Inspiration From the Incomplete: The Role of Asynchronous Awareness in Digital Art Creation.

Kyle Matthew Buza

B.A. Biology, Cornell University (1998)
B.A. Chemistry, Cornell University (1998)
M.Eng. Computer Science, Cornell University (2000)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning, on May 9, 2008 [June 2008]
in partial fulfillment of the requirements for the degree of
Master of Science at the Massachusetts Institute of Technology

Author **Kyle Matthew Buza**
Program in Media Arts and Sciences
May 13, 2008

Certified by **John Maeda**
Associate Director of Research, The Media Lab
Thesis Supervisor

Accepted by **Deb Roy**
Chair, Departmental Committee on Graduate Studies
Program in Media Arts and Sciences

Inspiration From the Incomplete: The Role of Asynchronous Awareness in Digital Art Creation.

Kyle Matthew Buza

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning, on May 10, 2008
in partial fulfillment of the requirements for the degree of
Master of Science at the Massachusetts Institute of Technology

Abstract

The majority of visitors to sites on the World Wide Web (WWW) have traditionally been only passive observers; consumers of previously created content. More recently, however, these users have been encouraged to contribute to these sites, opening the door to new forms of creative self expression. As we enter this new era of widespread collaboration and sharing made possible by the WWW, one question that remains is how to build appropriate communication channels to and from this new medium with respect to the tools used for digitally mediated creative expression. In this thesis, I will attempt to formulate a coherent set of characteristics that both creative programming environments and their associated WWW sites must possess to help improve, inspire, and support the work of creative individuals using these systems, which I will refer to as *architectures for web-based collectivity.*

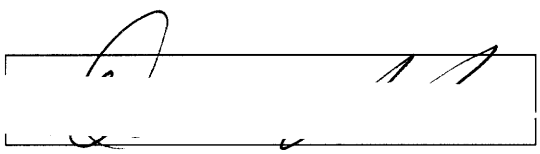
Inspiration From the Incomplete: The Role of Asynchronous Awareness in Digital Art Creation.

Kyle Matthew Buza

Thesis Reader **Robert C. Miller**
Associate Professor
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Inspiration From the Incomplete: The Role of Asynchronous Awareness in Digital Art Creation.

Kyle Matthew Buza



Thesis Reader **David P. Reed**
Adjunct Professor
MIT Media Laboratory

Acknowledgments

I'd like to thank my readers, David Reed and Robert Miller, for providing me with the valuable feedback upon which this thesis is based.

Takashi Okamoto and Luis Blackaller, for being my partners in crime.

Members of the Physical Language Workshop: Kate Hollenbach, Brent Fitzgerald, Amber Frid-Jimenez, and Laura Martini, for teaching me things I never knew.

Amna Carreiro, for taking care of things.

My parents, who respectfully dealt with my disappearance into the bowels of MIT for the past two years.

And my advisor, John Maeda, for seeing something in me that I did not, and helping me to become something I've never been.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "The net is not a place for 'professionals' to publish and the masses to merely download. Online, everyone is becoming an artist; everyone is a creator. The network is providing new opportunities for self expression, and demands a new kind of artist: the artistic instigator, someone who inspires other people to be creative by setting a positive example with their own work, and providing others with tools, context, and support. That support can be technical, aesthetic, or emotional– encouraging others to believe in their own capabilities and take the risk of trying to make something personally meaningful." [21]

The majority of visitors to sites on the World Wide Web (WWW) have traditionally been only passive observers; consumers of previously created content. More recently, however, these users have been encouraged to contribute to these sites, opening the door to new forms of creative self expression. As we enter this new era of widespread collaboration and sharing made possible by the WWW, one question that remains is how to build appropriate communication channels to and from this new medium with respect to the tools used for digitally mediated creative expression. In this thesis, I will attempt to formulate a coherent set of characteristics that both creative programming environments and their associated

WWW sites must possess to help improve, inspire, and support the work of creative individuals using these systems, which I will refer to as *architectures for web-based collectivity.*

1.1 Motivation

Over the course of the past decade, we have witnessed a rather dramatic shift in the granularity of information we obtain from the WWW. As network bandwidth has increased, and WWW sites have become nearly trivial to construct, massive amounts of data are being placed on the WWW, accessible to millions of people almost instantaneously. Accompanying this widespread accessibility of consolidated data collections is the emergence of new vehicles for pedagogical, as well as creative-inspirational purposes.

In this section, I will discuss the use of programming as a creative tool that, through exposure via the WWW, has the potential to facilitate the transition from creative individuals acting in isolation to ones that are more effectively able to teach and inspire others.

1.1.1 Programming as a Creative Medium

Accompanying the widespread adoption of the personal computer for creative purposing in the 1980s was a large dose of skepticism echoed from the halls of the traditional design community. Scripted, procedural experimentation was viewed as the antithesis of the staid and cerebral aspects of modern design[34]. As a result, computer programs were effectively placed underneath a microscope in order to help demystify the otherwise invisible creative processes made possible through their use.

In recent years, we have witnessed an explosion of these computer-based creative programming tools. They are everywhere, all addressing specific challenges in need of simplification. Programming tasks that were once

difficult have now been generalized and abstracted to streamline their adoption and integration into mainstream design culture. This transition has left both students and educators sitting at the microscope, perhaps with little recollection of how they arrived there in the first place. In this myopic state, everyone has been too busy staring at the trees to notice the forest growing around them.

This newfound fascination with code is unsurprising, as the expressive potential of this digital medium is enormous. It's possible that this interest is, to some extent, misguided. An infinitely malleable canvas should be just as inspiring to artists as it is terrifying. Immersed in the abundance of possibility, many individuals simply end up being stifled by it. Representing the full landscape of expressive potential offered by any creative digital tool is difficult. Today, on virtual desktops embedded within a 2D window-based interaction metaphor, our computers are only able to provide us with a tiny snapshot of the much larger creative domain offered by these tools, as we can only see so much on a 2D screen. In the digital realm, these snapshots serve as lightweight externalizations of the programs used to generate them; programs that themselves are notoriously difficult to describe in any other way:

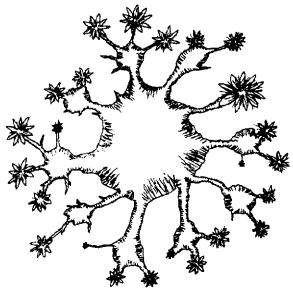
> "... The understanding of how programs work individually and in cooperation with each other.., remains very difficult to generalize, teach, communicate, or even preserve, due to lack of easy 'externalization,' i.e. representation, of ideas."[19]

1.1.2 Example-Based Pedagogy

The current set of creative tools leveraged by digital artists relies heavily upon the use of examples in order to educate new users of their expressive potential. These examples serve as motivating agents for new users to begin learning and exploring the new digital landscape. They become the provenance of new experiments and creations that often remain technically and aesthetically similar to the original. It is in this spirit that original versions of the Unix operating system (OS) were modified, serv-

ing to inspire the development of the open source Linux OS, and a rather large collection of variants inspired by Linux itself.

Whereas modern OS kernels contain millions of lines of code, aesthetically pleasing graphics programs often contain on the order of hundreds. New design-centric graphics tools are being created at an ever-increasing rate, each providing individuals with a new creative domain within which they may experiment, explore, and create. They are often designed to be simple, offering streamlined programming interfaces to their functionality. The resulting short, easily-digestible code snippets lend themselves to distribution on the WWW and serve as a pedagogical alternative to complex API specifications.



Original examples often serve as the substrate for future work.

When distributed via the WWW, these snippets serve as social cues to new socially influenced creative explorations, referred to as "artifacts" in the context of the cognitive sciences[40]. The aggregate sum of these artifacts can be viewed as a sub-landscape of expressive possibility offered by these environments.

1.1.3 Influencing Creative Behavior

Creative thinking and the artistic process have long been viewed as the product of the isolated individual mind. This notion has been brought into question in recent years, and is no longer generally accepted[26]. In an age of creative digital tools accompanied by streamlined mechanisms for sharing and distributing content created with them, it has become increasingly difficult to remain entirely uninfluenced by the masses.

We are entering a new chapter in the evolution of this type of socially-influenced creativity, transitioning from a more isolated model to one where sharing and exposure are increasingly relevant. Underlying these new relationships is the emergent collective coordination through the set of strictly local interactions occurring between group members known as *stigmergy*. In other words, local artifacts left by contributing members can be interpreted by each member of the community, affecting future be-

havior. This notion provides the theoretical foundation for Swarm Intelligence (SI) research, with applications to robotics, computer simulation, and art. The Portugese conceptual artist Leonel Moura, for example, has explored the production of emergent visual aesthetics through collections of independently operating robot actors.

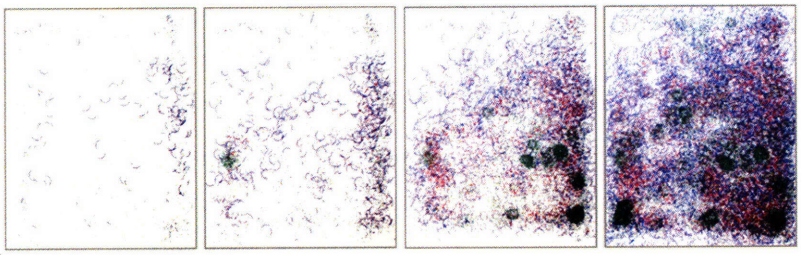


Figure 1-2: **The evolution of artwork created by Moura's stigmergic *mbots*, whose interactions are based solely on local sensory input.**

Figure 1-1: Leonel Moura's *mbots*, programmed to exhibit stigmergic behavior to create abstract artwork.

Just as the introduction of the camera during the Industrial Revolution influenced Impressionist artists like Monet and Degas, recent technological advancements have influenced the number of creative possibilities available to aspiring artists. Presented with a seemingly endless supply of tools, each with their own learning curve and subtle minutiae of expressive potential, individuals may become overwhelmed unless the paths to creative exploration are cleared, and access to expressive tools simplified.

1.1.4 The Role of the Web

The influence of technology on art and creativity is undeniable. In the age of the Internet and WWW, the stage is set for new forms of social creativity to take shape and have widespread influence. Universal access to the WWW is a relatively recent phenomenon, and the capabilities of WWW browsers continue to evolve rapidly. However, precisely how it can enhance new creative acts and host collaborative engagements remains unclear. As browser capabilities continue to improve, new mechanisms to support creativity on the web will appear. The recent appearance of web-based video editing tools and word processing applications is a reflection of this trend.

We live in an age where everything is on the WWW, and every day we

are provided with new tools to expedite the transfer of our data to it. From small snippets of text to images and video, WWW sites are opening their doors to end-user contributions. Storage is now inexpensive, and there is value in harvesting end-user data. While the time and resource investments put into sites like *Flickr*[5], *YouTube*[17], and *del.icio.us*[2] may once have been relatively high, open source copycats are now everywhere, and can be set up by anyone in a matter of hours. While these "consumer-grade" versions lack the scalability and polish of their ancestors, they demonstrate the openness and highly democratic nature of the web. It is through these web-based aggregation methods that new mechanisms for seeing large collections of data are possible.

Sets of images of the Eiffel Tower contributed by users from around the world have been consolidated into large-scale web-based collections that are now viewable by everyone. It is through this organization that users are able to obtain a perspective that extends beyond their own individual collection. This perspective shift was made available through the "collective comprehensiveness through overlapping patterns of unique narrowness", described by Donald Campbell in relation to his *fish-scale model of collaboration*[23].

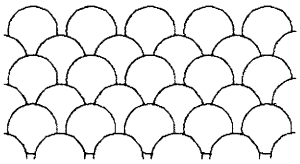


Figure 1-3: Campbell's *fish-scale model of collaboration*, also described as the "collective comprehensiveness through overlapping patterns of unique narrowness".

1.2 Defining the Problem

Whether we like it or not, technology will continue to invade and transform traditional notions of design. Elements of existing techniques will become easier with the aid of computer-based tools, and entirely new forms of technology-based design (e.g. motion graphics, web design) will continue to emerge. In light of these changes, both design educators and students are in need of new ways to understand the creative scope offered by new computer-based tools, as well as ways to obtain support and inspiration from existing communities of creators. Current code-based design tools, for example, are often difficult to describe in a few sentences, and are even more difficult to describe with respect to their expressive potential. How can we go about providing the users of these tools, as well

as their educators, with an easily digestible summary of the creative landscape made possible through their use?

