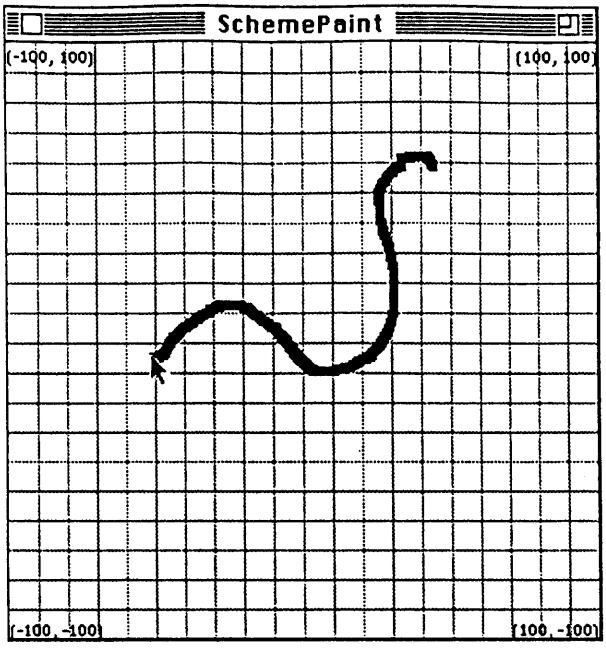


Figure 2: Drawing a line by hand in the graphics window.

shortly). Menu options allow the user to create a variety of "standard" shapes with the mouse; these shapes include circles, straight lines, rectangles, and arbitrary polygons.

5.2 SchemePaint's Embedded Graphics Language

Besides its direct-manipulation features, SchemePaint includes a Scheme interpreter and an "enhanced" Scheme language with an extensive set of graphics primitives. These primitives may be classified into several major categories:

- Turtle graphics
- Planar maps (and two-dimensional dynamical systems)
- Color-related features
- Miscellaneous features

▷ *5.2.1 Turtle Graphics*

SchemePaint's turtle-graphics-related primitives are the same core set that accompany most Logo interpreters. There are procedures for moving and turning the turtle (including `fd`, `rt`, `setpos`, `home`, `seth`), for changing turtle color and pen-state (including `set-pen-color!`, `pd`, and `hide-turtle`), and for examining the turtle's state (including `penup?` and `getpos`). Addition-

ally, the "turtle mode" selection on the palette window—mentioned a bit earlier—allows the user to drag the turtle about on the screen, and to turn the turtle. Thus, it is possible to first write an `octagon` procedure for the turtle:

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

and then to place an octagon at some desired screen location by dragging the turtle to an appropriate starting point and evaluating (e.g.)

```
(octagon 25)
```

in the Scheme interpreter.

Because SchemePaint is embedded in Scheme, it can take advantage of some of the elegant semantic features found in Scheme—notably the use of procedures as first-class objects. As a very brief example, one could edit the `octagon` procedure above so that it takes a "turtle-move" argument as well as a "side-length" argument:

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

Using the new version of `octagon` it is easy to express a "standard" octagon:

```
(octagon fd 25)
```

In addition, one can economically express any number of "octagon variations":

```
(define (zig side)
  (fd side) (rt 144) (fd (/ side 2)) (lt 144) (fd side))

(repeat 8 (octagon zig 20) (rt 45))
```

The picture produced by this last expression is shown in Figure 3.[5]

[5]The "background grid" option has been turned off in this and subsequent figures.

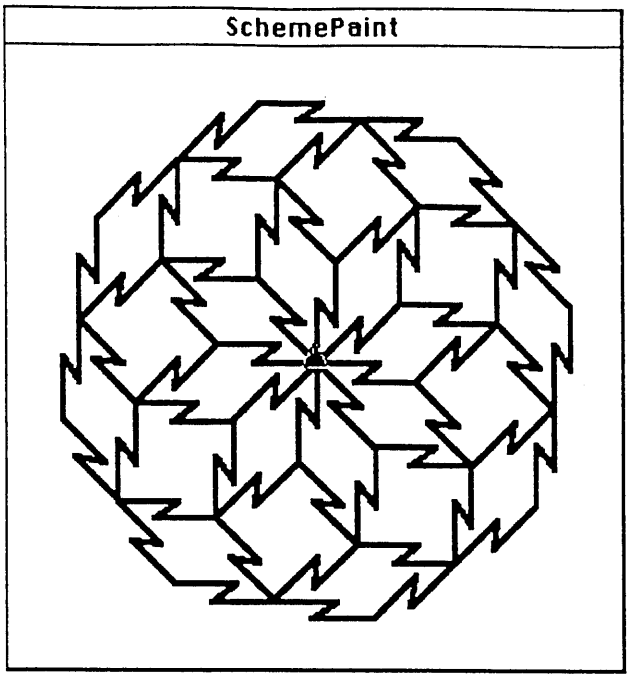


Figure 3: A turtle-drawn figure in SchemePaint.

Finally, SchemePaint's turtle graphics facilities include menu-selectable equivalents for some common turtle commands (including `clearscreen`, `hide-turtle`, `show-turtle`, and `home`).[6]

▷ *5.2.2 2D Maps and Dynamical Systems*

Besides employing turtle graphics, SchemePaint also allows the user to create and display point-, line-, and polygon-objects, and to apply to these objects a variety of planar maps (functions from the plane to the plane). As an example of how these facilities may be used, we can begin by creating a "point object" as follows:

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

Here, we have created a Scheme object representing the point (0, 0); this point may be displayed on the graphics screen using the SchemePaint prim-

[6]The motivation for including these menu selections is not that these particular commands are hard to remember, but rather that they are often used in "direct mode" (i.e., typed directly at the interpreter as opposed to being included in procedures). As a result, they constitute "cliches" whose execution is more easily specified by selection than by the occasionally tedious act of typing.

itive **show**:

```
(show the-origin)
```

When this expression is evaluated, the point (0, 0) is displayed in the current default foreground color on the graphics screen.[7] We now create a simple "translation map"—a map that adds 5 and 6 to the x- and y-coordinates (respectively) of its argument points:

```
(define sample-translation-map
  (make-map (x y)
    (+ x 5)
    (+ y 6)))
```

Our definition employs a SchemePaint primitive named **make-map** that creates a "map object" which can now be applied to a given point (or line, or polygon) to produce a new point (or line, or polygon). The appropriate **apply-map** primitive takes as its arguments a map and an object to which to apply the map:

```
(apply-map sample-translation-map the-origin)
```
(5.0 6.0)

We can display this new point on the screen by using **show**, just as we did with the origin:

```
(show (apply-map sample-translation-map the-origin))
```

And we can go further to show the results of applying our sample map *repeatedly* to the origin by using yet another primitive named **show-n-iterations**. This primitive takes three arguments: a starting point (line, polygon) object; a map to be applied; and the number of successive times that we wish to apply the map. It is used as in the expression below:

```
(show-n-iterations the-origin sample-translation-map 10)
```

By evaluating this expression, we see a succession of 11 points (including the origin itself) representing the results of applying the map 0-10 times. The graphics screen that we get is shown in Figure 4.

[7]The default screen coordinates correspond to the region of the plane from (-100, -100)

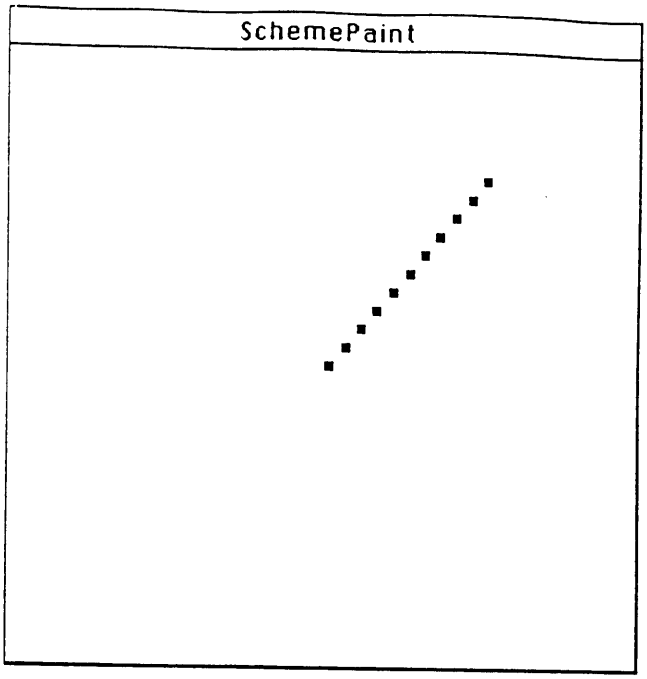


Figure 4: A series of points produced by iterating a translation map.

Again, it is not hard to start building complexity once a powerful set of primitives has been supplied. For example, here is a procedure that takes as its argument a list of map objects, and returns a randomly chosen map from that list:[8]

```
(define (select-random-map list-of-maps)
  (list-ref list-of-maps (random (length list-of-maps))))
```

If we supply a list of map objects to our new primitive, we will get back one of those maps, chosen at random. We can now write a procedure—analogous to the `show-n-iterations` primitive seen a moment ago—that takes a starting point, a list of maps, and a number n of iterations to use, and repeatedly applies *some* map (chosen from the list), n times, starting with the given starting point:

---

at the lower left corner to (100, 100) at the upper right. This arrangement may be altered by using a primitive named `reset-canvas-coordinates!`.

[8]This procedure employs the common Scheme primitive procedures `random` and `list-ref`. Again, because SchemePaint is embedded in Scheme, we can make free use of all the standard Scheme features.

```
(define (show-n-random-selections list-of-maps starting-point n)
  (define (loop ct this-pt)
    (cond ((> ct n) 'done)
          (else (let ((new-pt
                       (apply-map (select-random-map list-of-maps)
                                  this-pt)))
                  (show new-pt)
                  (loop (+ ct 1) new-pt)))))
  (loop 0 starting-point))
```

The `show-n-random-selections` procedure may be used as the basis for creating a wide variety of fractal shapes on the graphics screen. For instance, the following expression, when evaluated, will produce a picture of the famous Sierpinski triangle:

```
(show-n-random-selections
  (list
    (make-map (x y) (+ 40 (* x 0.5)) (* y 0.5))
    (make-map (x y) (* x 0.5) (+ 40 (* y 0.5)))
    (make-map (x y) (* x 0.5) (* y 0.5)))
  the-origin
  2000)
```

The resulting picture is shown in Figure 5.

This necessarily brief discussion can only provide a hint of the kind of experiments that one can perform with SchemePaint's dynamical systems procedures. In addition to those mentioned in passing here, there are primitives for composing maps (to create new, more complicated maps); for creating line and polygon objects; for resetting the plane coordinates corresponding to the graphics window; and for integrating two-dimensional systems of differential equations.[9] Some pictures produced with SchemePaint's dynamical systems procedures are shown in Figures 6-8, and in Color Plates 1 and 2.

[9] A paper devoted exclusively to SchemePaint and its capabilities is currently in preparation.

