Dynamic Drawing: Broadening Practice and Participation in Procedural Art

Jennifer Jacobs

B.F.A. Digital Arts, University of Oregon (2007)
M.F.A. Integrated Media Arts, Hunter College (2011)
M.S. Media Arts and Sciences, Massachusetts Institute of Technology (2013)

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Media Arts and Sciences at the Massachusetts Institute of Technology

September 2017

Author .....
Program in Media Arts and Sciences
August 18, 2017

Certified by .....
Mitchel Resnick
LEGO Papert Professor of Learning Research
Program in Media Arts and Sciences
Thesis Supervisor

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

Dynamic Drawing: Broadening Practice and Participation in Procedural Art

Jennifer Jacobs

Submitted to the Program in Media Arts and Sciences
on August 18, 2017, in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Media Arts and Sciences

Abstract

Computation is a powerful medium for art creation. *Procedural art*, or artwork defined by a computationally represented system of rules, relationships, and behaviors, enables creation of works that are flexible, adaptable, and capable of systematic revision. Yet the medium for creating procedural art, computer programming, can pose significant barriers for manual artists. Programming can be challenging to learn, and programming tools can restrict the concrete practices of manual art. An analysis of the creative opportunities of procedural art and the conflicts programming poses for manual artists raises these questions: (1) How can we create procedural art systems that are accessible and expressive for manual artists? (2) How can we support different ways of thinking and creating with representational mediums? (3) How can procedural art systems contribute to the process of learning and understanding representational mediums?

This dissertation explores these questions through two new systems that integrate manual and procedural creation. *Para* is a digital illustration tool that enables artists to produce procedural compositions through direct manipulation. *Dynamic Brushes* is a system that enables artists to create computational drawing tools that procedurally augment the process of manual drawing. Para and Dynamic Brushes were informed through interviews with artists and evaluated through multi-week open-ended studies in which professionals created polished artwork. These evaluations provided a framework for developing creative tools through extended work with creative professionals. Comparison of artwork produced with Para and Dynamic Brushes revealed specific trade-offs in expressiveness, ease of entry, and working style for direct manipulation and representational procedural tools. Overall, this research demonstrates how integrating manual and procedural creation can diversify the kinds of outcomes people can create with procedural tools and the kinds of people who can participate in procedural art.

Thesis Supervisor: Mitchel Resnick
Title: LEGO Papert Professor of Learning Research
Program in Media Arts and Sciences

Dynamic Drawing: Broadening Practice and Participation in Procedural Art

Jennifer Jacobs

The following people served as readers for this thesis:

Thesis Reader.....

Joel Brandt
Manager, Research Engineering
Snap Research
Snap, Inc.

Thesis Reader.....

Golan Levin
Associate Professor of Electronic Time-Based Art
Director, Frank-Ratchye STUDIO for Creative Inquiry
School of Art
Carnegie Mellon University

Acknowledgements

I express my heartfelt gratitude to my advisor, Mitchel Resnick. Thank you for the kindness and guidance you consistently provided while I was your student. When I first moved to New York, an acquaintance recommended I read Turtles, Termites, and Traffic Jams. This book fundamentally shaped my understanding of programming as a medium for making things and provided a lens for learning about the world. I never imagined that I would be fortunate enough to work with the author.

I also thank my readers. Golan LevinâĂŹs computational artwork motivated me to pursue programming during my M.F.A., and his feedback throughout the dissertation process pushed me to dream big and take on challenging problems. Joel Brandt taught me how to build systems and how to be a researcher. I joined the Media Lab with the goal of creating tools for others. Joel, you helped make this goal a reality.

Several other people and organizations were instrumental to the development of Para and Dynamic Brushes. Thank you to Radomír Měch for your guidance and support throughout my time at Adobe Research and beyond, and to Sumit Gogia and Michael Craig for contributing to the development of Para and helping to move the project forward. Thank you to the members of the Dynamic Medium Research Group, Bret Victor, Virginia McArthur, Toby Schachman, Joshua Horowitz, Paula Te, Chaim Gingold, and Luke Iannini for providing invaluable feedback and insights during the early stages of Dynamic Brushes' development.

Thank you to the many other mentors who helped me along this path. To Leah Buechley, who helped me see programming as a tool for creating line, form, and pattern, and who brought together the first engineering community where I felt at home. To Mary Flanagan, for teaching me that supporting women in programming requires fostering confidence, developing alternative pathways for learning, and connecting programming to a broad range of applications and interests. To Craig Hickman, who taught the Programming for Artists course that first got me hooked on coding, and who encouraged me to believe in my own abilities as a programmer. To Michael Salter, who fostered my love of drawing and my love of robots, and is still the coolest professor I have ever had. And to Christopher Coleman, whose art shaped the kind of work I wanted to make in the future, and whose drive to support the creativity of others inspired me to teach.

Thank you to the members of both the High-Low Tech and Lifelong Kindergarten Research Groups. To David Mellis, Jie Qi, Samuel Jacoby, Emily Lovell, Hannah Perner Wilson, Ed Baafi, Karina Lundhal, and Kanjun Qiu, thank you for sharing your ideas and expertise, your love of good food, and your friendship. To Amon Millner, Jay Silver, Eric Rosenbaum, Ricarose Roque, Sayamindu Dasgupta, Tiffany Tseng, Abdulrahman Idlbi, Champika Fernando, Alisha Panjwani, Shrishti Sethi, Juliana Nazaré, Moran Tsur, Carmelo Presicce, Shruti Dhariwal, Kreg Hanning, Stefania Druga, Natalie Rusk, Brian Silverman, Andrew Sliwinksi, and Abisola Okuk, thank you for encouragement and support throughout the four years that I was fortunate enough to work alongside you. And thank you to the broader community of Media Lab students: Amit Zoran, Sean Follmer, Daniel Leithinger, Lining Yao, Philippa Mothersill, David Moinina Sengeh, Nadya Peek, Che-Wei Wang, Taylor Levy, Peter Schmitt, Nan-Wei Gong, Edwina Portocarrero, Brian Mayton, Gershon Dublon, Nathan Matias, Erhardt Graeff, and so many others. I also thank the people that make research at the Media Lab possible: Stefanie Gayle, Linda Peterson, Keira Horowitz, John DiFrancesco, Tom Lutz, Kevin Davis, and everyone else who keeps the lab up and running.

This work required the support and engagement of many artists. Thank you to Emily Gobielle, Shantell Martin, Mackenzie Shubert, Nina Wishnok, and Erik Natzke for taking the time to share your experiences and your expertise. Special thanks to Kim Smith, Ben Tritt, and Fish McGill for contributing your talents, energy, and creativity as collaborators. I hope we have the opportunity to work together again.

Finally, thank you to my family for their love and support. To my Mom Ruth and my Dad Steve, thank you for teaching me to look at the world as a scientist and for supporting my love of art. To my Brother Bryan and my sister Kristine, thank you for providing encouragement throughout the difficult times. Most of all to Nick, thank you for sharing your love, your code, and your fries.

Contents

1

Introduction

Art, in the only sense in which one can separate art from technics, is primarily the domain of the person; and the purpose of art, apart from various incidental technical functions that may be associated with it, is to widen the province of personality, so that feelings, emotions, attitudes, and values, in the special individualized form in which they happen in one particular person, in one particular culture, can be transmitted with all their force and meaning to other persons or to other cultures.
-Lewis Mumford

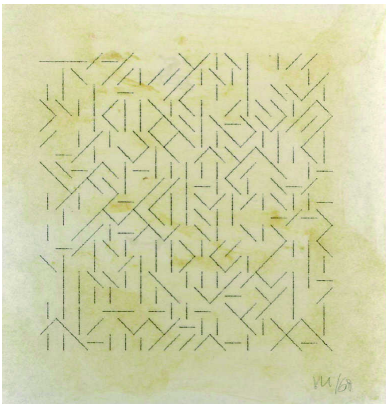

Figure 1.1: Vera Molnar, No Title, 1968, computer graphic, from the collection of the Digital Art Museum.
`dam.org`

The computer is a powerful creative medium. Initially applied to the task of performing complex mathematical calculations for specialized tasks, computers have since become integrated into nearly every aspect of modern human culture. Not long after their development, electronic programmable computers were applied to the creation of images, and shortly after this, people started to use them to make art (fig: 1.1). In 1968 the curator Jasia Reichardt assembled an exhibition entitled Cybernetic Serendipity (fig: 1.2). The show was one of the first to exhibit works produced with computers. These early experiments in computational art anticipated an on-going effort to integrate art and computing that has continued into the present. Today, emerging computational technology is rapidly adapted for artistic purposes as soon as it emerges. To date, artists have applied computer graphics, parametric design (Rosenkrantz and Louis-Rosenberg, 2015), robotics (Hughes, 2017), computational simulation (Bader et al., 2016), digital fabrication (Azzarello, 2017), artificial intelligence (McMullan, 2017), and perhaps most recently, computational biology (Sterling, 2015) as tools for art creation.

Making art is one of the things that defines what it means to be human (Mumford, 1952). In addition to creating tools for basic survival, there is evidence

that our earliest ancestors also took time to engage in decorative and symbolic forms of making. Similarly, modern humans invest a great deal of time and energy in developing practical tools and technologies to improve our lives, but we also take time to create artifacts that are beautiful, critical, provocative, and self-reflective. The adaptation of computers to art-making corresponds with humanity's longstanding impulse to adapt tools and materials as means for creative expression.

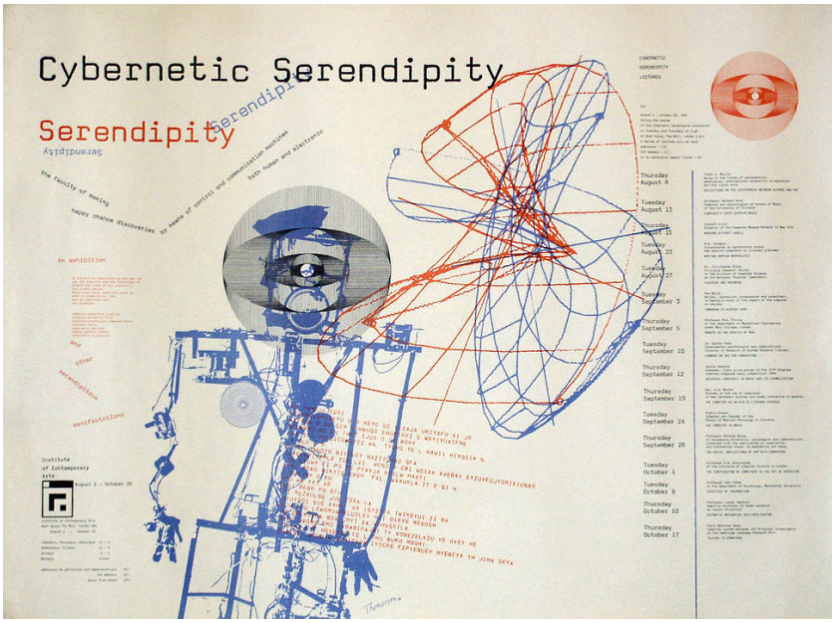

Figure 1.2: Exhibition poster from the Cybernetic Serendipty Exhibition by Franciszka Themersonr, from the Cybernetic Serendipity Archive. `cyberneticserendipity.net`

When viewed through the lens of human creativity, it's not surprising that people have applied computers to art. What is perhaps surprising are the ways in which computers have changed *how* we make art. The works in Cybernetic Serendipity were notable because they were not created by human artists. Instead, they were generated independently by machines and algorithms. It's true that humans have often relied on technological tools to execute work; however, computer-generated art is different. Early computers enabled artists to describe *procedures* that could be autonomously executed by a machine, independent of the artists' control. As a result, machines could produce works that were not manually executed by human hands, nor explicitly conceived of by human minds (Reichardt, 1969).

Corresponding with the growth and diversification of computational technology, *procedural art* now encompasses a wide range of practices and possibilities for artistic creation. Most procedural art is created through computer programming. Describing work through code enables artists to manage complex structures, automate processes, and generalize and reuse operations (Reas et al., 2010). Procedural art has also resulted in new kinds

of artworks. Art created with programming can be autonomous(fig 1.3), interactive (fig 1.4), parametric (fig 1.5), and generative (fig 1.6). In addition to enabling new forms of expression, the act of programming in itself supports specific ways of thinking about making art. Rather than produce discrete artifacts, artists can conceive of works as systems that contain numerous, or even infinite variations of an idea (Mitchell, 1990). Programming offers new forms of documentation, by enabling artists to describe their creative process as a series of unambiguous actions that can be executed in different contexts or scales (Reas et al., 2010). Programming affords new forms of artistic collaboration, as artworks described as programs can be shared with large groups of people, who can reproduce, remix, and redistribute them with no perceptible loss of quality (McCullough, 1996). Finally, just as manual art supports distinct forms of thinking and reflection, the process of programming offers opportunities for engaging with powerful ideas (Papert, 1980), and seeing the world in new ways (Kay, 1990).

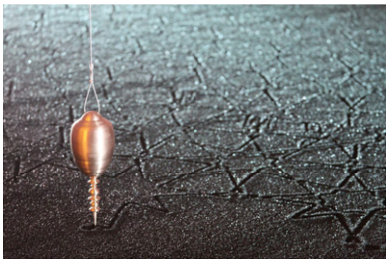

Figure 1.3: Entropic Order by Laleh Mehran, interactive installation comprised of a 2-axis "drawing" machine and 2,000 pounds of Black Beauty sand. `lalehmehran.com`

Figure 1.4: Mimus by Madeline Gannon, Julián Sandoval, Kevyn McPhail, and Ben Snell, interactive installation featuring an industrial robot arm that responds to human movement and gesture. `atonaton.com/mimus`

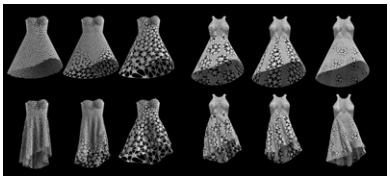

Figure 1.5: Nervous System Kinematics Dress by Jessica Rosencrantz and Jesse-louis Rosenberg. Dress variations produced using a parametric model and procedurally generated pattern of interlocking 3D forms. `n-e-r-v-o-u-s.com/projects/sets/kinematics-dress`

Figure 1.6: Selection from Clouds, series of generative prints by LIA. `liaworks.com/`

Yet, despite the creative opportunities of procedural art, it imposes barriers for practitioners of more traditional art forms. While art alludes to symbolic concepts, the act of producing art has traditionally involved manual manipulation of concrete media. As Mumford described it: "art uses a minimum of concrete material to express a maximum of meaning" (1952). Concrete artworks act as both the "interface" for art production, as well as constituting the work itself. Goodman illustrated this idea by distinguishing the notional nature of musical scores from the concrete qualities of sketches and paintings:

Unlike the score, the sketch does not define a work, but rather is one...No pictorial respects are distinguished as those in which a sketch must match another to be its equivalent, or a painting match a sketch to be an instance of what the sketch defines (1968).

Like musical notation, computer programs act as a description of an artwork, with the key difference that programs are designed to be executed by machines, not humans. Computer programming therefore, is a representational practice where artists edit a description of the work, rather than the work itself (Victor, 2011). As a result programming imposes a separation between the human author and the artwork thereby restricting the ability for manual creation. This poses a limitation for many creative practitioners. Manual engagement offers unique and rich forms for individual expression. Mumford stated that through the subtleties of the lines and forms created by different artists, we are able to experience *significant impulses that can come forth in no other way* (Mumford, 1952). In addition to enabling unique aesthetics, manual creation supports distinct forms

of thinking. Through analysis of designers in action, Schön demonstrated how manual and physical engagement played a key role in the process of generating new ideas (1992). Klemmer et al. document how gestural expression and epistemic action enable distinct forms of learning and understanding (2006). Overall, artists accustomed to working and thinking in terms of concrete media and manual engagement can encounter significant barriers when working with procedural tools.

Procedural tools also pose distinct learning barriers. Programming languages can restrict incremental forms of learning by requiring people to learn many concepts before accomplishing even simple tasks (Ko et al., 2004). Furthermore, while manual artists often learn through observation and action of tacit manual skills (McCullough, 1996), programming education often emphasizes a structured top-down approach of learning formal principles for application to concrete, predefined tasks (Harvey, 1991).

Artistic disciplines are shaped by their tools. The primary tools for procedural art, namely existing programming languages and programming environments, determine the kinds of creative practices that are supported or closed off when making procedural art. Programming, as it currently exists, has proven to be a remarkably powerful and expressive tool for many artists by enabling new kinds of artwork and encouraging new creative processes and mindsets. Yet programming also imposes barriers for people who are also invested in manual creation. Given that human cognition incorporates a variety of mentalities, including visual, kinesthetic, and symbolic, Kay argues that computational interfaces should support multiple ways of thinking because “no single mentality offers a complete answer to the entire range of thinking and problem solving” (1990).

The creative opportunities of computation; contrasted with the restrictions existing programming tools place on artists accustomed to visual, physical, and concrete working and thinking; suggest an opportunity for developing alternative programming tools that integrate procedural and manual creation. This objective raises the following research questions:

1. How can we create procedural art systems that are engaging, accessible and expressive for people with experience in manual art?

2. How can we support different ways of thinking and creating with representational mediums?

3. How can procedural art systems contribute to the process of learning and understanding representational mediums and tools?

To explore these questions, I developed two new tools for integrating manual and procedural art: *Para* and *Dynamic Brushes*. Para is a direct-manipulation procedural tool developed with the goal of supporting accessible but expressive procedural graphic art through a direct-manipulation interface. Dynamic Brushes is a system for enabling artists to create tools that extend manual drawing with procedural transformation, repetition, and automation. Dynamic Brushes builds on lessons gained through evaluating Para to combine drawing by hand with procedural manipulation and automation. The development of Para and Dynamic Brushes is informed through analysis of professional artwork produced through procedural and parametric methods, and through in-depth interviews with professional artists from a variety of domains. I use these tools as a platform to investigate methods and design principles for providing approachable entry points into procedural art, while retaining forms of expressiveness that are important to manual artists. I evaluated Para and Dynamic Brushes through a series of open-ended studies where professional artists created their own artwork and talked about their experiences.

1.1 Terminology

Throughout this dissertation, I describe distinctions and similarities between procedural and manual forms of art. Here I define a set of terms that I use to classify different forms of art practice in the remainder of this document:

- **Tool:** A device for making something.

- **Medium:** A domain or paradigm that affords distinct approaches to making. Mediums are often associated with specific tools. Computation and painting are mediums. Respectively, an integrated development environment (IDE) and a paintbrush are tools contained within these mediums.

- **Digital creation:** Works created using a computer. This includes both manual direct manipulation systems and programming.

- **Manual creation:** Works produced through human labor using concrete mediums and tools and hand-driven engagement. Painting, throwing a pot, and using a mouse to draw with direct-manipulation software are all examples of manual creation.

- **Procedural creation:** Works produced through a series of instructions

or a systematic representation. Common examples of procedural creation include works described with a computer program or works created using a parametric model. Procedural creation also encompasses manual human execution of instructions (Obrist, 2013). For the purposes of this dissertation, however, the focus is primarily on procedural works that involve unambiguous programs compiled and executed by a computer. I contrast manual and procedural processes as different modes of production, however it is possible for a single artwork to integrate both approaches.

- **Dynamic tools:** Digital tools that actively respond and change in response to actions and experiments by the artist. This definition references Kay and Goldberg's concept of *Personal Dynamic Media* (1977). Programming environments fit this description, however dynamic tools also encompasses systems that do not have an explicit procedural representation like a programming language, but still enable the artist to make systematic changes or describe procedural relationships. The most prominent examples include parametric CAD systems like Dassault Systèmes' SolidWorks or Autodesk's Fusion 360 .

