Aesthetic Programming

Paul A. Fishwick

Department of Computer & Information Science & Engineering
University of Florida
Gainesville, Florida 32611, U.S.A.
fishwick@cise.ufl.edu

December 5, 2000

Abstract

By marrying traditional methods for computer programming with an artistic temperament, we give birth to a new phenomenon: the *aesthetic program*. Our work builds on visual approaches in programming as well as modeling for software, where I envision a gradual evolution from program to model. The need for the aesthetic model is strengthened with the importance of personalized, individually-tailored, models. I, and my students, have formulated the *rube Project* methodology around the use of 3D web-based virtual world construction of models. Initial results suggest that these models are artistic, while containing sufficient symbolism and concise metaphoric mapping as to be executable on a computer.

1 Introduction

The merger of computer programming and art usually manifests itself under the rubric of terms such as *algorithmic X*, *computer X*, or *digital X*, where *X* is replaced by one of the arts, such as *art* or *music*. This naming trend denotes a qualification of art, or more particularly, how art is modified as a result of computer technology. However, if we reverse the word order to obtain "artistic computation," we find a relative dearth of information, with Sec. 4 overviewing the relevant literature. Software representations that lend themselves to an artistic representation tend to be minimalist and iconographic. More work needs to be done in inculcating the importance of the arts inside of the metallic computer box, and within the virtual spaces inhabiting the box. It's not just necessary that the box, monitor, and interaction devices be stylistic—we also need software that is aesthetic and of considerable sensory appeal. It is only by doing so, that we can reinvent programming to be more humane, and interestingly, to become closer to traditional engineering which has historically managed to forge alliances with style, as in the field of architecture.

When one thinks of computer programs in their complex and cryptographic text-based glory, the idea of aesthetics does not generally come to mind; however, the area of programming has developed its own aesthetics over the past four decades. Knuth [1] authored a classic series of books entitled *The Art of Programming*, and later created an approach to text-based software development termed *literate programming*[2], in which programmers develop more readable programs that, in themselves, serve as complete textual, typeset documentation. The resulting programs are attuned to what could be called a *text-based aesthetic*, with forms that go beyond mathematical formalism toward pseudo natural language, but are still fairly minimal when compared with the extensive history of aesthetic movements over several millennia in art. Much of this may well be due to the historical differences, with programming being a fledgling enterprise in comparison. Economic and cultural factors have also likely played key roles in the dominant role for text-based representation. Writing text is less expensive than making drawings or producing artistic pieces. Why are programs limited to text, and what directions can we follow to pursue path of artistic programs? To make software artistic, we need to endow software with *aesthetic diversity*. This implies looking to art for answers as to ways in which software can be made more useful, interesting, and comprehensive. Osborne [3] refers to this need for diversity as a widening of "our aesthetic horizons." We will henceforth use the adjective *aesthetic* to refer to "diversification of aesthetic choice" since, while text and 2D graphic-based software are in tune with a particular kind of aesthetic, this aesthetic is limited to fairly minimalist, abstract representation. The primary reason for the extant minimalism appears to be largely economic in character—that is, working in text is less expensive labor-wise. To the extent that the culture is affected by economy, the culture is amenable to change toward more aesthetic forms as the economy, associated with producing these forms, advances.

Aesthetic components and concepts are trying to make themselves known with software if the use of metaphor and analogy is an indication. Computing and software incorporate numerous metaphors [4, 5]. When we talk of program components, we speak of "looping around a section," "walking through code," "piping X into Y," "forking off processes (in Unix)," and "calling a sub-routine." Often, a mixed set of metaphors is loosely applied at the level of natural language, without making them concrete. For example, in Unix shells, one can pipe, fork, redirect and place jobs in the background. Little effort has been made to formalize and visualize these metaphoric constructs, since to do so would introduce extra overhead which was undoubtedly deemed prohibitively expensive when Bell Labs first gave birth to Unix in the late 70s. Interfacing with text would have to suffice. Most of these metaphors must live in the mental models [6] of their programmers' minds since they are rarely made concrete. While mental models represent important constructs, we should endeavor to free them from our minds, and to externalize them as real and virtual models, which have sensory and aesthetic qualities. Currently, most programming metaphors are trapped inside complex software representations and in individual mental models, which cannot easily be communicated. There are key aspects of software that rely on metaphor—namely, programming structures that have no obvious real-world equivalent. For example, states and events are conceptual, and so require metaphor to clothe them in aesthetics. In a distributed system, one might represent an ATM machine as something that looks similar to a real one, and yet ATM states and events do not have transparent, concrete equivalents. Fortunately, this situation opens the door to applying metaphor with a great deal of flexibility. A state can be a circle, sphere, plot of land, architectural space, or a partition in space-time.