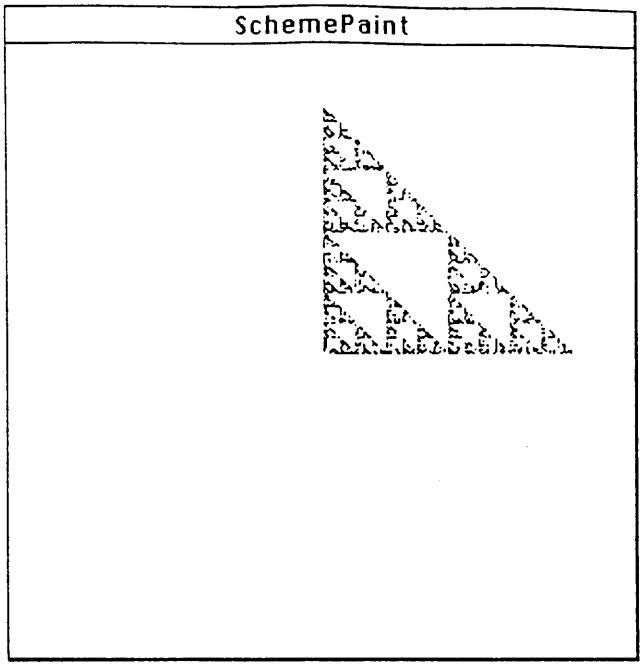


Figure 5: The Sierpinski triangle, produced by an iterative process of random selection among three affine maps.

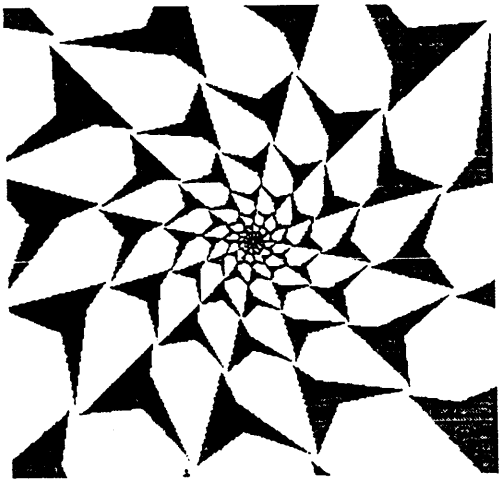


Figure 6: Iteratively applying a rotate-and-scale map to a polygon.

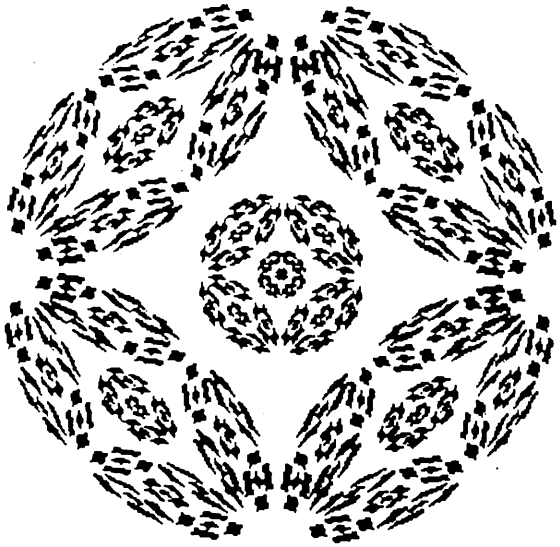


Figure 7: A fractal produced by iteration of a "superposed affine" map.

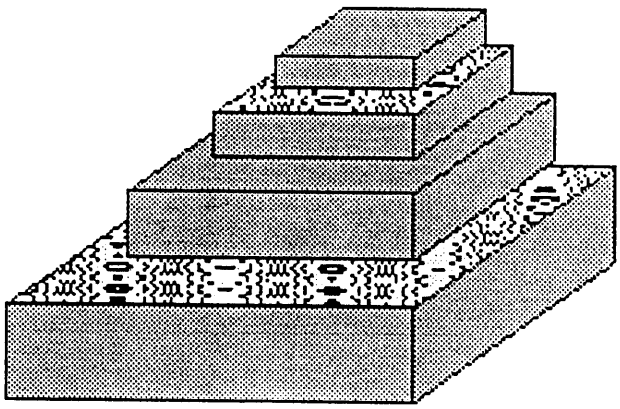


Figure 8: A design produced by iterating a scale-and-translate map. The pattern visible on two of the surfaces was done using the "programmable coloring" feature described in Section 5.3.1.

▷ *5.2.3 Color-Related Procedures*

SchemePaint's color procedures are similar to those described earlier in this paper. Color objects are composed of three real numbers (in the range 0-1) corresponding to the R, G, and B components of the desired color.[10] There are primitive procedures for setting the current pen-color and for setting the background color; for extracting the components of color objects; and for creating color objects from numeric components. Other primitives allow for interpolation between colors (as suggested in the earlier discussion), for filling screen regions with colors, and for retrieving the color of individual pixels on the screen.

Just as an example of how these color procedures may be used, we present a procedure that takes as its argument a color-object and alters that color by randomly shifting its red component up or down by 5 percent:

```
(define (shift-color-red-component color-object)
  (let ((red-pct (* 0.05 (get-color-object-red color-object)))
        (plus-or-minus (if (= (random 2) 0) 1 -1)))
    (make-schemepaint-color-object
      (with-lower-upper-limits
        0. 1. (+ (get-color-object-red color-object)
                 (* red-pct plus-or-minus)))
      (get-color-object-green color-object)
      (get-color-object-blue color-object))))
```

Naturally it would not be hard to extend this idea to represent the notion of shifting any component of a color object by any desired percentage; and we could then vary that idea to express the notion of choosing a color at random from a region of R,G,B-space; and we could vary *that* notion to write a procedure that chooses randomly between a discrete set of colors.

SchemePaint's color-related procedures may be used in conjunction with both the turtle graphics primitives and with the dynamical systems primitives mentioned earlier. Thus, evaluating a call to `set-pen-color!` will reset the

[10]Of course, the actual number of distinct representable colors may be much smaller than implied by this data structure—eight- and sixteen-color systems are not uncommon. The number of representable colors depends on the graphics hardware of the individual machine.

turtle's pen color or the (default) color in which points, lines, and polygons are shown on the screen.

▷ *5.2.4 Miscellaneous Features*

In addition to the features mentioned thus far, SchemePaint includes several other capabilities in its "enriched language" component. Text may be displayed on the graphics screen using a `draw-string` primitive (which takes a string and cursor position as arguments); this complements the direct-manipulation "text mode" of the program. SchemePaint likewise includes primitives for copying rectangular regions of the graphics window, for saving graphics screens in files, and for restoring graphics screens from saved files.

It may also be worth noting that SchemePaint makes free use of all the standard facilities of the MacScheme language environment, including its debugger, editor, byte code compiler, and file system.{S8}

▷ *5.2.5 SchemePaint Pictures*

Because SchemePaint allows the user to produce both hand-drawn and machine-drawn figures, the pictures that one can create with the program are much richer in scope than those created by either a "pure programming" or "pure interface" environment. Color Plate 3 is an illustration of the idea: it depicts a bee in a honeycomb.[11] The bee figure would be difficult to draw without a direct-manipulation interface (e.g., by writing a Logo program), while the honeycomb would be tedious (though admittedly not impossible) to create with a program such as MacPaint.

Color Plates 4-8 illustrate the same basic idea—combining hand-drawn and code-drawn figures—in a variety of ways. Color Plate 4 depicts a hand-drawn snake slithering through a Logo-style "rotated octagon" figure. The expression that generated this figure (using the earlier `octagon` procedure) is as follows:

```
(repeat 8 (octagon 30) (rt 45))
```

[11]The artwork for Color Plates 4-8 is by Orca Starbuck; this note is a special thanks for her generous and talented assistance. On the other hand, the author must be blamed for the artwork in Color Plate 3.

Those who insist on the conceptual difficulty of programming might be advised that this is the sort of "rotated polygon" figure routinely generated by moderately experienced fourth grade Logo programmers in classrooms across the country.

Color Plates 5, 6, and 7 include Logo-style figures found in the book *Turtle Geometry* by Abelson and diSessa.{1} Plate 5 includes recursively-drawn tree figures; Plate 6, the well-known fractal shape dubbed the "dragon curve"; and Plate 7, a figure generated by what Abelson and diSessa call an "inspi" procedure (an easy variant of the standard Logo spiral-drawing procedure). Finally, Plate 8 uses yet another recursive procedure, much like the tree procedure of Plate 5—except this one is used to make peacock feathers. In every case, the turtle-drawn figures may be combined freely and expressively with hand-drawn figures, thus providing the artist/user with the best of both worlds.

5.3 Further Directions in Interface/Language Cooperation

Thus far, our description of SchemePaint has focused on the interface and language aspects of the program in isolation from one another. Of course both portions of the program do interact with the graphics screen, and in that respect they may be said to "cooperate"; and moreover, there are numerous cases in which either language or interface may be used to effect some operation. (For example, the pen color may be changed either by direct selection from the palette shown on the screen or by evaluating a `set-pen-color!` call.) It must be admitted, though, that these cooperative strategies are rather straightforward. In this section we describe some additional features of SchemePaint—features intended as experiments in developing more symbiotic relations between interface and interpreter.

▷ *5.3.1 Programmable Colors*

The first of these special features —"programmable coloring"— cannot be found (to my knowledge) in any currently available graphics application. It is, however, a genuinely useful addition to the the graphical artist's repertoire. The idea is that the user can first write any function from x- and y-coordinates to colors, and can then dictate that the mouse must use this

function as its effective "pen-color." (Another way of phrasing this is to say that the pen-color depends on x and y.)

An example may help to explain this notion. Suppose we would like the pen-color to be red for all x-values less than 0 and green otherwise.[12] We can write such a function as follows:

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

We can now cause this function to be the effective pen-color by performing two operations. First, we evaluate a call to the special SchemePaint primitive `set-mouse-function!`:

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

And now, by selecting a special palette box labelled "*proc*" we can tell the mouse to use as its pen-color the very function that we just passed as argument to `set-mouse-function!`.[13] Now, when we drag the mouse across the screen, we will see a red line for positions at which the x-coordinate is less than 0, and a green line otherwise. Using the default coordinates, we can see the mouse change color as we drag it from the left half of the screen to the right.

Programmable colors allow the user to create a wide variety of enjoyable effects. We can create "noisy" colors (with a bit of randomness thrown in), colors that reflect some numerical function of position, colors that can only appear in certain regions of the screen. Here is an example: this function produces a color of red near the origin, blue at any distance more than 100 units from the origin, and "intermediate" values at distances of less than 100 from the origin:

[12]Recall that the default screen coordinates range from (-100, -100) in the lower left corner to (100, 100) in the upper right. Using these default coordinates, then, the suggested function would correspond to a red pen-color anywhere in the left half of the graphics screen and a green color in the right half.

[13]In general, selecting "proc" causes the mouse to use as its pen-color the function most recently passed as argument to `set-mouse-function!`. If we wish to change the mouse back to a "normal" coloring mode, we simply select one of the other color choices on the palette.

```
(define (red-near-origin-blue-far-away x y)
  (let ((distance-from-origin
         (sqrt (+ (square x) (square y)))))
    (if (> distance-from-origin 100)
        blue
        (interpolate-between-colors
          red blue (/ distance-from-origin 100.)))))
```

Here is another example: this function produces vertical black and red stripes of width 10 on the screen.

```
(define (vertical-stripes x y)
  (if (< (modulo (round x) 20) 10)
      black
      red))
```

SchemePaint's `set-mouse-function!` primitive is in fact a special case of a more general SchemePaint function named `set-mouse-drag-procedures!`. This latter function allows the user to specify the behavior of the mouse in "drag mode" (i.e., when the desired functionality involves holding the mouse button down and dragging the cursor across the graphics window). Here, the mouse behavior may be not only defined as a function of the current x- and y-coordinates of the mouse cursor, but also of the previous x- and y-coordinates[14], the color of the pixel at which the mouse is now (and was previously) located, and the time since the dragging operation began. The range of specifiable behavior using this general function is thus a superset of the (already wide) range expressible via `set-mouse-function!`.

