Fear of Code
An Approach to Integrating Computation with Architectural Design

by Nicholas Senske

Bachelor of Architecture (2003)
Iowa State University

Submitted to the Department of Architecture
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Architecture Studies
at the Massachusetts Institute of Technology

June 2005

© 2005 Nicholas Senske. All rights Reserved

The author hereby grants to MIT permission to reproduce
and to distribute publicly paper and electronic copies
of the thesis document in whole or part.

Signature of Author:
Department of Architecture
May 19th, 2005

Certified by:
William Lyman Porter
Leventhal Professor Emeritus,
School of Architecture and Planning
Thesis Advisor

Accepted by:
Julian Beinart
Professor of Architecture
Chair, Committee on Graduate Students
Fear of Code
An Approach to Integrating Computation with Architectural Design

by Nicholas Senske

Submitted to the Department of Architecture
on May 19, 2005
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Architecture Studies

Abstract:

Computation has enormous creative potential for designers. Unfortunately, many factors, such as the considerable investment required to learn programming, have prevented it from entering into common use. By analyzing the barriers to working with computation and suggesting directions for future research and implementation, this thesis seeks to enable a more inclusive dialogue towards the eventual integration of computation into the architectural mainstream. To this end, I propose a more responsible relationship between architects and their software through a combination of interface improvements, code analysis and management tools, and the formation of open source communities for support and development.

Thesis Supervisor: William Lyman Porter
Title: Professor Emeritus, School of Architecture and Planning
Thesis Reader: John Maeda
Title: Associate Professor, Media Arts and Sciences

Thesis Reader: Yanni Loukissas
Title: PhD Scholar, School of Architecture and Planning
ACKNOWLEDGEMENTS:

I wish to thank:

My advisor and readers:
Bill Porter for his wise council, patience, and experience
John Maeda for his lessons, his example, and his charity—"you matter"
Yanni, for being an attentive and supportive reader

Scott, my partner and my own personal Luddite (every technologist should have one)
Nancy Jones and the rest of the administrations staff for their help
and the SMArchS Computation class of 2005 for their friendship

To Jacqueline,
for being my inspiration and for showing me that there is more to life than computers

To my parents,
for their love, sacrifice, and support over the years
Table of Contents

1 Introduction
1.1 Context
   1.1.1 Economy
   1.1.2 Problem Statement
1.2 Intentions
   1.2.1 'Fear of Code'
   1.2.2 Scope
1.3 Necessity

2 Background
2.1 Programming
2.2 User Interfaces
   2.2.1 Command Line Interface
   2.2.2 Graphical User Interfaces
2.3 Programming Interfaces
   2.3.1 Integrated Development Environments
   2.3.2 Domain Specific Languages
   2.3.3 Visual Programming Languages

3 Related Work
3.1 Pedagogical Programming
   3.1.1 LOGO
   3.1.2 LegoLOGO
   3.1.3 Design by Numbers
3.2 Current Trends in Architectural Computation
   3.2.1 AutoLISP, RhinoScript
   3.2.2 Generative Components
   3.2.3 Design Tooling
3.3 Future Trends
   3.3.1 FormWriter
   3.3.2 Processing
   3.3.3 OpenAtelier

4 Challenges
4.1 Programming as Expression
4.2 Interface Issues
4.3 Design Principles
   4.3.1 Transparency
   4.3.2 Flexibility
   4.3.3 Dialogue
   4.3.4 Play

5 Improvements
5.1 Infrastructure/ Community
5.2 Code Management
5.2.1 Modularity
5.2.2 Analysis
5.2.3 Translation
5.2.4 Documentation

5.3 Design History
5.4 Evaluation Interface

6 Pedagogy
6.1 Identity Concerns
6.2 ‘Hello World’: Scenarios for Learning
6.3 A Culture of Computation

7 Conclusion

Bibliography
CHAPTER 1: INTRODUCTION

1.1 Context

Architecture is unique in its ability to assimilate the knowledge and techniques of other fields. Similarly, the computer is a receptive and malleable medium, capable of emulating almost anything that can be specifically described. However, because of this very flexibility, it has been difficult for computers today to be recognized independent of these associations. While graphical interfaces have led to an increase in computer usability, they do so by obscuring the presence of the machine. The computer, made ubiquitous by metaphor, is effectively no longer a computer. It is a drafting board, a painter’s canvas, a photo lab—in every scenario computation is present, yet absent, hidden beneath a layer of abstraction. This is not without precedent. Historically, before a medium is well understood, it is common to interpret it in terms of previous, better-defined media. In the early days of television, for example, the first actors were radio commentators. They were accustomed to narration and would describe their actions on camera to the audience. In the same way, using a computer to draw as one would with a pencil and paper does not utilize its full expressive potential. Just as architecture maintains its own identity to function apart from the fields it borrows from, designers must work to define the medium of computation as something distinctive, yet emerging from, that which it takes into itself—only then can its true potential emerge.

While many designers engage the computer in their work, very few do so from a computational perspective. Today’s digital culture is the result of the successful adaptation of the metaphors of the desktop, files, typewriter, mail, and so forth, to computer applications. In the field of architecture, a similar transition has been made, reproducing traditional modes of
representation as software. Consequently, the computer has found its way into almost every stage of the design process, becoming an indispensable part of contemporary practice. This translation of previously non-digital operations into digital tools will be referred to as computerization for the purposes of this thesis. The distinction made between computerization and computation is an important one. Working with computerized applications remains very human-intensive, often requiring significant direct manipulation. In comparison, computation is machine-intensive, executing procedures with such speed and quantity, often according to complex conditional rules, so as to produce things which are impossible for unaided humans due to constraints of time and attention. From this powerful idea, a different order of tools—indeed, a different way of thinking about design—becomes possible.

Conceptually, computerization involves preconceived or well defined notions, whereas computation is a more exploratory practice: It is about identifying and expressing our thought processes and using the power of the machine to carry them further. The result is a form of problem solving that can lead to unexpected solutions, rather than simply visualizing our expectations or automating outcomes (Terzidis 2003). But developing computational skill is difficult. Unfortunately, designers do not have much experience thinking about or expressing their intentions procedurally, and so the exciting potential of computation has mostly escaped the practice. However, with sufficient motivation this may soon change.

1.1.1 Economy

Before changes can take place in the profession there must be some justification for the expense of investing in new technology and training people to use it. When describing the

---

1 It should be noted, that as an idea, computation does not exclusively require the use of machines. However, within this thesis, computation specifically refers to its application by digital computers.
practical reasons architects might engage in computation, it may be useful to speak in terms of economies.

In the past decade, improvements in the areas of graphics hardware, modeling tools, rapid-prototyping, and fabrication (among others) have brought about a new wave of "digital" architecture that has captured the interest of the field. These new projects are said to be "computer-derived", but the present reality is that the computer is not as responsible for envisioning these new forms as much as it is for making them more affordable. Architects have long experimented with complex shapes on paper and in models, but only recently have computers brought a new economy of production to such "non-standard" work, reducing the cost of manufacture and construction enough for them to be realized.

For example, Frank Gehry, whose projects are most often associated with digital design, begins his unique geometries as physical models. While it is true the forms are later scanned and manipulated in the computer, this part of the process is mostly for logistical purposes. The software Gehry’s team uses allows them to efficiently develop and refine these surfaces for fabrication and accommodate changes in the design, reducing cost and maintaining the constraints of the budget (Loukissas 2003). Thus, digital models are used for the confirmation, manufacture, and delivery of known forms. While nonetheless critical to the realization of the project, these practices represent a computerized, rather than computational, process. The often misunderstood lesson of Gehry’s work is: computers themselves do not make creative forms, they enable them.

This fact is by choice more than by circumstance. Historically, the profession saw computers as an opportunity to place greater control in the hands of the architect by digitizing outside specializations and production routines, therefore freeing up time for more creative pursuits. While researchers saw that the computer had its own creative possibilities, the majority of architects had no desire to attempt such automation. They believed (and still do) that design was the most integral, most human act in the whole process—the one
thing that should never be given up to the computer. And so, direct manipulation, firmly under the control of a human architect, continues to be the order of the day.

Ironically, this decision has allowed others to control how we design, albeit in subtle ways. Our tools often define our capabilities and influence our ways of thinking. The proprietary software many designers use in their work contains a rigid structure of procedures and workflows; problem solving is often reduced to knowing the proper command or sequence. This can impose unnecessary restrictions on a designer’s process, discouraging original solutions beyond the scope of the program and promoting a dependence upon rote-knowledge over a rigorous conceptual understanding. There is an economy of representation that affects our choices. Simply stated, the more difficult something is to visualize, the less likely it is to appear in a design, and vice versa. Thus, software has the capacity to shape how we approach design. But the computer itself is a general machine, and is never bound to a single interpretation. Through programming, a designer can potentially overcome the limitations of pre-existing software and develop solutions that have yet to be generalized.

This change is not far off; in fact, it is already occurring. The techniques that fashionable architects like Gehry use are in demand and in the process of being assimilated by the pressures of the marketplace, making them cheaper and more widely available. The availability of affordable and powerful desktop machines means that almost any office has the computer hardware needed to take advantage of these methods. Students are being educated with the latest software programs to ensure they can work with these tools almost as soon as they leave school. At the same time, the construction industry is gaining experience with new machines and techniques that make use of data provided by architects. The convergence of these factors suggests that the market for some computerized processes is well on its way to being saturated—the tools are no longer enough to distinguish. As Bruce Mau

---

2 Besides, if design was somehow taken over by computers, there would be nothing more for architects to do.
Nicholas Senske
*Fear of Code*

wrote in *Life Style*: "The problem with software is that everyone has it." (2002). This might be called the *economy of originality*. Early adopters of the technology are becoming increasingly aware of their decreasing specialization, and are searching for new creative avenues. The more traditional of these are looking towards developing their own ‘autographic’ software with unique capabilities for use in their firms. Still others have turned towards ‘allographic’ or emergent computational systems that avoid such notions of authorship (Lynn, Rashid 2002). Regardless of the divergence of these approaches, the overall trend suggests that computation is likely to become a greater topic of discussion in the near future.

To summarize, there are several practical reasons why architects might become interested in computation. In shortened form, these are: efficiency, control, and novelty. Offices and architects with the resources are in the process of developing their own computational solutions and some will never be persuaded of their necessity. The discipline must strive to help those who find themselves in the middle, only then will computation fall into general use.

1.1.2 Problem Statement

Computation has the potential to play a much greater role in architecture, but there are fundamental issues of infrastructure and culture which must be addressed for a mainstream practice to emerge.

The need to know programming seems to be one of the primary factors preventing designers from engaging in computation. For most people, the prospect of learning a language based on principals of math and logic seems a daunting task. Teaching oneself is a considerable investment for anyone, and obtaining a formal instruction in programming is not a realistic option within the strict requirements of a professional design education. If coding is too difficult, it
would appear to follow that the right application or applications could make things easier, but this is not necessarily true or desirable. Historically, programming does not lend itself well to visual metaphors and efforts to promote a more user-friendly experience tend to rob computation of its rigor. Somehow, a balance must be struck to preserve the best attributes of human and machine, rigor and usability. Thus, the primary concern of this thesis is assessing the potential for a new computational interface for architects.

Beyond creating new tools is the issue of the relationship architects will have with them. Despite our best efforts, not all designers will be able to master computation, but it is important that they be aware of it. Architects should be knowledgeable enough to communicate as effectively with programmers as they would with any of their consultants. In addition, like any technology, there are patterns of proper use to be learned and even ethical considerations to be made. It is vital that architects recognize the advantages and disadvantages so they may apply it where it is most appropriate. Of course, these are deep issues of pedagogy not easily resolved, but infrastructure and learning materials can have an influence. This thesis considers how such resources might be designed so they may encourage users to make good choices with computation.

Along with the question of what to teach is the question of how to teach most effectively. Computation requires more to learn that mere rote operation: it is a disciplined way of thinking about design. This puts it at odds with the current application-centered approach schools have towards computing. What is more, technology moves far too quickly for even this kind of education to stay relevant for long—yet focusing on concepts has traditionally been difficult. Firms want technically capable hires that can be productive as soon as possible. With pressure from job-seeking students, schools are obliged to train them. Finding ways to cultivate both the discipline and skill necessary to use computation will be a challenge.
Finally, once more architects learn to program, some of them will begin to produce their own tools. With the right encouragement and liberties, such an effort could greatly benefit the field. This is a long-term scenario, but as educators and researchers, we should start to consider this possibility now. Realistically, most designers will not become accomplished programmers, but nevertheless, they have something to contribute. Ideally, every designer might someday take responsibility for their tools and take part in the development process. Whether this occurs or not, intellectual property rights will soon become a major concern for architects, both in terms of the software we use and the designs that we make. If the software industry is any indicator, serious potential exists for abuse. Long before damaging policies are dictated and enforced from an outside body, it may be possible to prevent this from happening by establishing community standards that support both creative freedoms and protect one’s work. Creating and sustaining an open altruistic culture that encourages sharing, peer review, and user contributions is a most important charge for the future.