As with most new techniques, the creation of artistic software and computing finds its first home in science fiction. In a November 1976 BBC episode of Dr. Who, entitled "The Deadly Assassin," the Doctor battles the Master in "the Matrix" which is a repository for all Time Lord knowledge [7, 8]. Similar *matrix* centered novels detailed the methods of cyberspace and virtual world manipulation, such as Gibson's Neuromancer [9] and Stephenson's Snowcrash [10]. These ideas culminated recently in the feature film "The Matrix", written and directed by the Wachowski brothers. Perhaps the longest running example of a matrix is found in the Holodeck of "Star Trek: The Next Generation" by Paramount Pictures. The concept of *program* in these media projects hints at programs as virtual objects and spaces, without there being anything formal or specific. However, it is hard to imagine Java being integral to programming the Holodeck; the ability to easily and quickly create 3D spaces suggests an altogether different paradigm for software development. Disney's film "Tron" presaged, not only the idea of a communicable, digital matrix, but more particularly, the virtual embodiment of aesthetic software. Users were represented by people who were similar to avatars in today's virtual space and software agent terminology.

2 From Program to Model

Programs have differing levels of detail, and translation among levels. The lowest level program is microcode. Continuing up the ladder of comprehensibility, we obtain assembly language and then hundreds of programming languages. Table 1 is pseudo-code that would easily map to one of these languages. This program is designed to allow people to edit text on a display monitor. This type of program is called a *text editor*. *Emacs*, *vi*, and *NotePad* are example text editors. Our program design begins with a *Main* entry/exit screen. From this screen, we transition into one of two modes: *Text* and *Cmd* (i.e., command). In text mode, the human enters text, as with a typewriter. In command mode, the user enters commands as to what to do with the text, such as cut, copy, and paste. There is a special *Macro* mode where a sequence of editing commands and data can be bound to

Table 1: Algorithm for a Text Editor

1:	*Main program entry/exit screen displayed*
2:	*If any key is pressed go to 3, otherwise go to 2*
3:	*Enter text mode*
4:	*Process text entered by user*
5:	*If key ESC is pressed, go to 7*
6:	*If key ˆM is pressed, go to 11, otherwise, go to 4*
7:	*Enter command mode*
8:	*Process commands entered by user*
9:	*If key ˆX is pressed, go to 1*
10:	*If key Q is pressed, go to 3, otherwise go to 8*
11:	*Enter macro mode*
12:	*Process macro entered by user*
13:	*If key $ENTER$ is pressed, go to 7, otherwise, go to 12*

an arbitrary key. Keys are those commonly found on a keyboard: ˆ M means control-M, ESC references the escape key.

Fig. 1 displays a two-dimensional graphic that represents a Finite State Machine (FSM). An FSM is defined as a model of dynamics, containing states and transitional arcs, where the model (i.e., machine) stays in a state until a condition on a transition becomes true. At such a time, the transition causes a change in state. Circles are states, and directed arcs are transitions from one state to another based on conditions that are labeled adjacent to each arc. The machine begins in the start state *Main*. The machine stays in *Main* until a key is pressed. Fig. 1 is a model of the software in Table 1, with the term "model" being defined, typically, as a visual representation of a program or system. It is a dynamic model since its components reflect the behavior of a system, as opposed to its physical or information-based structure. To the extent that art targets human senses, Fig. 1 shows promise over Table 1, and yet Fig. 1 yields a fairly iconic, diagrammatic display largely devoid of texture, sound and aesthetic content. Fig. 1 is a common diagram for the computer scientist, and most will recognize this form.

Figs 2(a) and 2(b) demonstrate two additional models for the editor software. The mapping of state and transition in Fig. 2(a) is sufficiently close to Fig. 1 that the mapping may be intuitively grasped. However, Fig. 2(b) is somewhat different. A *metaball* modeling approach (i.e., using liquid spheres that can attract each other, leaving connecting, organic bridges) was used to created oblong, directed spheres and a special welded join

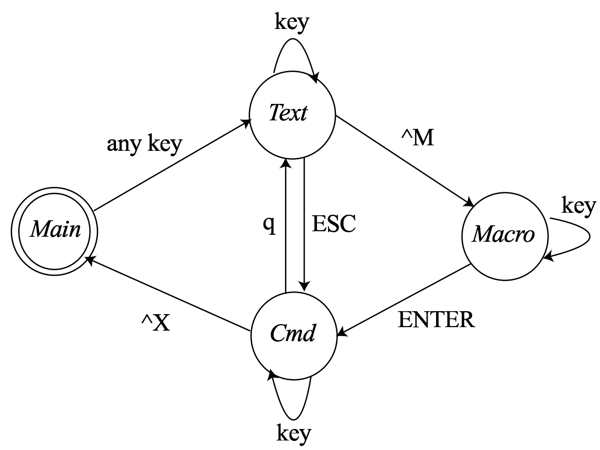
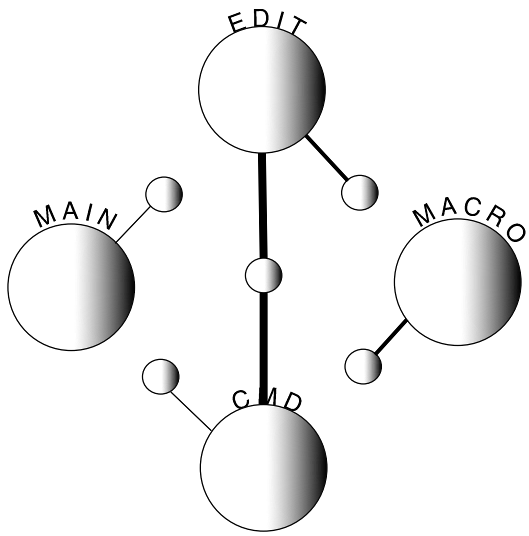


