Programmable applications for the arts: computational tools for hand, eye and mind

Michael Eisenberg

Commercial applications for the arts tend to enforce a division between the use of learnable direct manipulation interfaces and the use of powerful, well supported programming environments. In contrast, programmable applications integrate these two software-design paradigms (i.e. direct manipulation and programming languages) and thereby attempt to exploit the strengths of both. A sample graphics application, SchemePaint, is outlined, and some of the issues related to the creation of programmable applications for the arts are discussed.

Keywords: direct manipulation, interactive programming environments, end-user programming

PROGRAMMABLE APPLICATIONS: STRATEGY FOR SOFTWARE DESIGN

In the design of applications for the arts, there seems to be an unfortunate division between two broad philosophies of software construction. On the one hand, there are 'user-friendly' applications that work solely via direct manipulation interfaces, often advertising that they are designed for the 'nonprogrammer', while on the other hand there is the much more arcane and unapproachable world of programming environments, generally assumed to be the province of full-time computer scientists or (on occasion) extraordinarily technically minded artistic professionals.

This division between direct manipulation and programming has historical roots that are visible in the literature of human–computer interaction. Direct manipulation has long been presented as an alternative to programming. Indeed, the classic early paper on the subject referred to direct manipulation as 'a step beyond programming languages' [Shneiderman, 1983]. Similarly, Hutchins et al. [1986] assume a conflict between direct manipulation and programming when they write

In the end, many things done today will be replaced by Direct Manipulation systems. But we will still have conventional programming languages.

The contrast between direct manipulation and programming is a plausible one. The former design strategy emphasizes a concrete, visual representation of computational objects, and the latter a comparatively abstract 'linguistic' representation. Nevertheless, the very real distinctions between these two notions need not be reflected in an enforced dichotomy of system design. Instead, the potential exists for the design of programmable applications integrating the strongest features of both these paradigms of software design [Eisenberg, 1991; Eisenberg, 1992]. To support this argument, this paper describes a programmable graphics application named SchemePaint. SchemePaint includes both a Macintosh-style interface for creating drawings by hand, and a graphics-enriched interactive programming environment (using the SCHEME language) that permits artists to write brief but powerful graphics procedures. By combining tools that support hand–eye coordination with tools that support program construction, SchemePaint affords graphics artists an expressive range beyond that currently provided in many commercial applications. Moreover, SchemePaint incorporates a number of experimental features whose purpose is to explore the potential for a more 'symbiotic' style of cooperation between interface and interpreter; these features are intended to blur the distinction between language environments and direct manipulation, to the mutual enrichment of both.

The remainder of this introductory section is devoted to a brief description of programmable applications, and how they address the specific problems associated with either direct manipulation interfaces or programming environments in isolation. The following section provides an overview of SchemePaint, and includes several examples of graphical work created with the program. The third section discusses some recurrent issues in
the design of programmable applications, with particular attention being paid to creating applications for the arts. The fourth section concludes with a discussion of ongoing and future directions for research.

**Problems with direct manipulation interfaces**

By and large, the philosophy of ‘pure’ direct manipulation (typified by most of the better-known Macintosh applications) shuns the inclusion of user-accessible programming languages and instead provides users with a collection of (by assumption, simpler) interface elements: menus, dialog boxes, icons, and so forth. While such an approach often results in programs that are initially learnable, long-term users often begin to find these applications strangely and unexpectedly inexpressive. The user of a new graphics application may (for instance) discover that she would like to create a cycloid, or a random walk, or a fractal, or a maze, or a tiling pattern, or indeed that she would like to pursue any of a myriad of graphical ideas that happen not be supported by the application. When this occurs, the typical response of the direct manipulation designer will be to add more features: more menus, more dialog boxes, more icons. The resulting ‘enhanced’ system is often a baroque and intimidating collection of ad hoc features that address individual complaints but somehow remain, when viewed as a whole, insufficiently expressive.

