Aesthetics of Computation—Unveiling the Visual Machine

Jared Schiffman
S.B. Computer Science and Engineering, minor in Mathematics
Massachusetts Institute of Technology
June 1999

Submitted to the Program in Media Arts and Sciences
School of Architecture and Planning
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences at the
Massachusetts Institute of Technology
September 2001

© Massachusetts Institute of Technology
All Rights Reserved

Author: **Jared Schiffman**
Program in Media Arts and Sciences
August 10, 2001

Certified by: **John Maeda**
Associate Professor of Design and Computation
Thesis Supervisor

Accepted by: **Stephen A. Benton**
Chair, Department Committee on Graduate Studies
Program in Media Arts and Sciences

Aesthetics of Computation—
Unveiling the Visual Machine

Jared Schiffman
S.B. Computer Science and Engineering, minor in Mathematics
Massachusetts Institute of Technology
June 1999

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, on August 1, 2001, in partial fulfillment of the requirements for the degree of Master of Science in Media Arts and Sciences at the Massachusetts Institute of Technology,

Abstract

This thesis presents a new paradigm for the design of visual programming languages, with the goal of making computation visible and, in turn, more accessible. The Visual Machine Model emphasizes the need for clear visual representations of both machines and materials, and the importance of continuity. Five dynamic visual programming languages were designed and implemented according to the specification of the Visual Machine Model. In addition to individual analysis, a comparative evaluation of all five design experiments is conducted with respect to several ease of use metrics and Visual Machine qualities. While formal user tests have not been conducted, preliminary results from general user experiences indicate that being able to see and interact with computation does enhance the programming process.

Thesis Supervisor: **John Maeda**
Associate Professor of Design and Computation
MIT Media Laboratory

Aesthetics of Computation—Unveiling the Visual Machine

Jared Schiffman

Thesis Reader:

Harold Abelson
Class of 1922 Professor of Computer Science & Engineering
MIT Electrical Engineering & Computer Science Department

Aesthetics of Computation—Unveiling the Visual Machine

Jared Schiffman

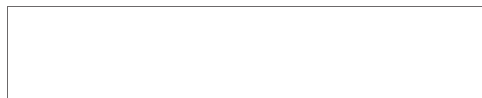
Thesis Reader:

Mitchel Resnick
LEGO Papert Professor of Learning Research
Associate Professor
MIT Media Laboratory

Aesthetics of Computation—Unveiling the Visual Machine

Jared Schiffman

Thesis Reader:



Whitman Richards
Professor of Cognitive Science, Media Arts and Sciences
MIT Artificial Intelligence Lab

Acknowledgements

The designs in this thesis are as much a product of my own work as they are of the work of the Aesthetics & Computation Group. The ideas contained herein are the result of five years of sharing a common space and a common ambition with a set of brilliant and creative designer/technologists:

Chloe Chao
Peter Cho
Elise Co
Rich DeVaul
Joy Forsythe
Ben Fry
Matt Grenby
Golan Levin
Bill Keays
Omar Khan
Axel Killian
Max Van Kleek
Reed Kram
Nikita Pashenkov
Casey Reas
David Small
Tom White

The fearless leader of this dynamic group is the source of inpsiration for all who dwell here. His words are jumbled. His designs are beautiful. Thank you, John, for everything.

This thesis would not be the document that it is now without the sincere effort of the three readers: Prof. Harold Abelson, Prof. Mitchel Resnick and Prof. Whitman Richards.

To my parents, thank you, for the quarter-century of love and support that you have always provided. For loving me for who I am and always being proud.

To Kate, my wife, so happy that you are.
For helping me forget that I was writing a thesis,
and for reminding me, even when I had not forgotten.
I love you.

Contents

Chapter 1 : Introduction
1.1 Two Rooms 15
1.2 Unveiling the Visual Machine 15
1.3 Making Programming Easier 16
1.4 Ease of Use 18
1.5 Contributions 20
1.6 Overview of Thesis 21

Chapter 2 : Context
2.1 With and Without Visible Computation 23
2.2 Text-based vs. Visual Representations 26
2.3 Modes of Synthesis 30
2.4 Graphic Design in Visual Programming 42

Chapter 3 : the Visual Machine
3.1 Motivation for Model 45
3.2 The Visual Machine Model 45
3.3 Two Elements 46
3.4 One Principle 48

Chapter 4 : Design Experiments
4.1 Turing 54
4.2 Plate 60
4.3 Pablo 64
4.5 Nerpa 68
4.6 Reverie 70
4.7 User Testing 72

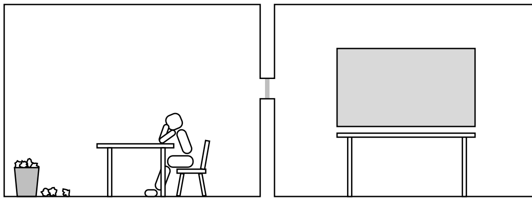
Chapter 5 : Discussion & Analysis
5.1 Successes 75
5.2 Challenges 77
5.3 Comparative Evaluation 79
5.4 Future Work 84
5.5 Conclusion 85

Appendix A : Prior Work 88
Appendix B : Issues and Justifications 100
Appendix C : Implementation 102

Chapter 1 : Introduction

1.1 Two Rooms

Imagine two rooms, separated by a wall which contains a small glass window at eye-level. A programmer is in one room and a computer (who might as well be a person) is in the other. [Searle, Turing] Standing at the window, the programmer communicates to the computer a long message which tells the computer what to do in its respective room. The computer then pulls a small curtain closed to cover the window. Minutes (or seconds) later, the computer returns to the window, draws open the curtain, and hands a message back to the programmer that contains the result of the computer's work. If the result is not correct, the programmer has no recourse but to rewrite the message and try again. However, if the curtain were never closed, the programmer could simply look into the computer room and see exactly when and where the mistake was made. Unfortunately, in modern programming environments, the curtain is always closed on computation.



1.2 Unveiling the Visual Machine

This thesis aims to improve the process of programming by introducing a new breed of visual programming languages and accompanying environments which make visible the computation that they create. These new entities are called Visual Machine Languages. While current text-based and visual programming systems enable the abstract specification of computation, they are incapable of displaying that computation. Visual Machine Languages, however, allow for both the specification and visualization of computation in a continuous graphical environment and with a single visual language. By making explicit the secret reality of computation which heretofore has been invisible, Visual Machine Languages could have a significant impact on the way that people create programs.

1.3 Making Programming Easier

Programming is the process of creating computation. As much as being a practice of science or engineering, programming is a craft. Like any craft, programming involves the manipulation of a medium, namely computation, by using tools, namely programming languages and environments. Computational media should be as malleable, vivid, and approachable as any craft media. Unfortunately, modern programming tools prevent computation from being accessible or even visible to the curious novice, not to mention the general population.

1.3.1 Programming Concepts are Simple

Surprisingly, the concepts which underlie programming are not difficult to understand. Explaining the ideas of sequential execution, conditionality, repetition, and proceduralization requires about one hour of teaching. Likewise, explaining the notions of variables, types, data structures, and even classes can be accomplished in an additional hour. These ideas can be understood quite easily when framed within a real-world context such as drawing, cooking or construction. Why then are semester-long college courses spent teaching students to utilize these programming concepts, when they could be learning about algorithms or proper programming practice? Perhaps, the answer to this question is that modern programming systems provide an incomplete set of tools and representations for dealing with computation as a medium.

1.3.2 Programming Systems are Incomplete

Programming, as it exists today, means specifying computation. The specification, however, is not the computation. In the same way that blueprints are not a building, and a score is not a symphony—code is not computation. It is essential to make this distinction.

Programmers spend all day looking at source code, but almost never see computation. When computation does occur, what the programmer sees is not the computation, but the output of the computation. It may be a window, or a button, or a string of text, but it is all just a remnant. It is not the computation. Oftentimes, when in search of a bug, a programmer will coerce

the computation to leave footprints in the output. Still, the animal is never seen. The absence of a means to view and interact with computation directly is why programming systems are incomplete.

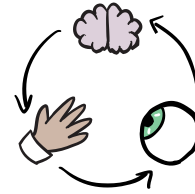
Debuggers are meant to reveal computation, but as most experienced programmers will agree, debuggers are hardly worth their weight in code. The reason that debuggers are generally ineffectual, is that they only show snapshots of computation in the past. These static images still require the programmer to infer what must have happened prior to the first snapshot and in between all the steps. Frustration is the general response to debuggers, since what the programmer actually wants is to watch the computation unfolding smoothly over time, changing slowly, gently, predictably and meaningfully, and being presented in an appropriate visual representation.

1.3.3 Programming with Visible Computation

Once computation is made visible and then interactive, programming will be a more complete experience. Programming will still involve the specification of computation. However, instead of having to "imagine the dynamic behavior of a program while…writing its static description" [Koike 1995], a programmer will be able to set a program running, and then actually watch and take part as the computation develops. Debugging may simply involve carefully watching for an unexpected change. Likewise, learning to program may become as easy as playing with a program and observing it in action.

Visible computation involves more than just observing computation. It also allows for direct interaction. For example, after beginning the execution of a program, a programmer may pause the computation, adjust a variable value or change a switching mechanism, and then continue the program running. At this point, computation will cease to be an abstract entity inside the computer, and will become a true medium to be sculpted like clay.

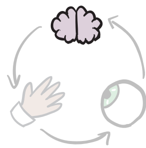
1.4 Ease of Use



Completeness for the sake of completeness is not a goal unto itself. The objective in creating a programming system with visible computation is ease of use. Ease of use may be dissected into three related topics: ease of comprehension (thinking), ease of construction (writing), and ease of consumption (reading).

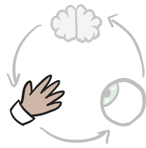
1.4.1 Ease of Comprehension

Designing a representation of computation which is simple to understand is the first step in developing a programming system which is easy to use. Ease of comprehension is an inverse measure of the amount of work that is required to move from one's understanding, having never seen a system, to one's understanding, having mastered the system. Certainly, the new ideas which are most easily understood are those that are most similar to one's preconceptions of how things should be based on prior experience. This model of understanding may explain why languages which are abstract (i.e. unfamiliar) often fail to capture the intuition of the user. More fleshed-out models of computation which behave more like real world systems should require less work to understand.



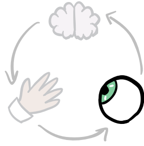
1.4.2 Ease of Construction

A programming language which is easy to comprehend may not necessarily be easy to create programs with. Attaining ease of construction is a much greater challenge than it may seem at first. Obstacles in the path of construction, such as arbitrary syntax in text-based languages, are the primary difficulty for most novices when learning to program. Even for professional programmers, errors of construction, in the form of syntax errors, are the most common type of errors. In addition to syntax, another factor in difficulty of construction is the type and size of the building blocks involved. Availability of an appropriate set of blocks can greatly enhance the ease of construction. For a domain-specific language, this means having domain-specific operations and data types. For a general-purpose language, there are a variety of potential solutions.



1.4.3 Ease of Consumption

Even a visual programming language which has both ease of comprehension and construction may still not be easy to read or consume. As Abelson and Sussman state, "programs must be written for people to read, and only incidentally for machines to execute." [Abelson] Readability is often an underestimated goal when designing a programming language. The goal of readability is not intended so much for the program creator but for the creator's colleagues who will later have to debug and extend the original code. While commenting one's code helps to solve this problem, being able to quickly navigate, parse, and interpret an entire program based solely on the code would be a dream come true for any software developer. Of course, the end goal of code consumption is code comprehension, which ties back to the original goal, ease of use.



1.5 Contributions

The goal of this thesis was to create new visual programming languages that advanced the field both aesthetically and in terms of user interaction. As a result of the work done in pursuit of this goal, this thesis delivers the following contributions:

1. Visible computation. Identification of the need for visible and interactive computation in order to deal with computation as a true medium.

2. The Visual Machine Model. This model represents a fully-articulated paradigm for the design of programming systems that make computation visible.

3. Five Visual Machine Languages. These five experimental instantiations of the Visual Machine provide a glimpse of how completely different the process of programming could be in the presence of visible computation.

1.6 Overview of Thesis

The following section is a preview of the remaining chapters of the thesis.

Chapter 2, Context, provides background information and support for the developments in Chapters 3 and 4. Chapter 2 begins with a look at programming as it exists today, without visible computation. The chapter continues with a discussion of the relative merits of textual representations versus visual representations. This is followed by a brief review of traditional models of computation and programming interfaces. The chapter closes with an analysis of the role that graphic design must take in the creation of visual programming environments.

Chapter 3, the Visual Machine, introduces the Visual Machine Model, which is a prescription for the creation of Visual Machine Languages. The Visual Machine Model requires the existence of a machine, a material, and a relationship between the two. The model also includes a design principle which stresses the importance of visual, interactive, and semantic continuity.

Chapter 4, Design Experiments, presents five Visual Machine Languages that were implemented according to the Visual Machine Model: Turing, Plate, Pablo, Nerpa and Reverie. Each of these designs takes a thoroughly different approach, both visually and conceptually, to fulfilling the Visual Machine Model. For each language, there is a description of the system's computational model, visual language, and method for demonstrating computation. Additionally, a sample construction is provided with each language.

Chapter 5, Analysis and Discussion, elicits both the successes within the design experiments, as well as the challenges and pitfalls which were encountered during the design process. This is followed by a comparative evaluation of all the designs, with respect to ease of use, and aspects of the Visual Machine Model. The thesis concludes with a look to the future and a brief summation.

The *Appendix* includes a presentation of select contemporary visual programing environments, notes on the Visual Machine implementations, as well as a few thoughts regarding the approach of the thesis.

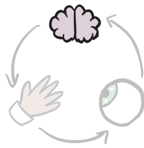
Chapter 2 : Context

2.1 With and Without Visible Computation

Stating that modern programming systems are incomplete may be considered heresy. Clearly, millions of people have already learned to program using current methods, and billions of lines of code have already been written. How could systems that enabled this possibly be incomplete? The answer to this question is suggested by the fact that so many problems associated modern programming could be alleviated if the computational product of programming were made visible. As Elizabeth Freeman notes, "viewing a program's execution is valuable when programmers need to correct bugs, improve performance, understand algorithms, and make updates." [Freeman p.305] This section examines how the inability of programming systems to display computation negatively affects their ease of use, as well as how having this capability in a programming system could greatly enhance the process of making programs.

2.1.1 Comprehending Computation

When attempting to program using modern programming systems, novice programmers (including those taking programing classes) construct their own internal representation of the computation through a process of trial and error. "If novice programmers are not given a model of a virtual machine, they may invent an inappropriate explanation, working from sources of information such as observing a debugger, extrapolating from tutorial code examples, or imagining the behavior of an internal agent." [Blackwell p. 246] If their learning process is successful, then their internal representation will be an accurate model of computation which can predict the behavior of a program based on its source code specification. For most programmers, however, this is a several year ordeal which is punctuated by moments of sheer frustration. Given this method of learning, arriving at a complete understanding of even one programming language is a major accomplishment.

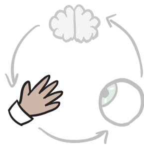


If programmers were able to view and interact with a cor-

rect model of the computation manifested via the source code, then the learning process would be completely different. Rather than having to construct their own model through trial and error, a novice programmer could simply observe the computation progressing smoothly and predictably on screen and adopt that model as his or her own. This process would be no different than watching a pin-ball machine in action in order to inform one's gameplay. From the beginning, a novice programmer would think in terms of the fundamentally dynamic processes involved in computation, instead of the static elements of the specification language. Not only would the presented model of computation be accurate, but it would also be the same model that is understood by all programmers who learned in this manner. This shared understanding might even engender effective communication between programmers.

2.1.2 Constructing Computation

Programming, as it exists today, is perhaps the only discipline in which the creator has essentially no view of the product that he or she is creating. While the specification of the computation is visible, and the output of the computation is visible, the computation itself is entirely invisible. Naturally, this makes constructing computation quite difficult. Since the output of the computation often reflects many lines of code, a mistake in just one line of code may not be noticed until it is manifested in the output at a much later point in time. This is a fundamental problem with the construction of programs.

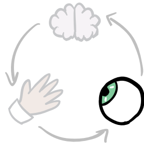


However, if the programmer were readily able to the view computation as it is occurring, then he or she would quickly notice that the mistake was made, since it would be visible at the exact point of error. Furthermore, if the computation was designed to be interactive, then the programmer might be able to fix the mistake on the spot, during the execution. In fact, one might even be able to construct an entire program during its own execution. This mode of construction might significantly reduce the number of mistakes that are left unchecked, since the programmer would always be in direct contact with the material being manipulated.

2.1.3 Consuming Computation

Even more difficult than constructing computation blindly is consuming invisible computation created by another person.

> *People who are confronted by a need to understand a program [written by another person] usually have only two alternatives: studying the source code, or running the program to see what it does. Ideally, a program would be understandable using one or the other of these methods; in practice, reading source code is impractically cumbersome for many programs, and construction of test cases to explain program behavior is often a tedious and speculative undertaking. [Jeffery p.4]*



Even code that is laced with comments and perfectly composed is almost impossible to understand without exerting a great deal of effort on the part of the reader. The code consumer must spend hours pouring over the code in order to reconstruct the mental map and other ideas that existed in the creator's mind at the time of creation. Simply determining the basic pathway that the computation takes through the code is an arduous task. These are the reasons why a software developing team of one can work much more quickly and efficiently than a team of three.

No doubt, making computation visible would greatly simplify the process of consuming computation. In order to determine the pathway of the computation, one could simply set the program running and watch as the flow of control smoothly makes its way around the space of the code. Repeating this process several times at finer levels of detail would give the consumer an excellent understanding of what the program is actually doing. One could also choose to run just a part of the code, and during execution, decide to insert values into the computation to see the effect. Providing an environment in which one can freely, quickly, and effortlessly play with and observe pre-existing computation could change entirely the way that programs are shared.

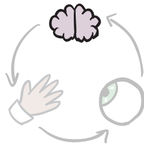
The following two sections present analysis and discussion of traditional design principles and techniques which are used in the creation of modern visual programming systems.

2.2 Text-Based vs. Visual Representations

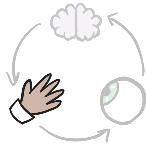
Text-based programming languages are now and always have been the most prevalent means of specifying computation. Despite twenty-five years of research in the field of visual programming, text-based systems have maintained their hegemony both in industrial and academic institutions. While these results are not encouraging for visual programming researchers, they continue with their mission because they believe that a visual representation of computation can be a superior representation to the text-based alternative. Certainly, there are instances in which each mode of representation is more appropriate than the other. Understanding when and why to apply each of the representations is a prerequisite for designing innovative and easy to use programming systems.

2.2.1 Comprehending Representations

Optimally, in order to improve understanding, the representation of computation that is presented to the user should resemble the user's prior internal representation. [Lindsay] "One reason for the difficulty ordinary people (nonprogrammers) have in programming computers is the conceptual gap between the representations their brains use when thinking about a problem and the representation they have to use in instructing the computer." [D.C. Smith p.330] In general, for most technical problems, including computation, people think in visual or spatial terms. Scientists, engineers and mathematicians tend to conceive of solutions to complicated problems visually, even when the final output is expected to be stated mathematically. Performing the translation process back and forth between one's internal visual representation and an external textual representation is bearable in mathematics where equations are at most a few lines long, but doing so in software development is a serious burden since even a short program can be a page long or more. Additionally, mathematical equations are meant to shared and read by people, while currently computer programs are written for the computer alone. Given the power of modern computers, it is absurd that humans still waste their clock cycles translating their thoughts to be easy for a computer to understand, instead of the other way around.



2.2.2 Constructing Representations



Theoretically, the process of composing programs using text could be a simple task. People have no difficulty writing long text documents in their native tongue, so why should writing a program be any different? If one could program in one's native language, then using text might be a natural choice for a programming system. The HyperTalk scripting language (Figure 2.2.2.1.) for the HyperCard development environment is so similar to English, that it is even readable by non-programmers. Unfortunately, even HyperTalk is constrained by its own syntax. Unlike any spoken or written language, even a minor syntax error in most text-based systems will cause all of the semantics to be lost (Figure 2.2.2.2.) While parsing the stilted

```
on mouseUp
   put field "input 1" into A
   put field "input 2" into B
   if the length of A is not the length of B then
      go to next card
   end if
end mouseUp
```

Figure 2.2.2.1. Example of code in the HyperTalk scripting language.

hierarchy of code with its intricate punctuation is a thoroughly taxing process, generating such code character by character is an even greater charge. Obviously, this is a weighty burden to put on any person. "Traditionally, the syntax of a source program bears no relationship to the runtime process that is the execution of that program. This is true for textual languages, and for the most visual languages as well." [Freeman p.305] Methods for composing programs which alleviate the problem of syntax are presented in the Section 2.3.