Over the past decade, the WWW has excelled in its ability to facilitate the distribution of just about all forms of digital media. New applications are created daily that allow users to enhance their creative potential. Some are simple, many are complex. As a result, the creative palette available to the digital artist is larger than ever, and continues to grow. With these applications in the hands of end users, they are beginning to produce unique content at an ever-increasing pace. One problem with this model is that because the majority of these tools are desktop applications, created content rarely leaves this isolated context. How can we leverage this explosion of created content to inspire potential artists and creative individuals? How can this content be exposed to others as examples to be modified for pedagogical purposes? How can experimentation be made both fluid and easily digestible? Finding answers to these questions will be challenging on a number of levels. The consolidation of data collected from end users is likely to involve issues of end-user privacy and trust, and mechanisms need to be built to encourage and maintain active participation.

In this thesis, I will attempt to demonstrate, from the perspective of the digitally-enhanced creative act, the most salient aspects of the WWW necessary to enhance end-user learning and creativity with respect to creative programming.

1.3 Thesis Structure

This thesis is structured as follows:

- *Background*: In this chapter, I will discuss the history of digitally-mediated creative acts, as well as the history of modern computer-based tools for creative expression and pedagogy. I will show how work created by these tools was disseminated, and today's role of

the WWW in this distribution process.

- *Experiments*: In this chapter, I will present two architectures for web-based collectivity, namely *OpenCode*, and *E15:Web*, an environment based on *E15*, a new design-oriented programming environment.

- *Evaluation*: In this chapter, I will present the user feedback obtained from the development of the OpenCode and E15:Web systems.

- *Conclusion*: In this chapter, I will summarize the lessons learned from this research, and propose future research directions.

Chapter 2

Background

In this chapter, I discuss the history of digitally-mediated creative acts, as well as the history of modern computer-based tools for creative expression and pedagogy. I will show how work created by these tools was disseminated, and discuss today's role of the WWW in this distribution process. In addition, I will briefly discuss new applications and architectures targeted at augmenting the 2D WWW browser experience – both visually and technically. These discussions will serve to contextualize the the experiments conducted in Chapter 3.

2.1 Definitions

As the base level of computer literacy continues to increase, the number of users interested in programming for the purpose of self expression will increase as well. In the context of rich, interactive media, a number of tools exist to facilitate this user-to-programmer transition. Each of these tools has been developed out of a need for more general environments within which expressive digital media can be created in a flexible manner. Visual programming environments like Max/MSP and text-based environments like Processing have successfully abstracted out specific elements of this development cycle. Max/MSP, for example, provides a

simplified interface to the *dataflow* of a program, while the Processing programming environment provides a *screen buffer abstraction* to simplify the creation of code-driven procedural animations. In this thesis, I will refer to these types of systems as *design-oriented programming environments*, or *DOPEs*. The architectural decisions made by the designers of each DOPE define the outline of the creative landscape made possible by each.

2.2 Computer Mediated Creativity

As the large mainframe computers of the 1960s transformed into their smaller, more lightweight counterparts, they began to enter into the homes of the general public. It was this transition that set the stage for a new generation of digital artists and computer-mediated creators. These individuals originally produced work in isolation, later creating custom systems for the sharing and collaboration surrounding their work.

2.2.1 Early Collaborative Systems

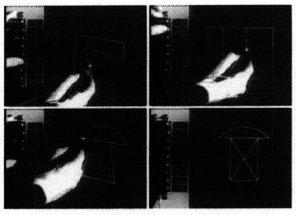


Figure 2-1: Ivan Sutherland's Sketchpad is considered to be the precursor to modern-day CAD tools.

In many respects, the provenance of Human Computer Interaction (HCI) with significant implications for the creation of new forms of digital media was in 1963, when Ivan Sutherland created Sketchpad[43], an interactive program that is considered to be the precursor to modern day Computer Aided Design (CAD) tools. This system demonstrated new techniques for both graphical user interface (GUI) design and a hierarchical end-user program structure. Sketchpad inspired Douglas Engelbart to develop the *oNLine System* (NLS)[30], an environment designed to facilitate networked collaboration. The NLS used a keyboard in conjunction with the first computer mouse in order to support the editing and organizing of shared files. More specifically, the "NLS Journal" subsystem was designed for the management of personal data, including short messages, mail, and data records. Many of these developments led to modern day groupware and hypertext systems.

The ideas explored by Sutherland and Engelbart have influenced many aspects of modern computer interfaces, and painted the landscape for the future of digital creativity and collaboration. The desire to build systems like the NLS — systems that enable human beings to share information with ease — has often been associated with the Memex[22], Vannevar Bush's fictional system he described in an *Atlantic Monthly* article published in 1945. This microfilm-based system allowed its users to navigate large document archives with ease in order to realize new relationships between disparate pieces of information. Motivating these thoughts was the conviction that the value of collaborative intelligence is greater than the sum of its parts. Widespread communication and collaboration are necessary requirements to maximize the potential of human understanding. Today, systems that are built upon these ideas are known as *collaborative systems*, or *groupware*, ideas upon which WWW-based collaborative systems such as Wikipedia are based. Unfortunately, the current state of these systems is still a far cry from Engelbart's original NLS vision.

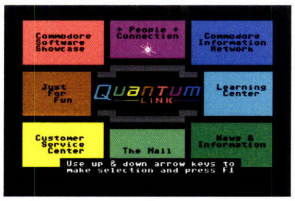
2.2.2 The Rise of the Personal Computer

The NLS was built around the first time-sharing computer, an SDS 940, which could support sixteen connected workstations. These workstations, today commonly referred to as *thin clients*, provided end users of the NLS with access to shared data. Unfortunately, the burgeoning popularity of the personal computer in the 1970s allowed people to use their computers within the privacy of their own home, severely impeding attempts at large-scale information integration enabled by time-sharing systems like the SDS 940. At the apex of this shift was the appearance of the Sinclair ZX80 personal computer in 1980. Targeted at the more general home user as opposed to computer enthusiasts themselves, the success of the ZX80 demonstrated the potential of this emerging consumer market.

With computers sitting in the homes of individuals, and only occasionally connected to a network, the sharing of user content was difficult, and

therefore much less common. However, because digital content creation was subject to more limited means of production during this time, large-scale systems for content sharing were unnecessary. Cameras were analog, and the toolsets used to create purely digital content were only just beginning to be developed. Today, in spite of attempts to return to the mainframe model of computing — where small, lightweight client computers communicate with large servers over the network in the NLS spirit — high performance personal computers remain popular with consumers. In the absence of a system to facilitate sharing between creative individuals in the form of a central time-sharing server, Engelbart's vision hasn't yet become reality.

2.2.3 Early Content Sharing Systems



The main menu in Quantum Link, an early BBS.

With the personal computer finding its way into the homes of thousands of individuals, the desire to remain connected to others and share digital information began to blossom. In the 1970s, Bulletin board systems (BBSes) became popular among computer enthusiasts, which provided ordinary computer users with the ability to share various types of media and communicate with others. One of the earliest systems was the *Community Memory*, a digital content distribution system also built around the SDS 940 used by the NLS. This system, as well as more advanced BBS-like systems of the 1980s like the Commodore 64/128-based *Quantum Link*, supported a rich variety of services that were precursors to modern internet-enabled functionality such as online news, chat services, games, and file sharing.

In addition to these more traditional forms of content, many BBS users were interested in sharing digital art within these communities. One such community, known as the *artscene*, began sharing artwork made from ASCII text on early BBSes, and later used *FidoNet*, a network used for inter-BBS communication in 1984. A few years later, artscene members began to form their own customizable BBSes for distribution of their artwork, which was effectively replaced by the internet around 1995. As network technologies advanced, so did the capabilities of the computers

themselves. ASCII artwork evolved into ANSI (16 foreground colors, 8 background) art, and later RIP (*Remote Imaging Protocol*), an early vector graphics protocol in 1993.

These works had value within these communities, often able to grant creators access to exclusive BBSes, and in some cases, monetary compensation. This value created competition between artists, which subsequently bred creative inspiration[31]. As these communities grew, many began to distribute collections of the generated ASCII and ANSI artwork (known as *artpacks* or *demos*) as well as music sequences, poetry, and code. In addition to providing creative inspiration to the recipients, demos also made it easier for individuals to modify and learn new techniques from existing work.

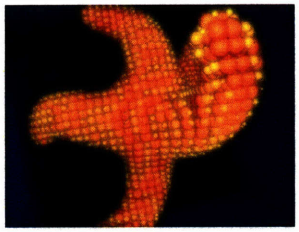


Figure 2-2: *sprite-o-mat*(2007): A recent demo by the Alcatraz group.

> "demos are the last bastion of passionate, crazed, enthusiast-only programming, crafted purely for the hell of it by inspired teenagers working entirely in their spare time. The teens create jaw-dropping audiovisual effects beyond the dreams of most multimedia designers. Constantly striving to better their rivals, devotees of the demo scene cram spectacular three- or four-minute presentations onto a single 800-Kbyte floppy disk, fitting them into tiny amounts of memory. Freely spread by disk-swapping over bulletin boards and other sites on the Internet, then replayed on home computers all over the planet, each demo becomes a piece of digital graffiti proclaiming the superiority of the gang that created it. Demos are made by the rock-and-roll groups of code." [31]

The tightly-knit nature of these groups allowed them to define their own standards with respect to the quality of the artwork disseminated within them.

2.3 Design Oriented Programming

As members of the demoscene were creating visually-compelling work by writing optimized assembly code, other attempts were being made to simplify the process of programming itself, making it more accessible to a wide variety of people. These environments have provided a new generation of musicians, artists, and designers with a rich collection of expressive digital tools.

The environments listed in Table 2.1 have served to influence the work presented in this thesis, and to demonstrate the need for new models for the interactivity and distribution of creative work. The characteristics of the DOPEs listed will be explained in more detail in subsequent sections.

Name	Description	REPL	Extensibility
Processing	Tool for developing visually-oriented software	No	L/-
Max/MSP	Visual programming environment for multimedia	N/A	L/-
Fluxus	System for LiveCoding 3D graphics	Yes	-/-
E15	Environment for creating 3D animations and visualizations	Yes	L/P

Table 2.1: **Listing of current design-oriented programming environments. *REPL* specifies support for *read eval print loop*-style execution. Some of these environments can be extended on the *language level* (L), *platform level* (P), or both.**

2.3.1 The Influence of Lisp

In 1960, John McCarthy published *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*[36], which described Lisp, a programming language constructed through the definition of a small set of operators and functions. Lisp had no distinction between compile-time and runtime, allowing users to write and compile code at the same time. In the years following, a number of Lisp variants have emerged, all using the same syntax (parenthesized lists, or *s-expressions*). The interpreted nature of Lisp served to enhance its interactive charac-

teristics, and it has been the basis for a number of simplified languages intended for pedagogical purposes.

In 1967, Seymour Papert and Wally Feurzeig created the *Logo* programming language, often referred to as "Lisp without the parentheses". It was designed primarily for educational purposes, and remains well-known as a tool for the creation of simple vector drawings, known as *turtle graphics*. Papert and Feurzeig designed Logo under the belief that computer programming was not simply a matter of accomplishing a task, but rather a tool that could be used to achieve a profound insight into a diverse set of academic pursuits. This theory of learning, known as *constructivism*, became manifest in the *Computer Clubhouse*[41], an after-school program designed to help children learn through technology and to socialize the computer-enhanced learning experience.

While the notion that programming can improve a student's ability to learn in a more general sense is relatively common in the context of educational programming, rigorous studies have yet to show a clear relationship[35]. Regardless, many Logo-inspired languages such as *Squeak*, based on Smalltalk, and *Scratch*[13], a visual programming environment for children, have emerged since this time. Other related systems have targeted specific application domains, like the Geometer's Sketchpad[33].

In the early 1980s, Hal Abelson and Andrea diSessa built *Boxer*[28], a programming environment attempting to leverage our natural conception of space to assist young programmers. The Boxer prototype foreshadowed the development of modern-day visual programming environments like Max/MSP by merging visual and textual elements into a single environment, and by providing a more intuitive and expressive mechanism for programming beyond simply editing and evaluating text. A collection of spatially arranged regions, called boxes, can contain program text, data, graphics, or other boxes, similar to the structure of Max/MSP.

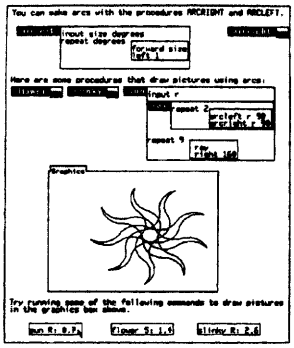


Figure 2-3: The Boxer visual programming environment was designed to harness our natural conception of space.

2.3.2 Processing

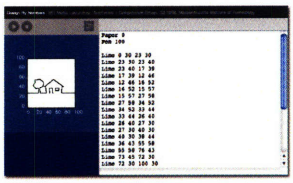


Figure 2-4: John Maeda's *Design By Numbers*

In 1999, John Maeda created Design By Numbers (DBN)[3], a pedagogical tool for designers to explore the use of code for creative purposes. The DBN WWW site describes the language as follows:

> "Design By Numbers was created for visual designers and artists as an introduction to computational design. It is the result of a continuing endeavor by Professor John Maeda to teach the "idea" of computation to designers and artists. It is his belief that the quality of media art and design can only improve through establishing educational infrastructure in arts and technology schools that create strong, cross-disciplinary individuals."