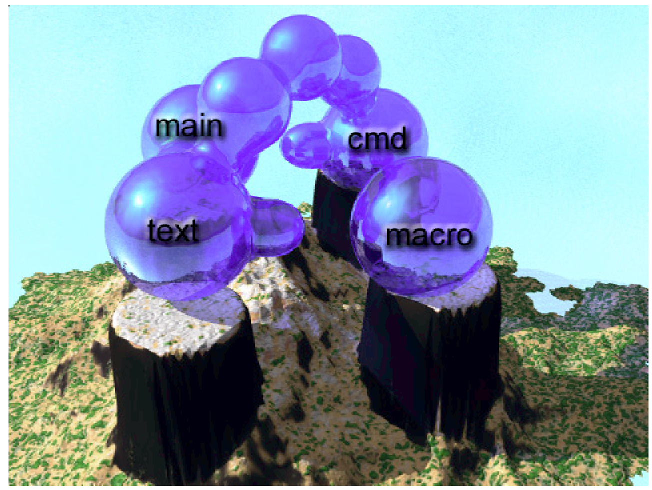
Figure 1: FSM with 4 states modeling a text editing program.

when a bidirectional set of transitions exist between states *text* and *cmd*. A landscape with towering butte-like structures is placed underneath the program, where each butte serves to demarcate a state. In these models, there are some notable differences with respect to Fig. 1, which is a *static* form. Figs 2(a) and (b) are manifested as *dynamic* graphical user interfaces so that not all information need be captured in a single perspective. By using the mouse and keyboard, and general navigational methods, additional models are viewed when interacting with the two models in Fig. 2. For example, in Fig. 2(a), the connecting lines change their width over time, designating the frequency of state transitions. In Fig. 2(b), state names only appear in response to a "mouse-over" event on the states. Fig. 1 fits within the *culture* and educational mindset of computer scientists, whereas new metaphoric mappings must be learned for Fig. 2. Nevertheless, it is common in modeling and programming for scientists to shop around for model types and metaphors and select the groups that best fit their cultural requirements. Modeling cliques are identifiable by the conferences they hold and journals that they edit; the scientists become adapted to a particular paradigm [11] to which they hold dear and find useful for their purposes. There is no singular best modeling approach. Fishwick [12] defines numerous formalisms and 2D designs for dynamic models.

Ugrankar [13] created a 3D architectural physical scale model of a virtual space to encode a six-state FSM that represents a dynamic model of water temperature dynamics in response to heat conduction from a plate. The left-hand most 4 states are identical in topology to Fig. 1, even though the semantics differ. This illustrates the powerful role of analogy in systems. Fig. 3(a) shows an overhead view of the model. The model has



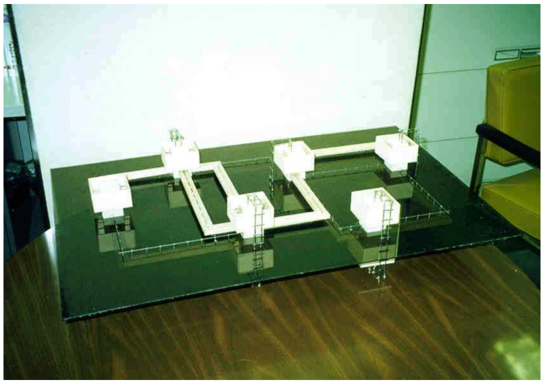
(a) 2D FSM



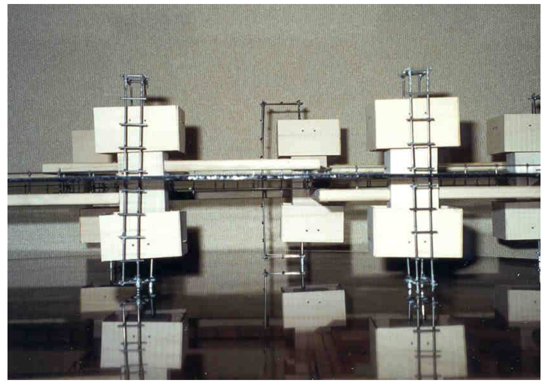
(b) 3D FSM

Figure 2: 2D and 3D model designs for the text editor. Created with Adobe Illustrator and MetaCreations Bryce.

cubical rooms and long corridors connecting rooms. The entire space is divided horizontally by a glass pane. Rooms, representing states, are present on both sides of the pane. Corridors on the top-half of the pane represent external state transitions, whereas corridors on the underside are internal state transitions. For temperature dynamics, internal transitions are those that occur because of a change of internal system state (i.e., temperature reaching a value and thus indicating a phase boundary), whereas external transitions occur due to causes external to the system (i.e., a knob being turned). Fig. 3(b) shows a closeup illustrating metal ladder-like constructs denote reflexive transitions where the dynamics results in the same state given the appropriate conditions. Conditions are marked with tape on each corridor. Avatars, that are not present in the physical model and which represent signals into the system, move along the corridors. This model was designed and created as a prototype for a virtual world using the Virtual Reality Modeling Language (VRML), which is our primary design language for *rube* worlds.