1.2 Intentions

Architecture already has the tools for visualization, what it needs are more tools to solve problems and extend our intellect. I strongly believe that if much of our work is to involve computers, we should have an understanding of computation to get the most out of them. In order to accomplish this, design computation cannot remain an esoteric pursuit relegated to a few graduate programs. Somehow, a computational perspective must become a part of the standard architectural education.

The intended audience for this thesis is primarily computational design researchers, like myself, who are interested in the challenges of sharing and applying our ideas in a broader context. For those new to computation, especially if programming is not their specialty, it might be a good introduction to the concepts of working with code, and perhaps
could offer some inspiration for future usability projects. Professionals might want to read about future trends for the field. Educators and administrators may be interested in some of the suggestions from a later chapter about pedagogy. While some groups may be more inclined to examine my work than others, the important thing is to have these ideas compiled into one source, so they can be seen and debated further.

The idea for this thesis came about as a result of my own experiences as a designer attempting to learn computation and my desire to make the process easier for others. I started by asking myself how a school like the regional Midwestern I came from could begin teaching computation. This model served as my inspiration because it was already familiar to me, and I felt that it was representative of the challenges that the discipline would face moving outside of its present havens. Realistically, making sweeping changes to curricula is not feasible. A better strategy is not to directly intervene, but to find ways to motivate change, support growth, and let the situation evolve. Therefore, what I am seeking with this work is to lower some of the barriers that make practicing computation so difficult for the novice, and to provide the resources for faculty and students to connect to the discourse and to each other.

When I set out to solve this problem originally, I naïvely tried to develop my own, simpler programming language for architects. I have found through my research that this line of inquiry was in error. Indeed, I discovered that there is no fixed solution (nor is one desirable) and the answers that do exist are difficult to accept. I hope that with the information I have provided in later chapters, the reader will reach the same conclusion and agree with the new direction I chose. The problems facing the adoption of computation are far beyond the scope of a single thesis. For this reason, I decided my contribution would be to help lay the groundwork for others: defining promising areas of future research and suggesting some practical, if minor, improvements that could be made. Lastly, I wanted to communicate the need for leadership in an effort as ambitious as this one. Although the final goal is to
create a self-sustaining culture, someone must set the tone for what is to follow.

After evaluating this thesis, I hope that at a minimum, the reader will learn something about computation and why it is such a difficult subject. Ideally, I want them to come away with a sense that they can contribute to the changes I propose, and possibly inspire them to take some action.

1.2.1 Fear of Code

"A computer is like an Old Testament god: a lot of rules and no mercy." -Joseph Campbell

Immediately, fear of code calls to mind the apprehension one feels when confronted with the darkest aspects of the computer: lost data, frozen systems, cryptic error messages, and other occurrences that plague our systems. Here, the façade of the user-friendly interface falls away and the machine is exposed. This frightening place is where code ‘lives’. It is no wonder that most people feel intimidated by the idea of venturing into such unfriendly territory. The frustration over lost work or the incomprehensibility of a foreign language are not unique to the computer, nor are they specific to architecture—there is more here than mere inconvenience.

On some level, computers represent a challenge to our identity and humanity, and, as an instinct, fear is supposed to protect us. Thus, by fearing computation, our intention as architects is to protect ourselves personally and professionally. Faced with

---

3 Incidentally, the title of my thesis, Fear of Code, is a reference to one of the Judeo-Christian virtues: “fear of God”. This phrase has always stuck with me—how can fear be considered a virtue? Taken literally, it would appear to refer to the threatening and overwhelming presence of a vengeful God, but, according to theologians, fear actually means the ‘right’ relationship with God: one of reverence and awe. With due respect to theology, perhaps by cultivating awareness of computation we can move away from our initial fears and towards a more enlightened relationship with the computer.
the prospect of yet another new technology, architects are concerned with the dangers of increased specialization and the demands of learning new skills. For many, the investment does not appear worth the effort. Learning new things is seldom easy and uses up valuable time. Those who become particularly skilled have more value to organizations as technicians and are often used for production more than design. In the long term, many worry about the potential for computation to undermine the value of the architect through automation and artificial intelligence.

None of these ideas are very new; they have been around since modern computers and the early days of CAD. But this does make them well-ingrained within our culture and difficult to dismiss. We must remember that the search for answers does not take place in a vacuum. For our work to operate successfully outside of academia, it is critical that computational researchers and educators examine their assumptions and strive to develop empathy for their audience.

Any technological change is also a cultural change—improving usability is only half of the answer. Better tools can help, but they can only do so much if there is no desire to learn. While there are technical issues to be overcome, there is also social conditioning to be undone. The boredom and fear we feel with our software do more to hurt our work than any bug or lack of features. As Malcolm McCullough wrote: “Even more important than how to use a computer is how to be when using a computer.” (1996).

1.2.2 Overview and Scope

The organization of the thesis is divided into two halves. The first sets up the necessary background for the reader and ends with a more comprehensive definition of the problem, while
the second discusses some possible strategies and potential avenues for further research⁴.

After the introduction, the second chapter lays the foundation for the reader by providing a general overview of computer interfaces and the application of computation. It begins by introducing the various concepts and classifications of human/computer interfaces that will be referred to throughout the rest of the thesis. The purpose here is to make the reader aware of the strengths and weaknesses of these approaches so they may be critically examined in later examples.

There have been many attempts to introduce computation to non-programmers (in some cases, specifically designers). In the third chapter, these types of programming environments are included in a review of precedents, along with scripting and specialized computational interfaces in next-generation software. Finally, current research and design trends will be examined to imply future directions for these kinds of systems.

After the formal introduction of the field and its works, the three chapters that follow comprise the bulk of the thesis, presenting the mainstream adoption of computation as a design problem and suggesting guidelines for the development of new tools and pedagogy. The first of these chapters concerns itself with an in-depth discussion of the technical and conceptual bottlenecks researchers face in developing programming-related tools and the difficulties designers have in learning to code. In the next chapter, I respond with a list of design principals that address these challenges and point toward some areas of research I believe show promise. In the interest of visualizing the possibilities, each of these areas is accompanied by a description and illustrations of a speculative tool or interface. Unfortunately, this cannot be enough; even the best tool fails if no one wants to (or cannot) use it. Thus, pedagogy is the topic of the final design chapter.

---

⁴ It is necessary in a thesis to distinguish between the work of others and the writer’s original work. The division between the two sections described here delineates this split.
It is the finding of this thesis that the complexity and vagaries of design and the breadth of computation suggest the penultimate solution is not a solution, but a process—success will rest on the individual and collective decisions designers make. The thesis concludes with a discussion of this idea in light of future directions this work might take.

1.3 Necessity

"The best way to predict the future is to invent it." - Alan Kay

Computation in architecture did not used to be such a foreign concept. Previous generations of architects needed some knowledge of programming to make use of the rudimentary CAD systems that existed at the time. As a result, they had a different understanding of working with the computer than we have today. Because of modern graphical user interfaces, ours is the first generation to grow up without the benefit of this understanding. Now that computer fluency is commonplace, designers are once again looking to push the envelope. Turning upon the establishment, the response of the avant-garde is, in a way, a return to former methods: towards a computational perspective.

This presents the field of computation with an opportunity, an emerging audience for the discourse which has been built up for decades, but has until now remained mostly academic and isolated. In newly built projects and the architectural press, the products of collaboration between architects, mathematicians, and programmers are building interest in the architectural community. Inspired by these designs, a growing number of students here at MIT are experimenting with the possibilities of code in their work, as I imagine they are elsewhere. The timing of this thesis is a key factor, for the momentum of these ideas is growing, but their direction is yet uncertain.

Progress does not occur by itself; someone must intercede on its behalf. Without intervention, software companies will seize upon the demand for new computational tools and lock
designers into proprietary arrangements, defeating much of their promise upfront. To this end, I propose that the traditional practices of software companies be left behind. We must be brave: the best solution is not always the most used or easiest. Our goal must be to empower individual choice, and to promote the path that leads to the most possibility; openness is never a given, it is a decision. As designers, researchers, and educators experienced in computation, we must step up and take responsibility: design is a form of leadership. Rather than wait and respond, we must begin—now—to envision what a positive pursuit of computation might look like and take steps toward it.
Chapter 2: BACKGROUND

Generally speaking, interfaces exist to associate machine control functions and output with abstractions humans can understand. People do not interact with computers directly; they interact with monitors, input devices, and programs. The conceptual separation this arrangement represents is critical. Interfaces serve to hide the machine of the computer from us, trading rigor for ease of use. Finding ways to improve the usability of computation, while preserving its rigor, continues to be a challenge. The purpose of this section is to reveal some of the strategies employed to address this problem and to provide a common framework from which to examine other kinds of interfaces presented in the thesis.

2.1 Programming

"It is always difficult to think and reason in a new language; and that difficulty discouraged all but men of energetic minds"

-Charles Babbage

The vast majority of computer users today are not programmers, but this was not always the case. In the early days of mainframes, programming was the only way to interact with the computer, albeit asynchronously. A user would submit their instructions in code, which used to be in the form of punch cards or tape, for computation. Because the same computer had several users, each request or “batch” often had to wait until the machine was free. The instructions would be read into the machine and the result could be picked up later by the submitter. This sequence was known as “batch processing”. Later, terminal machines replaced the physically recorded instructions with remote stations that somewhat resembled today’s desktops. Code or other commands were written on the screen with a keyboard and submitted to the mainframe over a network. This was known as “time-sharing”. For the most part, systems similar to these were the standard
until the development of the personal computer in the late 1970’s.

Generally speaking, computer operation is fundamentally the same as it was back in the early days, only much faster. Instructions and responses are relayed through graphical interfaces rather than code. Instead of waiting for time on a computer, we each have our own machine capable of simultaneous computations. Today, these processes happen so quickly that most people have no idea what is going on behind the scenes of their computer. Looking back at old systems is more than nostalgia—in a sense, it illustrates the basic idea of how computing works.

As our history lesson revealed, working with code is not so far removed from how we use computers today. If programming is another kind of interface, it might be useful to compare it to others that are more familiar to us. When discussing interfaces, it can be useful to speak of them as being “low-” or “high”-level. This is a common metric for the amount of abstractions a given interface employs in relation to another. As a continuum, the lowest possible level for an interface would be binary switches and the highest would be natural human language. It follows that the higher-level something is, the easier for the average person to understand. This is good to keep in mind, but it is not always the rule. Determining where to place a particular interface on this scale can be a matter of context. Programming languages in general are low-level to the average user, but some implementations can be considered high-level if they are developed for a specific audience. This illustrates another important point: generalizing interfaces can be useful, but there are many gradations possible within those types.

Programming is quite different from the kinds of interfaces most of us are accustomed to, not only because of how it is used, but because of what it is capable of doing. Interfaces today function by letting us choose things we would like to do, but if something we need is not defined by the system, we are out of luck. Programming has no such restrictions, but it can be hard work. Ideally, we could simply tell the computer what we
want to do and it would perfectly respond. Of course, such a thing is not currently possible, for reasons which will be better explained later. For now, it should be sufficient to say that it remains significantly easier for humans to learn machine language than for us to teach computers our own. This is a difficult arrangement, but it has served us well. If we can manage to write out in code exactly what we want the computer to do, it will execute those instructions perfectly. While it might not seem like much, this is actually one of the most significant properties of computation: explicitness.

But what makes programming powerful is also what makes it difficult for many people to understand. Specifying something exactly requires both intimate knowledge of the subject and clarity of documentation, things which do not come easy. Describing an activity procedurally, with perfect detail, is not a common activity for most fields, especially if it is something difficult to define objectively. For this reason, math or physics are much easier to put into computation than more subjective disciplines like art or architecture. But there are advantages to procedural logic that make doing so worthwhile. Once something has been defined, it can be reused exactly with almost no effort compared to the original task of writing the code. In this manner, complexity can grow tremendously by building upon previous complexity. This is referred to as time-compression, thus fulfilling the purpose of the machine—this is how the computer saves work.