▷ *5.3.2 Naming Hand-Drawn Objects*

The just-described "programmable colors" facility was one in which SchemePaint's language subsystem was used to communicate "instructions" to the interface subsystem. It is possible to work the other way round—to have interface operations send information back to the language subsystem. An example of this type of facility is provided by SchemePaint's use of certain

[14]That is, the coordinates for the last point at which the mouse was located during the dragging operation; this facility allows for the speed of mouse-movement to be taken into account in defining mouse behavior.

default names for several classes of objects (points, lines, rectangles, and polygons) created using the mouse.

As an example, suppose that we wish to draw a polygon "directly" on the screen, by selecting the proposed vertices one by one with the mouse; this is the standard method for creating polygons in most direct-manipulation paint programs, and is permitted in SchemePaint as well. To perform this operation in SchemePaint, we choose a "draw object" menu selection, and then (from within a dialog box) choose the type of object that we wish to draw—in this case, a polygon. Once the polygon has been drawn, it appears on the screen; and a corresponding polygon object is now the value of the Scheme name `*last-polygon-created*`. Thus, if we evaluate the name `*last-polygon-created*`, we now have a data structure representing the object just made via direct manipulation.

As in the case of programmable colors, this linkage between interface and language gives us a great deal of power and flexibility. For instance, suppose that we had earlier written a procedure to find the geometric center of any given polygon:[15]

```
(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)))))
```

We could now use this procedure to find the geometric center of our newly-drawn shape:

```
(find-geometric-center *last-polygon-created*)
```

Evaluating this expression returns a point object representing the center of our polygon. We could now plot this point (note that we do not need a special "show geometric center" menu option); or we could use this information to plot lines from the geometric center to each vertex of the polygon, as follows:

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

```
(for-each
  (lambda (vertex)
    (show (make-line
            (find-geometric-center *last-polygon-created*)
            vertex)))
  (polygon-vertices *last-polygon-created*))
```

Going further, we could now display new copies of our polygon at a variety of centers (for instance, we could plot a lattice-pattern of polygons); or we could plot a smaller copy of the original polygon centered at each vertex of itself.

Using the mouse to create a new polygon at this juncture would alter the binding for the name `*last-polygon-created*`; thus, if we wish to hold onto the data structure for some polygon that we have drawn, we need merely bind some other name to it before creating a new one:

```
(define my-first-polygon
  *last-polygon-created*)
```

Evaluating this expression will bind the name `my-first-polygon` to the newly-created polygon; now, if we use the mouse to create a new shape, the name `*last-polygon-created*` will be re-bound, but the name `my-first-polygon` will still be bound to the earlier data structure.

It should also be mentioned that there are analogous special names `*last-point-created*`, `*last-line-created*`, and `*last-rectangle-created*` appropriate to other types of objects created using the mouse.

▷ *5.3.3 Escher Kit*

SchemePaint has been designed as a rather general-purpose graphics program; but people often use graphics applications with some special purpose (or narrow range of purposes) in mind. For instance, they may wish to use their just-purchased paint application to make geometric diagrams; or architectural drawings; or circuit layouts; or political cartoons. In such a circumstance, it is not unusual for users to become so frustrated with the inexpressiveness of their graphics application that they will go out to software stores hunting for a "geometric-design" (or other special-purpose) graphics package.

We have already argued that programmable applications are far more powerful and expressive than "pure interface" applications; it it thus much more probable that they will satisfy the needs of a new user with some special range of tasks in mind. Moreover, in many cases, programmable applications lend themselves especially well to "specification" for some range of tasks: the application need only be augmented with an additional loadable "library" of special-purpose primitives and objects.

We will return to this subject at length in the final section of this paper; but as prelude to that discussion, it is worth presenting an example of a "special-purpose library" that may be loaded into SchemePaint. This is a library of procedures for drawing recursive tiling designs (like those made famous by the late Dutch artist M. C. Escher).[16]

Imagine, then, that we happen to be interested in using our SchemePaint application for (above all else) drawing tiling designs. Conceivably, we could write some procedures ourselves for this purpose; but, if we wanted to use an extensive package of tiling-specific procedures, we could alternatively load in a special library of Scheme procedures designed for users with interests like our own. (Presumably, this would be one of many "special-purpose" libraries made available to the SchemePaint user.) In the current SchemePaint version, this can be done via a menu selection[17], or via evaluating the following expression:

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

Having loaded the "Escher library" into SchemePaint, we now have a variety of new objects and primitives to play with.[18] Moreover, we have not merely extended our *language*, but have also extended our *interface*: the Escher library includes a new "Escher palette" window with which we can create pictures to use in tiling designs. The SchemePaint screen now appears as shown in Figure 9 below.

Without going into detail on the functionality of the newly-loaded Escher

[16]These procedures are modelled on the "picture language" invented by Peter Henderson {22} and described in a Scheme implementation in {3}.

[17]The "load file" selection supplied with MacScheme

[18]Indeed, the Escher library may be considered a kind of "second-level language" embedded within the SchemePaint language which is itself embedded within Scheme.

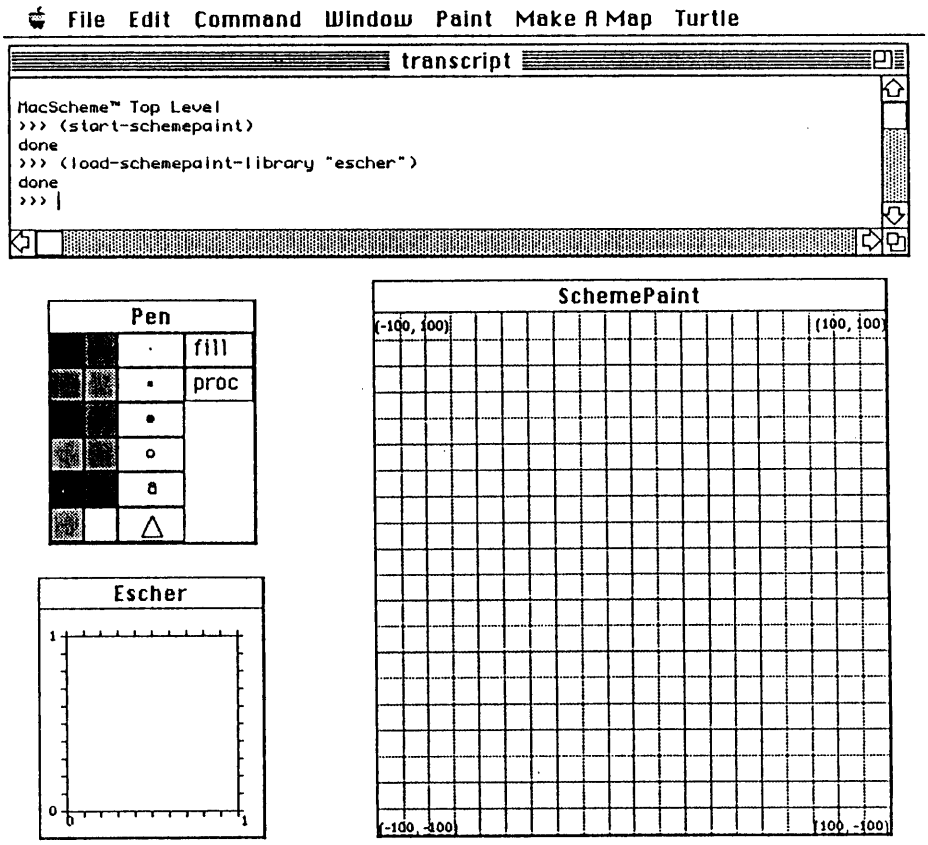


Figure 9: The SchemePaint screen, now including an "Escher palette" window.

library[19], we can now create "primitive pictures" within the new window (or via language), and can combine them together using special "picture-combiners" that create more complex pictures from simpler ones. Using the newly-added functionality, we can create tiling designs like those shown in Figures 10-11 below and in Color Plate 9. Note that we have not lost any of the "basic" functionality of SchemePaint in using our Escher library: Figure 11, for instance, includes a hand-drawn figure accompanying a computer-drawn tiling design.

One of the key aspects to note about this example is that the notion of a "loadable library" applies both to the language and interface subsystems of SchemePaint. Just as the original application is designed so that language and interface work in tandem, so the extensions to the application are likewise designed to augment the capabilities of both portions of the application. We will return to this topic in the final section of this paper.

[19]More discussion is included in a forthcoming paper devoted to SchemePaint.

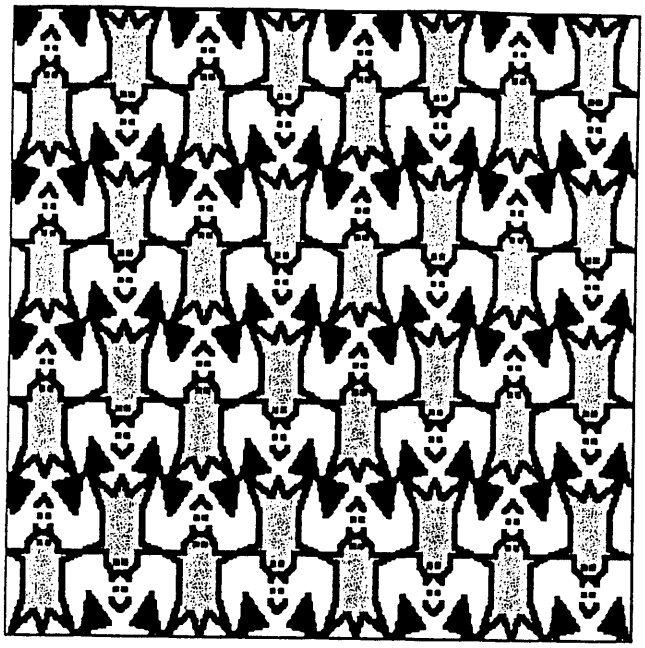


Figure 10: A tiling suggestive of tadpoles and seagulls; created using the "Escher library" of SchemePaint.

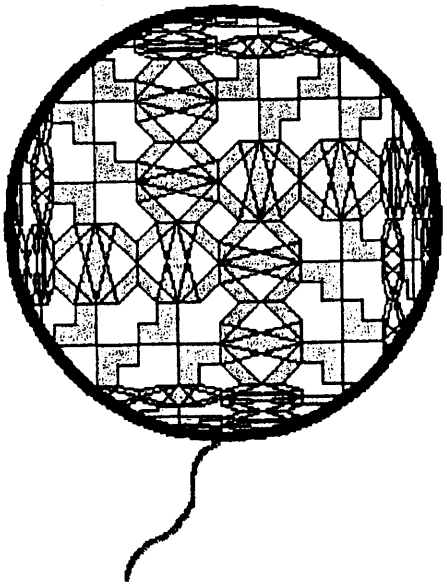


Figure 11: A balloon with an Escher-like decoration.

