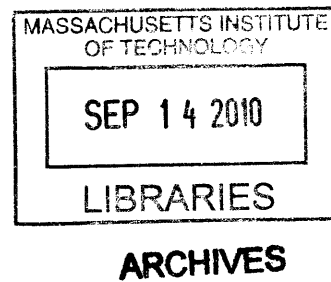


Kaleido: Individualistic Visual Interfaces for Software Development Environments

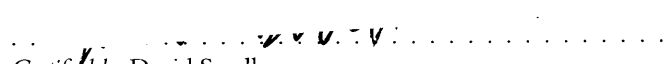
Agnes Chang
B.A. Media Arts and Sciences, Japanese Language and Literature
Wellesley College
May 2007


Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences at the
Massachusetts Institute of Technology
September 2010



© Massachusetts Institute of Technology, 2010. All rights reserved.


.....
Author Agnes Chang
Program in Media Arts and Sciences


.....
Certified by David Small
Assistant Professor of Media Arts and Sciences
Thesis Advisor


.....
Accepted by Pattie Maes
Associate Academic Head
Program in Media Arts and Sciences

Kaleido: Individualistic Visual Interfaces for Software Development Environments

Agnes Chang
B.A. Media Arts and Sciences, Japanese Language and Literature
Wellesley College
May 2007

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences at the
Massachusetts Institute of Technology
September 2010

Abstract

Programming, especially programming in the context of art and design, is a process of reconciling and shifting between individual creative thought and rigid conceptual models of code. Despite advances of programming support tools, the discrepancy between the contextual specificity of the author's intent and the uniformity of program structure still causes people to find the software medium unwieldy. Taking inspiration from the way in which sketching supports the creative process, in this thesis I argue that incorporating individualistic visual elements into the interface of our programming environments can make the creative coding process more intuitive.


I present Kaleido as one implementation of a programming environment that augments traditional textual representations of a program with user-generated graphical elements that act as an additional interface to the code. Kaleido enables users to create personally meaningful visuals for their code, thus allowing individuals to plan, organize, and navigate code in the idiosyncratic way we each think. This document presents the motivations, research, and design process that led to the creation of Kaleido, as well as a preliminary evaluation of a number of users' experience with using Kaleido, and finally a discussion of future and alternative possibilities.

Thesis Advisor:
David Small
Assistant Professor of Media Arts and Sciences

Kaleido: Individualistic Visual Interfaces for Software Development Environments

Agnes Chang

Thesis Reader



.....
Mitchel Resnick
LEGO Papert Professor of Learning Research
Lifelong Kindergarten Group
MIT Media Laboratory

Kaleido: Individualistic Visual Interfaces for Software Development Environments

Agnes Chang

Thesis Reader

Casey Reas
Professor
Design | Media Arts
University of California, Los Angeles

For my teachers

Acknowledgements

It takes a very special place and some very special people to foster genuine interdisciplinary work. One small page of acknowledgements does little justice to the many people who contributed to the web of forces that produced this work. I am merely a person who happened to organize and document the interconnected gossamer of ideas developed by people before me and around me. At those moments when I felt utter defeat, these people urged me on by renewing my confidence and by never wavering in their high expectations. In the end, really, this work is theirs, and not mine.

Dave, for the beginning of everything,
Richard and Jeff, without whom none of this work would ever have begun, or continued, or reached anywhere,
Mitch, for setting the standard, and for always believing,
Casey, for his dedication to students, even those who aren't his students,
Info Eco, for all that is the wacky, hacky, and happy big Eco family,
EricR, for setting off the spark on fine Friday at 4pm at an ML Tea,
Sajid, although many after me will claim to be his advisee, I'm going to be the first one to actually do so,
Nadya, for the beers and hugs that came to the rescue innumerable times and thankfully still do,
Susie, for being the very best UROP,
Friends at the lab, Santiago Alfaro, Doug Fritz, Wu-hsi Li, Amon Millner, Andy Cavatorta, Elly Jessop, Noah Feehan, Karen Ann Brennan, Taemie Kim, Pranav Mistry, the little band of folk that comprise taiwan@media, and many others, for the daily camaraderie, discussion, and inspiration,
Eclipse Foundation, for producing a tool that saved me thousands of hours despite my spending thousands of hours with it already,
Santiago Alfaro, Jonathan Bobrow, Andy Cavatorta, Keywon Chung, Timothy Gardner, Mary Huang, Taemie Kim, Wu-hsi Li, Kyle McDonald, Lia Napolitano, Jeff Orkin, Keith Pasko, Peter Torpey, for participating in the studies,
Eugene Wu, for being the goofball fellow thesiser at Tosci's,
Human Dynamics, Sandy, Taemie, Ben, and Daniel, and Sociable Media, Drew, Orkan, and Dietmar, for sowing the seedlings of questions in my head early on,
My roommates, Stewie, Noah, and John, for keeping me balanced,
My family, for never asking and never knowing but always being there,
sMITe, for making me a better and stronger person.

A Colophon (Of Sorts)

Omitting the napkins and pieces of scrap paper on which thoughts were first put into words, the original source of this document was created in the form of numerous fragmented plain text files. Preliminary formatting, editing, and management of references were completed in a LaTeX draft. Illustration management, layout, and final edits was completed in Adobe InDesign CS4.

The fonts used in this document are, for body text, Bembo, revived by Stanley Morison in 1929 based on the design of Francesco Griffo in 1495; for image captions, Whitney, designed by Tobias Frere-Jones in 2004; and Glypha, designed by Adrian Frutiger in 1977, for titles and subtitles.

In this document, there are 145 pages, 23,478 words, 218 illustrations, and 146,587 characters, including this sentence.

The creation of this document was fueled in part by mochas from Toscanini's Ice Cream in Cambridge, Massachusetts and lattes from Café Grumpy in Brooklyn, New York.

Finally, every large undertaking that is accomplished while listening to music will forever be associated with that music. Last.fm tells me that most of this work was done while listening to the soundtracks by Joe Hisaishi, Masuda Toshio (for *Mushishi*), and Yann Tiersen; and to Anathallo, The XX, Explosions in the Sky, Max Richter, Kim Hiorthøy, Loreena McKennitt, and Iron & Wine.

Contents

Abstract	3
Acknowledgements	11
1 Introduction	17
2 Context	29
3 Design	43
4 Investigation	55
5 Implementation	69
6 Evaluation	79
7 Conclusion	95
References	105
Appendices	111

How frequently must users think about how it works in order to make it work?

Andrea diSessa

Introduction

In this first chapter I explain the motivations behind this work, define a common vocabulary for discussion, and outline the goals and scope of this thesis. I demonstrate that programming is still unintuitive for artist-coders due to the “mental gap” that exists between individuals’ creative intentions and software structures. I present my arguments for an interface that supports the simultaneous and integrated creation of sketches and the creation of code to bridge the mental gap. I then explain my intuitions regarding such an interface’s potential to assist in the programming processes of ideation, navigation, documentation, and learning. Finally, I conclude with a summary of this thesis presented in the form of an overview of this document.

Foundations

... Noticing that the ball motion is slightly skewed she stops the output to review her code. Recalling that she programmed it after she wrote the painting code, she searches her painting file for the code fragment that governs the motion. Yet she does not find it there. Perhaps, because of the way the Java language structures animations, it is in the main loop instead... Certainly, in her mind the ball is a part of the wall, but because the ball has motion, the code for it is not located with the rest of the architectural elements, either... As she repeatedly scrolls through her many code files for this project, she remembers that she had recently reorganized her code to handle programmatic events. She reads through that file, and eventually she locates the code responsible, but by now she has forgotten what change she originally wanted to see in the motion.

Programming has evolved a great deal over the years to bring the power of computation to non-programmers. Yet, software remains an unintuitive medium for the creative work of artists, designers, and hobbyists. The true difficulty lies in a discrep-

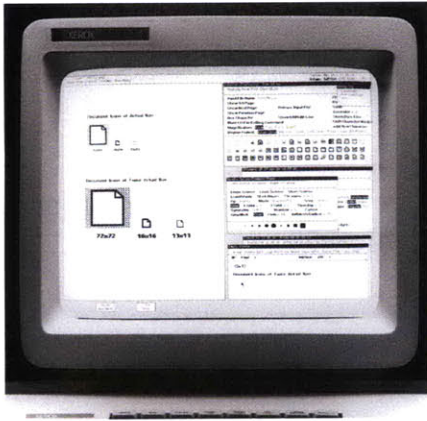


FIGURE 1-1 Xerox Star (1981), the successor of Xerox Alto, had one of the first graphics programs for non-programmers.

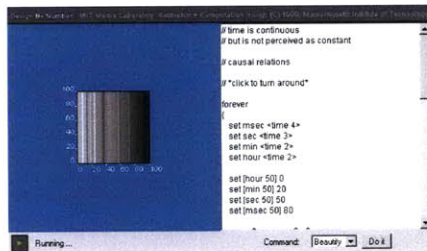


FIGURE 1-2 The interface of Design By Numbers (1999)[41], a platform designed for artists and designers to learn programming..

any between a person’s intentions and the software constructs that will produce the desired results. Meanwhile, creative people regularly employ visual sketches to externalize their thoughts and to develop their intentions. If sketches could be developed digitally side-by-side with code, could we facilitate the creative process by integrating people’s sketches as individualistic visual interfaces in our software development environments?

This thesis is an attempt to implement and evaluate some preliminary intuitions regarding the design of individualistic visual interfaces, with the primary aim of presenting a set of guidelines for the design of such interfaces and a starting point for further development to support creative coding.

The Creative Coding Process

Software for designers shipped with the earliest desktop machines such as the Xerox Alto (1973). Such graphics programs were WSIWYG editors that revolutionized many fields of design; however, the true powers of computation remained the exclusive realm of programmers. In the 1990s, John Maeda at the MIT Media Lab led an effort to develop Design By Numbers[41](Figure 1-2), a platform designed especially for designers and artists to learn to program in a general purpose language. In 1999, the popular rich-media environment Adobe Flash acquired scripting support with the advent of Flash 4 (then developed by Macromedia)[61]. Through efforts such as these, the interactive, dynamic, and generative powers of computational code were made available to the artist who did not have formal programming training.

Since that time, the activity of programming began to diversify from the industry paradigms of programming in large teams of software engineers. Today, a variety of creative code development environments allow individuals programming on personal machines to produce digital media and to choose software as their creative medium.

The process of designing with digital tools, just as with any other design material, has to do with the joint achievements of “doing” and “thinking”[43]. Certainly, the computer as a medium requires a new set of creative skills, especially regarding the handling of complex symbolic abstractions[ibid.]; toward this end, programming environments have developed a variety of programming paradigms and tools to assist the non-programmer.

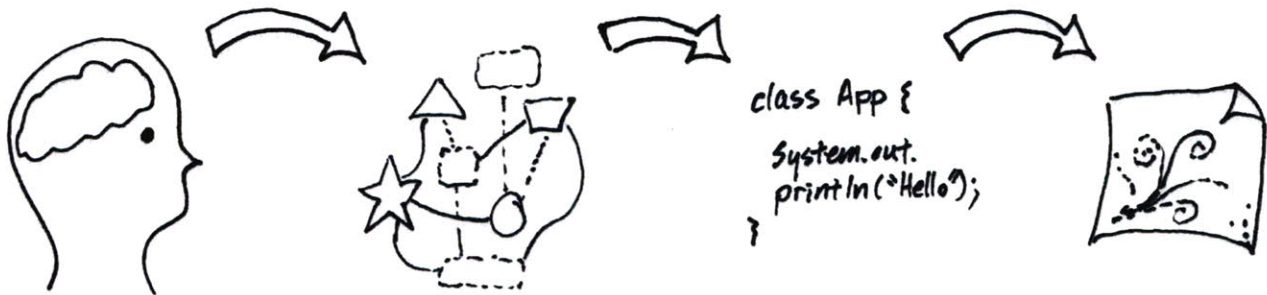


FIGURE 1-3 The creative process: ideation, mental model, code, output.

The individual starts with intentions, sometimes well-informed but often just a “hunch” (Figure 1-3). He will revise, reflect upon, and refine these intentions in his mind by externalizing them through visual sketches, or other manifestations that can be perceived through the senses.

Once the artist is ready to test his idea more concretely he will then transfer his process to the software development environment, to implement his idea in code and see how it looks as software output. Usually, the result is not satisfactory, and the artist revises the concept in his head and begins the cycle anew. As he reiterates over the creative process, the artist develops his intentions more fully as he learns about the affordances and limitations of the medium[29].

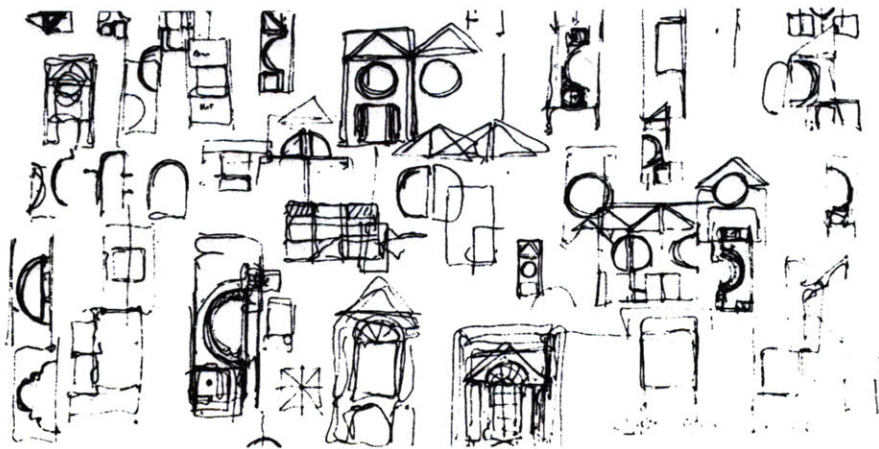


FIGURE 1-4 Iterations of sketches for architectural design (from [40]).

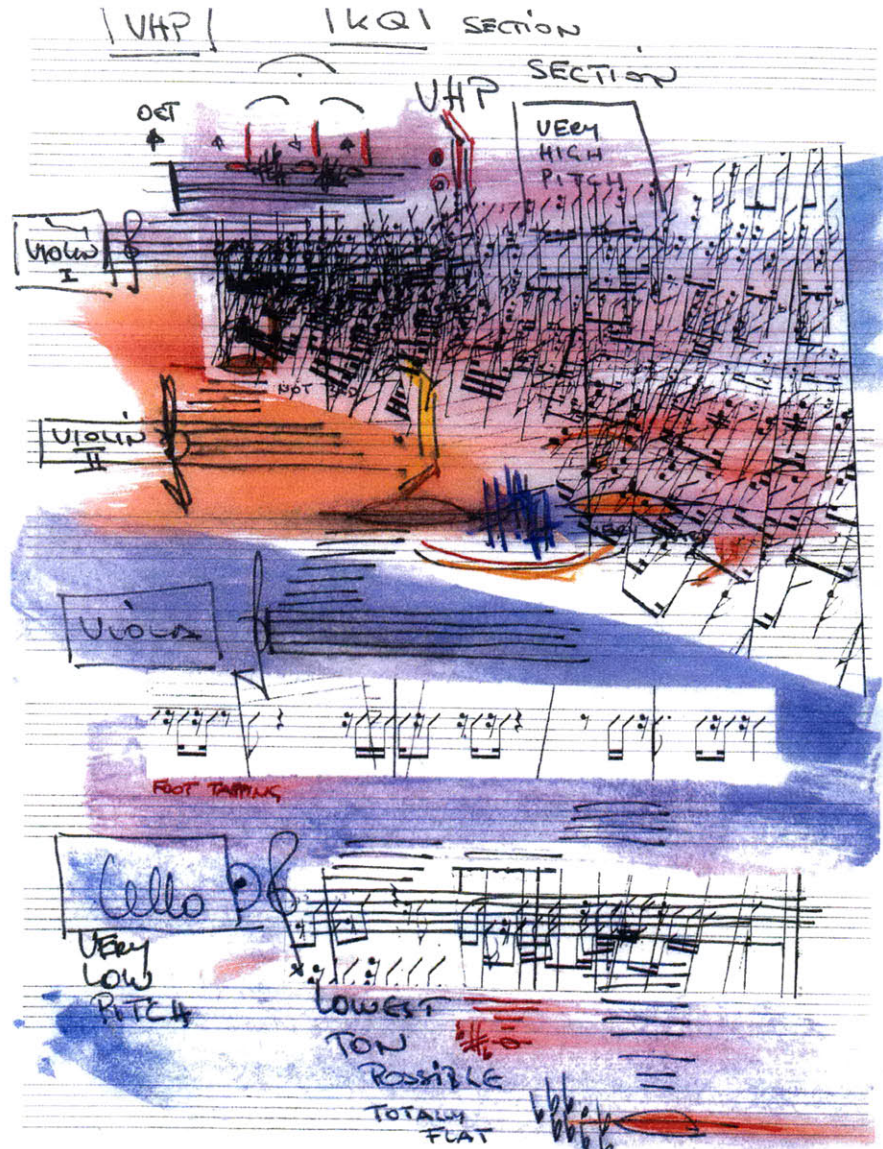


FIGURE 1-5 The artist-sculptor-musician Trimprin's externalizes his quartet via visuals[50].

The Mental Gap

Advancements in the design of creative coding environments have optimized language structure for visual-audio production and minimized programming minutiae through assistive tools. (I provide a review of such environments in Chapter II) Yet, despite the success of many of these efforts, programming remains generally difficult for artists, designers, and hobbyists because support tools can help with factual knowledge and factual representation but not with ways-of-thinking. Fundamentally, the difficulty in programming is caused by a mental gap -- a dissociation between an individual's mental conception of his piece of work, and the system structures of software.

Individuals exhibit a variety of approaches to thinking about problems[64]. A preliminary examination of some individuals' visual representations of their own mental models (Fig. 3) demonstrate that mental models comprise a variety of information: some are functional descriptions, such as "bounce a ball" (Fig. 3(a)), while others are system elements, such as "interface" and "buttons" (Fig. 3(c)). Donald Norman distinguishes between mental models and structural models as follows:

[structural models are] an accurate, consistent and complete representation of the target system... These are useful for understanding and teaching about systems. Mental models, on the other hand, are created by users as they interact with target systems and may not be equivalent to [structural] models. These are "what people really have in their heads and guide their use of things."[46]

Indeed, each individual's mental model is a combination of mental and structural models formed from personal experience[20], experience with various program-

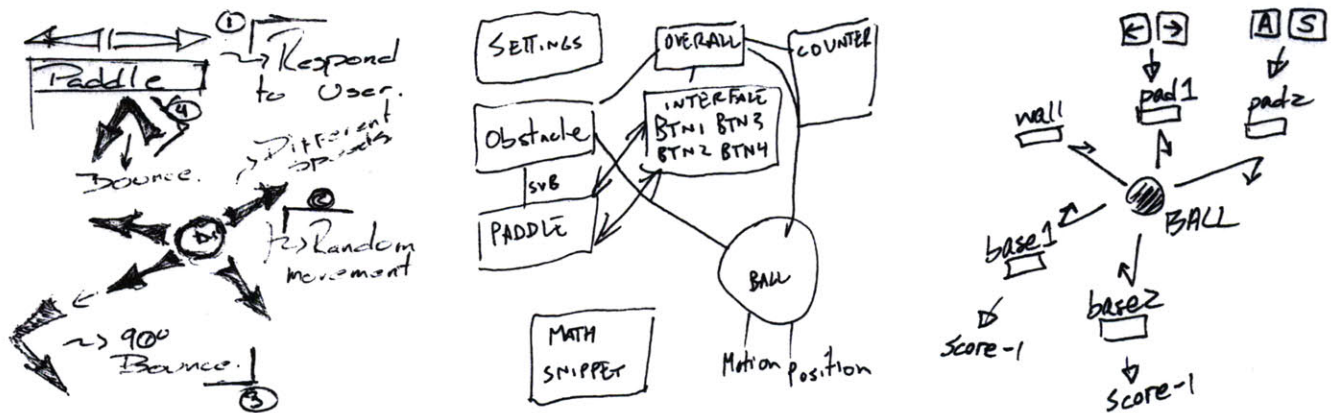


FIGURE 1-6 Three individuals' conceptualization of the classic two-player pong game.



FIGURE 1-7 The Funky Forest installation (2007)[66] by Theodore Watson is a project that is a result of creative code. It is built with openFrameworks.

ming paradigms, experience with the platform of choice, and experience with the specific context of the project they are trying to create.

In the particular case of creative coding, due to the iterative nature of creative work, the individual is continuously revising his mental model as he develops his concept. Furthermore, the artist's intention, and thus mental model, also continuously changes in scale and scope during this process: for example, when the artist is planning his intention could be a complete project description, but when he is tweaking and his intention could be as basic as "make it brighter".

In this way, the mental model is individualistic, context-specific, and continuously changing in the creative coding process. In contrast, programming languages are designed to allow correct program performance predictions in uniform terms across diverse tasks[20]. To write code, therefore, the artist must translate the mental model of his intention into a model that conforms with affordances of the given programming language. Since it would be impractical, if not impossible, to devise and master a programming language for every context in which a person would want to write a program, the cognitive translation from mental model to structural model is an unavoidable and integral part of programming.

For the software engineer, whose tasks have unambiguous pre-defined goals, his programming task comprises translating those goals into the most accurate, precise, and efficient structural model he can devise. For the artist however, each revision of his intentions and each adjustment in scale requires a corresponding translation, such that the artist must translate innumerable times in the process of creating a single software application.

At the same time, mental models are not completely imperceptible or immaterial prior to the completion of the finished product. Many people, especially visual-thinkers, externalize their ideas by making sketches and use the sketching process to develop creative concepts. Thus an artist's sketches visually represents his personal conception of his program.

If this visual information was digitized and made interactive so that it could act as an additional interface to the artist's code, an interface which was adapted to the artist's personal approach to his code, it could shorten significantly the number of cognitive steps between the artist's forming an intention to make a change to his project, and the code-level change required to affect his program output. Certainly, as creative programmers, the artist's focus should be spent on figuring out what he wants to make work, rather than how to make it work.

This thesis documents the development of a tool called Kaleido as an example of an individualistic visual interface. The concept of the individualistic visual interface

is motivated by the desire to help visual-thinkers program and endeavors to do so by enabling them to create personally meaningful visuals and to use these visuals to plan, organize, and navigate code in the individualistic way in which we each think.

Individualistic Interface

While the mental translation occurs in numerous places throughout the programming process, my intuition is that individualistic visual interfaces can assist in four interrelated aspects of programming: ideation, navigation, documentation, and learning.

Ideation

By supporting the creation of sketches side-by-side with code, the individualistic visual interface should help with new methods of idea-generation and idea-focusing while programming. First, by better supporting the consequential (as opposed to sequential) modes of problem-solving that are characteristic of the design process[54], and secondly by recording external representations and analogical connections that can be the catalyst for new ideas or serve as a reference point for the next divergence in path of design exploration[33]. Tightly integrating sketching into the creative cycle should shorten the feedback loop between idea and code. Meanwhile, since different types of drawing occur at different stages of design[28], the interface cannot claim to replace the many original sketching tools; rather, this work aims to discover the kinds of ideation possible with digital sketching.

Navigation

Navigation concerns the interface's ability to support the artist in finding the different parts of his program when he needs it. Once the artist establishes connection between his digital sketch and his code, the sketch essentially becomes a hyper-linked map of his program. This enables him to utilize visual dimensions, viz. color, shape, and spatial location, to organize and access different parts of his code. The challenge for designing the interface, however, will be to determine which visual dimensions to allow, and at which threshold might the breadth of visual variety yield more confusion than information to its reader.

Documentation

Digital sketches should be effective sources of secondary information that help people understand the program. Whether documentation is used when the program's original author returns to his program after a length of time, or when others try to understand a program written by another person, the digital sketch can act as



FIGURE 1-8 An electronic music composer's ideation sketch (from [16]).

a visual means of documentation. Visual documentation should excel at capturing higher-level information as well as at abstracting over some of the more minute idiosyncrasies of code. Code lends itself well to a bottom-up approach to comprehension but is much less helpful when reconstructing a higher-level view, and while the method by which people comprehend programs depends on complex factors[65][14][59], the visual interface should be able to help this process by offering alternative methods of comprehension. The difficulty in this case will be to design an interface for which documentation is easy to create and easy to keep up-to-date with changes in code, since documentation that is out-of-date will only cause confusion.

Learning

An interface that documents the author's thought processes should provide many opportunities for its reader to gain knowledge about programming and the use of software as a creative medium. Many forms of learning occur in the programming process, from the individual learning by revisiting his own old projects, to online communities sharing knowledge. Since the individualistic visual interface captures the author's strategy, which numerous studies have shown to be the most difficult part for novices learning to program[13][19], such interfaces could be effective in helping novices become more effective programmers. Further, individualistic visual interfaces foster interaction, experimentation, and visualization of code, all of which are characteristics which previous research has identified as essential elements in software environments for learning purposes[37].

For the purpose of this thesis I will be addressing creative code, and in particular creative code by artist-, designer-, and hobby-coders. Although the mental gap ex-

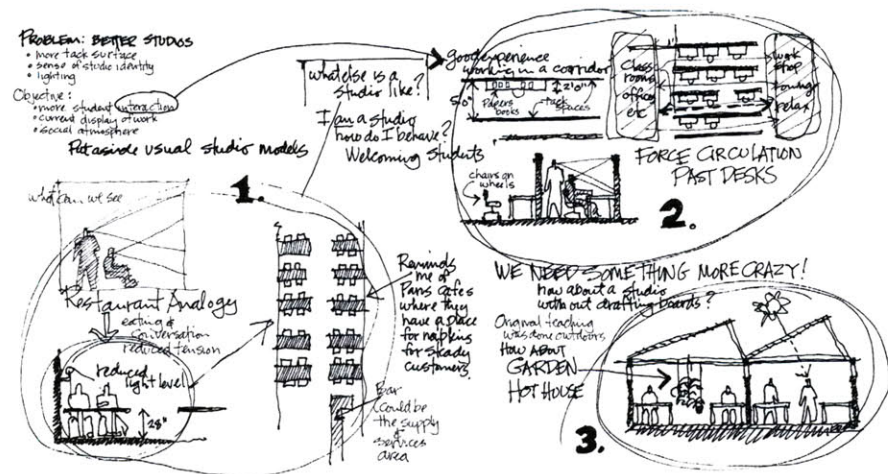


FIGURE 1-9 Sketches from a brainstorming session (from [40]).

ists in other forms of coding, and applies to software engineers as well as artist- and designer-programmers, I believe that artist and designer stand to benefit most from an individualistic visual interface for three reasons: firstly, software engineers are often well-trained through their education to habitually approach problems via a structural model and thus the cognitive translation is much less of a burden to them; secondly, as I mentioned before, the creative solution is open-ended and therefore the artist's mental model is continuously changing while software projects generally have well-defined goals from the onset; and finally, artists and designers are predominantly visual-thinkers who naturally think of their code in visual terms, and for whom visual representations are particularly evocative.

There are limitations to the individualistic approach, of course. Since mental models vary per individual, the interface should not and cannot make any assumptions about the mapping between visual components and program structural elements. It follows, therefore, that the interface cannot assume an exhaustive mapping and should support multiple relationships for a given object. For the same reasons, this project does not attempt to replicate any of the benefits of system-generated visuals: that is, the interface will not attempt to generate code from the model, or a model from the code.

While the lack of auto-generative functionality forestalls those unpredictable automated processes so opaque to novices, the trade off is that more responsibility is placed upon the programmer to create everything explicitly. A major challenge for the interface therefore will be to ensure that the process of working by hand is simple enough that the utility of the visuals outweigh the costs of creating and maintaining them.

Meanwhile, for the scope of this project, I will focus on the integration of mental models, externalized as sketches, with programming. As such, other representations of code such as data flow, control flow, and event flow, while each indispensable to our understanding of programs and helpful when integrated with the coding process, will be considered insofar as they are part of individuals' mental models, but not as models of information to integrate with the sketch and coding process.

Finally, the nature of mental imagery has traditionally been the subject of much scholarship and debate in the field of cognitive science. Researchers have sought to answer questions such as whether cognitive memory and computational processes are visual or spatial[24], pictorial (visual) or propositional (verbal)[8], or whether in fact mental models, propositional representations, and images are each one of three major kinds of representation[34], etc. For the purposes of this thesis, however, I will avoid such questions regarding the nature of human perception; rather, my primary concern with mental models will be its individualistic quality and the externalization process thereof when applied to the programming and creative processes.

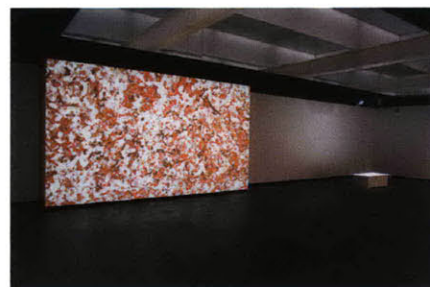


FIGURE 1-10 Process 11 by Casey Reas[51]. Each of the Process series is a text that defines a process and a software interpretation of the text.

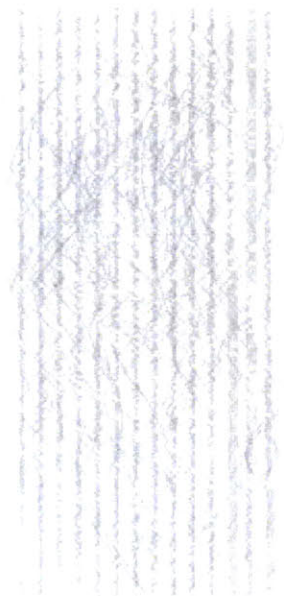


FIGURE 1-11 Dismap by Benjamin Fry (2003)[27]. A project that visualizes the operation of code.

Overview of Thesis

In this first *Introduction* chapter I have outlined the motivation behind this work, defined a common vocabulary to aid discussion, and laid out the goals and scope of this thesis.

In *Chapter II Context*, I provide an overview of past and current software tools that also explore the intersection of visual- and textual-programming, and discuss their contributions in relationship to the goals of this work.

In *Chapter III Design*, I identify a framework for the design of individualistic interfaces based on a review of some of the relevant literature, particularly those explaining the sketching and programming processes from a cognitive perspective.

In *Chapter IV Investigation*, I describe the methodology and results of a preliminary study conducted to gather empirical insight into the ways people naturally use sketches in their coding process.

In *Chapter V Implementation*, I describe the workings of the current version of the Kaleido system, as well as the implementation methods which include a manual for future developers.

In *Chapter VI Evaluation*, I evaluate the strengths and weaknesses of Kaleido by reviewing users' experiences as gathered from various venues.

In *Chapter VII Conclusion*, I summarize the findings and contributions of the Kaleido development environment, and I discuss points for future improvement as well as some alternative approaches to address the disconnect between individual creative thinking and the rigidly structural conceptual models of code.

The *Appendices* contain all original materials that were a part of this work. In particular, Appendix A contains all published materials used to explain, present, and support public use of Kaleido; Appendix B contains the study materials and anonymized results of the preliminary study; and Appendix C contains all study materials and anonymized results of the alpha (evaluative) study.

Mathematical reasoning may be regarded as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

...intuition consists in making spontaneous judgements which are not the result of conscious trains of reasoning.

...ingenuity consists in aiding the intuition through suitable arrangements of propositions, and perhaps geometrical figures or drawings.

Alan Turing

Context

In this chapter I provide an overview of past and current software tools that also explore the intersection of visual- and textual-programming, and discuss their contributions in relation to the goals of this work. In particular, I review the approaches of tools that support digital sketching, visualize programs, employ graphical interfaces for program exploration, enable programming via visuals, etc., as well as a group of creative environments that also integrate hybrid visual and textual programming paradigms.