There are specific instances when text is extremely useful for constructing parts of a computation within a greater representation. When user-defined identifiers are necessary within a programming language, text is the

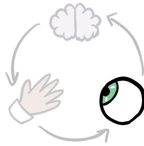
```
if (count==5)
   drawRectangle(10,10);

if (count==5)
   drawRectangle(10,10)

if (count==5)
   drawRectangle(10 10);

if (count=5)
   drawRectangle(10,10);

if [count==5]
   drawRectangle[10,10];
```

Figure 2.2.2.2. The first statement is syntactically correct. The other four are not.

appropriate representation specifically for those identifiers. Even when restricting names to a mere six letters, there are over 300 million possible identifiers available. Granted, most of these will be meaningless, but the point is that text-based identifiers remain trivial to generate, endlessly abundant, and easy to recognize. Creating visual identifiers is a much greater challenge. Since an identifier generally represents a specific concept, generating an appropriate name based on the concept's description is a much simpler task than generating a picture which conveys the same information.

2.2.3 Consuming Representations

While text is effective for the creation of identifiers, the sheer number of identifiers that are introduced, referenced and discarded within modern text-based program languages can result in programs that are incredibly difficult to follow. Every time that a name or identifier is used within a program, an internal reference is created. When many such references accumulate, an implicit web of connections is formed (Figure 2.2.3.1.) Unfortunately, in a text-based language, this complex multi-dimensional web is collapsed into a uniformly gray column of code. To unravel this tangled web, one must navigate the code along a one-dimensional axis, in the same manner by which one would undo a contorted knot.



```
(define (partial-sum i n e ee base)
  (- (quotient base (* i e))
     (quotient base (* (+ 2 i) ee))))

(define (a n base)
  (do ((i 1 (+ 4 i))
       (delta 1 (partial-sum i n e (* e n n) base))
       (e n (* e n n n n))
       (sum 0 (+ sum delta)))
      ((zero? delta) sum)))

(define (calc-pi base)
  (- (* 32 (a  10 base))
     (* 16 (a 515 base))
     (*  4 (a 239 base))))

(define (run)
  (display "How many digits of pi do you want: ")
  (let ((num (read)))
    (if (and (not (eof-object? num))
             (integer? num)
             (positive? num))
        (let* ((extra (+ 5 (truncate
                            (inexact->exact (log num)))))
               (base (expt 10 (+ num extra)))
               (pi (calc-pi base)))
          (display (quotient pi base))
          (display ".")
          (display (quotient (remainder pi base)
                             (expt 10 extra)))
          (newline)
          (run)))))

(run)
```

Figure 2.2.3.1. Illustration of the implicit web of references within text-based code.

The one-dimensional nature of text is by far its greatest detriment when representing computation. While computation may be reduced to one dimension, the nature of computation is that it is replete with multi-dimensional and non-linear structures. These not only include complex data structures such as hierarchical trees and circular lists, but also convoluted processes which diverge, jump, and twist. When represented visually, these kinds of structures maintain more of their natural form, and do not

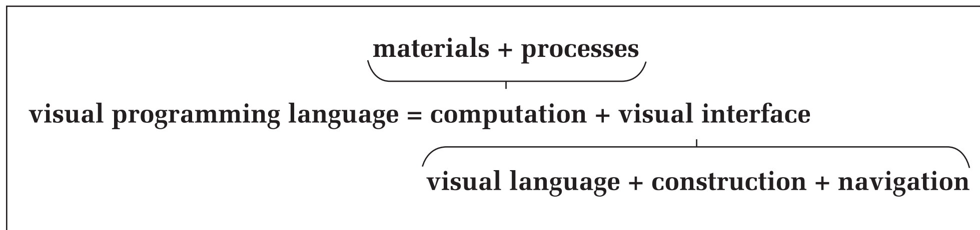
require a process of translation or reconstruction in order to be understood.

The limits of a representation are ultimately defined by the limits of human perception with respect to that representation. One comes to programming with pre-existing set of skills for processing both text and images. The processing of text relies on one's ability to read and understand written language. Written language is an extremely compact but powerful means of representation, since most of the information being conveyed is already stored in the mind of the reader. When processing text, the mind references the meaning of each of the language elements and constructs a new idea from the combination of many previously understood ideas. Unfortunately, most text-based programming environments stray too far from the standard syntax and vocabulary of written language and fail to capture any of this incredible intelligence. Instead, the user must learn an entirely new language, with a foreign syntax and strange vocabulary.

The processing of visual imagery is quite unlike the processing of text. While a person may choose not to read a block of text, he or she cannot turn off the neural mechanism which processes images into concrete objects and relationships. This process happens almost instantaneously and requires essentially no effort. The result of the primary visual encounter may not necessarily be a complete understanding, but will at minimum establish a framework for understanding finer details and events within the space. Additional visual processing can occur almost as quickly. For example, one can consciously pick out all of the blue objects in an image simply by scanning it from corner to corner. "Visualizations can expand [human] processing capability by using the resources of the visual system directly. Or they can work indirectly by offloading work from cognition or reducing working memory requirements for a task by allowing working memory to be external and visual." [Bonar p.16] Programming systems which exploit this natural visual intellect can shorten considerably the path to understanding.

2.3 Modes of Synthesis

This section presents a framework for the design of traditional visual programming languages. Every visual programming language consists of a particular computational model and a particular visual interface for specifying programs in that model. Computation can be generally cleaved into two related entities: computational materials and computational processes. A visual interface for specifying computation requires three ingredients: a visual language, a method of construction, and a manner of navigation. The two computational components and the three components of visual interface are examined here in terms of their respective modes. Rather than being strict taxonomies, these modes represent the most common instantiations of each of the components. By selectively combining these modes, one may synthesize the basic structure of almost any visual programming language.



2.3.1 Computational Materials

The various types of data that are processed by a program comprise the materials of computation. The computational materials available within a programming language are governed by the intended use of the language. Naturally, domain specific languages (such as Macromedia's Lingo or Pixar's Renderman Shading language) deal with a domain-specific set of materials. These materials will often have their own set of primitive operations which are built directly into the syntax of the language. For example, in the Pixar language, colors may be added and subtracted as if they were numbers. In addition to special material types, domain-specific programming languages usually include the standard materials of general-purpose programming systems.

computational materials	
General Purpose	**Domain Specific (eg.)**
number (int)	color
number (float)	sprite
boolean	account
character	transaction
string	equation
array	imaginary number
list	particle
composite	etc.

General purpose programming materials vary little from one language to another. Standard material types include numbers, boolean values, characters, strings, and arrays or lists. The ability to formulate higher-level composite structures is an essential aspect of any full-fledged general purpose programming environment. Of course, once data abstraction is possible, any kind of material is theoretically representable. Still, having an extraordinary material as a basic primitive of the language can make writing programs for the system a much smoother process. Furthermore, in a visual programming language, only primitive materials are visualized in their native representation, while constructed materials are visualized in terms of their parts.

2.3.2 Computational Processes

The modes in the domain of computational models have been examined in many prior theses [Travers 96], so the discussion here will remain brief and relevant to this thesis. The five most common computational models are: imperative, functional, procedural, object-oriented, and constraint-based.

As its name suggests, the imperative model deals with the execution of commands. The computer behaves as an active servant which can execute any sequence of commands in order. These commands generally direct the servant to act upon a set of computational materials which are always at the servant's disposal. These intransient materials are called state, and their existence is the most significant feature of the imperative model. The Turing Machine was one of the first imperative models of computation.

Unlike the imperative model, the functional model of computation exists without state. This does not mean that the functional model is without materials, but rather that the materials which are used are transient. The functional model is based around functions which transform (or map) input values into

computational processes

imperative
execution of sequential commands on intransient material (state)

functional
calculation of mathematical functions that map inputs to outputs (without state)

procedural
combination of imperative and procedural in which functions can affect state

object-oriented
simulation of interactions between objects that contain state and procedures

constraint-based
continual maintenance of truth of logical statements and rules

output values. These values are the materials of a functional language. One interesting aspect of a functional programming language is that every expression within the language results in a value. (This is not true of imperative languages in which a command may change the state but have no output value per se.) In a purely functional programming language, an entire program consists of one function, which may be the composition of several sub-functions. Any materials which are not included or referenced by the outputs of a function are lost after the computation occurs. Church's Lambda calculus served as the basis for the design of modern functional languages.

The procedural model of computation combines the imperative and functional models and is the model used by most modern programming languages. This model introduces the procedure, which is essentially a function, complete with inputs and outputs, but with the added ability to perform commands that change the state. With the inclusion of state, one can no longer rely on a function, when given the same inputs, to return the same outputs. When using a procedural programming language, one may choose to use a subset of the language to write a purely functional program, or to write a purely imperative program. For almost two decades, procedures were the dominant means of abstraction in software development. Most of the visual programming languages created for this thesis utilize the procedural model.

Over the past decade, the procedural model of computation has given way to the object-oriented model, which builds on the

procedural model. The salient feature of object-oriented computation is the object, which is a collection of related data and procedures. By associating procedures directly with data structures, the object-oriented model provides a very strong basis for the simulation of real-world events. In fact, the origin of object-orientation is in research on languages for simulation. These original models were called actor-based and then later agent-based and message-based models. The object-oriented model which grew out of these experiments is currently the dominant model of computation for software development. While procedures are still used within objects, it is the objects that now provide the primary means of abstraction.

The last model of computation, the constraint-based model, is not an extension of previous models, but an entirely separate model unto itself. When programming in a constraint-based language, one specifies a series of statements or expressions that must remain true throughout the course of the program's execution. For example, one may specify the constraint that a square must have equal length sides. Then, during execution, if one of the sides is changed by the user, the other side will adjust so that the statement remains true. In order for constraint-based systems to run, they must constantly compare the current state to the set of constraints and find solutions to make the whole system true. Needless to say, this is computationally intensive. A less general and less demanding method of constraint is the "predicate-consequent" statement, since the solution is always in the code. Constraint-based systems which rely on "predicate-consequent" statements are called rule-based systems. The most well-known constraint-based language is Prolog.

2.3.3 Visual Languages

While there appears to be a wide variety of design styles across the breadth of visual programming languages, almost all systems employ one of a few common modes of visual language for specifying computation. They are containment-based languages, connection-based languages, and matching-based languages. For completeness and clarity, two text-based programming languages will be presented alongside these visual languages. These text-

	scheme	C	control-flow *containment*	control-flow *connection*	data-flow	matching
statement	`(set! A 2)`	`A = 2;`	A = 2	A = 2	2	? 2

sequence	`(begin`
	`(set! a 2)`
	`(set! a 4)`
	`(set! a 6))`
		`a = 2;`
		`a = 4;`
		`a = 6;`
			a = 2
			a = 4
			a = 6
				a = 2
				a = 3
				a = 4
					2 4 6	? 2
						2 4
						4 6

conditional	`(if (= a 2)`
	`(set! a 4)`
	`(set! a 6))`
		`if (a==2)`
		`a = 4;`
		`else`
		`a = 6;`
			a==2 ?
			a = 4 a = 6
				a==2
				a = 4 a = 6
					4
					6
					2?
						2 4
						!2 6

repetition	`(do`
	`(i 0 (+ i 1))`
	`(sum 0 (+ sum i))`
	`((= i 5) sum)))`
		`while( i<5 )`
		`{`
		`i++;`
		`sum += i;`
		`}`
			while i<5
			i++
			sum+=i
				i<5
				i++
				sum+=i
					+1
					<5
					+	

procedure	`(define`
	`(proc a)`
	`(+ a 2))`
		`proc(int a)`
		`{`
		`return a+2;`
		`}`
			proc a
			a+= 2
			return a
				a+=2	+2	

based examples should be viewed as visual entities as well.

The two text-based languages examined here are Scheme and C. Most text-based programming languages visually resemble one of these two languages, since they represent two basic modes of syntax. Of the two, Scheme (Figure 2.3.3.1) has the much simpler syntax. Scheme utilizes a strict prefix syntax (e.g. "(+ 1 2)") and relies solely on parentheses for punctuation. Multiple sets of nested parentheses define the visual appearance of scheme code, which is often afflicted by a build-up of parentheses at the beginning and end of the procedures. Most scheme editing environments perform automatic indentation of the code, such that more deeply nested statements appear farther to the right.

```
(define (fact n)
   (if  (= n 0) 1
        (* n (fact (- n 1)))))
```

Figure 2.3.3.1. Scheme implementation of factorial.

C code (Figure 2.1.2.2) has a more complex texture than Scheme code due to C's varied use of punctuation in its syntax.

C statements are terminated by semicolons. Functions are called using parentheses with arguments separated by commas. Arrays are indexed with square brackets. Structure elements are accessed with periods and sometimes even arrows (“->”). In fact, every punctuation character on the keyboard, with the exception of ‘@’ and ‘$’ can be found in the standard C syntax. Punctuation provides boundaries for the most common visual structure in C code, the block, which is formed by enclosing code inside a pair of curly braces. These nestable blocks, which resemble Scheme’s nestable parenthetical statements, are used to define data structures, functions, conditional statements, and loops.

```
int fact( int n ) {
   if ( n==0 )
      return 1;
   return n*fact(n-1);
}
```

Figure 2.3.3.2
C implementation of factorial.

The nestable parenthetical and block structures of Scheme and C provide the basis for the containment-based visual language. Simply by treating the pairs of parentheses or curly braces as corners of a rectangle, one already has the basic elements of a containment-based visual language. The most well-known of many containment based systems is the Nassi-Schneiderman diagram (Figure 2.3.3.3), which adds a few enhancements to the simple rectangular model. Containment-based systems were born out of a desire to make programs more structured, and less spaghetti-like. [Glinert p.152] The enforced structure of this visual language prevents it from representing some computational processes.

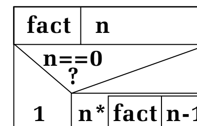


Figure 2.3.3.3
Nassi-Schneiderman implementation of factorial.

Connection-based visual languages offer a more free-form and less structured visual language for specifying computation. All connection-based systems have two basic components: nodes and connections. In general, nodes are points or shapes which exist in a two-dimensional plane, and connections are lines or tubes which connect the nodes to each other. Flow-charts are one the most common forms of connection-based system. They are used as a basis for the two dominant visual programming models to use a connection-based visual language. Glinert notes the relative merits of the flow-chart specifically:

Flowcharts seem to have been around forever. The unique strong point of this representation is that the meaning of a simple flowchart is immediately clear even to those people who have never seen one before. The disadvantage, of course, is that for larger programs flowcharts tend to turn into the proverbial "spaghetti ball" which masks logical structure.
[Glinert p.142]

Control-flow diagrams (Figure 2.3.3.4) were pioneered by Von Neumann and were the first visual representation of computation to make use of a connection-based visual language. These diagrams are semantically similar to containment-based diagrams and are based upon on an imperative model of computation. Hence, they focus on controlling the sequence of commands that are executed by the computer. Specifically, nodes contain commands to be executed, and connections guide the flow of execution from one command to the next. "Control-of-flow diagrams are usually preferable when emphasis is to be placed on the agents (things performing acts) of a computation than on the objects (data) being manipulated." [Glinert p.176] Process structures such as branches and loops are especially well-represented by control-flow diagrams.

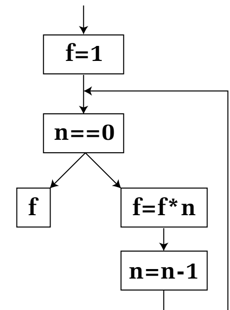


Figure 2.3.3.4
Control-flow implementation of factorial.

Data-flow diagrams (Figure 2.3.3.5) are the other significant model of computation to use a connection-based visual language. Data-flow systems are more reflective of a functional model of computation. Nodes in a data-flow diagram represent functions with inputs and outputs. Unlike text-based functional languages, data-flow functions can easily support multiple outputs, in addition to multiple inputs. Connections in a data-flow diagram serve as pipelines through which data flows from the output of one function to the input of another. Of all the visual models of computation examined so far, data-flow diagrams are the only systems which provide an explicit place for the display of material data.

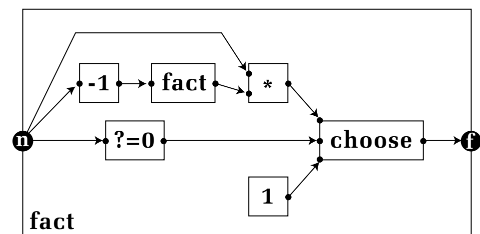


Figure 2.3.3.5
Data-flow implementation of factorial.

There is one final visual language that is quite different from the modes described above. Matching-based system are generally used to specify rule-based computational systems. These

visual language use two-column tables which contain inputs or predicates in the left column, and outputs or consequents in the right column. This two column structure may be rearranged visually to be more compact, but will still have the same basic topology. Much like data-flow diagrams, matching diagrams are focused on the material data used within a computation.

2.3.4 Construction Modes

Once a visual language of computation is established, the programmer must have some way to compose or construct programs using this language. This section presents four standard modes of construction: free-form text editing, syntax-directed text editing, free-form visual construction, and structure-directed visual construction.

The oldest but still most popular environment for creating programs is the free-form text editor (Figure 2.3.4.1.) With this mode of composition, the programmer creates a text-based program letter by letter. The main input device is the keyboard, with occasional use of the mouse to select and move text. One program may be spread across several files which are accessed as separate and distinct entities.

```
if ( (getFuncCall(A)==NULL
{
    printf("connection inv
    return;
}
```

Figure 2.3.4.1. A free-form text editor.

In a standard free-form text editor, a programmer may just as easily type “Mary had a little lamb” as he or she could type a syntactically correct program. More to the point, free-form text editors allow documents to be created which are not, in fact, runnable programs. This is the case because the level of representation of the text editor is “beneath” the level of representation of text-based programs. Programs are not made of individual letters. They are made of words and numbers and punctuation marks. These parts are called syntactic tokens and they are the building blocks of syntax-directed editing environments. A syntax-directed editing environment (Figure 2.3.4.2) can be thought of as a text-editor which automatically constrains the programmer to only construct syntactically correct programs. Since the editor has knowledge of the syntax the language, it can also suggest programmatic statements based on the current state of the program composition. Szwillus recalls the general benefits of syntax-directed editors:

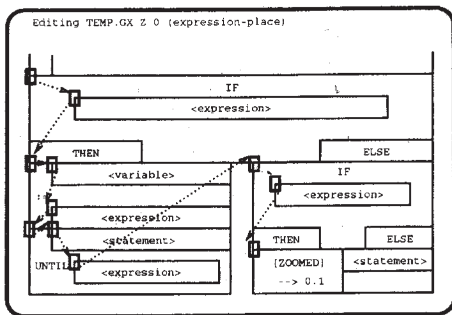


Figure 2.3.4.2 A syntax directed text editor.

Programmers at all levels of expertise can benefit from the typing time saved, the formatting, the error detection and, most significantly, the ability to conceptualize at a higher level of abstraction rather than focusing on the language details. Since the user interacts with editor in terms of language constructs, the means of expression is, in theory, closer to the programmer's understanding of the task. [Szwillus p.3]

Since syntax errors are the most common type of programming error, the features of a syntax-directed programming environment would seem to be a windfall for all programmers. Despite this logic, syntax-directed editors are viewed as tools for novices, and as too burdensome for professional programmers.

Obviously, a text-editor is the wrong tool for programming in a visual language. A drawing program may seem like the most natural interface for doing such programming. One could reference pictures of commands in a book and freely draw those commands on screen using a pen-based interface. In fact, there are several experimental visual programming languages which allow the programmer to do just that. A great deal of research has focused on recognizing hand-drawn computational diagrams. Even if this research succeeds, however, free-form drawing-based editors will have the same problem as free-form text editors: one may just as easily draw a picture of Mary and her little lamb, as draw a syntactically correct program. No doubt, the compiler would be confused.

The equivalent of syntax-directed editors for visual programming languages are structure-based editors. Structure-based editors deal with concrete visual objects, rather than hand-drawn pictures. The objects often take the form of literal building blocks, which may be found in a small onscreen library or may be generated through context-sensititve menus. Construction with these language objects is usually as simple as dragging and dropping them into place. Connections may be formed by drawing a line from one object's ports to another. Objects that work together may visually suggest their compatibility through their color or shape. Structure-based editors may also have built-in visual syntax checkers which can prevent the construction of a non-functional program. Most visual programming languages and all of the languages created for this thesis are structure-based visual program editors.