1.2 Contributions

This dissertation makes the following contributions:

- A summary of the tensions and opportunities in integrating procedural and manual art creation, based on themes from interviews with professional artists.

- The introduction of two new systems for integrating manual and procedural creation: Para, a dynamic direct-manipulation tool that supports procedural art through live, non-linear, and continuous manipulation, and Dynamic Brushes, an integrated tablet and stylus drawing system and visual programming environment that enables artists to create and apply their own dynamic drawing tools.

- The demonstration of two expressive procedural models that are compatible with manual art practice. Para's model is comprised of declarative geometric and stylistic constraints, visually represented ordered lists that can be used to specify how constraints map to collections of objects, and declarative duplication to enable dynamic copying. Constraints, lists, and duplication can be combined in different ways to produce different outcomes. The Dynamic Brushes model enables artists to describe procedural brushes using a combination of a state

machine with event-driven transitions, and declarative constraints on brush properties. This structure supports brush behaviors that operate both *in-response to* and *independent of* manual drawing by the artist.

- An evaluation methodology that examines the expressiveness of prototype art-creation systems in extended, open-ended practice with professional artists. This methodology is applied to the evaluation of both Para and Dynamic Brushes to examine how professionals work with both systems, and what kinds of artifacts they produce.

- A set of recommendations for building expressive procedural tools for manual artists, and a comparison of the creative trade-offs between representational and concrete dynamic systems, as determined from evaluation.

1.3 Dissertation Roadmap

Chapter 2 Compares different dimensions of manual and procedural art creation by synthesizing interviews with manual and procedural artists with prior research on learning, computational design, and traditional art and craft practice. In this chapter, I describe different tensions and opportunities that emerge when integrating manual and procedural approaches through three dimensions of creative practice: learning, process, and expression.

Chapter 3 surveys existing systems for procedural creation. I begin by looking at textual and visual programming languages developed for art and design applications. I then discuss dynamic direct manipulation systems—tools that enable people to create and manipulate procedural relationships by selecting and manipulating icons and images rather than writing code. I look at a variety of approaches to inference-based design including programming by example, and artificial intelligence and machine learning-based approaches that automatically produce designs based on a set of examples or parameters. Finally, I look at systems that are aimed at helping people to learn programming by making personal creative projects.

Chapter 4 describes the procedural direct manipulation software Para. I outline Para's design guidelines and describe the iterative design process that led to Para's procedural model of constraints, lists, and duplicators. I then describe Para's interface and functionality and demonstrate sample applications. I describe Para's evaluation in a breadth-based short-term workshop with multiple artists. I follow by describing an alternative study

method where a single professional artist used the software over an extended period of time. I conclude by describing how a small number of procedural constructs can provide value to professionals while reducing the learning threshold of procedural tools.

Chapter 5 describes the development, functionality, and evaluation of Dynamic Brushes. I begin by discussing how Para's success in engaging manual artists, combined with its limitations in supporting physical drawing motivated the development of a procedural tool specifically designed for drawing. I describe the process of prototyping and refining Dynamic Brushes' states and constraints-based model through preliminary tests with artists and sample brush behaviors. I follow with a detailed description of the Dynamic Brushes programming model and demonstrate its expressiveness through different brush behaviors. Finally, I describe the results of an extended study with two professional manual artists that builds on the methodology I piloted in the Para study.

Chapter 6 compares Para and Dynamic Brushes and examines the results of both evaluations through the lens of my original research questions. I use observations of the learning process of the artists in my studies and the stylistic diversity of the work they produce to describe the different ways in which direct and representational procedural tools enable accessible entry to procedural art creation, and how they support different forms of expression. I build on the experience of prototyping, developing, and evaluating both Para and Dynamic Brushes to recommend general approaches for designing procedural tools for manual artists.

Chapter 7 concludes by reflecting on ways to extend this research to new domains and new forms of computational creation. It begins by looking at future work suggested by Para and Dynamic Brushes. This is followed by a discussion of the implications of artificial intelligence for artists. The chapter concludes by demonstrating the value of multidisciplinary knowledge in developing creative tools and the importance of supporting people from non-traditional computer science backgrounds in developing their own computational systems.

2

Dimensions of Manual and Procedural Practice

In procedural and manual art disciplines, different mediums have produced distinct communities of practitioners, each with their own values, practices, and attitudes. In addition, individual creators within a discipline often have their own approaches and possess unique perspectives drawn from their personal creative process. My own transition from drawing, painting, and manual animation, to programming highlighted the contrasts between manual and procedural art practice, and the different creative opportunities offered by both domains. Programming also changed how I thought about making art; I began focusing on systems that could create multiple outcomes as opposed to my prior focus on specific artifacts. At the same time, I encountered tensions when transitioning to procedural tools. My technical drawing and painting skills were irrelevant when working with code, and I had difficulty maintaining the style of my manual illustrations to my procedural work.

As I transitioned from creating art to researching the development of creative tools, I began to talk with other artists. I was curious about how their perspectives on procedural or manual art differed from my own, and what they perceived to be the primary opportunities or challenges of both mediums.

I formalized these conversations into a series of in-depth interviews with procedural and manual artists. I spoke with four artists who worked exclusively with manual tools and three artists who incorporated programming into their work. In this chapter, I describe the primary themes that emerged from these interviews, organized into three dimensions of artistic practice: *learning*, *process*, and *expression*. In each dimension, I compare the experience of my interview subjects to previous examinations of creative practice in computer science, human-computer-interaction research

and studies of traditional art and craft practice. The learning section highlights the importance of physical engagement and transparent tools for manual learning, and demonstrates the steep learning thresholds of textual programming. The process section illustrates how both procedural and manual artists rely on exploration at different points in their process and reveals how manual artists place special importance on being able to work concretely. The expression section describes how procedural artists are drawn to programming because it enables them to create dynamic artifacts, and how manual artists are hesitant to use programming because it restricts their ability to create human aesthetics.

2.1 Learning

Learning is a core aspect of art creation, both for students and professionals. As professional artists take on new projects, they must often learn new techniques and work to refine existing skills. Many artists aspire to reach a point in their practice where their tools become transparent, meaning their methods of creation no longer interfere with achieving creative goals (McCullough, 1996). Artistic mastery is relevant to both manual and procedural art; however, the process through which skills are learned, and mastery is obtained is often different for manual and procedural artists.

Figure 2.1: Dance Everyday by Shantell Martin, 2017.
`shantellmartin.art`

2.1.1 Physical learning

Manual artists often focus on learning physical skills. This generally involves learning how to effectively manipulate a material, (ink, paint, clay, or wood for example), with one's hands or with manual tools, to produce a desired result. Manual manipulation itself can support specific forms of learning and understanding. Holding and manipulating a physical object can improve someone's ability to think about 3D space in general (Sennett, 2008). Physical and gestural engagement can also play a role in learning. Sennett describes how many traditional crafts lack abstract concepts altogether. There is no separation between understanding and practice. When translated to learning, rather than explicitly learning new concepts, manual artists refine skills and develop tacit knowledge through extended practice over time (Roedl et al., 2015). In many cases, instructors find it difficult to teach manual techniques through verbal instructions. Instead, they encourage students get a physical sense of a technique through direct material engagement (Needleman, 1979).

The manual artists I spoke with repeatedly described the role of manual engagement in learning. **Shantell Martin** is an artist who draws large free-form drawings with acrylic markers (fig: 2.1). Shantell started drawing as a child and transitioned to drawing as form of live performance early in her art career. Today, she draws "on everything", from walls, to clothes, to people. In our conversation, Shantell described how the process of drawing helped her learn to trust her physical actions and her intuition. Shantell favors working with permanent drawing tools because she sees learning opportunities in her mistakes:

I like drawing with inks, because you can't erase what you do..if you're trying to learn from [your mistakes], you have to live with them. And if it's very easy for you to undo them or erase them you don't have that time to sit with them and realize what you've done wrong.

Shantell also spoke about working in pencil and how the tactile experience of manually erasing mistakes acted as a learning opportunity:

To physically undo something you would have to erase it. You would have to go back over what you've done and be like, "Uh, that didn't feel right." And you physically have to undo what you did. I think that tactile, more physical experience lent for a [gradual] learning curve.

Figure 2.2: too much by Michael Salter, installation image, Rice Gallery, 2008. `michaelasalter.com`

Michael Salter is a studio artist who creates illustrations, animations and sculptures with a strong graphic style (fig: 2.2). His practice is a combination of manual drawing, digital design, and sculpture. In addition, Michael is a college professor and teaches courses in digital illustration for fine-art students. Like Shantell, Michael described a strong connection between physical drawing and developing an understanding of style, composition, and proportion. When introducing his students to digital illustration,

Michael encourages them to first focus on building up their manual drawing abilities because he believes that jumping straight to the computer without manual skill negatively affects an individual's style:

When I'm teaching people how to draw on a computer they skip that part. They just want to go to the computer. And what they've done is they've diluted and changed their idea... they've inhibited it, because they don't know the program as fluidly as their hand can draw... That's when the computer to me is bad for drawing.

Nina Wishnok is a fine-art print-maker and professional graphic designer (fig: 2.3). Nina described the importance of being able to view and manipulate physical versions of her design work in learning about proportion and composition:

Your sense of visual relationships is different... You like something on screen, you print it out and the type is too big. You do have to see it physically. The more experience you get, the more you can sort of make those translations even if you're working exclusively digitally.

Shantell, Michael, and Nina all work with digital tools in portions of their practice, and they all emphasize the importance of learning through physical engagement before transitioning to digital tools. Their experiences align with Klemmer et al.'s description of how physical manipulation allows for different kinds of learning. Physical manipulation of forms can help people understand formal relationships and gestural actions can greatly assist cognition and communication. Physical manipulation also supports epistemic action where people are able to understand the context of a problem by physically tinkering with aspects of their environment (2006). Inexperienced artists may also have an easier time learning about composition, structure, and style through physical manipulation because their hands and bodies are their most familiar interface. Working with digital tools can make some things easier, for example, drawing precise geometry, but it adds an additional cognitive load of learning to manipulate the controls of a digital interface (Norman, 2002). Physical drawing enables learners to focus on developing confidence in their gestures and refining their style.

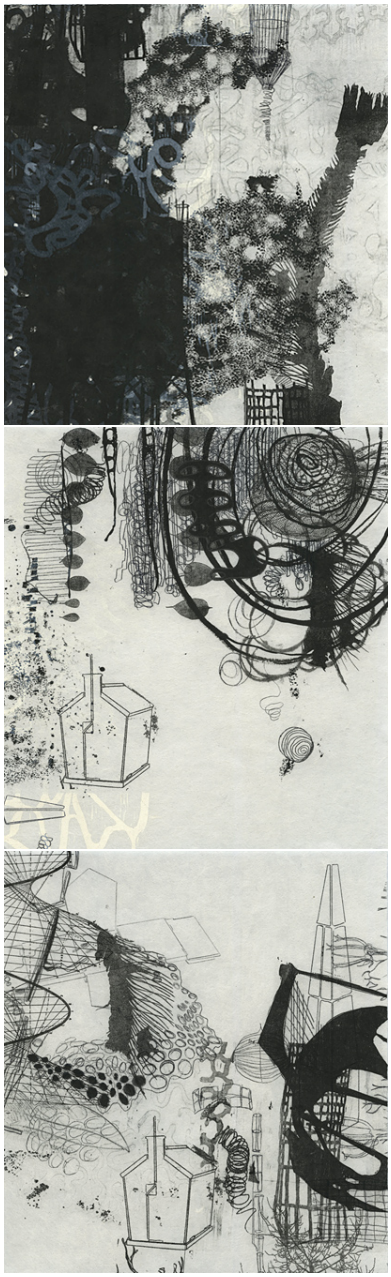

Figure 2.3: junk by Nina Wishnok, series of 5 monoprints (3 shown), 2012. `ninawishnok.com`

2.1.2 Perceptions of procedural learning

Procedural art involves learning programming, a domain that often is associated with learning analytic skills and practices. As a result, many forms of

programming can be challenging to learn for people with intuitive learning styles. College-level computer science students are introduced to programming through a structured methodology. They are taught subroutines, blocks, control structures, and other concepts in a specific sequence (Harvey, 1991). An unfortunate consequence of this approach is that learners might think programming has one correct way of accomplishing an outcome, instead of the possibility of multiple approaches. Turkle and Papert describe how students in introductory computer science courses struggled to use established programming architecture and workflows as starting points for learning. In the worst cases, this could result in students becoming alienated from the process of learning to program altogether. They describe a young female poet discouraged by the requirement that her programs be "black boxed" because she preferred to work with transparent systems of her own creation (1992).

The manual artists I spoke with described similar learning challenges. Shantell talked about her appreciation of drawing as a transparent art form, where observers could openly see all of the aspects her work. In contrast, she saw technology as deliberately opaque:

I think when we focus so much on efficiency, we forget sometimes about creativity…In the lab, you create so much technology and then you hide it away. You make something and then you put a pretty box over it.

Nina viewed Processing, a popular programming language for art and design, as a tool that imposed strict aesthetic limits for learners, similar to the process of using pre-programmed Photoshop filters:

I'm more interested in learning something technically that isn't just a visual end unto itself. I want to be able to incorporate it somehow and do something personal with it. [Processing], like any tool, has these limits of what you can do or you're stuck within the aesthetics of that tool initially before you can explore different areas.

Nina's perception of the limits of Processing is understandable, given the challenges of learning programming. For experienced programmers, programming offers a great deal more freedom than pre-programmed digital styles and effects; however, this freedom is inaccessible for many people.

The perception of computational technologies as opaque and programming as exclusively conceptual, stems in part, from how programming is conventionally taught and represented. Fortunately, learning programming can also be structured in ways that are exploratory, project-oriented,

and even physical. Papert describes how computers and computer programming offer unique learning opportunities by enabling people to explore and experiment with powerful ideas in mathematics, logic, and other domains. Papert also focuses on the importance of the body in learning. The programming language he developed for children, Logo, uses body-syntonic geometry in which movement commands are structured from the point of view of a computational "turtle" entity. By identifying with the Turtle people can use their knowledge about their bodies and how they move into the work of learning formal geometry (1980). Learning programing can also encompass a variety of skills, knowledge, and forms of engagement. Brennan and Resnick demonstrate how, in addition to learning concepts, programming also involves learning new practices, like debugging and remixing, and presents learners with pathways to new perspectives about the world and themselves. They advocate for forms of assessment that target different dimensions of computational learning, including portfolio analysis, artifact-based interviews, and design scenarios (2012).

2.1.3 Learning Thresholds

In comparing learning approaches in procedural and manual tools, it also is important to examine how much new information a learner needs before achieving meaningful results. While requiring years to master, pencils, pianos, and many other manual tools offer newcomers easy and intuitive starting points (Levin, 2000). Conversely, learners often encounter thresholds in programming where they must learn a great deal in order to produce simple results. Myers et. al. describe how most programming systems have "walls" where one must stop and learn many new concepts and techniques to make further progress, rather than systems where, for each incremental increase in the level of customizability, the user only needs to learn an incremental amount (2000). These severe learning thresholds are common in end-user programming and can present insurmountable barriers to people who are just beginning to learn to code (Ko et al., 2004).

Figure 2.4: Painting by Kim Smith. `kimsmithart.com`

Nearly every manual artist I spoke with described their experience encountering thresholds or barriers when attempting to use procedural tools. **Kim Smith** is a studio artist who creates large abstract paintings (fig: 2.5). She uses also uses digital design tools in her graphic design and illustration work, but primarily relies on physical media in her fine art. She has mastered the process of painting:

I love painting because I've been doing it my whole life, and I've put in those hours to the point where I don't have to think about it. When you get to the point where you're no longer fighting, that's the point where it's magical.

Figure 2.5: Painting by Kim Smith. `kimsmithart.com`

Kim is interested in the creative potential of procedural tools; however, she is frustrated by how these tools hinder her established practice:

One of the reasons I never use Processing is because I sit down and I don't know what I'm doing. When something gets in the way between you and making, it just stops. So I can make a square, I can make some shapes, ok great. But that doesn't do anything for me in terms of my artistic expression. It's like this big wall that is just slowing me down. I would imagine if I [were] good with Processing there would be a lot more I could do with it, and it'd be incredible. But the entry point for me is high. For a long time it's going to be in my way.

In-line with Kim's comment, Nina described an imbalance between the difficulty of learning procedural tools and interesting aspects of work possible using them:

When I start playing around with those tools, what I come up with or what happens visually is so clearly novice it doesn't even interest me visually... Depending on the software, that can be too much of a [learning] curve to really incorporate it.

Kim and Nina's comments demonstrate how manual artists can become stymied by the amount of effort required to produce simple procedural work, let alone expressive compositions. When thinking about supporting more approachable forms of procedural creation, a theoretical holy grail is

a system that enables immediate intuitive engagement while supporting unbounded expressive potential. Levin defines this property as "instantly knowable and infinitely master-able", and points out how the tradeoffs between accessibility and expressiveness make this quality near-paradoxical and often unobtainable (2000). Furthermore, Kay argues that no matter the system or language, many procedural concepts elude intuitive discovery. Things like recursion, abstraction, and inheritance are "like the concept of the arch in building design: very hard to discover if you don't already know them". He argues for simple and small systems that have a good match between the degree of interesting things one can make and the level of complexity needed to express them (1993). Like Kay, Victor recognizes programming as "a way of thinking" rather than a technical skill. As a result, he argues that learning programming will always be challenging because learning abstraction is hard (2013b). Victor has encouraged developers to create programming tools that make that challenge more tractable by encouraging powerful ways of thinking and enabling programmers to see and understand the execution of their program (2012b).

2.2 Process

Tools shape how artists work by encouraging certain processes and limiting others. In addition to shaping artifacts and aesthetics, (discussed in the next section), different artistic processes offer distinct forms of intellectual engagement and affect how artists conceive of ideas. An engaging process can encourage an artist to tackle challenging projects; a frustrating or dull process can demotivate an artist entirely.

In this section, I examine the forms of creative process that are important to manual and procedural artists, and describe how different tools support or limit their preferred forms of working. I describe the importance of continuous action, for both manual and procedural artists, how procedural tools and manual tools limit or support exploratory processes, and how concrete manipulation and formal representation shape artistic practice.

2.2.1 Continuity

Many of the artists described the importance of continuous action. Both Shantell and Michael described immediacy as one of the most powerful properties of manual drawing. Michael described how the power of drawing was centered on the speed at which his arm could give form to ideas:

When I see something in my head, the quickest way to get it into reality is that three feet, the 28 inches of muscle and bone. And I think that's magic... Right now for me, the freshest and easiest, and I think, the most exciting thing to do is to start a drawing and [not] stop until it's done. And because my work is primarily visual, one drawing influences the next.

Michael only started working comfortably on the computer when he could reproduce that speed that he had in his physical drawing. Similarly, for Shantell, the ability to work quickly and without hesitation is what shaped her overall style:

For me drawing is mostly movement. The way I draw is very head-to-hand. There aren't that many steps in between. Emotionally you want that channel to be as smooth and as free as possible. So I think the physical reaction is that you're allowing yourself to be free, you're allowing yourself to be spontaneous, you're allowing that movement to be a fluid as possible.

Coming from a background in performance, Shantell discovered how drawing in front of an audience pushed her to work without hesitation, and improved her work:

If you do stop, an audience isn't going to wait around for you. So you put this pressure on yourself where you just have to create and you don't essentially know what you're going to create because now you don't have time to think about it, you don't have time to plan it, you don't have time to hesitate, you don't have time to get distracted... it gives me that pressure, and I've been able to use that as a tool to drive me to do what I do.

In situations where there is no audience, Shantell described how she would turn a camera on herself and begin recording before she started drawing. The mere idea that she was under observation helped her work in a fluid manner.