5.4 Graphics vs. Other Domains

Because SchemePaint is presented here as an example of a programmable application, much of its purpose is to demonstrate the usefulness and feasibility of the idea of programmable applications in general. Naturally, no single example or domain can quite accomplish this task; the notion of programmable applications would be best exemplified by a collection of sample programs in widely varying domains. Such a collection would also illustrate the applicability of the programmable-application idea as a general theory of software design (rather than a one-shot strategy for a particular graphics application).[20]

In some ways, the domain of computer graphics is a good "polemical" choice with which to illustrate the strengths of programmable applications. In particular, it is possible to point to a picture such as Color Plate 4 and to identify visually (and unambiguously) which portions are produced by "interface" and which by "program"; and it is easy to argue further that without *both* these components, an application is inevitably limited in expressiveness.

In other ways, however, "computer art" is not the ideal domain for the purpose. Among other things, some of the functionality of SchemePaint may be mimicked by the commands of a system such as HyperTalk {S5} (which I will later argue is not a particularly good example of a programmable application) or by macros in some of the more elaborate paint applications. This leads some observers to comment of SchemePaint that it is "like HyperTalk but with a more extensive graphics language" or that it is "like SuperPaint macros" or "like Logo with a mouse." Such characterizations are partially (though far from entirely) correct, inasmuch as these various other systems can perform some SchemePaint-like functions.

Comparisons of this sort would be far less likely if the domain of the application in question were music, physics education, video editing, circuit design, or one of the many other possible domains to which the overall programmable-application design strategy could be applied. Again, for this reason, it will eventually be important for there to be a collection of examples, and a body of design lore surrounding them, all supporting the design

[20] We return to this theme in the final section, in the discussion of future work.

strategy argued for in this paper.

6. Objections and Responses

The notion of building programmable applications is not without its skeptics. Over the past several months, I have engaged in many conversations—even debates—about the issues discussed in this paper; and in the course of that time, I've heard a wide variety of thoughtful responses, both positive and negative. In this section, I present—I hope, faithfully—a summary of the most pointed objections to the programmable-application concept, and my own replies to these objections.

▷ *Objection 1:* The programmable-application approach for software design is predicated, first and foremost, on the notion that users want to program. This is visionary. Users—especially the great majority unfamiliar with programming—do not want to be bothered with the difficulty of writing code. Teachers of programming know how hard it is to learn, and how even understanding the simplest recursion is bought at the price of long study; to think that users will desert the simplicity of a direct-manipulation interface in order to spend months learning a new language is unrealistic. Software designers are therefore better advised to come up with newer and more extensive direct-manipulation interfaces, rather than to build programmable applications.

Response 1: If someone really and truly does not want to learn to program, ever, they shouldn't have to. Such a person will have to be content with the inevitably limited (and limiting) tools associated with "pure" direct manipulation interfaces. On the other hand, my own belief is that more people would enjoy learning to write programs than the "interface-alone" camp would have us believe. It is worth reminding ourselves, in this context, that most of the leading thinkers in the computer industry during the early 60's insisted that home computers were an idea without a future, a project bound to fail; the public out there just wouldn't want to deal with such a powerful, complex machine.

There is a chronic tendency among software designers—much like those in the entertainment industry—to underestimate "the public." (Typically,

"the public" is seen as that mass of people who couldn't possibly understand the things that the speaker understands.) Programmable applications should not be written to be obscure, but they don't condescend to their audience either: they assume that the user wants to create things that the designer could never in a million years anticipate. And if, in the future, we look back and say that the designers of programmable applications made the mistake of *overestimating* the public, then so be it: it would be a refreshing mistake to make.

In the same context, it is worth reiterating several points made earlier in this paper. First, a well-designed programmable application should have an extensive direct-manipulation interface, suitable for beginners wishing to "get the feel of" the system. (SchemePaint, for example, is equipped with what one might call a "version 1" MacPaint-like interface.) Thus, even confirmed non-programmers should be able to get some mileage out of these applications. Second, programmable applications should be designed with an eye toward leading the user gently into programming: hence the earlier discussion of embedded tutoring systems, sample "starter programs," and so forth. Along the same lines, it is worth noting that programmable applications would very likely be an excellent venue for teaching programming. One could imagine a computational art course, geared toward artists and structured around an application such as SchemePaint; and similar courses might employ applications for music, robotics, astronomy, whatever. It is perhaps no surprise that very few people *apparently* wish to learn programming when most introductory courses focus on topics such as sorting algorithms; these computer-science-driven examples are remote from the interests and passions of many beginning students. Conceivably, a culture rich in programmable applications would help foster a society rich in programming talent.

Finally, it is again important to stress that users shouldn't need extraordinary or arcane programming skills to make good use of programmable applications: if the application is well-designed, it will be possible to write interesting and useful programs of at most five or six lines. Consider once more the code used to generate Color Plate 4, or the Logo code to generate a spiral: these are simple, elementary-school level programs whose effects seem marvelous by direct-manipulation standards. Learning to write these programs is not a trivial task, but neither does it require deep mathematical

training. Programmable applications, then, should not be expressly geared for experts (though they should reward expertise); rather, by structuring their enriched languages around the key concepts of their domains, they should spark the imagination of the casual programmer.

▷ *Objection 2:* What you are suggesting here is just that people learn Scheme (or maybe "Scheme-with-a-mouse," or "Pascal/Logo/Boxer/Lisp/Basic-with-a-mouse"). Isn't this just adding a bit of frilly interface to a programming environment?

Response 2: Clearly there are overlaps in those features that make for a good programmable application and those features that make for a good general-purpose programming environment; and, as argued earlier, there are strong reasons to equip programmable applications with enriched versions of existing languages (rather than with brand-new special-purpose languages).

On the other hand, what is being proposed here is far more than just tacking a few mouse commands onto Scheme (or some other language). First, in designing a programmable application, the bulk of the effort will probably go into structuring the domain-enriched *extensions* to the language—in essence, the task here is designing new embedded languages within the overall syntactic structure of some existing language. Designing an astronomy application for Scheme is not "just Scheme," nor is a Pascal for video editors "just Pascal"; rather, these are long-term, subtle, and highly creative design projects in their own right. And even the SchemePaint example taken in isolation, including as it does extensions for dynamical systems and interface-related procedures (such as the `set-mouse-function!` procedure mentioned earlier) provides far more than just the standard set of Logo turtle primitives.

Fans of one particular programming language often feel slighted by the implication that not everything need be done in that one language. The programmable application concept, however, makes no *a priori* commitment about the suitability (or preferability) of any one particular language as the "base language" for all examples. As mentioned earlier, it is quite plausible to imagine an application such as SchemePaint in which the user can choose between a selection of different interpreters—in essence, the user could choose to configure the application as "PascalPaint," "LogoPaint," or whatever she

prefers.[21] This is not to say that all languages are equally good at representing all domains—every professional programmer is well aware that this is false.[22] Nevertheless, the question of which language is best is secondary to the question of whether some language (or set of languages) is provided in the first place.

Pursuing this point for a moment: users of multi-language programmable applications might well find themselves making choices among "best" or "primary" languages for given application domains. For database systems, a language based around Prolog might be ideal; for applications involving massive parallelism (such as cellular automata, or simulations of ant colonies), a language like *Logo {35} might provide the foundation of a perfect match; for algebra, a language like ISETL {8} might be preferred. Although these may not be the only languages provided with the given application, they may win over many users precisely because of their felicitous match with the application's domain. And just as novice programmers might learn Scheme through an application such as SchemePaint, so experienced programmers might come to learn a new language like ISETL through its special utility in a particular application.

In any event, the key design questions in creating programmable applications will tend to be somewhat language-independent. The important issue is not what one can achieve *given* that one starts with Scheme/Logo/Pascal, but rather what the artist/musician/astronomer needs in an application. Considerations of this kind might lead the designer toward some choice of primary user language: and if the choice leans toward an "enriched Pascal" rather than an "enriched Scheme," that's the way it goes. The domain, and not the designer's favorite language, should be the driving force behind the design process. Once the designer has created an "enriched Pascal" for some domain, she might then use that experience to guide the creation of an "enriched Scheme," "enriched Logo," and so forth—and even if these other

[21] Note that this question is potentially distinct from the question of how the application is *implemented*. For example, the application could be written in Scheme, but still provide Pascal, Logo, and Basic interpreters as options. This distinction relates back to the question of "tailorability" mentioned earlier; the important language system for the user is the one that focuses on the domain of graphics, not the one in which the overall application happens to be implemented.

[22] Though no two of them agree on what is true.

choices of language are not as good as Pascal for the given situation, they are nevertheless far preferable to having no user language whatever.

▷ *Objection 3:* Couldn't SchemePaint be done in HyperTalk? Isn't what you are proposing just a variation on that system?

Response 3: It is true that HyperTalk is a language-plus-interface programming environment in which applications—specifically, HyperCard applications—may be built; and the HyperTalk language could conceivably be used from within those applications. (In this sense, HyperTalk can act as "implementing language" and as "user's language.") Based on observation, however, it seems overwhelmingly the case that this system is used as a tool for *designers* to work with, rather than as the language base for the *users* of applications. Thus HyperTalk is a marvelous system in which to build thousands of opaque direct-manipulation interfaces, but is not designed (or employed) to extend those interfaces with languages from which the user can benefit. And again, even if it were used for this latter purpose, building (say) an "astronomy-enriched HyperTalk" remains as formidable a design task as building an enriched version of any other language. It can hardly be said, then, that the task of designing programmable applications has been even remotely facilitated (much less "solved") by the existence of HyperTalk.[23]

This is not to say that HyperTalk is not a useful system; but, at the risk of belaboring a point, it is a system geared toward the issues of "tailorability" mentioned earlier. That is, it is a system in which it is easy to talk about interfaces and applications, but *not* about the domains of those applications. As for SchemePaint, it is intended as an example (eventually one among many) of an application in which the user's view of the language is centered on the domain (in this case, graphics) rather than the implementation of the program itself.

[23] A more recent entry in the personal-computer world, Microsoft's Visual Basic system {S10}, is likewise billed as a language-plus-interface environment. Visual Basic, unlike HyperTalk, has the advantage of employing an existing language that many users might already know; thus, users of applications built in Visual Basic might be more inclined to write Basic procedures that work with those applications. My current and perhaps groundless suspicion, however, is that Visual Basic (like HyperTalk) will be used and marketed as a tool for designers rather than as the basis of extension languages for users.

7. Related Work; Future Work; a Research Agenda

In the previous sections of this paper, we have argued the desirability of programmable applications; discussed some of the central issues involved in creating these applications; and illustrated the design principles underlying these applications through SchemePaint, a representative example. In this last section, we begin by discussing some recent work related to the notion of programmable applications (with particular emphasis on work similar to SchemePaint). Then, after examining this work, we sketch a possible framework for an agenda of future design and research.

7.1 Related Work

As noted at various points in this paper, there do exist commercial programs that in one fashion or other illustrate the programmable-application concept. *Mathematica* is one example; although it does not make extensive use of direct manipulation in its interface[24], its domain-specific language suggests the type of power that a programmable application for mathematics can provide. The *Director* animation program, also mentioned earlier, is closer to the programmable-application ideal: it includes both an extensive interface (based on an earlier version of the program named *VideoWorks*) and a language, "Lingo," in which animations may be coded. *Director* also includes extensive programming support, providing (among many other excellent features) a scrolling box in which all the Lingo primitives are listed. Yet another example is the *4th Dimension* database system {S4}, which includes a visual programming language in which database searches and other operations may be written.