Digital Sketching

Mainstream end-user graphing tools include Microsoft Visio[3] (Figure 2-?), FreeMind[1] (Figure 2-1), ConceptDraw MINDMAP[15] (Figure 2-2), and OmniGraffle[32]. These tools each support multiple types of graphs such as system diagram, mind map, and hierarchical trees, and thus exhibit complex interfaces, or at least multiple modes that each have a specialized interface. The general paradigm of shape creation among these tools is a palette window from which the user can drag a shape onto the canvas (e.g. Figure 2-2, 2-4). A large number of palettes provide a great variety of shapes and icons. In the case of OmniGraffle, this drag-and-drop method of visual editing was implemented even for text-formatting (Figure 2-4). Unfortunately, the palette window often requires a significant amount of screen area. In contrast, the industry standard Adobe Illustrator[62] and Photoshop[63] employ the paradigm of tools which take parameters (Figure 2-5). This allows the user more expressive freedom, although it also places more responsibility and effort on the user. The corresponding interface uses tool trays that expand, organizing tools by category and minimizing the screen area of the interface.

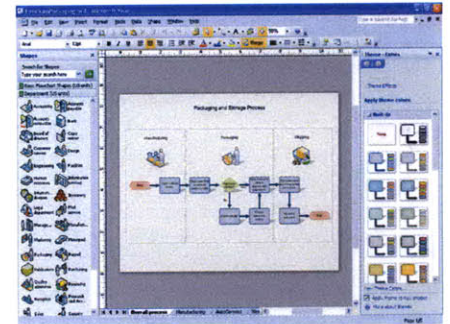


FIGURE 2-1 Microsoft Visio[3] also features various clip art palettes.



FIGURE 2-2 ConceptDraw MINDMAP[15] offers rich clip art galleries and uses a drag-and-drop paradigm for visual creation.

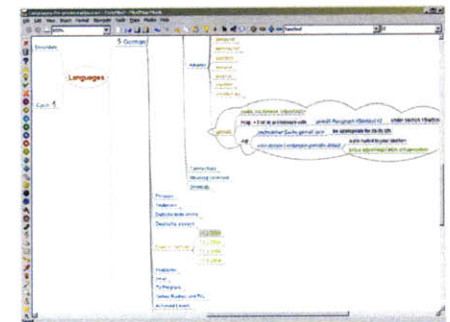


FIGURE 2-3 FreeMind[1], a mind-mapping application that allows the user to edit a hierarchical set of ideas around a central concept.



FIGURE 2-4 OmniGraffle's text-formatting palette that uses a drag-and-drop paradigm to apply effects [32].

Beyond the mouse-monitor-keyboard input paradigm, the field of Human-Computer Interaction has a long-tradition of research in systems that support digital whiteboard sketching with freehand drawing. The main research concern and design challenge for these input systems is the tension between enabling system recognition of discrete shapes and preserving the freedom of gestural input. Some systems support domain-specific recognition[47][39] while one system attempted to implement recognition across multiple domains[31]. In the case of individualistic visual interfaces, developing recognition systems is difficult when individual sketches vary so significantly. Instead, the system should probably simply let the user define discrete visual elements.

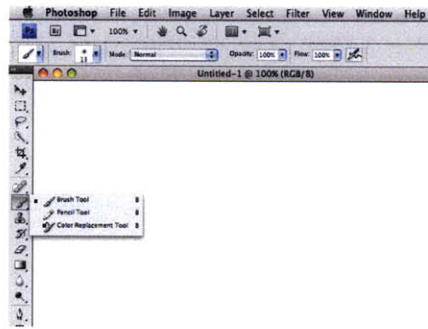


FIGURE 2-5 Adobe Photoshop[63] with tool trays for selection of similar tools, and the upper properties panel allows customization of the selected tool.

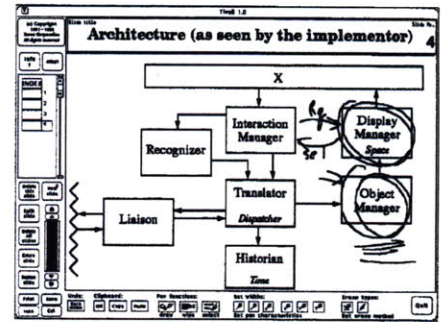


FIGURE 2-6 Tivoli (1993), an electronic whiteboard for informal workgroup meetings, integrated free-form input with discrete visual elements[47].

Programming As Visual Output

Many creative environments that are optimized for producing visual output via code include Adobe Flash[61] (Figure 2-7) and Dreamweaver[60] (Figure 2-8), where the paradigm is that of the system being aware of every discrete element of the visual output, and the artist dictating the actions of each element by attaching code to it.

With visuals acting as an interface to the code, these environments claim many of the benefits of individualistic visual interfaces; however, the visual representations in these cases must necessarily be composed of discrete output elements. Such paradigms work very well in cases where the artist's mental model is mimetic of the output but compromises the capacity to fully depict a mental model that includes abstract representations.

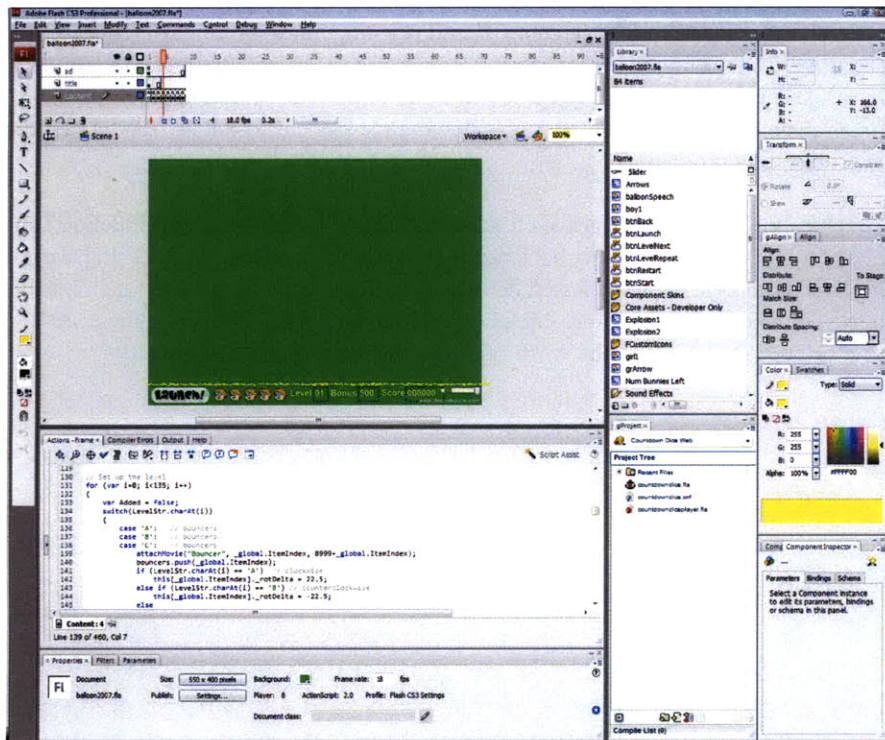


FIGURE 2-7 Adobe Flash interface (2010) with toolbar, timeline, canvas, code, and properties panels[61].

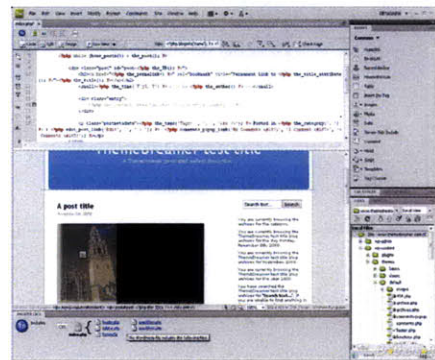


FIGURE 2-8 Adobe Dreamweaver interface (2010) with toolbar, code view, design view, and properties panels[60].

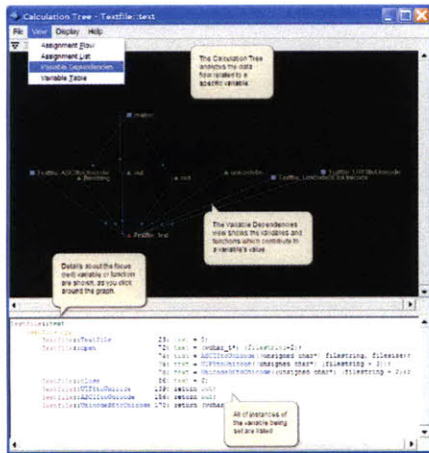
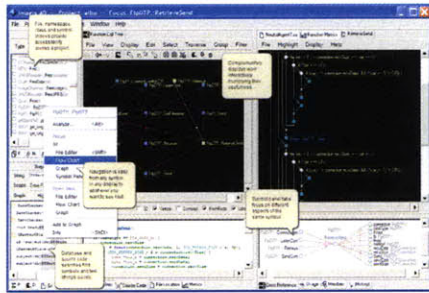


FIGURE 2-9 Imagix interface guides, top: function call tree and function metrics; bottom: variable dependencies[17].

Program Visualization

Many software comprehension tools focus on illustrating the hidden mechanisms of the software, i.e. the structural model, such as data flow and control flow. However, this information is presented often at the cost of omitting functional and context-specific information. Further, both software visualization tools and graphical modeling languages operate separately from the programming activity itself — that is, these computer-generated visuals, whether interactive or static, are meant to be studied in a separate activity from programming.

Software visualization designs vary from the aesthetic, the analytic, to the animated. “Botanical visualization of huge hierarchies”[35] visualizes hierarchical tree structures in the manner of 3D virtual botanical trees. The source code analysis program “Imagix”[17] (Figure 2-9) allows people to view both static and run-time information about their program – control flow, dependencies calculation tree, function calls with variables, etc. Similarly, the “SeeSoft”[23] (Figure 2-10) system allows users to map row-representations of code to program statistics such as version control, structural state (e.g. references), and run-time state (profiling). “Jeliot3”[44] (Figure 2-11), meanwhile, visualizes a program in action. It illustrates in real-time how a Java program is interpreted: method calls, variables, operation are displayed on a screen as the animation goes on, allowing the student to follow step by step the execution of a program.

Graphical modeling languages, such as the industry standard Unified Modeling Language[6], specify a standard visual language to describe system structures. Behavior Trees (Figure 2-12) employ a well defined notation to unambiguously represent the needs for a large-scale software-integrated system. However, neither of these are usable or even applicable to the work of artist and designer hobby-coders.

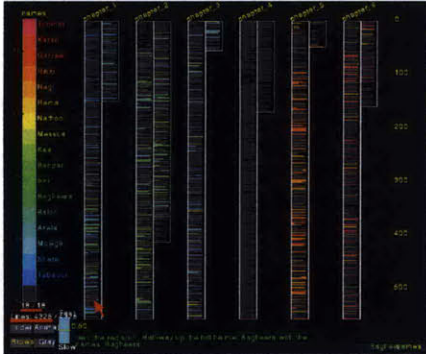


FIGURE 2-10 An example of the SeeSoft visualization showing locations of characters within a text[23].

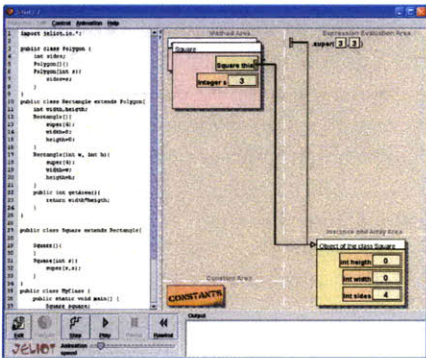


FIGURE 2-11 Jeliot 3[44] visualizes Java program execution through animation.

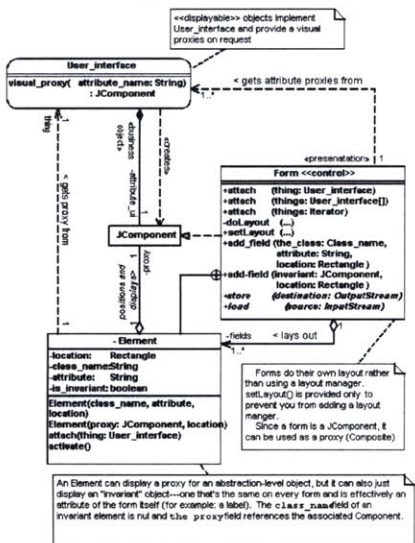


FIGURE 2-12 Unified Modeling Language class diagram[6].

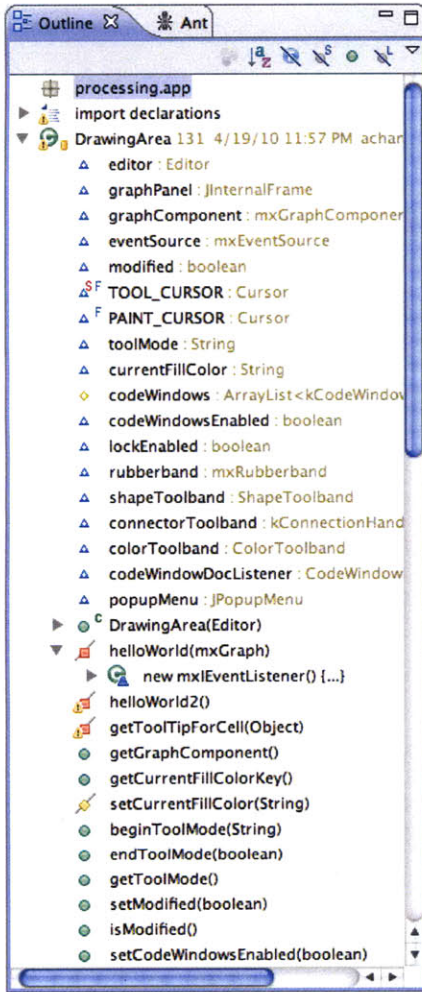


FIGURE 2-13 Eclipse outline view displaying program components (e.g. methods and variables)[26].

Program Exploration

Software exploration tools present graphical representations of static software structures linked to textual views. The distinction between these tools and software visualizations is that it is integrated with the programming activity and it is interactive — namely, it is an interface for accessing code. The open-source platform Eclipse[26] offers an outline tool that lists the structural elements of the currently open file. The information presented includes structure names, types (e.g. class, field, or method), and hierarchical relationships. The popularity of this tool demonstrates the need for different methods of navigating code beyond the list of project files and the scroll bar.

Past research projects include SHriMP[59] (Figure 2-14), whose authors defined a framework of cognitive dimensions intended as a guide for designing software exploration tools. They point out the problems that few tools support top-down comprehension models, and more support is needed for mapping domain knowledge to code. However, the SHriMP implementation suffers from visual usability problems, and while it considers cognitive dimensions, it imposes a specific graphical representation and therefore a particular model on the program. Further, it is also not optimized for creating as much as it is for understanding.

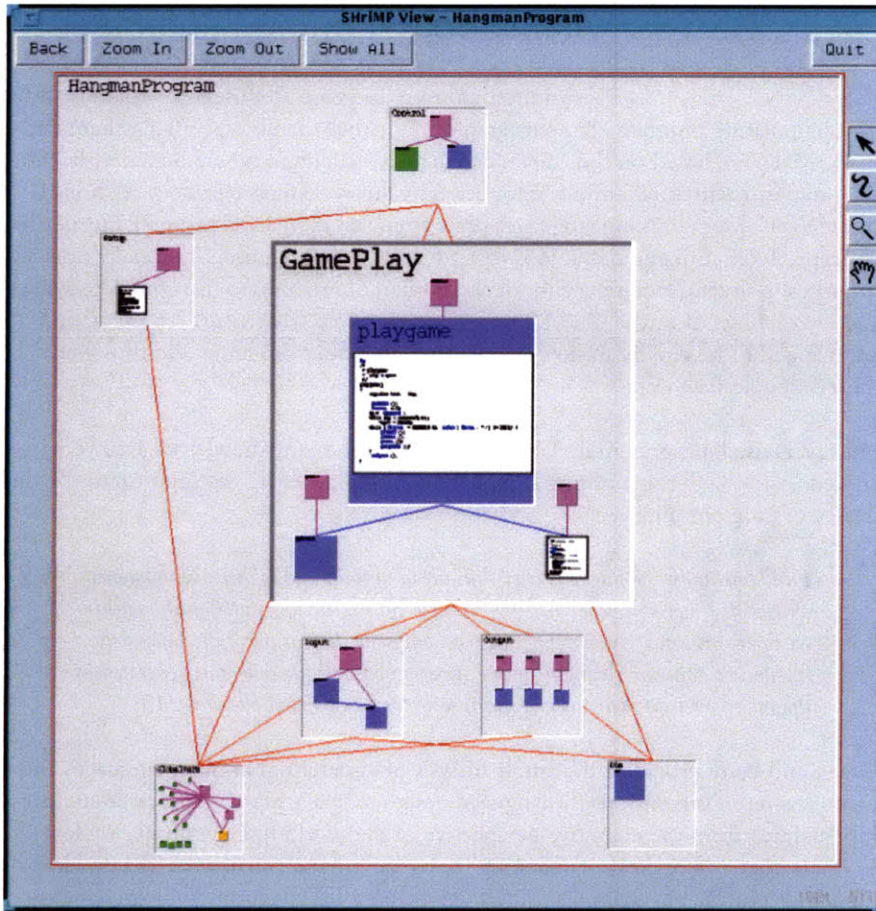


FIGURE 2-14 The SHriMP tool, designed to support the construction of a mental model during software exploration[59].

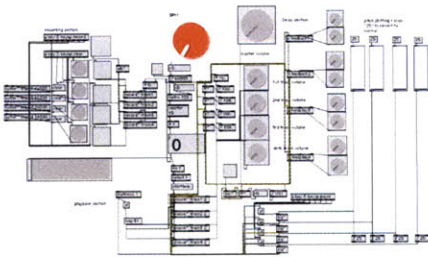


FIGURE 2-15 The Max/MSP visual programming language[18].

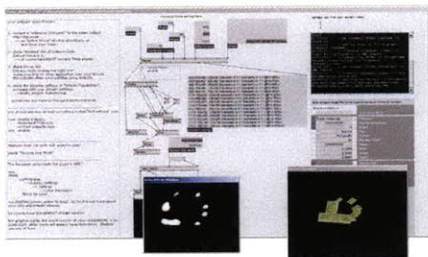


FIGURE 2-16 A complex project in Max/MSP encounters the problem of cluttered wires.

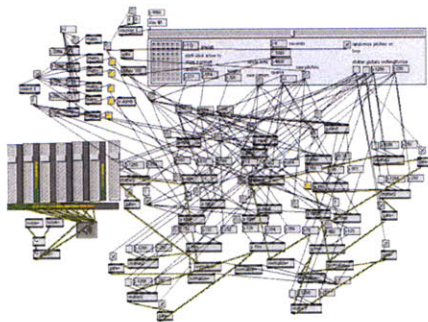


FIGURE 2-17 The vvvv programming environment, demonstrating a video project[7].

Visual Programming Languages

Visual programming languages let users create programs by manipulating program elements graphically rather than by specifying them textually. Many are based on the “boxes and arrows” paradigm where boxes represent the program’s structural entities, connected by arrows which represent relationships. Visual programming languages draw from the Human-Computer Interaction concept of direct manipulation[55] that advocates allowing users, especially novices, to “access powerful facilities without the burden of learning to use complete syntax and lengthy lists of cards” [ibid.]. The rationale was also that visual representations are mapped more closely to how people thought about programs and thus were more intuitive than code.

However, visual programming languages suffer from many drawbacks. Early critics argued that visual programming languages could not ever be the solution to the difficulty of programming:

Fundamentally... software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross references, dataflow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the diagrams generated by the many relevant views, it is difficult to extract any global overview.[13]

Green and Petre provide a thorough analysis of visual programming languages and identify a set of dimensions for language design, in particular a number of competing qualities, from the cognitive perspective of the user[30]. They point out, for example, that a “closeness-of-mapping” between program structures and problem domain is crucial, but that increasing the number of abstractions, which is powerful in certain scenarios, leads to hidden dependencies, which are confusing in other scenarios. The authors believe that in order for visual programming language to be as effective as their textual counterparts, further work must be done to resolve the cluttered wires problem (Figure 2-16), to support secondary notation (i.e. visual dimensions of color, pattern, etc.), and to support the ease with which a user can make a change in the program. Myers, when making a taxonomy of visual programming languages[45], similarly noted that poor representations made the visual program hard to understand once created and difficult to debug and edit, and that programs were tedious to edit once they got large.

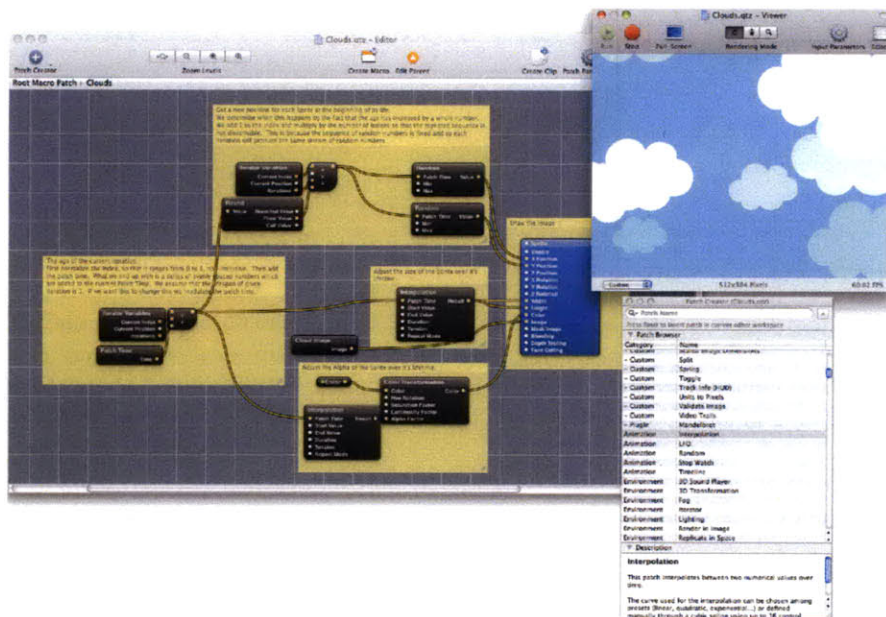


FIGURE 2-18 The Quartz Composer visual programming language and environment[9].

Currently, the only visual programming languages that have successfully gathered a broad user-base are those that employ structural perspectives which align closely with common mental models for specific types of tasks. Visual languages based on data flow, such as Max/MSP[18] (Figure 2-15), vvvv[7] (Figure 2-17), and Quartz Composer[9] (Figure 2-18) aptly employ the metaphor of river and tributaries for projects based on streaming audio and video. Each visual object acts as a filter that takes data input and outputs modified data, and that path which data travels is visually mapped out. This paradigm is referred to as node-based. However, such metaphors are not easily generalizable to other tasks.

The fundamental problem remains that the visuals are in general ill-mapped to people's mental imagery and the fact remains that no language can map to people's mental imagery in every case; any complex cross-domain visual programming language must necessarily need many abstractions, which increases the exact difficulty which the visuals were originally intended to alleviate.

Hybrid Creative Environments

In recent years, a number of hybrid visual-code environments have been developed, each with various design goals. These hybrid creative environments allow multiple perspectives and multiple interfaces to the programmer's code.

BlueJ

BlueJ[38](Figure 2-19) is a software development environment for programming Java, developed mainly for teaching programming. The main window displays a graphical representation of the program's class structure (a UML-like diagram, albeit many times simplified) and objects can be interactively created, edited, and tested, allowing easy experimentation. Object-oriented concepts such as classes, objects, and method calls are represented visually. The only drawback is that users don't have control over the creation of the graphical representation.

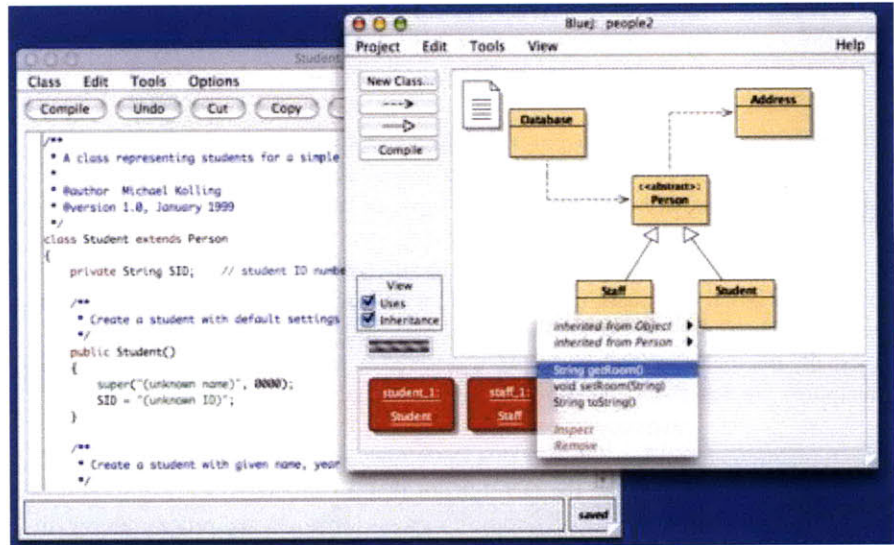


FIGURE 2-19 The BlueJ environment developed for teaching Java programming. The interface here shows a code view on the left, and the program view on the right[38].

CodeBubbles

CodeBubbles[12](Figure 2-20) provides a new paradigm for interacting with code by breaking code fragments down into small simple graphical containers which the user can then spatially arrange and navigate between. Significant design effort was put into displaying code comprehensibly within a restricted screen area, bubble layout options, as well as user-created bubble groups as identified by the color of the halos. Bubbles are “opened” or “created” as needed by the user. This paradigm breaks away from the sequential model, and by allowing the user a large amount of control over how they access code, it is in many ways an individualistic interface. The limitation is only that the user does not have much control over the visual expressiveness of the code bubbles aside from layout.

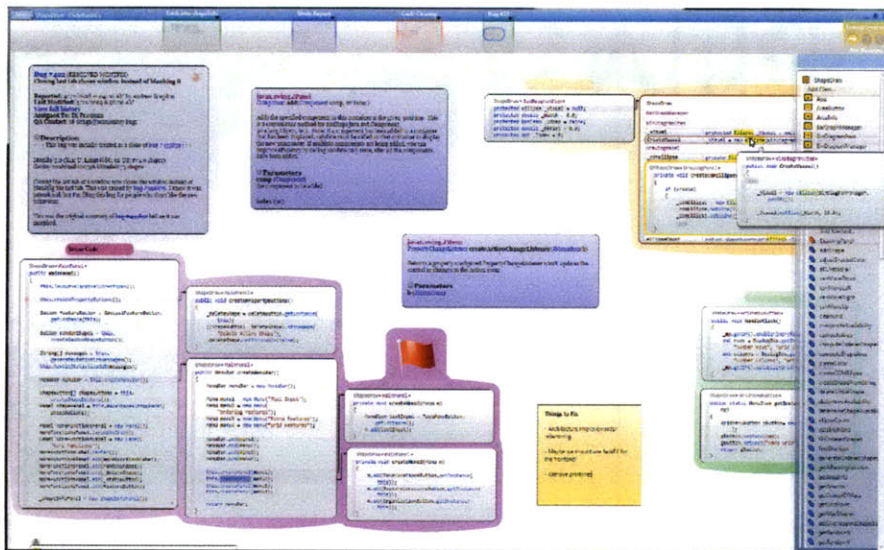


FIGURE 2-20 The CodeBubbles interface, each bubble created by the programmer containing a fragment of code[12].

NodeBox

NodeBox is an environment for 2D animation in Python, and NodeBox2[4] (Figure 2-21) provides a visual interface with a node-based paradigm that supports users frequently writing code to create custom nodes. The result resembles an environment such as Quartz Composer integrated with a code editor. The user has control over spatial organization of graphical elements and he/she can create their own nodes, but unfortunately the visual space does not support graphical elements without a structural counterpart, e.g. each node must be a method and is required to have an input/output.

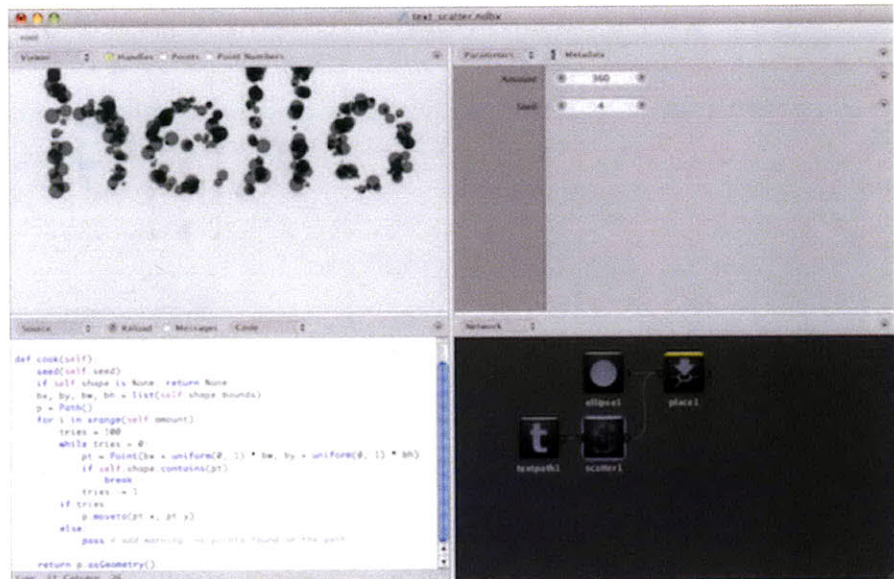


FIGURE 2-21 The NodeBox2 interface, the output on the top left, parameters on the top right, python code on the bottom left, and nodes on the bottom right[4].

Field

The Field programming environment[22] also takes a multi-paradigm approach, and more specifically, complements textual code with user-defined visual abstractions of code. The Field canvas (Figure 2-22) enables the user to arrange custom visual elements (boxes/buttons), with GUI elements (variable sliders), with elements of visual output (Bézier curve and control points). The result is an individualistic interface comprised of a mixture of visual UI elements that is created by the user throughout the programming process.

However, Field implements a visual paradigm distinct from this work, where visual elements are functionally powerful but semantically less so. For example, a particular variable that the user might need to tweak extensively during pilot testing could be attached to a slider element and made accessible at any time by placing it in a prominent location on the canvas; the slider element, however, is thus restricted to indicating a single variable, and cannot represent other types of information such as “ball bouncing math” or “3rd scene”. In this way, the paradigm of Field’s canvas is more similar to assembling your own control panels than sketching and integrating your sketch for different purposes. Nonetheless, Field demonstrates a successful approach to individualistic visual interfaces that offers intuitive and direct manipulation of the program from the visuals, and the design of individualistic interfaces could benefit from integrating aspects of this approach.

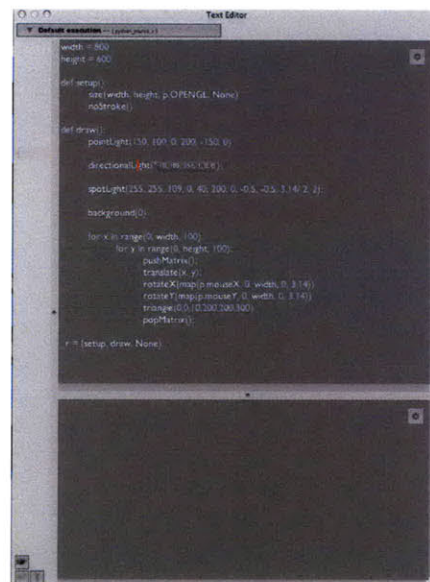


FIGURE 2-23 The code window of the Field environment[22].