Manual artists also value the efficiency that comes from continuous action. **Mackenzie Schubert** illustrates comics and graphic novels (fig: 2.6). Because of the high volume of illustration required for his work, speed is essential. As a result, Mackenzie doesn't have any nostalgia for traditional tools and materials. He is open to any process or technology that enables him to quickly complete pages:

If it's faster, then it's good. I don't think the return is three times better if it takes three times longer. I think the opposite is true, if there's a faster way to do something on the computer that looks ok, I don't think taking twice as long to do something the other way is twice is good.

Figure 2.6: Illustrations by Mackenzie Schubert.
`mackenzieschubert.tumblr.com`

From Mackenzie's perspective, speed is inherently linked to quality output. The faster he's able to execute an illustration, the better the aesthetic of the output. He described how he stopped drawing fingers on many of his characters, both because it was faster, and because he liked the visual effect it produced. Overall, the desire to reduce the time it takes to draw something significantly impacted the style of his artwork and pushed him towards a minimalist quality:

A lot of that reduction in time or effort, which are essentially interchangeable, makes work better or simpler, or more direct. It's easier to connect to or read... It's quicker and it's more immediate, and I think that shows. It can also feel a little more confident and less fussy.

Figure 2.7: Still from Secure Shell Series by Christopher Coleman, 2017.
`digitalcoleman.com`

Manual artists value tools that enable them to work quickly, continuously, and with confidence. These qualities improve outcomes and enable artists to intuitively act on ideas as they emerge. Although they work in a very different medium, many of the procedural artists described how programming also supported a quick and continuous process.

Christopher Coleman is an artist and professor working with wide range of emerging technologies (fig: 2.7). He comes from a background in engineering and uses programming to create generative digital forms, animations, and interactive installations. He is a colleague and close friend of Michael, and the two collaborate on projects that combine procedural and manual techniques (fig: 2.8). In our conversation, Christopher talked about how working with code helped him rapidly experiment with different versions of an artwork and iteratively revise his approach. He described how this was important when he collaborated with Michael:

I could quickly change what I was trying to do, change a lot of pieces, make a new piece of software, run the new thing through it. So, it enabled the speed of an exchange that you want in an art collaboration. I love the fact that I have just enough proficiency with Processing that in a day we could produce 20 different interesting iterations and then have a longer dialogue about successes and failures and ways to change and ways to improve.

Figure 2.8: Evidence of Intensive by Chris Coleman and Michael Salter, Digital Archival Print, 2013.
`digitalcoleman.com`

I also spoke with **Emily Gobeille**, an artist who creates procedural illustrations and immersive interactive installations (fig: 2.9). Emily has a background in both manual art and puppetry and has prior programming expertise. Her work is almost always done in close collaboration with her partner Theo Watson, a fellow artist and expert programmer. Despite favoring manual illustration with traditional media, Emily acknowledged that many illustration tasks that were much faster to create with code than by hand, and she often took advantage of this in her work. Emily also de-

scribed how collaborating with Theo speeds up her process. While Emily has experience writing her own computational tools, she often relies on Theo to create the procedural aspects of their work because he's the faster programmer:

He developed tools for me that I could actually use to draw. So it was kind of a mix of things he was outputting, things we were inputting. It was kind of a fun process . . . we wanted this whole kind of cause and effect.

However, Emily pointed out there were trade-offs with regard to speed and efficiency between manual and procedural tools. Just as Emily relied on Theo to complete some tasks with programming, Theo asked her to complete certain tasks manually:

Sometimes [Theo's] like, "Oh, you can do that in one second in After Effects", or something, where it would take him a really long time to program it. Then there are things that would take me forever to design, but he can do in two minutes.

Figure 2.9: Here to There poster series (1 of 2) by Emily Gobeille and Theo Watson, 2010.
`design-io.com`

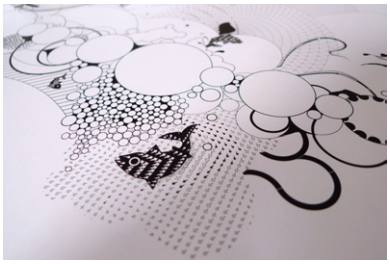

Figure 2.10: Closeup of Here to There by Emily Gobeille and Theo Watson, 2010.
`design-io.com`

Collaborations among people who are highly proficient at programming and manual creation offer the possibility for continuous engagement, where both members quickly respond to the ideas of the other. Yet, the success of manual-procedural collaboration is also dependent on artists having a close working relationship, mutual respect, and a deep understanding of the stylistic preferences of each other. For many artists, such collaborations are elusive. In situations where manual artists are working independently, incorporating procedural tools can slow down, or completely hinder the process.

2.2.2 Making the abstract concrete

In many traditional art-forms, *concrete engagement*, or manual contact with and manipulation of the piece, is the default. Tools involved in painting, sculpture, and printmaking, enable artists to lay down color, remove material, or otherwise modify a piece, by working on the artifact itself. Early computers were controlled through textual and numeric input, and prevented concrete manipulation; however advances in computer graphics and interface technology led to the development of interfaces where people could manipulate digital data by pointing at and selecting digital images. Shneiderman first coined the term *direct manipulation* to describe this form of computer interaction. Direct manipulation systems are defined by continuous visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest (Shneiderman, 1993).

Direct manipulation systems are now ubiquitous throughout computing, and they are especially popular as tools for art and design. McCullough describes direct manipulation as "the first glimmer of digital craft", arguing that while direct manipulation does not yet reflect all of the subtleties of physical touch, it is the closest approximation of physical craft manipulation in a digital system (1996). The emergence of haptic computing has led to research and development of computational systems that simulate some forms of physical or material feedback (Choi and Follmer, 2016; Bouzit et al., 2002; Nakagaki et al., 2016). The interviewed artists had different perspectives about the importance of physical versus digital manipulation. For Emily, the physical response was important. She preferred working with paper to working on a screen:

It's like a tactile thing. You need that real response, like if you push hard or if you push soft.

Michael and Mackenzie were not concerned about the differences between physical making and digital direct manipulation. Michael felt that he could work successfully both on and off the computer, as long as he was directly engaged in the drawing process:

I think drawing on a computer is like working in my studio. It is tactile to me. That idea of there being a line between the hand-made and the computer-made... that doesn't matter to me... Drawing on a computer is just as fun and tactile as doing it with charcoal.

While differing in their perspectives of the importance of physical ma-

terials versus digital ones, all the manual artists were unanimous about the importance of being able to work with concrete media and tools. In addition to being fundamental to manual forms of making, concrete engagement enables artists to discover new compositions, or make different associations. Lévi-Strauss described the concrete practice of bricolage: rather than thinking in terms of generalized knowledge and abstract relations, bricoleurs conceive of solutions through direct action with concrete tools and materials directly "at hand" (Lévi-Strauss, 1966). Concrete engagement also aides creative decision making; artists and designers often conceive of ideas through reflection-in-action and through direct engagement with physical media (Schön and Bennett, 1996) and digital interaction designers find it difficult to conceive of novel concepts while working in the immaterial domain of software (Ozenc et al., 2010).

Procedural art presents a challenge to concrete practitioners. The primary medium for creating procedural art, programming, is an extension of formal systems in mathematics and logic (Turkle and Papert, 1992). Programmers create work by manipulating a representation, which when compiled, produces a concrete outcome. The representational nature of programming is in part what makes it so powerful. Throughout the history of art, theorists have worked to create formal grammars that define critical, useful, and beautiful relationships between visual and physical objects (Goodman, 1968; Alexander et al., 1977; Mitchell, 1990). Programming provides the means to describe these grammars in a form that can be autonomously evaluated and executed. Applied to practical ends, this allows artists to manage complex structures and compositions in their artwork and define and automate repetitive processes that would be laborious to execute manually. (Reas et al., 2010). Programming enables work developed for one situation to be automatically restructured for another context or scaled to map to multiple applications simultaneously (Mitchell, 1990). Yet traditional programming languages also impose a separation between the artist and the artifact. These tools prevent concrete manipulation of the artifacts they produce during the creative process (Victor, 2011).

Although manual artists work through concrete practices, they often think about work in abstract or formal ways. Due to his background design and composition theory, Michael's artwork is guided by general design rules. He described how the imagery in his work is dictated by grids, cropping, focal points, and other compositional constraints. When asked about specific pieces, he articulated the structure behind them but also explained that a key aspect of what made his work interesting was being able to deliberately and arbitrarily break those rules at specific points:

[People] have this natural love of the grid. It's an aesthetic formalization of

math. And the only time that gets interesting is if the last one's left off, or one of them is missing, or the second row is all crooked. As soon as the grid is off a little bit, then it's interesting.

Michael did not identify as a programmer, however he described practices that were similar to approaches in procedural art. Michael described a series of in-progress pieces where he had established a rule set and then manually maintained it throughout the drawing. The result was a series of "generative" works:

I'm drawing a bunch of stuff that actually looks like code or generative, but it's not, it's filled with my flaws. But it has a short set of rules that I try to follow as I draw. I think there's an interesting idea of kind of taking a step back and trying to recreate that [generativity] by hand.

Unlike Michael, Shantell didn't approach her works with a formalized notion of structure and proportion. Instead, she possesed an intuitive understanding of spatial relationships. She characterized herself as someone who was mathematically inexperienced, but pointed out a paradox in that characterization:

I create these huge drawings in space, 200 [foot] drawings, and I also believe I have no idea about math. But, wait, how can I make this drawing physically feel balanced in space, and then feel that I have no understanding of math? Code, maths, drawing . . . it's very connected.

While drawing was primarily a visceral experience for Shantell, she had a strong desire to access quantitative representations of her drawings because she felt it might offer a different way of understanding her work. She viewed her drawing process as subject to specific rules and constraints established through years of practice. If the drawing process could be described as an algorithm, how could she uncover the underlying structure of that algorithm?

There is a huge amount of math, and spatial awareness, and composition in there, but I don't know what it is, and so, I'm really curious..what information is in a drawing that I can't see? What information is in a drawing, which I have huge, profound understanding of, but don't understand?

Nina, like Shantell also was interested in the procedural manipulation of information. Her printmaking pieces, while generated manually, were full of complex cellular structures and repeating patterns. She described a desire to extend this aesthetic in the form of data-driven imagery:

When I start to think conceptually, rather than purely visually, I do get more interested computational stuff—infographics and data driven graphics. I think it's really interesting to visually manifest [information] through programs. . . I'm intellectually interested in this stuff but I haven't personally found much of a bridge to move over into that direction.

Manual artists are interested in the opportunities offered by representational tools, both in terms of the creative domains they support and the opportunities they provide for introspection and analysis. Unfortunately moving to representational tools requires manual artists to abandon their concrete practices and mediums. Portraying the computer as tool for the mind (Krainin and Lawrence, 1990) can make the division between concrete tools and computational representation feel inevitable. But the computer is more than a tool; it is a medium that offers new ways of seeing and understanding the world (Kay, 1990). Turkle and Papert argue that the computer actually bridges the world of formal systems and physical things because it has the unique ability to convert abstract symbols to artifacts that people can see and interact with (1992). In thinking about practical approaches for bridging visual, figurative, and symbolic ways of thinking, Kay advocates for procedural systems that have links to the concrete. These tools should gradually scaffold the process of working symbolically by helping people explore symbolic relationships by manipulating images (1990). Building on these ideas, procedural tools for manual artists could be developed with symbolic models that are compatible with concrete manipulation or support transitions between working with a representation and working concretely.

Figure 2.11: portion of Creative Cloud by Erik Natzke, Commissioned Artwork, 2012. `eriknatzke.com`

2.2.3 Entry-points to exploration

Throughout the creative process, artists often explore different approaches, aesthetics, and ideas. Erik Natzke is a principle artist in residence at Adobe Research (fig 2.11). He creates sweeping gestural compositions with custom software of his own design. The software enables Erik to draw freely with a digital stylus, while the software modifies the color and quality of his strokes by referencing various input data. For example, the software samples color from different pixels of a pre-determined bitmap image to dynamically change the color of the line as he draws. Erik has structured the software in a way that exposes different parameters that control various visual effects. As a result, he doesn't start out a composition with a specific goal. Instead, as he works, he tweaks these parameters to experiment with different styles of drawing, which eventually help him refine the composition.

Like Erik, Kim also produces large compositions with gestural forms and color variation. Unlike Erik, her process involves physical paint and canvases, rather than software and code. When painting, Kim repeatedly lays down abstract layers of different-colored semi-transparent paint. This process allows for new shapes and colors to emerge through the layers. As she works, Kim repeatedly steps back to examine how the composition is changing. Similar to Erik experimenting with parameters of his software, Kim uses these emergent forms push the direction of the composition of her overall painting, and determine where she applies paint next.

Creative exploration is a fundamental part of the artistic process in both procedural and manual art. Whereas the ideal engineering approach starts with a clearly articulated problem statement and the removal of ambiguity (Crutzen and Kotkamp, 2008), art students are instructed to keep options open, explore alternatives, and celebrate ambiguity. This is because art creation rarely centers on a clearly stated problem (Do and Gross, 2007). Clear technical challenges may emerge in the process of creating artwork, but the overarching goal often is open-ended. Exploration can occur at different points in procedural and manual practices. For many manual painters and illustrators, the entire creation process is shaped by exploration, to the point that creative decisions are determined by results of successive experiments. Conversely, in procedural creation, exploration is interspersed throughout process along with structured forms of creation. Emily described how, for the procedural aspects of her and Theo's work, exploration often occurred at the end of a project. For example, when creating an interactive project, the majority of the work involved designing, developing, and testing different functionality. Once the project framework was built, they begin playing with the parameters of the system to produce the desired effect:

We'll just keep refining and refining. And that's to me one of the most interesting parts of what we do. The last two percent of our projects is just tweaking these variables, little teeny amounts, and it makes such a huge difference to what the experience actually feels like… our projects have a ridiculous amount of sliders, so we'll just sit there adjusting sliders. "Okay, this feels right… no, this one still feels like it's happening too fast," or "this is too big,"… We just adjust those things until it feels right, feels natural.

Manual disciplines, like drawing and painting, enable intuitive entry into exploration and support free-form association and recombination throughout the creative process. Drawing is an act of discovery where each successive stroke informs the next (Berger, 2008). Procedural systems may often intersperse the exploratory process with periods of analytic problem-solving and infrastructure development, but in return, they help

artists simulate near-infinite versions of a work (Kay, 1990), or systematically compare and experiment with different parameters without any risk to the work itself (McCullough, 1996). Because exploration can take different forms in procedural and manual art, one of the challenges of integrating manual and procedural practice is being able to move back and forth between either domain at arbitrary points in the creative process. When describing his desire to collaborate with multiple manual artists, Christopher talked about his concern that the other artists would be limited to only contributing to the prototyping stages of the project, and he would be responsible for completing the remainder of the project with computational tools:

[Michael], Justin and I have been trying to do a solid collaboration for years, but part of the problem is the throughput.. how does Justin's hand drawing stuff meet with Michael's digital stuff, meet with my coding-animation stuff? We're just trying to figure out the logistics of it. Justin did these amazing drawings and I was modeling them in 3D in Blender, but then it's like, is it all on me at that point? How do we make it sure that Justin's not just producing initial sketches and prototypes?

The challenge is to find ways to develop tools that help artists access both of these forms of exploration. One fruitful starting point in addressing this challenge is to design procedural tools that are "tinkerable". Tinkerable tools facilitate continuous exploration by emphasizing immediate feedback, easy modes of entry, and reconfigurable modular components (Resnick and Rosenbaum, 2013). Another approach is to give artists a head-start in the process of exploration by providing them with a catalog of pre-designed parametric structures that can be combined with one-another through a bricolage-like process (Efrat et al., 2016). This approach is limited in that it doesn't help artists create their own procedural functionality, but it does demonstrate how designing procedural systems to support recombination can enable a diversity of creative outcomes.

2.3 Expression

In computer science, the expressive power of a programming language is traditionally measured by the range of ideas that can be described with a given language. Measuring the expressiveness of programming for art applications is more complicated because it involves the breadth of different artifacts that can be created and how effectively a tool supports different aesthetics. For art tools, style matters as much as power does. In this section, I look at three different aspects of procedural expression: the creation

of dynamic artwork, the challenges of maintaining manual aesthetics when using procedural tools, and the ways in which procedural artists work to subvert computational aesthetics.

2.3.1 Dynamic Artifacts

Procedural tools enable traditional forms of art to be created in new ways. One can write a program that generates a painting, develop a parametric model that creates designs for a dress, or use a computer- controlled milling machine to create a sculpture. For people invested in these traditional forms of art, the ability to extend these practices with procedural tools is valuable and compelling. At the same time, procedural tools enable the creation of new forms of artifacts. Victor argues that because computers are dynamic tools, they offer powerful opportunities to create dynamic art—artwork that is responsive, interactive, and capable of change (2012c). The procedural artists I interviewed were drawn to working with programming because of its ability to create responsive artifacts. Christopher chose to incorporate computation into his art because it allowed him to create pieces that invited viewers of his work to engage in a level beyond observation. As he put it:

You can make arguments that sculpture and painting involve your whole body but . . . I think that sort of reactive work can only be done with technology . . . I'm really interested in that space. There's something visceral about that that I hope is affecting and helps you connect to it. The idea is stronger than just viewing or hearing something . . . That's a critical piece of why I'm involving so much technology . . . I want you to be in it, physically or otherwise.

For Christopher, creating interactive artwork supported his goal of engaging viewers of his artwork in a constructive dialog around issues of surveillance, online community, and public space, which are long-standing themes in his work. Emily is also strongly drawn to the dynamic qualities of procedural work. She expressed a strong attachment to manual and physical creation and frustration with aspects of procedural production. Yet Emily described how programming had one major advantage not available in any other medium:

Programming is what brings things to life. That to me is the most exciting part about it. [The work] can have a variety of inputs and processes, different ways to have some type of experience. You can't do it by hand.

In addition to creating dynamic artwork, several procedural artists described a secondary artifact that emerged from their practice: *tools*. Emily's collaborator Theo is one of the founders of a C++ creative coding framework called openFrameworks. Emily described how many openFrameworks add-ons contributed by her and Theo were the result of tools or processes generated in the process of completing work[1]. These tools are now used by others:

[1] Addons are contributed code libraries that extend the functionality of an existing system

As often as we can, we like to let people use the tools... It's nice to bundle [the code] up nicely for people to use. So I think quite a few have really ended up having a life outside of the project.

Christopher also saw opportunities for the byproducts of his artwork to support others:

What I have been trying to think of is how can my practice begin to spin off or strengthen tools. It's like, if I'm making a whole set of things to make my life easier, how do I share those instructions, or create a tool that helps other people do that thing?

For him, making computational tools was not only a by-product of making artwork but a central component of his artistic identity. Christopher described himself as a "critical arts engineer" with the responsibility to create technologically-driven tools for others to use. As he put it, his role was to "reclaim some of the black boxes and technologies that are being put out there for us to use." The role of computational artist as both artifact and tool creator resonates with Levin's point about the importance of artist-created computational tools. He argues that tools created "by artists for artists" fill a space not offered by commercial software by being more sensitive to the needs of artists and more open to supporting creative experimentation (2015).

There are many examples of procedural art tools that began as platforms to support the work of a small subset of artists, and later were evolved to support the broader procedural art community, many of which are discussed in the following chapter. The prominence and success of many of these systems demonstrate how procedural artists are well positioned to be tool creators. They have insight into the forms of expressiveness that will be most meaningful for other artists, and are often invested in the creative goals of others. The ability for procedural art to support tool development also demonstrates the inherent re-usability of code. Unlike many traditional mediums, programs represent an artwork in an abstract form with modular functions and processes. This quality, combined with the existing infrastructure for sharing code means that procedural artwork

is reusable and generalizable in a way that many other art forms are not.