Explicitness enables the other unique property of computation: programmability. A famous theorem called the Church-Turing hypothesis states that a perfect digital computer\(^5\) can, in effect, become any machine that can be expressed as logical operations. This hypothetical construct is sometimes referred to as a Universal or Turing Machine. In application it is an enormously powerful idea. Indeed, all of modern computing is based on programmability. Essentially, a computer could be thought of as blank slate waiting to be imprinted with potential new virtual machines. Given the right instructions, a computer

\(^5\) This is a hypothetical construct: given unlimited memory, unlimited time to operate, perfectly explicit instructions, etc.
can become a movie player, a paint program, or a video game. Also, because all instructions must be reduced to logical operations, translation is possible: audio can become video, color can become text, etc. Combining the properties of explicitness and programmability makes the computer a truly amazing machine; programming allows one to exploit it to the fullest.

For the computer to do anything it must first be completely defined—otherwise, all of those bits are meaningless. Computers begin and end with logical operations, represented by a series of 1’s and 0’s known as binary code. Special combinations of these comprise machine code which defines the most basic functions of computation such as storing and reading bits. A simplified equivalent of these instructions is called assembly language. This is usually the lowest-level of code a programmer encounters. Hand-written assembly can be very fast and makes efficient use of resources, but in practice it is very difficult to use. Instead, more abstracted, human-readable languages like C++ are used.

Accomplishing this involves several stages: what is known as an edit-compile-run cycle. First, computer code is written by the programmer. This is usually in a text-editor, which is very similar to a word processor but without the unnecessary features needed for desktop publishing. While it is legible to humans, a computer cannot do much with the code at this stage. For the machine to read it as instructions, it must be sent to a compiler, which translates it into executable (machine) code. At this point the compiled code is essentially the program, ready to be run. The process occurs as a cycle because any new changes made to the code have to be recompiled to see the results. This iterative part of the cycle, devoted to error correction, is referred to as “debugging”.

The process of working with programming is different than most architects are accustomed, but is not altogether unfamiliar. The closest analogy might be computer rendering, which is itself computationally intensive. An architect might spend a long time composing the scene, but without rendering
the image is not production-level: lights, materials, and post effects are not possible in the viewport, nor is the resolution high enough to print. The scene data must be sent to the renderer to be computed into a final output. Rendering is also an iterative process, like debugging code. Any changes in settings must be re-rendered to produce a result. In summary, coding is not as interactive as drawing, but like a finished computer rendering, the results can be incredible and take much less time than doing the same task by hand.

Finally, it should be noted that all computer languages, even binary, are readable by humans (Touretzky 2000). This makes sense, as they were designed by humans. Truthfully, there is little reason to read something like binary today, but other languages are quite readable. Although it is called 'code' and it may look like gibberish meant only for machines, it is a language all the same. And as a language, it is a capable medium for expressing and communicating intent. This is important to keep in mind throughout the thesis. Learning a computer language is difficult, like learning a language from another country, but it is not impossible—it just takes discipline.

2.2 User Interfaces

The notion of the user interface is significant because it expresses the intention that computing not be limited to specialists. In defining this relationship, a separation is created between those who program or otherwise work intimately with the computer, and those who merely operate one. Most people consider themselves to be users only, and do not want to concern themselves with minutia of how the system works. Why they make this decision, they relinquish some of their control and potentially limit themselves, but receive an easier experience. If computation is to succeed, this dimension of the user has to change. We can begin by studying the specific compromises and compensations imbedded within our decision to use a particular interface.
Those who design interfaces must weigh these choices all of the time. To describe this aspect of their work, they refer to the concept of ‘usability’. Simplified, this is the degree to which something is both “easy to use” and “useful”. There are many dimensions to usability, including cognitive, psychological, and physical factors. Like any design, a well-made user interface is tailored to its audience. How well an interface responds to their needs, its efficiency, and learning curve, contribute to its overall usability. Thus, success is determined by finding the proper balance of simplicity and function.

This is a short definition for a deep topic, but the overall idea will be useful to keep in mind over the next few chapters. With the intention of finding new answers, first we must understand why user interfaces developed, how they compare to one another, and what was lost in the transition away from programming.

2.2.1 Command Line Interface

The command line interface is a textual method of interacting with a computer, in contrast to the graphical user interface. With the command line, a user types in commands at what is known as a “prompt” and the computer executes them. These commands represent simple actions like copying a file or starting a new program. Because of its comparative simplicity and faster response rate, it is an improvement in usability over earlier interfaces like batch processing or programming. On the timeline, they entered into use after these, but before graphical user interfaces.

Classifying the command line interface is tricky. It is similar to a domain specific language in that it improves usability by relying on a limited set of abstractions. Although it makes use of syntax, it is not a programming language because it does not have the necessary operators (conditionals, for example). However, small programs called “batch files” can be written
by making lists of interface commands and saving them as a new command.

For architects, the command line interface in AutoCAD is a familiar example that illustrates one of the advantages text has over graphical manipulation. Graphical user interfaces (GUIs) are typically easier to learn, but once mastered, a command line interface is more efficient. Instead of taking time to visually acquire and perform several manipulations between the tool and its target, a skilled user can touch-type the command without looking. Physically, it is also possible to combine actions: one hand on the keyboard invoking tools and one on the mouse selecting objects. Experienced AutoCAD users rarely use icons to select tools because typing in commands is much quicker. Indeed, the most used commands are abbreviated to a single character. Once a command is learned, it is no more or less difficult to invoke than any other command, unlike icons which can become disorganized or hidden.

The other primary advantage of working in the command line is its lack of abstractions in comparison to graphical interfaces. While it is true that commands themselves are abstractions, the way they are applied is more explicit than manipulating icons. This is advantageous for many reasons. Typing in a specific file name might be tedious, but it might also be faster than having to find the target file or directory through menus. In addition, by requiring specification, the user is more aware of the operations being performed. Unlike GUIs, where the same icon might represent different functions for various reasons, commands are always well-defined by the system.

Finally, invoking commands is consistent. GUIs have changing menus and options depending on the circumstances, but the command line always works the same way. There is no interface to "get out" of—one can drop from the middle of a command back to a fresh prompt at will.

While they may appear antiquated to the average person, the strengths of command line interfaces have made them the
method of choice for more experienced users. This is confirmed by the fact that modern GUIs have “shells” or alternate interfaces that make use of the command line. Ironically, the reverse used to be true. The first commercial GUIs were basically shells or layers running as a separate program on top of command line interfaces. Clicking an icon essentially typed in the command the icon represented into the computer. While this remains conceptually true today, there is now much tighter integration between the GUI and the functions of the computer. Today the command line is the shell and not all functions of the GUI can be accessed at this level. Regardless, as long as typing is an available option for users, this kind of interface is likely to remain.

2.2.2 Graphical User Interfaces

The graphical user interface (GUI) is arguably the most commonly recognized aspect of the computer. Graphical interfaces are often more intuitive than text interfaces because they work by direct manipulation of images instead of typing and rely on metaphors instead of specific commands. For many, the experience of using one seems less intimidating than a command line interface and is easier to learn, as well. Because of this, they have had an enormous democratizing influence. Together with the personal computer, they are responsible for bringing computing within the grasp of the average person, ushering in the ‘information age’ of today.

While the software aspect of GUIs often receives the most attention, it is actually a combination of technologies working together that makes the experience possible. Display improvements and new input devices like the mouse and light pen developed alongside the programs that made use of them. Although widespread use took off in the early eighties, the modern GUI is actually over 40 years old. What is more, architecture played a major role in its early development. A pioneering paper that led to several innovations cited the need to develop computer interfaces specifically for architects (Engelbart 1962). Ivan Sutherland developed one of the
earliest GUIs as part of system called Sketchpad. This MIT project was a precursor to modern CAD systems. In the future, new kinds of hardware will be necessary in order to enable different ways of interacting with the computer, but, for now I will focus on software improvements.

At the heart of the success of the GUI is the innovation of introducing direct manipulation and a constant field of imagery to the computer. The cursor provides a focus for our visions and our actions—uniting our hand with the screen. Pointing is intuitive and moving things gives us immediate, constant feedback, something which no other interface can replicate. It makes visualizing our actions less difficult, allowing us to construct a better mental model of what we want to do. The graphical environment helps with this, as well, augmenting text with layers of symbolic representation. In addition, icons and other interface "widgets" allow for different kinds of interaction other than text manipulation—often several at once. The importance of reconciling vision with intent within the computer cannot be underestimated. Going beyond textual interaction dramatically changed people’s vision of the computer. For the first time, it could not only simulate machines, but it could almost become them by taking on their appearance and interface. In doing so, computers began to seem less foreign and more familiar.

Operating systems that have GUIs rely on a system of metaphors that resembles an office situation. For example, directories are folders, files resemble sheets of paper, and the desktop is the primary work space. GUIs that use metaphors are often intuitive because they relate things we are familiar with to similar computer operations. However, incorrect, inconstant, or ambiguous metaphors can actually be harmful to the user experience. One famous example is the Macintosh trashcan. In the Mac GUI, files are deleted by moving them over the trash icon; this procedure suggests “throwing away the file”. But the same procedure is also used to eject a disk—which make users very uncomfortable. Logically, one would

---

6 Software is more practical to develop in the short term and the end product is easier to spread to others.
think that the contents of the disk would be deleted if it was thrown away. The point is clear: metaphors must be consistent. They can work if their meaning is shared by the audience, but this cannot always be depended upon. The metaphors GUIs use come from the developers, not from the user, and unlike real metaphors, they are not open to interpretation. Most importantly, these metaphors are only labels for what lies beneath. While they leverage what is familiar, they prevent us from understanding what is really happening and potentially learning something new.

Performance is another issue of GUIs. During the evolution of interfaces, the speed of processors and memory, as well as storage capacities, had to increase before there were sufficient computational resources to support the load of a GUI on the system. This was necessary to sustain simultaneous operations between the environment and the programs running on it. Indeed, the GUI itself continues to be a draw on system resources as their complexity grows with each new version. An older system that ran Windows 98 cannot run a new copy of Windows XP. In contrast, the default installation of Linux has no GUI, and will run on much older systems. With this setup many applications run faster than their Windows counterparts. The lesson to be taken is that improving interfaces has traditionally been computationally expensive. Researchers might want to consider whether their advancements will force users to upgrade (and whether they can afford to), or if there are other ways to accomplish the same goal.

To summarize, GUIs are a major advancement in interfaces. They can make our lives easier, especially for common operations, but they do not solve all our problems. As we will see later, programming is particularly difficult to adapt to graphical abstractions.

### 2.3 Programming Interfaces

Superficially, programming shares many of the same difficulties of pre-GUI operating systems: arcane commands, a
text-only interface, and a lack of visualization. Since the development of the first user interfaces, there have been many attempts to apply the same logic towards making programming more accessible.

The three types discussed here are representative of the most common strategies for improving the usability of coding. Overall, programming interfaces have shown they can assist users in learning and writing code, but like any interface, each strategy has its limitations—a single interface may not be enough.

2.3.1 Integrated Development Environments

The most common programming interface is the integrated development environment or “IDE”. In contrast to running separate command-line programs for each stage of the edit-compile-run cycle, an IDE consolidates the development process into a single interface that may or may not be graphical. Although a few of them can work with multiple languages, most are specialized to a single language. IDEs are usually written as an interface layer on top of a preexisting language. However, with proprietary programming methods (such as the graphical languages described later), they may be incorporated into the language itself.

Graphical IDE’s are a less intimidating introduction to programming and can reduce the time it takes to learn a language. Some programmers eschew them as a fancy crutch, but for novice users they have many valuable features. A typical IDE consists of a text editor, compiler, and a debugger (special software for isolating and correcting errors). Graphical environments replace the most frequent commands with icons and make use of windows to show different representations of code structure, class libraries, and other helpful aids. Additionally, IDEs often have tools to verify proper syntax, visualize syntax with colored text, and even automate coding in some cases. Embedded help functions provide a reference for commands and syntax. Together, these tools have made
programming less esoteric and more like working with other software, such as word processors.