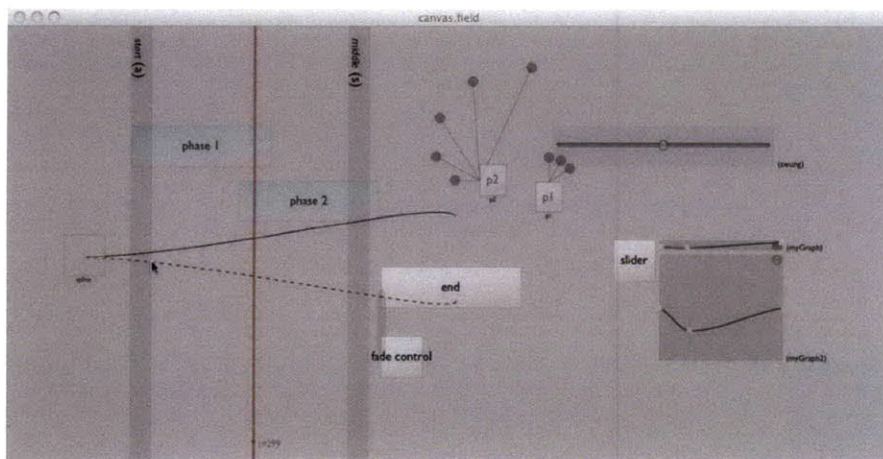


FIGURE 2-22 The Field canvas[22] enables the user to arrange custom visual elements (boxes/buttons), with GUI elements (variable sliders), with elements of visual output (Bézier curve and control points).

Artistic activity is a form of reasoning, in which perceiving and thinking are indivisibly intertwined.

A person who paints, writes, composes, dances... thinks with his senses.

Rudolf Arnheim

Design

In this chapter I identify a framework for the design of individualistic interfaces based on a review of some of the relevant literature. I draw heavily upon research explaining the sketching and programming processes, particularly from a cognitive perspective, to identify a set of design considerations, as well as the competing tensions among them, which guide the subsequent parts of this work.

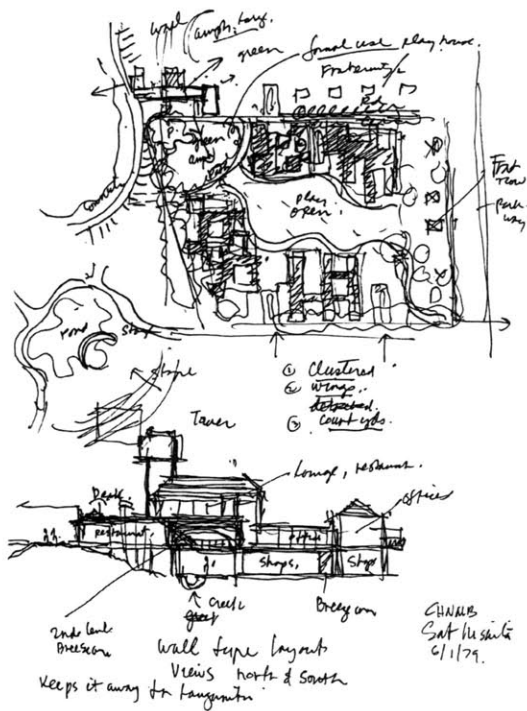


FIGURE 3-1 Site design sketch (from [16]).

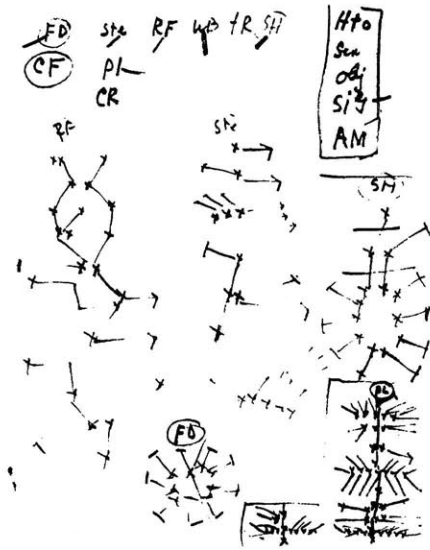


FIGURE 3-2 A choreographer's sketch (from [16]).

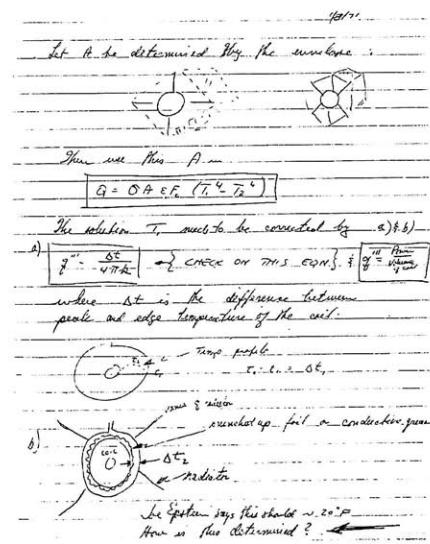


FIGURE 3-3 A scientist's research sketch (from [16]).

Individuality

At the fundamental level, this thesis is about integrating more elements of individuality, the basis of creativity[28], into the hitherto comparatively formal programming process. From the design perspective, it has long been established that design thinking is episodic, non-linear, and based upon prior personal experience[54]. Programming, however, is an equally individualistic activity; as Turkle puts it, "Your style of solving logical problems is very much your own"[64]. Each individual's mental model of their program is a combination of mental and structural models formed from personal experience[20], experience with various programming paradigms, experience with the platform of choice, and experience with the specific context of the project they are trying to create. Empirical studies of programmers' mental imagery have also demonstrated that mental imagery is "complex and non-uniform"[49].

Thus, an underlying theme for designing individualistic visual interfaces is paying especial care when making assumptions regarding the user's actions. For example, as discussed under the Expressivity guideline later in this chapter (page 48), any design of an individualistic visual interface should avoid system-generated visuals so that the interface neither selectively supports nor implies creations of a certain type. Neither should the interface assume that a certain code structure should be connected with a specific visual element, or vice versa.

At the same time, an interface cannot manifestly claim to support every possible visual depiction; instead care should be taken to identify the range of diversity and support variance when possible. More concretely, this work should identify, and the interface should support many ways of approaching the task: visual creation, code creation, program navigation, visual editing, etc.

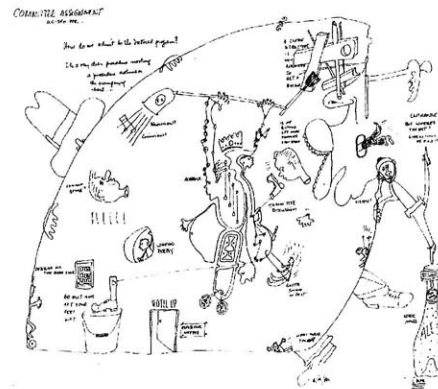


FIGURE 3-4 An educator's sketch to organize his lecture (from [16]).

Reflectivity

According to Donald Schön, “Designing is a conversation with materials conducted in the medium of drawing”[57]. In this reiterative cycle of the creative process, sketching serves as a method of thinking. Critically different from analytical problem-solving, the solution to designing is emergent rather than planned, and the sketch, as an externalization of mental imagery, also informs the development of mental imagery at the same time; which is to say, when sketching, “perception and conception occurs simultaneously.”[10]

The first implication of a reiterative process is that a critical aspect of the visual creation component in individualistic visual interfaces will be the quality of “viscosity”, defined as the ease with which edits can be made[30]. If changes are difficult to make, the interface can cause friction in the design process, thus causing frustration with the tool, or lower-quality creations.

The second observation is that reflective design also relies critically upon the availability of multiple versions of the design or a rich history of past design explorations and decisions from which to derive the next iteration[10]. Erasure marks on paper and pencil, for example, records rich information that shows the designer the trail of

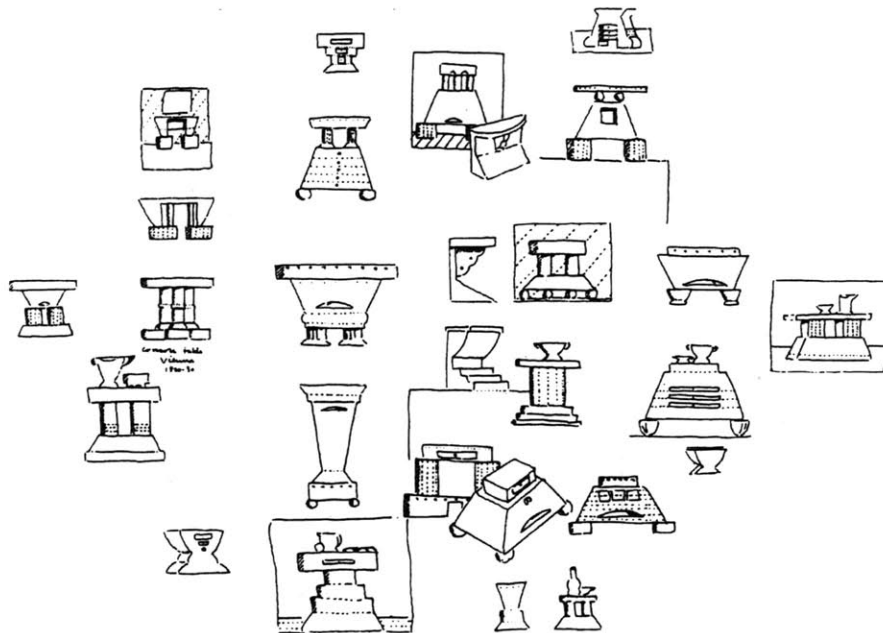


FIGURE 3-5 Table design studies (from [16]).

Ambiguity

Another notable characteristic of sketches is their depiction of ambiguous information. As mentioned earlier, design is a process of gradually discovering the solution in the sketch, and thus sketches “... that contain information that is fluid, vague, ambiguous, indeterminate, play an important role in solving ill-structured problems like design.”[28] Interpretation-rich visuals helps creativity; while graphics are monosemic visual elements for which the meaning of each sign is known prior to the observation of the whole, sketches are polysemic, i.e. the perception of which consists of decoding the image[11]. Thus, designers-in-training are urged to develop versatile and facile sketching skills so they develop more flexible and creative solutions[40]. Further, designers often intentionally make drawings vague or ambiguous to reflect the yet-indeterminate state of that particular design factor[31].

Yet, the fine gradient of indicators of ambiguity, such as a formless or blob- like shape, or lines drawn in a lighter hand, are difficult to fully capture in a digital system, and the creation of a digital mark that carries the same subtle information of a simple pencil mark, while certainly possible, could easily become a task in and of itself that detracts from the main task of designing. If rich-input systems such as pressure-sensitive digital pens are unavailable, one solution is to translate ambiguity into a binary state as a compromise between usability and functionality.

A closely-related consideration is that sketches are often incomplete, i.e. designers sketch more detail at certain levels and less detail for others[25]. Even in the design of software, engineers have been noted to pay different amounts of attention to different parts of the design as part of their unique perspective on the problem[42]. The implication for individualistic visual interfaces is that the interface should not assume that there is a “completeness” for the visuals, nor an exhaustive mapping between visuals and code.



FIGURE 3-7 Conceptual sketch (from [40]).



FIGURE 3-8 Sketch of Siena, Italy (from [40]).

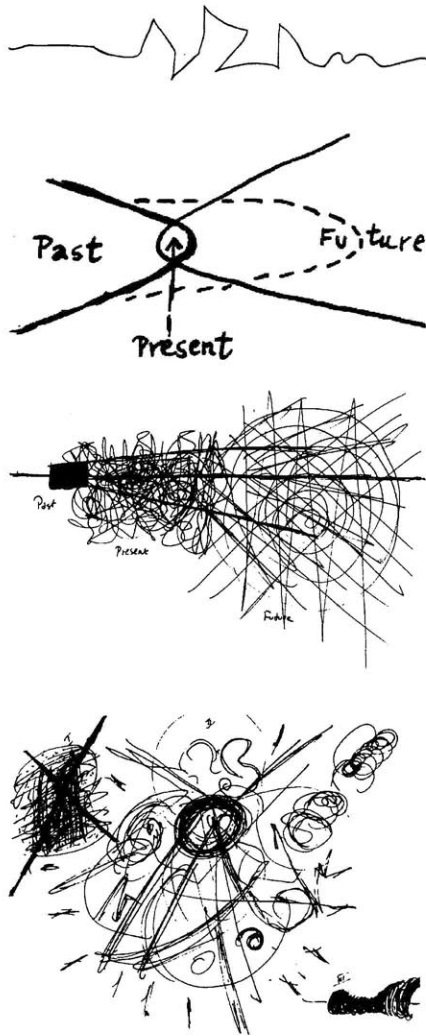


FIGURE 3-9 Sketches from a study, where participants were asked to depict the abstract concept of “past, present, and future”. Sketches here are from four individuals (from [10]).

Expressivity

The quality of expressivity concerns itself with the visual style and type of visuals that the interface should enable such that users are able to visually express what they wish. The extent to which the user-created visuals can express people’s mental models is fundamental to the interface’s ability to support the various programming activities of ideation, documentation, navigation, learning, etc.

The question therefore, is what types of visual representations do people naturally use to externalize their mental images. Arnheim observes that people use a range of visuals that form a gradient between the mimetic and non-mimetic, as well as “disembodied” shapes to depict perceptual features such as the expansiveness of a color, or the aggressiveness of a sound[10]. Meanwhile, in a study of programmers’ mental imagery, Petre and Blackwell revealed that “many of the visual images described bore some resemblance to standard external representations, although often these would be dynamic in the mind, changing with different dimensions, or augmented by other views or additional information.”[49]

Classical semiology from Peirce asserts that there are three kinds of visual-meaning relationships: iconic, symbolic, indexical[48]. An icon, also known as a semblance or likeness, is mimetic and possesses the character of the object it signifies (e.g. a pencil streak represents a geometric line); a symbol refers to its object through a conventional agreement (e.g. the meaning of a word is determined by societal agreement); and an index indicates through logical connection (e.g. a bullet-hole is the sign of a shot).

One implication is that any tool supporting the visual representation of mental imagery should enable the entire spectrum of mimetic, symbolic, and indexical visuals to allow the user to depict in the manner he finds most intuitive. This implies also that each type of visual representation should be equally simple to create with the interface. Meanwhile, the decision to prioritize expressivity for the individual necessarily occurs at the expense of other people trying to understand the visual is a fact of which the designer of the individualistic visual interface should be aware.

A second implication is the necessity of supporting a range of secondary notations such as texture, color, and layout so that users can convey additional meaning beyond the symbol. The competing factor in this case is visual cohesiveness and comprehensibility, the latter not just for the benefit of others but also for the user himself. Visual clutter quickly renders the individualistic visual interface ineffective, and so principles of visual communication must be considered when choosing the visual possibilities to enabled, to choose a set of possibilities such that visual coherence is optimized.

Finally, it should be noted that expressivity is often, although not always, in tension with the goal of simplicity (this chapter, page 53). If taken to the extreme, expressivity could take the form of support for innumerable visual options but such a decision would result in too steep a learning curve.

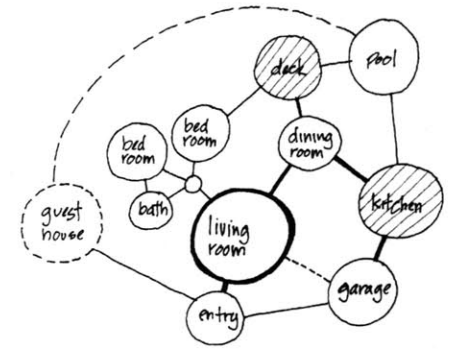


FIGURE 3-10 Graphic diagram of a house (from [40]).

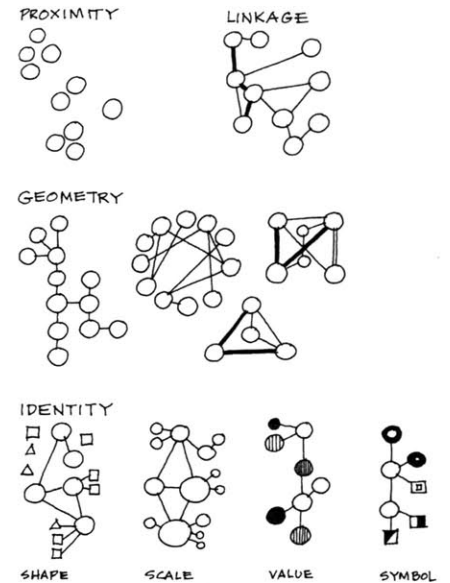


FIGURE 3-11 Graphical organizing systems (from [16]).

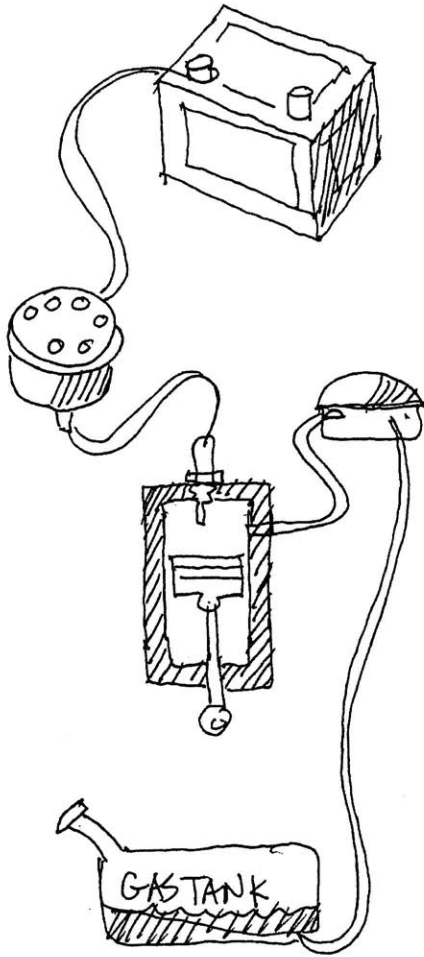


FIGURE 3-15 Sketch of a car (from [40]).

Connectivity

Connectivity in individualistic visual interfaces governs the creation, maintenance, and functionalities of cross-references between visual and code. As with the visuals, these cross-referenced links are determined by the user. This “hyperlinkage” functionality is a direct benefit of transferring the sketch process from analog to digital, and one of the primary motivations behind integrating sketch and code is the ability to access code via a mapping to user-defined visuals.

As mentioned before, mental models are individualistic, incomplete, and ambiguous. As such, links between code and visual elements are multi-modal and unpredictable: code-visual relationships can be many-to-many, many-to-one or none at all. The ideal interface would enable the user to create any of these links at any point in the process, from any scenario (e.g. when the user is writing code, when user to drawing, when user is editing, when user is about to draw, etc.).

Since links are complex, non-exhaustive, and non-uniform, clear indication of the state of the connections becomes of primary importance to the user. Embedded visual depiction is necessary to allow the user to see any connection in context at any point in time, however the visual solution is not immediately obvious since drawing visual lines between elements would be impractical in many ways, and presenting connections in a list would hardly be helpful. Another vital piece of information that the user needs to know at the same time is which visual elements are linkable, if the environment makes a distinction between linkable and not-linkable elements at all.

Finally, the power of sketch-code integration lies in how the linkages are carried through various programming activities ranging from writing, editing, to debugging. Thus, it is important to enable as many associated or synchronized operations as possible (without making assumptions) both to fully realize the potential of using visuals as “avatars” of the code, but also to prevent the code and the sketch from getting out-of-sync by reduce the workload of user to keep both sides constantly updated. Additionally, operations on linked elements need to be carefully considered because in different scenarios the user could want different treatments/outcomes of the linked relation.

Transparency

I refer to the transparency of the system as the system’s ability to provide continuous feedback and indication of the system state as the user interacts with it. When designing a system for non-experts to program, understandability and simplicity should be valued over efficiency[21]. While ambiguity and

unpredictability in sketching are important, the system itself should be unambiguous about what it knows so that the user may spend a minimal amount of effort predicting the results of his actions. In the case of individualistic visual interfaces, transparency is particularly important due to the extra layer of complex non-uniform information, and the multiple modalities in which the user could be working.

The modality should be made clear whether the user is in drawing mode or coding mode without the user needing to search for or process additional information; indication should be obvious but not obtrusive. The user should always know on which layer of information is the current focus of the interface, as well as which options are available to him in the current situation, and which are not. The same guidelines hold for other modes such as linking and editing visuals.

Simplicity

The interface components of the individualistic visual interface should be minimally intrusive so as not to detract from the main goal of doing creative work. As the popularity of writing tools that bring back “typewriter basics”[58] demonstrate, simpler functionality and minimal distraction is what is needed for creative tasks. As exemplified by the Processing IDE[5], for non-experts it is preferable to have fewer functionalities all of which operate in a way clearly understood in all scenarios so that the user never feels out of control.

Also, as discussed in the previous chapter, a basic but influential aspect is the amount of screen real-estate that the interface elements occupy, which should be as little as possible to maximize the space for creating (Context chapter, page 29).

Many creativity support tools[56] and applications for non-programmers [53] discuss a “low floor, high ceiling, wide walls” principle, referring to the combined characteristics of a low threshold to entry, higher-level complex functionality, and wide cross-domain applicability. For the developer, it means designing effective component parts that immediately function in very basic configurations, but which can also be configured to perform complex functionalities. The tension here, however, is that atomic building blocks can build theoretically everything, but at a certain threshold the immense effort required to assemble the blocks for a higher-level functionality renders the tool unusable.

Finally, simplicity notably comes into conflict with expressivity and ambiguity. For example, to avoid confusion over the identity or boundaries of a visual element which is important knowledge for establishing links to code, visual elements should be discrete and distinct, characteristics which forgo the expressivity of gestalts.

In the method of design, of small differences in start points leading only to the unpredictable, I looked into the non-linear and its special character, and was intrigued.

Cecil Balmond

Investigation

In this chapter I supplement the theoretical findings of the previous chapter with empirical observations of my own. In particular, I conducted a preliminary study to gain insight into the ways people naturally use sketches in their coding process, and I document the methodology, results, and implications of this study. In the study I asked people who thought of themselves as “visual-thinkers who program” to create drawings as if they were preparing to write code, and to do this for three different types of programs. The results were tabulated and used to inform the design of the Kaleido prototype.

Approach

Thus far, few studies have been conducted to examine people’s mental models of code. Petre and Blackwell[49] is the exception; however, they did not study the visualization of the imagery but rather relied upon verbal description, and further, they had limited their subjects to expert programmers. Thus, this study was conducted to discover the coding process of people who are creative coders, self-proclaimed visual-thinkers, and not generally of software engineering background. What types of visual styles do people naturally use? How do they use it in their coding process? Which parts of their current coding process do they feel is unintuitive, difficult, or needs support?

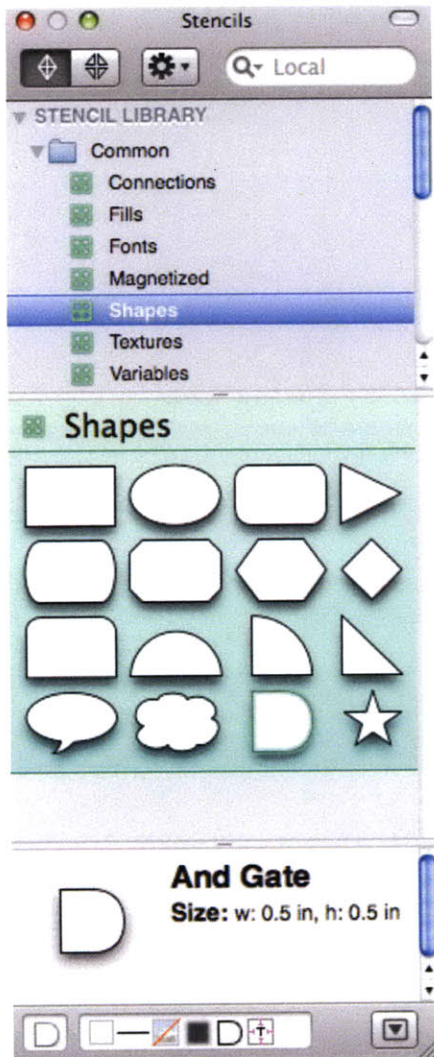


FIGURE 4-1 The Shapes palette from OmniGraffle[32] demonstrating the visual elements available.

Methodology

In order to recruit a greater diversity of participants (Processing users are distributed internationally), the study was designed to be accomplishable remotely and independently, such that subjects could do the tasks on their own time and at their own pace. The study consisted of three drawing tasks and a post-task questionnaire per person. Instructions were identical for all participants.

TASK NUMBER	TOPIC NAME	INSTRUCTIONS
Drawing I	Two-player Pong	napkin sketch
Drawing II	Mario side-scroller	use provided visual palette of shapes
Drawing III	Your program	use any tools, be creative

The first drawing task was designed to discover what visuals users naturally use to depict their mental representations. Users were instructed to treat the drawing as a “napkin sketch”, and “draw a sketch as if you were organizing your thoughts before starting to program.” The program scenario given was a traditional two-player Pong game, chosen to reflect the real-time interactive nature of the digital art projects which Processing and Kaleido seeks to address.

The second drawing task sought to discover what visuals users would use when given a limited, discrete set of visual elements, as is the custom when using contemporary graphing software like OmniGraffle (usage of text was unrestricted). The defined set of visuals consisted of equilateral polygons, flowchart symbols, power-point shapes (e.g. speech bubble, explosion symbol), and some system icons (e.g. human figure, keyboard, sound). The scenario chosen was a Mario-style side-scroller game, which incorporated elements of narration and interactivity as well as a game victory system. Further, users were encouraged (but not required) to choose three different colors in which to render their drawings, and to use legends if they found it useful for themselves.

The third drawing task was designed to try and discover unexpected ways in which people think about their programs. Users were simply instructed to choose a program whose source code they were willing to share with me, and to submit a creative visual depiction of it (“What if you could make a collage of your program? What if you could dance your program?”).

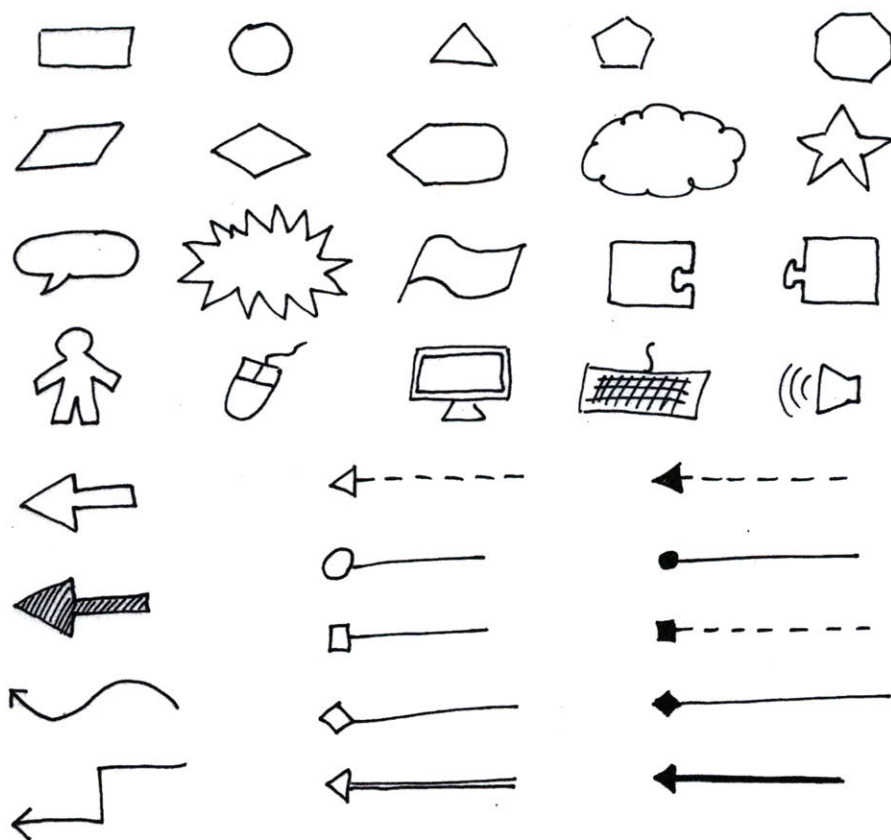


FIGURE 4-2 The given palette for the second drawing task.

The questionnaire consisted of ten short-answer / multiple choice questions that asked about the user's demographics, programming work, and experience, as well as six additional free-response questions regarding their creative process and how they thought current programming activity could be visually augmented. Participants were instructed to complete the questionnaire after the drawings, in hopes that, through the process of working through the drawings, they will already have had a chance to reflect upon their own processes.

The study was advertised to Processing community, OpenProcessing.org, OpenFrameworks.org, and through the mailing lists of university Media/Digital Art programs. Participants were given a brief overview of the Kaleido project, the purpose of the preliminary study, and were recruited specifically as "visual thinkers who program".

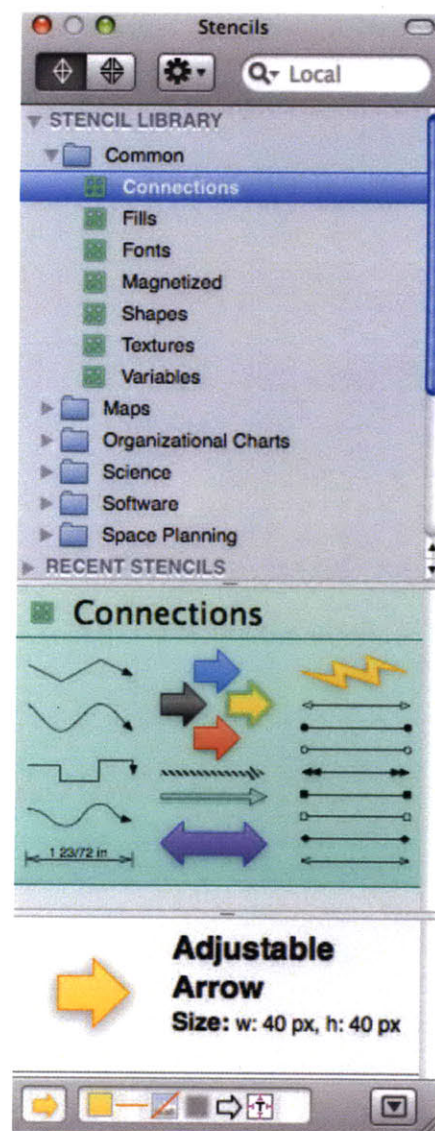


FIGURE 4-3 The Connectors palette from OmniGraffle[32] demonstrating the visual elements available.

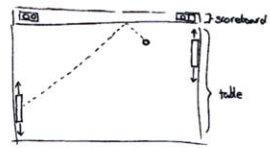
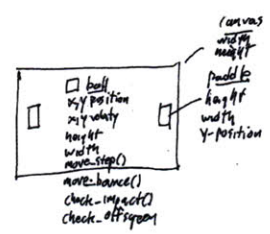
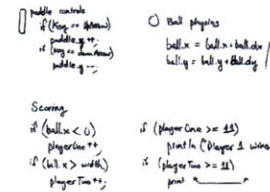
Results

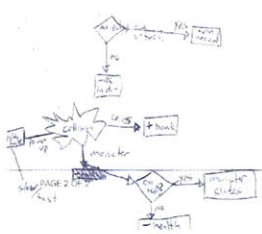
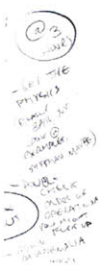
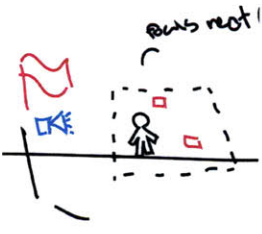
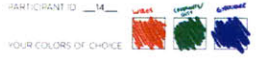
Demographics

A total of 11 people participated, one of whom completed only the survey and not the drawings. Of the participants, nine were men and two were women; nine were students and two were professionals. When asked to characterize their work, the top three keywords were “interactive” (90.9%), “graphics” (81.8%), and “web” (45.5%). Their averaged 8.45 years of programming experience, with a range of one to 23 years. Most (81.8%) had programming classroom experience in college, while 63.6% also said they were self-taught. 27.3% of participants’ first programming language was Processing, 18.2% was each Logo and BASIC. The top three languages people were comfortable using (81.8% each) were C/C++, Actionscript/Javascript, and Processing.