It's not easy to make effective tools for a general audience, however. Emily pointed out that some procedural functionality ended up being too specialized to be creatively appealing to other artists:

Sometimes it's too specific. I don't know if it would be useful to someone.

For procedural systems to function effectively as a platform for others, they must support a wide range of interesting and meaningful outcomes. Tools often are only successful for a general audience when released with documentation and learning support, which takes significant time, effort, and resources to develop, and also requires tool-makers to take time away from making art. Christopher highlighted the difficulty of simultaneously making art and supporting others in our conversation. Making artwork can inform the process of making good tools. At the same time, there are prominent tensions between the mindsets and objectives of tool-making and art-making.

2.3.2 Human aesthetics

One measure of a tool's expressiveness is the range different kinds of artifacts it enables people to make. Another important measure is the range of aesthetics that are possible to produce through the tool. Levin describes how individuals have their own spatio-temporal signatures and unique ways of moving through space. Successful creative systems, he argues, preserve the traces of this movement (2000). Computational tools, while expressive in a multitude of ways, can limit the ability for these personal physical characteristics to show through in the final artifact. Many of the artists I spoke with expressed frustration with the consistent aesthetics of procedural tools. Emily described the importance of imperfections in her work. Even with random variation, she found computational generativity too consistent:

All the lines are too straight, everything is too even. There's not enough of the little inconsistencies that you get by doing things by hand that I think are really nice...when you program something, it's easy like, "Okay, 5,000 particles. They're all the same," and I'm like "No, you need speed variation. You need size variation, color variation...Even if you're adding random, it still doesn't feel different enough, so I end up creating six versions of the same thing to try to get that variation to come out.

Her strategies to overcome the consistency of computational artwork often involve pulling in textures and colors from the natural world:

I spend a lot of time taking photographs or scanning . . . Like, take a leaf off a tree or something. I'm always scanning in textures that then I use digitally, because I hate that kind of flat staticky feeling . . . I try to incorporate many non-computer elements into the digital artwork.

Through collaborations, Michael has observed the creation of a great deal of procedural art. His feelings about procedurally-generated variation were very similar to Emily's in that he understood the advantage of automatically producing alternatives, but he considered the differences uninteresting:

I'll get hundreds of versions of the same thing...and people think that that's a variety of options. Like, "Look at all these things that I made. There's 100 different things." No, those are all the same thing. They're just slightly different because the computer allowed this infinitesimally small increment of change.

He described his appreciation for generative work created by a computer but had a personal interest in exploring those aesthetics through manual drawing:

From a distance, it may look perfect or generative or formulaic, but really when you look closely at it, it's riddled with my wobbles and whatever was going on in my head at the time.

Shantell also valued the variations that emerge from hand-created works. She also identified another important aesthetic characteristic of manual drawing, namely the ability for works to uniquely inhabit a distinct place and time:

If I were to draw on my wall right now and then I was to draw my wall right in 10 minutes, it's two different drawings. Why is that, and why are they so profoundly different? I think there's something special about things being in the foreground, things being relevant, things being in real time, things being experienced as now. I think we somehow inside, have this knowledge that these moments are special, especially when they can't be repeated. It's almost like an anchor in space and time where you're like, "Wow, this is happening now and only now, and in this place and only in this place and it'll never be repeated."

Shantell's description of how physical drawings connect us to specific

points in time is evocative of Benjamin's concept of *aura*. Machine-created artifacts in Benjamin's view, lack a connection to a distinct time and place. Mechanically reproduced works of art therefore dissociate the original work from its aura, and remove its unique aesthetic authority (1968). Digital tools take this idea further by enabling an objects to be rapidly re-purposed from one context to another without any loss of quality or affordance (McCullough, 1996). Some critics consider machine-driven reproduction to be responsible for a reduction in the authority and power of art (Berger, 1972). Yet from the perspective of many artists, the transmissiblity of digital data is a creative opportunity, not a negative consequence. Michael described how digital tools have enabled him to translate one idea into a multitude of outcomes:

My favorite thing about digital creation is the variety of ways to realize [something] . . . Printing, cutting, dimensionally, projecting, animating. I can do so many things with the thing. That's what I love about it.. I can make a thing that can be a drawing and a sculpture and an animation, all the same.

Although he is describing re-contextualizing art using manual digital tools rather than procedural ones, his description coincides with the affordances of computational abstraction. While digitized data provides the means to translate content from different contexts, programming provides a structure to formally represent an idea in abstract form and unambiguously specify how different aspects of it should adjust to different contexts.

The desire to maintain the originality of manual human expression is difficult to reconcile with the abstraction offered by computational tools. Many computational tools prevent an artist from preserving imperfections and personal traces, render these qualities ubiquitous rather than unique, or position an artificial agent in the role of generating unlimited numbers of unique works[2], which in-turn reduces the value of each individual piece.

[2] See the section on inference-based design systems in the next chapter.

Technological change has always shaped art, and as Michael's comment indicates, new technologies make new forms of creation possible. In re-evaulating Benjamin's critique of machine reproduction within the context of computational culture, Nadin argues that aura has not been lost. Instead, it's shifted from the artifact, to the unique and personal process of each individual artist. (1997). Enabling a diversity of processes in procedural tools implies the potential for greater diversity in procedural aesthetics.

2.3.3 Finding the edge

Along with the challenge of preserving human variation, the artists in my interviews described another aesthetic challenge that is particular to working with emerging technologies: the requirement for artists to subvert the software in order to produce new and interesting aesthetics. Emily described the way software offers a great deal of power and new opportunities, but still results in homogeneous end products:

I see a lot of people building things in Unity[3] *and it's an amazing engine, but it kind of all ends up looking and feeling the same.. You have to have a little more flexibility. I just feel like there's less instances where I'm surprised and wowed by things that are done procedurally than things that aren't, like the things that people are actually constructing or that their hands have actually touched.*

[3] Unity is a software tool for developing digital games.

Christopher takes this idea further, arguing that there is an expectation for digital artists to subvert or push the limits of existing software or hardware:

Working in the digital media field necessitates walking on a nice place between creating work which is conceptually interesting but also which carves out new aesthetic space.. or not new aesthetic space but what's called "Hip Aesthetic Space." . . . as an artist you have to find the edges of the box and break it constantly.

He described a race that occurs among new-media artists whenever a new technology emerges, using the resurgence of deep learning, convolutional neural networks as a tool for image generation, and Google's Deep Dream as an example:

There's this weird race to find the new edges of the new box every time an update is pushed out or a new platform is pushed out, because you know all the easy stuff is going to be consumed into a more easy popular culture, and so it's kind of... Again, it's that weird extra race that new media artists must choose, constantly finding new aesthetics or what are the aesthetics that everybody is sort of working in . . . you need to push just beyond.

Even artists who did not directly engage in programming identified this quality of procedural art. Michael talked about procedural art aesthetics as "a moving target":

That trend and those aesthetics are also much more obvious...even though they're beautiful, to me they still mark a point in space, which is the edge.

And that, for me, is baggage. Those guys carry it. Chris uses cutting edge shit that he's either making, hacking, or helping to develop. He is on the edge. He's defining it. And that's something that [he] has to own. But it's not something I want to have to deal with.

Subverting a given object or material is not unique to procedural art. Examples in other mediums include Nam Jun Paik's re-purposing of televisions as a sculptural material, and Nathalie Miebach's application of basket-weaving as a mechanism for three-dimensional data visualization. However, computational technology has accelerated the rate at which emerging aesthetics becomes cliché. In *The Civilization of Illiteracy* Nadin describes how art has changed as as a result of computation and information processing.

Today art is produced much faster, embodied— or disembodied— in and disseminated through more media and exhausted in an even shorter time— sometimes before it even comes into being…And never before were more technological and scientific means involved in the practical experience of art, always on the cutting edge, not only because art is traditionally associated with innovation. These new experiences make possible the transition from an individual, private, almost mystical, experience to a very public activity (1997).

At present, the ability to re-configure or transform existing software tools and technologies is dependent the ability to program. Artists without prior programming expertise, outside assistance in programming, or suitable pathways to learning programming have difficulty creating aesthetically-unique procedural works. One pathway to addressing this is to develop systems that help more artists to develop their own software tools. A greater number of procedural tool-makers offers two benefits: it could empower artists to create tools that reflect their own personal preferences and styles and, it could result in an increase in the number and diversity of computational tools available overall.

3

Systems for Creative Procedural Expression

There are many different ways to engage people in procedural art and design. In this chapter, I outline different categories of systems for procedural creation, describe prominent examples within each category, and analyze creative trade-offs between these different approaches. I begin by looking at textual and visual programming languages for creative programming. I then discuss dynamic direct-manipulation and inference-based design systems, two domains that support the creation of procedural artifacts or enable the manipulation of procedural relationships without requiring people to write code. Finally, I look at approaches for helping people to learn programming in the context of making their own creative projects.

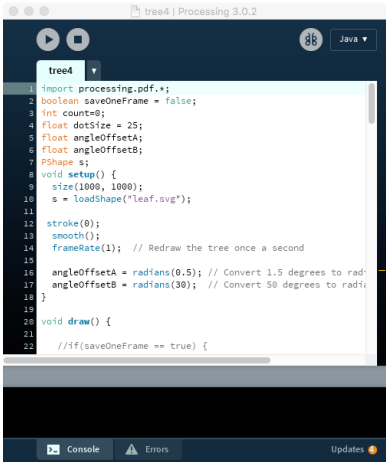

Figure 3.1: The Processing programming environment (top) and documentation listing all available methods. `processing.org`

3.1 Textual Creative Coding

A wide variety of programming frameworks and languages have been developed specifically targeting artistic production. These systems are varied in their applications and structure, but often have the unifying feature of being developed by people with interdisciplinary backgrounds in art and computer science.

A great deal of procedural art is created with general-purpose textual programming languages. To improve the process of applying programming languages to art, people have developed specialized frameworks, application programming interfaces (APIs), libraries, and IDEs that build on existing languages. These platforms simplify the application development process and offer streamlined access to functions for visual, audio, and interactive output. Processing, a Java-based IDE and programming framework created by Casey Reas and Ben Fry, is one of the most prominent

creative coding tools. Processing was inspired by John Maeda's Design By Numbers project (1999), and was designed to teach fundamentals of computer programming in a visual context. The design of Processing was informed by Reas and Fry's backgrounds in art, design, and computer programming Reas and Fry (2007).

Processing has evolved into a large community of practice and has been applied to fine art, commercial art, and design production. It also has inspired the development of other creative coding frameworks and tools. openFrameworks is a C++-based programming framework created in 2004 by Zach Lieberman, Theo Watson, and Arturo Castro. Lieberman initially used openFrameworks as a tool for his own professional work and as a platform for coursework at the Parsons School of Design in New York (2014a). openFrameworks was created to provide artists with low-level computational access to support computer vision, sound processing, hardware access, and other C++ libraries, and to serve as a platform for collaborative artistic production (2014b). Other prominent text-based creative coding platforms include Cinder, a C++ toolkit for computational graphics creation, originally developed by Andrew Bell, D3, a javascript-based library for data manipulation and data visualization created by Mike Bostock, and Arduino, a hardware toolkit and C++-based programming environment developed by Massimo Banzi, David Cuartielles, Tom Igoe, and David Mellis. The original Arduino programming environment built upon Processing, and the hardware platform was inspired by Hernando Barragan's Wiring platform (Arduino, 2017).

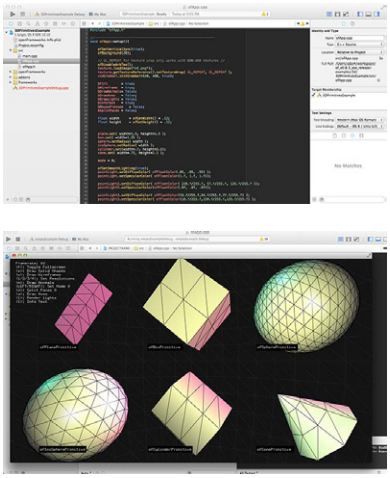

Figure 3.2: An example of an openFrameworks program in the Xcode programming environment (top), and the compiled application (bottom). `openframeworks.cc`

Textual creative coding platforms like Processing and openFrameworks have many advantages. Because they are based on general-purpose programming languages, they are extremely computationally expressive and highly extensible for people with programming experience. This extensibility, coupled with significant efforts to promote open-source development, enabled these platforms to foster large communities of external contributors (Lieberman, 2014b; Reas and Fry, 2007). These contributors developed new libraries that supported additional applications, thereby broadening the relevance of these platforms for different creative domains. Textual creative coding platforms, however, can present significant learning challenges for those new to textual programming. Similar to general-purpose languages upon which they are built, textual creative coding platforms require knowledge of textual programming conventions and use of lengthy programming libraries, which makes them subject to the same learning thresholds and barriers mentioned in the previous chapter.

3.2 Visual Programming

In recognition of the challenges of textual programming, people also have explored creating visual programming languages for art and design. While definitions of visual programming languages vary, for the purposes of this dissertation, they are defined as languages in which programming syntax is represented through images or graphical icons that can be directly manipulated by the programmer. For people new to programming, visual programming languages can provide a more accessible entry point into procedural creation than textual languages. Myers et al. describe how visual languages, by enabling direct manipulation of a programming syntax, provide people with the impression of directly constructing a program rather than abstractly designing it (1990). Depending on which visual representation they employ, visual programming languages may resonate with Kay's theory that active manipulation of images can scaffold the process of symbolic reasoning and understanding referenced in the previous chapter (1990).

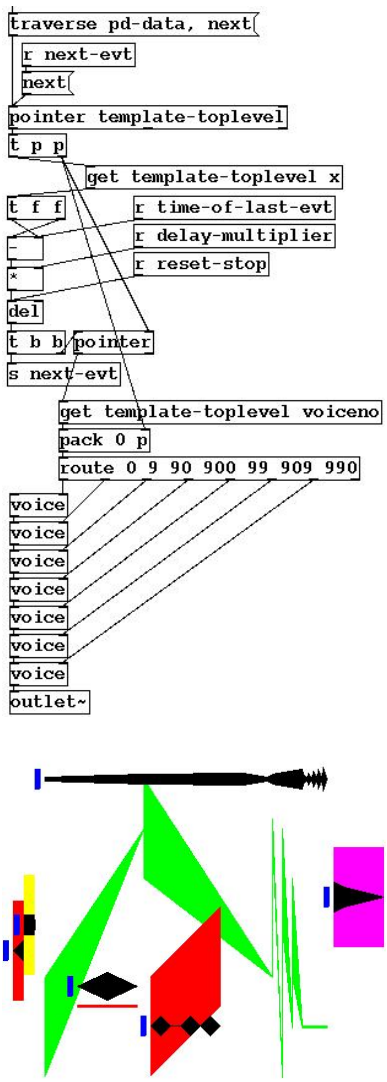

Figure 3.3: The Pure Data dataflow language. The patch (top) is sequences the graphical "score" shown above, mapping frequency and bandwith to the color and geometric properties of the shapes (bottom). `puredata.info`

3.2.1 Dataflow languages

Many of the visual languages for creative coding use a specific visual programming paradigm know as *dataflow*, in which programs are represented as a directed graph with each node performing an operation on whatever information passes through it. Dataflow programs are automatically executed when all the inputs register as valid (Johnston et al., 2004). In creative-coding contexts, dataflow languages treat incoming data, for example geometry, images, video, or audio, as a signal that is filtered through a sequence of operations to produce a desired outcome. One of the most established creative-coding dataflow platforms is Max, a commercial visual programming language that enables artists to procedurally process and generate sound, video, and graphics graphics through a series of different filters and processes. Max was derived from the Patcher visual programming environment developed by Miller Puckett in the mid 1980s (1988). Puckett also developed Pure Data from Patcher, an open-source visual programming language with functionality similar to Max.

Dataflow paradigms also can be applied to the creation of static artwork. Grasshopper is an add-on for the Rhino 3D modeling software, developed by David Rutten, which lets artists combine a variety of modules to generate and transform 3D geometry. Similarly, NodeBox is a dataflow environment created by the Experimental Media Research Group for generating

and processing 2D vector graphics, and has specialized functionality to process tabular data to produce data visualizations. From a computational

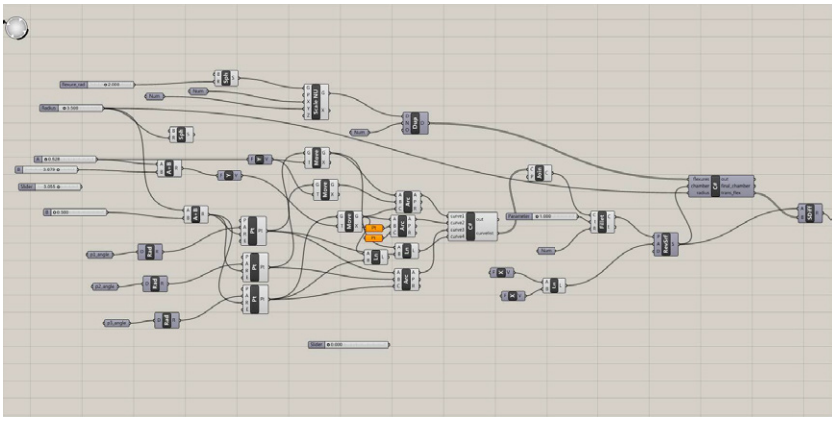

Figure 3.4: The Grasshopper language. Geometry is treated as a signal, filtered through a sequence of nodes that perform procedural transformations.
`http://www.grasshopper3d.com/`

perspective, dataflow languages have advantages and limitations. They are well suited to parallel processing tasks when multiple streams of input are being processed simultaneously (Johnston et al., 2004). This is particularly important for performance-based artwork because complex programs can still process data at a rate suitable for real-time output (Puckette, 1988). Dataflow models have some limitations for computational iteration. Many dataflow models do not support loops and therefore require different approaches for iterating over collections of data. This can pose a problem for programmers accustomed to working with imperative programming paradigms. (Johnston et al., 2004). Visual dataflow languages manage collections by making looping implicit. In systems like Nodebox and Grasshopper, nodes that receive lists of data execute their functionality for each element in the list. A similar limitation exists with recursion. While it is technically possible to create recursive programs in the dataflow paradigm, the one-directional structure of most visual dataflow languages makes recursion difficult to visualize and implement. Systems like vvvv get around some of these limitations by extending dataflow languages with imperative textual programming languages.

3.2.2 Block-based Languages

Block-based languages are another popular form of visual programming. Block languages are composed of visual shapes that represent variables, methods, and control structures, and can be snapped together to form programs. The shape and color of each block provides visual cues to how and where the block can be used. Block-based IDEs often enable creators to select from a menu of all available blocks to compose their programs (Wein-

trop and Wilensky, 2015). As a result, block languages can help new programmers avoid some of the challenges textual languages pose by enabling them to avoid syntax errors because blocks cannot be put together incorrectly, and by making new programming concepts more discoverable. Many creative block-based programming languages are designed for young people. Scratch is a block-based programming language and online community developed by the Lifelong Kindergarten group at MIT for creating storytelling and media-rich applications, including animations, interactive slide-shows, and games. Alice is another youth-oriented programming

Figure 3.5: The Scratch programming environment. `scratch.mit.edu`

environment developed at Carnegie Mellon University aimed at helping young people to learn core computer science concepts through the creation of 3D graphics, animations and environments using a block-based programming language derived from Python (Cooper et al., 2000). Scratch and Alice focus on the importance of providing learners with immediate feedback. Scratch programs are designed to be highly interactive. Programmers can click on a stack of blocks to execute the code immediately and can make changes to a stack as it is running.