The problem with direct manipulation interfaces is rarely due to the quantity of available interface operations. Rather, the problem is a predictable consequence of the design philosophy itself: the semantics of direct manipulation is simply too impoverished, all on its own, to accomplish a wide range of sophisticated (but not unusual) tasks. Specifically, the features of programming languages (the ability to control constructs, to create stable data structures, to build abstractions by naming objects and procedures) are precisely the collective ‘Achilles heel’ of direct manipulation. Indeed, these problems have long been identified. As Hutchins *et al.* [1986] observe:

> Not all things should be done directly ... Direct Manipulation interfaces have difficulty handling variables, or distinguishing the depiction of an individual element from a representation of a set or class of elements.

The association of direct manipulation with ‘user-friendliness’ is based on the assumption that, by contrast, a programming environment must perforce be distasteful to users. However, the judgment that users cannot or will not write programs is arguable. After all, our hypothetical long-term paint-program user might well have been tempted into some beginning projects by an application that provided the option of programmability, and the learning experience might have been greatly motivated by the fact that it was rooted in a specific, personal, graphics-related project (such as drawing a maze). As Nardi [1993] eloquently argues in her recent book,* We have only scratched the surface of what would be possible if end users could freely program their own applications ... As has been shown time and again, no matter how much designers and programmers try to anticipate and provide for what users will need, the effort always falls short because it is impossible to know in advance what may be needed ...

**Problems with general-purpose programming environments**

Despite the arguments of the previous paragraphs, it must be admitted that programming environments present their own limitations. First, there are many occasions when an iconic interface is in fact more expressive than a programming language. For instance, it is easier for someone with artistic skill to draw a portrait by hand (i.e. using a pen tablet and mouse) than it is to program a computer for the task. Going further, one might say that this example reflects a limitation of not only programming languages, but language in the broader sense. It is easier to draw a portrait than to explain in English how to accomplish the job. Here, arguably, lies the true strength of direct manipulation, not in replacing or superseding linguistic media, but in supplementing those media for tasks that seem continually to frustrate the attempt to render them in words (or code).

There is yet another problem with general-purpose programming languages, namely that they are typically too general-purpose, providing insufficient programming support for professionals outside the realm of computer science. A graphic designer (or musician, or video editor) should be able to sit down at a fully prepared programming environment in which the language has been specifically enriched for working in his or her particular domain. (A good rule of thumb is that it should be possible for the professional to use a domain-enriched language to write interesting programs of one-half of a page or less.) Simply because these objects and procedures can be built in a general-purpose environment, this is no reason for the burden of building them to be placed on the user. The artist is primarily interested in producing graphical work, and not in building graphics applications.

In sum, then, neither direct manipulation interfaces nor general-purpose programming environments can alone provide a foundation for sufficiently expressive applications. This is the motivating idea behind programmable applications. What we would prefer are systems that combine the accessibility of direct manipulation interfaces with the extensibility and power of programming languages. We now turn to a specific example of this design strategy for the domain of graphics.

**SCHEMEN PAINT: PROGRAMMABLE APPLICATION FOR GRAPHICS**

SchemePaint [Eisenberg, 1992] is a graphics application combining the elements of a direct manipulation interface (presented in the familiar style of many commercial applications) and a graphics-enriched SCHEME interpreter [Abelson and Sussman with Sussman, 1985]*. Figure 1 depicts the SchemePaint interface. The SchemePaint (canvas) window is where graphics work is created, the Pen window allows the user to choose among various pen colors and modes, and the Transcript window provides interaction with the SCHEME interpreter. The

---

*The program is written in MACSCHEME (LightShip Software, Beaverton, OR, USA) and runs on all Apple's color Macintosh-series machines (including the Ix, IIfx, and Quadra 700).
four menus at the right of the screen's top line are specific to SchemePaint, and provide the user with a selection of useful operations (for instance, the Paint menu supplies commands for clearing the screen, setting or clearing the background grid, and so forth); the four named menus on the left are supplied with the MACSCHEME system in which SchemePaint is written.