(a) Overhead view



(b) Closeup sideview of states.

Figure 3: Architectural space for 6-state FSM. Media: plexiglass, wood, metal bars.

3 The *rube* Project

In 1998 at the University of Florida, our research group initiated a methodology called *rube* [14, 15, 16], which helps modelers to construct models that incorporate aesthetics. The models are used to both model physical phenomena as well as to design programs designed via modeling. The following are the *rube* modeling steps:

1. *Choose system to be modeled:* In the case of a modeling initiative, a system is initially specified. For example, we know that we must model two trophic levels in the Everglades [17] if that is our physical target. In the case of software, there are two sub-choices. The first is one where we have a distributed system, where the model structure reflects the physical system component topology. The second choice is where we have a program to create with no apparent objects. These two sub-choices are usually part of the same software since even though a distributed system suggests objects and models, large chunks of software will require innovative design and frequent use of metaphor if we are to create models from them.

2. *Select model types:* We take the software and specify the formal dynamic model types to be used. Dynamic model types are plentiful and include automata, Petri nets, data flow networks, scripts, rules, and event graphs. This is the formal step of defining the nature of the dynamics in base model form—prior to exercising our

desire to employ metaphor and aesthetics.

3. *Choose an aesthetic:* Elements from the previous step must be mapped, via metaphor. The styles suggested by the aesthetic may be hierarchical. For example, the top level style could be architecture, and the second level style to reflect a particular type of architecture such as Classical Greek, Gaudi or Le Corbusier. If the style is painting, we could have Russian constructivist sketching or surrealistic painting in the style of Ernst, Dali or Magritte.

4. *Define Mapping:* Once we have defined a style, we must now carefully, and completely, create a mapping between the formal dynamic model type components and the stylistic components. For example, a collage sub-element of Ernst would map to a *place* of a Petri net if we had chosen the Petri net as our model type. It is critical to pay special attention in ensuring that the mapping preserves the full capacity of the formal dynamic model, otherwise the model will not be executable.

5. *Create Model:* This is the craft-worthy, artistic step of applying the mapping to create the model.

Regarding model-creation from programs, we now address the question of what programs are modeling. In the case of program construction for distributed systems, models are created of the objects and network composing the system. In the case of the non-distributed, ordinary program, we are free to create maps using the above methodology, by starting with model types. Here are guidelines based on programming elements:

- *Sequence:* Open path in space-time. Examples: (1) entity movement across paper following arc; (2) path through building; (3) evolution of entity or space over passage of time.

- *Condition:* Branch in space-time. Examples: (1) fork in a road; (2) portals leading from a room; (3) plant and tree branching.

- *Iteration:* Closed path in space-time. Examples: (1) race track; (2) closed circuit on a landscape.

- *Hierarchy:* Space-time hierarchy. Examples: (1) Scaling in space-time using physical encapsulation; (2) Positioning levels spatially via horizontal or vertical displacement.

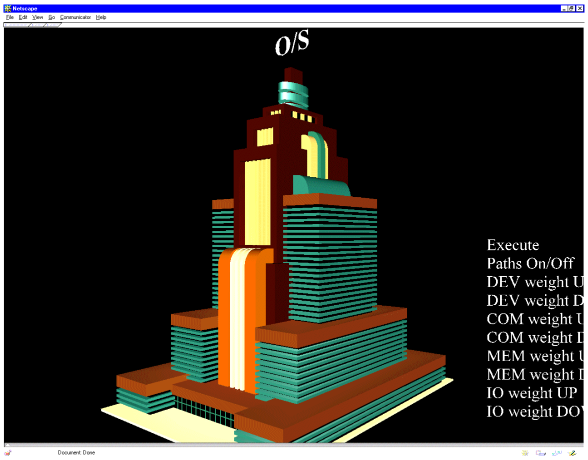
- *Recursion:* Self-referential Space-time hierarchy. Examples: (1) Zooming in and out of a point in space, or inside of an entity to find smaller, self-similar entities.

- *Assignment, Input and Output:* Human interaction using a sensor or tool. Examples: (1) measuring an entity; (2) using a tool to shape or color an entity.