2.3.2 Domain-Specific Languages

For I am a bear of very little brain and long words confuse me
- Winnie the Pooh

Domain-specific languages (DSLs), are high-level programming languages developed to fulfill a very specific niche. Because they are often less complex compared to general-purpose languages, they are also known as “little” languages. DSLs are useful because they can simplify writing code for certain kinds of tasks and are relatively quick to compose. They are developed by expressing commonly used methods from the application domain as a collection of higher-level abstractions. This creates dramatically less code than a general-purpose language would for the same task. For the user, commands are more familiar and there are fewer to remember. A well written DSL should not take long for someone from the intended audience to learn because concepts and vocabulary of the language should already be familiar.

There are caveats to this approach, however. Given the right language for a domain, it becomes easier to solve problems. But one must be careful when using or designing specialized languages, for the opposite is also true. If poor choices are made, one may not see the solution because the language prevents describing it.

Today, scripting is the most common method of programming used by architects. Scripting languages are a special instance of a DSL implemented as part of a software application. Scripting allows users access to a program’s commands and data structures in conjunction with basic computational operations. In their simplest form, scripts can be little more than a list of commands joined as a single operation. These are referred to as batches or macros. More complicated scripts can actually extend the capabilities of a program, creating entirely
new tools with their own interfaces. Unfortunately, every scripting language is so specific it cannot be used outside of its native program. This is the primary weakness of writing scripts. Other users must have the same program, and often the same version, to use someone else’s script. In the end, only so much can be done with scripting. Command structures and performance issues prevent alterations that would violate the scope of the original program. Nevertheless, scripting remains a powerful way to quickly customize software.

2.3.3 Graphical Programming Languages

Graphical programming languages (or visual programming), as their name infers, use visual expressions to represent the process of programming. Essentially a hybridization of a domain-specific language with a graphical user interface, they can help reduce the threshold for making useful programs. The basic idea behind them is to replace strict typing with symbols that can be manipulated with the mouse. By doing away with typing, graphical languages alleviate overly strict syntax errors like missing punctuation that often discourage novices. In practice, language commands are represented as modules, some containing text forms for strings, counters, or other sub-level interfaces. These modules often have multiple layers of identification, such as labels, shapes, and colors, making them easy to recognize. Programming with graphical languages involves instantiating copies of modules from a common library and making linkages between them. The user can arrange them into working programs by forming structures describing relationships like data flow or object orientation. This makes it useful as a teaching tool, visualizing concepts that can be difficult to explain using traditional languages.

As an introduction to programming, graphical languages might be beneficial for visually-oriented people like designers, but in practice they have limited application. They work best for shorter, more focused tasks; developing an entire drawing program would be a frustrating enterprise. Once a certain level of complexity is reached, graphical languages become
unwieldy. Organization becomes difficult as the screen starts to fill up: icons are harder to distinguish and linkages are obscured. This can happen quickly; graphical programming is seldom elegant. To keep them manageable and easy to learn, commands are intentionally kept few and are therefore general. Because of this, simple statements can sometimes require a considerable number of modules to code and might be better expressed through text. If more specialized commands were available, differentiating one from another would become a problem. Eventually finding the right “piece” ends up being more difficult than typing. These shortcomings are indicative of the fundamental challenge facing graphical languages: they are all based on textual programming which appears resistant to visual metaphors. To be successful, perhaps programming itself must be rethought.

For those who are new to programming, or who do not need to program very much in their work, graphical programming languages are an option. Designers may be among this group, and useful non-classical computational metaphors such as shape-grammars might be able to be expressed in this manner, but there are issues with this approach, as well. Visual programming remains the “holy grail” of programming usability. A powerful implementation may someday come to pass, but it is a problem that is likely to challenge computer scientists, and architects, for some time.

Some notable examples of visual programming languages are LegoLogo (described in the next chapter), the 3D shader language, RTShader, and the music performance language, MAX.
CHAPTER 3: RELATED WORK

"The future is [here] now; it's just not evenly distributed."
-William Gibson

Although great leaps forward are known to happen, in today's world technology moves so quickly that developments are more likely to be incremental or combinatorial in nature. Sometimes, the solution already exists in another field or is scattered among many sources. For this reason, when looking for innovation, it can be best to cast a wide net. This section gathers together past, present, and future efforts to teach and apply computation in an attempt to learn from their mistakes and extract their best qualities.

3.1 Pedagogical Programming

Surprisingly, the idea of teaching programming has a long history, dating back to before personal computers or sophisticated graphics. At the time, computers were rare. The thought that they could be used by the average person, much less a child, was radical. To meet this need, new pedagogical languages and frameworks were created with features to reduce complexity and hold peoples' imagination. While some of the languages in this section are intended for younger audiences, for the computational researcher interested in spreading the ideas of computation, there is much to learn from them. With their emphasis on visualization and simplicity, they have attributes that might translate well to a teaching language for architects. In fact, the last segment discusses something along this line: a programming language for designers that came from a language once meant for children.
3.1.1 *Logo*

Logo is a language and environment developed in part by Seymore Paupert, a mathematician and a close follower of educational researcher Jean Piaget. Paupert wrote a famous book about Logo, *Mindstorms*, that discusses how programming can help children to learn problem solving in subjects like math. Because of this book, Logo is best known as a pedagogical language, but it was originally a simplified version of LISP, with all the functions necessary for computer science applications.

One of its innovations made for teaching was the Logo environment’s “turtle”, a graphical representation that follows the user’s instructions. It functions as a powerful abstraction of a pen that possesses characteristics of direction and heading. This makes it much easier to visualize processes (to “think” like the turtle) and eliminates the need for the user to program their own abstractions for inscribing figures. Logo itself is a simple language. Some have said it is LISP but without the parenthesis, but this does not accurately reflect all that has been done to make it more intuitive for the novice. First, commands in logo are not as ambiguous compared to other languages. Most of them need little explaining and their names make sense. For example, to move the turtle, the commands are: FORWARD, BACK, LEFT, RIGHT, TURN, etc. along with simple numbers for angles and distances. What is more, when commands are entered, they can give immediate feedback, so the user does not have to think too far ahead of the program (at first). As fluency develops, more complicated non-linear problems can be tackled. For all these reasons, Logo can actually be fun to use; this might be its most important feature of all.

A later innovation was the tangible turtle interface, a device that emulated the screen turtle’s actions by moving around on wheels and drawing with an attached pen. A Logo program for a geometric figure could be downloaded into the “real” turtle and then physically drawn onto a piece of paper. Intriguingly, the opposite was also possible. A user could move the turtle
around to draw something and then download the code for the action into the computer. This is known as “programming by example” and together with the tangible expression of the code, they form a powerful combination for reinforcing the concepts of procedural thinking.

Logo retains the ability to use variables, data structures, and conditional operations, so recursive and iterative procedures are possible. New commands can also be defined in the language. From these simple instructions, complicated forms can arise. For example, there are even math textbooks about “turtle geometry” that teach about advanced subjects like fractals using Logo.

There are more than a hundred variants of Logo still in use today, some of them quite different from the implementation described here. Nevertheless, the language helped validate the idea that children can learn programming as well as learn from it, leading the way for other simplified pedagogical languages.

3.1.2 Design By Numbers

*If you want to teach somebody something well, you have to start at the very lowest level. It's like Karate Kid. Wax On, Wax Off. Wax On, Wax Off. Do that for three weeks.*

*Then knocking The Other Kid's head off is easy. –Joel Spolsky*

One of the criticisms most often heard regarding computer artists is a lack of discipline. This is understandable. Designers want results fast, and our tools are built with this in mind. Under these conditions, how does one develop rigor?

John Maeda’s Design by Numbers (DBN) is a pedagogical language intended for those with little or no programming experience. Though the software is written in Java, the language itself is based on Logo, borrowing its simple syntax and ‘pen-up’, ‘pen-down’ model. It is used to create visual expressions, but limited to one-hundred and one shades of gray
Nicholas Senske  
_Fear of Code_

within a 101 x 101 pixel display area\(^7\). In addition to drawing, the language also facilitates interaction and animation. Because it is similar to Logo, DBN code tends to be compact and easy to read. Compared to other languages, it is much easier to understand what a DBN program does just by looking at the code. This makes it simple to share and learn from other’s work.

DBN is a limited language, but this is intentional; its constraints are also its strengths. For example, Maeda’s decision to restrict the color palette to grayscale is often questioned by new users. Those new to computer graphics are not likely to understand, but manipulating color requires separate values for red, green, and blue components—at least three times as much information to control. By keeping all numbers in DBN below one-hundred, users can more easily visualize percentages, and thus feel a greater sense of control\(^8\).

The programming environment is only half of the picture. Equally important (if not more) is the beautifully designed book that accompanies the software. _Design By Numbers_ is as much a work of art as it is a manual—probably the most non-threatening tutorial ever produced. Maeda uses liberal amounts of whitespace to buffer code, illustrations, and text on the page, so the lessons never appear overpowering. His text descriptions are concise, yet colloquial. In a progression reminiscent of Paul Klee’s _Pedagogical Sketchbooks_, each concept has its own chapter, building from basic primitives to complex constructions. The experience of working through the lessons along with the commentary is truly enlightening; there is as much to learn here about design as there is about the basics of programming. Most tutorials are designed to make the user productive as soon as possible. Maeda’s book is not only demonstrative, but it cultivates an appreciation of the

\(^7\) This is a lesson in itself: \(0 + 100 = 101\). In computation, counting begins with zero (0). Rather than stop at 99, Maeda decided to finish with a round number, 100.

\(^8\) One hundred is also a nice, round number. As Maeda writes, most people can count to a hundred and can imagine a hundred of something. Another consideration: math always seems easier to do with smaller numbers.
processes computers use to display graphics and the thinking it takes to create them. This is essential if one is to move beyond mere rote knowledge. Future tutorials written for architects should strive for this level of comprehension.

DBN succeeds because it does not try to take on too much. While some interesting designs can be created, realistically, there are better tools out there. DBN is a pedagogical language, not a production language. This idea took me quite a while to get my head around, as these days, my time is quite limited and it seems as though everything I learn must have an immediate application. Now I understand that the simplicity and constraints of the language allow the user to focus on learning concepts. Putting aside fancy tools prevents distractions that might hinder the development of rigor. The experience of DBN is meant to be temporary; one cannot comfortably stay there for long. Once the possibilities of programming are understood, the small window begins to feel less cozy and more confining. The user soon realizes there can be no more growth without moving on. Rather than feel full and overwhelmed, constraints force us to take tiny bites of information—and leave us hungry for more.

### 3.1.2 LegoBlocks

LogoBlocks is a graphical programming language that uses the Lego brick metaphor to allow the user to program special microcontrollers. These ‘programmable bricks’ have peripherals such as motors, lights, speakers, and sensors that attach to regular Legos, allowing children to make their own interactive creations. Special software simplifies the process of writing and downloading code into the programmable brick. The combination of modular mechanical parts, Lego blocks,

---

9 This reminds me of one of Matthew Barney's earlier works, *Drawing Restraint*. In the description of the piece, he cites the Marquis de Sade, who wrote: "The best way of enlarging and multiplying ones desires is to limit them."

10 My first semester at MIT, I took a course called "Tools for Learning" instructed by Mitchell Resnick. The class included hands-on experience with some of the projects from Resnick's Lifelong Kindergarten group, including LegoBlocks and Scratch.
and programmability makes the possibilities almost unlimited. LegoBlocks has proven highly successful both for teaching children and as a commercially available product.

Commands in the language, each represented by different colors and shapes, have a strict hierarchy. Like puzzle pieces, "bricks" of code will only connect where these joints fit together, ensuring that the program is always syntactically correct. This significantly reduces the potential for errors and is very helpful to the novice. Learning is intuitive: even without any prior knowledge, one can experiment simply by stacking pieces that fit together. Because of the rigid structure and the finite space available in the interface, the possible combinations are somewhat limited.

In spite of this, it is surprising how much can be accomplished with only a few commands. One particularly interesting student design shown to our class used two motors attached to light sensors to make a car that would seek out light sources. If one side of the car could not "see" light, it would turn in the opposite direction until it did. This was accomplished with simple logic statements set to loop continuously, and, when finished, took up only twenty LegoLOGO pieces.¹¹

When my own class sat down to use the system, we had both programmers and non programmers with us. Less experienced students quickly learned the system and were quite pleased with how much they were able to accomplish. At the same time, the students who were programmers appeared frustrated by the interface. By the end of the class, their projects were no more advanced that the novice programmers'. This is yet another example of the effectiveness of limited systems for learning. At first, the level of accomplishment is rapid, but it soon levels off.¹²

¹¹ That may sound like a lot, but if one thinks of each piece as a word, it would only be two average sentences or less than ten lines of code.
¹² This is also a criticism heard of graphical user interfaces. I was said of the old Macintosh: "It takes only a half hour to learn, and in six months you will have learned nothing more about it."
The idea of a visual programming language will always be compelling, but for now they may be too restrictive. Tellingly, it did not take long before LegoBlocks users wrote several of their own compilers based on general-purpose programming languages. The demand was so great that Lego eventually started to sell a C++ development kit. This example illustrates the shortcomings of relying upon a single language—and the proper response. When the language prevented the LegoBlocks users from doing what they wanted, they found (or made) one that did.