In all these cases, the application designers have chosen to include a new domain-specific language—and in every case, the applications do pay some price for this decision (see the discussion in Section 4 earlier). The *Mathematica* language, for instance, is burdened with what to this eye seems a motley syntactic structure—a mixture of Lisp, C, and Prolog styles, all coexisting uneasily. Likewise the Lingo language, not taking advantage of an underlying general-purpose environment, does not include floating point numbers;

[24]For instance, one cannot specify a functional relationship between x and y in *Mathematica* by simply drawing the function on the screen with the mouse.

thus it is difficult to use Lingo to animate physical systems that require more than trivial computations. Quibbles aside, however, these recent programs do represent powerful, creative, and immensely impressive achievements in integrating languages into applications.

Applications that use a well-supported language as the basis of their "programming side" are harder to come by. Perhaps the best-known example, *AutoCAD* {S1}, is one of the most successful applications of all. *AutoCAD*—currently the best selling CAD software package {20}—is an extensive graphics program written for professional architects and designers, and including a huge library of files written in AutoLisp, a "design-enriched" Lisp dialect. Users of *AutoCAD* can perform a variety of graphics operations either via menu choices or through a command language (which can be augmented via AutoLisp procedures).

The choice of Lisp as a base language is a happy one for *AutoCAD*; the application has clearly benefitted from the contribution, by users and third parties, of loadable extension files (this is much like the "library" concept mentioned earlier in the discussion of SchemePaint). On the other hand, *AutoCAD* differs somewhat from the programmable-application "ideal" described here in that it uses its programming language more as the basis of a (remarkably extensive) command language, rather than as the basis of a true programming environment for users. In other words, the application uses Lisp more to assist experienced programmers and designers in providing new interactive commands rather than to lead new and inexperienced users into Lisp programming themselves.

AutoCAD, *Mathematica*, and *Director* are several of the more important current applications that include a high degree of programmability for the user. There are likewise several examples of programming environments which in some sense attempt to integrate programming and interface (or application) construction. Two examples—HyperTalk and (briefly) Visual Basic—were mentioned in the previous section. A third example, the Boxer system, is much closer in spirit to the concerns of this paper.

The Boxer language and environment {13, 14} is one in which many of the ideas of programmable applications are reified. Boxer includes a variety of creative features—some in the language, some in the editor—that allow

interface elements such as menus to arise naturally in the construction of programs. In philosophy as well, Boxer pays particular attention to the value of short, personalized, but nonetheless powerful programs. On the other hand, Boxer is a *general-purpose* programming environment, and as such it does not substantially ease the task of building large domain-specific applications; such applications would have to be written via embedding an enriched dialect within Boxer, as within any other general-purpose environment. Moreover, the Boxer environment is geared toward the use of one particular language—namely, the Boxer language; and conceivably, application designers working in Boxer might run into subtle conflicts with the semantic decisions (e.g., dynamic binding) automatically enforced by that language.

Before leaving this overview of work related to programmable applications, it is worthwhile to note several examples of programs similar in structure and/or design to SchemePaint in particular. Certainly, *AutoCAD* belongs in this category, and represents an immeasurably more massive effort. *AutoCAD* does not include procedures for turtle graphics, however (though presumably a package of this sort would not be hard to include within the program); and SchemePaint includes at least a few features (including programmable colors) not included in *AutoCAD*. Moreover, SchemePaint, unlike *AutoCAD*, has the goal of leading novice users into programming via an extensive direct-manipulation interface; whereas *AutoCAD* (as noted earlier) appears to place much more stress upon providing the user with an extensive command library rather than upon leading users into writing (even simple) programs.

Beckman {9} describes an elegant Scheme package for graphics; and a Boxer program including direct-manipulation graphics and turtle graphics has also been implemented by Bruce Sherin.{36} This latter program includes some functionality that SchemePaint does not—including, most interestingly, the ability to translate direct-manipulation commands into Boxer statements.[25] It is perhaps worth noting that some commercial graphics applications also include "macros" among their facilities—a typical arrangement allows for sequences of operations to be associated with new menu

[25]On the other hand, SchemePaint likewise includes features that the Boxer program does not, including procedures (and interface facilities) for handling color graphics and dynamical systems.

options—but inasmuch as parametrizable procedures and control structures are missing from these applications, they cannot be said to be truly programmable.

7.2 Candidate Domains for Programmable Applications

Earlier it was mentioned that SchemePaint is only one sample program—an existence proof—intended to demonstrate the viability of the programmable-application strategy of software design. Naturally it is important to show not only the viability but also the *versatility* of that strategy; and to that end a range of examples, in widely divergent domains, will eventually be required.

Possibilities for such additional examples are joyfully easy to supply. One can almost walk up and down the aisles in the local computer-software shop, thinking of all the glorious programs that one *could* own if only these magnificent efforts in interface construction had been matched by efforts in expressive language construction. What follows, then, is a brief (and somewhat fanciful) catalog of domains around which one might construct programmable applications:

Music Composition

A programmable music composition tool might contain, on the interface side, a piano keyboard and drum machine connected via MIDI interface to the host computer; screen icons for specifying notes, timbre, amplitude; waveform construction and manipulation kits; rhythm construction kits appropriate for use with the drum machine; and many other staples of current commercial programs. On the language side, one might construct "note objects" with appropriate parameters (pitch, timbre, attack, decay, etc.); these might be connected sequentially (using appropriate parameters such as portamento) into passages and melodies, or superposed into chords; passages might be augmented with "bar division objects," or compared against melodic libraries (a boon to any musician who has wondered if the tune he has just written can be found in some fake book); harmonic strategies could themselves be subject to linguistic representation, so that the musician could listen to the same basic melody with a choice of orchestrations. A user might write simple procedures to search a composition for a particular sequence of notes, or

for every instance of an augmented seventh interval; or he might write a procedure that determines whether a given passage may be sung by a soprano (and what range the singer would need to perform that passage); or he might write a procedure that searches for any note whose associated waveform is similar to that of a brass instrument (by some metric) and changes that note to sound slightly less "brilliant."

A Programmable Atlas

A particularly useful programmable application, combining features of databases and simulations, could be structured around a "programmable atlas" for students of geography. This application would include a direct-manipulation interface allowing the user to call up onto the screen any of a large collection of geographic maps; perhaps, by selecting a location on a global map, the user could interactively "focus in" on a particular region. Other interface choices might allow the user to search for geographic features (cities, rivers, volcanoes), or to choose between map "modes" (e.g., one could select contour maps or maps that highlight national borders). Through the language component, the user could perform complex database searches (for instance, one could write a procedure to find all cities of a certain minimum population within fifty miles of the San Andreas fault). Other types of procedures might also be envisioned: one could write, for instance, a procedure that takes as its argument a particular mineral resource and returns a list of those nations containing natural supplies of that resource. The graphical component of our hypothetical atlas would support a wealth of fascinating programming projects: we could imagine writing a program that takes as argument a particular geographic path and then graphs that path by using animated icons superimposed upon the usual "background" maps. In this way we could write procedures to view the itinerary of Magellan's expedition, Napoleon's Russian campaign, or the migration patterns of bird species.

Thermodynamics Simulator

Imagine an application to help high school students visualize the behavior of ideal (and non-ideal) gases. One might begin at the interface side: allowing students to position "billiard ball" gas particles of equal mass; to give these particles initial velocities; and then to interactively measure such quantities as average velocity (related to gas temperature), average changes in momen-

tum due to collisions with walls (related to pressure), how these quantities vary with volume or with number of gas particles, and so forth. On the language side, we might write procedures involving "gas particle objects" as arguments: thus we could investigate a variety of phenomena beyond the range of any purely interface-driven system. For instance, we might interactively "boost" the kinetic energy of a selected particle and ask how long a time, on average, is required before that particle "settles down" to an energy near the system temperature (an experiment analogous to "relaxation time" investigations in chemical kinetics); we might write a procedure to draw a histogram of the number of collisions experienced by a collection of particles within a given time period; we might investigate mixtures of gas particles with different masses (a gas composed of "billiard balls and ping-pong balls"); we might choose to experiment with non-elastic collision rules.

Graph Theory

A relatively straightforward educational application might be designed to teach the concepts and algorithms of graph theory. Graphs could be constructed via direct manipulation: the student would use the mouse to position vertices and connect these vertices via (possibly weighted and/or directed) edges. The graphs constructed in this manner would then be accessible from the programming component, so that standard algorithms (e.g., finding a minimal spanning tree) could be illustrated.

Artificial Creatures

As a final example, we might imagine an environment designed for researchers in the field of "artificial life," and based around the constructions described in Valentino Braitenberg's well-known book *Vehicles.*{11} Simple "creatures" (from the early chapters of the book) could be created via an interface designed for that purpose; these creatures might be "wired together" directly on the computer screen, and their interactions in a two-dimensional "world" could be observed in real time. More complex rule-based creatures, including the capacity for remembering past interactions with their environments, might be created via a "creature-construction language." This language would also include procedures to represent creatures' reproduction (and mutation) rules, strategies for cooperation and competition with other creatures, and so on. In addition, the language would provide primitives for measuring

(and performing statistical analysis on) experimental quantities; for example, the researcher could measure the average life-span of all creatures of a given "species" during some multi-species simulation.

These five examples barely begin to illustrate (much less exhaust) the potential range of programmable applications. In the course of this paper, still more possibilities have been mentioned, almost in passing: a circuit-design tool, a programmable astronomical observatory, a language-based video-editing tool. Hypertext and hypermedia systems show promise for combining visual browsers with languages extended to work with "hyper-databases"; design environments for creating (e.g.) mechanical devices or optical systems are also strong candidates for the programmable-application approach, combining as they do a visual (even aesthetic) dimension and the need for simulation in rich, complex domains.

7.3 A Research Agenda

There are polemical reasons for designing new programmable applications along the lines just discussed: presenting a wide variety of these applications would not only prove the range and versatility of the concept, but would act as a "living argument" for its appeal to users. The pleasure experienced by people working with these systems would far outweigh, in persuasive force, any theoretical (or conjectural) argument that a paper such as this could possibly muster.

Beyond this, however, an existing body of work in programmable applications would begin to lend some lore to this strategy of software design: there would be case histories to appeal to, variations to compare. We might learn, for example, whether users of multi-language systems ever switch their attention from one language to another; whether programmable applications develop a reputation of being particularly hard (or easy) to debug and maintain; what kinds of user communities develop around such applications. In this sense, a design agenda of the kind outlined in the previous subsection acts in part as a research agenda as well. Before we can adequately resolve research questions about programmable applications "in general," we need a cluster of examples, each acting as an object-to-think-with.

This is not to say that framing possible research questions even at this

point is useless. The very notion of programmable applications as a "standard" design strategy suggests new emphases for research in both language and interface design. The following paragraphs, then, outline several potential areas of research that are related to (or facilitated by) the creation of programmable applications:

7.3.1 Language Learnability

Over the past decade an impressive and informative body of research has been generated, dealing with the question of how people learn to program. Some studies focus on the types of syntactic and semantic errors commonly made by novices {5, 31, 37}, some on the pedagogical role of explicit structural models of the language {28, 29}, some on how students acquire "higher-order" goals of organization {15, 38, 39}, some on particular programming topics such as recursion {24, 25, 33}, and some on the apparent "styles" of coding or learning among students {12, 40}. One topic that tends *not* to be addressed, however, is the relation between problem domain and language learning. That is, because most studies have been conducted with students in computer-science courses, they have tended to assume that "programming" is a skill identifiable with the writing of computer-science-related programs. Computer languages are implicitly identified with their roles within general-purpose (and hence domain-unenriched) programming environments. This assumption is particularly apparent in the types of tasks given to experimental subjects: students are compared according to how well they write programs that (e.g.) read in a list of numbers from a file, throw out the negative ones, and take the average of the remaining values. Students could *not* be compared according to how well (for instance) they can write programs to alter the sound of a piece of music or edit a video, because (by assumption) these are tasks unavailable to novice programmers.