DBN was a Java application that could run as an Applet within a WWW browser. The DBN code was interpreted to generate static drawings on a small canvas.

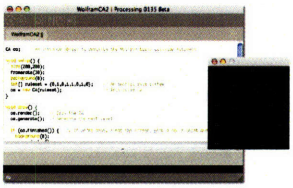


Figure 2-5: The Processing sketchbook

After contributing to different components of DBN, two of Maeda's graduate students, Ben Fry and Casey Reas, created Processing[11], a Java-based environment for developing graphically rich animations and visualizations. The Processing "language" is effectively Java syntactic sugar, performing compile time transformations for simplified drawing procedures (such as *rect()* to draw a 2D rectangle) into their Java equivalents. The Processing application itself is a simplified IDE, referred to as a *sketchbook*, for writing these types of programs. Since its release, Processing has had a substantial amount of success within communities of creative individuals interested in leveraging code for expressive purposes. Its Java foundation provides the environment with the following benefits:

- *OS Independence*: Java runtimes exist for the most popular operating systems, including Windows, Linux, and OS X.
- *Browser integration*: Applets made within the environment can easily be embedded within Java-enabled WWW browsers, allowing

users to publish their work.

- *Extensibility*: Users can write external Java libraries that can be leveraged by Applets created by the Processing environment, extending its functionality.

The simplifications made by Processing come at a cost, however, as the additional layers of indirection that both Java and the Processing runtime insert between a running program and the OS incur a performance cost. While these effects are, in many cases, forgivable, they do impose restrictions on the types of applications that can be developed by the system, as the relative computation time *per pixel* remains high. Sitting atop such a high pile of software abstractions, Processing is not particularly well-suited for performance-critical applications, or applications that require a large number of per-pixel calculations (video processing, for example).

Following Processing, a number of similar 2D graphics programming environments emerged, including the Python-based *DrawBot* (later becoming *NodeBox*), as well as *ContextFree*, a generative, grammar-based environment for producing static images. It is through these efforts that creative graphics programming has become accessible to a wide variety of aspiring programmers.

2.3.3 Max/MSP

Max/MSP, a visual programming environment for visual artists and musicians, allows users to create graphics and sound without having to write code. Max uses the the common "boxes and arrows" metaphor for visual programming, where objects with behaviors can be connected, or "wired" to others, establishing a direct communication channel between the two objects. Like in Boxer, these objects can be arranged hierarchially, and can pass messages between one another. However, whereas Boxer was developed in order to help children understand elements of program flow by leveraging its Lisp-like foundation, programmatic introspection is not an

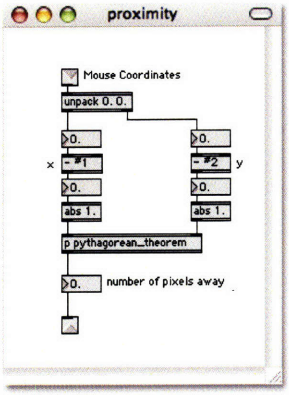


Figure 2-6: A simple Max/MSP patch

integrated component of the Max environment. Max programs (referred to as "patches") are able to be modified at runtime, allowing new objects to be created, and messages sent. This type of flexibility is possible as a result of the Max object abstraction, as opposed to any type of underlying language feature (like the Lisp-like underpinnings of Boxer).

A graphics extension of Max called *Jitter* provides a collection of Max objects for video and 3D graphics that supports more low-level access to the graphics hardware. Jitter allows users to establish relationships between 3D objects without having to understand low-level graphics libraries like OpenGL. In contrast to Processing, Jitter has more direct access to the underlying graphics hardware, and is somewhat more suitable for performance-critical applications. The tradeoff, of course, is in flexibility. Environments like Max that expose elements at a high granularity allow certain bulk operations to be performed efficiently at the expense of fine-grained control. It is for this reason that environments like Processing are better suited for data visualization applications than Max, because of the fine-grained control necessary in the context of these applications.

2.3.4 LiveCoding

Figure 2-7: LiveCoding in Fluxus. New code can be evaluated while viewing the result of the most previous evaluation.

Writing computer code in the *read-eval-print loop* (REPL) style began with John McCarthy's Lisp. By using a data structure that could contain both code and data, Lisp programs could run interactively, allowing a Lisp program to interpret Lisp code[36]. This is in contrast to languages like C/C++ that transform the program text into a static executable. Interpreted languages like Lisp allow evaluation results to be viewed immediately, and data structures to be modified dynamically. These language characteristics have been internalized into communities of artists and programmers interested in creating music and digital artwork in a highly interactive and flexible manner. This community has adopted the term *LiveCoding* to refer to this particular programming style.

The majority of programming languages used by the LiveCoding com-

munity such as *SuperCollider*[15] and *ChucK*[44] are particularly concerned with realtime audio synthesis. Other systems targeted at the generation of live, interactive graphics, have been built on top of Lisp, or languages Lisp-like in nature. *Fluxus*, for example, is an interactive graphics programming environment based on the PLT Scheme Lisp dialect. In Fluxus, the code written interactively by the user is transformed into an OpenGL texture and layered on top of the OpenGL scene where the program output is being displayed. In this way, the programmer can modify the interpreted Scheme code while viewing the realtime results underneath a virtual text editor. Writing computer code in this way allows users to interact directly with the runtime memory image of the interpreter context (control flow, method definitions, variables, etc.), making these Fluxus much more fluid and interactive in comparison to Processing or Max/MSP.

2.4 Creative Content Sharing on the Web

As the WWW evolved from a unique media presentation interface to a platform fostering participation and collaboration surrounding this media, new mechanisms for content aggregation and filtering emerged. Vast collections of images, programs, and video now reside online, with new techniques being created for end users to interact with them.

2.4.1 Web 1.0

BBS systems continued to grow in size and integrate existing internet services (such as email) until the 1990s, following the introduction of Tim Berners-Lee's WorldWideWeb, the first WWW client. WorldWideWeb was initially built with editing capabilities, allowing users to both browse and edit WWW content. The majority of subsequent browsers did not follow this trend. Mosaic, the first mainstream WWW client, achieved its success from many sources, although its innovative approach of embedding images, sound, and video within the page text was found to be an

immensely appealing alternative to users of the WWW browsers of the time.

Accompanying these developments was an almost universally accessible presentation interface for content, enabling a new environment for voyeuristic experiences of creative individuals. This punctuated transition from otherwise isolated and independent modes of creation to one of widespread exposure has influenced creative experiences in a variety of ways. Accompanying this exposure is increased awareness and subsequent competition reminiscent of the artscene.

2.4.2 Web 2.0

Almost as quickly as the WWW emerged as an accessible presentation interface for digital content did Tim O'Reilly coin the term *Web 2.0* to describe a new trend in WWW usage focused on creativity and collaboration. To a large extent, this transition was expedited by the realization of large internet-based companies like *Amazon.com* that the contributions made by customers (in the form of reviews) had the potential to be aggregated to provide other customers with product suggestions, potentially leading to higher sales. In short, these companies began setting "inclusive defaults for aggregating user data and building value as a side-effect of ordinary use of the application", a concept O'Reilly has described as "the architecture of participation".

It has been through this participation model that WWW sites like *Flickr* and *YouTube* have achieved widespread success. Through the large-scale aggregation of end-user content, these sites have provided end users with large collections of *collaboratively filtered* images and video, respectively. These effects have recently extended into creative WWW sites, such as *DeviantART*[4] and the *Rhizome ArtBase*[12]. These sites were built to serve as virtual galleries of user-submitted artwork. While similar in their overall objective, to create environments to present content from a diverse set of artists, they differ with respect to their curatorial model. The ArtBase is a submission-based system that is evaluated by the site curators.

Some pieces are accepted, others are rejected. DeviantART, on the other hand, allows all users to upload content. This content resides within their own virtual portfolio, subject to discussion with other members of the DeviantART community.

2.4.3 Web-based Content Repositories

As our exposure and interactions with WWW browsers become more common, the integration of existing creative tools then becomes a necessary consideration. The organization of the Processing WWW page (processing.org), is representative of the WWW presence of similar environments like Drawbot and ContextFree, which contain user forums, galleries of images, and examples.

Maxobjects[9] is a centralized repository for the submission of user-contributed externals for Max. These objects are compiled by members of the community that submit them directly to the site. While it is entirely possible that these objects could contain malicious code, the majority of the Max community download and run these libraries without hesitation. The relatively small size of the Max community maintains a degree of trust inherent within it.

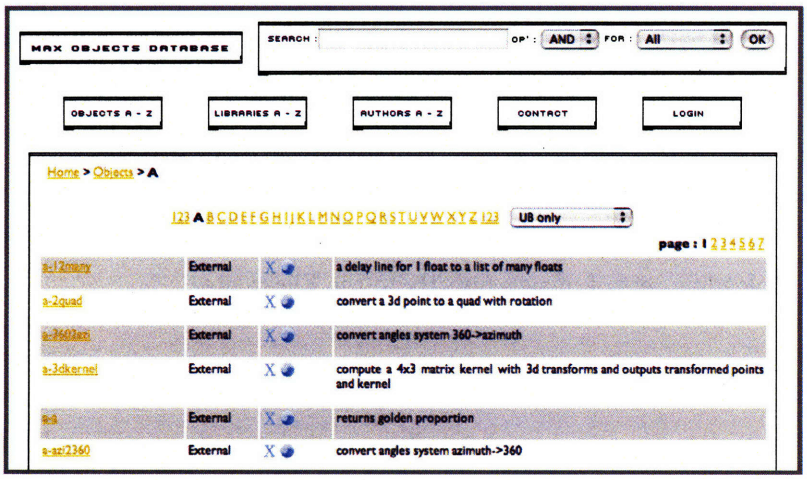


Figure 2-8: **The maxobjects.com WWW site is a large collection of user-submitted Max externals.**

As many Max objects are purely functional and lack a clear visual com-

ponent (users can download objects that implement quicksort, for example), the site is simply a set of links accompanied by short descriptions of the objects (Fig 2-8).

BuiltWithProcessing[1], is a WWW-based collection of links to Processing applets created by other users. Not being an actual database-backed repository of programs makes it difficult for the site to handle the disappearance of linked content. Many of the links found on the site no longer work, as the link destinations have been moved from their original location. Even though the site was created relatively recently (2006), many of the links already suffer from this issue.

2.4.4 Pushing Data To the Web

In the early days of the WWW, site content was mostly static; allowing site visitors to contribute content was not common. It was not until the so-called *participatory web* took hold in early 2000 that sites began to open up their doors, allowing large-scale contributions from end users. Participatory sites like Flickr and YouTube, for example, provide users with two mechanisms for the submission of content. The first resides purely within the context of the browser itself: providing users with a dialog box to specify the file to be uploaded. More recently, however, these services have realized the need to simplify this upload process *outside* of the context of the browser, and now provide lightweight desktop transfer applications like the *Flickr Uploadr* that allow drag-and-drop support for uploading sets of images directly to the Flickr site.

Other WWW sites like the music community *Last.fm*[8], passively collects data about a given user's listening habits from special audio player plugins. This data is uploaded to the Last.fm servers and used to recommend new artists to each user. Other sites, such as *Openstudio* and *ScratchR*[37], have integrated the end-user data submission process into the desktop application itself.

2.4.5 Representational Flexibility for Appropriation

Online, there are WWW-based content repositories for nearly every type of creative media, from images, video, and music, to applications and source code. For the purpose of creative appropriation, (such as the re-use of images or music), some of these media types lend themselves to manipulation better than others. Publicly available source code, for example, obtained through sites like *sourceforge.net*, are rather amenable to end-user modification. Images and video, on the other hand, are less flexible.

The flexibility offered by a human readable text-based format is not limited to source code, however. The *Openstudio* project from the MIT Media Lab, for example, allows users to create drawings through the use of a simple Java drawing application. These drawings are not stored as rendered images, but as an XML variant that allows other members of the Openstudio community to repurpose the work. Distributing content in this fashion is crucial in order to allow subsequent users to change, modify, or re-experience a given piece.

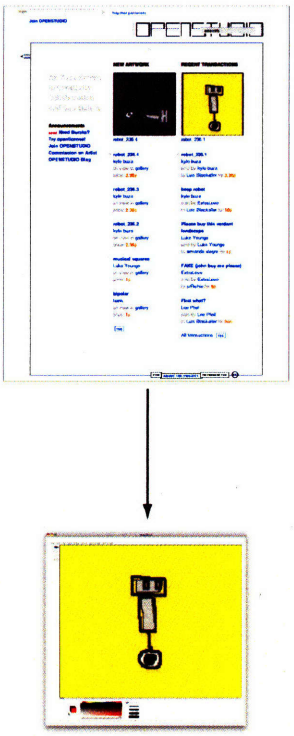


Figure 2-9: The Openstudio drawing application is run as a Java desktop application directly from the browser itself.