2.3.5 Navigation Modes

During the process of construction and afterward, a visual program must be observed and examined. Since all but the smallest of programs require more space than is available on one screen, there needs to be a means for navigation of the program space.

In a text-based or other one-dimensional representation, the most common means of navigation is scrolling (Figure 2.3.5.1.) Scrolling simply involves moving in a linear and continuous fashion about a vertical space. In a two- or three-dimensional system, scrolling becomes panning. When panning in a three-dimensional space, the motion of the pan must always be lateral (i.e. in the same plane as the image.) Scrolling and panning allow a document of any size to be viewed one screenful at a time.

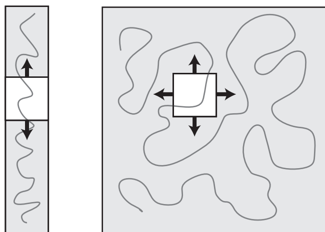


Figure 2.3.5.1.
Scrolling and panning.

In addition to panning, the other standard navigational technique is zooming (Figure 2.3.5.2.) In a one or two-dimensional system, zooming refers to the magnification of the onscreen image. Zooming may be used to examine the intricacies of a program, or to the view the whole program at once. In a three-dimensional space, zooming can be accomplished by moving the virtual camera along the direction perpendicular to the image plane. Obviously, when the camera is further from the objects it is capturing, those object will appear smaller in the image. Likewise, objects will appear larger when the camera is closer to them.

Panning and zooming are both linear and camera-relative navigational techniques which are independent of the space in which they exist and the objects within that space. Object-based

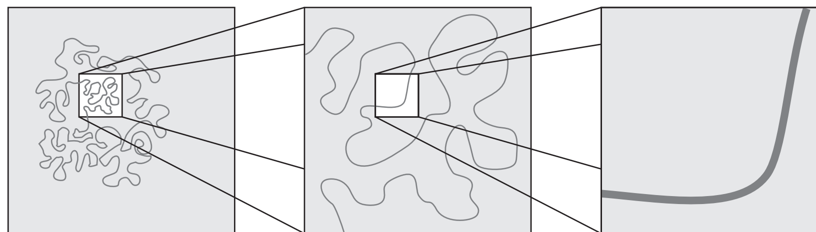


Figure 2.3.5.2. Zooming with three levels of magnification.

and path-based navigation (Figure 2.3.5.3) provide two alternatives to the aforementioned methods. These methods rely on the notion of a virtual camera which may be positioned about or directed towards a specific point in virtual space. Object-based navigation allows the user to move the current view with respect to objects which appear on screen. For example, one may center the camera on a selected object, and then rotate about it a full three-hundred sixty degrees. Alternatively, path-based navigation allows the user to move the camera along pre-defined paths through the virtual space. These paths may be defined by the user or may be an inherent part of the system.

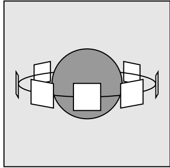


Figure 2.3.5.3. Path-based and object-based navigation.

One especially significant instance of object-based navigation is the navigation of hierarchical structures (Figures 2.3.5.4, 2.3.5.5.) [Glinert p.174] Certainly, this is an intuitive combination

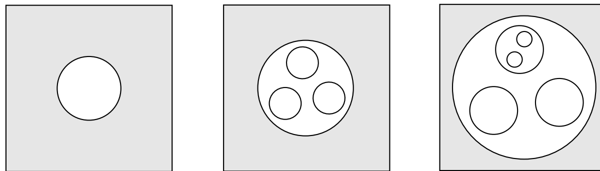
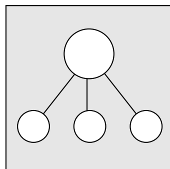


Figure 2.3.5.4. Hierarchical navigation through a containment-based hierarchy.

of organizational structure and navigational technique since it has been used by almost all personal computer users for the past decade. The general manner of hierarchical navigation is to begin at a root node and then progressively open and close nodes to reveal and conceal the child nodes. Semantically, this process of opening nodes is similar to zooming inward, since smaller and smaller pieces of the structure are revealed. Likewise, one may think of viewing just the root node, as viewing the entire program. Since one may simultaneously have a view of the top-most node down to the many leaf nodes, hierarchical navigation remains an extremely efficient means to navigate a potentially enormous information space.



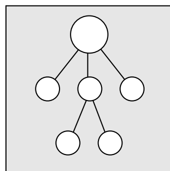


Figure 2.3.5.5. Hierarchical navigation through a connection-based hierarchy.

2.4 Graphic Design in Visual Programming

Ironically, graphic design has played little part heretofore in the field of visual programming. One need look no further than the covers of the major journals in the field to discover that graphic design is not a primary concern. Given that visual refinement has not been an objective in the past, it is necessary to answer the question of why aesthetics is a worthwhile goal for visual programming languages now. There are three reasons: improved communication, improved working environment, and improved perception of programming.

It is no secret that better visual design enhances the overall effectiveness of a message being conveyed. Newspapers, magazines, websites, and television programs all employ professional graphic designers to better communicate their message. As another visual medium, computer programs deserve no less. In fact, programs deserve even more attention since they are continually generating new information and conveying it to the user. "Fundamental to the concept of visual languages is the conviction that diagrams and other visual representations can aid understanding and communication of ideas." [Ichikawa p.121] Rather than being designed by a person directly, these visual representations must be formatted by the computer in real-time. For this process to be effective, one must establish clear, coherent, and cogent visual language. This visual language should be designed such that it will remain clear, coherent, and cogent in all modes of presentation.

Design is also employed to make environments more comfortable and pleasurable to be in. Few people would enjoy working in a bright red room, with bright green floors, and bright blue furniture. Yet this is the exact type of environment which has existed for many visual programming languages. The other extreme is the conservative black and white environment, which abandons color altogether. Of course, color is just one of the visual properties which contributes to the feeling of a space. Proper use of color, scale, form, and composition of visual elements helps to create a space in which one will be both contented and comfortable.

Design can also be used to change the perception of the programming process. More than being a practice of engineering or science, programming is a craft. Like any craft, programming

has its associated set of tools which make the craft possible and represent the process of the craft to the external world. Most people do not know how to make pottery, but they know about a potter's wheel. Likewise, most people do not know how to make horseshoes, but they know about anvils. Painters have brushes, chefs have pots, and weavers have looms. Programmers have a computer, and on the computer screen they have code, which lies flat and incomprehensible. Unlike the potter's wheel, anvil, brush, pot, or loom, code has no allure for the non-practitioner. When people see most craftsmen (or craftswomen) at work, they tend to stop and watch as if they were watching some kind of magician. No one has ever stopped to watch a programmer write yet another 'for' loop. Design can change this. Design can make the beauty and charm of computation apparent to non-programmers by altering the tools of the trade.

Chapter 3 : the Visual Machine

> *Few programming environments have addressed the problem of how to represent a program and its execution in an integrated way. Often, there is no relationship between the representation of a program and its execution: either the program execution is not represented at all and we are only shown the results, or the system uses two different visual vocabularies for source and execution…A visual vocabulary that can represent both programs and executions can make understanding program behavior simpler and allow debugging in the same environment in which the program was created.* [Freeman 1995]

> *Novices learn computer programming much more easily when physical or mechanical models for computation are suggested than when such analogies are not presented. This supports the notion that a graphical programming language, in order to be easy to learn, should bring a mechanical or physical model to mind.* [Glinert 1990]

3.1 Motivation for Model

Since there is no commonly held notion of what computation should look like, one may look to the world of physical machines for inspiration. Humans have, through their experience with objects in the physical world, an extraordinary learned ability to understand simple mechanical processes. While an eight-year old may not instantly understand the idea of gear ratios, he or she will have an intuitive understanding of how and why gears work. The thought that one gear could turn without the other is an impossibility in the eight-year-old mind. The cause and effect are subconsciously coupled through vision. Tapping into this intuition is the goal of the Visual Machine Model.

3.2 The Visual Machine Model

A Visual Machine Language is a visual programming language and environment which is able to visibly demonstrate through a dynamic machine-like process the detailed progression of the computation that it creates.

One may also think of a Visual Machine Language as a set of reconfigurable machines which act on a set of accompanying materials, and which may assembled to form larger, more complex machines. A language may be derived from the arrangement of the machines, in which the allowable machine interconnections form

the syntax, and the machine operations form the semantics.

The Visual Machine Model is an abstract recipe for designing Visual Machine Languages. This recipe consists of two essential ingredients and one design principle.

3.3 Two Elements

A Visual Machine Language requires the existence of both material objects and machine objects, and a well-defined relationship between the two.

3.3.1 Materials

The materials of computation are rarely seen during the process of computation. These materials are seen as they enter the computation as inputs and as they exit the computation as outputs, but are almost never actually seen while the computation is occurring. Much like real-world materials, these computational materials are susceptible to change. In fact, computational materials are much more resilient than their physical counterparts. Never will a computational material crack, or melt, or even become scratched. Amazingly, computational materials tend to offer infinite malleability without any perceptible damage to the material itself.

The visual representation of computational materials need not be constrained. There are, however, some factors to consider when designing such a representation. The most important aspect of a visual representation is that it be able to adapt to change. Since computational materials are constantly taking on new values, the visual representation must change as well, while still maintaining its identity as a singular object. Another factor to consider is compactness. Creating representations which can accommodate one thousand objects is much more difficult than doing the same for one object. One last consideration in the design of computational materials is the way in which materials can be connected and combined. Materials which support simple recombination can be used in a more fluent manner, than those which can only exist in isolation.

3.3.2 Machines

A computational machine is an object which processes a computational material. Generally, this processing involves making a change in a material. The simplest change a machine can make (short of doing nothing) is to make the same change in every material it encounters. More complex machines have controls which affect how the machine operates and in turn how the material is changed. In the real-world these controls are usually set by humans who operate the machine. In the computational domain, however, most controls are connected to other machines. Specifically, there are a class of machines which generate a machine response when exposed to materials configured in a specific way. This response can be used to control other machines. These response-generating "reader" machines are essential to the computational process since they allow information to flow back into the system of machines. They complete the feedback cycle.

Obviously, the visual representation of the machine is key to the success of a Visual Machine Language. The highest objective of a Visual Machine is to demonstrate clearly to the viewer that the machine is in fact making a change in a material. In a sense, Visual Machines are the antithesis of the black box. Unlike the famous black box, which hides all of its internal workings from the user, a Visual Machine should clearly show exactly how its internals operate. One should be able to see, step by step, how the material is changed between input and output. Of course, at some level, there will be machines which are indivisible—which have no visible internals. These machines should be explicit about exactly what changes are being made to the material, and to the utmost degree, suggest how that change was enacted.

3.3.3 Materials and Machines

The relationship between computational materials and machines is defined by the distinction between the two. Machines are active. Materials are passive. Machines will not act unless there is material to be acted upon. Likewise, materials will not change unless they are acted on by a machine. Hence, materials and machines are mutually dependent upon each other. This dependency should be made clear within a Visual Machine Language.

3.4 One Principle

The one design principle of the Visual Machine Model is: that all changes in both machines and materials must happen continuously in time. Maintaining continuity in a computational environment is no simple task, since computation is quantized into clock cycles, and since changes may occur discretely during program execution. Control jumps from function to function. Variables change in the blink of an eye, and objects are created from nothingness instantaneously. Hence, the notion of continuity goes against the grain of computation. Still, continuity is essential to the visualization of computation, for without it, one will quickly become lost in the process.

There are three aspects of continuity which must be addressed: visual continuity, interactive continuity and semantic continuity.

3.4.1 Visual Continuity

Visual continuity refers to the gradual and uninterrupted changing of the visual properties (color, form, position, etc.) of both materials and machines over time. Visual continuity is necessary since the human eye is not configured to follow discrete transitions. "Fundamentally, smooth continuous animations…can help…portray the individual operations of a process or algorithm. Gradual updates allow people's visual systems to easily perceive and understand the changes. The updates also provide context and facilitate tracking of patterns and actions." [Kehoe] Therefore, when designing a Visual Machine Language, an attempt should be made to visualize discrete transitions as if they were continuous. Transforming discrete processes into continuous processes is a central tenet of this thesis.

As a prerequisite for visual continuity, all machine and material objects must exist in a unified visual space. A visual space is unified if there is one viewing context, and if all objects within the system have a clear spatial relationship to all other objects in the system and are accessible to the user. A unified visual space enables the continuous processing of an entire program without a single visual context switch. Instantaneous visual context switches (e.g. those in a windowed environment) destroy any semblance

of continuity which may have existed prior to the switch. When such a switch occurs, one must spend several seconds or even minutes reorienting oneself and reestablishing the spatial/semantic mapping that existed before. Since such perceptual ruptures do not occur in a unified visual space, navigation becomes a much more free and natural process for the user.

Another condition of visual continuity for Visual Machines is that there be a single visual language used in the visualization of both specification and execution. Changes in visual vocabulary detract from visual continuity since the user must remap their current understanding to the new representations. In order to minimize this sort of discontinuity when moving from the specification to execution, the visualization of the execution should develop directly from the actual elements of the specification. This continuous visual transition is possible since there is a single graphic language in use. Additionally, a "uniform visual vocabulary simplifies the process of analyzing a program's behavior and eliminates the necessity of learning a completely new set of commands to examine and debug a program execution." [Freeman p.305] An overall graphic coherency to a Visual Machine Language also adds to the system's predictability.

3.4.2 Interactive Continuity

There are two basic modes of interaction for any programming language: composition and execution. In a Visual Machine Language environment, switching between these modes should be a simple and continuous process. Even during execution, the composition tools and the machine itself should remain interactive, so that changes can still be made while the machine is running. Additionally, one should be able to control the speed of execution, including being able to stop the machine and then set it back running. This continuity of composition and execution is designed to make the entire process of creating a program (writing, running, debugging, etc.) a less fractured experience.

3.4.3 Semantic Continuity

Semantic continuity refers to the perceived logical flow of a programming system. Semantic continuity is lost when a Visual Machine behaves unpredictably or illogically. For example, an object should not instantaneously appear or disappear, or ever appear in two places at once. If multiple representations are used to represent a single piece of information, then all of those representation should be encapsulated into a single object. Finally, in objects which do change over time, there should be some part of the object which remains stationary. Well-defined concrete objects are an essential part of a well-oiled Visual Machine.

During execution, semantic continuity depends on the convincing presentation of cause and effect. Establishing a sense of cause and effect is one of the most difficult tasks when designing a Visual Machine. By preceding all events with a clearly visible cause, the viewer will be more prepared to perceive and understand an event when it occurs. Even if the cause is simply a notifier of imminent change, having a moment to refocus in the right location, will greatly enhance one's perception of a smooth transition. If cause and effect are used effectively throughout the system, one will be begin to predict when and how changes will occur.

For a visualization of computation to attain the greatest degree of semantic continuity, it must be convincing. The user must believe that the processes that are unfolding on screen are actually happening. This does not mean that the visuals needs to be photo-realistic or involve virtual reality. Rather, a better model of synthetic, convincing images is a feature-length animated film. While the viewer of these films knows that the images are not "real," this knowledge never interferes with the viewer's belief in the characters, or understanding of the plot. In the same manner, a Visual Machine must be convincing in its presentation of the interaction between machine and material.

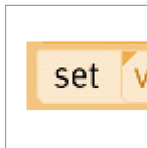
Chapter 4 : Design Experiments

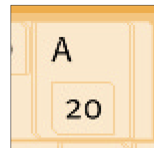
The Visual Machine Model defined in Chapter 3 lays the foundation for the development of the five Visual Machine Languages presented in this chapter. These various design experiments serve to establish a broad domain of systems which attempt to fully manifest the Visual Machine Model. Each of the designs makes use of a unique set of machines and materials to enable computation.

Machines

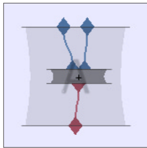
Materials

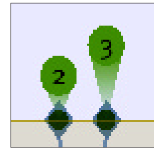
The first Visual Machine, Turing, is a reinterpretation of Alan Turing's theoretical model for computation, the Turing Machine. The Turing Machine's concise computational model and natural visual representation make it an excellent choice for a first Visual Machine Language.



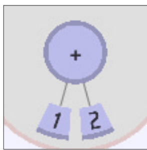


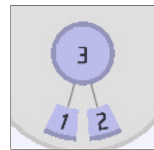
The second Visual Machine, Plate, is based upon a traditional text-based programming language. Plate separates the textual language into distinct syntactic elements and places each into its own plate object. By treating the text in a graphical manner, Plate enables novel means of program construction and consumption.



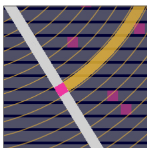


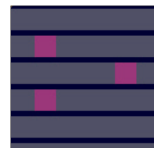
Pablo, the third Visual Machine, draws upon the vertical functional data-flow structure established in Prograph. This implementation is unique in that it actually displays the data that flows through its connection paths. Pablo is also notable for its fluid display of the calling of functions.





Nerpa, like Plate, is based on a text-based language. In Nerpa, however, that language is functional rather than imperative. Nerpa functions are represented as hierchical polar structures, which consist of code expressions on flat cards. These cards may flipped to reveal an evaluated value on their back side.





Unlike the four prior Visual Machines, Reverie is a programming system specifically built for a gallery exhibition. This design experiment attemps to connect references of addreseses in memory with nonlinear jumps in the computational process. The Reverie machine can be programmed with a single click.

As a set, these five designs provide an inital platform for testing the Visual Machine conjecture: that visible computation can improve the ease of comprehension, construction, and consumption of modern computational systems.

4.1 Turing

Turing is a Visual Machine implementation of Alan Turing's famous theoretical model of computation, the Turing Machine [Turing]. Turing was created as an exploratory first attempt in the domain the visual programming languages. While the Turing Machine is far from an efficient means of computation, Turing himself proved that theoretically any computation which could be completed mechanically (i.e. by any computer), could be completed by the Turing Machine. Hence, the Turing Machine is a universal computer. The combination of universality and conciseness, as well as the inherently visual model, made the Turing Machine an excellent choice for a first visual programming language.

The computational material of the Turing Machine is an infinitely long one-dimensional tape which consists of an infinite number of successive symbols from a finite alphabet. While the Turing Machine specification allows for any alphabet of symbols, the implementation presented here uses capital letters of the standard Roman alphabet (Figure 4.1.1). As a material, the symbol tape (Figure 4.1.2) seems quite spare, even though it can theoretically represent any structure. In this respect, the material resembles the eternally reconfigurable binary material of modern computers. Despite the limitation of the Turing alphabet, the tape is quite malleable. Any symbol on the tape may be changed to any other symbol during execution. Theoretically the tape has a beginning, but no end. Of course, a real-world Turing tape must eventually end due to the limitations of the host computer's memory.

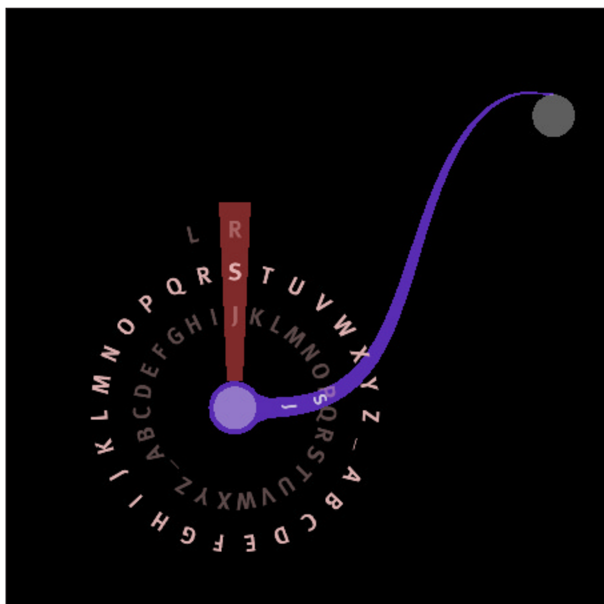


Figure 4.1.1. Three symbols must be chosen for every link.

Although the formal conception of the Turing Machine includes the tape as part of the "machine," this discussion will

treat everything that is not the tape as the machine. Technically, the machine is a finite state diagram. It consists of a set of abstract points within the program, which are called states, and a set of links between the states, which are called transitions. The states are visually represented in Turing as circular nodes placed by the user on a two-dimensional plane (Figure 4.1.2). There may be any number of states in a Turing machine. Traditionally, the transitions are simply represented as curved arrows which connect one state to another. In Turing, these transitions still connect one state to another, but the arrow has been reshaped into a more organic form (Figure 4.1.2). Every transition contains three pieces of relevant information: an input symbol, an output symbol, and a direction for the tape to move. The input and output symbols are chosen from the same alphabet as the one used for the tape. The direction is either left or right.