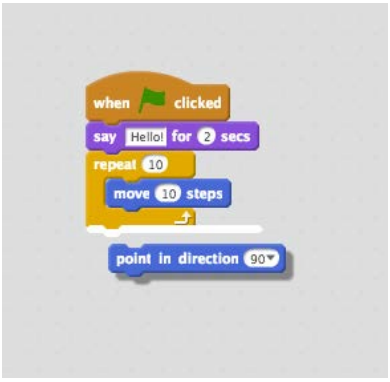

Figure 3.6: Snapping together blocks in Scratch to create a program.

In Alice, the results of each program action are animated smoothly over a specified duration, so that students are immediately able to see how their animated programs run. Scratch's visual programming language is designed to be highly tinkerable, where the shape of code blocks allows them to be snapped together. (Resnick et al., 2009). Block-based languages can also be applied to the creation of static designs. Turtle art, DesignBlocks and Beetleblocks are examples block based languages that can be used to create bitmap illustrations, 2-dimensional digital designs and 3D models respectively (Bontá et al., 2010).

The fact that many visual programming languages are created for children and new programmers does not mean that block based languages

are computationally inexpressive. All of the block-based languages described here are modeled after imperative programming structures and have comparable functionality to textual imperative languages. Furthermore, block-based languages can be used interchangeably with textual languages. Snap, Blockly, and Android App inventor are all examples of block-based languages that are designed to support programming tasks that are conventionally performed with textual languages. Snap was designed to be used in college-level computer science courses, App Inventor supports the creation of mobile applications, and Blockly is a general-purpose block based language developed by Google which can convert block-based syntax to another text-based language in real time.

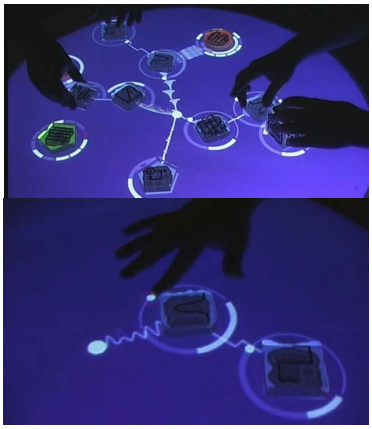

Figure 3.7: The reacTable interface. Top: multiple people manipulating the synthesizer, bottom: changing a parameter of a node.

3.2.3 Hybrid Visual Languages

Visual creative-coding systems represent an important initiative to provide an accessible entry-points into procedural art for people unfamiliar with textual programming. They also offer an alternative way of representing procedural relationships for artists and designers which can be extremely beneficial for certain mediums. One prominent example of how visual programming offer a new way of representing and manipulating an established artform is the reacTable: a physical ”dataflow” environment for electronic music production. The reacTable allows multiple musicians to manipulate physical blocks that represent different modular synthesizer components on top of a physical table. These components are automatically connected and disconnected based on their position on the table, and musicians can manipulate parameters by changing their orientation (Jordà et al., 2007). Oney et al.’s Interstate system supports user-interface (UI) development using a live visual notation that shows relationships between different elements of the HTML document-object-model. The visual notation of Interstate is structured on top of a programming model that combines declarative constraints and UI event-driven transitions between states (2014). Systems like the reacTable and Interstate are examples of how a carefully designed programming paradigm and environment can provide creators with access to procedural affordances and also support key aspects of an established practice.

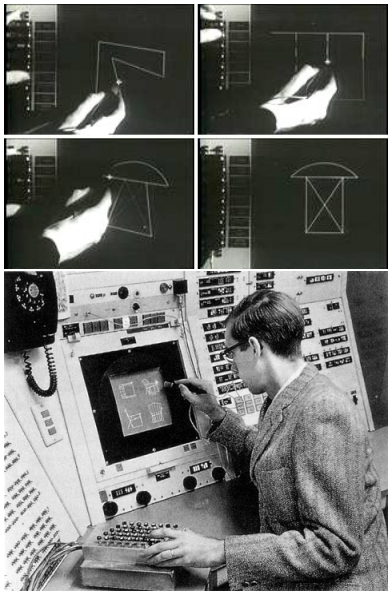

Figure 3.8: Sketchpad was one of the first graphic user interfaces ever created. It also demonstrated how parametric relationships could be created by manipulating images on a screen.

3.3 Dynamic Direct Manipulation

Visual programming languages can reduce some of the syntactic challenges of textual languages. Yet like textual languages, visual languages

require artists and designers to work through an abstract description rather than on a physical artifact. Another approach is to use direct manipulation as a mechanism to describe procedural relationships. While it is difficult to describe certain computational concepts in concrete form, direct manipulation can be used to describe dynamic relationships in ways that are comparable to programming languages. More-over, dynamic direct manipulation integrates some of the concreteness of manual tools and media with procedural expression. The earliest example of dynamic direct manipulation is SketchPad. Developed by Ivan Sutherland in 1963 using a light pen and x-y plotter interface, the software demonstrated how parametric relationships could be described by selecting and manipulating geometric shapes (1964). Sketchpad provided the basis for most parametric computer-aided-design (CAD) software that exists today. At present, dynamic direct manipulation has been applied to a wide variety of design applications. Victor, a prominent advocate of dynamic direct-manipulation systems, argued that they offered designers opportunities to create their own interactive systems rather than relying on engineers (2011).

Figure 3.9: Victor's Drawing Dynamic Visualizations. The left panel shows a recording of the actions taken by the designer. Designers can create dynamic visualizations thorugh a combination of modifying the steps in the recorded actions, and direct manipulation of the graphics in the center panel.

3.3.1 Dynamic Direct Manipulation Systems

The desire to combine the expressive potential of procedural relationships with the accessibility and intuitive quality of direct manipulation has resulted in a variety of dynamic direct-manipulation systems across different domains. Pygmalion was an early "iconic" programming language that mapped the visual characteristics of graphic icons described by the programmer to corresponding machine semantics (1975). Victor later demonstrated sample systems with applications in game development, interactive animation, and data visualization (2012a; 2012c; 2013b). Victor's presentations inspired Kazi et al. who developed systems to reduce the challenge of creating interactive infographics. Kitty and Skuid enabled the creation of motion graphics and basic animation through direct manipulation of a relational graph that is superimposed on the illustration (2014; 2016).

Figure 3.10: Xia's Object-Oriented Drawing system replaces traditional touch menus by enabling any sub-property of a graphic to be accessed like an object in a hierarchical structure.

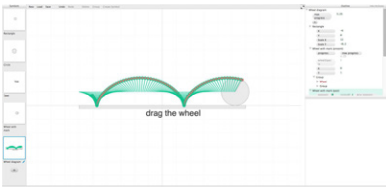

Figure 3.11: Apparatus is a Javascript-based tool for creating dynamic diagrams and data visualizations through a combination of selecting vector and manipulating vector graphics, and creating textual expressions.

For static illustrations, Recursive Drawing supports the creation of self-similar 2D artwork through an interface that allows artists to create designs on individual canvases and then nest these canvas in a hierarchical structure to produce recursive patterns (Schachman, 2012). Hoarau and Conversy demonstrated graphic design tools that enable designers to indicate dependencies between the properties of objects so that master objects can be used to update properties across an entire composition (2012). Blackwell created Pampliset, a direct-manipulation system for procedural

editing of bitmaps, where parameter values for bitmap transformations are represented as image layers and can be retroactively adjusted with changes propagating across a composition (Blackwell, 2014). Dynamic direct manipulation has also been applied to data visualization. Shachman and Horowitz's Apparatus (2015) enables the creation of interactive diagrams and data visualizations through an integration of direct manipulation of vector graphics and textual declarative expressions, and Kim et al.'s data-driven guides enable designers to generate graphical guides from textual data and then use direct manipulation to place and measure custom shapes with the guides to produce illustrated data visualizations (2017).

In addition to supporting the creation of art and design artifacts, dynamic direct manipulation can be used to create interactive systems, or support new interaction modalities. Xia et al. demonstrated how object-oriented principles could be used to reduce reliance on windows, icons, menus, pointer (WIMP) UIs for touch and tablet interfaces (2016). Xia expanded on this work with Collection Objects, a system for supporting dynamic selection of complex object-sets by unifying selection, grouping, and manipulation of aggregate selections into a single object that could be used to access distinct groupings of vector graphics and vector graphic properties (2017). Morphic is a UI construction environment developed by Maloney and Randall that allows developers to specify behaviors of user interface elements through direct manipulation of their position and ordering of their sub-components (1995).

3.3.2 Opportunities and Limitations of Dynamic Direct Manipulation

Dynamic direct-manipulation systems have the benefit of retaining many of the valuable qualities of static direct-manipulation systems. Unlike many traditional programming environments, they are *live* meaning that actions taken in these systems result in immediate visual feedback. Liveness in digital tools reflects the way objects in the real world respond to concrete manipulation, and is therefore an important factor in informing immediate and intuitive design choices (Maloney and Smith, 1995). In conjunction with liveness, dynamic direct manipulation systems often allow for continuous manipulation of procedural relationships without disrupting the underlying "program". This is in contrast to many representational programming systems, and in particular to imperative textual programming languages, where modifications to one portion of a program can result in unexpected side-effects, or produce errors if the modification is made to a part of the program that is referenced later on in the control flow (Abelson and Sussman, 1996). Many dynamic direct manipulation systems achieve this continuity by basing their interactions around declar-

ative programming, where people can express logical relationships but do not have to explicitly describe control flow, as is the case with imperative programming.

Dynamic direct-manipulation systems are limited in some ways compared to textual tools. Symbolic textual expressions can concisely and unambiguously represent complex relationships in ways that images cannot (McCullough, 1996). Furthermore, computational abstraction expressed through textual tools can greatly aid in complex organizational tasks. While the representational nature of programming can create a barrier for manual creation, it also offers opportunities for reflection. Code can simultaneously serve as a functional object and as a record of ideas and process (Levin, 2003); thus, writing code can enable active reflection on the relationships that define a person's work. Understanding abstraction is difficult (Victor, 2013a). Systems that enable artists to express relationships textually and then experiment with them through direct manipulation may scaffold learning and support new forms of creative expression.

Despite the creative potential of dynamic direct manipulation, there is still limited understanding of how this approach actually performs in creative practice. Much of the prior research lacks evaluation; Hoarau and Conversy are unsure if their interface is understandable due to lack of user evaluation, and Victor recognizes his tools are untested stating that they are "not necessarily the correct way of doing things" but rather a starting point for exploring new interfaces (Victor, 2013b). Many other tools have been evaluated, but focus on tool accessibility and efficiency through predefined tasks (Xia et al., 2016; Kazi et al., 2014, 2016). One of the objectives of this dissertation is to better understand the expressive potential of dynamic direct manipulation for professional art practice in comparison to representational procedural tools.

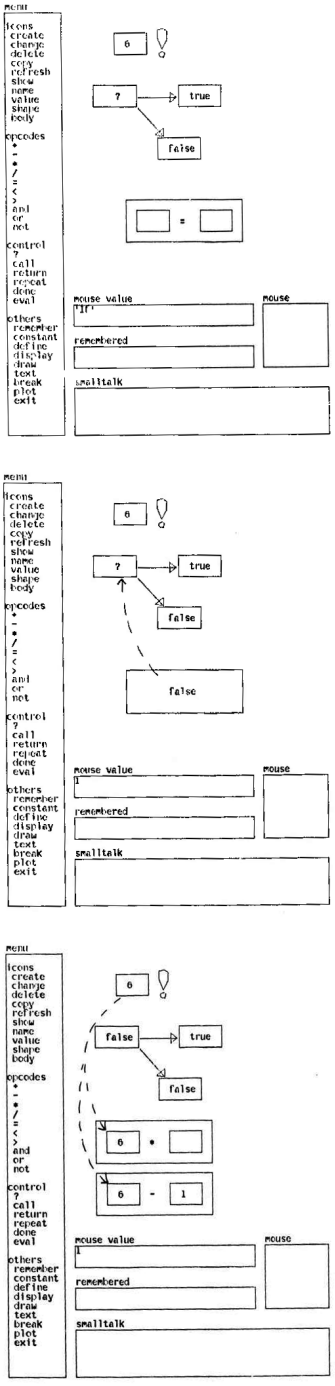

Figure 3.12: Pygmalion interface showing a sequence of operations to calculate a factorial, test it against a constant, and perform a different operation based on the result.

3.4 Inference-based Design

To this point, this chapter has focused on procedural systems that require active programming or some other form of procedural expression by the artist. An alternative approach to supporting procedural creation involves use of computational inference and artificial intelligence (AI). By enabling artists and designers to demonstrate desired outcomes and relying on the computer to infer intent, it is possible to automatically produce procedurally generated works while reducing the challenges of programming. In the following section, two examples of inference-based design systems are discussed: programming by example and design by example.

3.4.1 Programming by Example

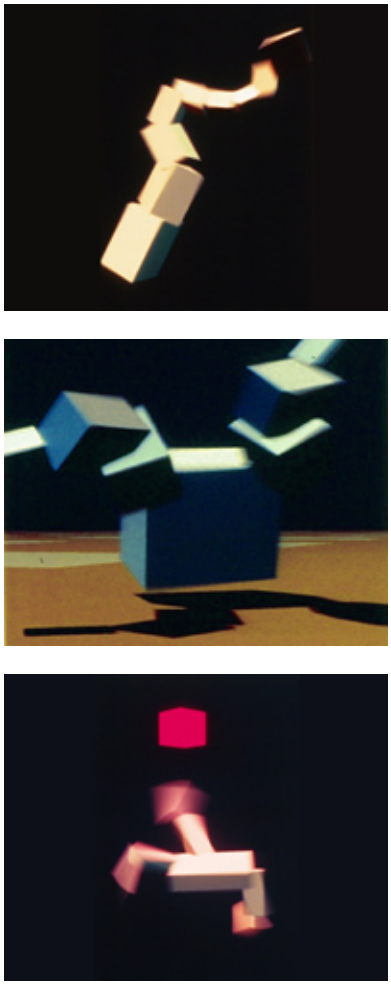

Figure 3.13: Examples of Sims' evolved virtual creatures. From top: a "swimming" creature, a "hopping" creature, and a "following" creature.

Programming-by-example systems try to guess or infer a program from examples of input and output or from sample traces of execution provided by a human programmer (Myers, 1990). This approach also is known as programming by demonstration. The reasoning behind this approach is that if a person knows how to perform a given task, it should be sufficient for them to demonstrate it and have the computer generate the corresponding program (Cypher and Halbert, 1993). Programming-by-example systems go beyond macro recordings by attempting to automatically generalize the observed procedure so that it can be applied to different contexts and applications. They generally replace constants in the recording with variables that usually accept a particular kind of
data (Faaborg and Lieberman, 2006).

Programming by example has been demonstrated in art- and design-based tools. Mondrian is a graphics creation program in which an interface agent records a designer's actions in a symbolic form and then tracks relationships between graphical objects and dependencies among the interface operations. The result is a program that can be used on "analogous" drawing tasks (Lieberman, 1993). For web-design, d.mix enables people to select elements from annotated websites which is then used by the software to generates human-editable textual code that produce those elements (Hartmann et al., 2007). Programming by example has also been applied to UI design. Landay and Myers developed a system called SILK, that enabled designers to sketch out a rough interface concept, which the system then transformed into an operational set of interface components. Programmers could then add callbacks and constraints to complete the application (1995). More recently, Dixon and Fogarty demonstrated Prefab, a system that infers UI structure from the pixels of an existing UI component and enables developers to augment existing UI elements with enhanced functionality without having to re-implement the existing interface (2010).

Dynamic direct manipulation systems can also incorporate programming by example. Pygmalion and a number of Victor's systems blend direct manipulation with recording functionality that tracks and display programmer's actions in a format similar to a program (Smith, 1975; Victor, 2013b). A consistent feature of these examples is that they all generate some form of representation or program that is designed to be readable and editable by humans. While not true in all cases, many systems using programming by example are designed to augment human programming rather than supplant it altogether.

3.4.2 Inferring designs from examples

Another approach involving machine inference avoids human programming altogether. In such systems, humans provide a set of example designs or design parameters, or iteratively select from existing options, and then rely on the system to produce novel results based on these suggestions. Examples of this approach include the evolutionary design tool Dreamcatcher (Autodesk, 2016), which enables designers to specify material constraints and general design dimensions and then select from generative designs to refine the result. Similarly, Talton et. al demonstrate the use of Bayesian algorithms on a pre-labeled set of hierarchical designs to produce novel variations (2012).

Inference-based design systems are often used to help designers automatically generate variations based on a given set of parameters. The Design Galleries project creates and organizes a set of perceptually different graphics or animations by varying a specific input parameter (Marks et al., 1997), and the PATEX system characterizes and suggests distinct 2D patterns from a sample input pattern (Guerrero et al., 2016). Computational inference can also be used to build catalogs of designs that resemble a set of example designs. Kalogerakis et al. demonstrated an approach for 3D shape synthesis that probabilistically generates large numbers of novel models from an input set of segmented models (2012a). Inference-based approaches like these are often well suited to scenarios involving labor-intensive tasks, such as those that require producing numerous variations (Talton et al., 2012), or require tuning parameters to achieve desired effects.

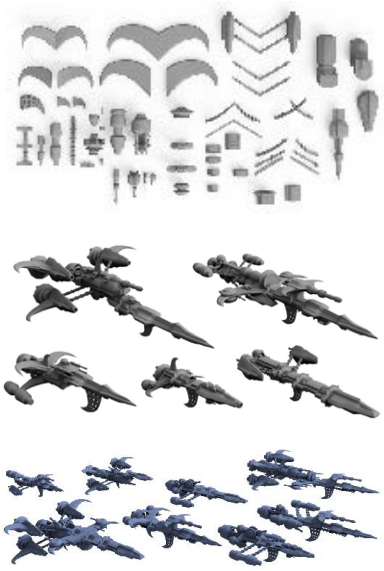

Figure 3.14: Spaceships generated by bayesian grammar induction. Top: The set of components. Middle: Example designs composed of the components. Random samples of new designs generated from the design pattern.

In other cases, inference can be used to produce surprising outcomes that human designers might not achieve without making use of this approach. Sims' Evolving Virtual Creatures system used an evolutionary algorithm to iterate on the functionality of virtual creatures. Each creature was tested by the system for their to perform a given task and only the most successfully were preserved, combined and mutated to generate successive populations. The system produced a diversity of virtual creatures, some that mimicked animal behaviors found in nature and others that exhibited unexpected traits (1994). Inference-based systems also can be used to embed expert knowledge in a design tool. DesignScape supports designers by providing layout suggestions (O'Donovan et al., 2015), Google's AutoDraw system replaces rough line drawings with pre-generated illustrations that match the subject matter of the original drawing (Motzenbecker, 2017), and Kalogerakis et al. created an algorithm for generating and applying cross-hatching textures to 3D models based on sample textures created by a human (2012b).

3.4.3 Limitations of inference-based systems

Inference-based systems offer intriguing opportunities for automating labor-intensive tasks, and suggesting different design options. Yet there are limitations to these approaches. Inference-based systems often are limited to design tasks when the designer can articulate or quantify what is desired (Marks et al., 1997). Machine-learning algorithms that enable computational inference usually require organized and labeled training data in order to function, or they require the pre-existence of a parametric space from which models are drawn (Chaudhuri and Koltun, 2010). Such datasets are often unavailable and difficult to generate in art and design tasks that focus on the styles and preferences of an individual artist. One of the biggest limitations involves interactivity. Because inference based and evolutionary design approaches are frequently subject to the general limitations of interactive machine learning, end-user involvement is often limited to providing data, answering domain-related questions, or giving feedback about a learned model. This can result in a design process with lengthy and asynchronous iterations that limits an artist's ability to affect the resulting models (Amershi et al., 2014). As a result, there are relatively few examples of inference-based systems that enable artists to move back and forth between manual creation and automated generation.