2.4.6 Execution From the Web

As a platform, the WWW browser presents a number of challenges to the developer interested in rich media experiences. The underlying reason for many of these challenges is security. Any data obtained from sources on the internet is considered a potential suspect for compromising the client machine. Allowing users to run arbitrary executables directly from the WWW would be a rather large security issue. However, there are cases where the integration of the desktop and the WWW browser makes sense, and can be handled in such a way that security issues are kept at a minimum. *Java Web Start*, for example, is a technology from Sun Microsystems that allows users to run desktop Java applications directly from the browser itself. Understandably, these programs are not downloaded as Java source code, but as compiled Java bytecodes, a

byte-compiled format that also includes metadata (known as *StackMaps*) for use by the runtime verification process to ensure the integrity of the downloaded program. Openstudio, for example, has leveraged this deployment technique for its drawing application (Fig 2-9). This step is necessary because this application performs operations that are viewed as security risks if they were to be run within the context of the browser.

2.5 The Future of the Browser

In this section, I will discuss three ways in which researchers are attempting to change the way we use traditional WWW browsers, relevant in the context of the *E15* environment, which is described in a later section.

1. *Transition to the desktop*: Motivated by both performance and mobility considerations, developers are migrating specific browser functionality to the end user's desktop.
2. *Modification through end-user programming*: Web browsers are beginning to include functionality, allowing users to write custom code to modify and manipulate traditionally *browser-only* data.
3. *The shift to 3D*: While early attempts to leverage 3D for richer browsing experiences have failed, these efforts are resuming.

2.5.1 The Marriage of the Browser and the Desktop

The rate at which WWW browsers are obtaining new features is constantly increasing, and the so-called "browser wars" are far from over. The Mozilla foundation, as well as companies like Adobe and Google, are currently building systems that extend the functionality of the browser onto the end user's desktop. *Google Gears*[6], for example, is a system that allows WWW developers to serialize web data so that WWW applications no longer need to be connected to a network in order to function. *Mozilla Prism*[10] is a system that enables the development of so-

called *Site-Specific Browsers* (SSB), accomplished through the creation of desktop-like applications that have the functionality of a single WWW application. One benefit of this type of architecture is the transition from the WWW browser itself, which runs in its own process, to an environment where each individual application can run in its own process, allowing the OS to effectively manage the concurrent execution of multiple applications. The end result is a collection of otherwise unmodified WWW applications (such as Facebook or Gmail), that are slightly more responsive than using them through a WWW browser.

While accurate browser usage statistics are difficult to obtain, the current WWW browser usage numbers are shared between Mozilla's Firefox, Microsoft's Internet Explorer, and Apple's Safari. Open source browsers like Firefox, along with WebKit, the application framework Apple has leveraged for their proprietary Safari WWW browser, have allowed application developers to bundle an entire browser within custom applications. For example, Linden Lab, the makers of Second Life, have bundled Mozilla's open source Gecko layout engine within their application to re-create the traditional 2D browser experience within their 3D online environment.

2.5.2 End-User Programming

Environments like Processing and Max are not applications in the traditional sense, as neither is used for any one particular type of task. This is in contrast to applications like Adobe's *Photoshop* or Apple's *Final Cut Pro*, whose application domains are relatively well-defined. The importance of extending these types of mainstream applications with programmability features has been recognized as a potentially effective mechanism to empower existing users without forcing them to go shopping for a different application in possession of the desired functionality.

Early research into end-user application programmability began with Michael Eisenberg's SchemePaint[29], a simple drawing application with an integrated Scheme interpreter. Applications augmented in this way are referred to as *programmable design environments* or simply *PDEs*, and

are the integration of programmability into standard design environments (like Adobe's Photoshop). Eisenberg stresses the importance of these environments with respect to their ability to empower end users with more expressive and modifiable application environments. Nardi explains:

> "We have only scratched the surface of what would be possible if end users could freely program their own applications... As has been shown time and again, no matter how much designers and programmers try to anticipate and provide for what users will need, the effort always falls short because it is impossible to know in advance what may be needed .... End users should have the ability to create customizations, extensions, and applications..." [38]

This need for extensibility has been met to some degree by application makers through the use of plugins. Many of the applications produced by Adobe, for example, include plugin systems to allow end users to extend their products. Mozilla's Firefox WWW browser also has a plugin architecture leveraged by many developers to augment and enhance the end-user browsing experience. Two Firefox extensions, *Chickenfoot*[20], and *CoScripter* are Firefox extensions that allow users to write code to automate web browsing and manipulate WWW pages.

Figure 2-10: The *Piclens* browser plugin provides users with a pseudo 3D view of WWW-based image collections

2.5.3 Augmenting the Browsing Experience

Proceeding in lockstep with advances in graphics card performance is a desire by OS and application designers to leverage this potential to achieve a more aesthetically pleasing user interface. Project *Looking Glass*, a Java-based 3D desktop environment by Hideya Kawahara at Sun Microsystems was designed to leverage these developments.

Early attempts at 3D browser-type functionality focused on the Virtual Reality Modeling Language (VRML) and X3D formats to describe 3D objects that could be displayed by a browser plugin. These efforts were un-

successful, and have been replaced by recent attempts at creating 3D environments within which 2D WWW data can be viewed. Also motivating this trend is the fact that OS and application designers are desperately trying to add more features and functionality to an already cramped 2D desktop display.

Figure 2-11: AT&T's *Pogo* 3D WWW browser

Browser plugins like *Piclens* allows users to view sets of images in a pseudo 3D environment without having to traverse the WWW pages containing the images (Fig 2-10). Unfortunately, this plugin requires site designers to provide the Piclens plugin with specific information in the format it requires. *Pogo*, a 3D browser being developed by AT&T is representative of a recent interest in extending the capabilities of the browsing experience beyond the existing 2D experience (Fig 2-11). While visually compelling, Pogo is facing an uphill battle with respect to acquisition of new users and providing marketers a convincing direction for WWW-based advertisements.

These developments are the result of the current trend in leveraging high performance graphics card functionality as well as a desire to provide users with a more visually-compelling WWW browsing experience. Because the construction of effective 3D user interfaces remains difficult, progress in this area is likely to remain slow.

Chapter 3

Experiments

Through the consideration of the WWW as an appropriate medium for the aggregation of DOPE-based creative acts, in this section I will describe two web-based systems that serve to accomplish the following:

1. Aggregate small-scale end-user contributions of code from entirely new or existing DOPEs.
2. Simplify the task of both locating and executing these small-scale code fragments.
3. Provide access to the results of this collective behavior.

I will first describe *OpenCode*, a web-based programming environment that allows users to write graphics programs within the context of the browser itself. By leveraging pervasive browser plugins like Java and Flash, end users can search for, compile, and execute programs without leaving the web browser. Secondly, I will describe a new DOPE, *E15*, for creating procedural animations and visualizations based on web data in a rich 3D environment. Lastly, I will describe *E15:Web*, a WWW site for E15 users that maintains an intimate connection with the E15 desktop application in order to collect and aggregate code snippets written by E15 users.

3.1 OpenCode: Programming on the Web

Today, websites containing user-generated content are increasingly common — more so than at any point in the past decade. Users may upload video to YouTube, images to Flickr, and edit entries in the online encyclopedia Wikipedia as they wish. News organizations are allowing individual users to submit content to their sites, and blogs can be set up in a matter of minutes. Once the data reaches these sites, the capacity for public exposure explodes, and the possibility for creative instigation emerges. Only as web browser developers find new ways to increase the rich media functionality of the browser will content not only be stored on the web, but created there as well. Closing the gap between the browser (the place of *presentation*) and the client desktop (the place of *creation*) has the potential to expose artists' artwork, and inspire others to create work of their own. Content created on the web in this fashion also has the potential to serve as a pedagogical tool as well, through the collection of certain elements of the creative process (the order of strokes in a drawing, for example).

In this section, I will describe the motivation that drove the development of the OpenCode system, how it is used, and a set of minor issues encountered during its development.

3.1.1 Rich Internet Application Programming

Modern web browsers typically come equipped to run two types of rich internet applications (RIAs) namely Sun Microsystems' Java and Adobe's Flash. Development of these types of applications typically takes place on the desktop, either with a set of command line tools or an Integrated Development Environment (IDE). In addition to being a domain for the creation of applications, these development environments are used by artists for purely creative purposes. Traditionally, the sharing and distribution of code from these creative programmers happens through a submission process to a central website where others can view and download sub-

mitted files. However, as is often the case, no effort is made by the program developer to ensure that the submitted code snippet will run on another user's machine. This has a tendency to cause frustration for new programmers, and serves as an unnecessary barrier to entry.

Figure 3-1: **The main OpenCode programming interface, containing an editor window, and collection of links to scripts submitted by other users.**

OpenCode is a web-based programming environment targeted at lowering this barrier. The system was developed in the Summer of 2006 by myself and colleague Takashi Okamoto. In OpenCode, users can look at snippets of code in both Java and the Java Processing dialect submitted by other users, and run these snippets within the context of the browser itself. Users of the site can modify existing pieces of code, and re-compile them to see the changes immediately, without the need to download a specific development environment to their desktop. This fluidity dramatically enhances the experimentation process and willingness of new users to experiment and play with computational art generation programs. OpenCode requires the end user to download no new browser plugins or add-ons. The only requirement of the system is a Java-enabled web browser. Be-

cause the applications are sent to the client browser as Java bytecode, security is not a concern.

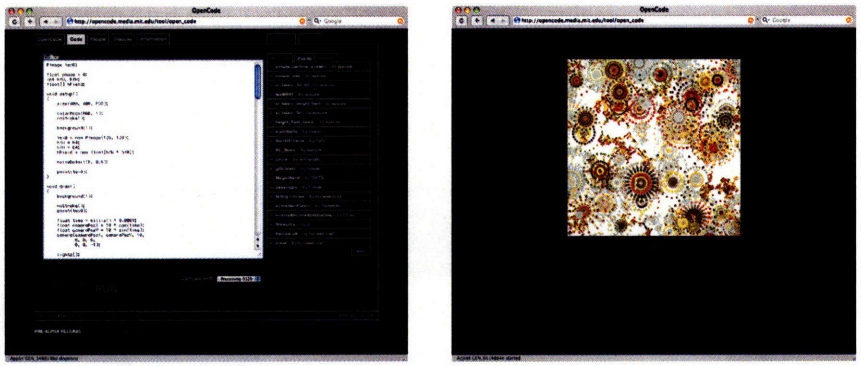


Figure 3-2: **Compiling a program in OpenCode. The program text is submitted to a compilation server, and the compiled applet is displayed in the browser window.**

3.1.2 Functionality

OpenCode is a web site as well as a set of web services for the compilation and download of graphical RIAs. In particular, Java and the Java Processing dialect can be written in the browser, compiled on the server, and run within the user's browser window. This "two click" programming model (click to load an example, click again to run it) allows users to experience graphically-rich applications along with their accompanying source code to simplify the experimentation process.

3.1.3 Implementation

The OpenCode system is a web application built using the *Ruby on Rails* web framework. The back end is composed of a set of Web Services, implemented as Java servlets. Programs written in the browser are sent to a compilation servlet, which executes a modified version of the Processing application to produce Java class files directly, circumventing the existing intimate relationship that Processing possesses with respect to the sketchbook and the source code transformation process. This servlet responds to the client request by sending back the appropriate application bundle (a Java Archive File (.jar)), or relevant error condition to the client

using the *JavaScript Object Notation* (JSON) data format. Because the OpenCode system only allows users to save programs to the server that compile successfully, each program available to visitors is guaranteed to compile and execute.

The Processing project has been in a state of so-called *perpetual beta* since its initial release in 2001. This type of development allows the community of developers to freely change aspects of the system while maintaining a subdued regard for overall system stability or consistency. One issue with this development approach is that new versions may adversely affect or break programs written for previous versions of the software. With OpenCode, we were able to lessen the severity of these changes by allowing users to build their applications against independent versions of the Processing software via a drop-down menu in the browser (Fig 3-3). In this way, multiple versions of the software may exist on the OpenCode site, and users can use more stable versions of the software if more recent versions contain regression bugs. In addition, this feature allows users to compare performance characteristics of each Processing release by simply selecting the appropriate build number from a drop-down list. In fact, we observed specific performance regressions in the Processing application while using this feature.