Because programmable applications are domain-enriched environments geared toward learnability, we can imagine using these applications to incorporate new questions of domain specificity into studies of novice programmers. As an example, we might ask of programmer "style" whether it is a constant across domains (does a "top-down programmer" show the same tendencies in music as she does in electrical engineering); or we might ask whether art students learn programming through a graphics application more

quickly and errorlessly than they would through a general-purpose environment (and if so, what this implies specifically about the role of "external" domain knowledge at the early stages of learning to program); or we might ask whether debugging techniques such as tracing are more easily assimilated within some domains than others; or we might ask about the transferability of certain language constructs (does a student who writes a recursive graphics program within his paint application ever attempt to find uses for recursive strategies in a physics-simulation system).

Certainly among *expert* programmers, there is good reason to believe that knowledge of a program's function—expressed in the language of the "external" domain—is important to the understanding of program texts. Pennington {32}, in a study of program comprehension, asked a group of forty professional programmers to read, modify, and summarize a moderately complex program (in either COBOL or FORTRAN). In analyzing the programmers' summaries, Pennington writes:

> "High comprehension... programmers almost uniformly use a cross-referencing summary strategy that *combines* statements about the domain world with statements about the program world." (Emphasis in the original.)

Similarly, in a study of software design, Adelson and Soloway {4} found that the professional programmers they interviewed exhibited strong differences in design activity depending on whether the task at hand was a familiar one or not. In particular, expert designers given a task in an unfamiliar domain were unable to form an overall mental model (what the authors call a "global model") of the running system, leading to the potential for design errors.[26]

These studies indicate that for expert programmers, programming skill is hard to separate from domain knowledge—the "real-world" knowledge that the programmer is using or modelling. It would not be surprising to find that domain knowledge likewise has an effect on how programming languages are learned—we might plausibly expect that it would be easier to learn program-

[26] Adelson and Soloway interviewed both expert and novice designers for their study; even their "novice designers," however, were professional programmers with several years' programming experience.

ming in a familiar (presumably non-computer-science) context.[27]

Programmable applications could well act as excellent environments in which to study this question of how computer languages are learned across many domains of student interest. Indeed, these applications could enlarge the population of experimental subjects: just to throw in an additional potential research study, one could compare the performances of art students learning "PascalPaint" to those of electrical engineers learning "Pascal Circuit Designer" to see whether (e.g.) certain syntactic errors or semantic misconceptions among students seem to transcend the differences between those students' interests and backgrounds.

Before leaving this topic it is worth noting that research into language learnability would reflect back upon the design of programmable applications themselves. If we find that students are fearful about moving beyond "interface-driven activity" then we might investigate strategies for making the programmability option more accessible within these applications. We might ask, for instance, whether sample projects should be incorporated within applications; and, if so, whether they can be ordered in a way that respects the difficulty, sophistication, and relevance (within the domain) of the concepts that they employ. Or we might ask about the value of new kinds of collaborations: whether, perhaps, teaming an experienced graphics artist with an experienced programmer around an application like SchemePaint can facilitate mutually educational interdisciplinary projects.

7.3.2 Incorporating Knowledge Bases into Programmable Applications

As many of the examples in this paper have suggested, the programmable-application design strategy lends itself especially well to large, complex, and open-ended domains—graphics, music, and engineering, to name a few. In domains such as this—domains that tax the creative powers of the user—it would be desirable if our software could provide us not only with a tool, but also (where feasible) with intelligent assistance. For instance, in the course of designing a digital circuit, we might like our application to have some

[27] Pragmatically speaking, we would at least expect that the students' *motivation* to learn programming would be higher in a familiar context—a point also made in the previous section.

representation of our intentions: the program could supply us with a library of similar devices, or it might offer a selection of test inputs for the circuit, or it might catch design bugs (e.g., a possibility of a "race condition" in the circuit).

Pursuing such examples, then, it would be exciting to investigate the possibility of augmenting our programmable applications with "intelligent" knowledge bases that assist the user in her work. One promising strategy would be to incorporate a suite of independently loadable, special-purpose knowledge bases that the user could call upon for assistance in performing specialized tasks. For instance, an artist using a programmable graphics application might wish to call upon knowledge bases that assist in (e.g.) drawing faces, or flowers, or architectural blueprints. By restricting our cadre of "intelligent software assistants" to these relatively constrained tasks, we avoid (to some extent) the difficulties posed by trying to invest our programs with tremendous networks of commonsense knowledge: it is easier to construct an assistant to help draw flowers than to construct an assistant to help an artist think of something to do with an empty canvas. Moreover, our hypothetical intelligent assistant need not *direct* the user, but rather would probably act in an advisory capacity—checking the user's input for syntactic (or perhaps factual) errors, advising the user of program functionality of which she may not be aware, suggesting relevant references or examples from a database of previous work. In this sense, the proposed intelligent assistants are close in spirit to the *critics* described by Fischer *et al* in their work on design environments.{17, 18}

In the same vein, one might think of the "programming tutors" mentioned in Section 4 as instances of the intelligent assistants described here—though presumably these tutors would be somewhat more directive in style. As an example: a composer using a music application might interrupt her work to load in a programming tutor that would lead her through some music-based procedures—perhaps even employing examples relevant to her particular musical interests. Or conversely, a skilled programmer using such an application might wish to load in a *music* tutor. By incorporating a suite of pedagogical assistants within a programmable application, we might well answer the familiar complaint against educational programming environments ("microworlds")—namely, that they are too non-directive, offering students

no assistance with the content of a given domain (cf. Fischer {16}.)

7.3.3 Software Engineering/Software Design Issues

In many ways, creating programmable applications presents novel challenges in software engineering and design. Rather than being structured as conceptual "pyramids" of nested problems and sub-problems and sub-sub-problems all packaged underneath an opaque interface, programmable applications are instead conceived as new *languages* in which to re-express a domain. The design task will thus stress concepts such as language learnability, fidelity of language primitives to a given domain, syntactic and conceptual consistency among sets of embedded languages, and integration of language with interface. At the same time, "classic" concerns such as program correctness, reliability, maintenance, and reusability all remain important, but are viewed in potentially new ways.

As a first example, we might consider once more the question of how to combine language structures and interface features symbiotically, and to ask whether there are general software design strategies for doing so. In SchemePaint, for instance, we were able to write procedures that specified an operative pen-color; perhaps we might want to pursue more ambitious versions of this kind of functionality. For example, in a system for 3-D graphics, we might want to write procedures that specify "mouse-distance" along the z-axis (into the screen); or we might want to write procedures that dictate the position of a light source such that subsequent solids drawn with the mouse cast an appropriate shadow. The point of these examples is that we would like to classify them as instances of more general strategies for the ways in which a language can augment the power of an interface.[28] Conversely, we might like to classify SchemePaint's "nameable mouse-created objects" as part of a spectrum that includes *Canvas*{S2} macros and other techniques

[28]These questions do of course overlap with the notion of "tailorability" discussed earlier; one might argue that we are simply talking about the question of how to make applications tailorable by using a language to customize an interface. On the other hand, the examples in the text stress the use of our application's language to alter the interface behavior *in terms relevant to, and derived from, the application domain* (here, graphics). In this sense, our examples depend crucially on the existence of a domain-enriched language, which to some extent separates them from the more general interface-altering functions usually suggested by the notion of tailorability.

for creating linguistic representations of direct-manipulation activities.

Another important question, raised earlier in Section 4, relates to how one goes about building "enriched languages" appropriate to some specific domain. Suppose, for instance, we wish to follow the example of Abelson and Sussman by writing a circuit-design language: should wires be primitive objects in the language? Should AND/OR gates? What are the consequences of choosing one set of primitive objects rather than another? Is there any way to gauge the appropriateness of these decisions in terms of how electrical engineers might learn or use the language? And if we address these questions in the domain of digital circuits, have we learned anything at all for designing (say) a music language? Questions of this type embody the bedrock design issues for programmable applications, and represent relatively new territory in the study of software engineering.

The use of loadable libraries, such as the "Escher-language library" of SchemePaint, raises yet another host of issues. Some of these issues are staples of software-engineering courses—how to modularize programs using "packages" and generic interfaces—but others are relatively new. For example, there is the question of *interface* modules—our Escher library augmented not only the SchemePaint language, but the interface as well. We might ask, then, if the potential "interference problems" between interface modules are different in kind from those between language modules; and, if so, what new design strategies can help overcome these problems. Yet another question is how we might actually foster a kind of "constructive interference" between language libraries: are there strategies for combining libraries that allow us to do more than simply take the set-union of two sets of primitives? For example, suppose we add to SchemePaint a library for 3D graphics. Can we now create three-dimensional Escher tilings?[29] Will our turtle commands or dynamical systems primitives have to be rewritten to accommodate the addition of a third dimension? Moreover, the same questions arise on an even larger scale when we think of combining entire applications. Suppose, as suggested earlier in this paper, that we can somehow combine a music and graphics application by loading them together. How much "cooperation" between the two applications can we expect? Can we create musical

[29]Some of Escher's works did indeed employ depth perspective and "space-filling" designs.

compositions whose amplitude is specified by a curve drawn interactively within the graphics application? Can we incorporate playable music scores into pictures (so that, e.g., when the user selects that score with the mouse the appropriate music is played)?

Turning away from these wild scenarios, we might return to the more mundane (but thorny) questions usually asked by software engineers. What, for instance, is the prospect of software reusability in the design of programmable applications—can the designer of (say) a programmable atlas make use of the graphics primitives incorporated into SchemePaint? What about software maintenance—can we count on the well-known staying power and longevity of programming languages to carry over into applications based on those languages? Are bugs harder to find in these systems than in other types of applications? What is the role of the user community in designing, extending, and even maintaining these systems? All these questions are part of the larger project of understanding the typical "software lifecycle" of programmable applications; and all will require active study as new programmable applications come into existence.

7.3.4 On Beyond Mice

Finally—and following up on a point made earlier in Section 4—we recall that there is an entire toyshop-full of new interface devices that are likely to appear in the near future. An active question for research, then, will be how to extend our programmable-application design strategy to ever more powerful and expressive interfaces. To take a rather less-than-exotic example, we might imagine incorporating a mouse-with-tactile-feedback into our graphics application: it might be desirable, for instance, to create the illusion that the mouse is "bumping into" certain colors or points on the screen, and to make the conditions for "bumping" programmable. Or—to go just a little further out—we might wish to write a program that moves a Logo turtle in three-dimensional space while we watch its progress with a 3D-viewing device. Or—a bit further out still—we might write a DataGlove application that allows us to associate touching a virtual object with a procedure call: for instance, we might extend our earlier "vehicles" program so that we could watch simulations of the vehicles moving in three dimensions and interact with them by touch—perhaps handing them food, blocking light receptors,

or "rewiring" them by hand. The temptation at this point is to suggest even further-out examples, but this would risk the accusation of composing speculative fiction; so our examples will conclude here, in the anticipation that ten years from now a reader seeing this paragraph will regard its caution as quaint.