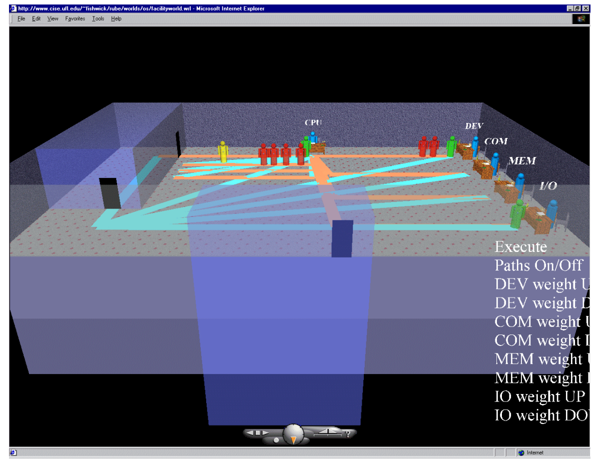
These guidelines are meant to be very general. We have just begun to explore approaches to mapping software to physical phenomena, therefore, these guidelines are purely heuristic in value. The categories reflect basic mathematical categories as well as broadly defined cognitive partitions, similar to those described by Arnheim [18, 19], Volk [20], and Ching [21]. They also roughly correspond to principles in programming languages [22]. *Space-time* is a multidimensional space with an orthogonal time axis, as in physics. Iteration and recursion are generally coupled with sequence and condition, and should be mixed where appropriate. One such example involves creating a hierarchy where loops are present, so that where a loop is located, a hierarchy is created through an encapsulating object. Consider a routine sorting or searching task, which tend to be highly iterative. Such a task can be wrapped into a physical *machine* that reminds us of the task. This wrapping represents the introduction of hierarchy.

Hopkins et al. [23] constructed an operating system kernel using VRML. An operating system is one of the largest pieces of software in the typical computer. It controls all peripherals and resources, allowing tasks to request resources, obtain service, and continue processing. A simplification of an operating system is where we model the average task (i.e., executable program) as requesting the Central Processing Unit (CPU), and then requesting a resource such as I/O or memory, and then iterating in this fashion until the program terminates. We employ a metaphor based on business workflow where tasks, represented as human agents, move on the floor of a building but stay within the boundaries of assigned tracks. This is not unlike waiting lines with guide posts and ropes, or colored strips placed on an airport floor, as a means of partitioning waiting lines. Our example task scheduler is a non-preemptive, dynamic priority scheduling system that contains tasks, four priority queues, and the following five types of physical devices with their associated queues: 1) CPU, 2) DEV (i.e., external device), 3) COM (i.e., communication, such as via the parallel, serial, and USB ports), 4) MEM (i.e., memory load/store), and 5) I/O (i.e., input/output, such as disk read/write). By employing a workflow metaphor, we map a task to a person and a device to a ”service facility,” which is a person behind a desk (see Fig. 4(d)). The priority and device queues map directly to physical space as waiting lines. Persons can travel over paths between the devices and queues.

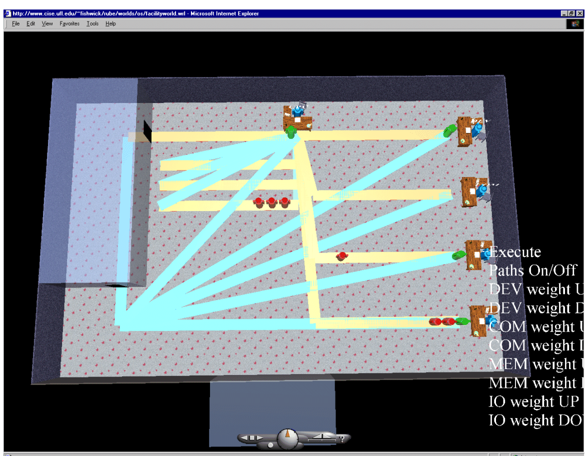
Fig. 4 displays different angles of the VRML world created with the rube methodology. Fig. 4(a) starts us outside of the Operating System, which is constructed as an art deco building. On one of the floors, we find the task scheduling ”workflow” identified by moving agents (see Figs. 4(b) through 4(d)). The aesthetics of the Operating System are fairly minimal, and represent an initial investigation into employing both music and 3D geometry in creating a program based on moving agents.



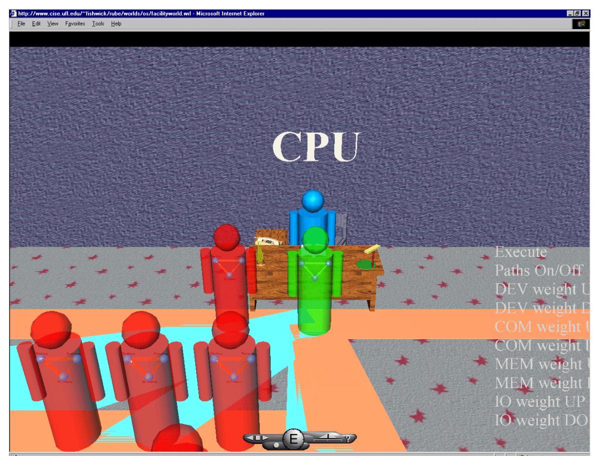
(a) View of operating system.



(b) Isometric view of operating system task scheduling.



(c) Overhead view of Fig. 4(b).



(d) Snapshot and zoom of CPU in action.

Figure 4: VRML browser window views of the the operating system from outside and on the task scheduling floor.