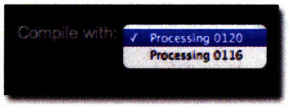


Figure 3-3: OpenCode users can select which version of the the Processing library to use when running their sketch.

3.1.4 User Model

Each registered OpenCode user has a custom profile page (Fig 3-4), which contains links to submitted applets, uploaded resources (such as fonts and images), and custom libraries. As the Java security model does not allow Java applets to access the local filesystem when run in the browser, any static resources a user might require for his program must be uploaded as custom user data, or made accessible through a URL. Through an interest in allowing users to extend the functionality of the system as much as possible, users are allowed to compile applets as well as custom libraries. In other words, users can write custom code representing some set of reusable functionality, compile it on the OpenCode server, and add it to his profile for subsequent use. On compilation and execution of an applet

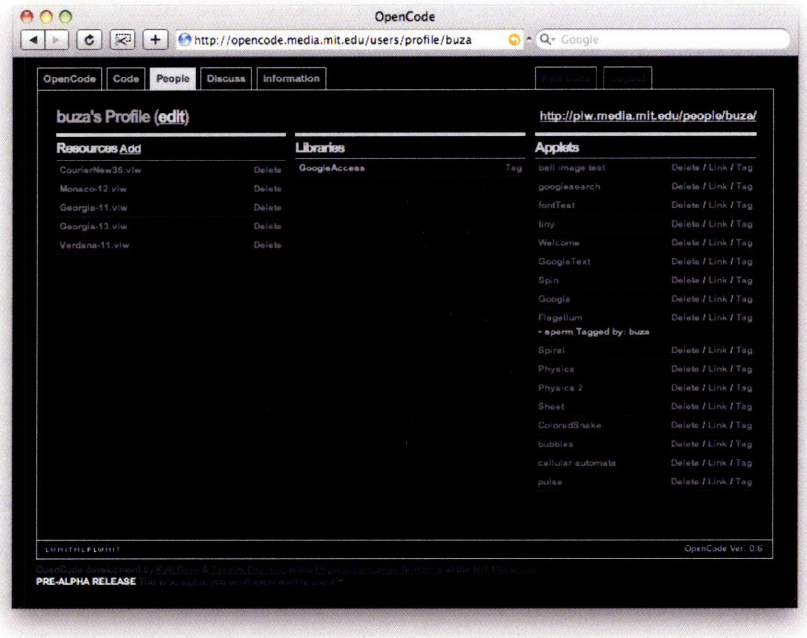


Figure 3-4: **The OpenCode user profile page, listing submitted applets, compiled libraries, and static resources.**

that relies on one of these libraries, the server uses this custom set of libraries as an equivalent of the Java CLASSPATH. If found, these custom classes will be added to the generated bundle that will be sent to the end user on execution of the program.

3.1.5 Issues

During its development, and as we continue to observe the growing community of users on the site, we have come to realize the unfortunate fact that even though OpenCode derives its flexibility from Java-enabled web browsers, the performance and security limitations imposed upon the Java plugin have severely limited the ability of our end users to create rich and compelling motion graphics and visualization experiences. For example, simple Java extensions for Processing (such as the Google API library) require the use of the Java Reflection APIs, which enables runtime object introspection of Java objects. This type of introspection is viewed by the browser JVM plugin as a potential security threat, and its use is prohibited in the context of the web browser. To circumvent this

issue, it was necessary to build custom servlets for each library that requires such functionality. This has limited the scalability of the system by increasing deployment overhead.

3.2 E15: A Web-Enabled Creative Studio

E15 is a system for dynamically constructing interactions within procedurally-generated and web-based data sets in a flexible 3D context. Similar to *Pad++*[18] by Bederson et al., E15 is not as much an *application* as it is an *environment* for creating lightweight 3D applications. While the E15 environment has been architected to allow programmers to easily add new system-level functionality, it is currently distributed as a base implementation that supports the following input types:

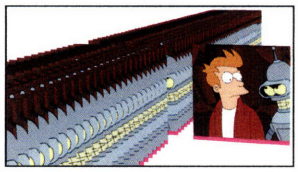


Figure 3-5: Rendering multiple frames of video in E15.

- *Web content*: Users can access web data through existing web-based data APIs, or can take advantage of an embedded web browser to obtain access to rendered WWW pages and a well-formed DOM. This content can be arranged within the 3D context, and procedure-based interactions defined at runtime.
- *Procedural 2D drawings and animations*: An abstraction layer above Apple's Quartz 2D API allows users to create procedural animations and visualizations with a simplified drawing API.
- *Video*: E15 allows users to render standard movie files, video from YouTube, and live video input within the 3D context.

Through the use of an embedded Python interpreter, programming in E15 does not require the conceptual context shift associated with the *compile, run, debug, repeat* programming style. In E15, 2D animations can be run, and all parameters (including class definitions) can be modified dynamically at runtime. In this way, the user can spend more time viewing the result of his program (even if it isn't perfect or complete), as opposed to typing and only seeing the result following compilation. This interactivity, combined with the ability to arrange large collections

of 2D designs or web data, gives the programmer a malleable environment within which an intimate iterative creative design process can be realized. E15 is implemented as an application for Mac OS X 10.5.

In this section, I will describe the overall architecture of the E15 DOPE, the elements that motivated its design, its main feature set, and ways in which end users can augment its existing functionality.

3.2.1 Inspiration

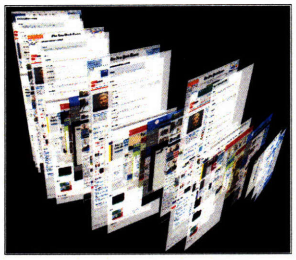


Figure 3-6: E15 was originally driven by a desire to create a flexible 3D environment within which traditionally *browser-only* data could be placed and spatially arranged.

In early 2007, I completed a number of short projects inspired by a desire to find new ways to recontextualize traditionally two-dimensional web content. By forcing users to view web content a single page at a time, modern WWW browsers have made it difficult for end users to leverage their spatial processing abilities. Applications like the Data Mountain[42] and the WebBook[24] have leveraged 3D to arrange sets of web pages spatially, but are relatively rigid in structure. The ability to generate, interact with, and procedurally filter large collections of WWW pages is not supported by these environments. As an early experiment, I wrote a simple domain-specific web crawler that collected full-page screen captures of every accessible page for three separate domains: CNN (*www.cnn.com*), Wired (*www.wired.com*), and Openstudio (*openstudio.media.mit.edu*). In contrast to visiting these pages through the context of a web browser, the ability to visually scan through these sets of unordered pages allowed me to fortuitously find pages of interest that would have been otherwise hidden in the massive link structure of these domains. These experiences led to the development of a more general system for the collection, organization, and layout of web data. This functionality is now an integral component of the E15 architecture as seen in Fig 3-6.

A second motivating factor in the design of E15 was to build an environment wherein the creation of 3D animations was as easy as programming the same animation in 2D. In DOPEs like Processing, the APIs for 3D are essentially wrappers around existing OpenGL commands, which only moderately alleviate the challenge of creating 3D animations. I found

that simply extending existing 2D content into a 3D environment creates a unique aesthetic otherwise difficult to achieve using traditional 3D programming techniques (Fig 3-7). By allowing end users to manipulate procedural 2D content and animations, E15 opens the door to a new realm of expressive possibility without requiring users to understand OpenGL-based 3D programming APIs.

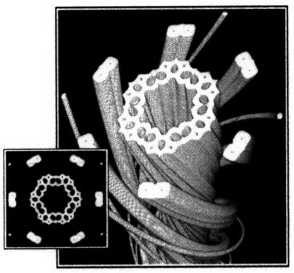


Figure 3-7: Repeating a single 2D animation frame in the 3D E15 context can produce compelling 3D artwork.

3.2.2 Language

Initial prototypes of the E15 system were written entirely in the Objective-C programming language, making it difficult to extend. This issue was resolved through the integration of a Python interpreter along with various architectural abstractions to allow the interpreter to operate in the traditional REPL fashion. Similar to the design of the scripting interface to Pad++[18], only certain elements of the E15 engine are suitable for manipulation through scripting. In E15, scene rendering and image manipulation methods are done in Objective-C for performance reasons, and object creation and manipulation is done via Python.

At the core of the E15 application is an OpenGL context. The embedded Python programming interface simplifies access to this context. Using this interpreter, Python statements can be evaluated, and both classes and method definitions can be dynamically modified. I chose the Python language for the following reasons:

- *Interactivity*: Without a separate compilation step, the so-called "edit-test-debug" cycle characteristic of interpreted languages is fast.
- *Readability*: Python abandons brackets in favor of indentation, and favors the usage of keywords in contrast to punctuation.
- *Maturity*: The Python developer community is large, and highly active. As a result, the interpreter itself is more mature, and currently performs more efficiently than other interpreted languages in its class.

Python is a language commonly used in education[32]. Applications for the One Laptop Per Child (OLPC), for example, are written in Python, and made available for viewing by the end user. In addition to being a valuable language to know in the context of industry, Python's widespread nature makes it easy to find online documentation and examples. This large community of developers is beneficial to creative programmers because of the rate at which additional Python modules are developed for new tasks. For example, developers have recently created Python modules to access a number of common web services such as *PyFacebook*, *Flickr API*, and the *Google data APIs*. Environments like Max and ChucK that rely on compiled or custom languages are more difficult to extend.

Python is a full-featured language that provides users with a rich toolkit for addressing a wide variety of programming tasks. This is in contrast to domain-specific languages like Scratch[13] and Alice[25], that only make certain *types* of programming easier. Alan Kay, in his foreword to *Watch What I Do*[27], a book about end-user programming and programming by example, bemoans the oversimplification of the programming task as a solution to providing end users with PDE-like flexibility:

> "... when we teach children English, it is not our intent to teach them a pidgin language, but to gradually reveal the whole thing: the language that Jefferson and Russell wrote in, and with a few style shifts, the language that Shakespeare wrote in. In other words we want learners of English not just to be able to accomplish simple vocational goals, but to be able to aspire to the full range of expression the language makes possible. In computer terms, the range of aspiration should extend at least to the kinds of applications purchased from professionals. By comparison, systems like HyperCard offer no more than a pidgin version of what is possible on the Macintosh. It doesn't qualify by my standards."

This statement suggests the presence of difficult decisions involved with the design of DOPEs or PDEs that seek to teach as well as enable end-user creativity. The extreme oversimplification of the programming task

may allow users to express themselves in the short term, but if what they have learned through their interactions with simplified interfaces cannot be applied to other application domains, then the simplification loses its pedagogical utility. For the most part, visual programming environments fail in this respect, as the only interaction mechanism is with the visual object abstractions themselves.

3.2.3 Extensibility

As mentioned in Table 2.1, languages like Processing and Max/MSP are able to be extended at the language-level only. Processing, for example, allows users to write external Java libraries to extend the functionality of the environment. In contrast, E15 is flexible at the language-level as well as the platform-level. In other words, users can install new Python libraries written by others and use them within E15 (a language-level extension), or they can directly modify the E15 source to support new content types (platform-level extensions). Developers are likely to want to use platform-level extensions for performance reasons. Platform level extensions are supported in the form of "pixel generators" (PGs), wherein developers write the code necessary to create their own data for display in the 3D context, and send it directly to the E15 OpenGL interface to generate textures. Python module extension templates are provided to simplify the development of PGs, and to allow developers to create Python functions to access this new functionality.

3.2.4 Interface

The E15 interface, shown in Fig 3-8, consists of the following three main components:

1. *3D View*: The content generated and manipulated by the Python script is placed in this context.

Figure 3-8: **The main E15 interface.**

2. *I/O Console*: Both system errors and Python standard streams are routed to this embedded view.

3. *Script Interface*: Python scripts are written and evaluated in this view.

3.2.5 Features

As an OS X application, E15 supports procedural access to the following technologies through the embedded Python interface.

- *WebKit*: A fully-functional web browser. Procedural loading of pages and access to the DOM are supported.
- *Quartz 2D*: Apple's Quartz 2D vector drawing API.
- *Core Text*: Rich text rendering capabilities are supported, including kearning and line break injection.
- *Core Image*: A collection of image filters that can be applied to E15 content.
- *Core Video/Quicktime*: Playback and management of video frames.
- *GLSL-based OpenGL shaders*: Users can write GLSL-based shaders and manipulate shader parameters procedurally within Python.

3.2.6 Browser Integration

Today, when end users interact with web content, they interact with the data in the context of one of three general forms, or *models*[20]:

- *String Model*: The textual representation of the WWW page, often poorly formatted and otherwise difficult to read.

- *Document Object Model (DOM)*: A more structured representation of a WWW page, typically constructed by a web browser. Represented as a tree structure, programmatic traversal of this structure is easier than dealing with raw HTML.

- *Rendered Model*: The representation of a WWW page that users are accustomed to seeing. In the rendered model, DOM elements exist at well defined locations in a 2D region.