Drawings

I used a loose categorization system for the types of information that people recorded in their drawings:

CATEGORY	DESCRIPTION	IMAGE EXAMPLE
output	graphic representations of program’s actual output	
class structure	organization of object-oriented classes	
functional modules	program subdivisions created according to function	

CATEGORY	DESCRIPTION	IMAGE EXAMPLE
logic flowchart	graphic representation of computer's decision flow	
pseudo-code	code fragments in loose syntax	<pre>if y > height skip y direction</pre>
task list	itemized list of programmer tasks	
UNIQUE TO DRAWING II		
three colors	used three colors in the drawing	
legend	made a legend (either for colors or for shapes)	

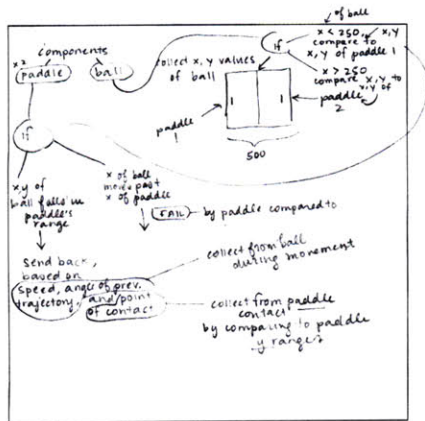


FIGURE 4-4 Flowchart as the base for information organization, with pseudo-code and output overlaid.

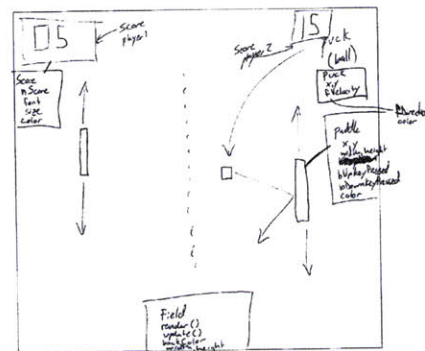


FIGURE 4-5 Output as the base for information organization with class structure information overlaid.

The response breakdown for Drawing I: Pong was:

TYPE	NUMBER OF USERS
output	8
class structure	4
functional modules	6
logic flowchart	3
pseudo-code	5
task list	3

The response breakdown for Drawing II: Mario was:

TYPE	NUMBER OF USERS
output	7
class structure	3
functional modules	6
logic flowchart	5
pseudo-code	2
task list	2
UNIQUE TO DRAWING II	
three colors	8
legend	3

According to this categorization system, all drawings ranged between two and four types, confirming the hypothesis that people combine multiple types of representations in their mental model. Most drawings used one type of information organization as the basis for laying out all the other information, e.g. using a flowchart as the basis with pseudo-code and output overlaid (Figure 4-4), or output as basis with class structure overlaid (Figure 4-5). Most drawings comprised one single drawing, while others drew multiple distinctly separable diagrams although each possibly comprised of multiple types of information (Figure 4-6).

In Drawing II, users used colors to distinguish different types of information. Examples of legend include “code/data”, “user input”, “game objects” (Figure 4-7), “wires”, “comments/gist”, “experience” (Figure 4-8), “objects”, “actions”, “controls” (Figure 4-9).

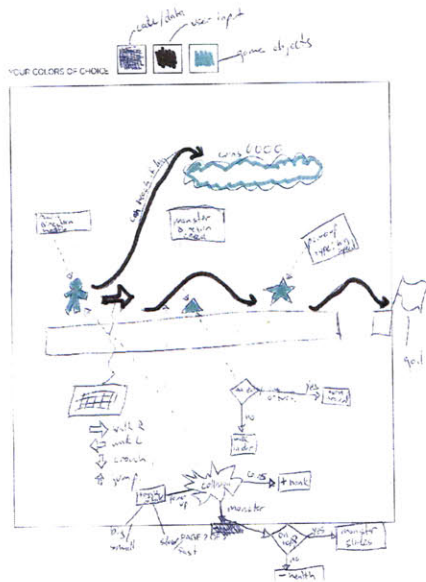


FIGURE 4-10 This participant used a star shape to signify a good object, and a triangle shape to signify a "bad guy" in the game.

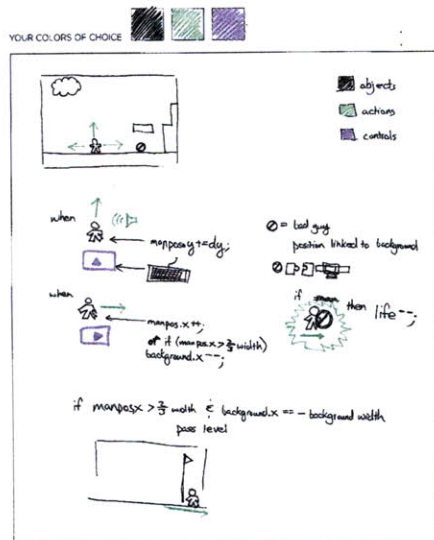


FIGURE 4-11 This participant used the cloud symbol to illustrate the game scenery, and the flag symbol to signify a checkpoint in the game.

In Drawing II the most commonly used shape was the humanoid and the plain rectangle (Figure 4-15). The non-flowchart shapes were often used in depiction of on-screen elements in the game; for example, a star shape as a good object, a triangle shape as a bad object (Figure 4-10), or a cloud symbol as part of the background (Figure 4-11), and the flag as checkpoints or goals (Figure 4-11, 12). System symbols were very commonly used, with all but two participants using at least one of the system symbols. After the humanoid, the keyboard was most often used to indicate user input.

Two participants created custom shapes that were not explicitly made available in the given palette, but could be created from the given shapes (Figure 4-13).

Text was often the predominant element of the drawings. However, there was not much variance in text appearance – a few participants made distinctions via all-capitalizations (Figure 4-14), and one artistically rendered (Figure 4-13), but those were the exceptions.

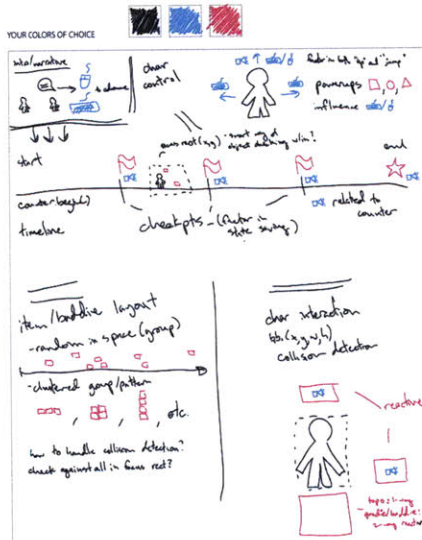


FIGURE 4-12 This participant used the flag symbol to signify a checkpoint in the game.

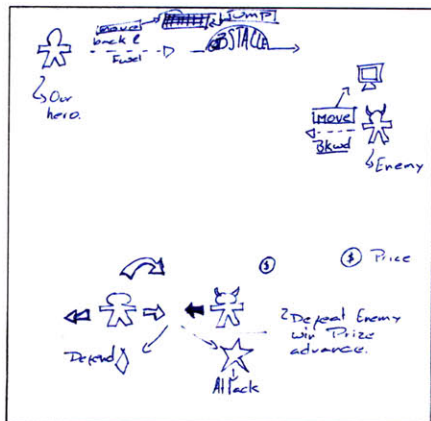


FIGURE 4-14 This participant created his own shape to indicate enemy game characters, and also morphed text to give added meaning.

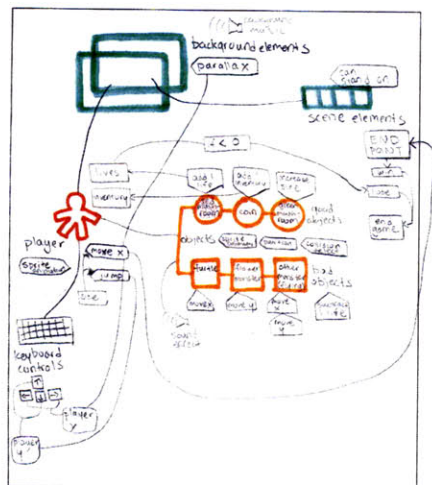


FIGURE 4-13 This participant used capitalization for emphasis.

■ % of participants who used it one or more times
 ■ % of usages that used only one instance

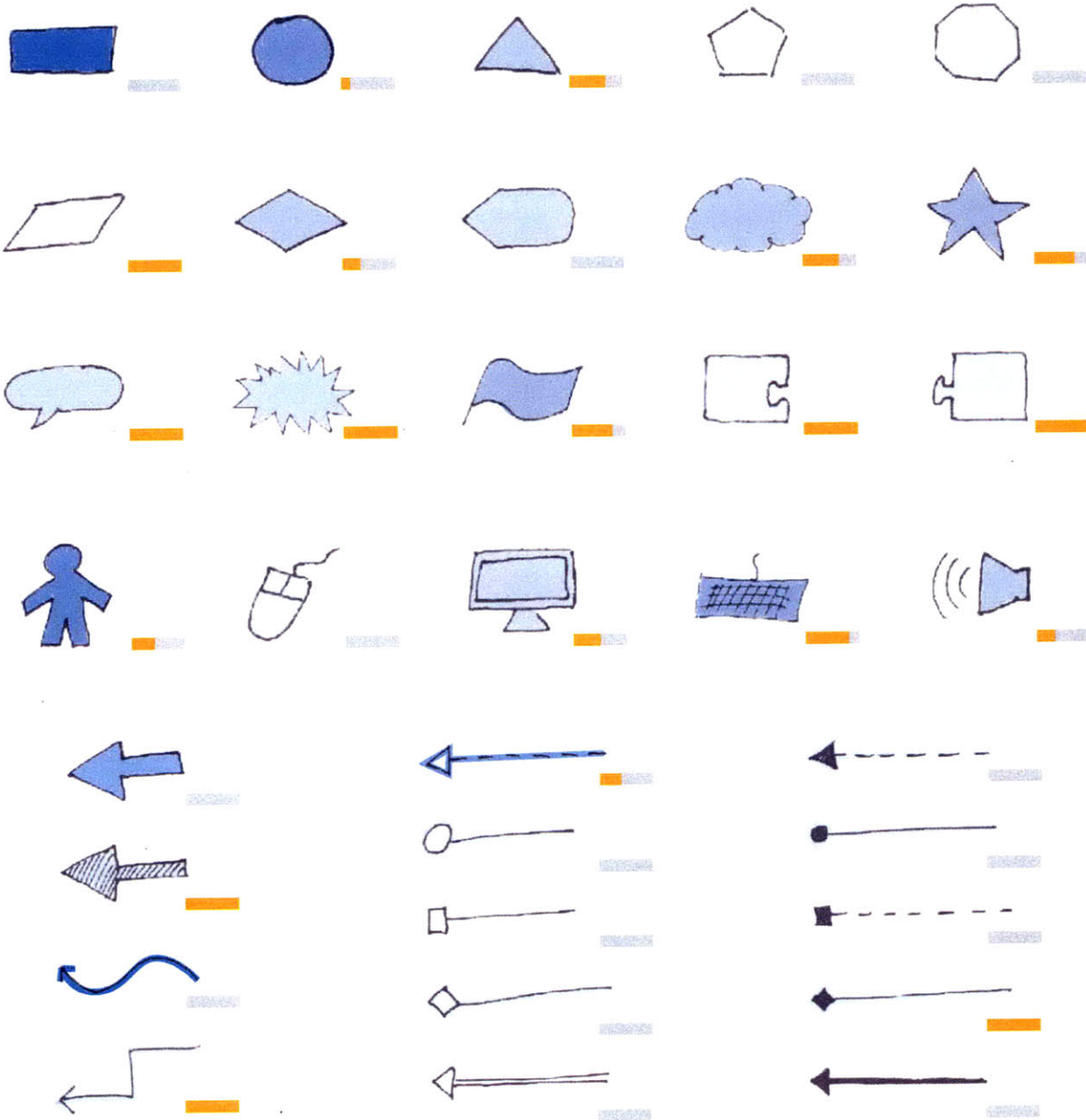


FIGURE 4-15 Palette shape usage frequency chart.

The responses for Drawing III were widely diverse. Most participants used images of their sketchbook from when they were working on their project (Figure 4-15), which reveal more of their process of conceiving the project rather than implementing it. Some participants collaged these images and annotated them as if explaining the concept to other people (Figure 4-16). Other participants created a visual representation of their program as if a poster (Figure 4-17). Many probably took the approach of explaining their program to other people (viz. me, the investigator) seeing as one user who used flowcharts in his two previous drawings opted for a visual depiction of the different types/layers of information his program would show to the user.

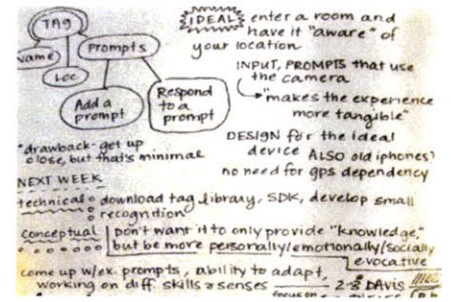


FIGURE 4-16 This participant copied her process sketchbook as a representation of her project.



FIGURE 4-18 This participant illustrated her process using images from various stages of the work.



FIGURE 4-17 This participant sent an artistic interpretation of his project.

Questionnaire

Participants revealed that they approached programming tasks in many ways: some start with sketching out the front-end user interface, based on the rationale that “the UI reflects the logical functionality of the program”; others enjoy starting with some existing code and “hacking” while they figure out what they would like to create: “I like gradually adding to code and refining an idea as I go”. Most claimed that they take multiple approaches, whether it be a combination of pseudo-code, flowchart, UI sketch, code hacks, or descriptive prose. One participant confirmed the need for approaching drawing and coding simultaneously: “[once I start coding] I’ll often go back to sketching... it’s not a linear process of working in one medium and then moving to another”.

Regarding the contents of their personal sketchbook, responses were split between doodles that included all variety of content (text, diagrams, visuals, both project-related and unrelated), and not using one to begin with. A minority of participants claimed not to own a sketchbook; some claimed they make project-related sketches anyway on loose-leaf paper, or digitally (one cited a dislike of the “permanence of order” of a sketchbook).

As expected, the idiosyncrasies that code allows become some of the hardest parts to recall when a programmer revisits their own old code. Naming conventions (e.g. xcount, xsize, xlen, xnum, xn) are easy to forget and hard to search for with a “Find...”, but a fundamental problem was “how all the parts interact with each other”. Part of this is the problem of “how I serialized [my program] (where each function is in the file, that sort of thing)”, and part of it is “how events are received by other parts of the program”. Other responses mentioned unusual arithmetic sequences that require re-evaluation to figure out their purpose, and specific project scenario details (e.g. formulas for musical scales and harmonics, official US military taxonomy, etc.)

The last section of the questionnaire asked participants how they thought current programming activity could be visually augmented. The most common responses called for call/stack traces and data/event flow, particularly for debugging. A couple of participants wanted visual navigation of program methods, put succinctly as “the equivalent of sticky notes in a book”. Finally, another participant notably wanted the ability to isolate one portion of the code so “you can check it and tweak it until it works, then put it back.”

Limitations of the Study

While the results of the study were revealing about the visual styles and the types of drawings people produced, some of the questions which this study did not address include:

- differences in drawings due to nature of the task
- reading other people's drawings
- development of the drawing over time / over the course of programming
- z-layering, visible vs. invisible grouping
- layout mechanisms (grid, snapping, etc.)
- scales / zoom-in

Beyond the observation of the end-results of sketching, future investigations should include an observational study of how users develop their drawing while programming — for example, some questions could be: how often do they switch between drawing and coding, when do they switch, and how does the result of their code affect what they draw, etc. For the visual interface navigation methods, a paper-prototype approach might be able to shed some light on these questions.

Summary of Findings

The study reveals that a limited number of discrete visual elements are sufficiently expressive, and that a drawing platform should allow functionally-representative visual elements such as the humanoid shapes and the keyboard shapes, or at least allow easy functional-representation via simpler visual elements that could be used as building blocks.

As expected, the drawings varied widely in style, but most significantly, all drawings were found to be a composite of multiple drawing systems; e.g. an illustration of the output, a diagram of the class structure, a logic flowchart, programmer's task list, etc.

It was discovered that, although users were creating visual representations, support for text is critical: comparing the drawings of users across the first two tasks reveal that people use a lot more text than expected, especially as their task grows more complex.



FIGURE 4-19 Examples of the paper-prototyping method.

You cannot expect the form before the idea, for they will
come into being together

Arnold Schoenberg

Implementation

In this chapter, I describe the detailed workings of the Kaleido system in its current version. I reference the design guidelines developed earlier, and I explain the rationale behind my design decisions for this particular implementation of an individualistic visual interface. I present an introductory user's manual to describe the system's functionalities, and finally, I provide a basic manual to guide future developers through the software engineering setup of the system.

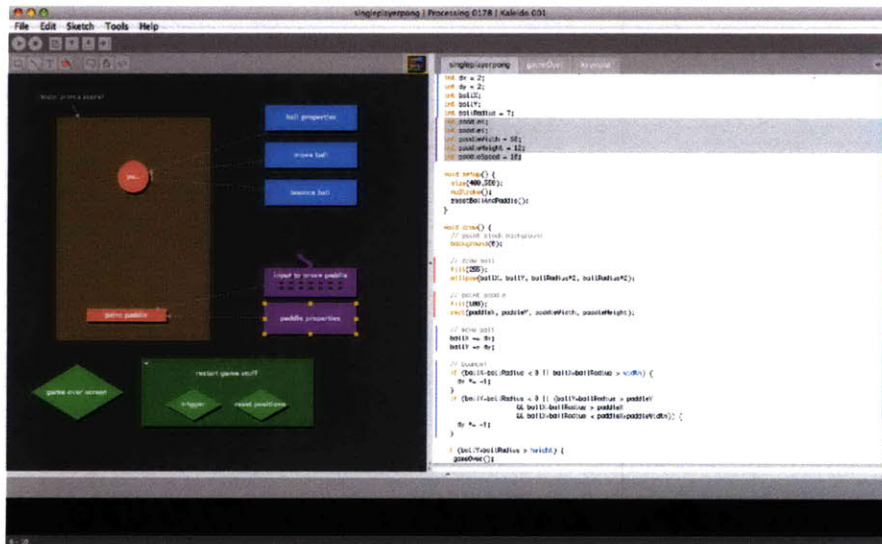


FIGURE 5-1 The Kaleido software development environment.

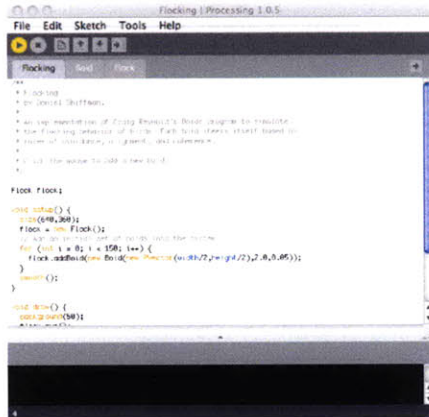


FIGURE 5-2 The Processing Development Environment version 1.0.5.

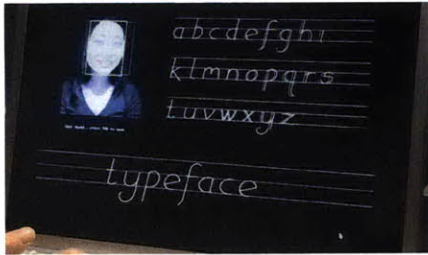


FIGURE 5-3 Typeface by Mary Huang creates a font that changes with your facial expression in real-time. Built in Processing.



FIGURE 5-4 The New York Talk Exchange data visualization by Aaron Koblin[36]. Built in Processing.

Project Setup

Kaleido is a simple software development environment based on the Processing integrated development environment (IDE)(Figure 5-2). Processing is a programming language based on Java, designed for the electronic arts and visual design communities with the purpose of teaching the basics of computer programming in a visual context[52]. The decision to implement this work in the Processing IDE is based primarily on a coincidence of purpose — namely the philosophy that “software is a unique medium with unique qualities” and that “programming is not just for engineers” [ibid.]. Another consideration was that members of the Processing community comprise the primary user base that this work intends to target. An alternative implementation in the form of an Eclipse Plugin was considered but dismissed due to its predominantly expert programmer user base.

On the left side of Kaleido’s environment (Figure 5-1) is the drawing area, and on the right, the text area. Each project, when opened, appears in a separate editor window. For any given project, the text area might contain multiple code files, but a single drawing area and a single drawing is shared across the multiple files. The lower half of the editor displays the debugging console. A basic toolbar above the drawing area offers all drawing functionalities. Keyboard shortcuts provide access to more advanced functions.

Drawing Area

The drawing area is a canvas that enables users to create digital drawings by means of the drawing tools. Users can control the placement, size, color, and labels of any of the visual elements. They can edit the text label of any drawn element by selecting it, and then clicking once to summon the label editor (Figure 5-7). The canvas size dynamically grows as elements are inserted, and to access empty space the user can simply zoom out. Drawings larger than the drawing area window can be panned via mouse and scroll bars, or the viewer on the far right of the drawing toolbar (Figure 5-5).

Visual Vocabulary

The decision to use discrete graphical elements (Figure 5-6) rather than free-form drawing was made early in the design process, the primary motive being implementation efficiency in consideration of the timeframe of this work. Alternative visual styles and input interfaces, for which many of Kaleido’s shortcomings become a non-issue although other difficulties are introduced (e.g. gestural drawing input), are discussed in the Future Work section of the Conclusion chapter (page 98). Meanwhile, this decision also simplified the visuals for easier comprehension – with discrete visual elements there would be no confusion over the boundaries of a object and with it the object to which a link applied.



As the preliminary investigation revealed, a limited number of shapes as well as a limited number of colors is sufficient to express mental models, while good support for text is necessary (see Investigation chapter, page 60 and 62). Thus, I chose to make available a handful of the most frequently used shapes, connectors, and created a “text box” shape with a transparent background that only contained text. To constrain complexity, text boxes and arrows were made not-linkable. The hypothesis was that people would use them to annotate their drawings rather than expressly denote a piece of the program.

The preliminary investigation also revealed that few if any users employed stylistic distinctions of text (e.g. size, weight, font, etc.) to differentiate meaning. Thus, for Kaleido I decided to focus on differentiating meaning via shape instead, and reducing complexity by removing all text styling options except the most basic bolded “title” and regular weight “description” fields in the shape labels (Figure 5-8).

Drawing Toolbar



SHAPES

Click and hold down to see other options such as  and . All shapes hold a title and description, and can be linked to code. Any new shape will be filled with the color indicated in the Color Fill button at the time of creation.



CONNECTORS

Plain basic connections in three style options (arrow, dotted, etc.), which can be attached to shapes or left dangling. Connectors can be labelled, but not linked to code.

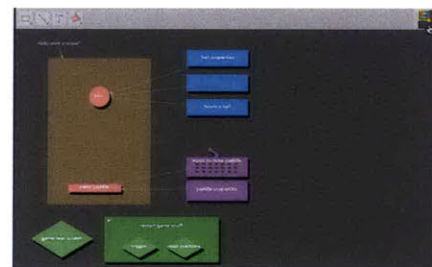


FIGURE 5-5 The viewer on the far right of the drawing toolbar can serve as a means of navigating the drawing.

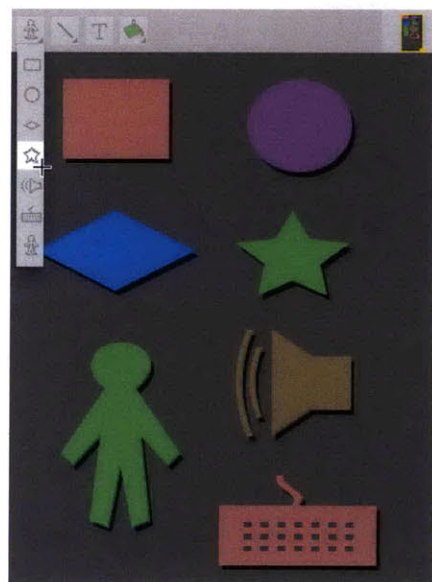


FIGURE 5-6 A demonstration of the available shapes and the shape menu drop-down.

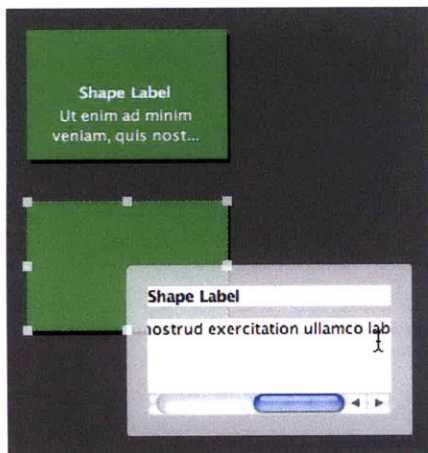


FIGURE 5-7 A demonstration of the label editor, as well as the display of labels.

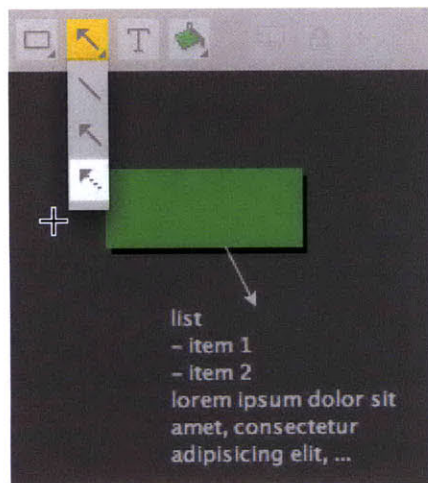


FIGURE 5-8 A demonstration of connectors and textboxes in Kaleido.



TEXT BOX

Creates a text box with a transparent background that holds a description.




COLOR FILL

Select this tool and click on a shape to fill it with the selected color. The currently selected color will be used to fill newly created shapes.




CODE WINDOW

Opens code windows on selected shapes that have been linked to code.  will hide the code windows of the selected shapes.




LOCK

“Locks” drawn elements to prevent them from being moved or resized. Elements can be unlocked via the  button. Drawn elements that are in locked mode will appear as if they are pinned to the canvas, while unlocked elements drop a shadow and appear as if “floating”.



LINK

Initiates a link between visuals and code, starting with the current selection in either drawing area or text editor. If a link currently exists (each drawn element can be linked only to one code fragment at a time, although any given code fragment can be linked to multiple drawn elements), the link can be removed by using the  button.

Locking

An implementation of a dimension of ambiguity on discrete predefined visual elements, a locking functionality was implemented, allowing users to individually lock visual objects to prevent them from being affected by editing functionalities. Visually, unlocked objects dropped a shadow and appeared to float on the canvas whereas locked objects, without any shadow, appeared to be glued directly onto the canvas. The locking functionality, in addition to being a convenience when manipulating multiple objects in a single vicinity, enables the user to make a distinction between visual elements which they felt were comparatively finalized versus elements which they were uncertain about. While ambiguity in sketches is important for enabling the artist to discover additional ideas, the locking functionality here is limited by its implementation of only a binary state of ambiguity.

Code Windows

Once a shape is linked, the user can open a separate code window on the shape to access the associated code. Code windows pop up on linked shapes and contain removable and resizable mini-code-editors (Figure 5-9). A code window can be opened on any link at any time, allowing the user to access code obscured from the small window that is the main text area. This enables a non-sequential and user-customized pattern of accessing code.

Text Area

The text area is the main code editing area, which functions in the exact same way as traditional programming text editors (the text area implementation is predominantly inherited from the Processing implementation), with the exception of the code margins. The code margins serve as an indicator of links. When a piece of code has been linked to a shape, the line of code will be marked on the margin with a strip of color that corresponds to the color fill of the linked shape (Figure 5-10). The text area can contain multiple tabs, each holding a code file within the project.

Visual-Textual Connections

Linking and Unlinking

The linking interface button was designed to function under multiple scenarios, including linking text to code, code to text, code to a new shape, as well as an unlinking functionality for shapes with existing links. Also, since linking is a multi-step action (user must select at least one shape and one area of text), it is imperative to give an indication when the interface recognizes that the user is in the midst of a multi-step action. Thus the link button also serves as an indicator of the link status (Figure 5-11).

Synchronization and Integration

It is critical to ensure that visual-textual connections are made clear and helpful to the programmer as he or she is at various stages of programming process, be it writing, editing, and debugging. This is a particularly important design issue since the mapping from visual to text is not exhaustive nor absolute in individualistic visual interfaces.

The first indication of a link is the selection synchronization, which occurs at every user action (Figure 5-12). Between the visual and the text sides, the selection that was actively created by the user highlights in the primary selection color (in this case, yellow) while its linked counterpart on other side highlights in the secondary selection color (gray).

Pronounced visual indicators in both visual and code side clearly demonstrate an entity's linked status at all times with minimal visual disturbance to the visual or textual appearance. On the visual side, linked shapes are filled with a bright shade of color, while linked code is marked with the associated shape's fill color in the code margin.

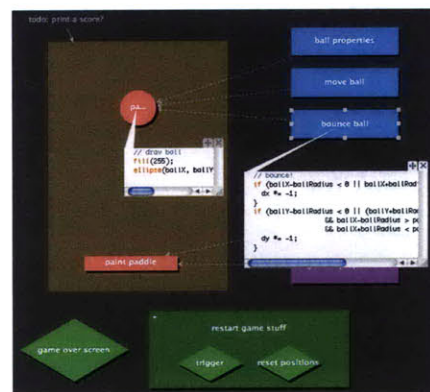


FIGURE 5-9 A demonstration of code windows, which can be freely moved and resized by the user.

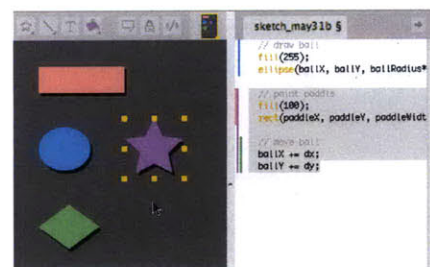


FIGURE 5-10 A demonstration of code margins to indicate a connection between visual and text.

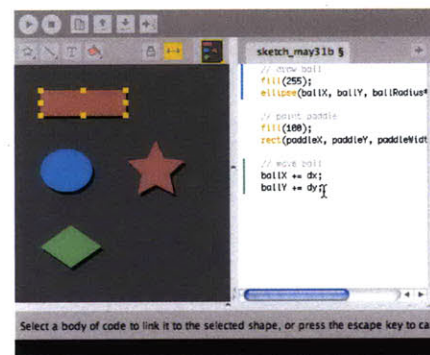


FIGURE 5-11 A bright yellow link button indicates that the user is currently in the process of creating a link.