4 Related Work

The Unified Modeling Language (UML) [24] represents a good beginning in viewing software as modeling, but this view centers around software engineering, rather than programming. Programming is typically viewed as a low-level activity underneath the umbrella of software engineering. This view should change if we are to more clearly represent programs as models, while relegating textual programs to the status currently occupied by assembly language—a necessary, but low level construct. Many languages targeted at novices [25] are model-based. The Logo language [26] was one of the first languages based on the idea of programming through the use of a turtle [27, 28] capable of carrying out a set of simple instructions, with graphical feedback for output. Karel the Robot [29] has similar aims with a robot replacing the turtle and by extending the functionality of the moving agent with respect to its interaction environment. Methods of programming by demonstration or example [30] are structured on agent and rule-based approaches to software development. The idea is to program through modeling, and where there may not be a natural physical object in a program, a microworld can be created so that the programmer benefits from thinking in terms that are natural, memorable, and aesthetic rather than artificial, arcane and ugly. Several commercial products exist, such as *Stagecast* [31], which allow modelers to create simulations using graphically-specified production rules. Modeling is the activity that allows humans to better reason about programs. The overall areas of *software visualization* [32] and *visual programming* [33] have produced many similar successes to those of programming by modeling, example and demonstration. There are areas where the importance of visualization and sensory-feedback are stressed, such as Human-Computer Interaction (HCI) and Visualization. In HCI, we obtain example research involving visualizing information [34] and texts on interface methodology [35, 36, 37]. Employing visualization and metaphors permits the human to better interact with the computer. For visualization, there has been significant work in visualizing data, program execution and software. *Data visualization* is, perhaps, the most active field where scientific and engineering data are viewed from multiple perspectives, in 2D and 3D, using a wide variety of icons and color range. Brown [38] discusses methods for visualizing the execution of programs in terms of input/output. Shu [39] specifies a dichotomy where we have *visual environments*, referring to program and data visualization, versus *visual language*, referring to the actual creation of software using visual methods. Early visual software developments were catalogued in an edited volume by Chang [40] and recently in a special issue of Communications of the ACM [41] on "programming by example" where the goal is to make programming easier through simulation and demonstration via rules.

A quick search through the literature on keywords "3D" and "programming" tends to sprout forth numerous articles and books on how to render 3D graphics. By turning this idea on its head, we imagine that 3D itself is used to do the programming, rather than vice versa. The area of 3D program structure is closely related research to ours and represents a relatively new area that holds much promise, especially with new 3D web-based technologies such as Java3D and VRML. Programming in 3D had to wait for efficient methods for 3D programming, but significant work has been done. Lieberman [42] pioneered one of the first efforts in transitioning from 2D programming to using 3D elements. Najork [43] created the *Cube* language, with cubes representing program nodes and pipes for flow. Oshiba and Tanaka [44] built *3D-PP*, which contains regular polyhedra connected with lines for representing declarative, logic-based programs.

In reviewing *algorithmic art*, representing the former concept of software creating art, there are many players. Cohen's AARON [45] was a system for semi-automatic generation of paintings. Verostko [46] discusses the interrelationships between art and software from the standpoint of one (programming) being used to create the other (art). In an essay called *Programming as Art* [47], Verostko states "As we shall see, an algorithm may be an instruction for any kind of procedure. For the artist or composer this means that any kind of algorithmic inquiry on the nature of form is possible." The artist uses the computer program as any other tool. More mechanical, and less organic, procedures for creating art and form also exist [48, 49]. These formal methods tend to capture pattern-based generation of art using tools familiar to the computer scientist, such as production rules and automata. Maeda's *Aesthetics + Computation Group* [50] focuses on interactive demonstrations of this bridge area, often using Java applets for the underlying technology [51]. Gelernter [52] presents cogent arguments for alternate, and beautiful, representations for machines and computing. The emphases on HCI and software visualization, with a focus on the human interface, strikes a common chord with art. Differences come about mainly in the degree of creativity and metaphor employment afforded. An approach to visualizing arbitrary data or programs may involve a color landscape, but rarely does one see a rich landscape filled with color and texture. The primary reasons for this condition appear to lie with culture and economy.

5 Summary

Today's program in Java or C++ bears little resemblance to Fig. 4, and there are good reasons for this condition. Even though the state of the art, and information economy, fertilize the roots for ubiquitous 3D development, we have many legacy codes and tools, and the tools are nowhere near the point of general acceptance for aesthetic programming. A lot of further research is required, along with a gameplan for getting from A to B. Our work depends on a modeling framework, and even though significant efforts such as UML exist, free tools are not available as they are for Java, C and C++.