3.2 Architectural Computation

This section is intended to provide a measure of the level of progress attained in the implementation of architectural computation today. While acknowledging the ongoing contributions of researchers to the field, in this section, for the sake of practicality, I focus on work that is already accessible to architects. My selections are by no means comprehensive, but I believe they are representative of the types of resources presently available to students and practitioners. The goal is to establish the state of computation in practice, specifically the tools and educational materials available, as a point of reference for discussing improvements later in this thesis.

3.2.1 AutoLISP, RhinoScript

While there are many scripting languages in use, these two were selected because they are currently being taught at MIT's Department of Architecture. AutoLISP is the scripting

---

13 A few caveats are in order. Computation is a fast-moving field; I made the choice to include some prototypes (as of this writing) in this assessment, for reasons which I feel are justified. Generative Components (3.2.2) is still under development, but it is expected to be released soon. I believe that as a software product with unique programming characteristics, it is the first of a new generation of computational tools. Also, because its focus is primarily towards architecture/engineering, it is better suited to the tone of this section than the next. Likewise, the DesignTooling (3.2.3) site is still under construction, but it is already a publicly available resource with significant value.
language for Autocad and has been used for many years by architecture offices to create automated routines and customizations, among other things. RhinoScript works with Rhino and currently finds application in automating tasks such as fabrication preparation and surface-making. Contrasting the two, the most obvious difference is that AutoLISP is an older language from a much older product, whereas RhinoScript is more recent. This is reflected in the selection of the language each is based upon, their programming interfaces, and their extensibility.

Scripting is powerful because it makes use of a program’s existing commands\textsuperscript{14}. Autocad and Rhino are good candidates for learning scripting because they both employ what is known as a ‘parser’ or command line interface\textsuperscript{15} that mirrors the commands represented in menus and icons. These redundant interfaces can be helpful in learning the program because the same command can be invoked in many different ways. Efficiency is not the only benefit of using the command line: by becoming familiar with the parser commands and their variables, the user is actually learning to script at the same time.

Because it relies on a program’s command structure, the scope of scripting can be quite limited. For the advanced programmer, lower level compilers are available for adding functions that fall outside the bounds of the original software, but this is an extreme option given the performance issues and the effort required.

AutoLISP and RhinoScript are solid introductions to programming for architects. The interoperability between the languages and the user interface makes them a good choice, especially for experienced users who already know the commands. An added benefit is that because they are closely

\textsuperscript{14} It should be noted that not all commands, especially newer features, are available for scripting. This is a common problem, related to the fact that scripting is not a high priority for most architects, and thus, the same is true for developers.

\textsuperscript{15} There are even scripts that can change Rhino’s parser to use the more familiar AutoCad commands!
based on existing languages, learning to script also teaches the user about LISP or VBA. Additionally, books and online resources for these well-known languages are widely available, which can help in the process.

While other programs would seem just as good as any for learning to script, this may not be true. Programs like Maya and 3dsMax use their own proprietary scripting languages—learning them does not pass as easily to another language and educational materials are scarce. Also, both of these programs have extremely complex interfaces, so parsing, although available, is inefficient and therefore, an uncommon usage pattern. Nevertheless, scripting is currently one of the best ways for architects to begin learning and using computational techniques in their work.

3.2.2 Generative Components

Generative Components (GC) is a tool under development for architects that combines multiple levels of visualization and manipulation to bring computational design to a wider audience. As such, it offers a bridge between coding and modeling that has both pedagogical and production potential. The general usage pattern of GC is for users to associate geometry with constraints or functions defined within the scene. By combining direct manipulation with computational tools, users are able to generate forms that were previously difficult to do with programming or modelers alone. The multidimensional setup between the geometry, code, and alternate visualization is a powerful tool for situational awareness and for learning by association. For example, formulae can be assigned through a scripting language or from predefined sources. These linkages are visible in a window as a graph representing the scene. Through this diagram, users can get a sense of the relationships between objects and manipulate the linkages to make changes. It is not meant as a production modeler, so many of the tools designers are familiar with for making forms are not available. The philosophy is to use simple components and apply
computation to make them into more complex constructions. In addition to scripting, there are tools to simply these processes. One such tool is a function to propagate a component into an array. The ease of combining pre-existing tools like this one and user-defined functions makes GC a capable and extensible platform for both production and exploration.

Based on interviews with students from MIT who participated in workshops or studios with the software, the learning curve is similar to other CAD software. Within a few hours, novices were able to make some interesting forms that would be difficult with other tools. However, based on anecdotal evidence, students with programming skill had a tremendous advantage. This was not only because they could program more complex functions using the scripting languages, but also their conceptual knowledge was better trained. Finally, the most successful user group also had previous experience with parametric modeling tools like CATIA. Thus, they could pre-visualize their intentions within the model of the software environment. This raises some interesting questions. The user group at MIT that performed best with GC was students with knowledge of both programming and parametric software. This is a very rare segment of the general population. If this is the kind of knowledge required to make best use of the tool, it might not be a candidate for mass-adoption. It would be revealing to do a study of how well students can learn these skills from within GC and to speculate on what kind of educational model would be most supportive.

While a step in the right direction, GC can only do so much. Geometric modeling is a rich area for architects, but it, too, is limiting. One of the powers of programming in more general languages is the ability to incorporate outside sources into new compositions. Currently, GC does not have the ability to import something like a sound file for use in a scripted

---

16 With the academic environment at MIT, there is a concentration of these students that skews an objective investigation of the software. Even an average student at MIT is likely to have more exposure to these kinds of concepts than students at other schools.
constraint. This is one possibility that would make it even more interesting to use, but it is only one example among many scenarios not available in the software. Inevitably, the addition of an interface sacrifices generality. However, the innovations GC brings to CAD tools have lowered the learning curve for some forms of computational expression that was previously out of reach to most architects, and for this it should be applauded. GC remains under development, so it may still change in the future. Because only a few test groups have been able to spend significant time with the software, it is difficult to say how popular it might be or the impact it could have on practice.

3.2.3 Design Tooling

The Internet is a tremendous resource for those looking to teach themselves. With the speed of software development and design culture far outpacing the cycle of print media, websites can be a vital tool for learning the latest techniques. In addition to the currency of the information, the ability to use interactive examples and download files enhances the experience.

With this in mind, the Design Tooling website (designtooling.mit.edu) is a self-described “repository of knowledge about computation for designers” initiated by students and faculty at MIT’s Department of Architecture. Divided into five main categories, each contains a summary, references, and a mixture of sample code, tutorials, and examples. The information provided by Design Tooling was previously all but impossible to find together in one place. Rather, one would be much more likely to find a single project or page focused on a particular area, with little regard to how it relates to the rest of the discipline. Along with reference sections, the site also hosts a community forum for the discussion of improvements and general computation questions.

17 Bidirectional Design, Metric Design, Evolutionary Design, Design Machines, and Sketching by Computation
There are many publications or websites that showcase computational work, but very few of them will reveal the specifics of how it was accomplished. For this reason, the “Sketching by Computation” section, which is about algorithmic form generation, is by far the best of the site. Its descriptions are deep, yet succinct, with multiple examples of projects given. The RhinoScript/VBA tutorial is sufficient to get new users started (although it is very text-heavy; some images would be helpful) and the geometry libraries provided are most useful. Several notable books on computer programming and architecture have been written in the past, but none of them have been updated since computers have matured and interest has grown—a current source that is not afraid to be technical is a welcome addition. Overall, the comprehensiveness of this section represents the promise of the site to educate, but, unfortunately, this level does not hold up throughout.

Today, it is rare to see an attempt to present the discipline of computation at once, and the site should be applauded for this. However, the primary shortcoming of the Design Tooling site is its lack of coherence. I imagine that as a first introduction to computation, the experience of the site would be overwhelming. There does not seem to be a sense of priority or progression. A new user would probably ask: “What is all this?” “What is important?” “Where does one begin?” Based on the kind of information available, it would be difficult for someone who was not already invested in this area to apply it to their work. Some kind of introductory module to accompany the others would be ideal. Also, the level of content for some sections is limited. There are often not enough interactive examples or sources of code to make use of what is being explained. Consistency is also lacking in the samples that are provided; there are too many platforms represented. I would recommend that all the tutorials and examples share a common programming language. Consolidating this would make learning easier for the novice and hopefully reduce the work

\[18\] Processing would seem like a good candidate for this.
for the site's designers. Finally, the presence of a forum is laudable, as it suggests the intention to start a community around the site. Design Tooling should extend this idea to collecting information, projects, and code to fill in the gaps left in some of the areas. Overall, I believe the website represents the kind of resource computation needs: something that is freely available, current, and always developing.

On a final note, my criticisms should be taken with a grain of salt—Design Tooling is a work in progress. The same concerns are shared by the development team and are to be addressed in the near future. In the meantime, the site stands as a great resource and an example of the external identification and transparency our discipline needs.

3.3 Current and Future Trends

A study of my topic would be incomplete without mentioning the current research in the field. These three projects were selected because of the diversity of their approaches and their intended audience. The first two are evolutionary refinements to existing practices, while the last is an attempt to rethink the way we use computers for design. All of them show promise towards fulfilling the objectives of this thesis. Regrettably, in spite of their innovations, there are still many challenges left unanswered. Nevertheless, I feel that these threads represent positive trends towards the pursuit of computational design in the future.

3.3.1 FormWriter

Pedagogical languages have proven successful at teaching computational concepts, but for the most part, they are intended for younger users. Something as simple as LOGO might seem inappropriate to a college student. Design by Numbers is better at teaching design concepts in the context of programming, but its output is severely restricted. Students have to switch to another language if they want to produce
data for their projects, which takes up even more of their limited time. With the demands of architectural education, this is not a favorable situation. Scripting is another option. However, it may be too complex of an environment for an introduction to basic concepts. FormWriter, a 3D geometry language under development by Mark Gross and his Computational Design group at Carnegie Mellon, could be a happy medium for the needs of architects (Gross 2001).

FormWriter\textsuperscript{19} is intended for algorithmic form generation and shares many similarities with LOGO. By writing instructions for a 3D “flying turtle”, users can describe geometry within a navigable viewing window. As in LOGO, this abstraction is a more intuitive way of thinking for novices because it allows the user to “draw” rather than specify geometry directly from points or complex equations, which is the case with other languages. For architects, working with volumes in three dimensions is much more familiar and useful than being restricted to drawing on a flat surface. FormWriter also fits into the demands of the studio by offering production capabilities. Users can export their models to .DXF files so they can be 3D printed or sent to other modeling applications.

FormWriter is designed to be easier than scripting or general-purpose languages like C++. The language itself is very straightforward; each line of code produces immediate feedback, so it quickly becomes intuitive. By introducing concepts like recursion and iteration, these simple methods can produce complex forms that are too difficult to model with CAD programs. Parameterized and conditional constructions are also supported by the software. From only a little code, a surprising amount of depth is possible. With these features, FormWriter could fulfill the current need for a pedagogical tool that is both useful and comparatively easy to learn. Unfortunately, development has not progressed for some time

\textsuperscript{19}FormWriter does not currently have a public release, although a limited Java implementation exists online. The latest in-house version only works with Mac OS 9, but other versions for Windows and Linux are planned in the future.
and there is little documentation available for anything more than the most basic functions. Gross has stated he plans to continue working on the tool in the future.

FormWriter distinguishes itself by focusing on the particular demands of architects. It does not feel intimidating, nor is it condescendingly simple. In the future, with access to more computing platforms and better educational materials, it could prove to be a popular pedagogical tool.