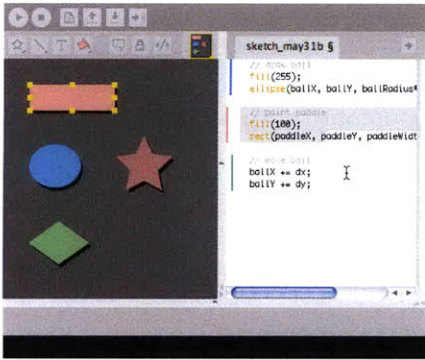


FIGURE 5-12 Synchronized selection across visual and textual representations of the program.

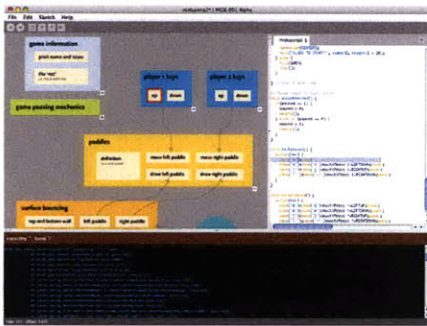


FIGURE 5-13 An early-stage mock-up to integrate connections with debugging mechanisms.

Finally, designs were made for visual-textual associations to be also integrated with debugging. Ultimately unimplemented, these included matching the font color of error printouts with the fill color of the shape linked to the culprit code, an analogous “visual error printout” highlight in the visuals, etc. (Figure 5-13).

Operations on Linked Elements

The following table describes the results of performing certain common operations upon linked shapes. In the current implementation, for all cases the link is destroyed in the process; this decision was made in the interest of both ease of implementation and predictability of behavior, particularly since the way links should be processed when copy-pasting across multiple files is not immediately obvious. However, as noted in the next chapter, in some scenarios users wanted to maintain the connection through the operation (see page 88); thus, future implementations should include options for a connection-preserving versus a non-preserving copy function.

SOURCE	DESTINATION	RESULT
linked shape	drawing area	a duplicate of the shape
linked shape	text area	a duplicate of the code to which the shape was linked
linked text	drawing area	a new text box containing a duplicate of the text
linked text	text area	a duplicate of the text

Keyboard Shortcuts

Keyboard shortcuts were implemented to help users work with the interface more fluidly. With the exception of a few more advanced graph-navigation functions, all other keyboard shortcut actions are also accessible from the right click pop-up menus or from the menu bar at the top.

⌘ Z	UNDO
⌘ Y	REDO
⌘ X	CUT
⌘ C	COPY
⌘ V	PASTE
	Pasting text into the drawing area will create a new text box to hold the clipboard contents. Pasting a linked shape into the code editor will paste a copy of the code linked to the copied shape.
⌘ A	SELECT ALL

⌘ D	SELECT NONE
⌘ /	COMMENT / UNCOMMENT
⌘]	INCREASE INDENT
⌘ [DECREASE INDENT
⌘ F	FIND...
	This search function currently only searches the code. Search functionality for the drawing area is planned for future releases.
⌘ G	FIND NEXT
⌘ T	OPEN CODE WINDOW
⇧ ⌘ T	CLOSE ALL CODE WINDOWS
⌘ L	LOCK SELECTED ELEMENT
⇧ ⌘ L	UNLOCK SELECTED ELEMENT
⌘ +	ZOOM IN
⌘ -	ZOOM OUT
⌘ 0	ACTUAL SIZE
⇧ ⌘ drag	PANNING
<spacebar>	EXPAND ELEMENT (IN NESTED ELEMENTS)
<backspace>	COLLAPSE ELEMENT (IN NESTED ELEMENTS)
<delete>	DELETE ELEMENT
F2	EDIT ELEMENT DESCRIPTION
↑	SELECT CONTAINER ELEMENT (IN NESTED ELEMENTS)
↓	SELECT CONTAINED ELEMENT (IN NESTED ELEMENTS)
←	SELECT PREVIOUS ELEMENT
→	SELECT NEXT ELEMENT

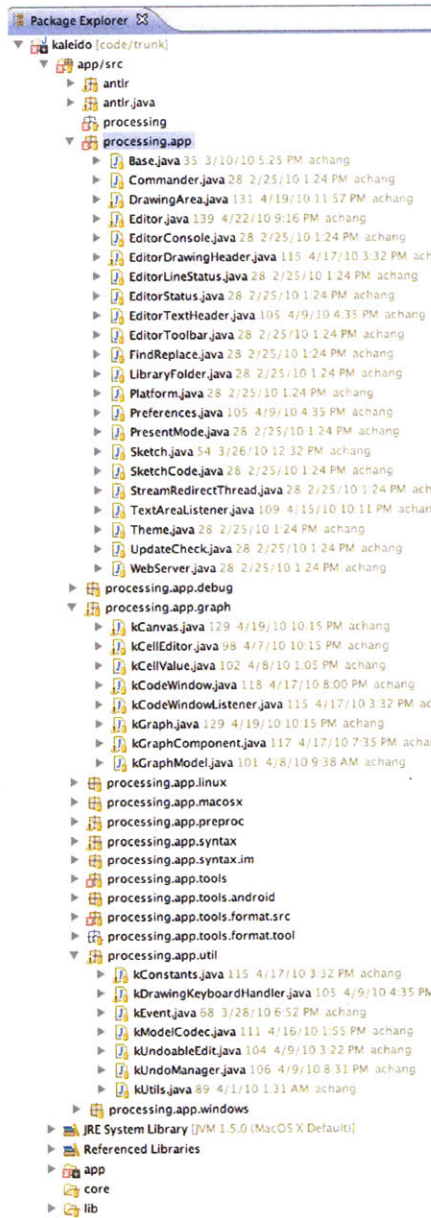


FIGURE 5-14 Package organization for the Kaleido code base.

Developer's Manual

Libraries

The current system is primarily based on the Processing Integrated Development Environment (IDE)[5], available under the GNU General Public License, and the drawing component is currently implemented via a version of the JGraphX library[2] that is also available under the GNU Lesser General Public License.

The bulk of the source code is from the Processing project, and the Java packages still retain their original names. As such, the system depends on Java 2 SE 1.5.

The graphing library upon which the Drawing Area is built is an open-source graphing library whose Java distribution (i.e. the one used here), nominally referred to as “JGraphX”, is in reality a port of the “mxGraph” JavaScript library written by the same developers. The Java package names of the graphing library is thus named “mxgraph”. To minimize confusion, the remainder of this section will refer to the graph library as mxGraph.

The mxGraph library was developed with a basic graphing tool in mind (Figure 5-15), and thus for Kaleido’s purposes it was chosen over alternative Java graphing libraries that focused on creating charts (i.e. bar charts and pie graphs) rather than flowchart-style graphs.

Kaleido Setup

The main Java package of Kaleido is still Processing’s `processing.app`. Significant changes were made to the `ProcessingEditor.java` class to incorporate the `DrawingArea` component, which is a `JDesktopPane` that holds `JInternalFrames` (in order to implement code windows). The drawing area visuals are all part of the `kGraphComponent` class, derived from `mxGraphComponent` in the `com.mxgraph.swing.mxGraphComponent` library. This is placed in an internal frame at the base-most layer and maximized at all times within the `JDesktopPane`. Meanwhile, each code window is a combination of three internal frames, all of which are also placed inside the `DrawingArea`.

Aside from the `DrawingArea`, which was placed in `processing.app` to be on the same hierarchical level as the `Sketch`, all Kaleido-only classes are placed in `processing.app.graph` and `processing.app.util`. Kaleido classes generally extend `mxgraph` classes, leaving the bulk of the mechanism declared within the original graph library while only containing minimal Kaleido-purpose customizations.

UI theme and color settings etc. use Processing settings whenever possible. For drawing area elements, rather than mimicking Processing's text file approach to storing settings, the Java class `kConstants` stores all fixed values for Kaleido (with the unfortunate drawback that none of it can be changed by the user).

The `mxGraph` library comes with a robust event handling system which Kaleido adopts. Kaleido classes for this purpose as bundled in the `processing.app.util` package. Processing on the other hand, has no event handling (tracking text selection changes accurately, for example, proves tricky).

File I/O

Kaleido saves project files in the same way as Processing, namely in a folder named after the sketch, with `*.pde` text files storing the code. The drawing is stored in an `*_graph.xml` file in the same folder. In this way, the user may open and edit Kaleido project files in Processing, and vice versa. In the case of using the Processing IDE to open a Kaleido project with a drawing, the drawing will simply not display and the user cannot edit it, but the stored file will remain.

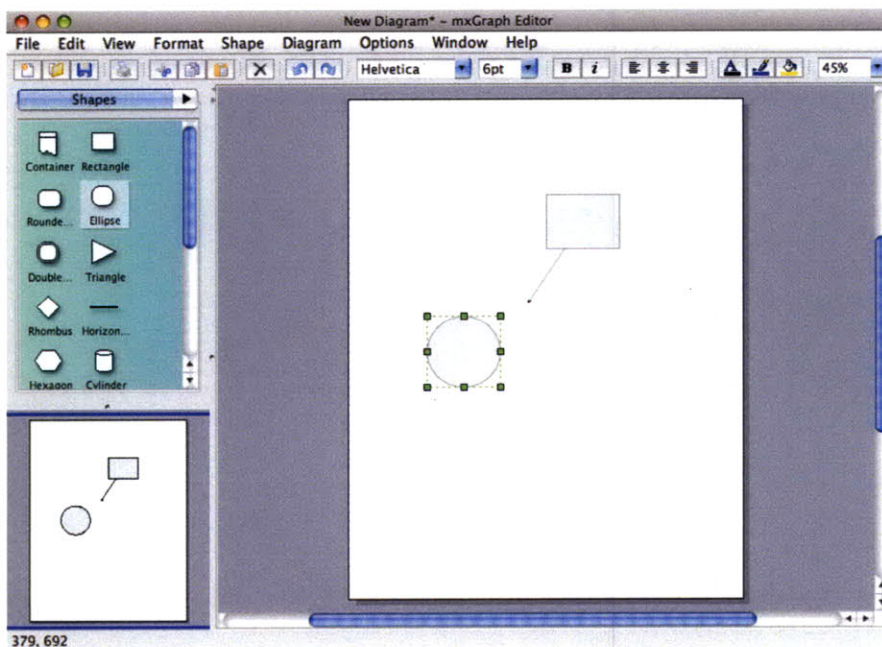


FIGURE 5-15 The JGraphX[2] graph editor example program.

Start with a problem, forget the problem, the problem reveals itself or the solution reveals itself and then you re-evaluate it. This is what you are doing all the time.

Paul Rand

Evaluation

In this chapter I evaluate the strengths and weaknesses of Kaleido by reviewing users' experiences with the prototype. An initial evaluation was conducted on a small-scale. I describe my observations of a small number of users with varying programming backgrounds working directly with the prototype, and the results of my interviews with them individually. Kaleido was also made publicly available on the Internet and was promoted in various similar-interest online communities, and the general response was positive.

Approach

Since the prototype is in the early stage of development where it is difficult to predict users' reactions, to avoid making assumptions by imposing quantitative measures I took a qualitative approach to evaluation. The primary goal of the project was to improve people's coding experience, a measure I deemed more easily evaluated via free-form responses from individuals, rather than variously-interpreted quantitative metrics. Further, since one of the purposes of the evaluation is to discover how users appropriate the tool in ways the designer (myself) did not expect, I did not think that a quantitative study would be able to capture such user responses.

Feedback was primarily gathered from observation and interviews with a small number of users during individual sessions, working in Kaleido on specific tasks I designed for the purpose. I will refer to this part of the evaluation as the "Alpha Study". Aside from the Alpha study, I also exchanged conversation and files with a few individuals who worked on tasks in Kaleido independently. Finally, to obtain feedback from the general community, a demo video, website, and documentation for the project was published online. A general Kaleido experience survey that was

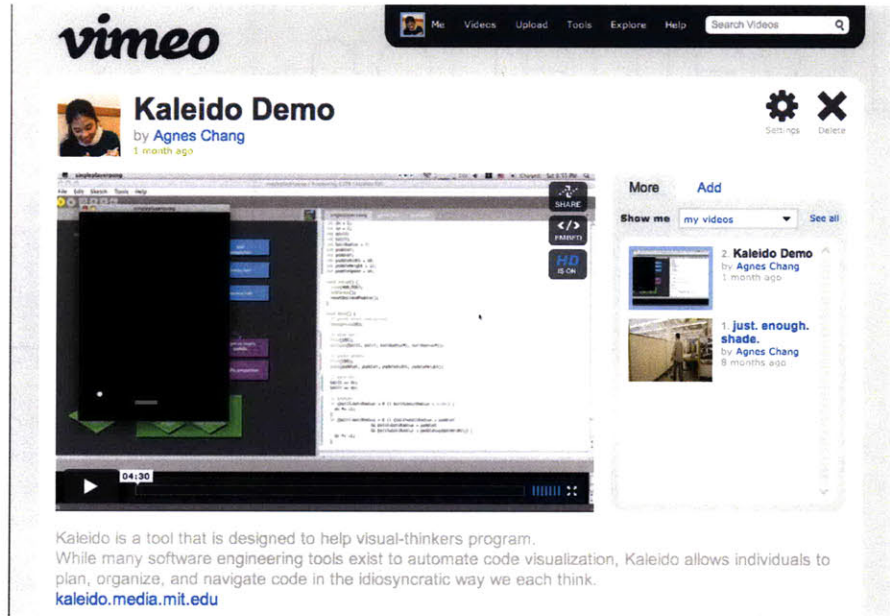


FIGURE 6-1 The vimeo video page of the Kaleido project.

given to Alpha Test interviewees was also advertised online and a small number of internet visitors participated. In this way, some user response was also gathered from internet social media sites on which the project was publicized, such as vimeo, (Figure 6-1) twitter, and the forums of various online communities with shared interests.

Alpha Test Methodology

For the Alpha Test, I sat down individually with each user for approximately an hour per person. For their first of two tasks, users were asked to use Kaleido starting from a blank file, and try to implement a two-player Pong game. The goal of this task was to observe how users might adopt the tool to their own workflow. The time allotted to this task was about 40~50 minutes, but users were asked to stop whenever they felt they had developed a good sense of how the tool might fit into both their short-term and longer-term workflow habits, and could discuss it with me. Most users did not complete the first task.

For the second task, users were instructed to make a small modification to a relatively complex program that was accompanied with a set of visuals I had created beforehand (Figure 6-2). The goal of this task was to study how the accompanied visual affected the user's ability to comprehend a complex program that they did not write. This task was allotted 10~15 minutes, and again users were asked to discuss their reactions.

At the end of the session, users were asked to fill out the same questionnaire that was made available to the public. Each of the tasks, as well as the survey, was designed to spark discussion. Screen capture and audio recording of the entire session (with the exception of a couple technical difficulties that resulted in loss of data) was archived for each Alpha Test user.

The image shows a software development environment with a flowchart on the left and code on the right.

Flowchart:

- START** leads to a **draw loop** box containing:
 - do stuff for each ball
 - check if balls hit each other
- The **draw loop** leads to a **for each ball** box containing:
 - check if balls hit wall
 - paint ball
 - calculate/run particle system
 - add a particle every other frame
- The **for each ball** box leads to a **particle system** box containing:
 - resizeable array list to hold all my particles
 - update particles (and kill the o...)
 - ways to add particles
- The **particle system** box leads to a **particle** box containing:
 - calculate my vector
 - draw myself (ellipse)
 - draw myself (vector)
- There is a note: "otherwise we'd get too many" pointing to the "add a particle every other frame" step.

Code:

```

ballparticles  Ball  Particle  ParticleSystem
//
// Adopted from Processing examples
// Simple Particle System
// by Daniel Shiffman
// and
// Circle Collision with Swapping Velocities
// by Ido Greenberg.
//

Ball[] balls = {
  new Ball(100, 400, 20),
  new Ball(700, 400, 30)
};

PVector[] vels = {
  new PVector(2.15, -1.85),
  new PVector(-1.65, 2.42)
};

ParticleSystem[] ps = {
  new ParticleSystem(1, new PVector(width/2,height/2,0)),
  new ParticleSystem(1, new PVector(width/2,height/2,0))
};

void setup() {
  size(640, 360);
  colorMode(RGB, 255, 255, 255, 100);
  smooth();
}

void draw() {
  background(0);

  for (int i=0; i<2; i++){
    balls[i].x += vels[i].x;
    balls[i].y += vels[i].y;
    ps[i].run();
    if (frameCount % 2 == 0) {
      ps[i].addParticle(balls[i].x,balls[i].y);
    }
    fill(180,180,180,60);
    stroke(180,180,180,60);
    ellipse(balls[i].x, balls[i].y, balls[i].r*2, balls[i].r*2);
  }
}
    
```



FIGURE 6-2 The “ball particles” program and output used for the second task in the Alpha Study.

Users were asked to watch the Kaleido Demo video beforehand so that they gained a general understanding of the tool's capabilities prior to working with the interface. At the beginning of the session, users were requested to do whatever they usually do when programming — whichever way they usually approach tasks. Users were asked to keep a running commentary about their thoughts as they worked, e.g. what they were trying to do, if they found that a particular button wasn't working the way he/she expected, etc.

To allow users to adhere to their habitual workflow as much as possible, they were also allowed external resources: I offered the Processing online reference, and users were free to access Google and the rest of the internet. Meanwhile, I also acted as a reference to save time. Subjects were free to ask me what different parts of the interface were, or if they wanted to do something how they could achieve it (i.e. which combination of button presses), but I would not actively offer suggestions for what they could do (i.e. link code). They were also free to ask me Processing syntax questions (i.e. what the four parameters of the `ellipse()` command were).

Users were all students at the Media Lab with diverse backgrounds ranging from a user who has programmed over 20 years to a user who started learning programming two years ago. A couple of users were also participants of the earlier Kaleido preliminary investigation (in which participants were also asked to think about a two-player Pong game), but these two studies were conducted four months apart.

Alpha Test Results

First User: Karen

To the first interviewee, whom I shall call Karen, maintaining a consistency in her drawing was important. As she started her first task, one of the troubles she encountered was deciding what each of the visual elements should mean. She said,

At the beginning, with that rectangle, I couldn't make up my mind, what that should represent, because there was no constraint of what type of thing they're supposed to be, so I decided to snap onto the flowchart model, because I want to get this working so I don't want to come up with a whole new scheme... so just relying on that existing sort of notion.

As she discovered new functionalities of the drawing tool, she would go back over her drawing and update them to utilize the new expressions (“I want to make sense of the two labels...”) At the end of her first session, every shape she had created was linked.

Karen found the interface was most useful when she used it for navigation. For her second task, Karen immediately navigated to the method `checkBoundaryCollision()` by using the rectangle labeled “check if balls hit wall”. She first modified the drawing, and then wrote the code to finish the task. She finished very quickly.

A point of frustration occurred for Karen when she discovered that the drawing interface was too “slow” to use for ideation; drawing one shape at a time was not immediate enough to jot down thoughts that were moving rapidly at that point in the workflow. Instead, she suggested using some auto-generation to create many shapes at once which the user can then layout and modify:

It would be great to allow users to make the diagram faster... type a [bulleted] list that is the flow of the program that will generate a diagram for you and then you can drag it out.

As a reference, she suggested the SmartArt functionality available in Microsoft Powerpoint.

As she revealed in the survey, Karen thought that the visuals were extremely helpful for navigating your code, very helpful for reminding yourself of the program structure, and barely helpful for planning the program logic. She thought she would use the tool more for teaching than ideation or sharing with others.

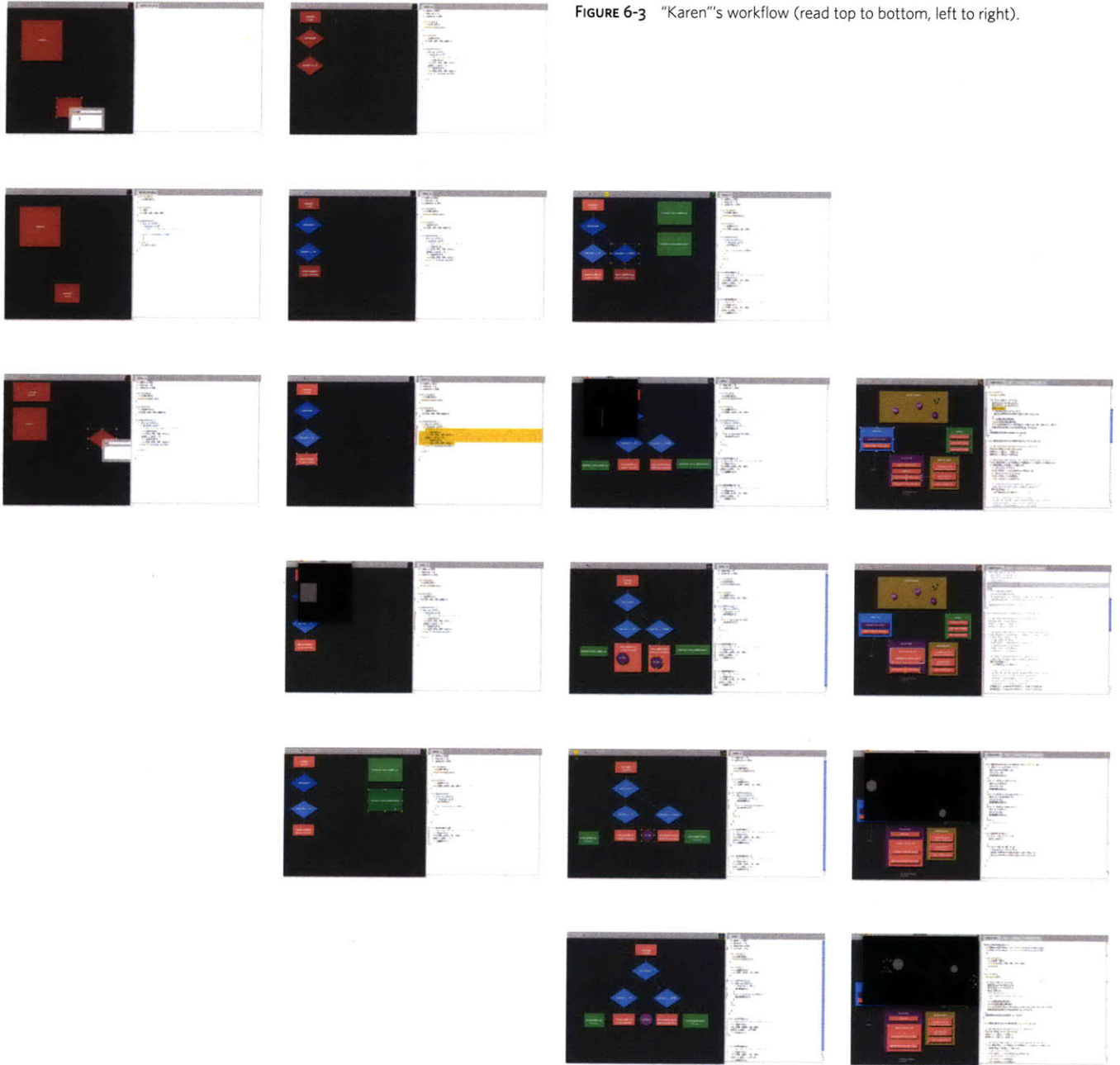


FIGURE 6-3 "Karen"'s workflow (read top to bottom, left to right).

Second User: Will

“Will” is a software engineer. He started his first task by making a rectangle, but he did not make any more activity on the drawing side by the time he finished the programming task. Neither did he make any comments in his code. During our discussion, Will explained that he did not avoid the visuals on purpose, but rather once he started writing code, he got into his usual workflow, and focusing on the task, forgot that the drawing side was available to him.

The thing is, it's a habit – if you are already very comfortable with programming, like I've been programming for like 90% of my life, then I won't use the visuals...

He said, however, that it was also his habit not to do any planning before he wrote code, but just to write methods as they occurred to him. He said he usually works on documentation as a separate phase after coding. He didn't claim to be a visual-thinker, but he thought that on a longer-time scale he could see himself adopting the tool for documentation. He said he often will restructure and organize his code by starting over with a blank file and copy-pasting the old parts back as needed, and he could see himself using something like Kaleido in his restructuring process. He did say, however, that he wished that the visuals were more informative regarding the program structure, e.g. he wanted some way of distinguishing between a method definition and a call to that method, and also some way of seeing whether something was iterative or recursive.

In his second task, Will again finished in record time. He found the visuals hugely effective in helping him navigate the program. He thought that visuals for navigation purposes doesn't need as much structural accuracy or detail to be very effective, but for writing purposes the lack of structural description in the visuals is prohibitive.

One additional suggestion Will had was that it would be convenient to be able to generate text boxes (since they appear like comments) from comments typed in the code. A special markup such as “///” could denote a comment to be synchronized and linked with a text box in the drawing area. In his survey, Will said that the visuals were extremely helpful for navigating your code, very helpful for reminding yourself of the program structure, and barely helpful for planning the program logic. For him, the tasks for which Kaleido was most suited was for documentation and sharing with other people, and least suited for ideation.



FIGURE 6-4 "Will"'s workflow (read top to bottom, left to right).

Third User: Tori

“Tori” was a user who used the visuals concurrently with writing code – she would make a few shapes, then write code; adjust those shapes, then adjust the code, etc. Like Karen, she put a lot of thought into what each visual element meant (“these are both paddles so I want them to be the same size, same shape, and same color.”) She found she could use the visuals to replace commenting:

Oh, I don't have to comment as much, because I was going to put a comment here but I have this already.

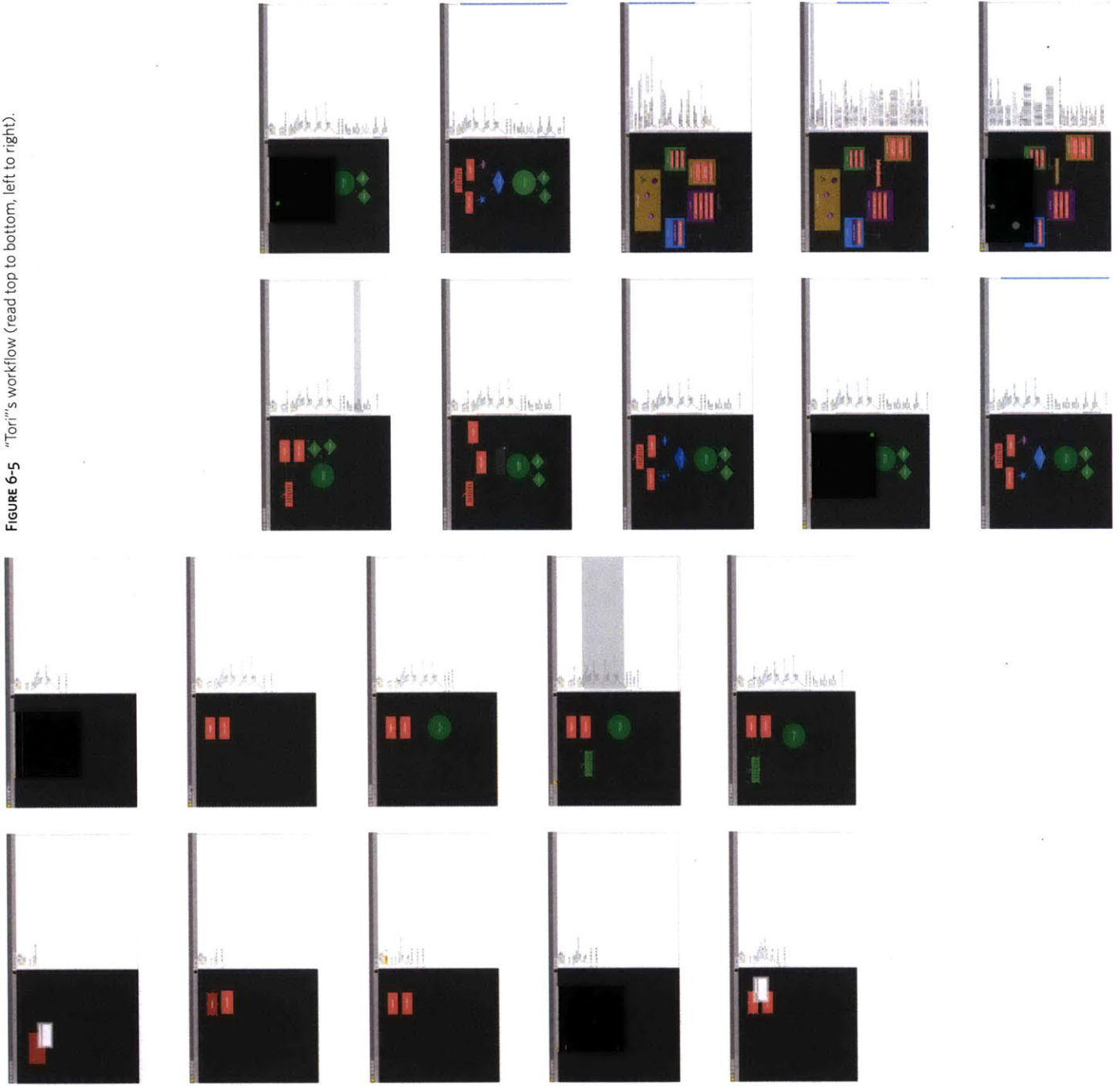
In her second task, Tori's method of using visuals became an obstacle when she tried to work with another person's visual system:

I think the visuals are very helpful if the author was also there to explain them, but I don't think it can stand-alone as, like, a “read me” documentation, because the visuals were different from what I expected so it ended up causing more confusion.

Tori also repeatedly ran into the situation where she wished to copy-paste linked shapes or linked code and maintain the link information through the process; however, in Kaleido's current implementation, all copy-paste operations lose the object's link information in the transfer process (see Implementation chapter, page 74).

Thus, unlike Karen and Will, Tori concluded that visuals were extremely helpful for planning the program logic, very helpful for reminding yourself of the program structure, and barely helpful for navigating your code. Tori thought she would use Kaleido primarily for ideation, documentation, and sharing with project collaborators.

FIGURE 6-5 "Tori"'s workflow (read top to bottom, left to right).



Fourth User: Sam

Sam started his first task immediately by illustrating the various visual components of the Pong game in the drawing area. He added descriptions of each object's (functional) behavior in text boxes alongside, and then proceeded to write those same behavior descriptions as comments in the text area. It was then that he wished the two would automatically synchronize:

If I'm using pseudo-code on the left side and writing the same thing as comments on the right side, then maybe they could just go over automatically

After finishing the comments he proceeded to code, using the comments as guidelines, and did not make any changes to the visuals for the rest of the task. He also thought that this wasn't a conscious decision of his, and he reasoned that:

... maybe because I planned it all at the beginning I didn't need it anymore... but also I think writing pseudocode as you program is a habit that you have to train yourself to do, so this is like that, you have to train yourself to use it.