Acknowledgments

This paper reflects the influence of many people—though of course they may disagree with some, most, or all of what is written here. Foremost to acknowledge are Hal Abelson, Andy diSessa, and Gerald Jay Sussman. I would also like to thank Barry Dworkin, Wally Feurzeig, Mark Friedman, Matthew Halfant, Paul Horwitz, and Roy Pea for wonderful conversations. Orca Starbuck generously contributed her time and artistic talent to create five SchemePaint pictures. Liz Bradley, Arthur Gleckler, Mitchel Resnick, Bill Rozas, and Franklyn Turbak provided feedback, criticism, and encouragement. Barry Kort, Brian LaMacchia, and Kalyani Raghavan all pointed me toward helpful examples and information.

Bibliography

[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] Adelson, B. and Soloway, E. "The Role of Domain Experience in Software Design." *IEEE Transactions on Software Engineering* SE-11:11, Nov. 1985.

[5] Anderson, J. and Jeffries, R. "Novice LISP Errors: Undetected Losses of Information from Working Memory." *Human-Computer Interaction*, 1:2, 1985.

[6] Anderson, J. and Reiser, B. "The LISP Tutor." *Byte*, 10:4, 1985.

[7] Barr, A.; Beard, M.; and Atkinson, R. "The Computer as Tutorial Laboratory: the Stanford BIP Project." *International Journal of Man-Machine Studies*, v. 8, pp. 567-595, 1976.

[8] Baxter, N.; Dubinsky, E.: and Levin, G. *Learning Discrete Mathematics with ISETL*. Springer-Verlag, NY 1989.

[9] Beckman, B. "A Scheme for Little Languages in Interactive Graphics." *Software—Practice and Experience*, 21:2, 1991.

[10] Bonar, J. and Cunningham, J. "Bridge: Tutoring the Programming Process." In *Intelligent Tutoring Systems: Lessons Learned*. Psotka, J.; Massey, L.; and Mutter, S., eds. Lawrence Erlbaurm Associates, Hillsdale, NJ 1988.

[11] Braitenberg, V. *Vehicles*. MIT Press, 1984.

[12] Coombs, M.; Gibson, R.; and Alty, J. "Acquiring a First Computer Language: A Study of Individual Differences." In *Computer Skills and the User Interface*. Coombs, M. and Alty, J., eds. Academic Press, London, 1981.

[13] diSessa, A. and Abelson, H. "Boxer: a Reconstructible Computational Medium." *Communications of the ACM*, 29:9, 1986.

[14] diSessa, A.; Abelson, H.; and Ploger, D. "An Overview of Boxer." *The Journal of Mathematical Behavior*, 10:1, 1991.

[15] Ehrlich, K. and Soloway, E. "An Empirical Investigation of the Tacit Plan Knowledge in Programming." Yale University Department of Computer Science Research Report no. 236, April, 1982.

[16] Fischer, G. "Enhancing Incremental Learning Processes with Knowledge-Based Systems." In *Learning Issues for Intelligent Tutoring Systems*. Mandl, H. and Lesgold, A., eds. Springer-Verlag, NY 1988.

[17] Fischer, G.; Lemke, A.; and McCall, R. "Toward a System Architecture Supporting Contextualized Learning." In *Proceedings of AAAI'90.*

[18] Fischer, G. and Mastaglio, T. "Computer-Based Critics." In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences.* Jan, 1989.

[19] Foley, J. "Interfaces for Advanced Computing." *Scientific American* 257:7, 1987.

[20] Grabowski, R. with Huddleston, D. *Using AutoCAD.* QUE, Carmel, IN 1991.

[21] Gray, T. and Glynn, J. *Exploring Mathematics with Mathematica.* Addison-Wesley, Redwood City, CA 1991.

[22] Henderson, P. "Functional Geometry." *1982 ACM Symposium on Lisp and Functional Programming.*

[23] Johnson, W. and Soloway, E. "Proust." *Byte*, 10:4, 1985.

[24] Kessler, C. and Anderson, J. "Learning Flow of Control: Recursive and Iterative Procedures." *Human-Computer Interaction* 2:2, 1986.

[25] Kurland, D. and Pea, R. "Children's Mental Models of Recursive Logo Programs." *Journal of Educational Computing Research*, 1:2, 1985.

[26] Lewis, C. and Olson, G. "Can Principles of Cognition Lower the Barrier to Programming?" In *Empirical Studies of Programmers, Second Workshop*, Olson, G.; Sheppard, S.; and Soloway, E. eds. Ablex, NJ 1987.

[27] Marcus, A. and van Dam, A. "User-Interface Developments for the Nineties." *IEEE Computer* 24:9, 1991.

[28] Mayer, R. "Different Problem-Solving Competencies Established in Learning Computer Programming With and Without Meaningful Models." *Journal of Educational Psychology*, 67:6, 1975.

[29] Mayer, R. "A Psychology of Learning BASIC." *Communications of the ACM*, 22:11, 1979.

[30] Miller, M. "A Structured Planning and Debugging Environment for Elementary Programming." In *Intelligent Tutoring Systems.* Sleeman, D. and Brown, J., eds. Academic Press, Inc., London, 1982.

[31] Pea, R. "Language-Independent Conceptual 'Bugs' in Novice Programming." *Journal of Educational Computing Research,* 2:1, 1986.

[32] Pennington, N. "Comprehension Strategies in Programming." In *Empirical Studies of Programmers, Second Workshop,* Olson, G.; Sheppard, S.; and Soloway, E. eds. Ablex, NJ 1987.

[33] Pirolli, P. "A Cognitive Model and Computer Tutor for Programming Recursion." *Human-Computer Interaction* 2:4, 1986.

[34] Rees, J. and Clinger, W., eds. "Revised^3 Report on the Algorithmic Language Scheme." MIT Artificial Intelligence Laboratory Memo 848a, September 1986.

[35] Resnick, M. "Animal Simulations with *Logo: Massive Parallelism for the Masses." In *From Animals to Animats.* Meyer, J. and Wilson, S., eds. MIT Press, Cambridge, MA 1991.

[36] Sherin, B. Personal communication.

[37] Sleeman, D.; Putnam, R.; Baxter, J.; and Kuspa, L. "An Introductory Pascal Class: A Case Study of Students' Errors." In *Teaching and Learning Computer Programming.* Mayer, R., ed. Lawrence Erlbaum, Hillsdale, NJ 1988.

[38] Soloway, E. and Ehrlich, K. "Empirical Studies of Programming Knowledge." *IEEE Transactions on Software Engineering,* SE-10:9, 1984.

[39] Spohrer, J.; Soloway, E.; and Pope, E. "A Goal/Plan Analysis of Buggy Pascal Programs." *Human-Computer Interaction* 1:2, 1985.

[40] Turkle, S. *The Second Self.* Simon and Schuster, NY 1984.

[41] Vardi, I. *Computational Recreations with Mathematica.* Addison-Wesley, Redwood City, CA 1991.

[42] Wagon, S. *Mathematica in Action.* W. H. Freeman, New York, 1991.

Software

[S1] *AutoCAD.* Autodesk, Inc. Sausalito, CA.

[S2] *Canvas.* Deneba Software. Miami, FL.

[S3] *Director.* MacroMind, Inc. San Francisco, CA.

[S4] *4th Dimension.* Acius, Inc. Cupertino, CA.

[S5] *HyperCard.* Apple Computer, Inc. Cupertino, CA.

[S6] *ImageStudio.* Letraset, USA. Paramus, NJ.

[S7] *MacPaint.* Claris Corporation. Santa Clara, CA.

[S8] *MacScheme.* Lightship Software, Inc. Beaverton, OR.

[S9] *Mathematica.* Wolfram Research, Inc. Champaign, IL.

[S10] *Visual Basic.* Microsoft Corp. Redmond, WA.

Color Plate Key for SchemePaint Pictures

Plate 1. A "basins-of-attraction" map created using the dynamical systems package. The colors red, green, and blue correspond to the three complex cube 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 dynamical systems 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.

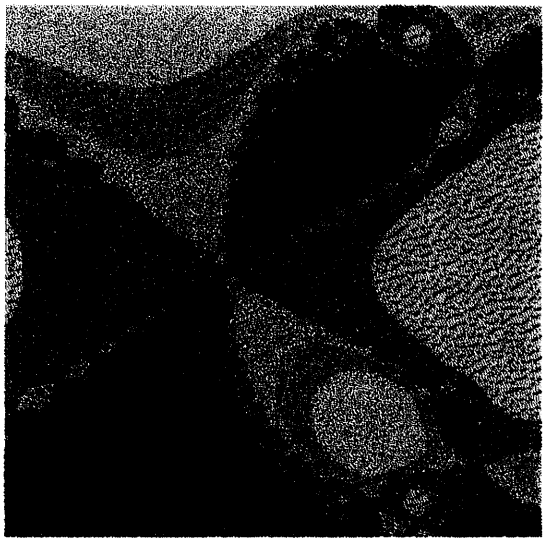
Plate 6. The "dragon curve" decorates a hand-drawn seahorse.

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

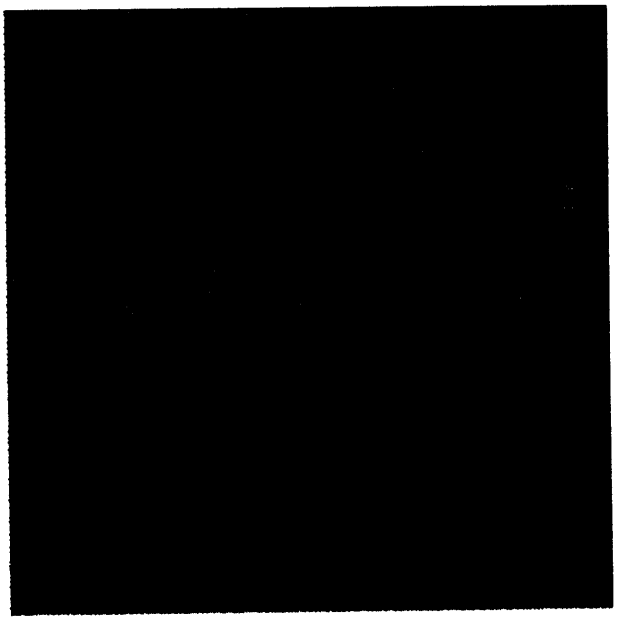
Plate 8. A peacock displays fractal feathers.

Plate 9. An Escher-esque tiling of fish and butterflies.

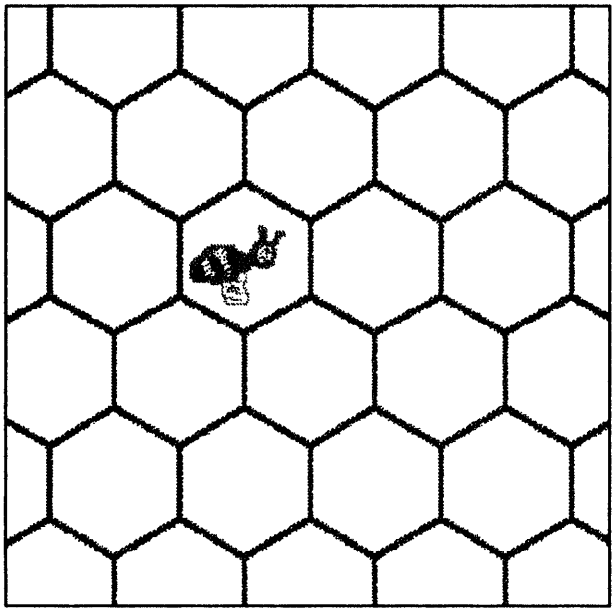
Artwork for Plates 4-8 by Orca Starbuck.