Constructing a Turing program has the feeling of drawing a computer. The screen begins with a blank black slate except for the half-inch high tape which extends from the center of the screen to the right. When the user clicks the mouse, a translucent circular state is formed at the location of the cursor. Once two or more states are formed, the user may then hold the mouse down over any state and begin drawing a transition to any other state. The shape of the transition is primarily determined by the user's gesture, but also by an internal smoothing mechanism. The result is a modified drawing experience in which the transition is a live and active entity. Once the mouse is let up, the transition is frozen into place. The user may then program the three elements of transition, by clicking on the transition form, and selecting the input and output symbols as well as the direction from a circular menu of options (Figure 4.1.2). Independent of the dexterity of the user, the result is an organic

Figure 4.1.2.
Turing's visual elements.

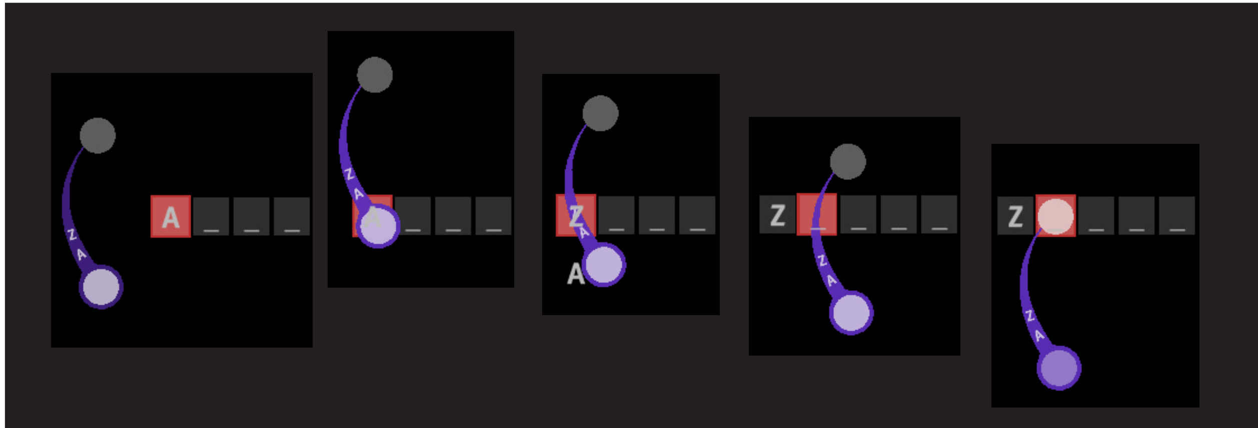


Figure 4.1.3. Computation of one Turing transition.

blue and green web of connections. Since the Turing canvas is an essentially infinite two-dimensional plane, the user is free to draw out the Turing program so that it reflects his or her own mental map of the program space. This may include the visual clustering of functional units, and other self-imposed organizational techniques (Figure 4.1.4). The user may navigate this computational landscape via standard panning and zooming techniques.

Prior to this point, the process has been one of specification. Here is where the computation becomes visible. The mechanical operation of the Turing Machine is relatively straightforward. The machine begins with the tape rewound to its beginning which appears at the center of the screen. A start state which is chosen by the user lies on top of the first tape symbol. When the machine is run, it examines the first symbol on the tape and the input symbols of all the outgoing transitions from the start state. If the there is a match between the symbol on the tape and any of the input symbols, then that transition is selected, and the process of transitioning begins (Figure 4.1.3). During the transition process, the machine changes the symbol on the tape to the output symbol of the selected transition, and then proceeds to fluidly move the tape left or right depending on the direction indicated. Since the machine is always aligned with tape, one perceives that the machine is changing the material directly. While all of these events are occurring, the machine smoothly propels itself to a new position, such that the state at the end of the selected transition becomes aligned with the next symbol on the tape. Both the current symbol on the tape and the newly appointed current state

now lie at the center of the screen, and the process repeats itself. The current tape symbol is compared to the input symbols of the outgoing transitions, a match is found, and the transition begins. If no match is found, the machine simply halts.

Running one's Turing program simply involves clicking on the play button at the bottom of the screen. If the transitions are properly configured, the machine will begin to dance about the tape, following the transitions, with the locus of execution always centered squarely on the screen. Meanwhile, the tape smoothly shifts left and right to a regular rhythm. As the symbols on the tape are changed, the previous symbol falls off of the tape and then off the screen. While the machine is running, one may adjust the speed of execution or even bring it to a temporary halt. When the program ultimately halts, the machine literally stops, and one knows the execution is finished. One may, however, create loops in the machine, which may cause it to run forever. As with any control-flow diagram, these loops simultaneously exist both as visual and semantic entities. As Turing proved in the 1930's, there are some Turing programs for which one can never know if they shall eventually halt or run forever.

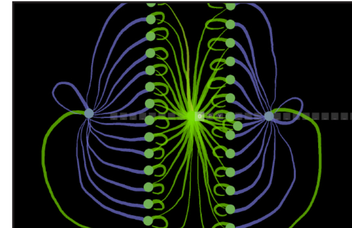
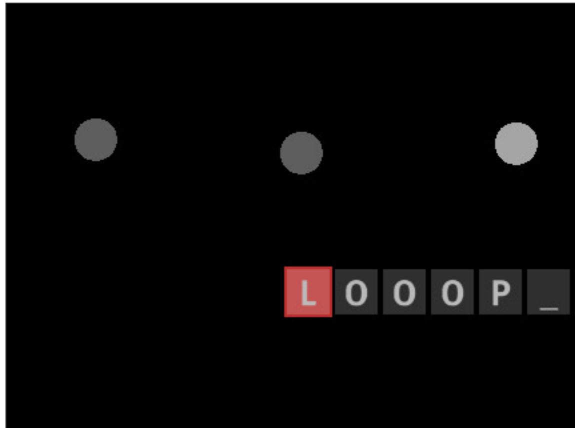
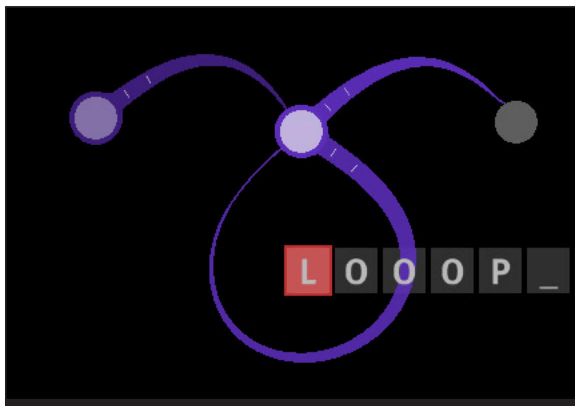
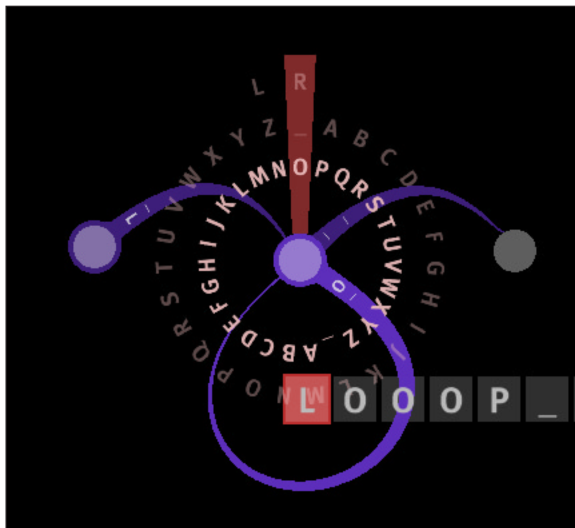


Figure 4.1.4.
An organizational technique.

Turing Example A

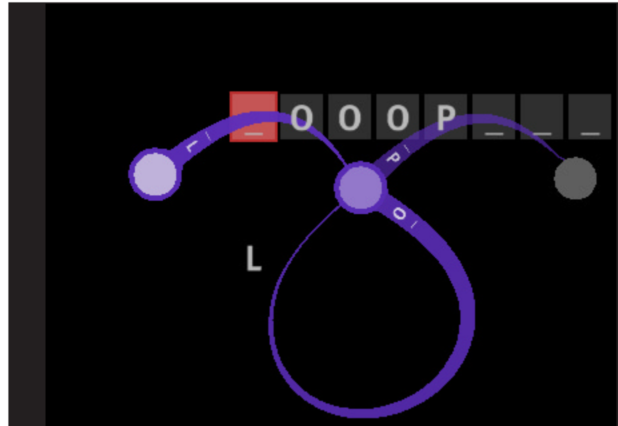


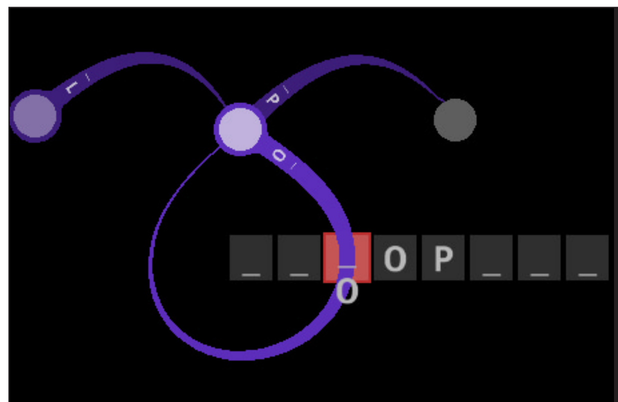


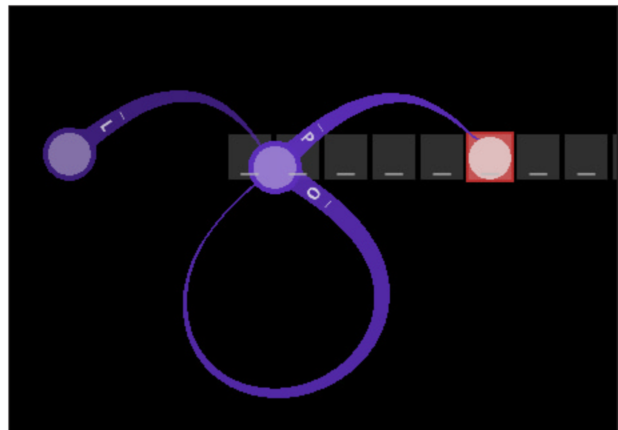


The three images above show the construction of a Turing program which recognizes an 'L' followed by any number of 'O's, followed by a 'P'. For example, "LP", "LOP", and "LOOP" are all recognized by this program.

The three images below show the progressive execution of the Turing program constructed in the images on the left. As each letter is found, it is replaced by a blank space. When the execution is complete, the tape will only have blank spaces.

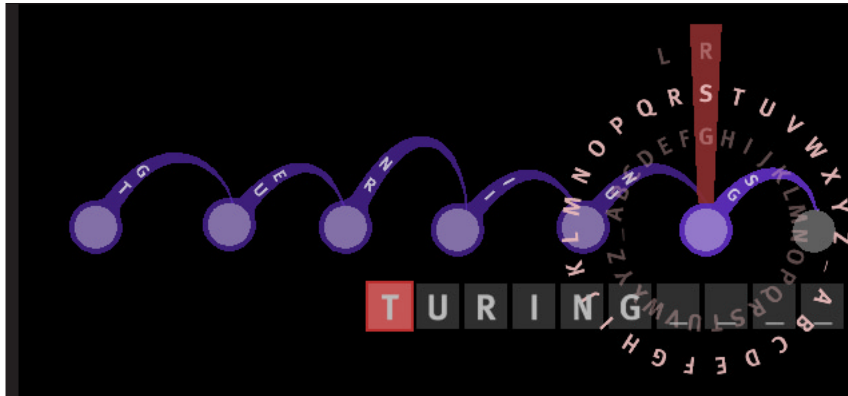


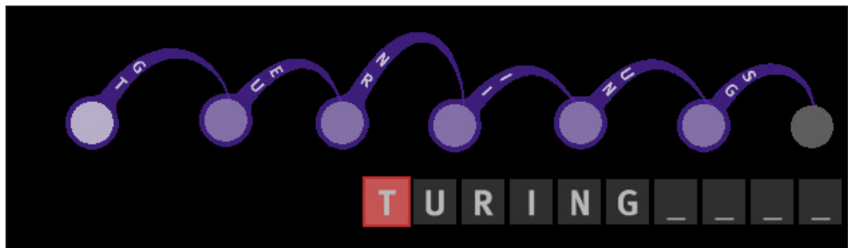


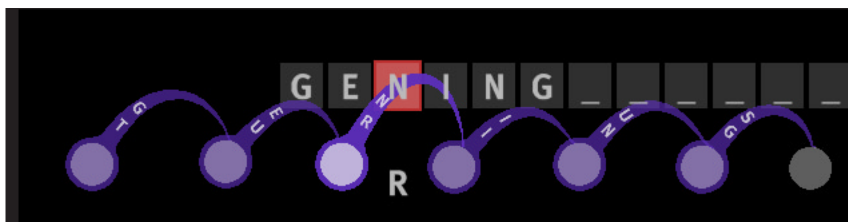


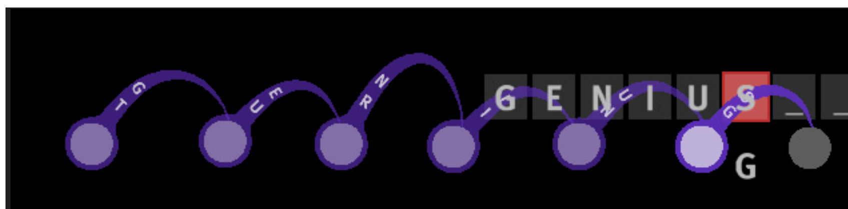
Turing Example B

The first three images display the construction of a Turing program which changes the word "TURING" to the word "GENIUS". The last two images show the execution of the program.









4.2 Plate

Just as Turing began with the Turing Machine model, Plate began with a traditional text-based programming language as it basis. The concept behind Plate is to make the syntactically distinct elements of a language into independent graphical objects, called plates, which can then behave in a machine like manner during execution. This textual machine processes computational material which is visually located within the code itself. Unbeknownst at the time of creation, Plate is an example of a syntax-directed editing environment.

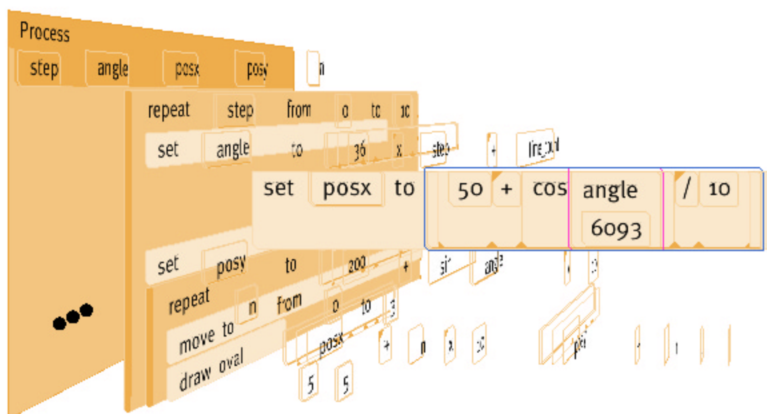


Figure 4.2.1. During step-by-step execution, the program layers are folded back.

The material which is processed by Plate is the standard set of computational data types of a modern programming language. These include numbers, characters, boolean values, strings, and lists. While Plate could be expanded to handle more complex data types, these were the only supported types at the time of publication. The visual manifestation of these materials is a straightforward textual represenation. Given the graphic context of Plate, representing the materials in a more visual manner would not be difficult to implement.

The machine components of Plate are the text-based program plates (Figure 4.2.2). Plate is based upon a very simple imperative text-based programming language similar to Pascal. Every element of the language receives its own plate, for a total of about ten distinct types of code plates. These code plates act like templates or forms into which other plates are placed. For example,

there is an assignment plate which reads "set _ to _" and which accepts a variable plate on its left slot and an expression plate, constant plate, or function-call plate on its right slot. There are block-oriented plates such as the "if _ then _ else _" plate which accepts a truth-valued plate in the first slot, and any number of function or assignment plates in the second and third slot. All plates exist inside re-nameable function plates which may called using function-call plates.

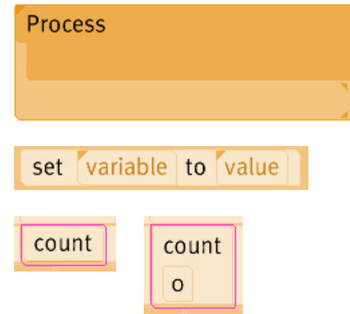


Figure 4.2.2. Plate's visual elements.

Although the material and machine components of Plate seem thoroughly traditional, the methods of interaction, visualization, and evaluation gives the program a more machine-like feel. At the heart of the Plate environment is the plate. Almost all plates contain other plates. In fact, the only plates which contain no other plates are constant value plates. Most plates are generated from a contextual menu which is accessed from an already existing plate. Once a plate is made, it may be dragged and dropped into an empty space within another plates. For example, plates such as the "set _ to _" assignment actually contain place-holder plates for the two slots. These place-holder plates not only set aside a space for a future expression, but they also can suggest a possible expression via a contextual menu. Place-holder plates also serve another purpose. They prevent wrongly typed expressions from being placed in their slot. For example, if the user places a constant number plate into the right-hand slot of the "set _ to _" plate, the left-hand place-holder will only accept a number variable plate. This interactive type-checking feature is uncommon in most syntax-directed editors.

Variable plates are quite special. Since variables are always associated with a function, they are created from a contextual menu at the top of all functions, which contains a list of variable types. When a variable is created, a permanent version of the variable is placed in a line of variables at the top of the function. While this "original" variable may seem to serve as a declaration, its true function is as a source of variable instance plates. By simply clicking on the "original" variable and dragging away, a user can make an instance variable which can be placed anywhere within the

code. Once an instance is made, it will always be an instance. Its identity cannot be changed to that of another variable. Fortunately, if one changes the name of any instance variable, all other instance names update simultaneously. Variable plates have another secret feature—they contain values. If a variable is selected and the 'enter' key is pressed, the plate expands to reveal the variable's value plate. This value itself may be selected and changed. All instances of a variable show the current value of the variable when expanded. This feature allows convenient access to all of the computational material which is stored by the program.

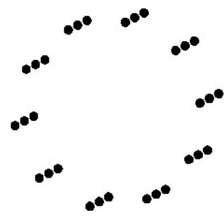


Figure 4.2.3. The program and its visual output (the ring) exist in one visual space.

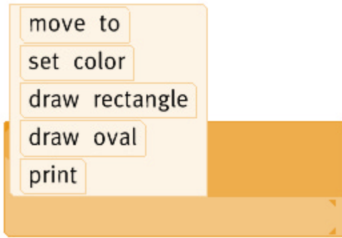
Plate has two modes of execution: full-speed and step-by-step. The details of the computation are visible in both modes, though they change rapidly during full-speed execution. When the program is run at full speed, the entire program is executed once per frame. This strange execution model has some nice side-effects. One is that building a plate program which creates an animated graphic is unusually simple. Another side-effect is that when the program is changed during execution, the update in the behavior of the program occurs instantaneously. For example, one can quickly understand the effects of a repeat loop, simply by changing its bounds.

When the program is run step-by-step, a wireframe evaluation box moves from point to point in the program, highlighting the locus of execution. If a block plate is collapsed, the internal commands are treated as one step. Meanwhile, with each step, the hierarchical layers of the program separate and fold outward into the third dimension (Figure 4.2.1). This effect helps to concentrate the users attention on the locus of execution, and to aid the user in understanding how the program is constructed. During this process, one may at any time expand a variable to see its current value.

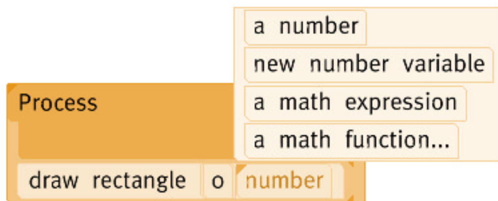
Plate Example

This page documents the construction of a simple Plate program which draws five squares.

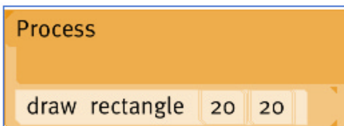
First, an empty process is created.