3.1.3 Processing

Processing is a ‘programming language for visually oriented people’ developed by MIT Aesthetics and Computation Group alumni, Ben Fry and Casey Reas. Pedagogically, it is the conceptual offspring of Design by Numbers, but the language does not share the artificial constraints of its predecessor. Indeed, while it has features that make it suitable for teaching, it is production-ready. In addition, Processing is not a fixed language, but a platform for future growth and development. It already boasts some impressive capabilities for designers, such as interface libraries, 3D, particle systems, file export, sounds, and much more. Rather than restrict output to a proprietary environment or format, Processing creates Java ‘applets’: small applications that can run in web browsers or by themselves. This allows users to share their work on the web, regardless of the viewer’s operating system.

To assist with learning the language, Processing has several different “modes” of coding for different levels of users. The basic mode is similar to a command line interface and is intended to introduce primitives and commands. The more advanced modes use a syntax which is based on the one used by Java, but is much simpler. The advantage of this model is that users can graduate from one level to the next as their skills improve, and never have to leave the same environment.
Processing is committed to open source development, which means that users are free to download and modify its source code as they see fit. The benefit of this arrangement is that users can contribute new functionality to the language and help to fix bugs themselves. While the majority of its development continues to occur through Fry and Reas, users have already made several additions to Processing that extend its capabilities. One of these, called Wiring, is a series of libraries that allow the language to interface with hardware. Other additions include database plugins and sound processing utilities.

Another significant aspect of the open source philosophy is the community that it encourages. Designers use the Processing website to share their work, help each other, and discuss the state of the language project. Because of this discourse, the user base has grown steadily and some aspects of the work, like documentation, have almost become self-sustaining.

One of the current disadvantages of Processing is the unfinished and somewhat disorganized state of its tutorials. The primary website (processing.org) has two basic tutorial sections, a progressive, concepts-oriented tutorial and more feature specific tutorials. Unfortunately, the material in the concepts tutorials is not engaging and several sections remain unfinished at this time. The feature tutorials are well organized, but it is difficult to get a sense of what each one actually does because of the way they are organized. There are only text descriptions for the features and they do not seem to be in any particular progression, which only makes it more difficult to decide how to proceed. There is a sort of manual available, but it seems as if it is intended for users who already know some programming—much of it is devoted to how computational procedures and concepts from other languages are carried out in the environment.

It is difficult to determine yet whether Processing is a good first language for designers. Many design programs all over the world already have classes that use it to each computation, but there is no hard evidence to suggest that it is significantly
better at this than Java or even Flash. A forthcoming text might provide some much needed introduction to basic programming concepts as a bridge to the more advanced work seen on the website. Once this happens, perhaps Processing will be better equipped to address the full spectrum of user types.

3.3.3 OpenAtelier/ Treehouse Studio

In one of the first discussions I had with John Maeda while preparing this thesis, he told me about his latest project, OpenAtelier. Earlier in his work he had focused on the possibilities of programming for designers, so I was very surprised when he told me he was not convinced of the potential for computation anymore20. “Computers aren’t the answer,” he said. “It’s people. We should be finding new ways to put more people together on the same problems.” This shift in mission is reflected in the name change of Maeda’s Media Lab research group from the Aesthetics and Computation Group to the Physical Language Workshop21. One of their main projects, Treehouse Studio, is a collaborative community for introducing people to the digital arts. To this end, the group is currently developing its own network architecture and a suite of freely available web-based tools, encompassing a wide range of media such as drawing, video, three-dimensional modeling, and even some programming. Treehouse Studio is a broad endeavor and although it does not immediately fit into the theme of teaching computation, I feel that as an educational and production model some of its ideas could be applied in this direction.

---

20 This is a simplified account of the conversation we had. John sees algorithms, indeed, all current tools, as limiting. He does not believe we are seeing anything original coming from computation. I think John has a point, and his new approach is a step in the right direction. However, I believe that architecture, which has yet to fully embrace computation, hasn’t even begun to feel out the possibilities for itself. Assuming there is only so much that can be done aesthetically with algorithms, issues of cost and functionally still remain. In architecture, there will always be room for more efficient solutions.

21 This is a reference to one of the founding Media Lab groups, Muriel Cooper’s pioneering Visible Language Workshop.
John Graham has written extensively on the benefits of web-based applications in his book *Hackers and Painters*. He speaks from experience (and possibly with some bias) as one of the first developers to write a web-based e-commerce storefront, still used by Yahoo to this day. The advantage of centralized tools is that they are always available to the user, regardless of their system. Provided a connection is available, accessing the tools should be as simple as loading a website. For the developer, updates are taken care of on the server-side. This makes it easier and faster to do things like fixing bugs and adding new features. Changes in the program take affect as soon as the user connects to the server, eliminating the need for manual client downloads. However, there is increased responsibility as the developer must also maintain the infrastructure as well as the hosted programs (Graham 2003).

Users benefit not only from more reliable tools, but from the community built around them. According to the Treehouse Studio description, files saved with the tools can also be stored on the server, meaning that a designer can effectively work from any computer that has access to the internet. Similarly, members of a group could give each other access to their centrally-stored files. One could easily envision a possible scenario where all stored materials are publicly available, and designers are able to use them freely to make new compositions.

The design goals of the Treehouse framework are progressive and intriguing: an open source code policy, extensible/customizable tools, and the ability to set variable tool constraints or even disable tools for the purposes of pedagogy. If successful, this could be a model for a computational learning and working environment for architects.

Performance remains an issue with web-based applications, but improvements in bandwidth, memory, and system speed will make it a more viable platform in the future. Maeda may be correct about the limitations of working with computer algorithms, but maybe that is because they have always been a solitary effort—one person, one computer. Perhaps with
distributed collaborative work by people and machines, new breakthroughs can be made and this outlook may change.

CHAPTER 4: CHALLENGES

With the necessary background in place, the next discussion will be about what we can learn from these precedents and what must be done to move forward. The challenges facing the widespread adoption of computation come from many sources. To begin, designers often have trouble reconciling how they view their work with the mindset necessary to program. As we have seen in previous chapters, the interface itself is part of the problem. It can be an obstacle of the worst kind—a seductive one. Last, the very identity of the user has been socially constructed to produce a kind of passive relationship that is detrimental to producing the kind of rigor necessary for computation. Defining these challenges is the first step towards constructing a strategy for overcoming them. From here, we can begin to draft well-informed guidelines for future cultural and infrastructural solutions.

4.1 Programming as an Expression

"Writing computer code has to be one of the worst forms of human expression. I'm not 100% sure why...All your human creativity has to compress itself into a tiny ball and recede, while the mechanical creativity has to surface as the dominant force in order to get the job done."

—John Maeda

Programming is certainly useful in other fields, but for designers its value may be seen as suspect and its routine difficult to assimilate with our own. If programming is the best means we have to work with computation, then we should examine the conceptual and perceptual barriers that hinder us from adopting these methods.
To begin, for many reasons, programming is not commonly seen as a creative act. Our society has not yet developed an appreciation for computer generated things. For tasks like optimization, we can agree it saves a person from tedious work. But suggesting the automation of what designers cherish most seems like sacrilege. There are questions of authorship that arise from this arrangement. What exactly is being expressed and whose expression is it? With computation, the critical designer is leery—a fine line exists between using the machine and becoming its caretaker, between giving it work and having it taken away.

What is more, actually doing programming seems boring, sedentary, and unromantic. For someone used to drawing or otherwise manipulating forms on the computer, programming can be uninteresting. In contrast to more fluid activities like drawing or model building, which engage the senses and give immediate feedback, programming is discontinuous and not easily visualized (McCullough 1996). For the designer, there is a disconnection between the written code and the executed result—the analytical and the visual. Concentration is compromised when the user must move in and out of these two different modes. Visual literacy is highly valued by designers and is accentuated by our image-saturated culture. The kind of literacy programming demands is of a different kind than we normally exercise; it is much more tied to composition or mathematics, skills that architects do not receive as much practice in. Thus, we are somewhat ill-prepared for programming because of the focus of our education.

Writing programs has a pace that designers are unaccustomed to, as well. The working style of coding is typically deliberate and methodical. It can be difficult to jump into or otherwise sketch out a concept quickly; a great deal of advance planning and foundational code is often needed. This can make it difficult to place within the design process. If it is used early it may be too restrictive; used late and it might require too much startup time to depend on in a crunch.
The strict nature of code is another potential source of annoyance. Converting one's thoughts into rules is difficult, but with programming, even these rules must follow other rules. Punctuation and syntax need to be perfect or the program might not work as expected; even the smallest mistake can have consequences. This is the curse of explicitness.

The thought process required to program is does not often come easily to designers, one might say architectural culture is not "procedure rich". We are unaccustomed to taking the process we use to arrive at a solution and reducing it to a series of procedures—and we would like to think that such a thing is not possible. As students, designers are socialized early on to obfuscate their logical intentions so they may be presented to others in pleasing ways; it should come as no surprise if later many of us find it difficult to extract those thoughts in a form suitable for code. However, if we were forced somehow to express ourselves in terms of procedural instructions, we might find that we are not as intuitional as we thought. In architecture, there is a significant amount of repetition. Even on our own, we often repeat ourselves. Once we find areas that require repetition, a well-defined procedure can become a tool, and we can begin to see design with a computational perspective.

Cultivating procedural thinking and explicitness will not be easy, but with the right motivation, it could happen. To help this process, there must be an effort not only to change architects' perception of code through education and examples, but to bring code more in line with other, more familiar forms of expression.

4.2 Interface Issues

"By using GUIs all the time we have insensibly bought into a premise that few people would have accepted if it were presented to them bluntly: namely, that hard things can be made easy, and complicated things simple, by putting the right interface on them."

—Neil Stevenson
In attempting to spread computation, computational designers must face the paradox: can there be both mass-adoption as well as rigor? To put it another way: is it interfaces or programming we should be pursuing with regards to computation? Tools can have a tremendous impact on how we design. To begin, we might want to question what this impact is and what consequences it carries.

Applications necessarily carry a set of assumptions about the way the user will work. To use a tool effectively, we must begin to think as the author intended. Inevitably, our habits are shaped by our exposure to various tools; often, it can become difficult to think without them. Creating an interface that is intuitive often involves making use of similarities like common metaphors or sharing features from other applications to create a more familiar setting. What this tends to do is homogenize these tools and stifle the introduction of new concepts and metaphors that might change how we use them. Thus, interfaces tend to steer us in directions which we may not have intended for ourselves. They become so ingrained in our view that it can be difficult to realize this is happening. One advantage to learning programming is that it calls attention to this arrangement.

In truth, everyone has a particular way of seeing things—there can be no perfect general interface. Ideally, designers would all be able to craft an environment for themselves that meets their needs, habits, and preferences. With programming, this is possible because the user is responsible for building assumptions into the tool. In the end, a program is a very personal artifact; one’s thought process is expressed by the code.

Procedural interfaces like visual programming languages would seem to hold potential for computational designers, but given the research, this does not seem likely. Despite all attempts, there is no current graphical interface that can replace programming. The idea of expressing procedural logic with graphical manipulation appears to be flawed. There will
never be enough graphical abstractions to match the versatility of a general purpose language. Conceptually, trying to overcome this by using lower-level abstractions would result in a language that is essentially typed, thus negating any advantage. As an introduction to programming, visual languages might still be useful, but they are so limited that designers would soon exhaust their possibilities.

If our tools shape the way we think, then, for many reasons, programming is a better tool for thinking in. It has the ability to extend our intellect, unlike interfaces which impose their own mindset. Learning to code requires discipline, but this should not dissuade us as much as it does. Think of what we are asking from our interfaces: to be freed from having to think too much. Rather than trying to replace programming and lose the opportunity to improve our minds and our designs, we should invest in infrastructure that works together with code to improve the user experience. This might not completely solve the paradox of mass-adoption, but it could help to make programming more accessible while preserving its strengths.

"(If one asked God about interfaces) He would probably tell you that life is a very hard and complicated thing; that no interface can change that; that anyone who believes otherwise is a sucker; and that if you don't like having choices made for you, you should start making your own." (Stevenson 1999).

4.3 Design Principles

The challenges preventing computation from reaching the mainstream are deeply rooted in culture and cannot be easily dismissed by any single intervention. Instead, it will require a constant effort, developing over a period of time to affect a lasting change. If we consider improving the adoption of computation as a design problem, a useful strategy might be to establish some principles to follow as we work towards this goal. By distilling the essence of a constructive response into some simple statements, it will be easier to remind ourselves as researchers and designers what we are looking for in our
solutions. I believe that summarizing these opportunities for improvement might inspire some new projects in the future.

It should be said that these principles are not rules to follow as much as suggestions. They come from my own study of them problem, and are welcome to interpretation. If anything, they are only a start. I hope that by producing them here, others might begin to challenge them or otherwise build upon my thoughts on the matter.