Color Plate 1

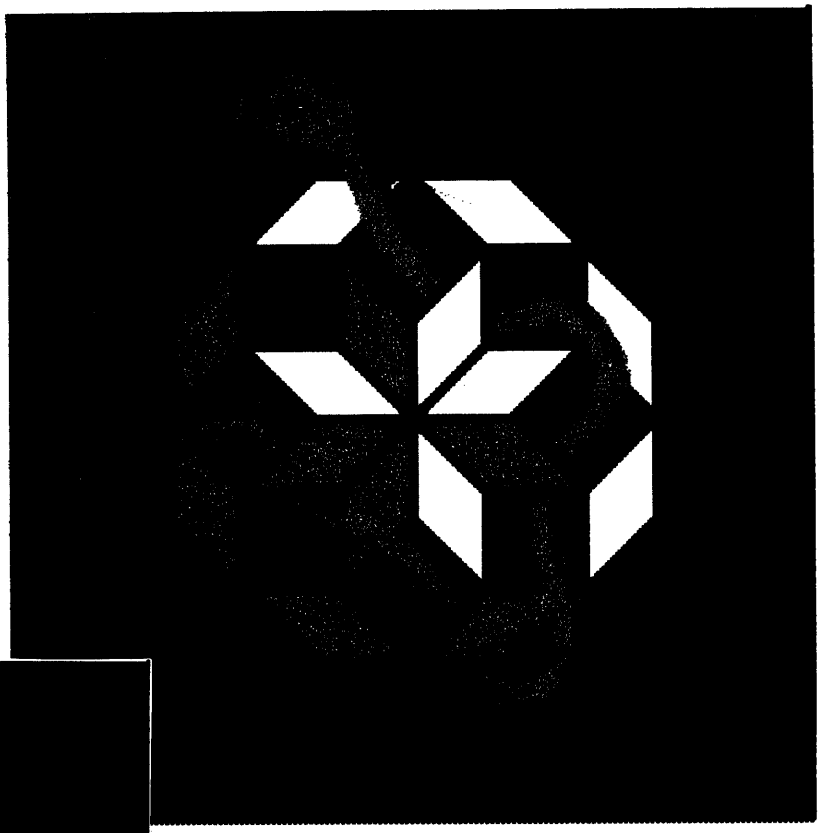


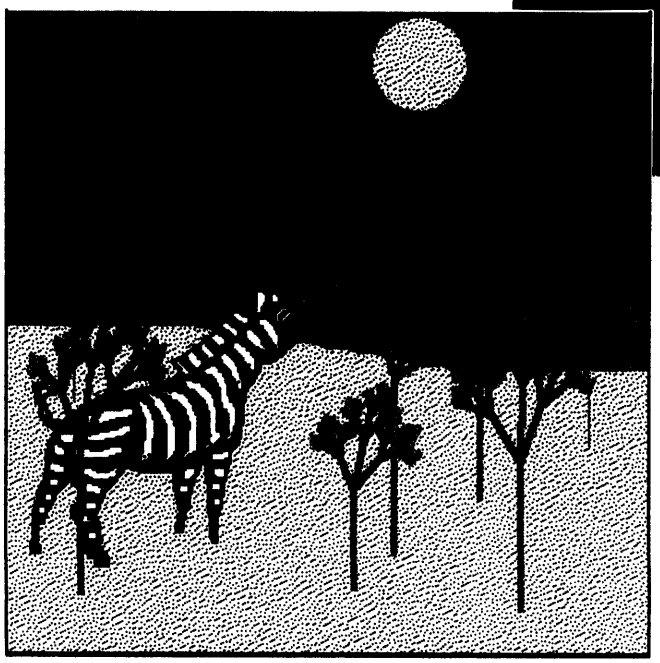
Color Plate 2



Color Plate 3

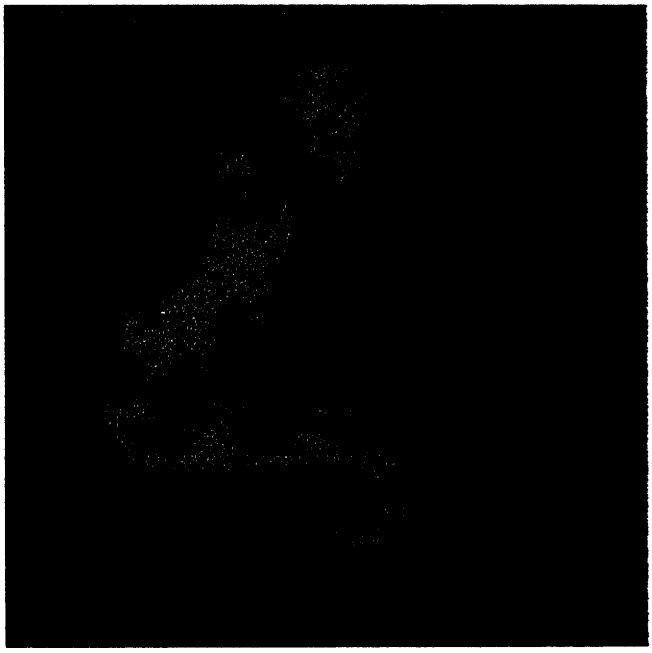
Color Plate 4

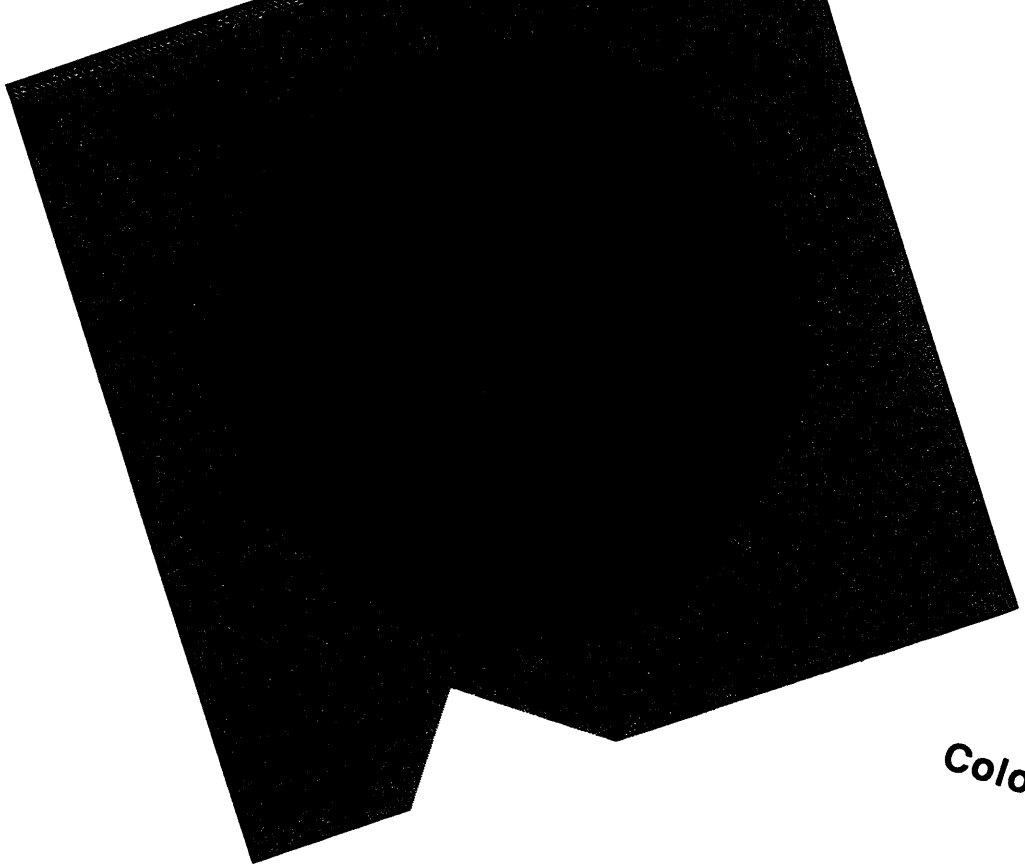




Color Plate 5

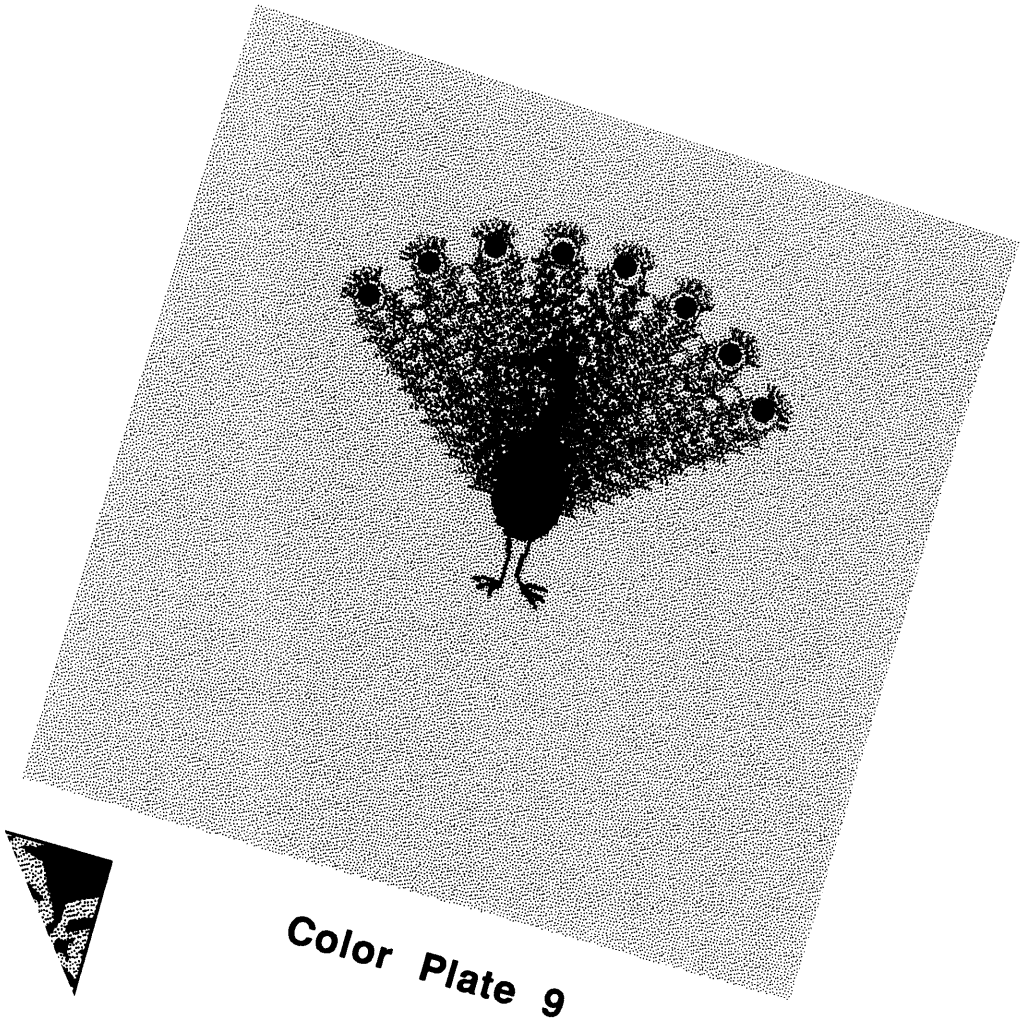
Color Plate 6





Color Plate 7

late 8



Color Plate 9