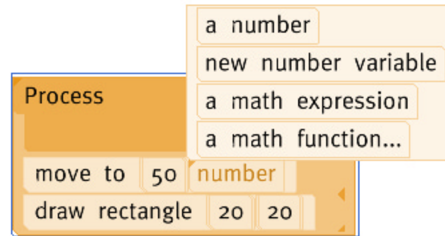
Then, "draw rectangle" is selected from the command menu.



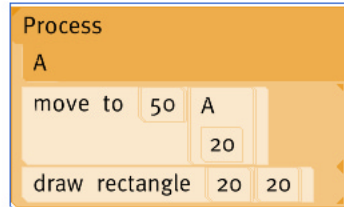
A number is placed into the first slot of the "draw rectangle" function.



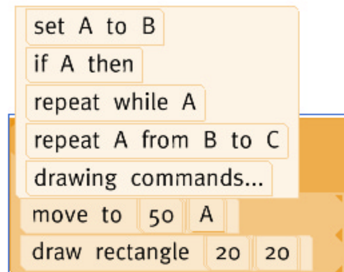
The program is run. Note the square.



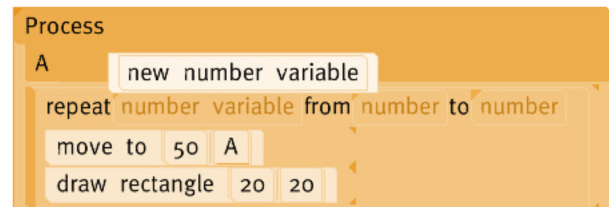
A "move to" command is added to the top and a number variable is created for the second slot.



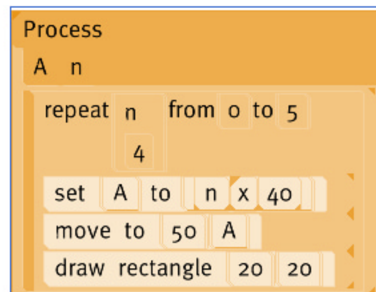
The variable is expanded and its value is revealed.



A repeat loop is added to the process.



A new number variable is created for the repeat loop.



The program is run. Note the five squares.

4.3 Pablo

Pablo is a Visual Machine Language based on the functional dataflow paradigm. The design of Pablo begins with the model of dataflow as realized in the Prograph environment. While Pablo resembles Prograph in its basic structure and semantics, it also is home to several of its own unique innovations, including the ability to visualize computation.

The most innovative aspect of Pablo is the visual relationship between the material and the machine during execution. In traditional dataflow implementations, one makes the connections through which the data theoretically flows during execution, but one never sees the actual data flowing down the links (Figure 4.3.1.) Although this may seem unnecessary to some, it serves several purposes. For the novice programmer, it reinforces the conceptual model behind data flow as well as parameter passing. For all programmers, it provides immediate and well-placed access to all the material data that is manipulated by the program. This not only includes the material data at the inputs and outputs, but

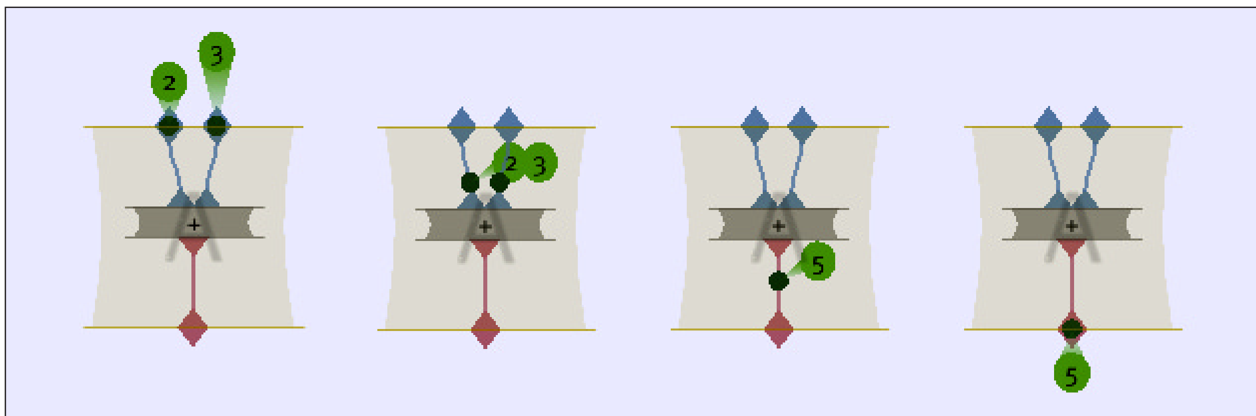


Figure 4.3.1. Animation of an evaluation of an addition computation.

all of the intermediate data that is generated along the way. In a traditional development environment, much of this data would never be seen, unless hunted down in a separate debugging environment. As with Plate, all the data is accessible, and all of it is exactly where one would expect it to be.

In traditional functional dataflow environments, functions exist inside windows and function calls are dislocated entities

which remotely reference the original. Within Pablo all functions exist in a unified visual space and there is no distinction between functions and function calls. What is traditionally thought to be a function call is created by dragging an instance of the called function into the calling function. This creates a collapsed copy

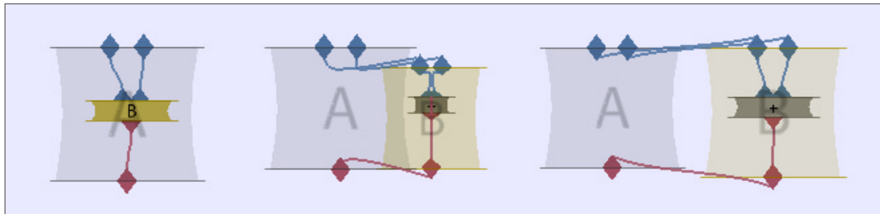


Figure 4.3.2. Animation of a function call. Function A calls function B.

of the called function within the calling function. When inside the calling function, one may make connections to the in-ports or away from the out-ports of the newly introduced function. If this collapsed function is selected and opened, it simply expands to full size and moves directly to the right of the calling function (Figure 4.3.2.) Meanwhile, all of the connections to and away from the called function remain intact despite the fact that function is expanded and outside of the original function. This expansion process occurs automatically during execution when functions are called. When a function is finished, it collapses back into its caller's domain. The result of this rightward expansion is a clear visual representation of the function stack (Figure 4.3.4.) Of course, one may choose to stop the visual unfolding and let the execution continue invisibly. When evaluating recursive functions with great recursive depths, this option is quite useful.

As with variables in the Plate environment, all instances of a function remain identical in Pablo. During the editing process, a given function may be represented visually in two places at once, since two instances of the function may be open at the same time. Fortunately, when a change is made to any instance of a function, that change is automatically reflected in all of the other instances. Even during runtime, the execution may be

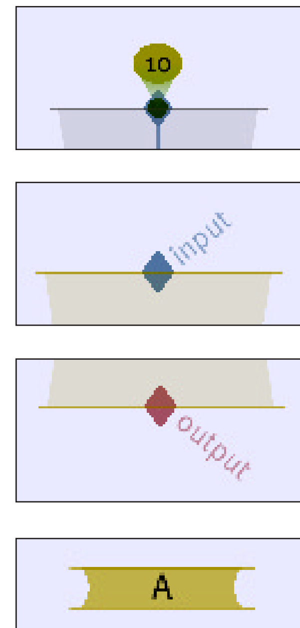


Figure 4.3.3 Visual language elements in Pablo. From top to bottom: a value, an inport, an outport, a closed function.

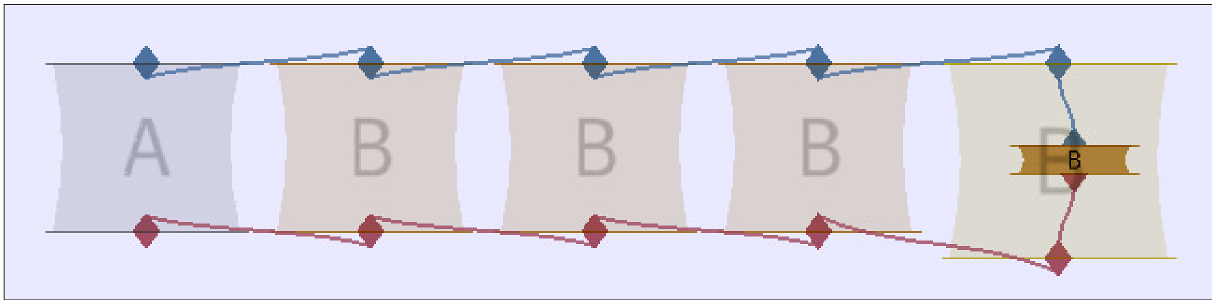


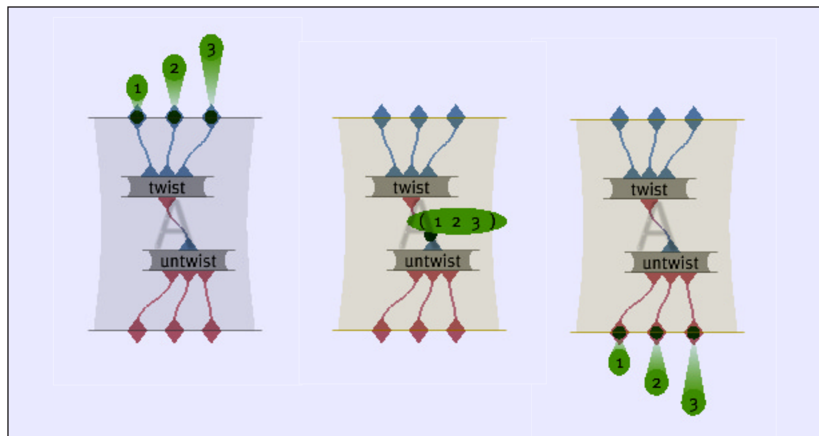
Figure 4.3.4. A recursive function demonstrates how the function stack is an inherent feature of the visualization.

paused for changes to be made.

While most of the editing within Pablo takes place with respect to the two dimensional plane of the screen, all Pablo objects exists in a three-dimensional space. Functions may be rotated to be viewed from the top, allowing multiple function stacks to expand simultaneously in a star-like formation from a single source function. Additional exploration of the three-dimensional space has yet to be done.

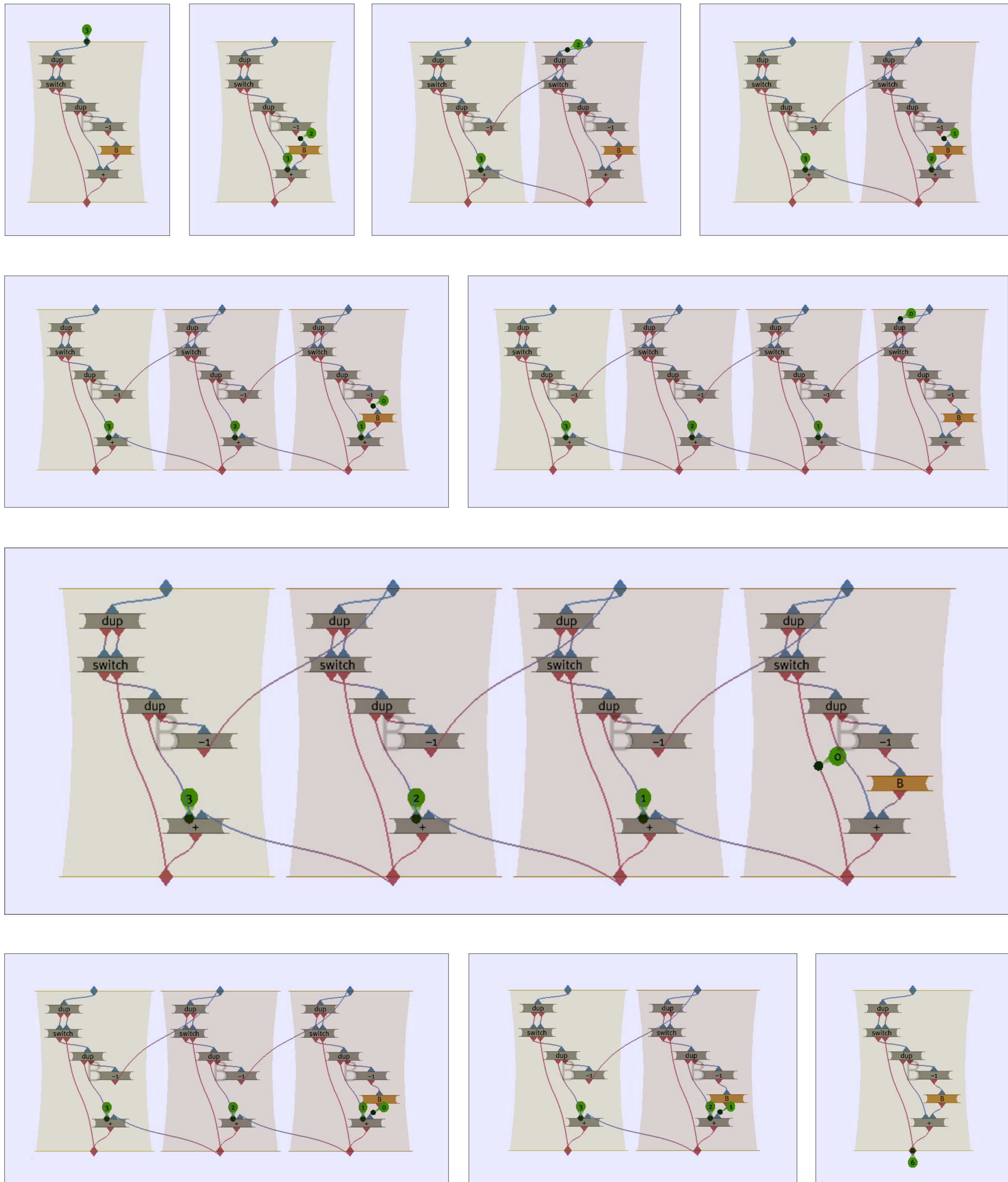
Within Pablo there is an explicit blocking and pooling behavior which allows data to arrive asynchronously as well as in bulk. For example, if only the first of two data values is present at an addition function's in-ports, then the system will wait for the second to arrive. Additionally, if another value arrives at the first in-port, an ordered pool will form at the port. When a value arrives at the other in-port, the first value will be extracted from the pool, leaving the second value to wait. When a second value arrives at the second in-port, another addition will take place. This process of blocking, pooling, and depooling occurs naturally within Pablo.

Figure 4.3.5. The twist and untwist functions allow multiple values to pass through one thread.



Pablo Example

This example shows the visualization of a recursive summation function. The recursive function is specified as in the first frame, and an integer value 3 is loaded in as input. The rest of the images are generated during the animation of the execution. The final result of the summation from 0 to 3 is 6.



4.4 Nerpa

Nerpa is a Visual Machine Language which is similar to Plate in that it is built upon a text-based language. The visual aspects of Nerpa serve to replace the structural elements (i.e. the parentheses) of a traditional text based language, while the text provides the majority of the specific semantics. As with Plate, the visual elements also provide a means for visualizing the computation within the language specification. The materials used within Nerpa are essentially the same materials that are used in Plate. Unlike Plate, which rests on an imperative model of computation, Nerpa is a purely functional language.

From a programmer's perspective, the most interesting feature of functional languages is that every expression within the program evaluates to a value. This is not only true of every complete program expression, but also every subexpression within an expression within a greater program expression. Knowledge of this property was an inspiration for the Nerpa environment.

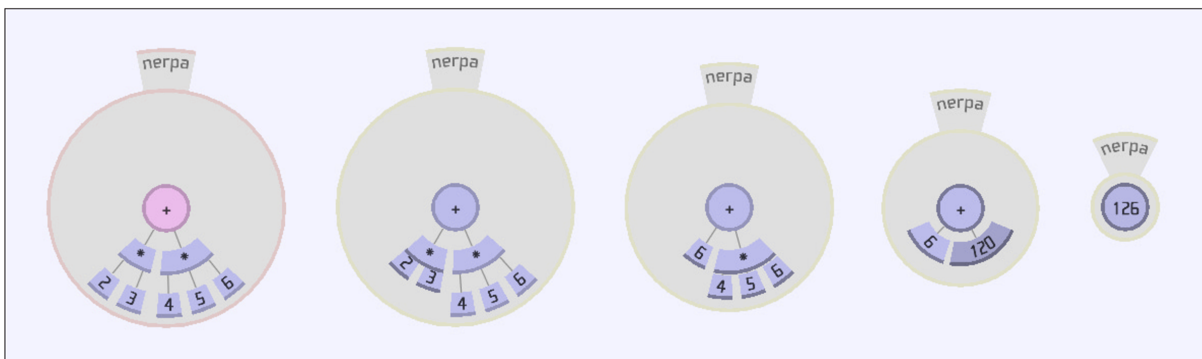


Figure 4.4.1. An animation of the evaluation of a simple arithmetic computation.

In Nerpa, every programmatic expression exists on the front side of a two-sided plate (Figure 4.4.2). On the back side of every plate is the value which results the from the evaluation of the expression on the front. Constant numbers, for example, have the same value on both sides since the evaluation of a constant number value is that number value. This is not the case for function calls in which the expression on the front contains the text of the function name and the back shows the value returned by the function. Nerpa plates may be flipped to show their back or front at any time,

Figure 4.4.2. A Nerpa plate may be flipped to show its value.

prior, during, or after execution has occurred.

The high-level visual structure of Nerpa is a simple hierarchical tree displayed in polar form, instead of the normal top to bottom display. Every distinct tree represents one function, which may take in as well as return values. The hierarchical tree is a direct display of the expression hierarchy that is found in the purely text-based functional language. The children of a Nerpa plate represent the arguments (or inputs) to the operator or function found on the parent plate. This representation is similar to the parse trees generated internally by text-based language compilers.

An entire Nerpa program consists of a set of circular Nerpa functions which may be placed at any position on the two-dimensional plane. At the center of every tree is the root plate. The final value returned by any function will exist on the back side of the function's root plate. Naturally, functions also take arguments. Function variables are displayed dangling off the bottom of the function circle. When one function calls another function, the variables of the called function are the first expressions to be given values. From here, the computation progresses inward.

The inward progression of computation (Figure 4.4.1) is central to Nerpa's machine-like operation. When an operator is about to process its arguments (which are its child plates), it pulls the argument plates inward towards itself. Only once the arguments have been pulled so far that they rest behind the operator function, the operation takes place and the operator plate receives a value on its back side. This inward collapsing process continues until only the root plate is left showing, and it has a value on its reverse side. Once the computation of the function is complete, one may unwind the structure and examine each of the evaluated values along the way. This form of debugging provides convenient access to all the materials that are used during the execution.

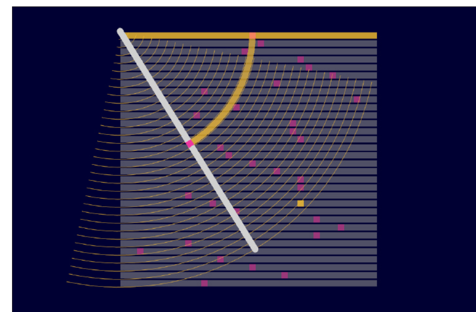
4.5 Reverie

In the process of creating the fully functional Visual Machines described above, one less functionally-oriented Visual Machine was designed specifically for an art gallery setting. This piece, Reverie, was intended to communicate the idea of the Visual Machine, and convey the essence of computation in its most minimal form.

The Reverie project evolved from a thought experiment about the semantics of memory addressing and referencing, and specifically, how those features facilitate instantaneous jumps within an otherwise linear process. The relationship between the reference of a memory address and a discontinuity within the computation felt somewhat strained. The goal of Reverie was to make a connection between the data that represented the process and the actual progression of the computation. After some initial sketch work, one solution was found and realized in Reverie.

The material used in Reverie is essentially part of the machine. The material is the code that the machine processes. Interestingly, Reverie's execution has no effect on the computaional material. If the the execution did affect the material, the result would be a self-modifying machine. What is effected by Reverie's execution is the state of the execution itself. At the heart of Reverie is the idea is that a program is a machine which consists solely of information.

Reverie is a pure micro-computing machine. It runs a thirty-two line program, literally. The thirty-two lines are stacked vertically with a calculated spacing to form an exact square on screen (Figure 4.5.1). Each line contains one micro-address slider which takes on one of thirty-two distinct values. The system is programmed simple by clicking on any of the thirty-two lines of "code." When one does so, the slider moves horizontally to the closest of the thirty-two positions.