The language supplied with SchemePaint is an 'enriched' SCHEME in which an extensive collection of graphics-related procedures have been included. Figure 1 shows the user employing one of SchemePaint's embedded libraries — in this case for turtle graphics [Abelson and diSessa, 1980; Papert, 1980]. Here, the user has written a new SCHEME procedure to draw an octagon by having the turtle (programmable cursor) repeat eight times a 'forward' move followed by a rightward turn of 45°:

\[
\text{(define (octagon side) (repeat 8 (forward side) (right 45)))}
\]

This octagon procedure is now added to the user's graphical vocabulary. Indeed, there is no essential difference between the status of this new procedure and any of the built-in graphics procedures associated with SchemePaint. In the final Transcript line of Figure 1, the user has employed the new octagon procedure by making a pattern of eight octagons:

\[
\text{(repeat 8 (octagon 30) (right 45))}
\]

Thus, in the course of five lines of code, the user has been able to create a geometric effect that might well be a severe challenge in many direct manipulation systems. Although the programming skills involved in creating this design are not trivial, neither do they require years of mathematical training. In fact, similar patterns are routinely generated by elementary-school-age children after less than a semester of LOGO experience. Moreover, the creation of this new figure does not preclude the user from employing the mouse to draw lines by hand. One SchemePaint picture (created by O Starbuck and reproduced in Eisenberg [1991] in color-plate form) in fact combines the simple rotated-octagon design of Figure 1 with a hand-drawn snake. Figure 2 likewise depicts the integration of computer-drawn and hand-drawn designs.

In addition to its turtle-graphics procedures, SchemePaint includes libraries for creating Escher-like tiling designs, for working with dynamical systems in both 2D and 3D, and for manipulating colors. (Newer additions include extensions for spline generation, customized region-filling procedures, and procedures for experimenting with 'geometric substitution' algorithms [Glassner, 1992]. These libraries are designed as independent 'sublanguages', with an eye toward modularity. Thus, the user who requires an introduction to programming may begin by writing (for example) simple turtle-graphics procedures, and may later 'graduate' to working with the dynamical systems and color-manipulation packages. SchemePaint also includes a number of features that represent experiments on the theme of interface-language cooperation. One such feature, 'programmable mouse colors', allows the user to specify (via a procedure from position to color) the coloring algorithm to be used by the mouse when used as a pen. Another feature, 'free-running brushes', permits the user
Figure 2 Two SchemePaint pictures; (a) hand-drawn peacock with a fractal (turtle-drawn) tail, (b) butterfly with Julia-set wings posed on mostly hand-drawn flower. [(a) Artwork by O Starbuck.]

Figure 3 New interface window that accompanies the 3D-turtle embedded language in SchemePaint. [The top two panes depict a ‘wireframe’ turtle in two orientations. The second row provides buttons with which to turn the turtle about the x,y,z axes. The third row provides buttons for turning the turtle about its own ‘internal’ heading vectors. The fourth row provides commands for taking a ‘snapshot’ of the turtle’s state and restoring that state, along with transferring the turtle’s state between this (interface) window and the canvas window. The fifth row allows the user to specify the interface turtle’s position and heading directly (via numerical input).]
Programmable applications for the arts: computational tools for hand, eye and mind: M Eisenberg

Once the user has arranged the 3D turtle’s heading to her liking by using the buttons in the interface window, that heading may finally be transferred to the mobile turtle in the SchemePaint window by selecting the box labelled ‘t→c’ (for ‘turtle window → canvas window’). When this box is selected, the ‘actual’ mobile 3D turtle now takes on the heading of its cousin in the interface window. At this point, the ‘actual’ turtle may be manipulated by expressions such as (3D-forward 20), as shown in Figure 4. The net effect is that the turtle’s heading may be chosen ‘by eye’, using the interface buttons as manipulanda, while the resulting heading may then be transferred to the mobile turtle as a precondition for working with the newly loaded language primitives. In this manner, the system permits a style of activity that moves back and forth between direct manipulation and programming. Figure 5 shows a SchemePaint picture of a 3D botanical structure generated with the use of the 3D-turtle interface and language.