When procedurally accessing web data through programming languages like Perl or Python, web content is obtained most often as raw HTML. This form is often malformed and generally difficult to leverage for the extraction of useful data. E15 provides access to web data in each of these forms by leveraging an embedded WebKit-based browser that allows a degree of web page intimacy otherwise restricted to web browser plugins. Through WebKit, E15 has full access to the rendered DOM, and through the Python interface, can evaluate snippets of JavaScript code in the rendered page context. This allows users to inspect and modify the page content similar to the Mozilla Firefox plugin *Greasemonkey*[7]. This allows users to inspect, modify, and extract components of web pages and use them to construct new visual relationships from this data.

Fig 3-9 shows an E15 session wherein a WWW page is loaded in the embedded browser, and an E15 script containing JavaScript code to extract image links is evaluated to produce a set of images to be loaded into the 3D view.

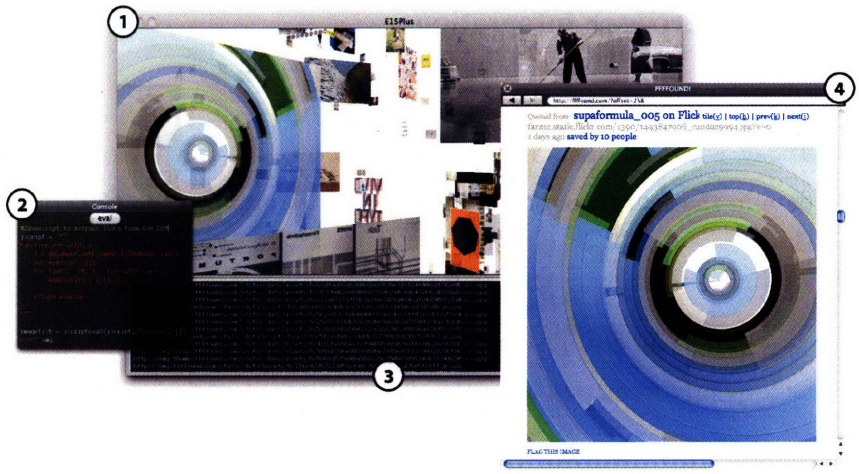


Figure 3-9: **Using the embedded E15 web browser to access the DOM and evaluate snippets of user-defined JavaScript to collect web data.**

The elements of Fig 3-9 are as follows:

1. *3D View*: Contains the result of the collected image URLs, arbitrarily positioned in 3D.

2. *Script Interface*: This window contains the Python script and the accompanying JavaScript to be evaluated in the context of the current WWW page. In this example, the JavaScript to be evaluated simply traverses the DOM and collects URLs for the images referenced on the page:

```
function getimgurls() {
    imgtags = document.getElementsByTagName("img")
    var imgurls = [];
    for (var i = 0; i < imgtags.length; i++)
        imgurls[i] = imgtags[i].src;
    return imgurls;
}
```

3. *I/O Console*: The URLs of the images found on the given page are printed to the console.

4. *Web View*: Contains the loaded WWW page of interest. JavaScript can be evaluated in the context of this page to modify and extract data from the DOM.

3.2.7 Web API Access

Today, a number of websites are making their data accessible to programmers through Web Service APIs, which allow developers to create new applications from existing web data. Flickr, Facebook, and YouTube all offer these APIs to developers, often under the restriction that they not be used for archiving. After all, the data these sites possess is their most valuable resource.

Figure 3-10: **Using the Google Data APIs to search, display, and view YouTube content in less than 100 lines of Python code.**

By using these APIs, E15 users can collect, analyze, and display data collected from popular sites like Facebook (Fig 3-11), YouTube (Fig 3-10), and Flickr. In addition, users may write Python snippets to arrange and perform data-driven actions based on this data. Unfortunately, because these sites want to maintain their IP, only a limited amount of data can be obtained from these APIs.

3.2.8 DOM Access

To allow users to go beyond the limited data available from existing Web Service APIs, E15 provides a mechanism to access the data contained in the rendered page DOM. This gives users the ability to create more rich visualizations by augmenting the data obtained from the APIs with

data obtained from the DOM. For example, Fig 3-11 shows the result of an E15 script that uses the Facebook Web APIs to collect basic information (images and friend relationships), supplemented with data obtained by accessing the rendered page DOM. In this case, message headers are extracted from the DOM as the user browses the Facebook site. This data is used to build a visual history of a user's Facebook messages, which would otherwise be extremely difficult to accomplish with existing browser plugins.

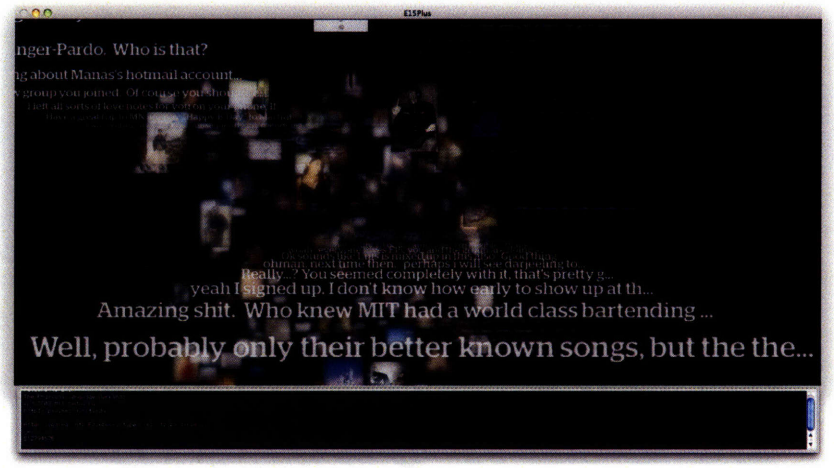


Figure 3-11: **Augmenting the Facebook API-based visualization with inbox data extracted from the page DOM.**

3.3 E15:Web: Collecting the Pieces

Through an extension of the ideas explored through OpenCode, E15:Web is the WWW component of the E15 application. Similar to the OpenCode WWW site, E15:Web was designed in collaboration with colleague Takashi Okamoto. In an age where the social benefits of the WWW are increasingly relevant, E15:Web serves as the web-based back end to existing E15 application functionality. As new users download E15 and leverage it for creative purposes, E15:Web ties together these disparate end-user explorations. As a publicly accessible repository of E15 scripts where examples can be created, modified, and commented upon, E15:Web evolves in lockstep with the growing E15 community. E15:Web maintains an intimate connection with the E15 application, allowing users to load scripts found on the E15:Web site directly into E15. The ability to asyn-

chronously observe the work created by others, and directly modify and execute this work is reminiscent of Hal Abelson and Andrea diSessa's description of the benefits of end-user programmability found in Boxer:

> "One major benefit of programmability is that even professionally produced items become changeable, adaptable, fragmentable, and quotable in ways that present software is not. Not only would professionals be able to construct grand images, but others would be able to reconstruct personalized versions of these same images." [28]

In contrast to sites like Flickr and YouTube, where the potential for reappropriation of the media remains low, E15:Web enables the sharing of *interpreted code snippets*, unfinished by nature, serving as templates for interactive programming *sessions*. By relaxing the notion of *completeness* inherent within compiled languages like C/C++, the sharing of these snippets may allow the barrier to experimentation and contribution to be lowered.

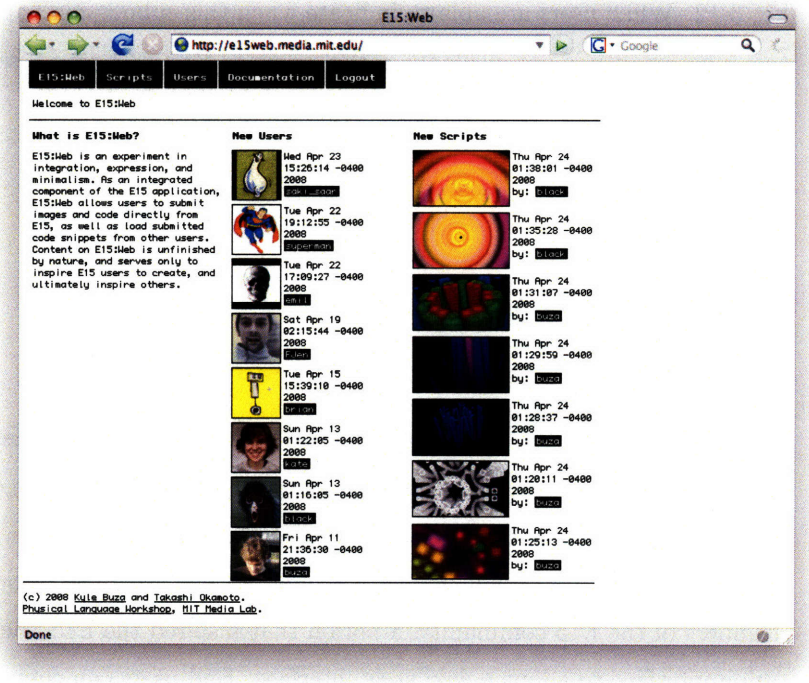


Figure 3-12: **The E15:Web index page, showing recently submitted scripts and active users.**

The E15 application is *stateless* in the sense that no examples are bundled with the application download itself. The entire collection of examples resides on the E15:Web site (Fig 3-12), allowing the community of E15 users to have increased awareness with respect to the activity of the E15 community at large. Providing an interface for exposing these custom experiences via the WWW is important, and the ultimate role of E15:Web.

In this section, I will describe the E15:Web site, and demonstrate how it integrates with the existing E15 application.

3.3.1 Submissions

The central component of E15:Web is an integrated submission system built into the E15 application that sends both E15 screen captures and the script evaluation history to the E15:Web server. The ten most recent submissions are displayed on the main E15:Web WWW page shown in Fig 3-12. While users are able to indicate when this submission happens, this functionality is not available in the standard E15 API set for security reasons.

3.3.2 Documentation

Because of the stateless nature of the E15 application, API documentation is not part of the downloaded application binary. In the spirit of the *perpetual beta* mindset characteristic of many Web 2.0 applications, we anticipate the E15 API will change. Changes should be brought about by members of the E15 community, a notion that inspired the E15 documentation interface to adopt a wiki-style interface. End users can comment upon and directly modify the E15 documentation contained within the E15:Web site as shown in Fig 3-13.

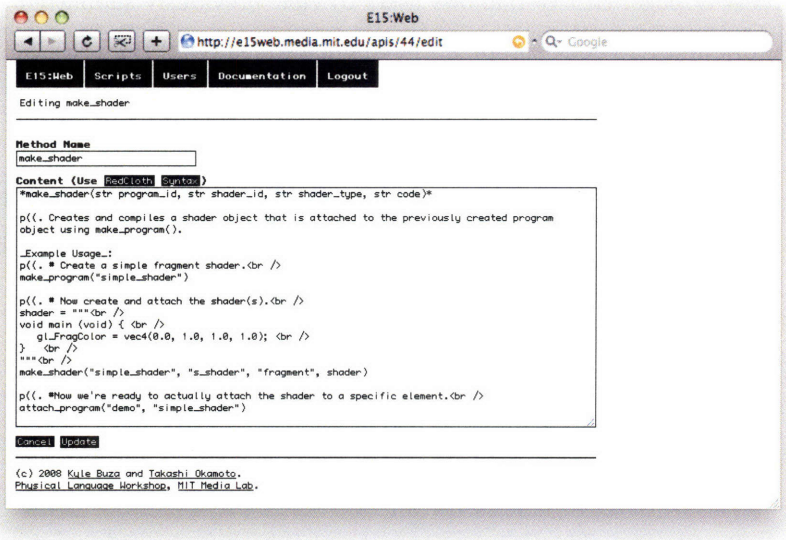


Figure 3-13: **The E15:Web API documentation interface. Each entry can be modified and commented upon by end users.**