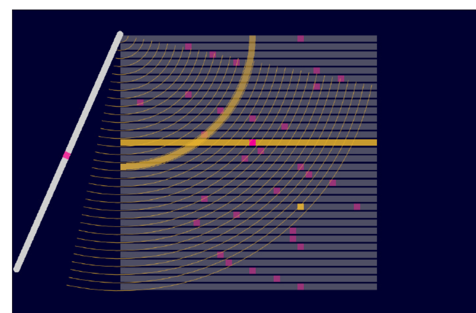


Figure 4.5.1. The white beam draws and arc which references an addressed line.

The visual connection between the address sliders of each line and the unfolding of the process is a long rotating beam (Figure 4.5.1) which is anchored in the top left corner of the square. The beam traces an arc (Figure 4.5.2) across the square, which starts at the top line's address slider position and arrives at the left-end of a line within the stack. In other words, the beam helps the address slider select the line that it is addressing. In this model, addressing always occurs downward and with respect to the line of the address. For this mahine to work, the lines must wrap around the top and bottom of the square. While this looping memory solution was functionally non-optimal, it enabled the piece to run continuously regardless of configuration. This property seemed appropriate for a piece to be shown in a gallery.

Since Reverie revolves around the idea of a visual connection between material and process, it is not surprising that the result resembles a mechanistic solution. This reference to industrial contraptions wholly contrasts the discrete nature of computational machinery. While Reverie is far from computationally compete, it set a precedent with its machine aesthetic and operation, which help to refine the idea of a Visual Machine.

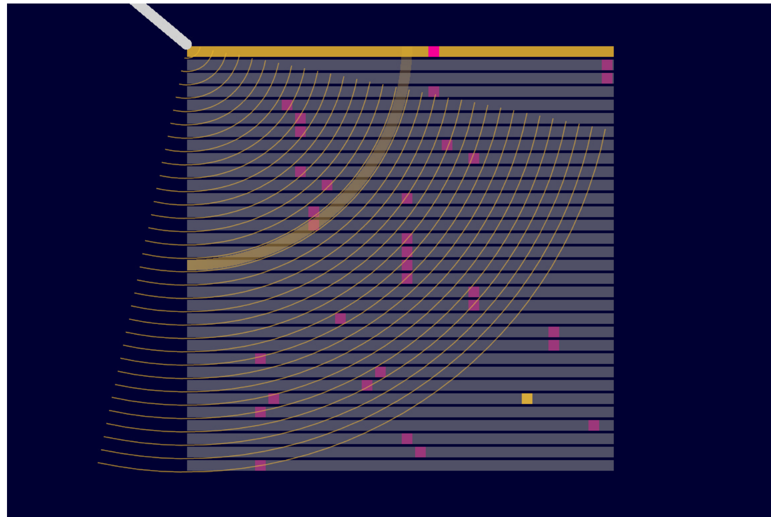


Figure 4.5.2. Every step in the execution of Reverie is a cycle.

4.6 User Testing

Unfortunately, due to lack of time, formal user testing for ease of use could not be conducted for the five design experiments. Informal user testing has taken place for each of the systems.

One setting in which all of the designs have been casually tested is the weekly demo session for Media Laboratory sponsors. During each of these events, a specific project is presented and discussed. Sponsors are allowed to interact with the project and often provide valuable feedback. When the sponsor fails to understand how the system works, then it is clear that the ease of use needs to be imporoved. In general, both programmer and non-programmer sponsors had positive reactions to most the of the Visual Machines. Turing, Plate, and Pablo were particularly well received.

More direct user testing was conducted for the Turing Visual Machine. Turing was presented and used during a two-day, six hour workshop for high school students at High Tech High in San Diego, California. Most of these students had no programming experience, and none had prior knowledge of the Turing Machine. Approximately two-thirds of the students in the class were able to work through several simple Turing problems, and a handful of students successfully solved a few very difficult problems. The difficulty that the students faced was not in comprehending Turing's operation or even how to construct a program, but rather how to think procedurally and to decompose the problem into its parts. To the students, Turing was like a video game—simple to play with, difficult to master. This is how programming systems should be.

Independent testing and analysis of Turing, Plate, and Pablo was performed by undergraduate research assistant (UROP) Joy Forsythe, who had taken one computer science class (6.001) at MIT. For each of the systems, Joy was given a brief introduction (five minutes or less), which included at least one example program. She was then asked to use the Visual Machines for several hours and write down her thoughts. Joy was able to create several significantly complex programs in all three systems. She was even able to figure out many unexplained features of Pablo, simply based on her own experimentation. These discoveries were made possible not only by Joy's intellect but also by the fact that she could interact with the visible computation in real-time.

Chapter 5 : Discussion & Analysis

5.1 Successes

The design experiments presented in Chapter 4 represent first steps into the domain of Visual Machines. While none of them is a complete programming system, their development as a whole was successful in several respects.

5.1.1 Unification

Every one of the experiments combined a means of both specifying and visualizing computation in a single visual space and with a unified visual vocabulary. The way in which the visualization develops directly from the language of the specification represents a genuine innovation. This unification of the two parts mutually enhances both sides of the process. The specification becomes easier to understand because it is informed by the visualization, and the visualization is more comprehendible because it maintains the representation of the specification. Since all of this functionality lies in one continuous space, never is a context switch or mental remapping required.

5.1.2 Maintenance of Continuity

Much of the work in developing each of the Visual Machines involved maintaining the feeling of continuity. In general, this work was worth the effort expended. All of the designs display a high degree of visual continuity. The position, scale, form, and color of the objects always changes smoothly between events. Specifically, the transformation of discrete computational transformations into fully continuous visual transformations is especially successful. Additionally, continuity of interaction is well supported. One may freely switch between tasks such as construction, navigation, or execution without a break in the flow of the process. The results of this continuity are programming systems that feel unconstrained, and visualizations of computation that are ease to follow.

5.1.3 Aesthetic Approach

While none of the five design experiments deserves a gold medal for graphic design, they do represent a significant advance of aesthetics within the specific field of visual programming. The examples that were developed for this thesis were done so in the context of an environment which earnestly values imaginative and refined visual design. This factor alone separates the Visual Machines from other work in the field. The amount of consideration applied to the choice of the colors, visual forms, typography and motion resulted in a series of programming systems that are not only easy to use but a pleasure to see.

5.1.4 Variety of Computation

In the early stages of the Visual Machine Model, it was unclear whether the model would support many types of computation. Given the variety of the underlying computation models in the experimental designs, it is clear that this is not a problem. Within the five designs, there are implementations of imperative, functional, and procedural models of computation. Certainly, object-oriented and constraint-based computational models could also be developed into a Visual Machine Language.

5.1.5 Integration of Evaluation

The story of the development of the experiments is as much a story of implementation as of design. One of the great successes of the implementation story is the integration of evaluation mechanisms directly into the visual language representations. While many visual programming environments simply export text-based code for compilation, the designs created for this thesis all incorporate their own means of evaluation. The great advantage of this approach is that all of the information about the syntax and semantics of the language as well as the actual runtime computation is available for immediate feedback through the visual representation. If this were not the case, the visualization of the computation would be nearly impossible.

5.2 Challenges

In the process of developing the design experiments, certain issues arose again and again. These issues turned out to be the real challenges in creating a Visual Machine Language that fulfilled the Visual Machine Model.

5.2.1 Completeness vs. Concretization

By far the greatest conceptual design challenge was to imagine a system which addressed all the necessary parts of computation. Many excellent potential designs were rejected because they could not be extended to perform a complete computation. Some designs focused too heavily on material while others were concentrated entirely on the machine aspect. Even systems that could handle both machine and material at the operational level, had to be concerned with higher level process issues, such as how material travels from one machine to the next. Fundamentally, the difficulty lay in the process of concretization. Deciding what to make concrete and what to leave as an implicit part of the computation proved to be a mind-wrenching process. Starting from a pre-existing programming language ensured that a language would be complete, but ultimately resulted in designs that were less conceptually innovative.

5.2.2 Visual Causality

The greatest visual design challenge was in creating machines which imparted a sense of causality in their interactions with the materials. Semantic continuity suffered. Of all the aspects of continuity, 'cause and effect' was the most difficult to attain. Part of the problem was that there are so few real-world references for the mutation of abstract information such as numbers and strings. Having values replaced rather than mutated was one solution to this problem, but it had the unfortunate side-effect of generating trash material. In most of the systems presented here, material is changed when it and another object make contact. This "billiards-style" visualization is passable, but still does not connote the idea of real machine-like cause and effect. These critical events of change should make the machine appear more like a real machine and the material feel more like a real material.

5.2.2 Architectural Space

The architectural use of space within all of the Visual Machines suffered, not because of the difficulty of the problem, but because it was often treated as an afterthought. The Visual Machine Model itself is focused much more on the relationships between pairs of objects (machines and materials), than on the relationships between hundreds of objects. Most of the Visual Machines deal with space in the same generic manner. They allow objects to be collapsed to save screen space. Of course, when an object is collapsed, not all of its information is being shown, A better use of space would allow for a more organic and yet organized arrangement of objects. It would be nice if the program felt like it was more involved with the space—piercing, dividing, and wrapping the three-dimensional volume. Finally, an improved navigation system could be guided by the spaces between objects, rather than locked into the standard set of axes.

5.2.3 Rhythm

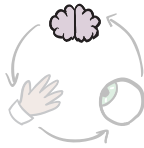
A less significant, but equally frustrating absence in all of the Visual Machine designs is a lack of rhythm during execution. A major component of real cartoon animation is the careful use of time when composing motion. Distortion of time in cartoon animation emphasizes the important aspects of a movement, and prepares the viewer for imminent events. Repeated distortion of time creates a beat around which structures may evolve. The same kind of rhythm exists in real mechanical machines, especially those with motors. Even without the sound effects, these machines have a definite cadence. Perhaps, Visual Machines would feel more machine-like if they had more of a pulse.

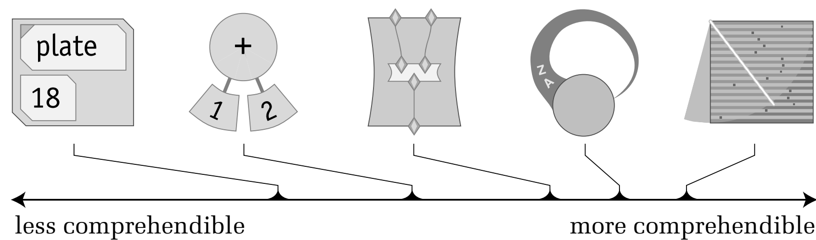
5.3 A Comparative Evaluation

Since all of the Visual Machine implementations presented in this thesis share a common set of goals, and exist under a common model, it make senses to evaluate them with respect to each other. In order to perform this evaluation, five axes were chosen against which each machine was measured. One should note that the axes are not intended to establish a taxonomy for Visual Machines. Nor are the axes designed to be orthogonal. The first three axes are defined by the metric used throughout this thesis: ease of use. The first three axes are ease of comprehension, ease of construction, and ease of consumption. The following two axes relate specifically to the Visual Machine Model. Those axes are Material-Machine Balance and Machine-like Feel.

5.3.1 Ease of Comprehension

As discussed in Chapter 1, ease of comprehension relates to the amount of work that is required to move from one's current understanding to a complete understanding of the system being measured. Naturally, the Visual Machines with the simplest models of computation require the least work to understand. Both Turing and Reverie fit this description. The dataflow model of Pablo makes it the easiest to understand of the more fleshed-out programming systems. While the operators in Pablo may be arbitrarily complex, they are all manifested by a common input/output style representation. Once the user comprehends the basic connection model in Pablo, the rest of the understanding should fall into place. Like Pablo, Nerpa exhibits a common visual model for all operations. The difference is that Nerpa is much more dependent on textual representations. Additionally, there are many more discontinuities in Nerpa, especially during function calls. The Visual Machine which requires

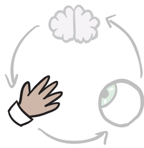


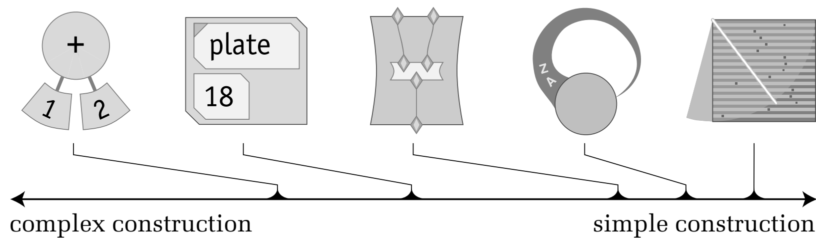


the most work to learn is Plate. Plate's text-based model and various plate instantiations prevent its immediate comprehension. Realistically, Plate is no easier to understand than any traditional text-based programming language.

5.3.2 Ease of Construction

Assuming that the semantics of a Visual Machine are understood by the user, one can begin to measure ease of construction. Ease of construction is a measure of the difficulty involved in actually creating the computation. This metric values systems which minimize the number of steps involved in composition and are free of obstacles along the way. Once again, the simpler models of computation, Reverie and Turing, have the upper hand. Of the two Reverie is significantly simpler to use since the user need only to click the mouse once to change the computation. Both of these Visual Machines utilize a small number of highly constrained elements. Pablo shares a construction method with Turing in that they both involve making connections. The color-coded inports and outports of Pablo ensure that this means of construction is simple, reliable, and free of obstruction. When compared to freeform text-editors, constructing

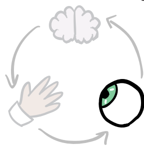


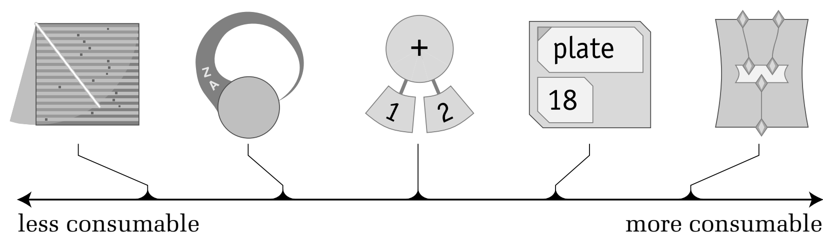


programs in Plate is a quantum leap forward in ease of use. The language level objects, automatic syntax-checking, and context-sensitive code suggestion make Plate a wonderful tool for programming in a text-based language. Unfortunately, constructing programs with text will never be as simple as doing so with connections. While this hurts Plate, it is devastating to Nerpa which has none of the complementary construction features of Plate.

5.3.3 Ease of Consumption

Of the three components of ease of use, the one that is most affected by the visualization of computation is ease of consumption. This metric measures how well seeing can lead to understanding within a visual programming environment. Pablo takes the lead in this category. When a Pablo program is running, the whole state of the program is plain to see. The flow of data down the connections guides the eye when following the execution. All of the material values are shown, and even the function-stack is clearly represented. Far behind Pablo, but still respectable is Plate. In this case, the text-based representation helps. Even with code that is convoluted, one may still read the function and variable names and gain some understanding of the what the program does. Additionally, Plate provides convenient access to the whole state of the program via the values stored in the expandable drawers. Plate's the three-dimensional deconstruction of the layers of code serve to focus the viewer on the exact code at hand. Unlike Plate, the text of Nerpa is not in a readable format. The minimalist hierarchy of operations in Nerpa is not always simple to parse. While the runtime visualization of Nerpa's computation greatly enhances one's perception

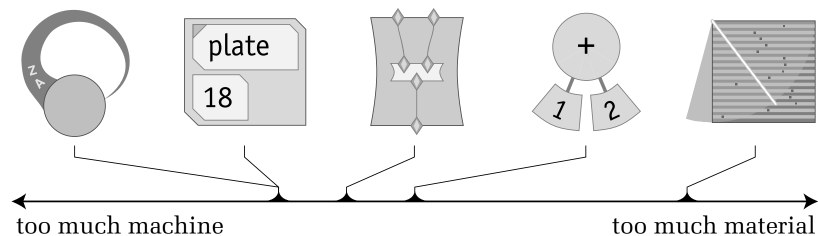




of the flow of the program and the transformation of data, it is not enough to make it highly consumable. The same minimalism which afflicts Nerpa causes both Turing and Reverie to be inconsumable. While the minute transitions within each of these systems is simple to see and understand, the greater gestalt of the running program is nearly impossible to capture. The result is computation which is neither predictable nor memorable.

5.3.4 Material-Machine Balance

When interacting with a computational medium, it is best to have equal access to both machine and material. A painting program, for example, deals almost exclusively with the material, but generally has no tools for building a machine. Most programming environments, on the other hand, are concerned solely with building a machine and ignore the material. The ideal Visual Machine Language would have a perfect balance of the two. One should be able to view and interact with any part of the machine, and any part of the material, at any time during execution.

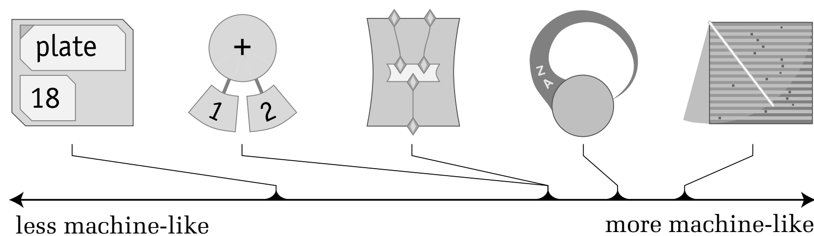


The Visual Machines designed for this thesis are still too machine-oriented. Of all the systems, only Nerpa achieves a perfect balance between the two. The functional nature of Nerpa along with the dual sided faces enforce the machine material balance. For every bit of machine on one face, there is just as much material on the back face. Pablo, Plate, and Turing are all slightly too machine oriented. The tell-tale signs that a system is too machine oriented are that the primary interface is for building machines, and that the materials can only be found inside of the machines. Reverie, on the other hand, is entirely too material oriented. However, given the direct connection between the materials and processes of Reverie, it makes sense that materials are dominant, since they effectively control the machine.

Fortunately, designing programs that deal with the material directly is easier than designing programs for building machines. What is needed most is the integration of current material-editing programs into computational environments. Obviously, there is a still a long way to go before the machine-material balance is achieved.

5.3.5 Machine-like Feel

The last axis examined here is a measure of aesthetics that relates specifically to the Visual Machine Model. Machine-like feel describe how well a Visual Machine Language imparts the idea of an actual machine being run on actual materials. Of all the Visual Machines, Reverie feels the most like a mechanical object. In fact, one could implement Reverie with physical materials without changing the design significantly. The machine and the materials are so tightly coupled that they are difficult to separate. A similar response is created by Turing. Given that Turing, the man, was also inspired by mechanical machines, it is not surprising that a Visual Machine implementation of the Turing Machine feels like a real machine. The Turing tape is definitely the stronger of the two parts in shaping the machine-like feeling. This feeling in Turing is lessened by the freeness of the movement, and the lack of a realistic cause and effect display when changing symbols. Despite being highly diagrammatic, both Pablo and Nerpa do appear somewhat machine like. Pablo's connections resemble cables which the values slide down as if on an assembly line or sky lift. In Nerpa,



the continuous contraction of the diagram has the feeling of a complex pulley system or collapsing camera iris. Nerpa's flipping faces are reminiscent of the letters on a train station's ever changing schedule boards. Unfortunately, but predictably, the machine-like feeling is almost nonexistent in Plate, due to its textual nature. Alas, the notion of machine-like 'word processor' is not an idea which has a real-world equivalent.

5.4 Future Work

There are several viable directions in which the research that was begun in this thesis could be continued. Each one of these subdomains has enough unencountered material to be the subject of another whole thesis.

One rich area of research would be to refine the techniques for visibly demonstrating the processes of computation. Specifically, the demonstration of cause and effect could use significant improvements. This research could also incorporate more technical evidence from the field of perceptual psychology. Another approach could make use of the refined techniques of traditional character animation. Imbuing the computational elements with a hint of intentionality might make the process feel more causal.

The design experiments created for this thesis are closer to functional sketches than complete programming systems. For each of the systems, a great deal of work could be done to carry them on the path to completeness. Certainly, one necessary line of research could involve increasing the complexity and the scalability of the Visual Machine Languages. If visible computation is ever to be a part of a mainstream programming environment, there must be more support that it is feasible and usable in a substantial working system. Due to the complexity and scale of this work, this research would be best done by a small group of researchers, rather than just one person.

One aspect of research which is absent from this thesis is experimentation in the form of in-depth and formal user testing. User testing, especially in the realm of user interfaces, is a laborious but necessary process. While the testing with the high school students and with the undergraduate researcher provided good qualitative information, more scientific methods are necessary to obtain results that are more verifiable. The ultimate test for a Visual Machine Language is for it to be understood and usable after only a brief demonstration of its capabilities.