Finally, while it is fair to describe the application as a working prototype, it should nonetheless be mentioned that SchemePaint has been used over extended periods of time by approximately 40 people, including graphic artists, local elementary and middle school students, undergraduate and graduate students at the University of Colorado, USA, and a member of the university’s faculty in the Department of Fine Arts. (If one includes in this count several workshops for high school students at the university, the number of SchemePaint users exceeds 100.) The program has been used as a gentle introduction to SCHEME programming for children as young as ten, and has more recently been extended by a graduate student for use as a medium to generate decorated ‘origami folding nets’ (these nets, once produced on the screen, can be printed out on a color printer and folded into custom-designed paper models).

KEY ISSUES IN DESIGNING PROGRAMMABLE APPLICATIONS FOR THE ARTS

Creating learnable applications languages

The (necessarily brief) description of SchemePaint in the previous section suggests a variety of open and perhaps controversial questions about the design of programmable applications. One recurrent question involves the ‘language issue’, i.e. given that we wish to construct a new programmable application, should we include a brand new application-specific language, or (as in the

Figure 4  Example of use of interface
[At the left, the user has used the ‘axis-rotation’ buttons to change the turtle’s heading so that it points along the vector (1, 0, 1). By selecting ‘t→c’, this heading is then transferred to the 3D turtle in the SchemePaint (canvas) window at the right. Now the user evaluates the expression (3D-fd 20), causing the 3D turtle at the right to make a forward move in its new direction.]
case of SchemePaint) an ‘enriched’ version of an existing general-purpose language? Plausible arguments on both sides of this issue may be advanced, and, while some current language-based commercial applications (such as Hypercard* [Goodman, 1990] and Mathematica” [Wolfram, 1988]) have leaned toward the application-specific-language solution, others (such as AutoCAD® [Grabowski, 1991] and the venerable Emacs editor [Stallman, 1986]) have opted, like SchemePaint, for the use of LISP dialects. Regardless of one’s opinions on the merits of these various language choices, it should be noted that programmable applications may well be designed to accommodate a multiplicity of languages, allowing the user to select whichever interpreter he or she prefers. Conceivably, the selection of available languages might be arranged in a hierarchy ranging from ‘novice-oriented’ languages geared for learnability [diSessa and Abelson, 1986; Papert, 1980] to more elaborate languages (perhaps incorporating other concerns such as runtime efficiency).

Yet another school of thought on this issue rejects the notion that users should be confronted with the complexity of programming, and holds that induction- or demonstration-based techniques (such as programming by example [Cypher, 1993]) can provide the user with the expressiveness of programming while simultaneously relieving him of the need to master an unwieldy programming syntax. While much interesting work continues to be done along these lines, the design philosophy of SchemePaint is based on the notion that a ‘conventional’ programming environment, i.e. a programming environment that makes no attempt to shield the user from notions such as iteration, conditionals, procedural abstraction, and so forth, is both more expressive and (when integrated within an application) less distasteful to users than is generally assumed. Nardi [1993] advances interesting arguments along these lines, pointing out that such formal symbolic systems as baseball scorecards and knitting patterns may be regarded as task-specific (and hence, for those interested in their respective domains, motivating) languages. At the same time, Nardi expresses caution about the expressiveness of programming by demonstration:

Programming by example techniques by themselves ... will not solve the problem of giving end users the power to write complex domain specific applications. The difficulty of expressing terminating conditions and conditionals, the need to inhibit random/exploratory user actions, the lack of error correction mechanisms, and the high cost of the inferencing capabilities of programming by example systems make offering real computing power to end users via these techniques quite difficult. A challenge for the future is to determine how to fit specific programming by example techniques into larger programs that offer power and flexibility to end users.

Learning programming through applications

Programmable applications, as suggested by the previous paragraphs, may well be an excellent way of teaching programming. One could imagine a course in (say) ‘computational art’ geared toward students in fine arts and based upon an interactive application such as SchemePaint. (In fact, as noted earlier, SchemePaint has been used for this very purpose.) It is significant in this respect that much of the most careful work in the learnability of programming (e.g. Rist [1986], Cunniff and Taylor [1987] and Spohrer and Soloway [1986]) has focused on computer-science-related tasks (such as reading in lists of numbers, filtering out the negative values, and sorting the remaining values). More domain-oriented tasks (such as coloring a region via some specified algorithm, or repeatedly transcribing, altering, and replaying a piece of music) are by implication less typical of the kinds of problems that programming novices would encounter. Programmable applications could spur research into the learnability of programming languages suited to the interests and professional vocabularies of learners.