4.3.1 Transparency

Interfaces work by selectively abstracting concepts and information to make them easier to use. They are powerful, but they are limiting. Unfortunately, designers often have no control over these choices. Thus, what is needed is transparency: clarity of vision means both having a wide field and the ability to focus when necessary.

One should not assume what a designer will find useful, nor should too many options cloud the search for solutions. Information and control down to the lowest level should be available to the user, while at the same time recognizing the need to hide what is not needed or wanted. It should be possible to translate information into alternate representations to help users to think about according to their own needs and perspective. As the number of available options continues to proliferate, the ability to manage tools and information effectively is of the highest concern.

4.3.2 Flexibility

An environment for design computation should strive to give the user as much agency as possible. To this end, tools should be outside of the current product cycle to which creates a consumerist, artificial dependency for features, improvements, and bug fixes from software vendors. Flexibility means being adaptive to needs that are both shared and individualistic.
A new, more open model that gives users greater access to the development cycle is in order. Instead of controlling one’s options upfront, a flexible system should place resources in control of the designer and provide the means to organize them into a solution. It should offer ready tools, but accommodate the process of developing more. This includes alternative solutions, allowing the designer to begin free of assumptions and bootstrapping a new tool from scratch. Finally, for novice users, it should be possible to modify or combine existing tools without code.

4.3.3 Dialogue

While we continue to search for more powerful tools, cultivating rigor and technique with existing tools is also important. One way to accomplish this, and to help in the refinement and development of new tools, is to find ways to add more people to the discussion. Dialogue encourages growth and identity. It gives users a stake in how things are done; it helps create culture.

The tool should create a computational environment that is both personal as well as shared. Though dialogue, each individual’s toolset should ideally evolve into a mixture of the familiar as well as the unique. This is closely tied to the principle of Flexibility (see above). To this end, open source methods should be encouraged to enable the sharing and distribution of code, tools, techniques etc. and also as a forum for peer review, bug reports, and revisions.

4.3.4 Play

Most people would agree that programming is not much fun. New tools should make an attempt to find some technological response to this condition: in other words, to invite play. Play is working without fear of failure. Indeed, it is working with failure, to learn. Managing this is critical to changing the experience of working with code.
Designers should be able to ‘play’ with computation, to experiment and test their ideas, without fear of breaking something, ‘going into the mineshaft’, or generally feeling uncomfortable or detached from their experience. A tool should not punish ignorance; it should enable the construction of knowledge. Error messages should be intelligent: understandable to the average person and capable of suggesting solutions. Playful environments should have the ability to maintain a continuous experience by preemptively fixing minor errors that would otherwise force the user to stop and fix them. Finally, the system should be able to connect designers with each other to solve problems and get back to work (playing).
CHAPTER 5: IMPROVEMENTS

From my research, I developed some illustrations in the following sections that represent strategies for improving the experience of working with code and building communities of support around computation. I would not say they are particularly revolutionary, but rather they combine existing technologies and methods to make something new. It should be said that these tools and interfaces are not meant to suggest a final product. They are a way to visualize how the design principles described in the previous chapter might be implemented. It is important to note that the concepts surrounding the tool are more important than the form it takes. Each of these sections should be thought of as a possible area of research that might warrant further attention. For now, these images and their accompanying descriptions are intended to bring some form to the ideas of this thesis and encourage more imagining to take place.

5.1 Infrastructure/ Community

These speculative tools function as a layer or a shell atop other design programs, running in the background or otherwise in a symbiotic relationship to their “host” applications; this is so they may potentially work with any computer language or application, rather than being restricted to any one in particular. They are intended to help mediate the use of code within design applications in more productive ways than is currently possible. In addition, the tools are connected to the Internet, integrating browser-like capabilities that allow for communities to develop around the exploration of design computation.

The communities are meant to be inclusive, allowing users of different skill levels to work with computation as well as make a contribution to the discourse. With this system, a designer might choose not to program and only use pre-made tools from
the network. This person could still assist others by helping to document these tools, reviewing their effectiveness, and suggesting other tools. At the next level, users may only want to engage in a casual practice of programming. Their usage might include limited customization of existing tools or writing small programs to share. They could also help to write comments, answer questions, or make tutorials. Next would be the experienced programmers who would actually make most of the tools or otherwise take a leadership role. In this manner, every kind of user has a role to fulfill, and a different way to satisfy their computational needs.

5.2 Code Management

Designing a computer language is a difficult endeavor, one few computer scientists have undertaken, much less any architects. What is more, one can never be sure that it will always be up to the task at hand; new knowledge and new situations demand flexibility. To think that architecture could rely on a single computer language to represent its desires and intentions is the same as suggesting we only work with one medium. It would appear to be a better strategy, then, to take a more conservative approach: capitalizing on the material that already exists, accommodating as many languages as possible, and being prepared for any future languages that may come along.

Programming languages work, but they are too difficult for many of us in their current state. Rather than remake a language into something less rigorous, perhaps re-presenting or repackaging could be the answer. Instead of putting too much focus into writing new languages, we might want to consider tools to better organize and manage code.

Some of the cutting-edge research in computer science is in the area of code analysis tools. This involves examining code in both its source and compiled state and performing

\[22 \text{http://del.icio.us/tag/sourcecodeanalysis} \] hosts an ongoing list of links on this subject
computations to create “metadata”, or data about data, that can be used to improve the code. Code analysis can be used to optimize performance, automatically correct errors, and even write documentation.

Code management could be the solution to improving the usability of computation without sacrificing its rigorous qualities. We need something that can accommodate pedagogical and general purpose languages, and leverage the existing wealth of knowledge and resources from other fields. By investing in tools that can tell us more about code and help us to customize it for our own use, we create the ground work for a unified and universal practice.

5.2.1 Modularity

Extending the manageability of code is the first step towards improving its usability. With the modular system illustrated on the next page, segments of code are packaged in such a way that they are able to be treated like objects. A library of collected modules allows the user to search for code by what it does. The module can then be dragged out of the library as a block and inserted into an editing window. Green colored modules in the illustration are fixed and have no dependencies. Yellow colored modules have interfaces that allow the user to customize their settings. Some modules are specifically meant to be combined with others and have graphic symbols to indicate this. Once inside the editing window, packages can be broken down into their component code and edited manually.

The addition of modules to the coding process makes it easier to quickly piece together commonly used procedures and write new tools in less time. Developers can package their own code into modular formats using special descriptors, or outside utilities may be used to automatically separate functions and convert them into modules (for more, see Documentation). For novice users
there is less writing of actual code involved. They can mix and edit the modules together and then upload the new combinations as tools. In this way, it is possible for users of different skill levels to make new contributions to the collection of tools and modules in the community.
5.2.2 Analysis

Programming inevitably involves errors, but the arcane way in which they are usually presented can make the experience even more difficult. If debugging could be made more intelligent, it would make programming a more continuous process. Syntax errors are the most common for new users—even the smallest mistake can cause a program to fail. In my hypothetical system, utilities would attempt to fix these kinds of minor errors automatically by performing code analysis. In the illustration, these are represented by the color green, meaning that they do not require user intervention. The yellow coded errors have been corrected to the best of the system’s ability, but the certainty of the solution is not guaranteed. Users can check on these errors themselves and decide whether the fix is appropriate. Red errors cannot be fixed by the system and must be corrected manually. However, this does not mean the system cannot assist in the process. By linking directly to a syntactical database and user discussions about similar errors, the system makes it easier to find the information needed to fix the problem.
ERROR CHECKING-- parabola.lsp
(viewing LOCAL as SPACEWAR)

(setq counter 0)
(setq upper_points nil)
(setq lower_points nil)

(setq wide (* wide 1.0))

▲(setq x_inc (/ wide (* num_div)))

user intervention requested (record/ chat/ mark)

(repeat (+ 1 num_div)
(setq x (- (/ wide 2.0) (* counter x_inc)))
(setq y (+ (* (- a) x x) h))

(setq pt (list x y))
(setq upper_points (cons pt upper_points))
(setq counter (+1 counter))

check reports (3) STOPS, (2) ATTENTION, (3) PASS

line 32: (diag: syntax check, nearest) parenthesis found and added OK -- keep/ undo
line 32: (diag: variable, missing) no declaration for "num_div" -- go to 32.
line 36: (diag: context, polym) suggest LIST -- commit/ cancel/ evaluate

Analysis Interface
5.2.3 **Translation**

A shortcoming of scripting is that it is restricted for use in a single application. If there were a way to convert code for one program into suitable code for another, it would maximize the amount of these resources available to architects. The interface on the next page is a hypothetical illustration of how this might look, but the idea is not so farfetched. Recall that one of the characteristics of computer languages is the ability to translate low level code from one form to another. It is possible, through utilities called decompilers and interpreters, to disassemble code and render it to a form that is readable in a different language. This is done often with legacy or older languages that few programmers write for anymore. The system utilizes three different methods for accomplishing the translation. The first is the low-level operation described above and the second makes use of publicly available specialized translation programs designed explicitly to work between two languages. The third involves running the code in the original application and attempting to interpolate transformations, transposing them into another program’s functions. With this combination of translation options, this sometimes unreliable operation has a higher chance of success.
examine bytecode...
processing LIST to ARRAY...
function array_sort(array_object, comparison_function, user_data)
dim i = array_length(array_object)
do while(i >= 0)
i = i - 1
dim j = 0
do while(j < i)
dim a = array_object(j)
dim b = array_object(j + 1)
5.2.4 Documentation

One potential source of improvement is not to change the code itself, but to improve user comprehension through better sources of documentation. The most common resource programmers have to understand a program is the comments that authors write into them. Comments are short statements within the text of the code that are meant to be human-readable descriptions. This is recognized as good working procedure when writing a program, especially if the code will be used by others or the author plans to return to it later. However, because it is discretionary, there is no guarantee that the documentation will be there. Nor is its quality assured: comments can often be missing, incorrect, or ambiguous. Comments are helpful because they travel along with code and they are contextual. For novices, they can be a way to learn about how programs work. But their inconsistency is a problem. Therefore, an opportunity exists to make programming more accessible by improving the availability and quality of comments.

To this end, I have provided an example of a potential documentation interface. The code window I illustrated in Figure 1 contains the program locally, but is also connected to the internet. Comments are highlighted and their sources are marked by the presence of colors and icons. Human authored comments are assigned a “person” icon and each individual’s is marked with a particular color. In this case, the original author has been marked in red. Authors are linked to their comments, so other users contact them to ask them questions or make corrections. In addition, it is possible to make edits directly to the comment and then put them to a community vote. Also, comments can use hyperlinks to connect them to more descriptive documents elsewhere on the web.

There is research into making code self-documenting\(^\text{23}\) that could make its way into an interface like this one. It works with the syntax and the semantics of the language to produce

\(^{23}\)\url{http://c2.com/cgi/wiki?SelfDocumentingCode} is a good reference site, with links to many papers on this topic.
contextual reports that are inserted into the code. Because of the nature of the content, these comments are more useful for the original author or programmers who already know the language. As artificial intelligence improves, more meaningful comments will be possible. The modular code used by the system also has self-documenting features (Figure 2.). If there is no description for a module, or the description is vague, labels can be generated locally. Users can remove them if they are incorrect or save them to the module. With the combination of syntax-based indexing and human tagging, users will receive more relevant searches when looking for code.

To describe how this might work, I have included some automated comments in my example (Figure 1.). One of these processes, represented by the powder blue icon and box in the diagram, looks for common patterns and attempts to express them as concepts or formulae that the user can then read about in a more detailed manner. Another reads back a line of code as a collection of simple statements, which can help the user to understand what it means. In the example, the crimson comment box contains a sample of this kind of expression. Working together, the on- and offline systems can compliment each other. Users might even begin with automatically generated comments and use it as a starting point for composing a better explanation.