As if the problems of using graphical methods in programming were not enough, we also have an issue in the education of computer science students who grow up with text-based programming. Making the leap from fairly minimalistic, diagrammatic models and code to representations that encourage massive infusions of art is not going to be easy. The educational challenges are paramount, and represent a cultural gap for most computer scientists who view artistic representations as flourishing, syntactically excessive, and luxurious. The cost of convincing visual and auditory renderings is ever-decreasing, causing a revolution in the way that future generations expect to interact with the world. The cultures must be bridged and connected if aesthetic software is to succeed. In preparation for the game-console generation to enter University, and for their expectations of immersive environments, at the University of Florida we have created a new set of engineering programs in unison with our College of Fine Arts. These programs are called *Digital Arts and Science* [53] (DAS), and their aim is to educate a new breed of student who is as familiar with sketching, textures, sculpture, form and music as with data structures, discrete math, and translation theory. Our vision is that these students will have the aesthetic sensibilities to take advantage of the rapid technologies that now support fast 2D/3D graphics and audio.

Even though we began our studies with model construction with an extra dimension, it became increasingly clear that the primary thrust of this work was in aesthetics, since one can equally construct flat models that have meaningful and desirable qualities. Our modeling examples are still primitive, and no doubt professional artists can weave far greater patterns. Even when we consider 3D programming, work has been done before in this area, but what remains to be done for the future is tantamount to a renaissance in modeling—to free models, not just from flatland, but also from simple geometries. Much of the work in 3D programming languages, for example, implements 3D iconography, which needs to evolve into landscapes, cities, organic systems, physical architectures, and style-guided formations if the very idea of aesthetics is to take hold—the point being that it *does* matter whether one uses an octagonal polyhedron versus an art deco house. One is aesthetically-challenged and Platonic whereas the other promotes familiar sensory appeal.

Only recently, have computer scientists been able to economically create 2D and 3D graphical user interfaces. Even now, there are too few tools as compared with text editors. Two-dimensional graphics, while popular and accessible, is still somewhat of a struggle as compared with text. Some toolkits exist such as Java Swing and Tcl/Tk, but more are needed. Economy, therefore, drives programmers, and computer science in general, toward text for model representation. The VRML97 standard, bringing along with it easier methods for representing and sharing 3D, is barely three years old. Economy, however, is not the only hurdle to jump. As programming was originally borne out of mathematical representation, programmers tend to be culturally minimalist in matters of representation. The new economy of 3D, including game console capabilities and prices, is bound to alter this culture, but it may take considerable time. When presented with the representation of a cathedral to represent a program, the typical programmer questions the choice based on minimalism and material efficiency; why use a cathedral when a circle or Latin letter will do? The point is that we *do it* only out of efficiency. There is no absolutist intrinsic value in minimal representation; it is one of many aesthetic points in a magnificently large space of possibilities. In art, we see this in movements in works of those employing the minimal aesthetic—two examples being constructivism [54] and Bauhaus [55]; they represent wonderful aesthetics, but the potential for art is not limited to them. Neither should programming be limited by minimalist text and graphic symbol sets as our technological economies support more fulfilling interactive representations.

The study of aesthetic models in the representation of computer programs represents a small potential when compared with the more encompassing subject area of computer science. Models are used frequently in most sub-areas: databases, machine organization, organization and architecture, data and program structures, and artificial intelligence. These areas will all benefit from direct artistic influence. The same approach to aestheticism can be applied to other art forms, such as storytelling and theatre. Stories can be mapped onto model structures, and serve to entertain as well as to educate and remind us about the target system. Our implementation on models for aesthetic computing is directly inline with Alan Kay's definition of computer literacy [56]:

> "Computer literacy is a contact with the activity of computing deep enough to make the computational equivalent of reading and writing fluent and enjoyable. As in all the arts, a romance with the material must be well under way. If we value the lifelong learning of arts and letters as a springboard for personal and societal growth, should any less effort be spent to make computing a part of our lives?"

Acknowledgments

I'd like to thank the three anonymous referees, as well as Roger Malina and Rejane Spitz for their comments on an earlier draft of this paper. I would also like to thank students, past and present, for their continued efforts in *rube*: Robert Cubert, John Hopkins, Linda Dance and Taewoo Kim. The following are research sponsors for our modeling and simulation work: Air Force/Rome Laboratory (contract: F30602-98-C-0269) and the Department of the Interior (grant 14-45-0009-1544-154).

References

[1] Donald Knuth. *The Art of Programming: Volumes 1 to 3*. Addison-Wesley, 1968.

[2] Donald Knuth. *Literate Programming*. Stanford University Center for the Study of Language and Information (Lecture Notes, No. 27), 1992.

[3] Harold Osborne. *Aesthetics and Art Theory: An Historical Introduction*. E. P. Dutton & Co., Inc., 1968.

[4] George Lakoff and Mark Johnson. *Metaphors we Live By*. University of Chicago Press, 1980.

[5] George Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. University of Chicago Press, 1987.

[6] Dedre Gentner and Albert Stevens. *Mental Models*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[7] Terrance Dicks. *Doctor Who and the Deadly Assassin*. British Broadcasting Corporation, 1977.

[8] The Deadly Assassin. *Doctor Who Magazine*, (108):44–47, January 1986.

[9] William Gibson. *Neuromancer*. Ace Books, 1984.

[10] Neil Stephenson. *Snowcrash*. Spectra Books, 1993.

[11] Thomas Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 3rd edition, 1996.

[12] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1995.