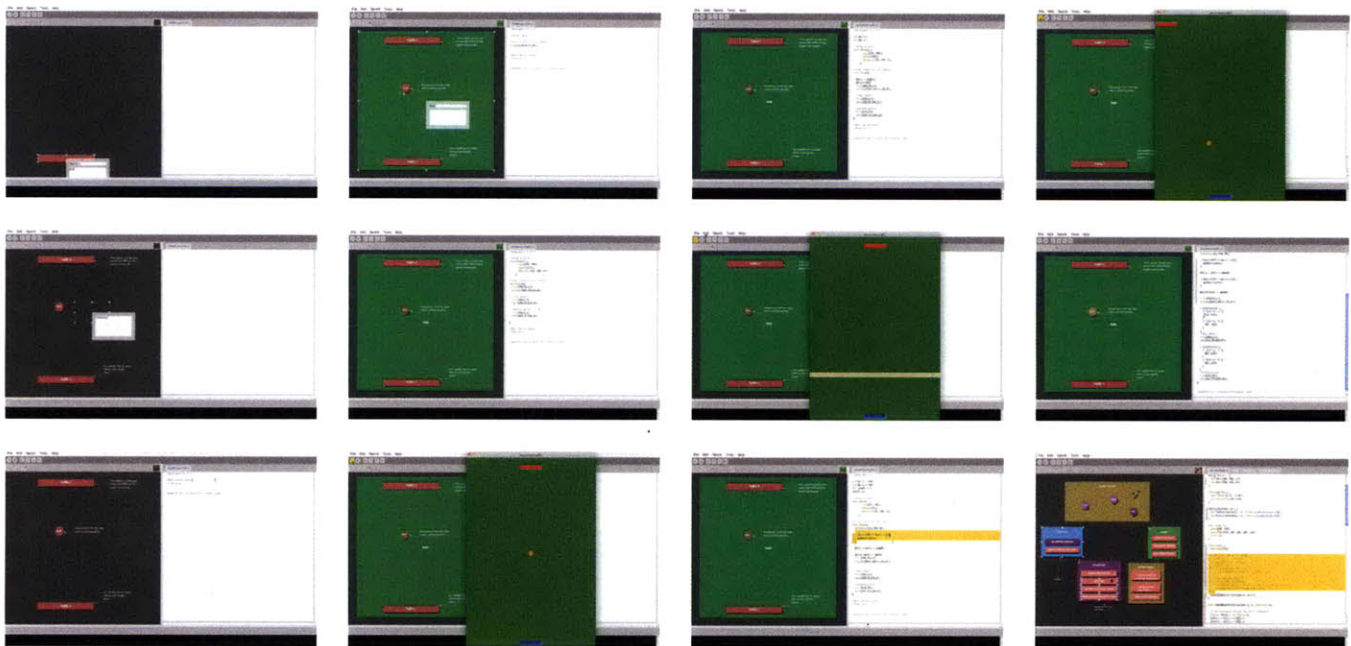


FIGURE 6-6 "Sam"'s workflow (read top to bottom, left to right).

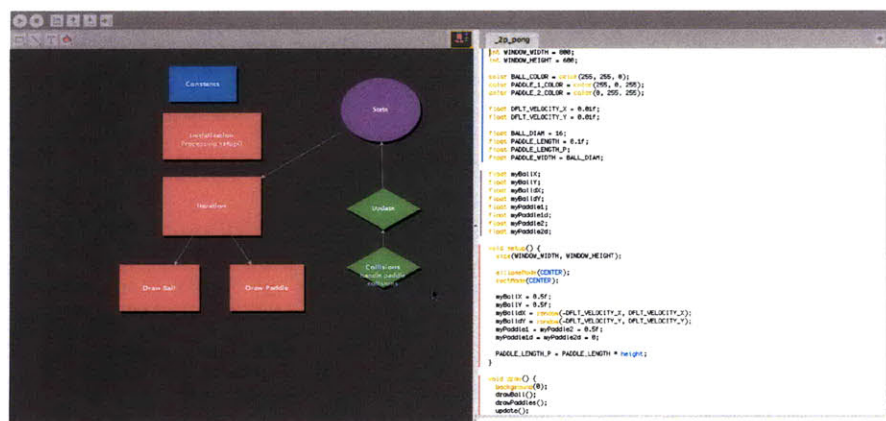


FIGURE 6-7 “Phil”’s final version of Pong.

Fifth User: Phil

Phil started by creating and labeling a rectangle “Constants”, and then wrote a fair amount of code before returning to update his drawing. He also wished for auto-generation and synchronization of documentation between visual and text:

It would be nice if [when] I entered a comment on the right side, it would show up on the left side and then I can position it... because I want the same thing on the left side as the right.

In the survey, he summarized his experience:

I used the visuals as a form of documentation of the conceptual structure of the program. By the prominence of the visuals alongside of the code, I felt compelled to keep the visual representation up-to-date with the code as I worked. This is very different from how I usually approach code documentation (e.g. Javadoc), which is to go back and add it at the last moment before giving the code to others. While working, I considered that the visual representation would provide not only useful documentation to others, but to myself if I were to revisit the code in a couple of days’ time or longer. If the code in the task given became substantially longer, the visuals would have made an excellent navigation tool. As it was in the first task, having only one class that was short enough to scroll easily, I felt like I was navigating mostly in the code view. In the second task, I thought the visual representation was essential to quickly apprehending someone else’s code (a task which I usually find tedious and very unpleasant). In this case, I used it

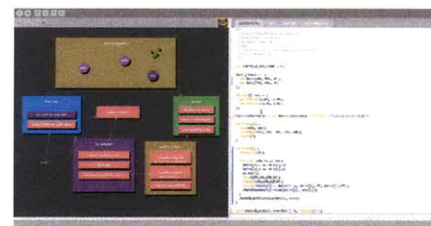


FIGURE 6-8 “Phil”’s final version of the “ball particles” program.

as a quick glance of the structure and behavior of the code, rather than reading through all of the classes before even getting started tackling the problem. Having the visual be a navigational index into the code was invaluable.

Phil concluded that visuals were extremely helpful for reminding yourself of the program structure, very helpful for navigating your code, and somewhat helpful for planning the program logic. Thus he thought he would use Kaleido primarily for sharing, documentation, and learning, but last for ideation and teaching.

Other Methods

Via twitter, the Kaleido demo video spread quickly among the digital media community and sustained a couple hundred viewers per day for the first week. The project was also publicized on the forums of Processing, openFrameworks, and Arduino. General comments on the video and concept were very positive, although the website showed low traffic on the Download page, and no activity on the Kaleido Forum.

Through a mailing list for Processing Educators, educators responded very positively, and were generally excited about the prospect of using Kaleido to explain code to students. As one educator who tried Kaleido independently and filled out the survey said:

I am very interested in using it to explain examples to students. I would love to provide it to students as a tool so that they can see what code links to what object without further instruction.

Limitations of the Study

Further methods of gathering feedback, while beyond the scope of this thesis, would be based upon long-term evaluation. First, it would be ideal to conduct workshops and work with educators to introduce the tool. The nature of Kaleido is such that, as part of an individual's workflow which is guided by habit, its usage, mastery, and thus full-potential, can only be acquired through practice (compare the acquisition of the habit of writing pseudo-code in the programming process). Thus, in order to overcome initial user reluctance to break habit and adopt new tools, a more thorough evaluation method would be to introduce the tool actively and to observe users over time.

Further, the projects in the alpha study were controlled, and thus Kaleido's role in fostering higher-levels of creative ideation remains to be examined through users developing self-initiated projects. Finally, the interviews I was able to conduct

also skewed in the direction of people with significant programming experience (although often not Processing experience), rather than the initial target audience of novice programmers, so future evaluations should aim to achieve a balanced representation of user backgrounds.

Summary of Findings

It was observed, and confirmed by many of the interviewees, that people have ingrained coding habits that they've already developed over many years without use of the new tool. Thus, it is difficult to evaluate in a short time frame before people have time to adopt the tool into their workflow. Regardless, educators demonstrated particularly positive response about the potential of the tool to explain programming to novice programmers.

Most users found the drawing immediately effective for use in modifying other people's code. However, users also generally wished for faster ways to create visuals, including semi-automated methods (e.g. Figure 6-9), saying that more immediate visual-creation would raise Kaleido's potential as an ideation tool. One user thought that a more inductive UI design (i.e., inviting, proactive, "draw something here!"-sort of UI) would help make the visual-creation process smoother.

General users, meanwhile, differed widely on their opinion of the best use of tool. Some users thought the tool was most suited for ideation, and worst for navigating (because they had a certain idea in their head of what each visual should mean, and another person's drawing just confused them); while other users thought the opposite (because it's not fast enough for ideation, or the lack of precision in structural description doesn't allow you to plan.)

That fact that there were no guidelines or conventions for mapping visuals to meaning perplexed some users, while other UI functionality designs such as the shape label editor trigger and the unpredictable effects of copy-paste operations on linked shapes posed difficulties to multiple users. Finally, a few users also inquired about creating custom shapes by combining multiple shapes to act as one, but in general users did not seem to experience any dissatisfaction with the tool's ability to create the graphics they wanted.

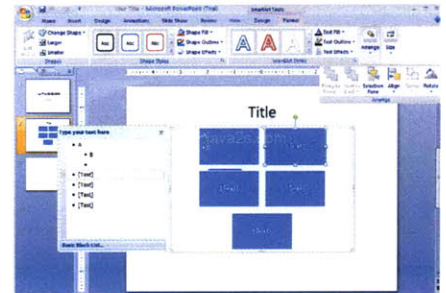


FIGURE 6-9 Microsoft PowerPoint's SmartArt functionality that quickly generates diagrams from bulleted lists, etc. This was suggested by one of the study participants to improve Kaleido's utility as an ideation tool.

The real voyage of discovery consists not in seeking new landscapes but in having new eyes.

Marcel Proust

Conclusion

In this chapter I summarize the findings and contributions of this work, and I discuss points for improvement of the current Kaleido system. While certain usability issues must be resolved before Kaleido can fully support the declared goals, preliminary evaluation demonstrated that integrating sketches as individualistic visual interfaces can indeed facilitate the programming process for people with a variety of programming styles. I discuss some successes and shortcomings of this work that can be generalized for the benefit of similar endeavors in the future. Further, within this topic there remains many unexplored possibilities, and I outline some alternative approaches to addressing the dissociation between individual creative intention and mechanical models of software. Finally, I conclude this work with some projections ahead.

Successes

Preliminary evaluation demonstrated that individualistic visual interfaces are effective for documentation and navigation. While a few cases occurred where a person reading another's Kaleido drawing — in essence attempting to “read their mind” — caused more confusion than if the person had simply read the code, the majority of users responded positively to the interface's potential as a navigation aid. The qualities of individuality, expressivity, and connectivity, as outlined in the design guidelines for individualistic visual interfaces, were successful in supporting these activities.

Regarding expressivity in particular, the visual vocabulary made available in the drawing area appeared successful. Users did not express discontent or feelings of

being restricted despite the limited range of visual elements, and drawing compositions reflected a variety of styles.

Overall, while users differed widely on their opinion regarding the best use of the tool, everyone found at least one functionality very helpful to their way of working. In this way, the interface also successfully accommodated a variety of programming styles.

General positive feedback to the project concept reveals that people have a need for tools that would support the visualization, documentation, and use of context-specific information. Although further evaluation work needs to be conducted to determine how novice programmers in particular use Kaleido, digital media educators were particularly receptive to the concept, concurring with the belief that inclusion of context-specific information could overcome some of the difficulties novice programmers have in comprehending computer programs.

Shortcomings

A number of improvements can be made to the Kaleido interface; indeed, while the current release is stable, certain refinements are prerequisite to bringing the software prototype to a stage where its usability fully supports the software goals. However, more importantly, certain design guidelines for individualistic visual interfaces outlined in Chapter II were not implemented in Kaleido, while others need improvement. While suggestions for improvement to methodological aspects of this project have been incorporated into their respective chapters in this document, in this section I summarize the design shortcomings of this work and identify areas of further development.

Perhaps the most critical aspect in which Kaleido fell short of expectations was the immediacy of the visual-creation mechanism, and thus, the reflectivity of the interface. Evaluations revealed that the effectiveness of individualistic visual interfaces in supporting ideation, which was one of the original motivations for this work, is highly dependent on how quickly users can create the visuals they want. Kaleido currently implements a mouse-drag-and-buttons paradigm of producing visuals that was modeled after popular graphics editing applications. However, this paradigm was found to be too unwieldy by most users to keep up with rapid trains of thought, and alternative methods of creating visuals, such as those discussed later in this chapter, need to be explored for future implementations of this work to fully support ideation.

A related issue is that users expressed discontent with the visual-creation mechanism when they wished to create information duplicated in both drawing and text areas. Kaleido's lack of support for any visual auto-generation, while purposeful, caused users to find repetitive operations tedious, e.g. duplicating drawing annotations as code comments, or vice versa. It should be noted that influencing factors may include the difficulty of visual creation as well as habit fostering the need for information to be duplicate across code and visual. In any case, this led to user suggestions that special commenting codes, e.g. `///
"` at the beginning of a line, be enabled to automatically generate a basic shape.

Transparency and simplicity appeared well-received as users quickly mastered usage of the interface, and once familiar, did not express confusion for the remainder of their use of Kaleido. Spatiality proved useful for the purposes of documentation and navigation; however it has not yet been completely evaluated since users did not create complex or large enough drawings to require visual navigation.

While connectivity proved successful in helping navigation, it is also an aspect in which Kaleido could be developed further so that linkages are integrated with more parts of the programming process. For example, the performance of familiar object operations such as cut-copy-paste upon linked objects versus unlinked objects (see Evaluation chapter, page 95) needed further refinement while the use of the visual representation for debugging (e.g. Figure 5-13 in Implementation chapter, page 74) was not fully implemented.

Qualities that were unable to be implemented in Kaleido and remain for future work include the history-keeping functionalities to support reflectivity, as well as ambiguity. Complementing the decision to implement Kaleido with discrete graphical elements was the decision to simplify the dimension of ambiguity to be represented as a binary state of "lock" versus "unlock" (see Implementation chapter, page 72). This aspect of the drawing was not used by users in the evaluation, most probably due to the short amount of time spent on the tasks.

Finally, further work is needed to gather more conclusive evidence about the successes and shortcomings of Kaleido. Long-term evaluation, particularly with novice programmers, self-initiated projects, and in classroom contexts, would shed light on the roles in which an individualistic visual interface assist in the creative coding process.

Future Work

Kaleido is one implementation of an individualistic visual interface. Based on the same set of design guidelines, many other possibilities exist. The following section explores alternative approaches to fostering the vision that, “As a carrier for pluralistic ideas, the computer holds the promise of catalyzing change, not only within computation but in our culture at large.”[64]

Gestural Input

Kaleido’s current implementation could immediately benefit from using a pen input system to create the visuals; for example, a rectangle gesture with the pen would create a rectangle shape, an ‘X’ gesture could delete the shape, etc. As users of Kaleido revealed, the current method of visual creation via tool buttons lacked immediacy and hampered the expression of their thoughts. With the use of a pen tablet or a touch-input device, a gestural input system could dramatically improve Kaleido’s utility as a sketching and ideation tool.

Free-form Drawing

While Kaleido’s adoption of discrete graphical elements is arguably easier to comprehend, manipulate, and use as an interface, a partially-restricted form of pen drawing could preserve some of the benefits of discrete visual elements while allowing more dimensions of individualistic visual expression. Pen pressure, angle, etc. could be used to generate visual distinction between shapes. The drawing interface of the CrayonPhysics game[10], for example, allows the user to create a distinct shape with its own behavior (in this case, physics) as soon as the user closes the loop on any single line. Further, as the game demonstrates, the visual appearance of the pen input can also be partially controlled so as to yield a harmonious overall aesthetic while still allowing the user a large degree of personalization.

Specialized Graphical Palette

On the other hand, the choice of visual elements could not only be a question of “which are the ones users habitually use?”, but rather a question of “could there exist a new graphical set of symbols designed specifically for depicting mental models in the programming process?” We find it intuitive to attach meanings to flowchart symbols and stick figures simply because most of us have acquired familiarity with the conventions in which they are used. While a new palette might be difficult to adopt in the early stage, a novel graphical system designed to depict both individuals’ intentions as well as program architecture could potentially reduce mental bottlenecks in the programming process.

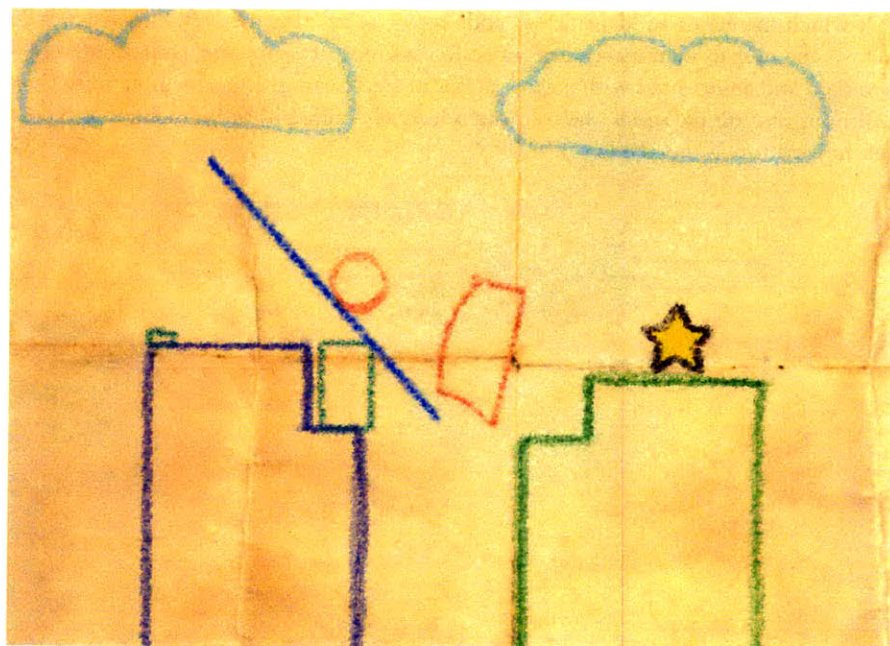


FIGURE 7-1 A screen shot of the Crayon Physics game. The player draws shapes that behave according to physical laws (gravity, mass, momentum, etc.) to solve the puzzle. From [10].

Visual Overlay of Drawing and Code

Finally, in Kaleido's current interface, the visual information is segregated from the code in two distinct panels, while the connections between these two layers of information lack a direct visual representation. An alternative approach to recording, presenting, and navigation between individualistic visual information and text-based structural information is to overlay these two layers of information visually on top of each other.

Since a significant portion of the programming process involves navigation between disconnected locations in code, a carefully-designed system for visual travel between these layers, e.g. zoom and pan at multiple levels, could be an effective way to express the relationships, that is the visual-textual connections, between these layers. The additional spatial dimension grants users the freedom to place not just the associated visuals but also the code itself in a personalized organization system that is restricted by neither program structure, alphabetization of file name, or output of the program. Further, an execution scope tool could select specific areas of the canvas and execute only the code within the selected area.

For example, the code body could be used to create a visual map of the entire project which maintains an identifiable "code topography" (Figure 7-2). This visual layer allows the user to identify sections of code via visiospatial memory, and it could be overlaid and augmented with a canvas layer of free-form graphical annotations. By zooming at multiple scales, the user can selectively highlight or hide different types of information as they code.

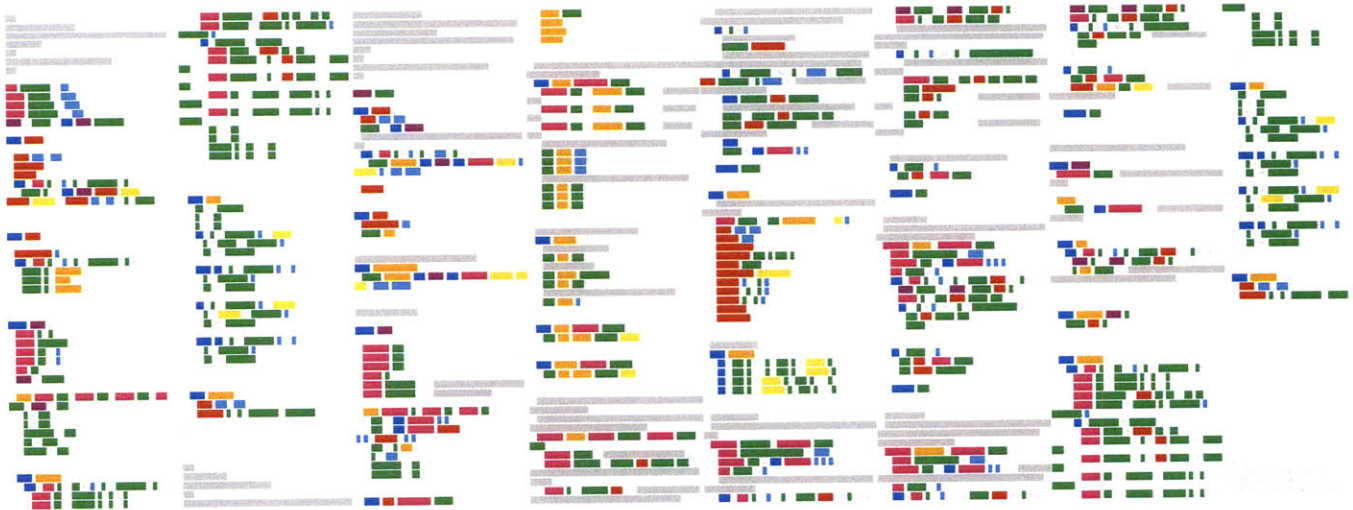


FIGURE 7-2 A "code topography" as one of multiple information layers.



FIGURE 7-3 Multiple levels of information from code, annotation, to the project landscape.

Projections

...With the atmosphere of fireflies at dusk in her mind, she imagines the dance-like path of the ball... perhaps it would appear a little hesitant as it approaches and lively as it recedes; or maybe, the hue of the sky changes to echo to the ball movement. She floats her display to the upper-left of her project space where she had placed the cluster of code fragments that represent her motion experiments. Her code floats in the background as her doodles hover in front. She nudges some aside to select her approach-and-recede doodle, and duplicates it along with its associated code. She reshapes the new doodle, giving it a slight convexity in the middle, and the ball seems to acquire a twist in its behavior. As the same time, she wonders what sort of dance would result if she mixes the underlying math functions with one of the sky variables? She does not know, but she makes a second copy, and sends the doodles back to bring the code into focus. With a few keystrokes the code is altered, and a moment later, the output reveals an unexpectedly beautiful motion spiraling and growing across the space...

Programming tools today visualize various aspects of programming activity ranging from code representation, program structure, to program execution, with the aim of making programming more intuitive. While these tools are tremendously helpful for reading and comprehending programs, the fundamental difficulty of programming remains the dissociation between the structural nature of code and the unstructured nature of individual creativity. Computer programs must necessarily be structured in order for the computer to execute them, but each person's way of thinking about their programs is their own.

Visual sketching in traditional creative disciplines such as music, architecture, and design, are a process through which the artist develops, discovers, and documents his creative ideas. In the same way, software development environments should enable users to think about code via our visual senses. By allowing users to create and associate personalized visuals with their code, we can reduce the difficulty of translating between creative concept and code structure. The goal is not to do away with writing code, or to replace visualization thereof, but rather, to realize that to truly explore the creative possibilities of computation, we need to reconsider current tools that restrict us to thinking about code in specific ways.

This thesis is about recognizing that while code is fundamentally structural, computational media is created in the same way a child builds a sand kingdom on the beach: forming, reforming, and experimenting; placing, relocating, combining, and exploring all different directions at once. It's about recognizing that, in order to fully embrace the creative possibilities of the computational medium, we should explore the many ways in which we can design tools to support this special creative process that blends the uniformly structural with the individually unique.

References

- [1] Freemind. <http://freemind.sourceforge.net/>.
- [2] Jgraph. <http://www.jgraph.com/>.
- [3] Microsoft Visio. <http://office.microsoft.com/en-us/visio/>.
- [4] Nodebox 2. <http://beta.nodebox.net/>.
- [5] Processing. <http://www.processing.org/>.
- [6] Unified modeling language. <http://www.uml.org/>.
- [7] vvvv. <http://www.vvvv.org/>.
- [8] J. R. Anderson. Arguments concerning representations for mental imagery. *Psychological Review*, 85(4):249–277, 1978.
- [9] Apple Computer, Inc. Quartz composer. <http://developer.apple.com/graphicsimaging/quartz/quartzcomposer.html>.
- [10] R. Arnheim. *Visual Thinking*. Faber and Faber, London, 1970.
- [11] J. Bertin. *Semiology of Graphics*. University of Wisconsin Press, Madison, WI, 1983. translated by William J. Berg.
- [12] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeptura, and J. J. L. Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proc. of the 28th International Conference on Human Factors in Computing Systems (CHI)*, 2010.
- [13] F. P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20:10–19, 1987.
- [14] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.
- [15] ConceptDraw. Mindmap. <http://www.conceptdraw.com/en/products/mind-map/>.
- [16] N. Crowe and P. Laseau. *Visual Notes for Architects and Designers*. Wiley, New York, 1983.
- [17] Imagix Corporation. Imagix 4d. <http://www.imagix.com/>.
- [18] Cycling74. Max 5. http://cycling74.com/products/maxmsp_jitter/.
- [19] S. P. Davies. Models and theories of programming strategy. *International Journal*

- of *Man-Machine Studies*, 39:237–267, 1993.
- [20] A. DiSessa. Models of computation. In D. A. Norman and S. W. Draper, editors, *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., 1986.
 - [21] A. diSessa and H. Abelson. Boxer: A reconstructible computational medium. *Communications of the ACM*, 29(9):859–868, 1986.
 - [22] M. Downie. Field. <http://openendedgroup.com/field/>.
 - [23] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft: a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–986, 1992.
 - [24] M. J. Farah, K. M. Hammond, D. N. Levine, and R. Calvanio. Visual and spatial mental imagery: Dissociable systems of representation. *Cognitive Psychology*, 20(4):439–462, 1988.
 - [25] J. Fish and S. Scrivener. Amplifying the mind’s eye: Sketching and visual cognition. *Leonardo*, 23(1):117–126, 1990.
 - [26] The Eclipse Foundation. Eclipse. <http://www.eclipse.org/>.
 - [27] B. Fry. Dismap (2003). <http://benfry.com/dismap/>.
 - [28] V. Goel. *Sketches of Thought*. MIT Press, 1995.
 - [29] G. Goldschmidt. The dialectics of sketching. *Creativity Research Journal*, 4(2):123–143, 1991.
 - [30] T. R. Green and M. Petre. Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.
 - [31] M. D. Gross and E. Y.-L. Do. Ambiguous intentions: A paper-like interface for creative design. In *Proc. of ACM Conference on User Interface Software Technology (UIST)*, pages 183–192, 1996.
 - [32] T. O. Group. Omnigraffle. <http://www.omnigroup.com/products/omnigraffle/>.
 - [33] T. T. Hewett. Informing the design of computer-based environments to support creativity. *International Journal of Human Computer Studies*, 63:383–409, 2005.
 - [34] P. N. Johnson-Laird. *Mental models: Towards a cognitive science of language, inference and consciousness*. Harvard University Press, Cambridge, MA, 1986. Harvard Cognitive Science Series, Vol. 6.
 - [35] E. Kleiberg, H. van de Wetering, and J. J. van Wijk. Botanical visualization of huge hierarchies. In *Proc. of IEEE Symposium on Information Visualization*, 2001.
 - [36] A. Koblin. New York Talk Exchange (2008). <http://www.aaronkoblin.com/work/NYTE/index.html>.
 - [37] M. Kölling. The problem of teaching object-oriented programming, part 2: Environments. *Journal of Object-Oriented Programming*, 11(9):6–12, 1999.
 - [38] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

- [39] J. A. Landay and B. A. Myers. Interactive sketching for the early stages of interface design. In *Proc. of the SIGCHI conference on Human Factors in Computing Systems*, pages 43–50, 1995.
- [40] P. Laseau. *Graphic Thinking for Architects and Designers*. Wiley, 3rd edition, 2001.
- [41] J. Maeda. *Design By Numbers*. MIT Press, 2001.
- [42] A. Malhotra, J. C. Thomas, J. M. Carroll, and L. A. Miller. Cognitive processes in design. *International Journal of Man-Machine Studies*, 12:119–140, 1980.
- [43] M. McCullough. *Abstracting Craft : The Practiced Digital Hand*. MIT Press, 1997.
- [44] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proc. of the Working Conference on Advanced Visual Interfaces*, 2004.
- [45] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.
- [46] D. A. Norman. Cognitive engineering. In D. A. Norman and S. W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., 1986.
- [47] E. R. Pedersen, K. McCall, T. P. Moran, and F. G. Hulasz. Tivoli: an electronic whiteboard for informal workgroup meetings. In *Proc. of the INTERACT '93 and CHI '93 conference on Human Factors in Computing Systems*, pages 391–398, 1993.
- [48] C. S. Peirce. *Peirce on Signs: Writings on Semiotic*. University of North Carolina Press, Chapel Hill, NC, 1994. edited by James Hoopes.
- [49] M. Petre and A. F. Blackwell. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51:7–30, 1999.
- [50] K. Quartet. Trimpin sketches from 4cast. <http://www.kronosarts.com/pages/trimpinsketches.html>.
- [51] C. E. B. Reas. Process 11 (2006). http://reas.com/iperimage.php?section=works&view=&work=p11_install1&cid=0.
- [52] C. Reas and B. Fry. *Processing: a programming handbook for visual designers and artists*. MIT Press, 2007.
- [53] M. Resnick. Computer as paintbrush: Technology, play, and the creative society. In D. G. Singer, R. M. Golinkoff, and K. Hirsh-Pasek, editors, *Play = Learning: How play motivates and enhances children's cognitive and social-emotional growth*. Oxford University Press, 2006.
- [54] P. G. Rowe. *Design Thinking*. MIT Press, 1991.
- [55] B. Schneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, 1983.
- [56] B. Schneiderman. Creativity support tools: accelerating discovery and innovation. *Communications of the ACM*, 50(12):20–32, 2007.
- [57] D. A. Schön and G. Wiggins. Kinds of seeing and their functions in designing. *Design Studies*, 13(2):135–156, 1992.
- [58] Hog Bay Software. Writeroom: Distraction free writing software for Mac and iPhone. <http://www.hogbaysoftware.com/products/writeroom>.
- [59] M.-A. D. Storey, F. D. Fracchia, and H. A. Meller. Cognitive design elements to

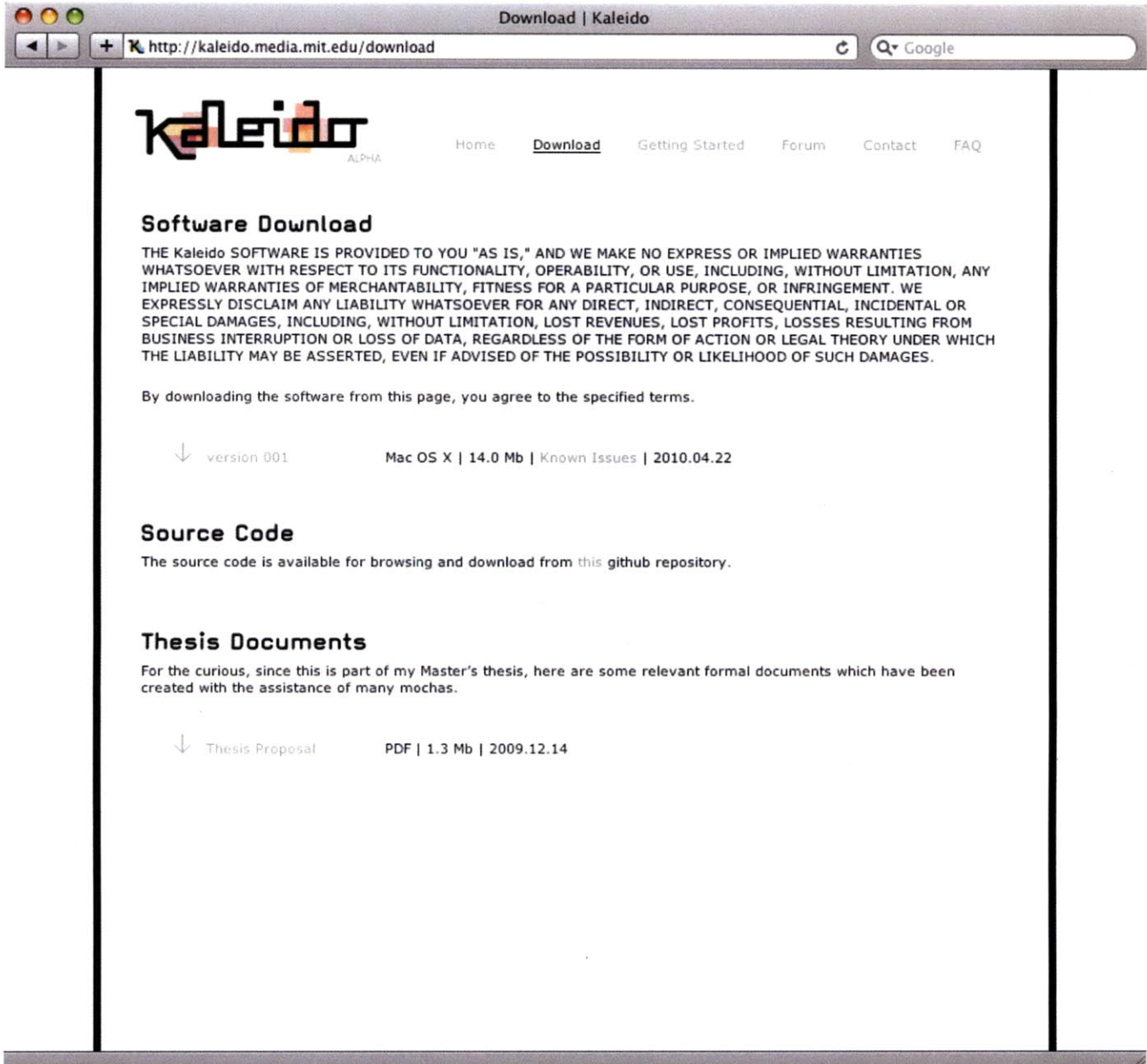
- support the construction of a mental model during software exploration. *The Journal of Systems and Software*, 44:171–185, 1999.
- [60] Adobe Systems. Dreamweaver. <http://www.adobe.com/products/dreamweaver/>.
- [61] Adobe Systems. Flash. <http://www.adobe.com/products/flash/>.
- [62] Adobe Systems. Illustrator. <http://www.adobe.com/products/illustrator/>.
- [63] Adobe Systems. Photoshop. <http://www.adobe.com/products/photoshop/>.
- [64] S. Turkle and S. Papert. Epistemological pluralism: Styles and voices within the computer culture. *Signs*, 16(1):128–157, 1990.
- [65] A. K. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proc. of the 16th International Conference on Software Engineering*, pages 39–48, 1994.
- [66] T. Watson. Funky Forest (2007). http://www.theowatson.com/site_docs/work.php?id=41.