5.5 Conclusion

This thesis has demonstrated that making computation visible and interactive is not only a desirable goal but one that may be attained. While the ease of use of such systems has not been proven experimentally, general user interactions thus far indicate that programming environments which visualize computation could greatly improve the process of programming. Specifically, computation could become easier to comprehend, easier to construct, and easier to consume.

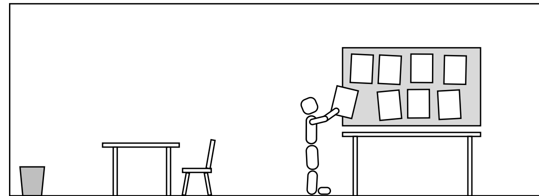
Both traditional visual and text-based representations were examined with respect to the three aspects of ease of use. Visual representations were noted for their proximity to the programmer's internal model and capability to represent complex structures, while text-based representations were deemed best suited for representing natural language. Given the visual slant of the thesis, the role of graphic design in the field of visual programming was then discussed. Several reasons were given for the need for deliberate and refined graphic design in visual programming systems.

Working in the mode of graphic design, a conceptual framework, the Visual Machine Model, was developed in order to guide the development of the programming systems which would make computation visible. This model described a specific breed of visual programming systems, called Visual Machine Languages which draw from the world of mechanical objects and processes in order to represent computation concretely. A Visual Machine Language consists of well-defined material and machine objects which interact to perform computation. The Model also defined the criteria of visual, interactive, and semantic continuity so as to prevent the viewer from becoming bewildered during program execution.

With the Visual Machine Model as a backdrop, five design experiments were designed and implemented to test the feasability of the Model, and to serve as sample points in the domain. Each of the five designs was based on a unique computational model and specific visual language. The programming systems were created such that the dynamic visualization of computation evolves directly from the static visual specification of the computation.

In general, the design experiments were successful as functional sketches, in that they enabled simple programs to be written within each of them. The integration of the program specification and computation visualization in a unified visual space and with a single vocabulary was particularly effective. The designs, however, were far from perfect. The greatest visual design challenge was to develop convincing causal reactions between the machines and the materials, which impart the sense that the computational material is being changed. Other shortcomings included the paltry use of the three-dimensional space as well the lack of rhythm during the program execution.

The development of the Visual Machine Model and accompanying Languages suggest a future where a programmer could craft computation in a manner befitting the complexity of the task. No would longer would the computation be hidden from the programmer. No longer would the programmer be separated from the program.



Appendix A: Prior Work

A.1 Interactive Visual Programming Languages and Environments

In this section, several intriguing visual languages and environments will be presented. The examples that have been chosen were selected because they represent innovative interactive visual models of computation. While there are many integrated development environments which are considered "visual programming languages", most of these still use text of their primary means of representation. The focus here is on truly visual programming languages.

A.1.1 Prograph

One of the most successful truly visual programming languages is Cox and Pietrzykowsky's Prograph [Cox 1990], now distributed by Pictorius, Inc. Prograph is a visual, object-oriented, data-flow language which makes use of many special constructs to achieve a high degree of usability. Prograph iss unique in that it is one of the few scalable visual programming languages.

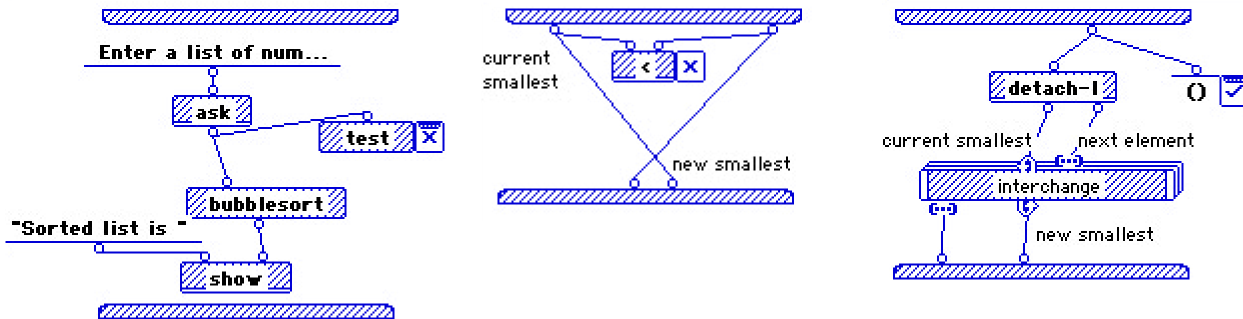


Figure A.1.1.1

Since Prograph is a data-flow language, it is essentially based on a functional model, with the exception of user-defined persistent data. A function or method in Prograph consists of a downward flowing network, which exists between an input bar at the top, and an output bar at the bottom (Figure A.1.1.1). Connectors which attach to the top bar, define the input parameters of the function. Likewise, the output connectors lie along the bottom bar. A method may have any number of inputs and outputs, including none. One should note that Prograph makes no use of named local variables.

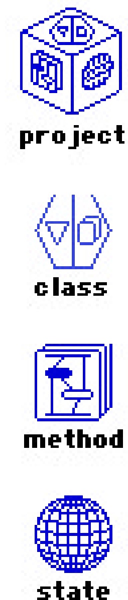


Figure A.1.1.2

The graphic language of Prograph is relatively straightforward. The designers of Prograph were certainly concerned with creating a clean and tidy programming system. The design is simple and monochromatic and follows in line with traditional Macintosh user interfaces in its use of windows and icons (Figure A.1.1.2.) Hexagonal icons represent classes. Triangular icons represent data, network icons represent code, and spheres represent persitent state. These icons are used mostly in the navigation of the greater program

space, which exists in a hierarchical stack of windows.

The graphic language of Prograph's functions consists of nodes, connection points, and connections. Nodes resemble narrow rounded rectangles, which contain function names, and give additional visual clues about the node's function. Connection points are small circles that lie on the outer edge of the nodes, and may be moved around its edge. Between connection points lie connections, which are simply straight black lines.

Prographs's sophistication is the result of several carefully designed features which augment the computational model. The first such addition is the iteration construct. In a traditional data-flow language, iteration is quite difficult due to the constraints of feedback. Prograph, however, allows the use of a special iteration-style function which automatically is executed once for each of the items in an input list. Another unique prograph feature is the syncronization connector. This is a special connector which ensures that one function is executed before another when they exist in parallel. This feature is necessary since Prograph is not a pure functional language due to potential side-effects of functions.

While Prograph is a highly effective programming environment for the intermediate or advanced programmer, it might too difficult for the programming novice. The compactness of representation and the abstract nature of the whole system belie its ease of use. Navigation of the whole program space could certainly be more optimal. Since the modular data flow network is broken into functions, each of which exists in its own window, one cannot easily move quickly between two distant parts of the program. Additionally, one cannot view the program in one visual space. Finally, while Prograph claims to have an animated debugging capability, it is actually a fast step-through of the evaluation, with almost no actual animation. Despite these shortcomings, Prograph represents a significant point of progress in the field of visual programming.

A.1.2 Pictorial Janus

Of all the visual languages and environments presented here, Pictorial Janus is the most relevant to this thesis. That is because it is the only system that truly combines a visual programming language with an animated program visualization, and uses a single visual vocabulary. Pictorial Janus is a visual implementation of an obscure concurrent logical programming language called Janus.

The visual vocabulary of Pictorial Janus is not based around specific shapes or icons, but rather is concerned with the topological relationships between objects. There are only two kinds of graphical objects handled by the system: closed contours and lines. [Duecker 2000] From these primitives, one may construct higher-level semantic objects. The central objects of Pictorial Janus are agents and messages (Figure A.1.2.1, A.1.2.2). An agent is visually represented as a closed contour with input ports on its exterior and rule objects in its interior.

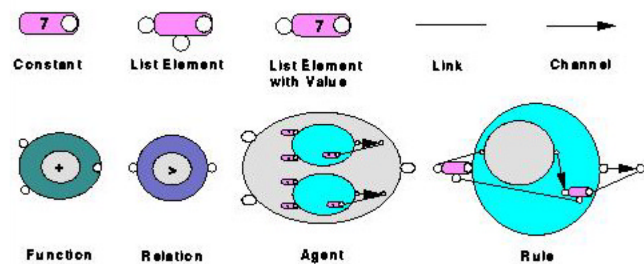


Figure A.1.2.1.

Messages, which include constants and list elements, are generally pill-shaped and may contain values represented in textual form. Connections between agents and messages occur both explicitly as lines and arrows, and implicitly as adjacencies between objects. While the user may attach names to the agents, the names do not affect the computation.

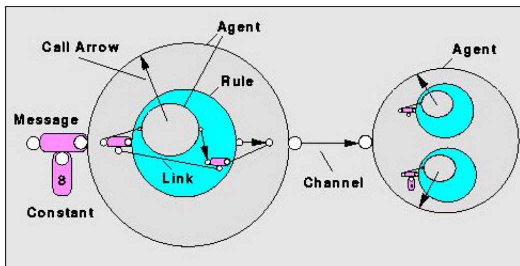


Figure A.1.2.2.

Remarkably, all the semantics of Pictorial Janus can be stated in terms of the visual relationships between objects. In this sense, it is a true visual language. The semantic model, while concise, is somewhat abstruse. It has the flavor of both lambda calculus and message-based systems. At the high level, agents act in parallel and communicate back and forth via messages. The real

computation, however, takes place inside the agents. Every agent contains a set of rules. These rules resemble their container agent in that they have the same input interface. The rules, however, have constant values, called preconditions, attached to their input ports (Figure A.1.2.3). If the input matches the precondition, then the agent assumes a new subconfiguration defined by the rule. [Stasko 2000] Once the subconfiguration becomes the main configuration, a process of linkage shortening occurs. This process effectively pulls the input values through the computation and turns them into output values. In a sense, this can be thought of as the simultaneous binding of variables and evaluation of expressions.

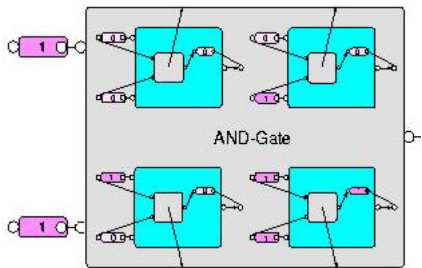


Figure A.1.2.3.

By far the most appealing aspect of Pictorial Janus is its use of animation (Figure A.1.2.4). While the static representation of the code specifies the program, "the execution is the animation." [Bonar 1990] All animations begin with the exact static visual representation of the code as created by the user. From this point, the individual objects are gently pulled across the two-dimensional plane by the continual shortening of the agent linkages. Once all the linkage shortening has occurred, a new rule is selected. Unfortunately, there is no animation associated with the matching process or the rule selection. At this stage, the current agent dissolves, as the new subconfiguration is smoothly scaled to take its place. The process then repeats itself until either there are no inputs left or until no matching rules apply. While the animation occurs very smoothly, it is difficult to follow since multiple objects move at once. Certainly, additional time spent with the system could raise one's understanding to the level of intuition.

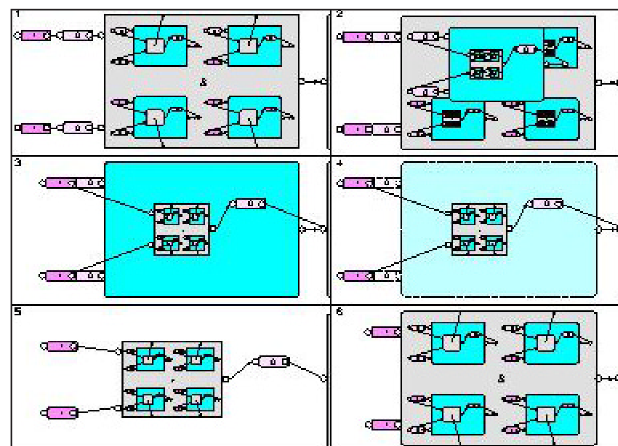


Figure A.1.2.4.

A.1.3 Incredible Machine

The Incredible Machine is not a programming language, per se. It is an educational video game which resembles a Rube Goldberg fantasy. The game has a number of predefined puzzles where the user is presented with a scene and is given a set of parts to assemble and a goal, such as "Get the ball into the basket," or "Trap the mouse in the jar." The parts include standard mechanical fare, such as gears, levers, and pulleys, as well as cats, mice, cheese, bombs, balloons, and more. Each part operates in a specific way which may be combined to solve a problem. A sample solution might be to use a balloon to lift the gate, so the mouse can escape, followed by the cat which powers a treadmill used to generate electricity to turn on a light bulb.

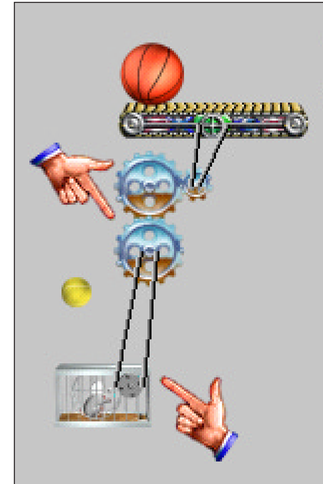


Figure A.1.3.1

The various interactions between the parts is rather stunning. No part can accomplish any task on its own. Almost all of the parts belong to a family of working parts. There are the mechanical parts (gears, levers, pulleys (Figure A.1.3.1)), the electrical parts (generators, lights, motors), the food-chain parts (cheese, mice, cats), forcing parts (fans, elevators, balloons) and finally structural parts (walls, tubes, buckets and balls (Figure A.1.3.2)). These categorizations, however, are never made explicit, since there are many parts which bridge the categories. The motor and generator are both electrical and mechanical. The cat, when running, can power the generator. Then there's the alligator, which strangely has the purpose of bouncing balls off of its tail. There are certainly hundreds if not thousands of possible interconnections between all the parts. The result is a space of millions of potential machines.

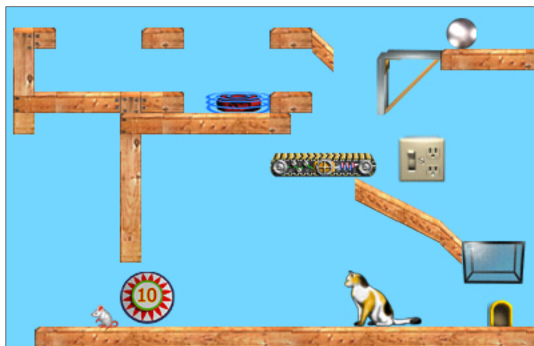


Figure A.1.3.2

From a computational perspective, one can begin to think about each of the parts as being either of piece of data or a process. The ball maps more closely to a piece of data since it is the object

being manipulated, while the motor maps to a process. While all of the part interactions are causal, they are not always predictable. Hence, much of the game involves trial and error.

Much like the classic Turing Machine, these machines have one point of termination. As with any computation, the scene must be reset or rewound before it can be run again. When the goal is eventually reached, the game is over. Interestingly, there is always more than one way to achieve the goal. Similar to any other computation, a machine naturally has side-effects on the way to achieving a goal. Unless the goal state and the initial state are identical, which never occurs, there is always some change of state along the way between the start and the finish. Though these side-effects are secondary to the objective, they could be made central to the computation, or even made into necessary sub-goals.

Since The Incredible Machine is aimed at children and young adolescents, the design of this game is cartoon-like. The visual design of the game is complete and professionally executed. Naturally, all of the interactions are animated. The animation is based on a simple physical simulation (Figure A.1.3.3) so the objects move fluidly, naturally, and continuously. That any person age eight and older can easily the understand the process as it

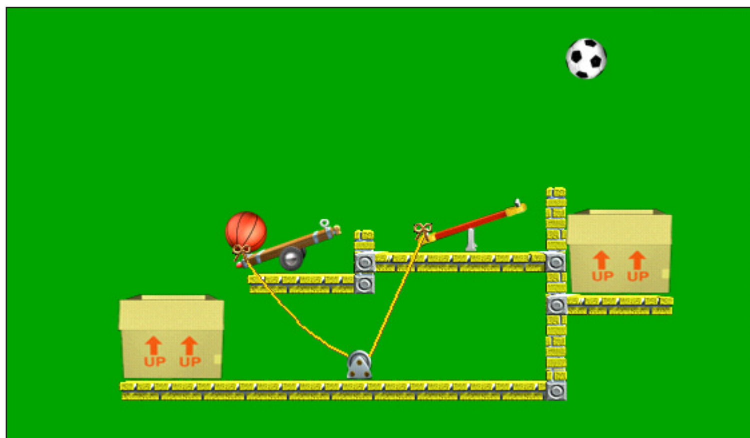


Figure A.1.3.3

unfolds is a testament to the quality of the animation. Given the consideration and craftsmanship that has shaped The Incredible Machine, it is not surprising that this educational game has had such success.

A.2 Software Visualization

A brother field to visual programming is the field of software visualization. "The main difference between VP [visual programming] and SV [software visualization] is the goal involved: VP seeks to make programs easier to specify by using a graphical (or "visual") notation while SV seeks to make programs and algorithms easier to understand using various techniques. " [Kehoe 1999] Software visualization as a domain comprises two subdomains: program visualization and algorithm visualization. While there has been some confusion as to the distinction between these two domains, a recent authoritative source provides the following definitions:

> *Program Visualization is the visualization of actual program code or data structures in either static or dynamic form.*
>
> *Algorithm Visualization is the visualization of the higher-level abstractions which describe software.* [Stasko 2000]

Perhaps a more realistic definition is that algorithm visualization deals specifically with algorithms, and that program visualization deals with everything else about a program.

A.2.1 Program Visualization

Most program visualization software relies on dynamic graphs called execution monitors to display the changing state of a program as it is processing. "The primary tasks of an execution monitor is to collect information about a program's execution and present that information to the user in an understandable way." [Jeffery 1999] Although this seems like the description of a debugger, the two are not the same. Unlike a debugger, which is focused on individual program steps and specific variable values, execution monitors use visual techniques to display a large amount of data at once, which is collected over time or across the breadth of the program. These monitors may display either data which is harvested directly from the execution, or data about the execution which is synthetically generated after the fact. [Kehoe 1999] Direct display and synthetic display are the two types of program execution monitors.

Figure A.2.1.1. Direct display of memory allocation.

Figure A.2.1.2. Synthetic display of procedure calls.

Direct display monitors show data which actually exists inside the computer during the execution of a program. This data includes the value of any variable in the program at any time. A common display is to simply graph a variable value as it changes throughout the course of the program. Direct display monitors may also display process data which relates to the program. For example, a monitor may graph the progression of the program counter, or in the case of an interpreted language, the active line number. [Jeffery 1999] This profiler-oriented information can give one a sense of how and when the different parts of the program are used. The process data which is available also includes the function stack and any values contained therein, such as the currently active function. One common direct display monitor is a map of all the memory used by a program and how it is allocated within the whole memory space (Figure A.2.1.1.) Much of the work in direct display monitors concerns not what data to show, but how to access the data during execution without affecting the running program. Since interpreted languages maintain more information about the program during execution, they can provide much more data to the direct display monitors with much less harm to the computation. [Jeffery 1999]

Synthetic display monitors show derived data—that is, data which does not exist inside the computer during a program's execution. A synthetic display monitor may show data derived from material data, such as the average value of variable over time, or a histogram of the variable's many values. Information derived from process data can also be used, such as with a monitor which displays the exact number of times that every function within a

program is used (Figure A.2.1.2.) In fact, one common use of synthetic display monitors is to count the number of times that a specific action or sequence of actions occurs within a program. A synthetic display monitor may also measure the total amount of free memory and graph that value over time. The number of possible options for synthetic display monitors is as large as the number of functions that one could write to synthesize the information. [Jeffery 1999]

A.2.2 Algorithm Visualization

The great majority of the research that is done in the field of software visualization is on algorithm visualization and animation. "The original motivation for algorithm animation was to explain an algorithm to an audience for educational purposes." [Jeffery 1999] Explaining algorithms is still the primary function, with the secondary function being a tool for debugging. Since, in most systems, one must construct a visualization on a per-algorithm basis (Figures 2.5.2.1-3), these techniques are not often used in debugging. "The dynamic, symbolic images in an algorithm animation help provide a concrete appearance to the abstract notions of algorithm methodologies, thus making them more explicit and clear." [Kehoe 1999] Because of the educational focus, a majority of the published algorithm visualization work deals exclusively with standard textbook style algorithms.