[13] Prajakta Ugrankar. Finite State Automaton for Boiling Water Multimodel, May 2000. Independent Study (University of Florida).

[14] rube Project. http://www.cise.ufl.edu/~fishwick/rube, 1998.

[15] Paul A. Fishwick. 3D Behavioral Model Design for Simulation and Software Engineering. In *2000 Web3D/VRML Conference*, pages 7–16, February 2000.

[16] Paul A. Fishwick. A Modeling Strategy for the NASA Intelligent Synthesis Environment. *Journal of Space Mission Architecture (JSMA)*, (1):23–42, 1999. Center for Space Mission Architecture and Design, Jet Propulsion Laboratory.

[17] Paul A. Fishwick, James G. Sanderson, and Wilfried F. Wolff. A Multimodeling Basis for Across-Trophic-Level Ecosystem Modeling: The Florida Everglades Example. *SCS Transactions on Simulation*, 15(2):76–89, June 1998.

[18] Rudolf Arnheim. *Visual Thinking*. London, Faber and Faber, 1969.

[19] Rudolf Arnheim. *The Dynamics of Architectural Form*. University of California Press, 1977.

[20] Tyler Volk. *Metapatterns: Across Space, Time and Mind*. Columbia University Press, 1995.

[21] Francis D. K. Ching. *Architecture: Form, Space and Order*. Van Nostrand Reinhold Co., 1979.

[22] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley, second edition, 1987.

[23] John F. Hopkins and Paul A. Fishwick. Synthetic Human Agents for Modeling and Simulation. *Proceedings of the IEEE*, 2000. Submitted for publication.

[24] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[25] John F. Pane and Brad A. Myers. Usability issues in the design of novice programming systems. Technical report, Carnegie Mellon University, 1996. Report CMU-CS-96-132, http://www.cs.cmu.edu/~pane/ftp/CMU-CS-96-132.pdf.

[26] Seymour Papert. *Children, Computers and Powerful Ideas*. Basic Books, New York, 1980.

[27] Harold Abelson and Andrea diSessa. *Turtle Geometry*. MIT Press, 1980.

[28] Mitchel Resnick. *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1997.

[29] Richard E. Pattis, Jim Roberts, and Mark Stehlik. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley and Sons, 1994.

[30] Allen Cypher, Daniel C. Halbert, David Kurlander, and Ellen Cypher, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[31] Stagecast Software. http://www.stagecast.com.

[32] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[33] Tadao Ichikawa, Erland Jungert, and Robert R. Korfhage, editors. *Visual Languages and Applications*. Plenum Press, New York, 1990.

[34] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufman, 1999.

[35] Donald A. Norman. *The Design of Everyday Things*. Doubleday Books, 1990.

[36] Jakob Nielsen. *Usability Engineering*. Morgan Kaufman, 1993.

[37] Jef Raskin. *The Humane Interface*. Adisson-Wesley, 2000.

[38] Marc H. Brown. *Algorithm Animation*. MIT Press, 1987.

[39] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold Company, 1988.

[40] Shi-Kuo Chang, editor. *Visual Languages and Visual Programming*. Plenum Press, 1990.

[41] Henry Lieberman. Programming by Example. *Communications of the ACM*, 43(3):73–74, March 2000.

[42] Henry Lieberman. A Three-Dimensional Representation for Program Execution. In E. P. Glinert, editor, *Visual Programming Environments: Applications and Issues*. IEEE Press, 1991.

[43] Marc Najork. Programming in Three Dimensions. *Journal of Visual Languages and Computing*, 7(2):219–242, June 1996.

[44] Takashi Oshiba and Jiro Tanaka. 3D-PP: Visual Programming System with Three-Dimensional Representation. In *International Symposium on Future Software Technology (ISFST '99)*, pages 61–66, 1999.

[45] Pamela McCorduck. *Aaron's Code: Meta-art, Artificial intelligence and the Work of Harold Cohen*. W. H. Freeman, 1991.

[46] Roman Verostko. Epigenetic Painting. *Leonardo*, 23(1):17–23, 1990.

[47] Roman Verostko. http://www.verostko.com.

[48] George Stiny and James Gips. *Algorithmic Aesthetics: Computer Models for Criticism and Design in the Arts*. University of California Press, 1978.

[49] William Mitchell. *The Logic of Architecture: Design, Computation and Cognition*. MIT Press, 1990.

[50] Aesthetics + Computation Group. http://acg.media.mit.edu/.

[51] John Maeda. *Design by Numbers*. MIT Press, 1999.

[52] David Gelernter. *Machine Beauty: Elegance and The Heart of Technology*. Basic Books, 1998.

[53] Digital Arts and Science Programs. http://www.cise.ufl.edu/fdwi, 1999.

[54] Anatole Kopp. *Constructivist Architecture in the USSR*. St. Martins Press, New York, 1985.

[55] Margret Kentgens-Craig. *The Bauhaus and America: first contacts, 1919-1936*. MIT Press, 2000.

[56] Alan Kay. Computer Software. *Scientific American*, 251(3):53–59, September 1984.