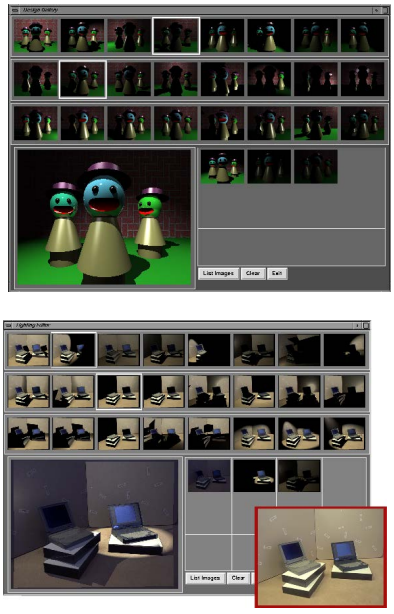

Figure 3.15: Design Galleries interface showing options for light selection and placement for two images. Designers work by select the desired option from a series of generatively produced variations.

An additional limitation of inference-based-systems is tied to the creative value of programming. One of the perceived benefits of inference-based tools is that they do not require an artist to interact with a programming representation. For people who are altogether uninterested in learning to code, these systems might be a good fit. This dissertation in part focuses on how active engagement in creating and manipulating procedural relationships can support the artistic process, and therefore demonstrates tools that require some degree of programming by the artist. Computational systems that automatically generate the procedural representation may be more accessible at the outset, however the representations produced by inferring designs from example are rarely human-readable, nor do they reflect the conscious actions and decisions of the person using the system (Amershi et al., 2014). Artists using AI-based systems may emerge with an attractive artifact, but these systems do not provide opportunities to describe, examine, and reflect on the creative process in the way that programming does. Programming-by example systems that create a human readable program are an exception, and suggest one accessible entry point for new artist programmers. However, many programming-by-demonstration systems are not designed to support people in learning how to program, therefore if the objective is to engage people in creative programming while scaffolding the learning process, it may be necessary to take an alternative approach.

3.5 Learning Through Creative Programming

This dissertation hinges on the idea that programming is a powerful creative tool both in terms of the kinds of artifacts it enables people to make and the kinds of creative thinking it encourages. Therefore in addition to systems that try to reduce or remove the need to program altogether, there is also value in tools and approaches that support the process of artists learning how to program. This section considers different approaches in helping people to learn programming for creative applications including efforts to contextualize learning programming for art and design applications, creative programming environments for young people, approachable forms of parametric design, and the role that example projects play in supporting people in programming their own applications. Collectively, these approaches suggest strategies for lowering barriers for new programmers while preserving meaningful forms of personal expression.

3.5.1 Connecting programming to relevant applications

Two tools, Processing and openFrameworks aim to be accessible to artists by maintaining a specific set of principles across their frameworks and libraries and by re-positioning programming in a context that is relevant to artists and designers (Reas and Fry, 2007; Lieberman, 2014b). The importance of meaningfully contextualizing programming for learners has been demonstrated in computer science education research. In an effort to improve completion rates for their introductory computer science course, and support students majoring in other disciplines than computer science, Mark Guzdial developed an introductory Python course structured around media and image manipulation. Guzdial also restructured course objectives to emphasize reading, understanding, and manipulating programs, as opposed to writing programs from scratch. Following these changes, students, particularly women, demonstrated an improved understanding of how programming connected to personal objectives or relevant professional skills. As a result of these changes, course retention rates greatly increased (2003). In the space of physical making, Blauvelt et al. described strategies for blending computation with craft materials (1999). In line with this idea, Buechely invented the LilyPad, a physical computing platform derived from the Arduino but modified to support creating electronic and interactive clothing and textiles (Buechley and Eisenberg, 2008). The LilyPad is a powerful example of how re-contextualizing an existing computational system can broaden and diversify engagement. A subsequent study of the LilyPad community demonstrated that unlike other physical

computing communities, the majority of people using the LilyPad were women (Buechley and Hill, 2010).

3.5.2 Learning-oriented programming environments

In addition to positioning programming in a relevant context, the design of the programming languages and platforms greatly affects learning. Complex programming libraries and APIs are of little use to people who are still learning the basics of coding. At the same time, over-simplified systems offer limited creative options. In supporting people in learning programming, it is important to balance approachability and expressiveness. Often this can be achieved by presenting a small but carefully designed subset of programming concepts in a relevant creative context. Resnick and Silverman asserted that for young people, *a little bit of programming goes a long way*, and focused on developing tools with a minimal number of carefully selected programming concepts. Logo (Papert, 1980) and Scratch (Resnick et al., 2009) are two programming languages that demonstrate this approach: they apply carefully designed simple languages with computational drawing and interactive storytelling, respectively. This enables children to explore a variety of creative outcomes while minimizing learning thresholds (Resnick and Silverman, 2005). Also in the space of programming tools for young people, Kay and Goldberg's Dynabook and the accompanying programming language Smalltalk aimed to create a unified framework between different modes of expression including drawing and painting, music creation, and simulation, to enable a diversity of end applications through a common set of techniques (Kay and Goldberg, 1977). Additionally, DiGiano and Eisenberg describe the idea of *self disclosing design tools*, software that gradually introduced designers to programming by disclosing elements of the programming language as a person uses graphic portions of the software. Self-disclosing tools can be described as forms of incidental learning or learning by observation where people notice patterns in the world from which they can incrementally make useful generalizations (1995).

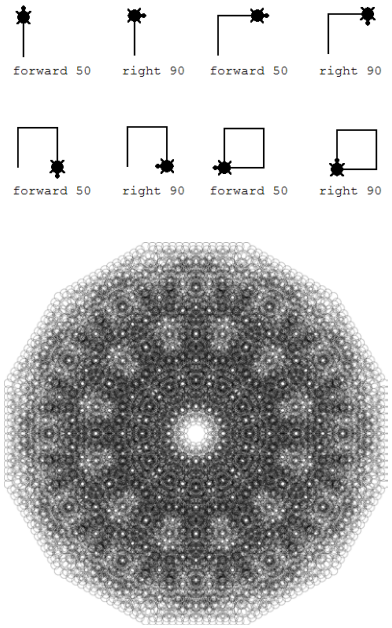

Figure 3.16: Top: Logo programming commands that control the position of the turtle, bottom: artwork created with Turtle Geometry by Ben Trube `bentrubewriter.wordpress.com/tag/turtle-geometry`

Alice, and many more recent learning-oriented programming tools were developed to teach children core computer science concepts. Conversely, systems like Dynabook and Scratch, while designed to scaffold the process of learning programming, are mainly intended to help young people apply programming to the creation of their own projects. In this respect, Scratch has been extremely successful, and the Scratch online community contains over 20 million diverse projects that span different applications, interests, and experience levels (Scratch, 2017). This is a testament to how the objec-

tives of supporting learning and supporting diverse creative outcomes can work in harmony rather than conflict with each other.

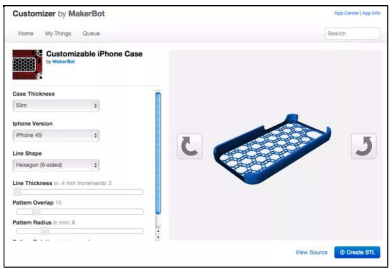

Figure 3.17: A project in the Thingeverse Customizer interface. `thingiverse.com/customizer`

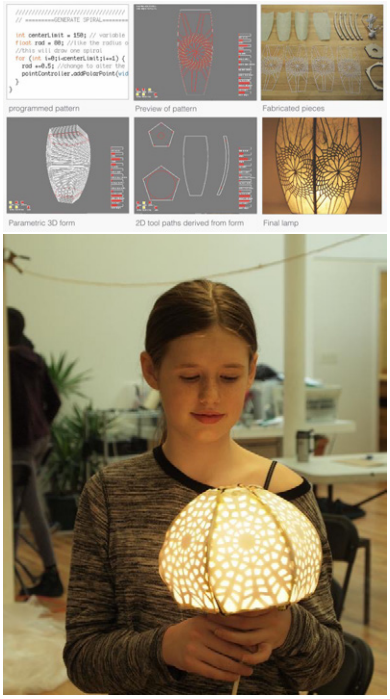

Figure 3.18: Top: Codeable Objects interface and workflow, bottom: a completed artifact.

3.5.3 Approachable Parametric Computer-aided Design

Parametric CAD is another domain of procedural art that presents challenges to people new to programming. The increasing availability of digital fabrication machines has created an entry point to CAD for people who are not professional engineers or designers. However parametric CAD tools present the same challenges as many programming languages, but with the added challenge of requiring complex 3D-geometry manipulation and the ability to design components that are viable for physical fabrication. As a result, when developing parametric-design systems for newcomers, there is often a tension between ease of use and expressiveness. Many professional GUI-based CAD tools, for example SolidWorks, Inventor, Fusion 360 and Catia, enable designers to create constraints between different components of a design and make it feasible to maintain specific properties as other aspects of a design are modified. These features offer powerful opportunities for manipulating complex assemblies and producing variations but can present challenges for learners (Maleki et al., 2014).

When adapted for people without formal knowledge in math, geometry, and logic, many entry-level parametric CAD tools often enable *customization* of existing designs rather than requiring *composition* of new designs from scratch. For example, Nervous Systems has adapted several generative jewelry algorithms to online applications that enable people to adjust dimensions and some aesthetic aspects through sliders or by clicking and dragging on portions of the design (2015). Thingeverse, a website where people to upload 3D models designed for 3D printing, enables designs created in the OpenSCAD textual CAD programming environment to expose specific parameters of a design to manipulation via sliders through the Thingeverse Customizer (2015). Customizable parametric models can increase participation in CAD and digital fabrication, and often are ideal for people who are interested in quickly personalizing a design; however, they are poorly suited for creative engagement or supporting learning. A 2015 study of the Thingeverse community by Oehlberg et al. demonstrated that customizable parametric models did not result in diversity of creative outcomes nor scaffold learning of advanced, open-ended CAD tools (2015).

Researchers have taken alternative approaches to support learning in parametric design. One approach is to link parametric design with the established skills of a given audience. For example, Torres and Paulos' Meta-

Morphe presents a framework for digital fabrication design in a format similar to web scripting, making parametric design accessible for people with web programming experience (2015). Another approach is to create simplified procedural tools that retain forms of expressiveness relevant to a specific domain. Jacobs and Buechley assisted learners in expressive forms of computational design by linking a constrained programming environment with physical crafting (2013). This approach is similar to the strategy advocated by Resnick and Silverman. While customizable parametric models do not support learning more expressive forms of design on their own, sliders and other direct-manipulation control mechanisms can serve as useful extensions for more expressive procedural tools. Although not targeted at CAD, Hartmann et al.'s Juxtapose interface assists programmers in exploring parameter variations by automatically creating control interfaces for tuning application variables at runtime (2008), and the DressCode programming environment combines a fabrication-specific set of methods with a linked direct-manipulation graphic editing environment (Jacobs et al., 2014). Both systems enable programmers to selectively expose specific parameters of their program which can then be manipulated with sliders in a graphic editor.

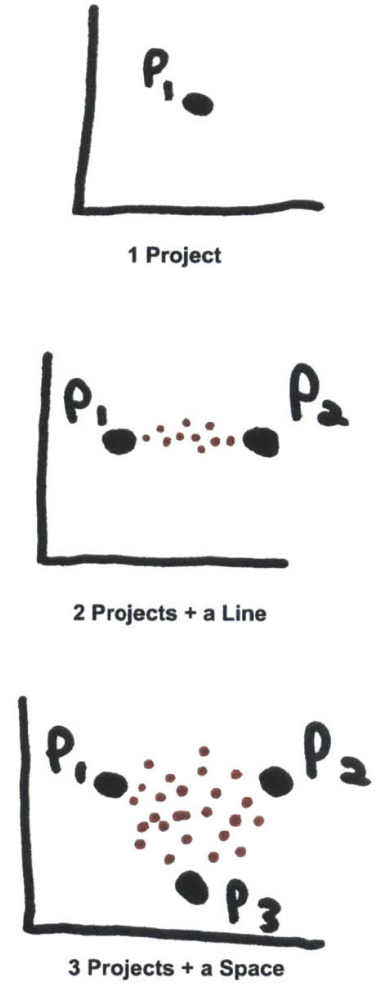

Figure 3.19: Silver's illustration of the Sample Project Space (2014).

3.5.4 Procedural creativity support through examples

Examining, modifying, and remixing procedural projects created by others provides another important form of learning. Processing, Arduino, Scratch, and openFrameworks all have large repositories of example projects, developed by their creators and by other people who use the software. Examples play an important role in supporting procedural learning for newcomers and experts alike. In a study of inexperienced programmers learning recursion, Pirolli and Anderson showed how new programmers relied heavily on examples to guide their solutions to challenging programming tasks. Furthermore students demonstrated the ability to generalize from knowledge compiled from examples to solve entirely new problems (1985). Professional programmers often solve problems by sourcing multiple examples online and synthesizing them to suit a new application (Brandt et al., 2010).

The quality and structure of examples play a significant role in their effectiveness. Renk et al. described how learning by example can be a more effective than learning through problem solving due to the reduced cognitive load of referencing existing information, but emphasized that the effectiveness examples are determined by the type of learning activities employed by the individual learners and the characteristics of the presented exam-

ples (Renkl et al., 1998). Overly complex or inconsistent examples can confuse learners, or undermine specific programming concepts (Malan and Halland, 2004).

The nature of examples is particularly important when the objective is to support creativity. In addition, it is possible to examine the expressiveness of a system by looking at what kinds of examples are readily created by the system. Silver described the concept of “The Sample Project Space” as a way of evaluating a creative tool. One sample project creates a point. A second sample project that is distinctly different from the first creates a line of new creative possibilities, interpolated between the two points. Three sufficiently different and evocative examples create a space of possibilities with a huge number of potential projects interpolated between the points (2014). Simultaneously developing a system and a corresponding set of examples can inform the process of iterating on the system features, build an important resource for learners, and help communicate the creative potential of a system.

4

Para: Procedural Art Through Direct Manipulation

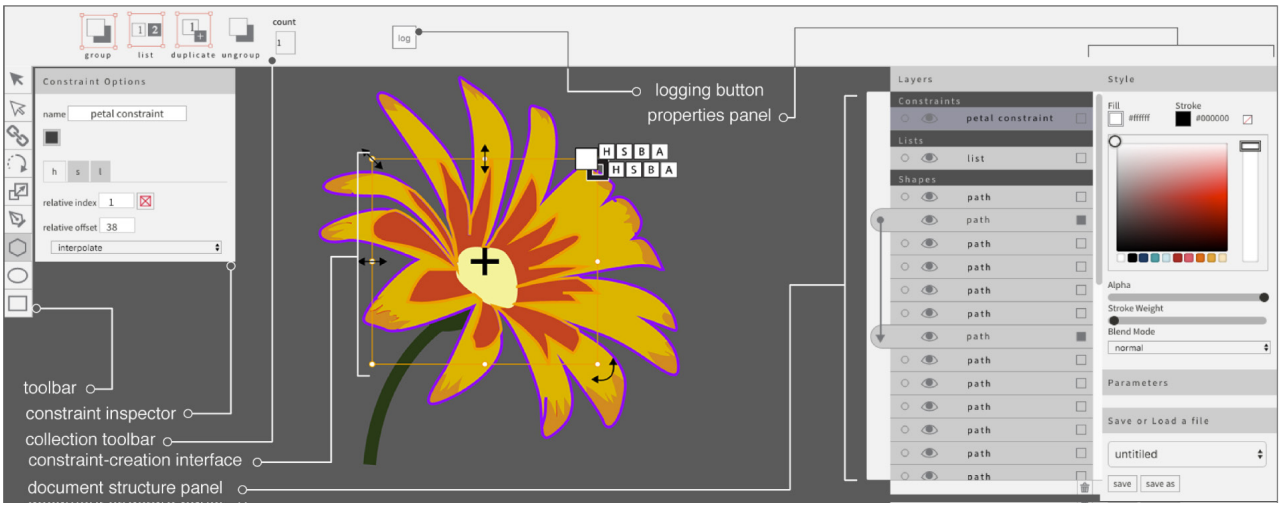

Figure 4.1: Para's interface.

As the previous chapters illustrate, procedural tools offer many creative opportunities. Simultaneously, they pose significant learning challenges, and manual artists interested in using procedural techniques must often adopt tools and practices far removed from those to which they are accustomed. This chapter presents Para, a software tool developed to integrate manual and procedural art creation in a form that is both approachable for new programmers and expressive for professional artists. Para is a dynamic direct-manipulation tool that supports procedural art through live, non-linear, and continuous manipulation. The development of Para was motivated by a combination of factors: the relevance of computation to visual artists, the challenge of learning programming, and the significant interaction differences between representational programming tools and software for manual art and design. Para's development was also strongly connected to my personal experience transitioning from manual art to

Figure 4.2: Para enables artists to quickly set up constraints using direct manipulation interactions with their art. Here, an artist has set up a series of color constraints that makes exploring color variations highly efficient.

procedural creation. Overall, Para's design was based on the hypothesis that enabling people to describe procedural relationships through direct manipulation would enable an approachable pathway for procedural art creation, and extend manual art practices rather than conflict with them.

As outlined in the prior chapter, much of the prior work in developing accessible procedural tools for art has focused on creating simplified textual programming languages and environments, augmenting GUI-based tools with visual programming languages, or inference-based procedural tools that remove the need for programming altogether. The goal of Para was to augment the conventions of graphic art software to support the creation and modification of procedural relationships exclusively through manipulation of concrete artwork. In doing so, I built on prior work in dynamic direct manipulation for data visualization, interactivity, and CAD, but with the new objective of supporting and extending the practices of professional manual fine artists. This goal was challenging because it required developing a procedural representation that supported expressive creation yet was compatible with direct-manipulation conventions.

Para is also distinguished from prior work by how it was evaluated. The development of Para was driven by the desire to better understand the *expressive* potential of dynamic direct manipulation in professional art practice; therefore, the system is evaluated through extended open-ended studies and polished, original artwork is presented in the results. This required developing a tool that was suitable for professional art practice. It also led to the development of an evaluation methodology that supported in-depth use of the tool and in-depth conversations with professional artists.

The following sections focus on the procedural model and features that make up Para. The system is structured around core declarative procedural operations that can be expressed through direct manipulation, are easy to use, and enable diverse creative outcomes. These operations include declarative geometric and stylistic constraints, visually represented ordered lists that can be used to specify how constraints map to collections of objects, and declarative duplication to enable dynamic copying. Constraints, lists, and duplication can be combined in different ways to produce different outcomes. The results of two open-ended studies demonstrate how the direct manipulation of procedural relationships can extend manual practice. These studies evaluate the accessibility and expressiveness of Para in the hands of professional artists by examining how they worked and the artifacts they produced. To my knowledge, these were the first open-ended studies in the domain of dynamic direct manipulation for art.

4.1 Software Design and Implementation

Para is a software tool aimed at accessible, expressive integration of procedural techniques in visual art through a direct-manipulation interface.[1] Para was designed using an iterative process that involved analyzing professional procedural artwork, exploring different prototype interactions and procedural representations, and consulting professional procedural artists to evaluate the initial direction of the software.

[1] Para is available for use at `http://paradrawing.com/` and the source is at `http://github.com/mitmedialab/para`.