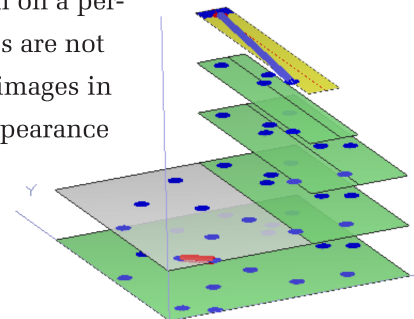


Figure A.2.2.1. Closest pair visualization.

The trailblazing work in algorithm animation was done by Marc H. Brown with his BALSA system. Brown formalized the idea of "interesting events" [Zeus 1992] and how they are embedded into the code of the algorithm being visualized:

> *An algorithm is annotated with markers that identify its fundamental operations that are to be displayed. These annotations, called interesting events, can have parameters that typically identify program data. Each view controls some screen real estate and is notified when an event happens in the algorithm. A view is responsible for updating its graphical display appropriately based on the event. Views can also propagate information from the user back to the algorithm.* [Brown 1992]

The notion of interesting events is important for two reasons. First, the presentation of these events communicates to the viewer of the algorithm the basic steps of the algorithm. They also serve to establish an appropriate and efficient granularity for processing the code which implements the algorithm. If no such events are used, then there may be a flood of unnecessary intermediate changes which may overload the animation system as well as the viewer's perceptual capabilities.

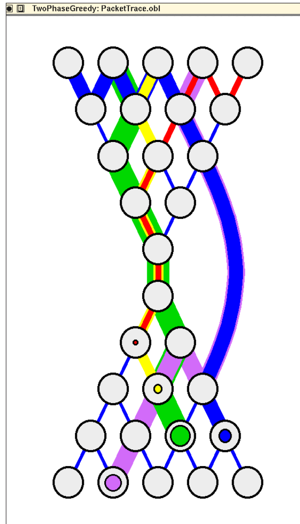


Figure A.2.2.2. Packet routing visualization.

Almost all algorithm visualization systems utilize animation. Even the original MacBalsa had basic animation capabilities. The goal of algorithm animation is to show the structure of the data being manipulated clearly, and animate the changes in the structure as smoothly as possible. "The Tango algorithm animation systems…emphasizes support for smooth animation between states in the visualization in order to improve, the quality of the animations and reduce the difficult with which animations are programmed." [Jeffery 1999 p.Citrin 1995] The Tango system developers introduced a path-transition paradigm which enables the exact description of an object's movement at the right time during the visualization. [Kehoe 1999] Strangely, when discussing animation, researchers in the field are still quite conservative with respect to it usage, in spite of the clear advantages.

> *Because smooth updates have durations, however, they are best suited for illustrating the fine-grained operations of a process working on a relatively small data set. If continuous animations are applied to large systems or data sets, they may simply take too long to occur and the viewer will become impatient.* [Kehoe 1999]

Fortunately, modern computers are able to run algorithm animation software fast enough such that any number of transitions can be visualized. Despite their instant appeal to all who witness their animated progressions, animated algorithm systems have been criticized for their applicability and for their effectiveness in improving learning. The simple argument against the applicability of algorithm visualizations is that they must be constructed one at

a time for each new algorithm. Additionally, the small set of elements and operators which one has access to when designing a visualization can seriously limit what can be expressed.

> *The main focus of the traditional visualization systems is on how to make concrete pictures, and they are customized for specific application domains. Therefore, the existing visualization systems cannot be used for a wide range of applications...There is a trade-off between expressiveness and generality of the visualization. If more specific and concrete pictures are given to programmers to represent the problem, the visual representation tends to be less general. [Koike 1991]*

In addition to this debate, there is a whole other argument about the correct way to embed or attach animation code to one's program code. Needless to say, it is always a taxing process, regardless of the algorithm.

Algorithm animation systems have been used extensively as interactive software aids within undergraduate level algorithms classes. Certainly, once one already understands the algorithm (i.e. is an expert with respect to that algorithm), then the animation generally makes much perfect sense. However, for a student who comes to the algorithm without any understanding of its working, the animation may not help. "Students just learning about an algorithm do not have a foundation of understanding upon which to construct the visualization mapping." [Kehoe 1999] This visual mapping from semantic information to visual structure is an essential part of diagrammatic understanding. Without a well-defined mapping established prior to the viewing of the animation, the viewer will quickly become lost. Reinforcing this mapping through a process of construction may help to alleviate this problem.

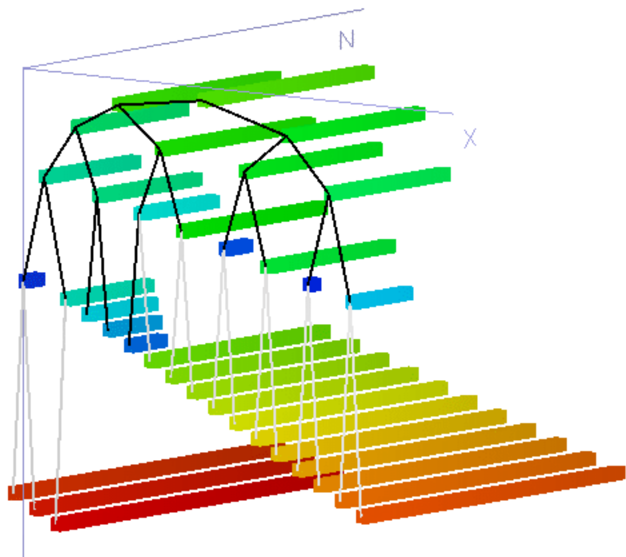


Figure A.2.2.3. Heapsort visualization.

Appendix B : Issues and Justifications

This section presents some theoretical issues which relate to the context of the Visual Machine design.

B.1 Justification of Bottom-up Approach

The programming systems which were designed for this thesis implement relatively low-level, non-domain specific languages. The reasons for taking such a low-level and general approach are twofold. First, the elements of computation which are presented here exist in all programming languages. For example, even the most high-level, domain-specific programming language will require some implementation of conditionality. Second, designing interactive representations of the low level machinery presented here is usually the most difficult aspect of creating a visual programming language. Doing the same, however, for the domain specific elements of a language is a simpler process, since these elements will most likely already have a natural visual representation and set of interactions.

B.2 Justification of Absence of Metaphor

The notion of the machine in the visual machine model serves two purposes: it acts as a visual reference for designing a dynamic system, and it acts as a philosophical model for thinking about the relationship between process and material. Machines should serve as a point of departure when designing a visual programming language, but not should not necessarily be used as a metaphorical object within the language itself. One should note that the visual machine model does not specify the use of metaphorical or analogical elements. Rather it simply specifies the two parts of the system, their relationship, and a design principle which describes their dynamics. While the objects within a visual machine are meant to move, react, and change in a mechanical fashion, they are not required to be metaphors for real-world machines or any other real-world entities. The Visual Machine Model is metaphor-neutral.

B.3 Literal vs. Magical Operation

When discussing his Alternate Reality Kit (ARK), Randy Smith of Xerox Parc recognized a "tension between the learnability of literalism and the power of magic." [R.Smith 1990] This discussion is equally appropriate in the analysis of the Visual Machine Model. Smith presents the argument as follows:

> *The designer of a system for use by novices can have great advantages by basing the interface on a known metaphor. If the computer behaves like a system already understood by the user, the learning time will be greatly reduced. Interface features that are true to the designer's metaphor might be called literal. The learnability of literalism makes it beneficial....However the designer can always provide the user with enhanced capabilities, assuming there is a willingness to break out of the metaphor. These features allow the user to do wonderful things that are far beyond the capabilities of literal features. Capabilities that violate the metaphor to provide enhanced functionality might be called magical. The power of magic is also beneficial.* [R.Smith 1990]

Unlike the Alternate Reality Kit, the Visual Machine Model makes limited use of real-world object metaphors. Hence, there are no specific semantics which must be observed. Still, there is a literalism in the behavior of the machines and materials which must not be violated. In a sense, the Visual Machine Model sets up an expectation of literal physical behavior. In order to support this expectation, one must maintain continuity and the relationship between material and machine, as well as exclude certain non-literal (i.e. magical) operations. Magical operations in a visual machine language include the instantaneous change of an object, or changes that are made without a clearly visible causes. Choosing between these magical options and literal behavior proved to be a difficult process in the creation of each of the Visual Machines.

Appendix C : Implementation

This section describes the basic methods of implementation for the five design experiments presented in Chapter 4. The features of the Visual Machine presented here represent the major building blocks as well as some of the finer details.

C.1 Underlying Architectures

All five designs are hand-programmed in C++ (a text-based programming language) and make use of the Aesthetics & Computation Group's proprietary 'acu' and 'acApp' libraries. These libraries, which are based on the OpenGL and GLUT libraries, serve as the underlying graphics, windowing, and event-handling architecture for all five designs. The most significant aspect of this architecture is that it supports continuous animation as a default behavior. In fact, programs created with these libraries must completely redraw their display on every refresh of the screen. What this means to the programmer is that there is no additional computational cost to animate an object versus having it remain stationary. In addition to supporting animation, the ACG libraries contain methods for drawing anti-aliased text among other features.

C.2 Object Structures

While there is little shared code between the five projects, there is a basic object structure used by all the designs. Each systems deals with a set of custom language elements (such as Turing nodes or Plate variables), which are represented as standard C++ objects. These objects maintain the identity of the language element, the associated state, as well as connections that it may have to other objects. These objects are also responsible for displaying the visual representation of the respective language element. In most of the designs, there is a master list of the top-level objects and subordinate objects (such as Plates within Plates) are maintained in a hierarchical tree structure supported the objects themselves.

C.3 Interaction Methods

To enable interaction, the systems always one maintain one active (i.e. selected) object. This active object is the only object that can be altered. When the active object is dragged away from its current location, it is usually detached from the hierarchical structure in which it rests. When the user releases the active object, it is reattached to the hierarchical structure, perhaps in a different location. When at rest, the active object may receive information about the mouse position or keyboard activity in order to change its own state.

C.4 Integrated Evaluators

In all five Visual Machines, the evaluation methods are always built directly into the language element objects. In a sense, there is no distinction between the interface and actual functionality, since the interface itself contains all of its own functionality. This tight integration allows all of the state of the computation to be directly accessible to the visualization. In general, the computation proceeeds in a bottom-up fashion through the hierarchical structures. Since no parsing needs to be done (the grammar is defined by the visual connections), the process is similar to evaluating a pre-computed parse-tree. All intermediate data is stored directly in the language elements, rather than in a set aside table. The result is a relatively responsive and efficient means of evaluation.

C.5 Discrete Transitions Made Continuous

One of the most notable aspects of the Visual Machine designs is the way in which discrete transitions become continuous transitions during evaluation. This process is accomplished by associating a continuous floating-point number variable with every discrete transition. This variable is initially equal to 0.0 and slowly progresses to 1.0 over the course of the transition. The number is also used to determine the changes in position, color, and form of the active object between the initial state and the end state. Normally, these visual changes occur through a process of blending between two known values. Only when the transition variable reaches 1.0 will the next transition be processed. By adjusting the

size of the incrementation variable, one may control the speed of the execution. An increment of 0.01 will require 100 frames while an increment of 0.5 will requirest just two.

C.6 Instances

Perhaps the most complex feature in all five Visual Machines is the instancing that is used in Plate variables and Pablo functions. An instance of an object essentially mirrors the state of another object. For example, when a name is changed on one Plate variable instance, all equivalent instances automatically update their names because they are mirrring the same original variable. Likewise, when a change is made to one Pablo function instance, all other instances update themselves to reflect that change. The difficulty in the implemenation of instances lies in the fact that all instances have two sets of state: their local state, and their instance state. The local state of a Plate variable is the plate's position, and whether it is expanded or collapsed. The instance state is the name of the variable and the variable's value. If an instance and the original object become out of sync, then the instance is responsible for updating it state. Doing this efficiently for complex data objects, such as lists and structures, requires a well designed architecture.

C.7 Platforms

The development of the five Visual Machines took place over a two year span, which was interrupted by many side projects. The first year of development was performed on an SGI Octane, running the IRIX operating system, the EMACS text editor, and the CC compiler. An Apple Macintosh G4 running Metrowerks CodeWarrior served as the programming platform during the second year of development.

References

[Abelson 1996] Abelson, H and Sussman, G. Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, 1996

[Atwood 1996] Atwood, J.W, et al. *Steering Program Via Time Travel*, Proceedings of the 1996 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1996.

[Bertin 1963] Bertin, Jacques. Semiology of Graphics: Diagrams, Networks, Maps. University of Wisconsin Press, Madison, WI, 1983.

[Blackwell 1999] Blackwell, A.F, et al. *Does Metaphor Increase Visual Language Usability*, Proceedings of 1999 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1996.

[Bonar 1990] Bonar, J, et al. *A Visual Programming Language for Novices*, appears in *Princicples of Visual Programming Systems*, ed. Shi-Kuo Chang, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Brown 1992] Brown, M. *Zeus: A System for Algorithm Animation and Multi-View Editing*, Digital Equipment Corporation, 1992.

[Burnett 1999] Burnett, M. *The Future of Visual Languages*, Proceedings of 1999 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1999.

[Card 1999] Card, S, et al. *Readings in Information Visualization: Using Vision to Think*, Morgan-Kaufmann Publishers, San Francisco, CA, 1999

[Carlson 1995] Carlson, P, et al. *Integrating Algorithm Animation into Declarative Visual Programming Language*, Proceedings of the 11th Symposium on Visual Languages, IEEE Computer Society Press, 1995.

[Chang 1990] Chang, S.C, et al. *Princicples of Visual Programming Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1990

[Citrin 1995] Citrin, W, et al. *Programming with Visual Expressions,* Proceedings of the 11th Symposium on Visual Languages, IEEE Computer Society Press, 1995.

[Costagliola 1997] Costagliola, G. et al. *A Framework of Syntactic Model for the Implementation of Visual Languages*, Proceedings of the 1997 Symposium on Visual Languages, IEEE Computer Society Press, 1997.

[Cox 1990] Cox, P.T, et al. *Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism*, appears in *Visual Programming Environments : Paradigms and Systems*, ed. Ephraim P. Glinert, 1990.

[Demetrescu 1999] Demetrescu, C, et al. *Smooth Animation of Algorithms in a Declarative Framework*, Proceedings of 1999 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1999.

[DiSessa 2000] diSessa, A. *Changing Minds: Computers, Learning and Literacy,* MIT Press, Cambridge, MA, 2000

[Duecker 2001] Duecker, M, et al. *An Introduction to Pictorial Janus*, <http://jerry.c-lab.de/~wolfgang/PJ/introduction.html>, Heinz Nixdorf Institut, 2001.

[Edel 1990] Edel, M. *The Tinkertoy Graphical Programming Environment*, appears in *Visual Programming Environments : Paradigms and Systems*, ed. Ephraim P. Glinert, 1990.

[Freeman 1995] Freeman, E, et al. *In Search of a Simple Visual Vocabulary*, Proceedings of the 11th Symposium on Visual Languages, IEEE Computer Society Press, 1995.

[Gardin 1989] Gardin, F. and Meltzer, B. Analogical Representations of Naive Physics, Artifical Intelligence, 38, Elsevier Science Publishers, 1989.

[Geiger 1998] Geiger, S, et al. *SAM - An Animated 3D Programming Language*, Proceedings of the 1998 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1998.

[Glinert 1990] Glinert, E. *Pict: An Interactive Graphical Programming Environment*, appears in *Visual Programming Environments : Paradigms and Systems*, ed. Ephraim P. Glinert, 1990.

[Glinert 1990] Glinert, E. *Non-Textual Programming Environments*, appears in *Princicples of Visual Programming Systems*, Shi-Kuo Chang, editor, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Gooday 1996] Gooday, et al. *Using Spatial Logic to Describe Visual Languages*, 1996.

[Griebel 1996] Griebel, P, et al. *Integrating a Constraint solver into a Real-Time Animation Environment*, Proceedings of 1996 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1996.

[Haarslev 1995] Haarslev, V, et al. *Visualization of Strand Processes*, Proceedings of the 11th Symposium on Visual Languages, IEEE Computer Society Press, 1995.

[Haarslev 1995] Haarslev, V. *Formal Semantics of Visual Languages Using Spatial Reasoning,* Proceedings of the 11th Symposium on Visual Languages, IEEE Computer Society Press, 1995.

[Heltulla 1990] Heltulla, E, et al. *Principles of Alladin and other Algorithm Animation Systems*, appears in *Visual Language and Applications*, ed. T. Ichikawa, Plenum Press, New York, 1990.
[Huang 1990] Huand, K.T, *Visual Interface Design Systems*, appears in *Princicples of Visual Programming Systems*, ed. Shi-Kuo Chang, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Ichikawa 1990] Ichikawa, T, et al. *Visual Language and Applications*, Plenum Press, New York, 1990.

[Igarashi 1998] Igarashi, T, et al. *Fluid Visualization of Spreadsheet Structures*, Proceedings of the 1998 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1998.

[Jeffery 1999] Jeffery, C.L. *Program Monitoring and Visualization: an exploratory approach*, Springer-Verlag, New York, 1999.

[Kahn 1996] Kahn, K. *Seeing Systolic Computations in a Video Game World*, Proceedings of 1996 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1996.

[Kehoe 1999] Kehoe, J, et al. *Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study,* Georgia Institute of Technology, Atlanta, GA, 1999.

[Koike 1995] Koike, H, et al. *A Bottom-Up Approach for Visualizing Program Behavior*, Proceedings of the 11th Symposium on Visual Languages, IEEE Computer Society Press, 1995.

[Liu 1996] Liu, Z. *A System for Visualizing and Animating Runtime Histories*,Proceedings of 1996 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1995.

[Lindsay 1996] Lindsay, R.K, I*mages and Inference*, appears in *Diagrammatic Reasoning : Cognitive and Computational Perspectives*, Glasgow, et al, MIT Press, Cambridge, MA 1996

[Maeda 1999] Maeda, J. *Design By Numbers*. MIT Press, Cambridge, MA, 1999.

[Myers 1990] Myers, B. *Creating Interaction Techniques by Demonstration*, appears in *Visual Programming Environments: Paradigms and Systems*, ed. Ephraim P. Glinert, 1990.

[Norman 1993] Norman, D. A. *Things that make us smart.* Addison-Wesley, Reading, MA, 1993.

[Reiss 1990] Reiss, S. *Working in the Garden Environment for Conceptual Programming*, appears in *Visual Programming Environments: Paradigms and Systems,* ed. Ephraim P. Glinert, 1990.

[Repenning 1997] Repenning, A. *Behavior Processors: Layers Between End-Users and Java Virtual Machines*, Proceedings of the 1997 Symposium on Visual Languages, IEEE Computer Society Press, 1997.

[Searle 1980] Searle, John R. *Minds, Brains, and programs*, appears in *The Behavioral and Brain Sciences,* 3, 417-457, 1980.

[Sheridan 1992[Sheridan, T. *Telerobotics, Automation and Human Supervisory Control*, MIT Press, Cambridge, MA, 1992.

[Shu 1990] Shu, N.C, *Visual Programming Languages : A Perspective and a Dimensional Analysis*, appears in *Visual Programming Environments: Paradigms and Systems*, ed. Ephraim P. Glinert, 1990.

[R.Smith1990] Smith, R, *Experience with Alternate Reality Kit: An Example of the Tension Between Literalism and Magic*, appears in *Visual Programming Environments: Paradigms and Systems*, ed. Ephraim P. Glinert, 1990.

[D.C.Smith 1999] Smith, D.C, *Pygmallion: An Executable Electronic Blackboard*, appearss in *Watch What I Do*, <http://www.acypher.com/wwid/Chapters/Pygmalion.html>, 1999.

[D.C.Smith 1996] Smith, D.C, *VL'96 Special Event: Perspectives from the Pioneers, David Canfield Smith*, Proceedings of 1996 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1996.

[Stasko 2000] Stasko, J, et al, *Software Visualization: Programming as a Multimedia Experience*, MIT Press, Cambridge, MA, 2000.

[Szwillus 1996] Szwillus, G. *Structure-Based Editors and Environments*, Acadamic Press, New York, 1996.

[Travers 1996] Travers, M. *Programming with Agents: New metaphors for thinking about computation*, PhD Thesis, MIT Media Laboratory, 1996.

[Tufte 1990] Tufte, Edward R. *Envisioning Information*. Graphics Press, Cheshire, Conn, 1990.

[Tufte 1992] Tufte, Edward R. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Conn, 1992.

[Turing 1937] Turing, Alan M. *On computable numbers with an application to the Entsheidungsproblem*. appears in *Proceedings of the London Mathematical Society*, ser. 2, vol. 42, 1937.