User communities/collaborations

Yet another issue in programmable application design, and one perhaps especially important in the arts, involves creating applications that can serve to foster collaborations between professionals in a variety of domains. Certainly, an application like SchemePaint permits collaborations between programmers and graphic artists (some SchemePaint pictures have in fact been produced in just this way). More generally, the existence of language-based applications permits users to think of embedded ‘sublanguages’ as tangible, reproducible artefacts which may be passed among practitioners, progressively enriching the vocabulary of the entire professional community. For example, a user who develops a package of procedures for blending colors in some new way might be inclined to donate that package (e.g. via user-oriented publications) to fellow users, and these procedures might then become the basis for still further elaboration by other users. This portrait is qualitatively different, moreover, from the way in which current commercial applications typically get extended: generally, the role of application extension is ceded to professional programmers, and the ‘add ons’ produced are opaque in their construction. Thus, rather than enriching the professional community, it could be argued that the current style of application extension tends to reinforce the barrier between working artists and those who employ programming as a means of expression. Programmable applications, and the user communities that could grow around their use, offer a hope of dissolving that barrier. (Compare this with Gantt and Nardi’s [1992] description of the role of local program developers (‘gardeners’) among communities of users of programmable CAD applications; Mackay [1991], by contrast, offers a less encouraging view of this type of collaborative process.)

Pursuing this point a bit further, it should be noted that programming languages can boast singular advantages as a public means of communication: users can produce their innovations in an economical, accessible, and extensible fashion. (The Mathematica community, by way of example, has capitalized on this strength of programming languages by producing an ever-growing collection of Mathematica-based textbooks and a quarterly journal.) Indeed, programming languages in this

---

*Apple Computers Inc., Cupertino, CA, USA.
†Wolfram Research Inc., Champaign, IL, USA.
‡Autodesk Inc., San Rafael, CA, USA.
§In fact, the menu labelled ‘Languages’ in SchemePaint permits the user to choose between SCHEME and an (admittedly very sparse) graphics-enriched BASIC interpreter.
Programmable applications for the arts: computational tools for hand, eye and mind: M Eisenberg

respect afford some of the same advantages that Ong [1982] ascribes to written forms of natural language:

A modern grapholect such as 'English' to use the simple term which is commonly used to refer to this grapholect, has been worked over for centuries. . . It has been recorded massively in writing and print and now on computers so that those competent in the grapholect today can establish easy contact not only with millions of other persons but also with the thought of centuries past. . . [It] bears the marks of the millions of minds which have used it to share their consciousness with one another.

FUTURE DIRECTIONS AND RESEARCH

New domains

The first order of business in programmable-applications research is to generate a much larger collection of sample systems, providing a body of lore from which to draw. In the absence of such a broad spectrum of experimental systems, it is difficult to answer questions regarding the proper role of domain specificity in language design, the viability of multilanguage systems, and the role of user communities (among many other issues). Certainly, some current commercial applications do incorporate various degrees of programmability (ranging from macros and special-purpose 'scripting languages' to fully fledged interactive programming environments), but in many cases these represent only a partial exploration of the available design space, and most commercial applications include at best only tentative steps toward programming (e.g. macro facilities that fail to include control constructs, visual languages that fail to allow procedure naming and abstraction, and so forth).

Within the arts, music composition might be a particularly promising domain for programmable application design, inasmuch as new applications could integrate the direct use of electronic piano keyboards (or other instrumental interfaces) with interactive programs. To take one rather simple (and no doubt simple-minded) example, the user might write a program to slightly alter the tuning or timbre of the electronic keyboard depending in some algorithmic fashion on the average pitch (or duration, amplitude etc.) of the previous n notes played. Programmable applications focusing on animation, video editing, and 'virtual reality' environments are similarly exciting prospects.