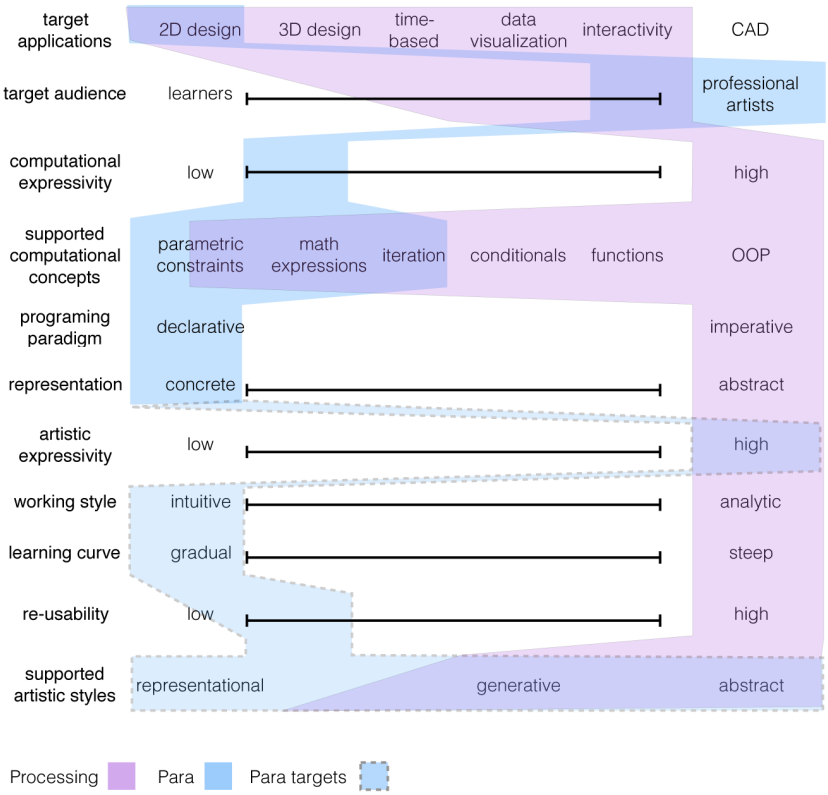

Figure 4.3: Para design features and targets, compared to those of the Processing programming environment.

4.1.1 Artwork analysis

Para's design was informed by compiling procedural artworks by prominent procedural artists including Joshua Davis, Casey Reas, Marius Watz, Emiliy Gobielle, Theo Watson, Christopher Coleman, and Erik Naztke. Analysis of this artwork revealed procedural aesthetics and methods consistent across the artworks (table 4.1, common). It also highlighted qualities found in only a few artworks that blended manual and procedural

creation (table 4.1, rare). This led to a set of design guidelines, concrete approaches and outcomes that the system should support. Following this initial analysis, preliminary versions of Para were prototyped by developing interactions that supported similar approaches and aesthetics but were compatible with direct manipulation. The primary conventions the system sought to preserve were image-based communication of structural and organizational relationships, support for live and non-linear edits, removal of textual commands and expressions, and integration of interface components from direct-manipulation art tools.

Common	Rare
Complex forms and patterns through procedural duplication and object-oriented programming.	Close integration of procedural and manual illustration.
Iterative variation across scale, form and color.	Procedural transformation of hand-drawn artwork.
Regular geometric/ symmetrical distribution of form.	
Generative compositions through random distributions (normal, uniform, etc.)	.

Table 4.1: Techniques identified in professional procedural artwork.

4.1.2 Prototype procedural models

The initial development of Para involved experiments in adapting different programming paradigms to support direct manipulation. A previous project, DressCode, contained a linked representation that mapped direct-manipulation edits in a graphic drawing environment to textual imperative programming statements (Jacobs et al., 2014). This process worked well for initialization actions but performed poorly for editing actions because subsequent edits to the same object posed the dilemma of either modifying one iterative statement or adding multiple iterative statements in sequence. To avoid this issue with Para, the system's functionality was modeled around declarative programming. This shifted the focus from allowing people to describe sequential design actions to enabling then to describe relationships between elements of a composition. Two different forms of declarative relationships were prototyped in early versions of Para. The first was modeled after prototype inheritance, a form of object-oriented programming where objects are instantiated from other objects rather than from abstract classes and references to properties not explicitly declared on the target object are automatically referenced from the parent object (Taivalsaari, 1996). For Para, this concept was applied to vector shapes cloned from existing shapes. Until a stylistic property was explicitly set on the cloned shape, it would correspond with the value of its parent. This enabled changes to the parent shape to propagate automatically throughout a composition. The second procedural approach used declarative constraints. Unlike the inheritance model, constraints had to be explicitly created by the artist by specifying a given relationship between two object properties, but they also supported a greater degree of

computational expressiveness. Rather than being limited to objects with parent-child relationships, these constraints enabled the creation of arbitrary procedural relationships between any geometric or stylistic property of any two or more objects.

Constraints offered a way to create a variety of procedural relationships while reducing the number of procedural concepts that an artist would have to learn. For this reason, future iterations of Para focused on the constraint model, and discarded the inheritance model. The prototype constraint interactions were iteratively revised into a unified system that could produce sample artwork. I informed this revision through regular meetings with the artist Erik Natzke and informal testing with artists and designers. The following section describes the interface and procedural model of Para in detail and illustrates how its features enable different forms of expression.

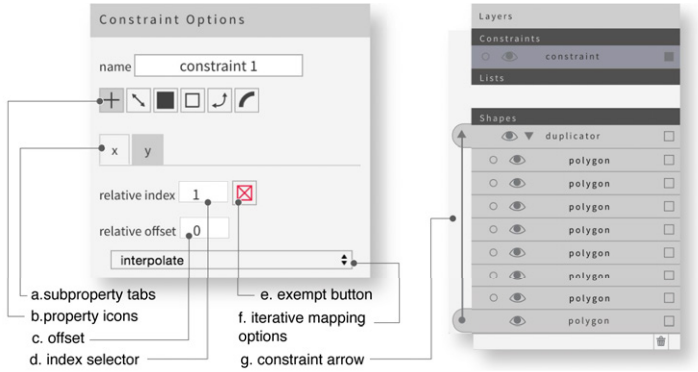

Figure 4.4: Detail of constraint inspector and document structure panel which lists existing constraints. Selecting a constraint indicates directionality through an arrow between reference and relative.

4.2 Interface

Para was developed as vector drawing software because it enabled the interface and interactions of the software to resemble existing software used by manual artists. The interface features analogs of elements in digital illustration software, including tools for manual shape creation, selection, and transformation (fig 4.1 a); a panel for adjusting stylistic and geometric properties of vector artwork (fig 4.1 g); and a document structure panel that displays the current components in the drawing (fig 4.1 e). Para is implemented in JavaScript using Backbone.js and the Paper.js vector graphics scripting framework. Although Paper.js contains methods for managing hierarchies of vector graphics, Para does not use this functionality. The procedural nature of Para required developing a new representation for document structure, which is presented below.

Iterative Behavior	Description
Interpolation	Evenly distributed values sampled from a Lagrange polynomial derived from the reference.
Random	A uniform distribution of random values between the min and max values of the reference. Random seed is re-calculated when min or max is manually changed.
Gaussian	A normal distribution of values with the mean derived from the first reference object and a standard deviation derived from the distance between the first and last object.
Radial	Polar values based on a diameter determined by the min and max values of the reference. list.
Alternate	A series which cycles through the values of the objects in the reference (e.g for reference values 360,45,250 the result will be 360,45,250,360,45…)

Table 4.2: Iterative constraint behavior options.

4.2.1 Constraints

In programming languages, complex relationships often are comprised of simpler, low-level operations and relationships. Para builds on this approach through object-to-object constraints, which serve as the fundamental building block for advanced procedural functionality. Constraints allow artists to create relationships between geometric or stylistic properties of a minimum of two vector objects: a *reference* (the object whose properties are referenced in the constraint) and a *relative* (the object being constrained). Constraints satisfy three important conventions of direct manipulation. First, they support non-linear edits without creating errors or inconsistencies. Second continuous direct-manipulation edits to a constraint reference results in live updates to corresponding relatives. Third, constraints provide a visual means of describing mathematical relationships between graphical forms. When created, constraints preserve the current state of the drawing. In their simplest form, one-to-one constraints are represented as expressions of the form $f(x) = x + o$, with x the value of the reference property and o the existing offset between the reference property and relative property prior to constraint. This format preserves any difference in values between the reference and relative properties when the constraint is created, enabling artists to describe constraint relationships by drawing them rather than by writing them as a mathematical expression. The constraint tool enables the creation of *one-to-one constraints* by selecting the relative object and reference in succession. When the reference is selected, a set of icons appears over the reference object (fig 4.1 d), which enables the artist to specify the property to constrain: position, scale, rotation, fill and stroke color, stroke weight, or a sub-property of these. Para extends Constraint.js (Oney et al., 2012) to handle constraint propagation and added functionality to detect and prevent constraint cycles.

Para's constraints are different from geometric constraints in CAD tools because they correspond to aesthetic applications including color variation and ornamentation. Figure 4.2 demonstrates one application of con-

straints to produce variations in the colors of a flower illustration. Here, the fill color of the inner red portion of the flower is constrained to the fill color of the yellow center and the outer orange portion is constrained to the red portion. As the hue, lightness, or saturation of the center is changed, the colors of the outer petals shift while maintaining their original offset. This enables global relative color changes across a drawing from a single edit (fig 4.1 h). This also demonstrates how edits propagate across chained constraints to secondary relatives. By varying which sub-properties are constrained, artists can fine-tune how different parts of the drawing respond to an edit.

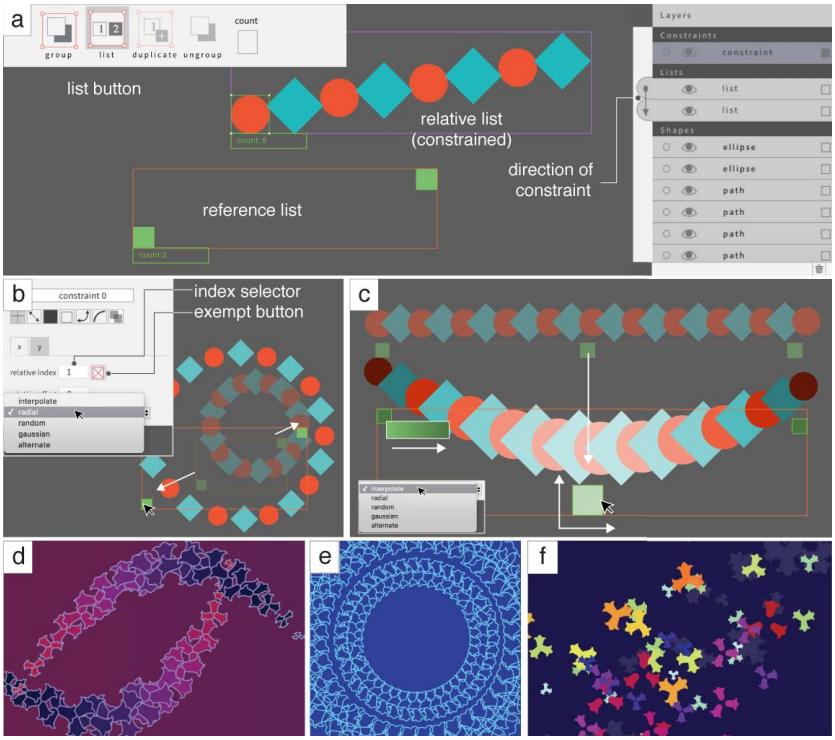

Figure 4.5: Variations on many-to-many constraints. (a) Basic list constraint between 8 relatives and 2 references. (b) Radial pattern produced through radial distribution with two reference shapes. (c) Arc patterns and color gradient produced with polynomial interpolation. (d-f) Variations of polynomial, radial, and random distributions.

4.2.2 Lists

Discussions with procedural artists highlighted the importance of supporting efficient manipulation of large numbers of visual elements. Textual tools enable management of multiple objects through abstract datatypes to store collections of data and control structures like loops to efficiently generate iterative variations across collections. In Para, multiple objects are managed through lists: visually represented, ordered collections of artwork that can be procedurally manipulated. Lists are higher-level structures and

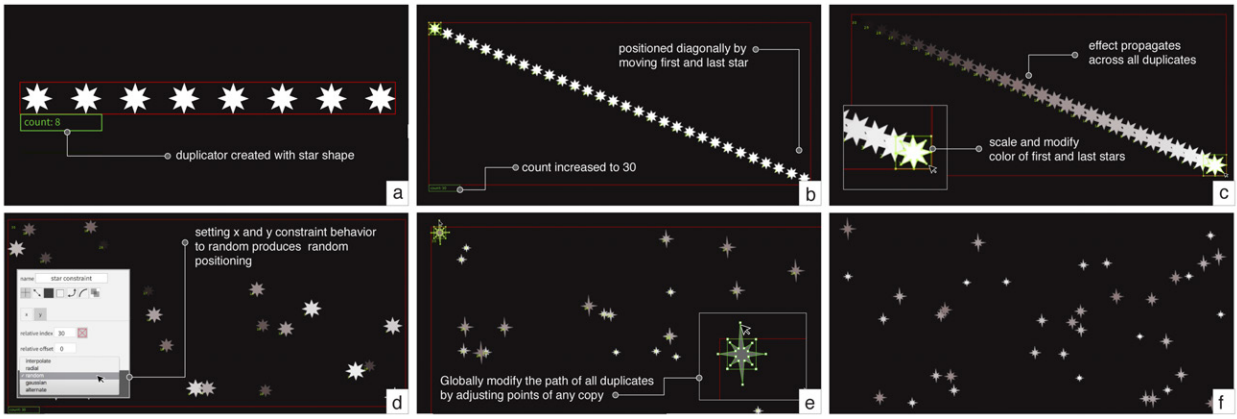

Figure 4.6: Random starfield creation: (a) duplicating a star, (b) creating a row of stars, (c) creating variation in scale and color, (d) producing a random distribution, (e) manually manipulating points of star, and (f) finished starfield.

can be used to map transformations to multiple objects. Lists are created from selected objects using the list button and appear in the list section of the document structure panel (fig 4.5 a). Manual transformations on a list are mapped to each member. For example, rotating a list will rotate each member around its own center, rather than the centroid as happens in geometric groups. This behavior allows *one-to-many constraints* where multiple objects are subject to one constraint. In the flower example, I can constrain the 14 small orange outer portions of each petal (fig 4.1) using a list and a single constraint.

Para's lists also facilitate iterative variation similar to that found in textually generated procedural art. This is achieved through *many-to-many constraints*: constraints in which both the reference and relative are lists. Many-to-many constraints create iterative mappings where each object in the relative list is constrained differently based on its index. This behavior is possible because unlike other dynamic direct-manipulation tools (Hoarau and Conversy, 2012; Kazi et al., 2014), Para's lists are ordered. The values to constrain the relative are calculated by interpolating across the values of the objects in the reference list. Figure 4.5 demonstrates how a many-to-many constraint can be used to dynamically modify the distribution of a pattern. In this example, the shapes of a repeating ellipse and diamond pattern are constrained on the x and y axes by a reference list containing two green rectangles (fig 4.5 a). The artist can update the distribution of the diamond ellipse pattern by modifying the position of the rectangles. In the process, the geometric offsets are maintained between each member of the reference and relative.

Different types of distributions can be created by altering the content of the reference list. If the reference list contains more than two members, a non-linear reference value set is produced (fig 4.5 c). These values are sampled from a Lagrange polynomial that is produced from the values of the objects

of the reference list. Moving the second rectangle in the reference list up or down changes the polynomial and results in a curved distribution of the relative pattern. This interpolation technique can produce many different types of non-linear distributions across any property, including parabolas, waves, and ellipses, by adding more shapes to the reference list. Because interpolation does not support all distributions, Para allows artists to select different mapping behaviors for constraints from a drop-down menu in the constraint inspector (fig 4.5 c). Table 4.2 lists the current options and their calculation mechanisms and figure 4.5, b, e, and f demonstrate distributions derived from radial and random mappings. The constraint inspector also enables artists to modify the individual offsets for each member of the reference list using the relative index selector, or exclude a relative member from being affected using the exception button (fig 4.5 b).

4.2.3 Duplicators

Prior analysis of procedural art demonstrated the importance of automatically creating new forms for generative compositions. Constraints and lists facilitate procedural relationships, but cannot generate new objects by themselves. To address this, Para contains Duplicators. Duplicators are specialized lists with the ability to declaratively control the number of objects they contain. Duplicators are initialized on a single object. Doing so generates or deletes new shapes when the count is increased or decreased. When first created, duplicators contain self-referencing constraints on all geometric and stylistic properties. Practically, this results in iterative variations across all members of a duplicator on any property by directly manipulating either the first or last objects. Updates to the number of copies of a duplicator will preserve these variations. Like other constraints, duplicator constraints can be modified using the constraint interface to change offsets, specify exceptions, and change constraint behavior for different properties and sub-properties. Duplicators also can act as references or relatives in manually created constraints. To preserve correspondence across the geometry of their members, duplicators enact a constraint-like behavior on the paths of all of the duplicates wherein changes to the points of any individual shape are propagated across the other copies.

Duplicators build on the low-level functionality of lists and constraints to enable creation and management of distributions with many shapes, for example, the creation of a random star field in Figure 4.6. A single five-point star is drawn and duplicated (fig 4.6 a). The start and end copies are manually altered by position, scale, and fill color to produce a diagonal line of stars growing in size and brightness (fig 4.6 b). The x-position constraint

mapping is changed to random, producing a vertical gradient from light to dark (fig 4.6 c). Changing the mapping of the y-position constraint to random results in a field of stars with a random distribution of brightness and scale (fig 4.6 d). The random distribution can be recalculated by manually dragging the first or last star, and the number of stars can be increased by modifying the count. The shape of each star can be altered by modifying the points of any individual star (fig 4.6 f).

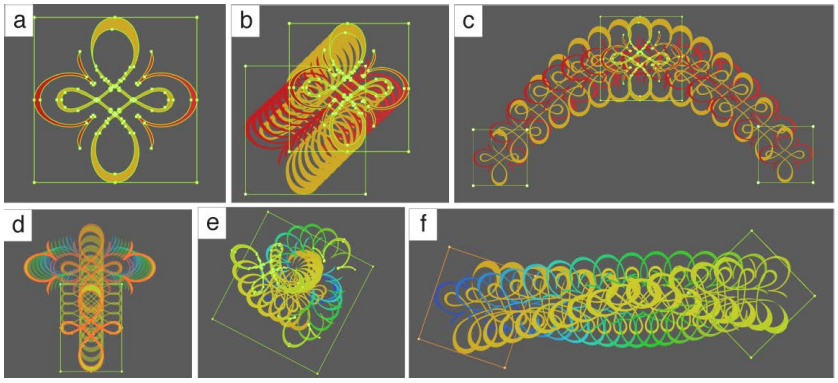

Figure 4.7: Group duplicators. (a) Group with three members. (b) Duplicator initialized on group. (c) Arc pattern produced with three references (selected). (d) modifying position/color of individual group members (orange). (e-f) variations from modifying group members.

Geometric groups also can be converted into duplicators, thereby enabling more complex distributions (fig 4.7). Groups are a structural way to make compound objects and do not serve a procedural purpose on their own. For group duplicators, constraints are created between each respective member of a group across each copy to ensure that correspondence between each copy is maintained if the structure of a single group is altered (fig 4.7 d-f). In Para's internal representation, duplicator constraints are identical to many-to-many constraints. Duplicator constraints exist between a reference list consisting of the first and last member of the duplicator, and a relative list consisting of the duplicator's children. Self-referencing is possible because similar to lists in conventional programming languages, objects in Para can be in multiple lists simultaneously.