3.3.3 Direct Web Execution

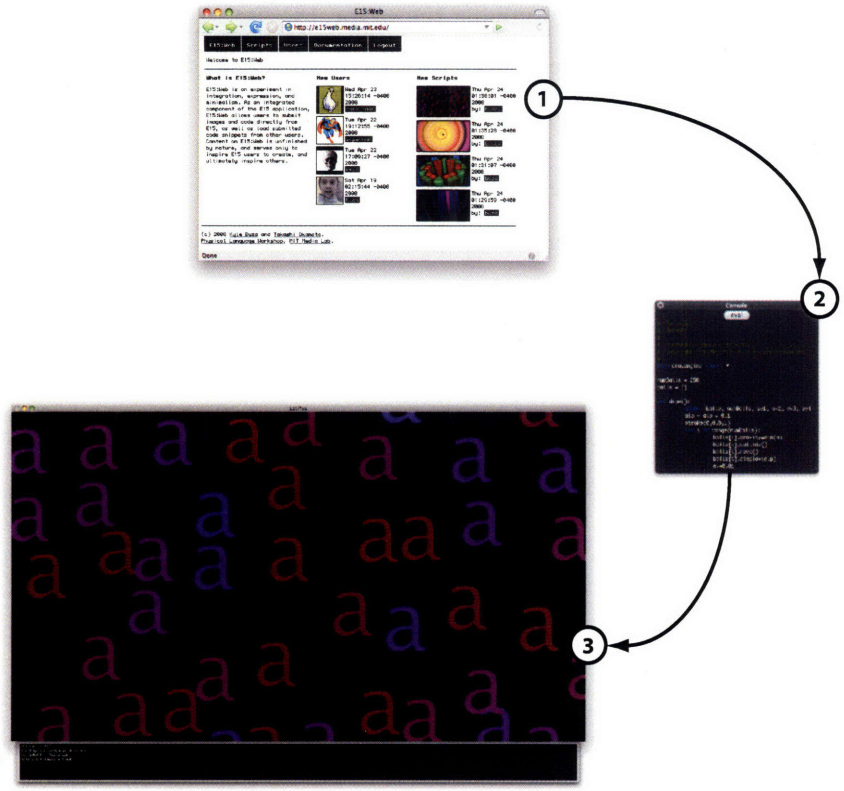


Figure 3-14: **Direct web execution.**

By establishing an intimate relationship between desktop application and

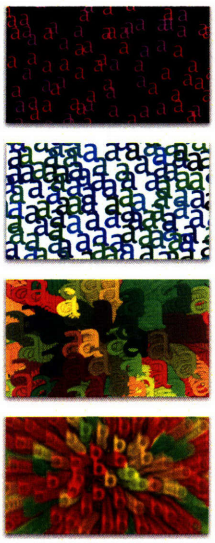


Figure 3-15: Upon loading the script from E15:Web, users may modify script parameters, procedure definitions, and apply filters.

WWW page, E15:Web allows users to view previously submitted artwork and E15 code snippets; they may also load and run these snippets directly from the WWW page itself.

Through the definition of a custom *.e15* MIME type, the embedded E15 browser is able to define custom actions when E15 users click on links to E15 scripts. This process, shown in Fig 3-14, proceeds as follows:

1. *Load*: Clicking on an image in the embedded web browser loads the associated script into the script interface.
2. *Observe*: The E15 user can observe the downloaded script prior to evaluation. While this step is not absolutely required by E15 or E15:Web, it is currently included for security purposes, allowing the user to ensure no malicious code is contained in the downloaded snippet.
3. *Evaluate*: After running the script, the user may then continue to evaluate additional Python code to modify and customize the downloaded animation or visualization.

After the user has run the script obtained from E15:Web, class and method definitions can be changed, filters applied, and parameters modified. In contrast to environments like Processing, these changes are made *during* script execution, as E15 scripts are not compiled. Starting with the original script, Fig 3-15 shows a sequence of modifications a user might make to produce his own visual aesthetic.

3.3.4 Search

E15:Web allows users to perform searches for scripts based on either the API procedures used, or any arbitrary text contained within the script itself. In this way, all scripts that use specific Python libraries or particular sets of procedures may be viewed as a collection in E15:Web. As an example, a user may search for all scripts submitted to the E15:Web site

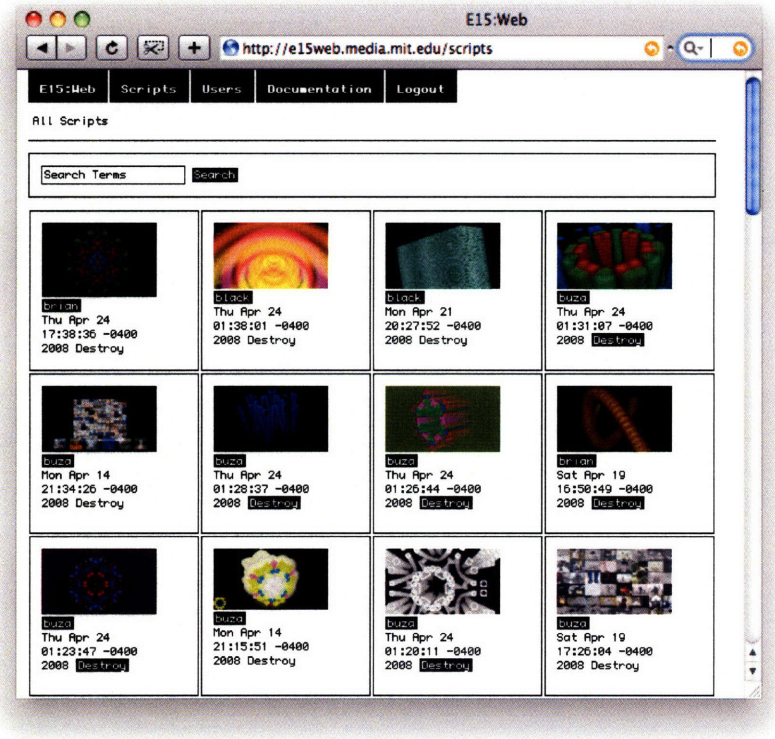


Figure 3-16: **Users may search for scripts that use specific API procedures.**

that use the *rect* procedure and reference *flickr* for accessing data from the Flickr site with the search string "rect + flickr".

The E15:Web search interface is shown in Fig 3-16.

Chapter 4

Evaluation

In this chapter, I will provide the results of a peer review of the OpenCode system, and show how this feedback has been integrated into the E15:Web architecture. I will then provide a common set of three characteristics encountered during the development of these systems.

4.1 OpenCode

Following its release in December of 2006, OpenCode now has a total of 862 registered users and 362 publicly accessible programs as of April 24, 2008. Together with Takashi Okamoto, OpenCode was presented at *Massaging Media 2: Graphic Design Education in the Age of Dynamic Media*, an AIGA conference in April of 2008. The project was received well by educators interested in integrating programming elements into their design curricula.

To evaluate the influence and role of OpenCode, I have conducted a peer review, obtaining feedback from five individuals that were found to both:

1. Have previously created an OpenCode account without external

provocation.

2. Be either an educator that has leveraged Processing in their curricula, or has made significant contributions within the existing Processing community.

To evaluate the success of the application, and to determine which features are lacking, these individuals were asked the following four questions:

1. *Redeeming Aspects*: What do you like about OpenCode?

2. *Negative Aspects*: What do you dislike about OpenCode? (Or, what features would make it more interesting to you?)

3. *Target Demographic*: In its current form, what demographic do you think OpenCode is most suited for?

4. *Other Repositories*: Today, there are a number of web-based repositories for Processing programs (*builtwithprocessing*, *openprocessing*, etc.). Do you actively use these or similar types of repositories?

4.1.1 Redeeming Aspects

In general, the *idea* behind OpenCode is well respected. While various WWW sites have been created previously that allow end users to compile and run arbitrary code chunks within a browser-based sandbox, to the respondents' knowledge, none have been created that allow in-browser compilation of Processing applets.

The ability to both peruse, execute, and modify collections of submitted programs is appealing, as code obtained from other sources on the WWW can be difficult to compile. The centralized OpenCode build structure ensures that all submitted programs are guaranteed to run for all users.

4.1.2 Negative Aspects

Respondents suggest that perhaps too much emphasis has been placed upon the compilation step, versus any sort of traditional elements of community in the collaborative sense. The ability to comment upon, browse, and search for programs submitted to the system are features commonly requested.

Also mentioned is the lack of an easily digestible Flickr-like gallery of images that represent the submitted programs. As static images are difficult to procedurally obtain from time-based media such as the programs on the OpenCode site, they are typically obtained through some form of an end-user submission process. For these reviewers, the ability to visually scan large collections of images generated from different programs is more important than simply looking at the code itself.

One respondent mentioned the disconnect between the code residing on the OpenCode server and the end user's desktop. Maintaining multiple copies of programs is undesirable, and effective ways to synchronize the two collections would be needed.

In comparison to the Processing IDE itself, the majority of respondents noted the lack of tabbing mechanism for maintaining collections of files containing distinct functionality. This allows users to effectively structure their work environment without having to manage a single monolithic file within the browser. In addition, the ability to create end-user libraries was not immediately found to be useful, as the interface to this functionality was unclear.

4.1.3 Target Demographic

The respondents were somewhat split on this subject, with some believing that the system could serve the community of learning programmers by providing them with a continuously growing collection of working ex-

amples. Others believed that, in the absence of end-user tutorials and active forums, the system is not well-suited to students new to the Processing dialect.

4.1.4 Other Repositories

Of the online repositories for Processing programs that exist today, none were used by any of the respondents (including the creator of the *builtwith-processing* site). Instead, sites like Flickr, the video sharing site Vimeo[16], and social bookmarking site del.ici.ous[2] were referenced as alternatives. Existing WWW-based collections of applets like *builtwithprocessing* were criticized as containing excessive numbers of broken links.

4.1.5 Additional Comments

One respondent made the following comment with respect to exposing his work through the use of video and imagery in contrast to making the source code itself accessible:

> "Perhaps it is due to comfort with those older media formats that archiving work using them feels more natural. Perhaps it's also the fact that I want to keep working with my code, and putting it on the web for others to see feels like freezing it in a particular state. By defining artifacts as images rather than code, I feel a little more liberated to continue work on my desktop."

In the history of interactive graphics programming, the vast majority of applications have been written in compiled languages like C/C++ and Java, limiting end-user interactivity. In these applications, the interaction flexibility must be written by the program developer, as the end user is not allowed to modify the program behavior without recompilation. For example, parameter-changing sliders or file-choosing mechanisms must

be implemented up front by the application designer. In an age of fast computers and a newfound appreciation of strictly interpreted languages that can run in the REPL style, end-user expectations of interactivity are likely to change. Allowing the end user to directly modify the internal object state and class structure of an interpreted program at runtime, they become interactive *sessions*, wherein notions of application *completeness* begin to fade away.

4.1.6 Discussion

One interesting result that arose from this evaluation is the heavy reliance upon image and video-based WWW repositories for inspiration, as opposed to code-based repositories. Because these respondents were already comfortable with the programming process itself, the *how* of a particular design or aesthetic appears to become less interesting than *what*. As anticipated, the notion of *completeness* associated with written programs is a concern with many users. They appear to be comfortable with revealing their work as imagery or video as opposed to the code itself.

4.2 E15:Web

E15:Web was developed in part as a response to some of the problems inherent with the OpenCode system. Initially, E15:Web was designed to alleviate the following internally-perceived issues with OpenCode:

1. *Performance*: Running interactive graphics programs within the context of a WWW browser is slow in comparison to the end user's desktop.

2. *Resource Management*: Managing external resources (images, movie files, etc.) referenced from user programs is difficult when constrained to the browser.

The end result was a simple web-based submission system for images and code. Through the internalization of the feedback obtained from OpenCode, the following additional features were incorporated into the E15:Web site:

1. *Code Search*: As the collection of submitted scripts grows, the need for mechanisms to perform keyword searches within the set becomes necessary. Users may search for specific strings within the archive of submitted scripts.
2. *Image Galleries*: The desire to have easily digestible representations of the work in the form of static images or video is common. As an application that runs on the end user's desktop, image generation and submission is much easier than in OpenCode.
3. *Tutorials*: In order to position E15:Web as a tool that can be used for pedagogical purposes as well as creative exploration for more advanced users, it must contain an evolving collection of examples and instructions on new features and techniques. A wiki-style format allows users to create, modify, and comment upon examples.

4.2.1 Data Retention

The initial design of E15:Web allowed users to delete submissions if they so desired, permanently removing them from the E15:Web database. However, it became apparent through user feedback that code versioning and access to structured repositories of personal work are important. As a result, a decision was made to provide users with the ability to hide submitted scripts from public view, retaining privacy as well as versioning history.

4.2.2 End User Privacy

As with any end-user application that sends information to the WWW, privacy must be maintained. Even though the majority of E15 scripts

do not contain particularly sensitive personal data, this is not always the case. During the development of E15:Web, submissions were made by certain users that contained personal data in the form of personal web API keys. In particular, the Flickr API was being used to access and modify the users' Flickr image collection. Using these APIs requires the end user to create personal API keys for both authentication and access control. Upon recognizing this issue, efforts were undertaken to provide a general mechanism for allowing users to protect private data. Prior to submission to E15:Web, these privately tagged pieces of data must be replaced with their sanitized equivalents. The solution to this problem was designed to proceed as follows:

```
import flickrapi

api_key = '19817caf4f1eed49d9649bf4346dd0bf'
api_secret = '9382b9882c0ce9e8'
```

In this case, the API keys are private, and should not be submitted to the E15:Web server. E15 supports the following C-style comment syntax for protecting private values:

```
import flickrapi

api_key = '/*19817caf4f1eed49d9649bf4346dd0bf*/'
api_secret = '/*9382b9882c0ce9e8*/'
```

These specifiers tell the E15 runtime to replace the content between them with a special value that is not retained in the script that will be submitted to E15:Web. The script that will be submitted to the site is as follows:

```
import flickrapi

api_key = 'SECRET'
api_secret = 'SECRET'
```

Even if the user forgets these specifiers, E15:Web was augmented to contain a "Hide" button for each submitted script, allowing users prevent

their script from being accessible to the general public.