Embedding knowledge-based assistance in applications

Programmable applications, at their sparsest, provide little in the way of domain-specific assistance for users. A SchemePaint user, for example, cannot browse through collections of previous work with the program, and she cannot load interactive tutorial systems to assist her in learning about the system, and the program does not provide domain-specific critiquing components [Fischer et al., 1991]. A large variety of approaches may thus be explored with the intent of making programmable applications more supportive; some of these approaches touch upon issues long associated with artificial intelligence research. (For instance, the recognition of user goals required by an application endowed with an active critiquing component is similar in scope to the type of user modelling required by intelligent tutoring systems.)

A recently begun project in collaboration with G Fischer is addressed toward integrating programmability with supportive domain-specific design environments [Eisenberg and Fischer, 1994]. To this end we have developed a prototype system named SchemeChart, centering on the design of charts, graphs, and information displays; a user working with the application is able to create both 'standard' charts (such as bar graphs) and more elaborate charts via an embedded programming environment. SchemeChart permits the user to gain familiarity with the embedded language by browsing an iconic 'catalog' of standard chart designs; when a particular chart-type is selected, the user can access an editable window of program constructs relevant to the creation of the given chart. The system likewise includes software critics reflecting some of the graphical-design principles reflected in texts such as Tufte [1983].

Integrating language with newer interface devices

Future programmable applications for the arts should be aimed at taking advantage of the variety of new interface devices on the immediate horizon: pen-based systems, 3D viewing devices, DataGloves [Foley, 1987; Marcus and van Dam, 1991], and so forth. Again, these newer devices can be seen as opportunities for new kinds of interaction with programming languages, rather than as tools that 'protect' users from the assumed complexity of the underlying language medium. One might imagine, for instance, that a graphic artist equipped with a head-mounted viewing device could write a program specifying a particular algorithm to be employed in accordance with his visual attention. The program might specify, for example, that the next object to which the artist attends should be highlighted, recolored, and (if the artist's visual attention remains focused on that object) moved forward in the axis perpendicular to the plane of the picture. Thus, the advent of an input device such as this one suddenly allows the notion of 'attention span' to be directly and naturally incorporated into an interactive programming language. Yet a more ambitious longer-term prospect would be to create complete 'application machines' including domain-specific special-purpose hardware, interface devices, and language environments. Such application machines could represent the endpoint of a continuing process of detente between direct manipulation and programming, and we could finally view direct manipulation not as a 'step beyond', but as a step alongside, programming languages.

ACKNOWLEDGEMENTS

Julie DiBiase, Chris DiGiano, Ann Nishioka, Orca Starbuck, and Robert Tarrall all contributed to the work described in this paper. The author would also like to thank Hal Abelson, Wally Feurzeig, Gerhard Fischer, Paul Horwitz, and Gerald Sussman for especially helpful discussions and insight. The research is partially supported by the US National Science Foundation (grant
Programmable applications for the arts: computational tools for hand, eye and mind: M Eisenberg

IRI-9015441) and by an NSF Young Investigator grant (IRI-9258684).

REFERENCES

Cypher, A (Ed.) Watch What I Do MIT Press, USA (1993)
Eisenberg, M ‘Programmable applications: interpreter meets interface’ Technical Report 1325 Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA (1991)
Eisenberg, M and Fischer, G ‘Programmable design environments: integrating end-user programming with domain-oriented assistance’ Proceedings CHI’94 (to be published)
Foley, J ‘Interfaces for advanced computing’ Scientific American Vol 257 No 7 (1987)
Grabowski, R and Huddleston, D Using AutoCAD Quick Carmel, IN, USA (1991)
Prusinkiewicz, P and Lindenmayer, A The Algorithmic Beauty of Plants Springer-Verlag, USA (1990)
Rist, R ‘Plans in programming: definition, demonstration, and development’ in E Soloway and S Iyengar (Eds.) Empirical Studies of Programmers Ablex, USA (1986)
Spohrer, J and Soloway, E ‘Analyzing the high frequency bugs in novice programs’ in E Soloway and S Iyengar (Eds.) Empirical Studies of Programmers Ablex, USA (1986)
Wolfram, S Mathematica A System for Doing Mathematics by Computer Addison-Wesley, USA (1988)