Appendix A

Kaleido Materials:

Website Pages

Demo Video Script and Screenshots



The screenshot shows a web browser window titled "Download | Kaleido". The address bar contains the URL "http://kaleido.media.mit.edu/download". The page features the Kaleido logo (with "ALPHA" below it) and a navigation menu with links for Home, Download, Getting Started, Forum, Contact, and FAQ. The main content is organized into three sections: "Software Download", "Source Code", and "Thesis Documents".

Kaleido
ALPHA

Home [Download](#) Getting Started Forum Contact FAQ

Software Download

THE Kaleido SOFTWARE IS PROVIDED TO YOU "AS IS," AND WE MAKE NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER WITH RESPECT TO ITS FUNCTIONALITY, OPERABILITY, OR USE, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR INFRINGEMENT. WE EXPRESSLY DISCLAIM ANY LIABILITY WHATSOEVER FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST REVENUES, LOST PROFITS, LOSSES RESULTING FROM BUSINESS INTERRUPTION OR LOSS OF DATA, REGARDLESS OF THE FORM OF ACTION OR LEGAL THEORY UNDER WHICH THE LIABILITY MAY BE ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY OR LIKELIHOOD OF SUCH DAMAGES.

By downloading the software from this page, you agree to the specified terms.

↓ version 001 Mac OS X | 14.0 Mb | Known Issues | 2010.04.22

Source Code

The source code is available for browsing and download from [this github repository](#).

Thesis Documents

For the curious, since this is part of my Master's thesis, here are some relevant formal documents which have been created with the assistance of many mochas.

↓ Thesis Proposal PDF | 1.3 Mb | 2009.12.14

The screenshot shows a web browser window titled "Getting Started | Kaleido". The address bar contains "http://kaleido.media.mit.edu/gettingstarted" and the search bar contains "Google". The page features the Kaleido logo (with "ALPHA" below it) and a navigation menu with links for Home, Download, Getting Started (underlined), Forum, Contact, and FAQ.

Tutorial

- ↓ Hello Ball

Interface Guide

- ↓ Visual-Code Associations
- ↓ Drawing Area
- ↓ Code Windows
- ↓ Text Area
- ↓ Code Margin

Reference

- ↓ Drawing Toolbar
- ↓ Keyboard Shortcuts

Welcome! Kaleido is a programming environment interface that was created to make it easier for visual-thinkers to develop applications by providing the means to create personal meaningful visual interfaces to their own code. While many software engineering tools exist to automate code visualization, we wanted a tool that could allow individuals to visually plan, organize, and navigate through code in the idiosyncratic way we each think. We thought that, if so many of us make napkin sketches to lay out our thoughts before we program, why don't we build a tool that allow ourselves to use these sketches as an interface to our code?


The first part of this guide is a simple and fun tutorial to demonstrate some of the ways Kaleido can help you in the coding process. The second half explains each of the different parts of the interface in further detail, and includes a comprehensive listing of the available keyboard shortcuts. You can also look at the example projects provided with the Kaleido application package by accessing them from the *File* → *Example* menu.

Kaleido is based on the Processing project, acting as an augmented interface for the Processing Development Environment and the Processing Language. This guide assumes you're already familiar with Processing. If you would like a refresher course on Processing you can take a look at their [Getting Started](#) guide.




The latest version of Kaleido can be found at the [Download](#) page.

Hello Ball Tutorial

In this tutorial we will learn to use Kaleido to help us write a simple program in which a ball will follow your mouse around. This means that, in each frame of the animation, our program will fill the canvas background and draw a ball at the the mouse position.

So first let's make a ball shape to represent our ball. Click on the shape tool  and hold


Getting Started | Kaleido
http://kaleido.media.mit.edu/gettingstarted
Google








```


void draw() {
  //white
  background(255);
  //blue!
  fill(0,0,255);
  ellipse(mouseX,
    mouseY,40,40);
}

```





Link the setup method code to the setup shape  by clicking on the shape, clicking on the link button in the toolbar (notice that the link button stays highlighted like this ) , and selecting the code. A linked shape appears brighter and linked code is marked in the margin. Selecting any part of linked code will cause the shape to be highlighted in gray, and selecting the shape will cause the linked code to be highlighted.





Next let's also make a shape for the draw method. This time let's select the code first, and then select the rectangle tool  . Notice now that the cursor is a crosshair and the link button is highlighted. Draw a shape, and the link will automatically be established. Let's label this shape "draw loop" and color it blue.

We can also use code windows to write our code. Double-click on "draw loop", or click on the code window button  to open the shape's code window.

Inside the code window, fill in the draw method with the code on the left. As you type, you can see that any changes you make in the code windows is reflected in the main text area. This can be particularly useful when you are working with multi-file projects.

Our program is pretty complete, and just to finish our drawing, let's link the line of code that draws the ball to the our ball shape  by selecting the code, clicking on the link button  , and clicking on the shape. Notice that any code can be linked to multiple shapes, although each shape can only be linked to one continuous fragment of code.

Getting Started | Kaleido
http://kaleido.media.mit.edu/gettingstarted
Google










```


void setup() {
  size(400,400);
  noStroke();
}


void draw() {
}




```




So first let's make a ball shape to represent our ball. Click on the shape tool  and hold down the mouse button to see the available shapes. Click on the circle tool  and your cursor should change into a crosshair .

Make a circle on the drawing area by clicking and dragging the size of the shape. When a shape is selected, it will have a yellow highlight around it , and the small square handles allow you to resize the shape.

Open the label editor by clicking once on our selected ball. Let's type in "draw ball" in the first field, and "use mouse position" in the bigger field. When you're done, click outside the label editor to save. Whenever you want to cancel out of the label editor without saving, simply hit the escape key  on your keyboard.

Next let's make a shape to represent our program setup. Let's use the rectangle shape again  and this time simply label it "setup". Since we think of "setup" and "ball" as very different sorts of things, let's color the setup rectangle a different color by clicking and holding down the paint bucket button to select green . Once your mouse has changed to a paint bucket cursor , click on the shape you want to color.


Now let's link some code to our shapes. Type the code on the left into the text area. This prepares the canvas we need, and gives us a code stub in which to fill in the animation code.

Link the setup method code to the setup shape  by clicking on the shape, clicking on

Getting Started | Kaleido


http://kaleido.media.mit.edu/gettingstarted

Google

Let's run the program! Click on the run button 


To learn more about Kaleido, explore the rest of this guide, take a look at the example projects by accessing them from the *File* → *Example* menu, and join the discussions in the [forum!](#)

Interface Guide




Visual-Code Associations

Our primary motivation was to enable programmers to make personalized sketches of their code digitally, and be able to use their digital drawings as an idiosyncratic interface to their code. Thus Kaleido enables the programmer to create visuals along side their code editor, and associate individual visual elements (i.e. link) with any given fragment of code, giving programmers visual ways to organize and navigate their projects.



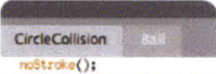
Drawing Area

On the left side of Kaleido's environment is a canvas that enables you to create digital drawings via the drawing tools. You can control the placement, color, and labels of any of the visual elements. Edit any drawn element by selecting it, and then clicking once: the label editor (as pictured) should show up.



Code Windows

If you have linked a particular visual element with a code fragment, you can open a code window to gain an additional view of your code regardless of where you navigate to in your code editor. You can edit code through code windows just like the main text editor.



Text Area

Getting Started | Kaleido
http://kaleido.media.mit.edu/gettingstarted
Google

CircleCollision **Ball**

```

noStroke();
}

void draw() {
  background(51);
  fill(204);
  for (int i=0; i< 2; i++){
    balls[i].x += vels[i].x;
    balls[i].y += vels[i].y;
    ellipse(balls[i].x, balls[i].y, 20, 20);
  }
}

void draw() {
  background(255);
  ellipse(x,y,40,40);
  x += dx;
  y += dy;

  if (x > width ||
      dx <= -1;
      fill(random(255), random(255), random(255)));
}

```

Text Area

This is the main code editing area, which functions the exact same way as traditional programming text editors.

```

void draw() {
  background(255);
  ellipse(x,y,40,40);
  x += dx;
  y += dy;










  if (x > width ||
      dx <= -1;
      fill(random(255), random(255), random(255)));
}

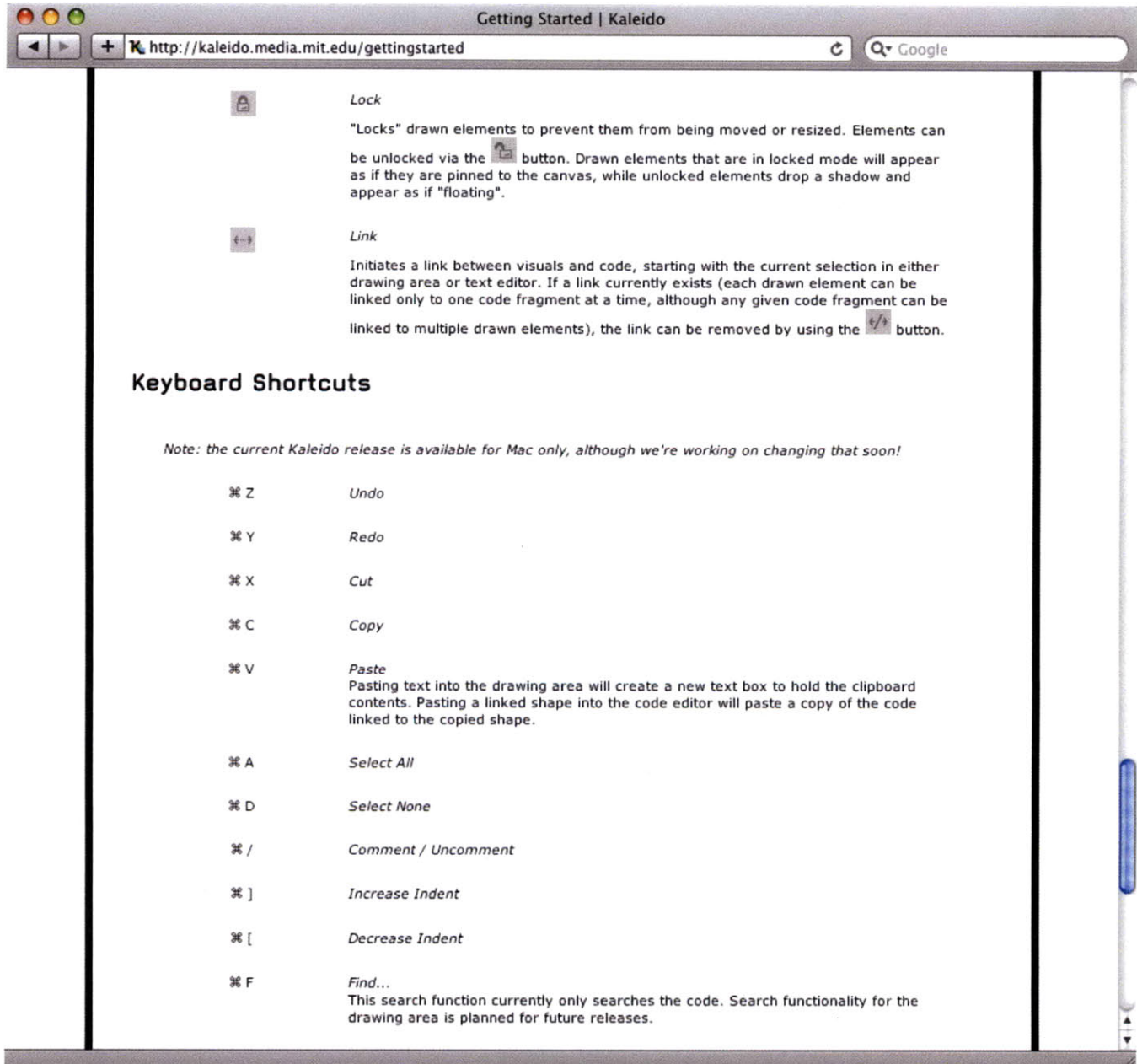
```

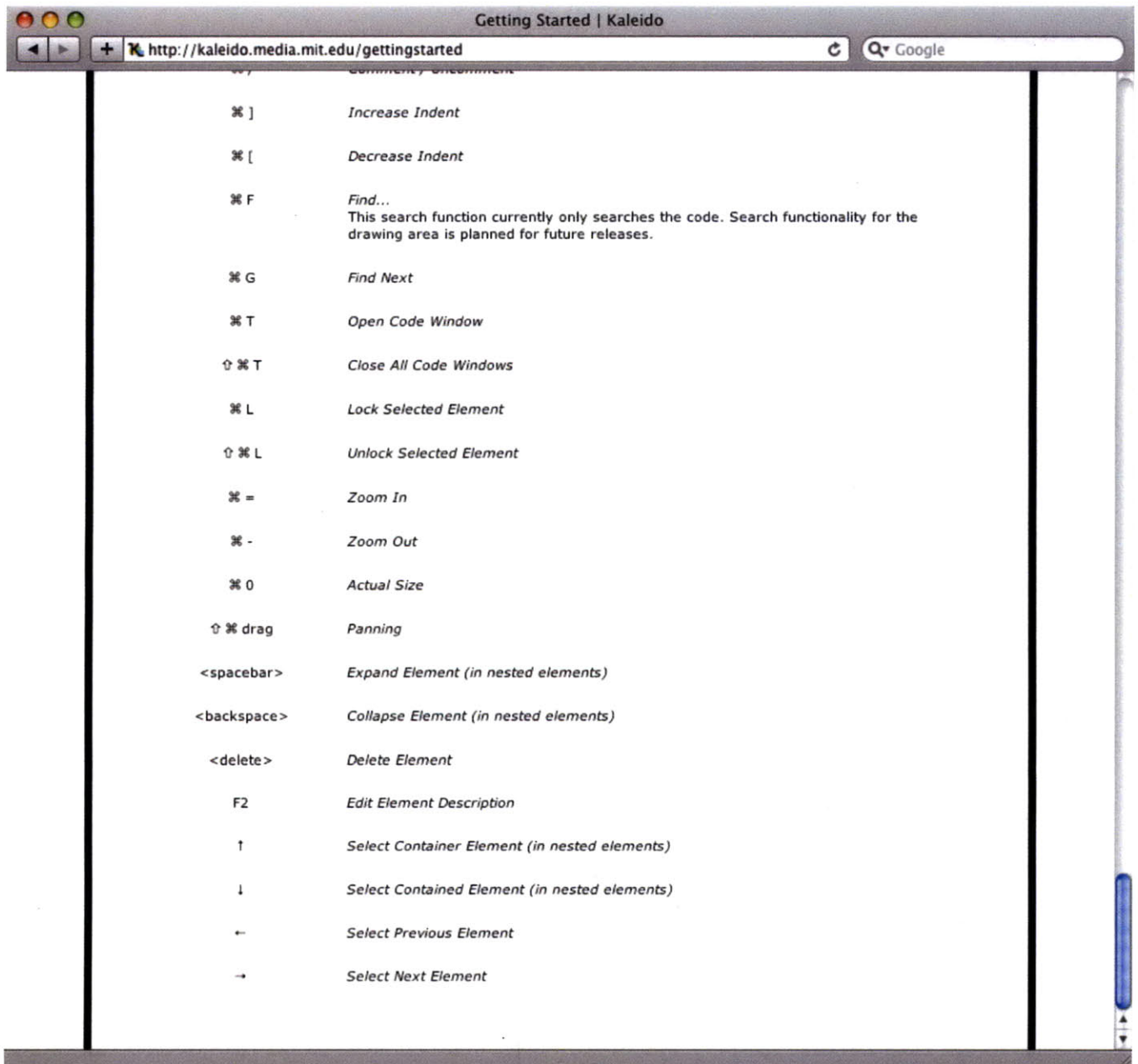
Code Margin

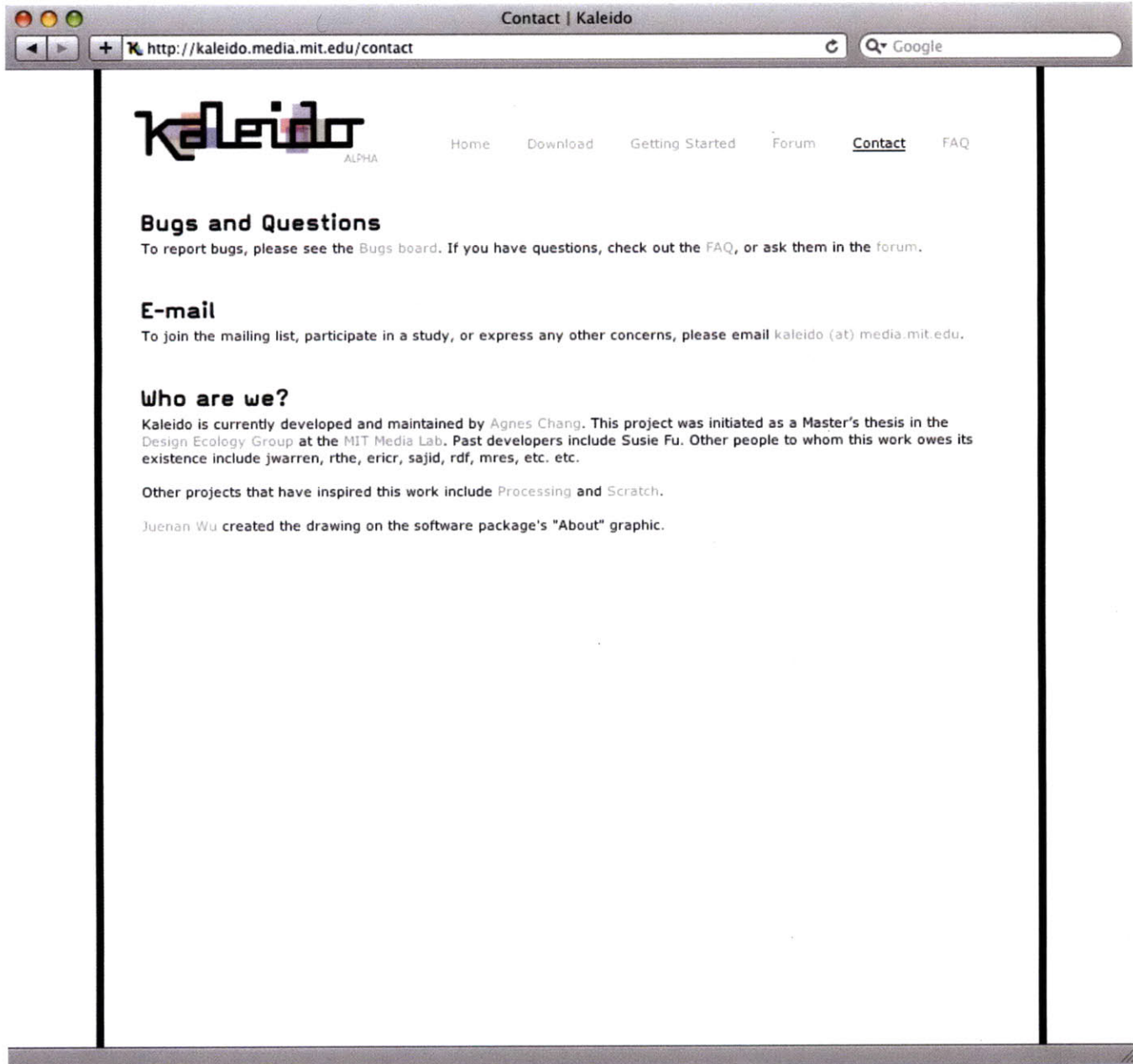
When a piece of code has been linked to a shape, the line of code will be marked on the margin with a strip of color that corresponds to the color fill of the linked shape.

Drawing Toolbar

	<p>Shapes</p> <p>Click and hold down to see other options such as  and . All shapes hold a title and description, and can be linked to code. Any new shape will be filled with the color indicated in the <i>Color Fill</i> button at the time of creation.</p>
	<p>Connectors</p> <p>Plain basic connections in three style options (arrow, dotted, etc.), which can be attached to shapes or left dangling. Connectors can be labelled, but not linked to code.</p>
	<p>Text Box</p> <p>Creates a text box with a transparent background that holds a description.</p>
	<p>Color Fill</p> <p>Select this tool and click on a shape to fill it with the selected color. The currently selected color will be used to fill newly created shapes.</p>
	<p>Code Window</p> <p>Opens code windows on selected shapes that have been linked to code.  will hide the code windows of the selected shapes.</p>
	<p>Lock</p>








Forums | Kaleido

http://kaleido.media.mit.edu/forum

Google



Home Download Getting Started **Forum** Contact FAQ

Login to post new content in the forum.

Forum	Topics	Posts	Last post
<input checked="" type="checkbox"/> News and Announcements What's happening? Find out the project's latest status.	1	2	7 weeks 1 day ago by clone
<input checked="" type="checkbox"/> General Discussion Is Kaleido the tool you always wanted? Is it absolutely useless? Share your experience and help us determine future directions.	2	3	1 week 5 days ago by marlonj
<input checked="" type="checkbox"/> Gallery What did you make with Kaleido? Share and discuss each others' work and works-in-progress.	1	1	8 weeks 6 days ago by achang
<input checked="" type="checkbox"/> Bugs and Troubleshooting Technical difficulties? Report them here!	1	1	9 weeks 1 day ago by achang

FAQ | Kaleido

http://kaleido.media.mit.edu/faq

Google

kaleido ALPHA

Home Download Getting Started Forum Contact **FAQ**

- ⊕ Is Kaleido compatible with Processing projects?
- ⊕ Is Kaleido open source?
- ⊕ How do I get started?
- ⊕ Who works on this project?
- ⊕ Why is it called Kaleido?
- ⊕ On what platforms can I run Kaleido?
- ⊕ It crashed! It's broken! I found a bug; what do I do?
- ⊕ I disagree with this whole idea.

Is Kaleido compatible with Processing projects?

Yes! Opening PDE-created projects in Kaleido will simply yield a blank drawing to start, but the project should run and execute with out any problems. Opening a Kaleido project in the PDE simply omits the drawing area and all associated functions, but your code will appear and function in exactly the same way.

Is Kaleido open source?

Like the Processing initiative on which it is based, the Kaleido environment is released as open source under the GNU General Public License. The source code will be made available via an easy-to-access interface soon.

How do I get started?

Check out the [Getting Started](#) page. Ask in the [forum](#) if you have any questions about the process.

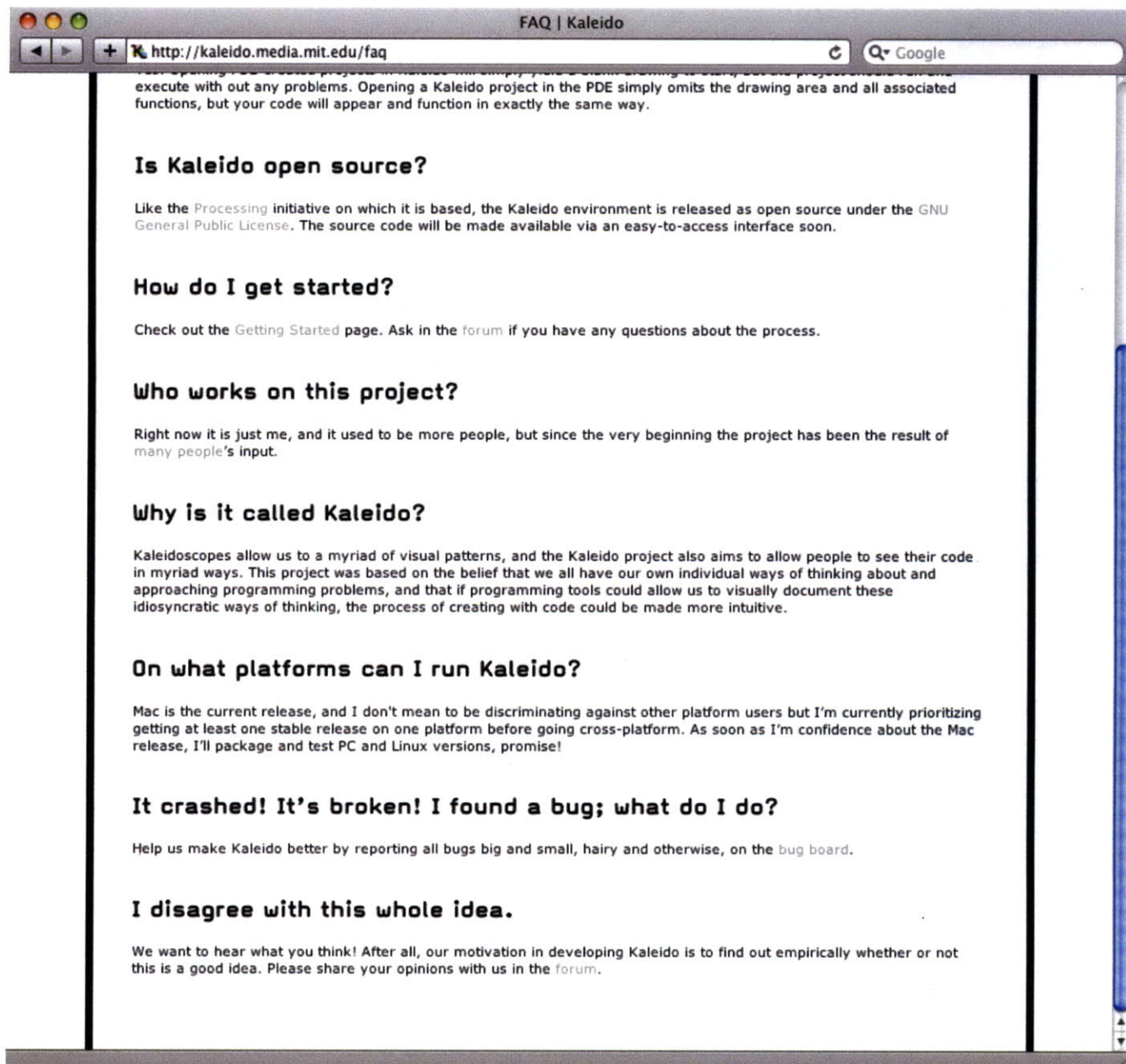
Who works on this project?

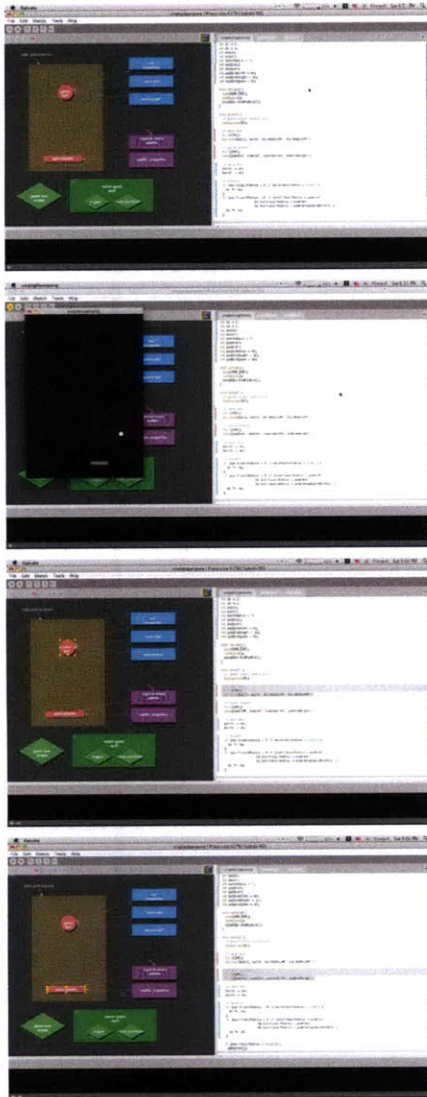
Right now it is just me, and it used to be more people, but since the very beginning the project has been the result of many people's input.

Why is it called Kaleido?

Kaleidoscopes allow us to a myriad of visual patterns, and the Kaleido project also aims to allow people to see their code in myriad ways. This project was based on the belief that we all have our own individual ways of thinking about and approaching programming problems, and that if programming tools could allow us to visually document these idiosyncratic ways of thinking, the process of creating with code could be made more intuitive.

On what platforms can I run Kaleido?





Kaleido Demo Video Transcript

Kaleido is a tool that is designed to help visual-thinkers program. Kaleido makes the creative process easier by letting you create personally meaningful visual interfaces for your code.

Kaleido can help you quickly remember the different parts of your program. This program, for example, is a single player Pong game built in the Processing language.

Everyone knows how a Pong game works, namely that you have a user-controlled paddle that moves left and right to catch a bouncing ball. However, there is a gap between knowing how the program works and knowing how the program is built.

With Kaleido, you can make drawings as you program to help your process. Here we see a ball-shape, and if we select it, it points us to the pieces of code that paints the pixels of the ball to the screen. This box next to it is linked to the code that makes the ball bounce.

If we click on this rectangle that looks like a paddle, it points us to the code that paints the paddle, and meanwhile this keyboard symbol is linked to the code that moves the paddle when the user presses on the arrow keys.