4.2.3 Code Management

E15 programs, as lightweight code snippets, do not need a profound code management architecture. However, as creative programmers have come to expect this type of functionality, it is likely that there may be commonly referenced script snippets that users use often. One example of this functionality is the following Python script that can be appended to any E15 script that will submit saved images not only to E15:Web, but to the Flickr WWW site as well.

```
import flickrapi

def onSubmit(e):
  api_key = '19817caf4f1eed49d9649bf4346dd0bf'
  api_secret = '9382b9882c0ce9e8'
  flickr = flickrapi.FlickrAPI(api_key, api_secret)
  (token, frob) = flickr.getTokenPartOne(perms='write')
  flickr.getTokenPartTwo((token, frob))
  flickr.upload(filename=e, description='Auto-generated submission from http://e15web.media.mit.edu.
```

In order to manage these types of small, reusable chunks, an API procedure was added to enable an E15 include mechanism as follows:

```
includescript("http//e15web.media.mit.edu/users/buza/flickrscript.py")
```

These scripts are contained within the user's profile on the E15:Web site, with access control being managed by the existing browser session. In other words, these scripts are kept private — only accessible to the currently logged in user.

4.2.4 Discussion

As of April 2, 2008, E15 and E15:Web are in early alpha stages of development. E15 application binaries were distributed to interested testers,

who were allowed to create E15:Web accounts. Early debugging and evaluation sessions have proven difficult. Currently, evaluation has proceeded on an individual basis, through personalized one-on-one sessions. In its current form, users are currently spending more time getting up to speed with the environment itself, as opposed to being active contributing members of the E15:Web community.

4.2.5 Extensibility

As mentioned in Section 3.2.3, the E15 platform allows developers to augment its functionality through the creation of "pixel generators", embedded within the E15 runtime. To demonstrate this functionality, I ported an Atari 2600 emulator into E15 as a pixel generator (Fig 4-1). This effort took less than two days to implement.

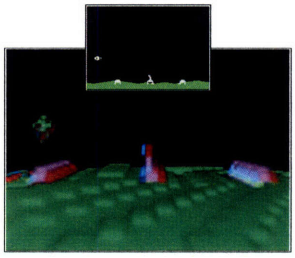


Figure 4-1: *E15:Atari2600*, an embedded Atari 2600 console in the 3D E15 context. Subsequent frames can be arranged to provide a sense of depth during gameplay

4.2.6 Security

Because E15:Web allows users to execute arbitrary Python scripts, security is an important concern. For example, a malicious user could easily embed a small Python snippet like the following into their E15 script to cause significant harm to an unsuspecting end user:

```
import os

for file in os.listdir(os.sep):
  os.remove(file)
```

While this is a concern, Java code exhibiting similar functionality can just as easily be embedded within a program written in the Processing dialect of Java. Sites like *maxobjects.com* distribute sets of Max objects: compiled binaries that are executed when added to a Max patch. Members of the Max community trust the content contained on this site, and are not concerned about potential security issues. In a similar fashion,

E15:Web creates a security microcosm like maxobjects.com and Wikipedia that users can trust because of its self-governing nature.

4.3 Challenges

During the development of both OpenCode and E15:Web, the following issues and challenges were encountered and addressed:

- *Trust*: As WWW-based systems focused on the aggregation of end-user content, these systems needed to respect end-user privacy and minimize security risk.
- *Reusability*: As current DOPE users have come to expect specific code organization functionality such as the use of tabs in the Processing DOPE, building mechanisms supporting libraries or other organizational techniques was required.
- *Cumulativity*: The organization and aggregation of end-user contributions required additional infrastructure on the server back end.

Chapter 5

Conclusion

The goal of this thesis was to formulate a coherent set of characteristics that creative programming environments and their associated WWW sites must possess to help improve, inspire, and support the work of creative individuals using these systems. In order to accomplish this goal, two WWW-based systems and a custom DOPE were created:

- *OpenCode*: A web-based programming environment for developing development graphically rich RIAs.

- *E15*: An environment for dynamically constructing 3D interactions within procedurally-generated and web-based data sets.

- *E15:Web*: A WWW site and management system for E15 images, code snippets, and code management.

In the context of this thesis, I refer to OpenCode and E15:Web as *architectures for web-based collectivity.*

5.1 OpenCode

OpenCode was built in an attempt to streamline the experimentation process with respect to writing code within the context of creative programming environments. By embedding the development environment within the browser itself, end users are no longer required to download and install a full desktop application before initiating development or example-based experimentation. While the majority of respondents found the in-browser compilation feature useful for simple experimentation, many users expect a more rich set of community and collaboration features from today's web applications. In addition, respondents appear to find more utility and inspiration from image and video-based WWW repositories like Flickr and Vimeo than they do from the ability to view program source code. Users also expressed an element of discomfort with respect to distributing the source code of their work online as they did not view them as finished or complete.

From this feedback, the following were isolated as important components to be considered in future versions of the system:

- *Image Galleries*: With an accessible web-based collection of images, users can obtain a general view of available content without having to run each program individually.
- *Incompleteness*: The use of compiled languages like Processing makes users feel the need to finish their code before placing it on the WWW.

5.2 E15:Web

The lessons learned from OpenCode were integrated into initial versions of E15:Web, which has continued to evolve over the course of the past three months. In particular, E15:Web was built around a submission interface within the E15 application that sends both images and code snip-

pets to the E15:Web server. On the E15:Web page itself, these submissions are viewed in a gallery-style format, similar to existing sites like Flickr. In contrast to systems like OpenCode, which present challenges with respect to the generation and collection of static images, image generation and submission is integrated into the E15 runtime itself. Both the E15 application and E15:Web are early in the development process, and evaluation of the E15:Web site has proceeded through individual one-on-one debugging sessions and direct email communication.

As mentioned in Section 4.3, three technical challenges were encountered during the development of OpenCode and E15:Web — namely, *trust*, *reusability*, and *cumulativity*:

- *Trust*: Ensuring the content obtained from these systems is not malicious, and the data sent to these systems respects end-user privacy.
- *Reusability*: Ensuring the content obtained from these systems remains flexible and amenable to creative appropriation.
- *Cumulativity*: Creating an environment within which user created content can be collected and arranged.

5.3 Future Work

5.3.1 Granularity of Sharing

As E15:Web is still in early stages of development, it remains to be seen whether or not it's able to inspire a new community of creative individuals. It's possible that users are just as apprehensive about sharing short, reusable code snippets as they are sharing source code to clean, polished applications. If this is indeed the case, additional research must be conducted to determine precisely how much of the creative process users are willing to expose on the WWW in the form of source code.

5.3.2 Realtime Video Collection

Modern GPUs afford reasonably efficient runtime video capturing techniques from OpenGL-based graphics applications. A natural progression of the E15 architecture would be to integrate a realtime video generation mechanism in addition to the current image generation technique. Through this functionality, direct sharing of video content can be supported.

5.4 Final Thoughts

We are currently observing a transition from isolated, creative acts to ones that serve to both teach and inspire others. As we begin to understand the salient components of the WWW that are relevant to this transition, new WWW/desktop application hybrids will begin to emerge, fully integrated with the ability to share and expose aspects the creative process to the community at large. Only after these applications are developed will creators and educators be able to take a step back and see the expressive potential offered by digital creative tools.

Bibliography

[1] Built with processing www page. Website. http://builtwithprocessing.org/.

[2] del.icio.us www page. Website. http://del.icio.us/.

[3] Design by numbers www page. Website. http://dbn.media.mit.edu/.

[4] Deviantart www page. Website. http://www.deviantart.com/.

[5] Flickr www page. Website. http://www.flickr.com/.

[6] Google gears www page. Website. http://gears.google.com/.

[7] Greasemonkey www page. Website. http://www.greasespot.net/.

[8] Last.fm www page. Website. http://www.last.fm/.

[9] Maxobjects www page. Website. http://maxobjects.com/.

[10] Mozilla prism www page. Website. http://wiki.mozilla.org/Prism.

[11] Processing www page. Website. http://www.processing.org/.

[12] Rhizome artbase www page. Website. http://rhizome.org/art/.

[13] Scratch www page. Website. http://scratch.mit.edu/.

[14] Secondlife www page. Website. http://www.secondlife.com/.

[15] Supercollider www page. Website. http://www.audiosynth.com/.

[16] Vimeo www page. Website. http://www.vimeo.com/.

[17] Youtube www page. Website. http://www.youtube.com/.

[18] Benjamin B. Bederson, James D. Hollan, Ken Perlin, Jonathan Meyer, David Bacon, and George W. Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. *Journal of Visual Languages and Computing*, 7(1):3–32, 1996.

[19] Laszlo Belady. Large software systems. Technical report, Yorktown Heights, NY, January 1978.

[20] Michael Bolin. *End-User Programming for the Web.* PhD thesis, Cambridge, MA, USA, 2005.

[21] Amy Bruckman. Cyberspace is not disneyland: The role of the artist in a networked world, 1995.

[22] Vannevar Bush. As We May Think. *The Atlantic Monthly*, 176(1):101–108, jul 1945.

[23] Donald Campbell. Ethnocentrism of disciplines and the fishscale model of omniscience. *Interdisciplinary relationships in the social sciences*, pages 328–348, 1969.

[24] Stuart K. Card, George G. Robertson, and William York. The webbook and the web forager: video use scenarios for a world-wide web information workspace. In *CHI '96: Conference companion on Human factors in computing systems*, pages 416–417, New York, NY, USA, 1996. ACM.

[25] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3d system for novices. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493, New York, NY, USA, 2000. ACM Press.

[26] Mihaly Csikszentmihalyi. *Creativity : Flow and the Psychology of Discovery and Invention.* Perennial, June 1997.

[27] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration.* MIT Press, Cambridge, MA, USA, 1993.

[28] A. A diSessa and H. Abelson. Boxer: a reconstructible computational medium. *Commun. ACM*, 29(9):859–868, 1986.

[29] Michael Eisenberg. Programmable applications: Interpreter meets interface. Technical Report AIM-1325, 1991.

[30] Douglas C. Engelbart. Toward augmenting the human intellect and boosting our collective iq. *Commun. ACM*, 38(8):30–33, 1995.

[31] Dave Green. Demo or die. *Wired*, page 142, 1995.

[32] Mark Guzdial and Andrea Forte. Design process for a non-majors computing course. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 361–365, New York, NY, USA, 2005. ACM.

[33] R. Nicholas Jackiw and William F. Finzer. The geometer's sketchpad: programming by geometry. pages 293–307, 1993.

[34] John Maeda. *Design by Numbers.* MIT Press, Cambridge, MA, USA, 1999. Foreword By-Paola Antonelli.

[35] Richard E. Mayer, Jennifer L. Dyck, and William Vilberg. Learning to program and learning to think: what's the connection? *Commun. ACM*, 29(7):605–610, 1986.

[36] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[37] Andrés Monroy-Hernández. Scratchr: sharing user-generated programmable media. In *IDC '07: Proceedings of the 6th international conference on Interaction design and children*, pages 167–168, New York, NY, USA, 2007. ACM.

[38] Bonnie A. Nardi. *A small matter of programming: perspectives on end user computing.* MIT Press, Cambridge, MA, USA, 1993.

[39] Takashi Okamoto and Kyle Buza. Visualization of Social Interactions in Facebook. In *Proceedings of the International Conference on Weblogs and Social Media.* AAAI, 2008.

[40] J Rambusch, T Susi, and T Ziemke. Artefacts as mediators of distributed social cognition: A case study. pages 1113–1118, 2004.

[41] Mitchel Resnick and Natalie Rusk. The computer clubhouse: helping youth develop fluency with new media. In *ICLS '96: Proceedings of the 1996 international conference on Learning sciences*, pages 285–291. International Society of the Learning Sciences, 1996.

[42] George Robertson, Mary Czerwinski, Kevin Larson, Daniel C. Robbins, David Thiel, and Maarten van Dantzich. Data mountain: using spatial memory for document management. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 153–162, New York, NY, USA, 1998. ACM.

[43] Ivan E. Sutherland. Sketch pad a man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 6.329–6.346, New York, NY, USA, 1964. ACM.

[44] Ge Wang and Perry Cook. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 812–815, New York, NY, USA, 2004. ACM.