In Figure 3, I show the ability to “flag” pieces of code. Flagging marks lines within the program and passes a link to them onto the network for other users to see. The most common use of these would be to ask questions or to seek help. In addition to this function, flagging also calls up a dialogue box that allows the users to search for other code, flags, or messages referencing similar contexts. In this manner, the user may be able to find answers without having to wait for a response. Flags stay linked to the code so that other users can read through them in addition to the comments. This “information-rich” code is superior to regular code as a teaching tool and from a production sense.
Nicholas Senske

_Fear of Code_

---

**DOCUMENTATION MODULE-- parabola.lsp**

(viewing GLOBAL-ALL as SPACEWAR)

| (setq counter nil) |
| (setq upper_points nil) |
| (setq lower_points nil) |

;3] repeat to create a series of points on the parabola
;points are stored in 'upper_points
;First, computes the increment for x

(setq wide (/ width 2.0)) ; make sure this is a real number and
;not an integer before division below

(setq x_inc (/ wide num_div))

; Then, repeat to make the series of points on the parabola

 repeat (+ 1 num_div) |
| (setq x (+ wide x_inc)) ; x decrements by x_inc in each |
| (setq y (- (* a x x) h)) ; formula is \( y = -ax^2 + h \) \( \text{parabola} \) |
| (setq pt (list x y)) |
| (setq upper_points (cons pt upper_points)) |
| (setq counter (+ 1 counter)) |

; this list grows in each loop

Figure 1.
Figure 2.
LEARN/ BROWSE MODULE—parabola.lisp

(viewing GLOBAL as SPACEWAR)

class Eye
{
    int ex, ey;
    int size;
    float angle = 0.0;

    Eye(int x, int y, int s) {
        ex = x;
        ey = y;
        size = s;
    }

    int size;
    float angle = 0.0;

    Eye(int x, int y, int s) {
        ex = x;
        ey = y;
        size = s;
    }

    void update(int mx, int my) {
        angle = atan2(my-ey, mx-ex);
    }

    void display() {
        push();
        translate(ex, ey);
        fill(255);
        ellipse(size/2, -size/2, size, size);
    }
}

Figure 3.
5.3 Design History

For those with little coding experience, programming by example is an option. Here, I show an interface for capturing user operations from CAD programs that functions as a design history. In practice, this is very similar to the Photoshop Actions tool, in that the user can selectively record functions from the program and use them to automate tasks. Once inside the history window, actions can be duplicated, moved, deleted, or otherwise manipulated to produce new functions. This is an easy way to program simple macros, but with the interface's ability to export them to code, more possibilities are opened up. By associating this code with the actions that generated it, users can learn more about programming. This is a powerful way to transition users from applications to working with computation.
HISTORY MODULE (viewing GLOBAL as SPACEWAR)

Reading from last state:
(SCENEid #GBF0912)

interpolating:
(SCENEid #GBF0912) x h00000000
....
(SCENEid #GBF0912) x h000000001
....
 storing: xcoord 30, ycoord 40

HISTORY -- RECORDING FROM: AutoCad2007 [maison.dwg]

2D RECTANGLE
X: 30
Y: 40
EXTRUDE
X: 10
SELECT (SAME)
COPY
MOVE
X: 5
Z: 10
END

Design History Interface
5.4 Evaluation Interface

While there are computational filtering techniques for taming generated forms, aesthetic evaluation is not easily computed nor is it considered desirable. In these instances, a human has to intervene. Unfortunately, there is no efficient system for producing variations, presenting them in a meaningful manner, and relating user decisions back to the code. Typically, this involves a tedious cycle of moving between analyzing textual information and contemplating visual information: making changes to the code, viewing the result, submitting new changes, and repeating. This process can be jarring mentally and generally disruptive to the momentum of design.

In contrast to this method, the evaluation tool in the illustration presents the user with multiple potential scenarios for different variable choices represented by small thumbnails of the outcome. These thumbnail images become the interface—the user can manipulate them to set up a new range of trials or pick the visualization closest to his or her intentions and have the settings applied.

As a learning tool, novice users can explore how code works by taking existing samples and running them through the evaluation engine to see how changes affect the outcome. In a production environment, it would be useful for quickly examining a range of possibilities and isolating designs for further study. Parametrically generated models are ideal candidates for this procedure.

With widely-available parallel processing set to arrive in the near future, the ability to perform simultaneous complex visualizations will be a common feature of many programs. This will dramatically increase designers’ ability to experiment in ways that have never been possible due to performance bottlenecks.
Evaluation Interface

void setup()
{
  size(200, 200);
  noStroke();
  fill(102);
  rect(0, 0, width, height);
  offset = width/2;
}

void draw()
{
  // Smooth the mouse
  nmx = mouseX;
  nny = mouseY;
  if(fabs(mx - nmx) > 1.0 || fabs(my - nny) > 1.0)
  {
    mx = mx - (mx-nmx)/20.0;
    my = my - (my-nny)/20.0;
  }

  // Evaluation Matrix Setup
  2 x 2

  x: mouseX
  y: mouseY
CHAPTER 6: PEDAGOGY

After the discussion of new tools in the previous chapter, this section examines how computation might be received and taught within the existing discipline. The first two sections are concerned with how to prepare the individual, while the remaining section moves into the conclusion of the thesis by considering how even those who would not embrace computation may still benefit from the experience and make a contribution to the discourse.

6.1 Identity Concerns

"We're looking for ways to lose control of the design process."
- Joshua Ramus, partner, OMA

Specialization is an accepted fact of most professions, but integrating programming within the profession brings unique challenges. Computation is different because it may concern the very identity of the architect.

There is a serious ongoing debate about where architects fit into a world where the programs seem to be taking over. Many are worried about the potential of intelligent automated systems to take over designer's jobs. Others revel in the depersonalization of architecture with talk of self-assembling 'emergent' systems. These architects like to speak about how working with algorithms allows one to give up control and therefore arrive at expressions the designer would never think of himself. Computation has the potential to surprise us with complexity, but this does not mean that it is beyond our control. It is popular to speak of computer generated forms as if the computer is somehow responsible for them. But as John Maeda spoke during a lecture: “You don’t call cookies ‘oven-generated’”. The computer is the means to ‘bake’ the design, but the designer has made the ‘dough’, choosing the input for the program and making the rules. The final decision of output
also rests on the designer. In the end, “giving up control” is actually a carefully orchestrated process.

For similar reasons, design will never become a fully automated task. Artificial intelligence is still inadequate for doing most mundane things, much less a complex and subjective affair like design. John Maeda once said during a class: “If we cannot teach every student to be a good designer, I do not believe we can teach a computer.” To be honest, the kinds of things that can be reliably automated for architecture are those that should be: tedious tasks that do not require much thought or introspection to perform. It is less likely that people will be replaced because of such programs, as the parameters and density of their work will change. Historically, this is what occurs with industrialization.

Regardless of our fears or our zeal for the computer, the machine has no will of its own; it has no needs, and no clients to satisfy. In the wake of this ambiguity of identity stirred by generative systems, as we try to spread computation one of the most important lessons we can teach is that intent will always reside with the designer.

6.2 ‘Hello World’: Scenarios for Learning

‘Hello World’ is traditionally the first exercise in computer language tutorials. It is the simple act of getting the computer to print the phrase “Hello World!” In thinking about how to introduce people to computation, we may want to consider what the design equivalent would be. What is the most constructive way to begin?

I am often asked by other students how one begins to work with programming. This is interesting to me, because MIT is one of the best places to receive a formal education in the subject, from the perspective of architecture as well as computer science. But as architects, we have a strong belief in individuality and a tradition of self-education, especially when it comes to technology. This is because mainly because
technology moves much faster than pedagogy can possibly adapt. Students must often teach themselves in order to keep up with new programs and techniques.

In the process, what they tend to miss are the concepts behind the tools, which is a difficult perspective to develop on one’s own. A conceptual education is important because as the tools change, the concepts do not. Knowing these can lead to more effective usage patterns and makes it easier to adapt to new tools. With computation, this kind of high-level discussion is critical; rote learning will not take one very far. For educators, striking a balance between the broader conceptual lessons and keeping up with the specific technical ones is a challenge.

One possible solution is to have instructors focus more on teaching important concepts and to encourage communities where students and educators can learn the latest programs and techniques together. This is one possible use for the tools from the last chapter. The online component could support an emerging discourse for programming, and provide an up to date resource for users. The goal for the broader context is not to get too immersed in any particular tool, but to develop sufficient faculty for one’s intentions and have the foundational knowledge to transfer this knowledge to new platforms.

Returning to the earlier question about beginning to program, it can be difficult to say exactly because everyone learns differently. Paul Graham lists some rules of thumb that can help (2003). The best way to start is to have a purpose: a project or a problem to solve—this will drive the effort. Instead of beginning from nothing, at first, one should try to find some existing code and start modifying it. Later, it might be helpful to look at other programs and examine their source code. Finally, talking to other people who program will help to solve problems and generate new ideas. Notice that each of these rules of thumb is supported by the speculative tools from the last section. My intention was to suggest this process of learning to the user through the framework, with the online component facilitating interaction with other programmers.
Another question I am asked is: “Which language is best?” Although many programmers have a favorite computer language, this is simply a matter of preference. There really is no “best” language. General-purpose languages accomplish roughly the same thing, but some are better than others at accomplishing certain tasks. Knowing their various strengths and being able to use them when appropriate is more important than knowing one language and trying to do everything with it. As with software applications, if one learns the concepts, picking up new languages is not difficult.

But to begin teaching oneself, I would suggest something with constraints that focuses on fundamentals. Design by Numbers is not as fashionable as it used to be, but I still prefer it over Processing because it is so limited that it allows one to focus. Then again—speaking of focus—Processing has a large library of inspiring user-submitted examples. Ideally, the best scenario for learning to program is to do as the weight trainer: pile on the resistance and seek motivation.

6.3 A Culture of Computation

“*I was playing against myself and against something I couldn't recognize.*”

-Gary Kasparov, *after his defeat by chess computer, Deep Blue*

To dismiss computation as just another tool overlooks the profound impact it is capable of. New technology is a new way of seeing: a cultural change that can affect everyone in some way. The overall tone of this thesis sets a positive outlook for the adoption of computation, but, realistically, not every architect will make use of it. Some will not be able to learn it well, and others will not want to use it in the first place. Regardless, there is much to be discovered from encountering it and many roles to play in a computational pursuit of architecture.
Nicholas Senske  
*Fear of Code*

Much has already been said about the productive potential of computers, but it is first and foremost a tool for thinking in. Even if one does not apply programming in their designs, taking the time to learn can change the way one thinks about their work and about other computer programs. Most of our time is spent thinking about what to design; thinking about *how* we design is something we rarely ever do. Instructing the computer to do something is a way of teaching ourselves to externalize our process. Even without the application of computation, this can be helpful for coming up with new ideas. Also, learning to program makes one more aware of why our tools work as they do. Understanding limitations is important to getting the most out of our software; if users are so compelled, computation even provides the means to overcome them. If nothing else, learning something about programming might help someone to appreciate the fact they do not have to program.

A mature and productive discourse needs its critics, and computation is no different—even its detractors can learn and contribute. For those who worry about the threat of automation, learning more about computation is the surest way to dispel these fears. Designers who take issue with the digital are best served by speaking from within:

"*The best way for doubters to control a questionable new technology is to embrace it, lest it remain wholly in the hands of enthusiasts who think there is nothing questionable about it.*" (Brand 2005).

The introduction of computation has already had an impact on the practice of architecture, but for the most part, it has yet to significantly engage the culture. With a broader pursuit ahead, design itself will change as we adopt a different view of our process and our relationship with our tools.
CHAPTER 7: CONCLUSION

This thesis attempted to suggest some how a widespread practice of computation might develop. It began with the reasons encouraging this practice and a brief introduction to the problems it faces. The next sections presented the background on computer interfaces and precedent work. After the preparatory work of these chapters came a summary of challenges and a response in the form of design principles for future work. Then followed a proposal for tools based on these principles and a discussion speculating on directions for pedagogy. The intention was to summarize this area of computational research and to synthesize these findings into potential directions for further study.

Over the course of the thesis my perspective changed to the point where I had to completely rethink my approach. When I began, I focused on lowering the barriers to usability, but I did not realize the compromises that were necessary. This led me to the paradox of trying to simultaneously cultivate both mass-adoption and rigor. I realized too late that in these situations attempting to cover both angles (as I did with the tools I suggested) often ends up producing a weak solution. In the end, despite the infrastructural improvements, what my approach lacked was a source of motivation for others to use the system. Because of my experience with this thesis, I plan to move away from tool-making and pedagogy for a while and focus on producing more designs. Perhaps with this additional work, I will have a different outlook on the problem.
Bibliography

Theory/ Discourse of Computation in Architecture


CAD/Interfaces/Tools


Programming (General)


Pedagogy (Programming/ Computation)


Pedagogy (Architecture)

Sociological and Cultural Dimension of Computers/Computation

Websites

Miscellaneous