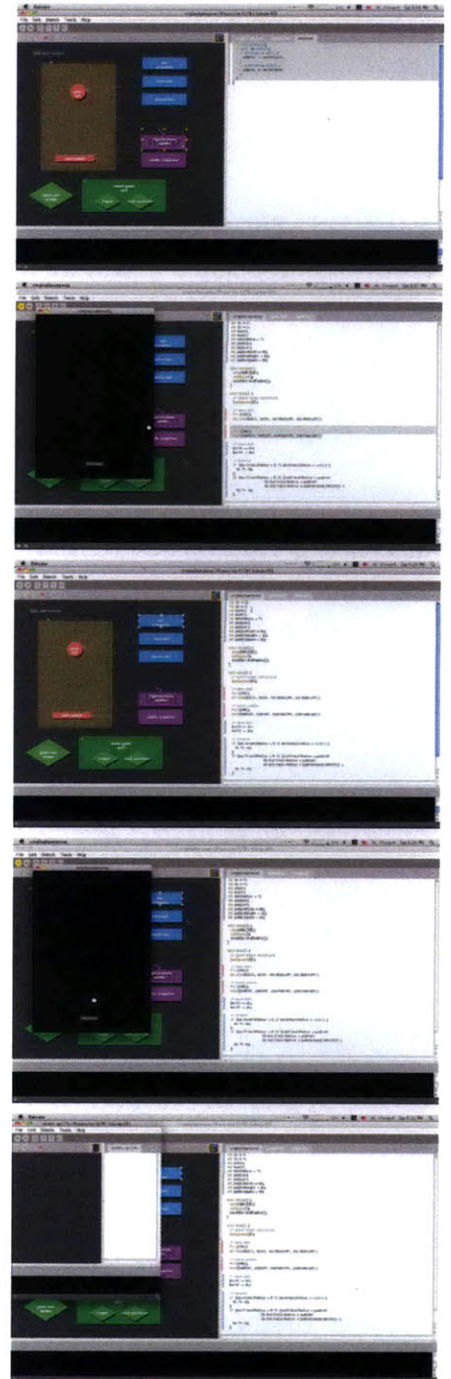
With the help of visuals, the mass of code quickly becomes understandable. As we are jumping back and forth working on different parts of the program, we can look to the drawing area at any time to find the piece of code we need.

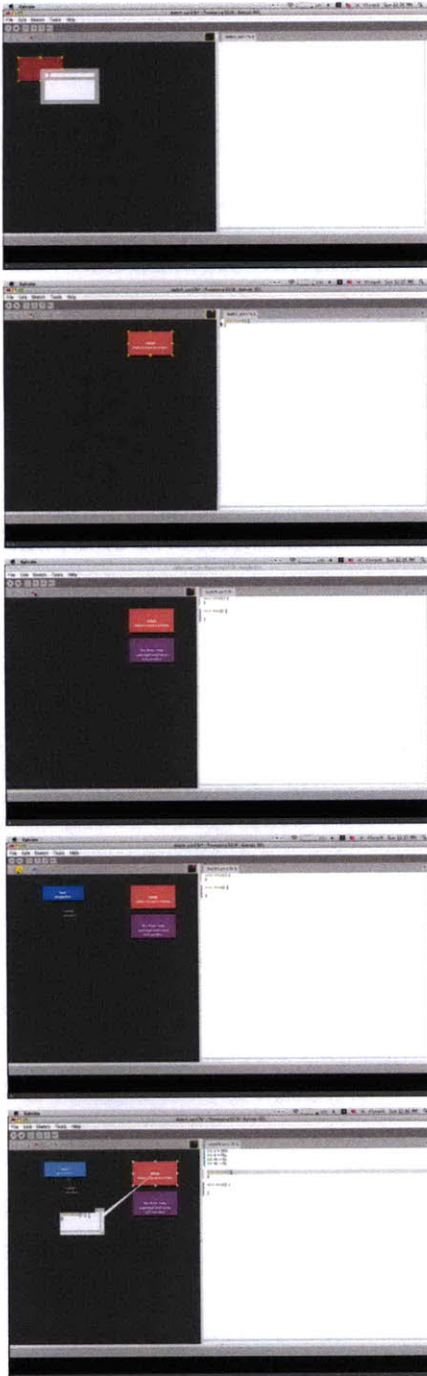
For example, when I run this Pong game I might discover that the ball is moving too slowly and I would like to make it faster.

So where do I put in the adjustment? It is hard to do a normal text search because I don't know if the code I am looking for called "speed" or "ballspeed", or even something else?

But I have a blue block here that says "ball properties" and by clicking on that block now, I see that I had named the variables "dx" and "dy" instead. So let's change those values to make it twice as fast. Now let's save, and run it again.

Now that looks much better.





So Kaleido helps you edit your program without wasting a lot of time just looking for the piece of code you want to change. Kaleido can also help when you are starting out with a new project.

Let's say for example that I want to make a bouncing ball that changes color every time it bounces.

Let me quickly think through what parts of the program I am going to need:

I will need to setup because I want a square canvas. So I'm going to make a box here, but I'm also going to write the code skeleton for this. I can link my code to the box by selecting the code, clicking on the link button, and then selecting the shape. Now the box is now a little brighter, and in the code margins here there is a marker to let us know that the code is linked.

Then, in the draw method that is called every animation frame I know I'll need to change the ball's position and paint it so it moves a short distance each frame. So let's make a box and a skeleton for that too. Notice that you can also establish links by first selecting the shape and then the code.

Let's also give the draw shape a different color using the color tool.

I will also need the ball properties as the Pong game, which I will also color blue to make it easy to remember.

And just to make sure I don't forget anything, I'm going to make a quick list here of what I need.

This looks like a good enough outline to me for now, so I'm going to start filling in some code. Let's write the variables to define the ball's speed and position and link that to the ball properties.

You can also open code windows on linked shapes to see the linked pieces of code while looking at other parts of your program in the main text area. Of course, you can also make edits in the code windows.

Now that I've setup the canvas I'm going to paint an ellipse in the draw loop and update the ball's position variables.

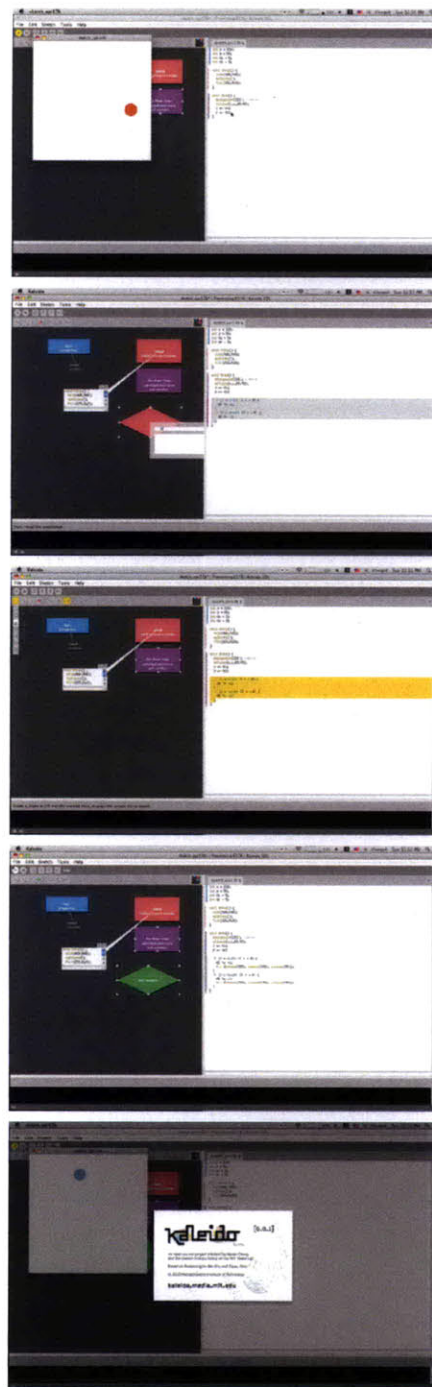
Now I think I have enough code to run the application. Let's see how it looks so far. That looks about right except I haven't put in the bounce.

So now every time I move the ball I'm going to also check that if the ball's position has gone out of bounds I need to change it's direction.

To help me find this code later, I'm going to add another shape. If you select some code and click on the shape tool, the new shape will automatically be linked to the code. Let's give this shape a different color too.

Finally, each time the ball bounces I also need to give it a random color. So here we go.

Kaleido is a free open-source project available from kaleido.media.mit.edu. Kaleido is based on the Processing project. Welcome to Kaleido.



Appendix B

Investigation Materials:

Instructions

Drawings

Questionnaire

KALEIDO PRELIMINARY STUDY

Dear [participant name],

First of all, we really appreciate your taking the time to participate. Below you will find step-by-step instructions for completing this preliminary study. The entire activity should take about an hour, although you are of course welcome to take as much time as you like. Feel free to work through it in multiple sessions. Our only request is that you email everything to us by **January 25th**, which is the end date of this study. If you have any questions please do not hesitate to contact us at *kaleido@media.mit.edu*.

Thanks again,

Agnes Chang

STUDY CONTENTS

- Consent form
- Drawing 1: Two-Player Pong
- Drawing 2: Mario-style Traveler
- Drawing 3: Your Program
- Online Questionnaire

INSTRUCTIONS

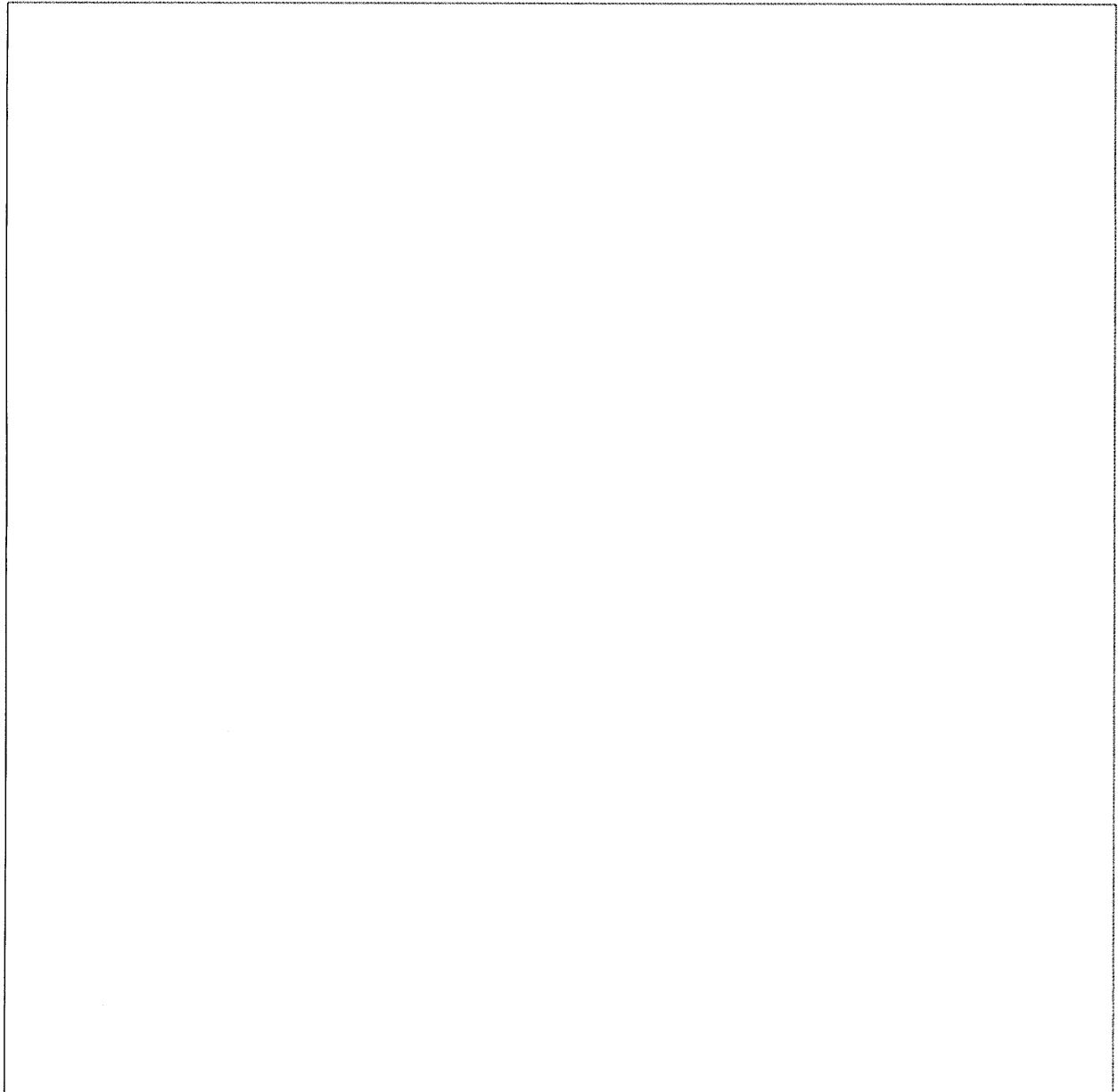
1. Read through the consent form and sign and date the last page.
2. Complete the drawing activities in order.
3. Complete the questionnaire available online at <http://www.surveymonkey.com/s/kaleido> (Your participant ID is __)
4. Please scan and email the consent signature, and your three drawings to *kaleido@media.mit.edu* (If scanning poses a difficulty let me know, I can send you a postage-paid snail-mail envelope instead.)
5. Email us your code for Drawing 3, and PayPal account name or your name and mailing address for a check.
6. I will transfer \$15 U.S.D. to you by check or PayPal.
7. Thank you so much!

KALEIDO PRELIMINARY STUDY DRAWING 1: Two-player Pong

Imagine that you will be coding a traditional two-player Pong game: a computer program that two paddles controlled by two players using the keyboard, and the ball that bounces back and forth between the walls and the paddles. Players earn a point when the other player's paddle misses the ball.

Using a pencil or pen in the space below, please draw a sketch as if you were organizing your thoughts before starting to program. Treat this as a "napkin sketch" for yourself to think through how you would code the Pong program: it doesn't need to make sense to other people, and there is no such thing as a better or worse sketch.

PARTICIPANT ID 21

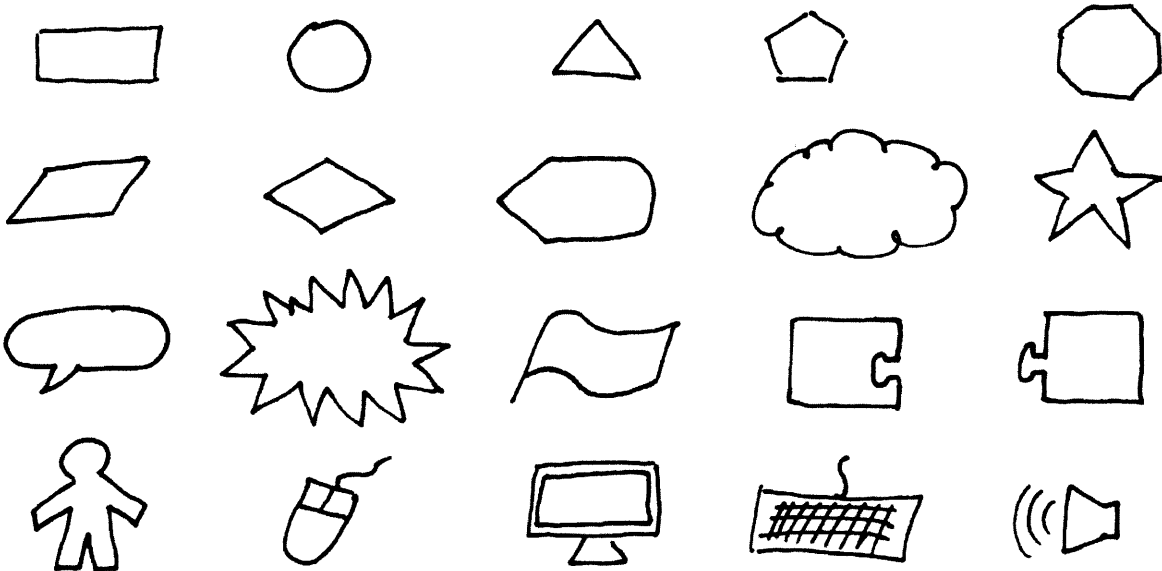


KALEIDO PRELIMINARY STUDY DRAWING 2: Mario-style Traveler

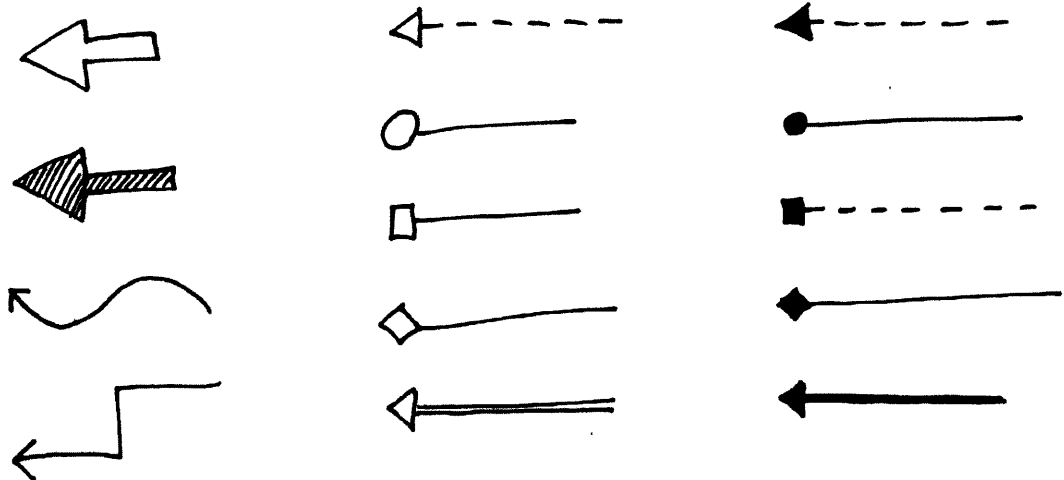
Imagine that you will be coding a Mario-style Traveler game (a.k.a. "side-scrolling video game"): a central character with a basic background story sets off on a virtual journey. The player will be able to experience some short narrative, then navigate the character safely through a space that contains benevolent objects as well as some harmful obstacles. At the end of the journey there is something to indicate to the player that they have completed the game.

Using three different colors of pencil or pen (e.g. black, red, & blue), please use the following shapes and connector styles to draw a mental depiction of your Traveler game. Color in the three mini boxes at the top to let us know which three colors you chose, but you are not required to use all three colors. You are free to use text any way you like, and feel free to make a legend if you find it useful for yourself. You are also encouraged to use more paper.

SHAPES



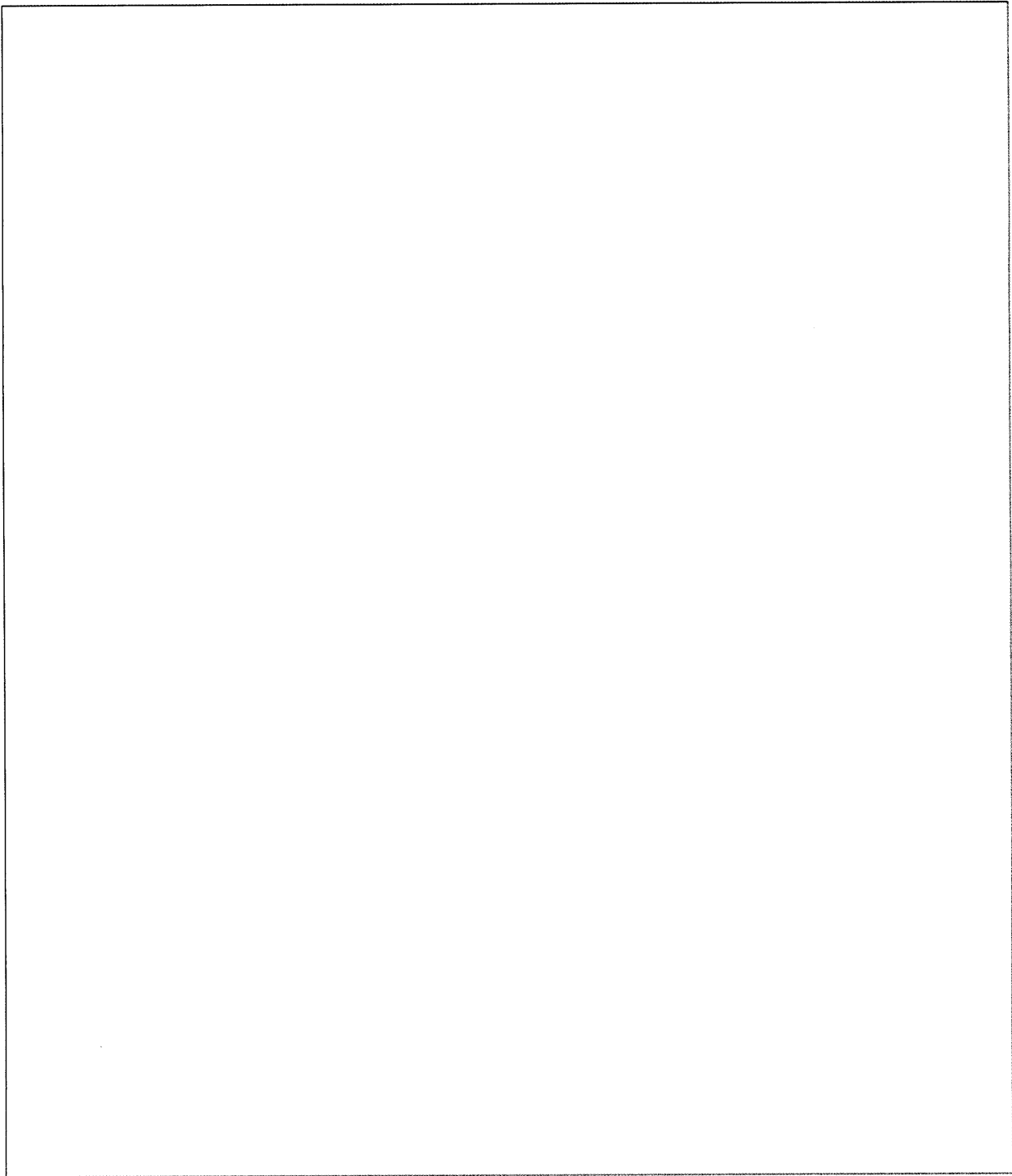
CONNECTORS



KALEIDO PRELIMINARY STUDY DRAWING 2: Mario-style Traveler (cont.)

PARTICIPANT ID _____

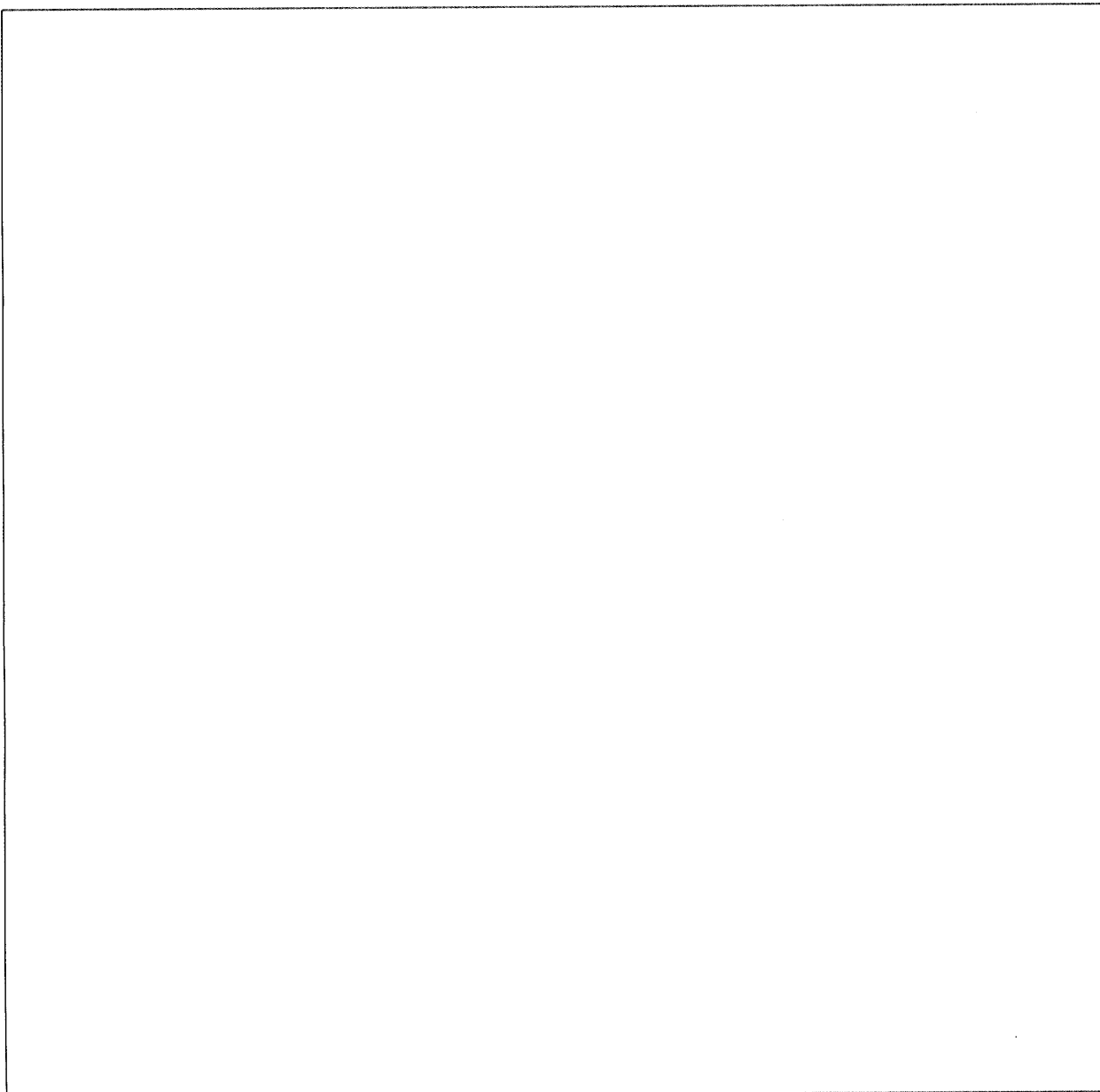
YOUR COLORS OF CHOICE



KALEIDO PRELIMINARY STUDY DRAWING 3: Your Program

Select a program that you have written. It doesn't matter what language it is in, or what type of program you choose, as long as you feel comfortable sending us the source code with your drawing. Draw a visual depiction of your program. This time, feel free to use digital tools (Photoshop, Illustrator, graphing programs, etc.) or mixed media, or any combination thereof. What if you could make a *collage* of your program? What if you could *dance* your program? Needless to say, your representation need not fit in the box below -- be creative and have fun!

PARTICIPANT ID _____



KALEIDO PRELIMINARY STUDY QUESTIONNAIRE

PARTICIPANT ID _____

NAME _____

CITY, STATE/COUNTRY _____

GENDER male female

AGE 18-22 22-30 30-40 40+

OCCUPATION student educator professional

HOW WOULD YOU CHARACTERIZE THE NATURE OF YOUR WORK? (circle any that apply)

graphics video audio robotics web
interactive installation animation 3D-modelling other _____

HOW MANY YEARS HAVE YOU BEEN PROGRAMMING? _____

FORMAL PROGRAMMING CLASSROOM EXPERIENCE (circle all that apply)

high school college graduate school none/self-taught

PROGRAMMING LANGUAGES YOU KNOW (circle all that apply, circle your FIRST language twice)

BASIC Logo C/C++ Java Action-/Java-script
Python Ruby Processing MaxMSP/vvvv other _____

HOW DO YOU APPROACH PROGRAMMING TASKS? (rank 1 as first, and leave blank if not applicable)

____ write pseudo-code ____ research other people's solutions
____ draw a diagram/sketch ____ start coding and figure it out as you go
other (please describe):

KALEIDO PRELIMINARY STUDY QUESTIONNAIRE (cont.)

PARTICIPANT ID _____

Note: feel free to illustrate, or to use extra paper to answer any of these questions.

HOW WOULD YOU DESCRIBE THE CONTENTS OF YOUR PERSONAL SKETCHBOOK?

WHAT ARE THE HARDEST PARTS TO REMEMBER WHEN YOU REVISIT YOUR OWN OLD CODE?

WHAT VISUAL FEEDBACK CAN YOU IMAGINE WOULD BE HELPFUL WHEN ...

WRITING CODE?

RUNNING THE PROGRAM?

DEBUGGING?

Appendix C

Evaluation Materials:
Questionnaire

KALEIDO EVALUATION STUDY QUESTIONNAIRE

NAME _____

CITY, STATE/COUNTRY _____

GENDER male female

AGE 18-22 22-30 30-40 40+

OCCUPATION student educator professional

HOW WOULD YOU CHARACTERIZE THE NATURE OF YOUR WORK? (circle any that apply)

graphics video audio robotics web
interactive installation animation 3D-modelling other _____

HOW MANY YEARS HAVE YOU BEEN PROGRAMMING? _____

FORMAL PROGRAMMING CLASSROOM EXPERIENCE (circle all that apply)

high school college graduate school none/self-taught

PROGRAMMING LANGUAGES YOU KNOW (circle all that apply, circle your FIRST language twice)

BASIC Logo C/C++ Java Action-/Java-script
Python Ruby Processing MaxMSP/vvvv other _____

HOW DO YOU APPROACH PROGRAMMING TASKS? (rank 1 as first, and leave blank if not applicable)

____ write pseudo-code ____ research other people's solutions
____ draw a diagram/sketch ____ start coding and figure it out as you go
other (please describe):

KALEIDO EVALUATION STUDY QUESTIONNAIRE (cont.)

FOR HOW LONG DID YOU USE THE KALEIDO ENVIRONMENT?

average ____ hours per day for ____ days

HOW EASY WAS IT TO CREATE THE VISUALS YOU WANTED?

intuitive somewhat intuitive neutral somewhat difficult very difficult

PLEASE DESCRIBE HOW YOU USED THE VISUALS IN YOUR CODING PROCESS?

HOW MUCH DID THE VISUALS HELP YOU...

PLAN YOUR PROGRAM LOGIC?

extremely helpful very helpful somewhat helpful barely helpful not helpful

REMIND YOURSELF OF PROGRAM STRUCTURE?

extremely helpful very helpful somewhat helpful barely helpful not helpful

NAVIGATE YOUR CODE?

extremely helpful very helpful somewhat helpful barely helpful not helpful

OTHER (please specify):

DO YOU THINK THE VISUALS WERE WORTH THE EFFORT?

yes somewhat neutral not really not at all

KALEIDO EVALUATION STUDY QUESTIONNAIRE (cont.)

WHAT CHANGES WOULD YOU SUGGEST FOR THE INTERFACE?

WOULD YOU USE THIS TYPE OF INTERFACE REGULARLY FOR YOUR WORK? WHY OR WHY NOT?

DO YOU THINK THIS TYPE OF INTERFACE IS MORE SUITED FOR PARTICULAR KINDS OF WORK? FOR WHOM DO YOU THINK IT WOULD BE PARTICULARLY USEFUL? PLEASE EXPLAIN.

FOR WHAT KINDS OF WORK DO YOU THINK THIS TYPE OF INTERFACE IS *UNSUITED*? PLEASE EXPLAIN.

FOR WHICH OF THE FOLLOWING DO YOU THINK YOU WOULD USE THIS TOOL MORE? (rank the following from 1 through 7, where 1 is "most often")

___ problem-solving

___ documentation

___ ideation

___ sharing with project collaborators

___ learning

___ sharing with the open-source community

___ teaching

___ other (please describe):