4.3 Evaluations

The evaluation of Para had two objectives: to evaluate the accessibility and expressiveness of Para in an open-ended setting and to understand how procedural tools could support manual artists. Two studies were conducted, targeting both objectives. The first evaluation was a breadth-based workshop with 11 artists, aimed at revealing the creative trade-offs between direct and textual tools. The second evaluation was an in-depth

Id	Background	Manual Art	Digital Art	Coding
p1	graphic design	4	5	1
p2	graphic design	4	4	1
p3	design	3	3	2
p4	printmaking	5	5	1
p5	art and CS	3	3	3
p6	illustration	4	4	1
p7	installation design	5	5	5
p8	architecture	5	5	1
p9	illustration	4	4	1
p10	art and CS education	4	4	5
p11	Product Design	4	5	2
Kim	painting graphic design	5	5	2

Table 4.3: Participant backgrounds and experience in manual art, digital art and coding (1 being no experience and 5 being expert.)

study with a single artist, aimed at evaluating Para's performance in realistic creative practice. Each study was structured around the following evaluation criteria, which was derived from the opportunities and challenges highlighted in the background research and analysis of professional procedural artwork:

Ease of use: Is it easy for novice programmers to use the tool? Can people understand the tool's artistic applications?

Creative outcomes: Does the tool support the creation of different procedural aesthetics? Does the tool enable creating variations of a piece?

Process: How does working with the tool compare to traditional art tools? Can artists integrate their prior expertise? Can people move between manual and procedural creation?

Reflection: Does the tool encourage thinking about procedural relationships? Does the tool affect how people think about making art?

In each study data was collected through recorded discussions, surveys, and participant artwork. Surveys contained attitudinal questions relating to my evaluation criteria, using 5-point Likert scales, with 5 as the optimal response. Survey results, discussion transcripts, and artwork were analyzed with respect to the evaluation criteria. The majority of results are qualitative and triangulated from open-ended survey responses and group or individual discussions. When scale data are used, the median and standard deviation are presented.

4.3.1 Workshop study methodology

The workshop was conducted with 11 participants from a range of art and design backgrounds. The majority were inexperienced programmers (table 4.3). The workshop lasted 14 hours over two days and covered Para and

the textual procedural art tool Processing. Para was compared to Processing because Processing shares many of the same accessibility objectives as Para. The goal was to compare Para's accessibility and expressiveness to textual procedural tools; therefore the study did not compare Para to non-procedural design tools like Adobe Illustrator. Illustrator emphasizes manual drawing tools, which are not the focus of Para's innovation. Participants were first introduced to Para and given incremental demonstrations in the use of duplicators and groups, followed by a period for free experimentation. Participants then were introduced to Processing and provided instruction in syntax, drawing and transforming shapes, loops, mathematical expressions and conditionals and then given opportunities to freely experiment. The instruction in Processing was based on approaches for beginners from the Processing Handbook (Reas and Fry, 2007), and demonstrated methods for producing designs that were similar to those possible using Para. Following the instruction period, participants had three hours to work on a piece of their choice with either or both tools. Participants were also to use other tools in conjunction with Processing and Para (e.g. Photoshop and Illustrator) for bitmap manipulation and detailed manual drawing, tasks not intended to be supported by Para or Processing.

4.3.2 Limitations

The studies required professionals to rely on prototype software with less sophisticated drawing tools than commercial software, which people found limiting. This is common in systems-building research. Although Processing was the most relevant textual procedural art tool to compare with Para, it also includes features for animation and interactivity. These features do not exist in Para and are, therefore, not directly comparable. This factor was compensated for in the study by focusing on Processing's capabilities for static procedural art. Para's cycle-prevention feature was not implemented prior to the workshops; however, I notified participants of the issue, and with few exceptions, people did not create cycles. The in-depth study surveys the experiences of one person; however, evaluations with additional artists from different backgrounds would likely reveal additional insights. The variety of aesthetics and styles produced by one person suggests the potential of this approach to support a variety of artists. Finally, it is challenging to measure expressiveness because different artists are expressive in different ways. Therefore, the results present the artwork created with Para in order to provide concrete examples of its expressiveness.

4.3.3 Workshop Results

Overall results indicate that participants found the direct-manipulation interface of Para more intuitive to use than textual programming in Processing. People who specialized in graphic design and illustration prior to the workshop felt that Para fit well with their current practices and they were interested in using it in the future. People with backgrounds outside of graphic art and illustration were less interested. Participants felt that textual code could provide greater control for experienced programmers. In the following section, I detail these results in the context of the evaluation criteria.

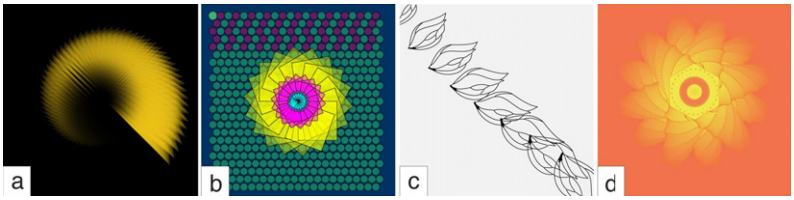

Figure 4.8: Artwork from workshop. Left: conditional-design from Processing. Right: Designs generated from multiple hexagonal duplicators from Para combined with a center motif from Processing.

Ease of use: All participants were able to follow the instructions for using Para but struggled to understand instructions for Processing. These observations were substantiated by survey responses: seven participants stated that they felt Para was easy to learn, and three felt Processing was easy to learn. Many participants said the difficulty of Processing increased their desire to practice it when they had the benefit of instructor support. Conversely, participants said Para was familiar and felt confident they could learn it independently.

Process: All participants experimented with both Processing and Para throughout the workshop. Participants exhibited a mix of responses with regard to how Processing fit with their prior art practice (mean:3.6, std:0.92); however, nearly all participants expressed interest in using the tool in the future (mean: 4.3, std:0.47). In survey and verbal responses, participants indicated they struggled with understanding the logic of Processing programs. Several people described their process as one of trial and error. In addition, people were frustrated by not being able to adjust artwork manually. Despite the challenges in learning Processing, several people talked about how they could exercise greater control and perform tasks not possible in Para. Four participants stated that Para fit well with their existing practice, and six (including the first four) stated that they could see themselves using Para in future work. People who had expertise in manual art and minimal programming experience were the most interested in using Para in the future. For people with interest in using procedural tools for 3D design and interactivity, Para was less relevant by design.

Primary frustrations with Para centered on reduced sophistication of the direct-manipulation tools in comparison with commercial equivalents.

Creative outcomes: Because of time constraints, participants were not able to create polished artwork with either Para or Processing, although many made a series of sketches that suggested different directions for more sophisticated work (fig 5). Participants who relied exclusively on Processing primarily created variations of a radial distribution example presented in the instruction, incorporating randomness on opacity, scale, or rotation of shapes (fig 5 a,b). Participants who used Para exhibited a range of approaches: one person used portions of an illustration created prior to the workshop in a duplicated pattern (fig 5 c), while another experimented with the duplication functionality to simulate 3D rotated columns. Another participant generated a series of incrementally rotated geometric shapes. Using Illustrator, he combined these forms with a radial pattern from Processing (fig 5 d). Eight people stated that Processing enabled them to create things they would have difficulty creating otherwise (mean:4, std:1). Ten people stated that Para enabled them to create things they would have difficulty making otherwise (mean:4.1, std:0.83).

Reflection: In discussions and surveys, most participants felt that textual programming could be a powerful tool for art; however, they had a mix of reactions to using Processing. For several participants, using Processing underscored their prior associations of textual programming as inaccessible. The majority of participants felt that they would have to practice extensively with Processing to create sophisticated projects. In discussion and open-ended responses, participants expressed greater levels of confidence using Para because it had features similar to digital illustration programs. Others described Para's interface as more intuitive than Processing. Several people wrote that they found the live feedback in Para helpful; however, these responses were solely from people with prior experience with textual programming tools. None of the people new to programming commented on the live aspects, suggesting that people who only use direct manipulation may take liveness as a given in digital tools. Several participants talked about using Para and Processing for different purposes. One participant said Para was ideal for creating complex static illustrations, but Processing offered the opportunity for interactive pieces. Another participant said that Para and Processing were appropriate for different points in her artistic process, and said she would use Processing if she had a clear goal in mind.

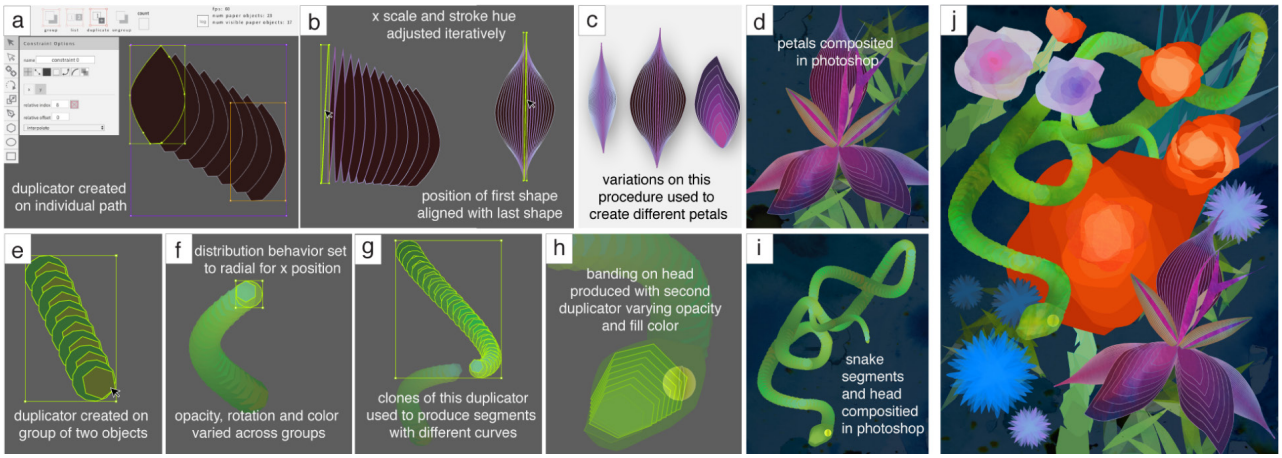

Figure 4.9: a-j Illustrative work with para: (a-d) process for creating orchid, (e-i) process for creating snake, (j) finished piece, composed in Photoshop and textured using image overlay. k-l examples of Kim's illustrations prior to study.

4.3.4 In-depth study methodology

Figure 4.10: Examples of Kim's illustrations prior to study.

The group workshop underscored the learnability of Para but did not demonstrate how Para performs in actual art practice. I performed a second study to understand what kinds of artwork a person could create with Para through extended use. I commissioned the professional artist Kim Smith to use Para for two weeks to create several pieces of art. Kim has extensive experience in abstract painting (fig 4.11 a), realistic illustration (fig 4.10), and graphic design. Kim is an expert with both manual and digital tools (table 4.3). She had an interest in applying procedural techniques in her work but was reluctant to invest the time needed to learn programming.

I met with Kim in person six times over the course of the study, every 2-3 days, for 1-2 hours apiece. First, I introduced her to Para in a 1-hour training session. In later meetings, I reviewed the artwork she produced and discussed her experience. I gave her the choice of what to create during the study, but suggested experimenting with abstract and illustrated styles. Similar to the workshop, I allowed her to incorporate other digital tools into her work with Para. In the results, I distinguish between portions of artwork produced with Para and those produced by other means. In addition to written surveys and interviews, Kim wrote short reflections after every working session. I used automated surveys within Para to assess her experience while using the system. Automated dialogs appeared every 20 minutes asking questions about her current task, goals and difficulties, and a snapshot of her work was saved. Throughout the study, I used Kim's feedback to iterate on Para's functionality.

4.3.5 In-depth study results

Kim found Para to be intuitive and felt she could use it more effectively and with greater confidence than textual programming and parametric CAD tools. She produced seven finished pieces and applied Para to abstract and illustrative styles. In her artwork, she used duplicators and iteratively constrained lists, but did not find one-to-one constraints as useful. Kim felt Para was compatible with her painting practice and also offered new opportunities for creative exploration. Using Para altered the way she thought about her artistic process, and *she requested to continue using the software afterwards.*

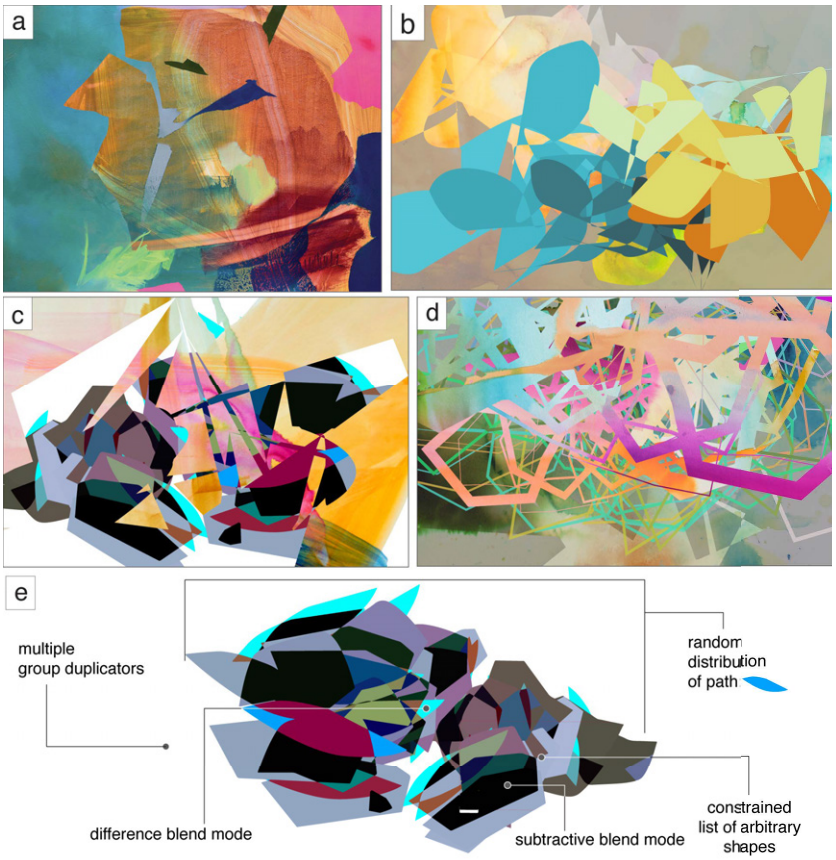

Figure 4.11: Comparison between Kim's prior art and abstract work created with Para. (a) sample of prior art, (b-d) finished pieces containing procedurally created forms overlaid with hand-painted imagery in Photoshop, (e) composition created in Para to produce c.

Ease of use: Kim's feedback from the experience sampling indicated that she struggled initially with Para's vector selection and editing tools, which were less refined than those in commercial software. Otherwise, she described the Para interface as accessible and enjoyable. Specifically, she said the emphasis on visual manipulation was "welcoming and friendly," and contrasted this with using complex CAD software and textual program-

ming tools. She also stated that Para made her "comfortable starting with a blank canvas" because the visual interface gave her confidence that she could produce something and develop a process. When asked about the minimal procedural feature set, she said,"I think there's a lot of options in a very small parameter set."

Kim found some procedural aspects of Para to be difficult. Initially she struggled with understanding the order in which shapes should be selected for one-to-one constraints. I modified the constraint tool to automatically pull up the property selection interface for selected objects when the tool was enabled. Kim said these revisions made the tool more intuitive. Although she experimented with one-to-one constraints and felt they might be useful for other artists, Kim never found a way to use them in her work. Kim also encountered organizational challenges when creating extremely complex work. At times it was difficult for her to identify and select specific procedural relationships. I partially addressed this by adding the ability to label constraints, lists, and geometry; however, organization remained an issue. In discussion, Kim pointed out how organization was also a challenge in non-procedural digital art tools, but felt that organizational mechanisms were particularly important for tools like Para because they made it easier to quickly build complexity.

Creative Outcomes: In week 1, Kim created four abstract pieces using Para (see examples in (fig 4.11 b,d). She produced them by drawing vector shapes in Para, and used duplicators to create multiple copies with iterative changes in color, scale, and position. In week 2, Kim made two more abstract pieces. These incorporated procedural variation of opacity and blend modes, and constrained lists fig 4.11 c). She also made an illustration (fig 4.9 j), that applied duplicators, lists, and iterative constraints to the creation of flowers (fig 4.9 a-d) and a snake (fig 4.9 e-i). In abstract and illustrative pieces, Kim used Photoshop to add additional color and texture by using painted imagery as a transparent overlay and background.

Process: For her abstract work, Kim described her use of Para as "exploratory" and "experimental." She felt the unexpected effects and forms possible through Para's generative features (random and Gausssian mappings) were an advantage because they generated starting points for new work:

There is an element of applying a procedure and not knowing exactly what will happen . . . and this is a really great quality about Para. This software allows for an experimental process . . . In some ways, it mirrors the experience of painting for me, in that frequently I am open to chance and random occurrences within my materials and techniques.

Kim associated the exploration possible with Para with traditional art media, stating:

In a lot of ways [Para] feels similar to traditional tools. The freedom to implement quickly, understanding what's happening, but not needing to have the foresight ahead of time [allows for] a more sculptural way of creating.

She later described the importance of Para's emphasis on visual communication:

I appreciate the lack of metrics and numbers . . . I understand that they're there, but I'm more interested in what something looks like . . . I have this freedom where I'm not doing math. It's doing the math for me. I'm just interested in what I'm trying to make.

Because she was interested in variations in color and layering, Kim requested support for opacity and blend modes[2], which were features she relied on in Photoshop. I added UI components for opacity and blend mode that matched those in Photoshop and made it possible to iteratively constrain these properties. Kim incorporated these features into her abstract work (fig 4.11 c,e) and described how procedural control of opacity and blend enabled her to use subtractive masks and filters in a dynamic fashion. The ability to quickly and effectively incorporate features from Photoshop into the procedural structure of Para demonstrated how this approach could extend existing digital art tools.

[2] Blend mode determines how an object is composited with those beneath it.

Kim initially focused on producing abstract work because this was representative of her fine-art practice. She felt Para was useful for illustrative work, but felt the goal-oriented nature of illustration required using the system with greater control. The lack of refined drawing tools made the illustrative work laborious at times. While Kim found constraints and duplicators useful in both abstract and illustrative work, she used other procedural aspects less. Lists were confusing at first, and it was labor-intensive to manually constrain them. I simplified lists to function similar to duplicators with automatically created self-referencing constraints. Kim said this modification corresponded with her objectives, and applied the new lists in her illustrations (fig 4.9 j) and abstract work (fig 4.11 c). Producing numerous variations with Para led Kim to identify the need for versioning functionality.

Reflection: Kim reflected on how Para was uniquely suited for combining abstract and illustrative forms:

I think there's a space between abstraction and representation[3] *. . . this ambiguous in-between space where Para could be very useful.*

[3] Referring to abstract vs realistic (illustrative) imagery.

She also described how the process behind a work of art was as important as the work itself:

I want to create things that surpass the tool and it's not so obvious that some other process did all the interesting work.

As a result, she felt more successful when using Para in a way that obscured the use of obvious procedural aesthetics. In the closing interview, Kim described how working with Para had changed the way she thought about her own art process:

The inclusion of Para changes not just the visual output. It changes the process, and that's a lot of what I think about — how the tool changes the way you make things.

4.4 Opportunities, and Limitations of Para for Manual Artists

The components of Para provide one entry point into procedural art. Here the creative opportunities that resulted from the design of those components are analyzed around three principle questions. Was Para compatible with the skills of manual artists? Did Para offer meaningful creative opportunities? What are the limitations of dynamic direct manipulation for procedural art? Evaluation of Para demonstrated how extending manual drawing tools with declarative relationships allowed experienced artists to leverage existing skills. The studies also provided evidence for how "tinkerable" procedural tools retain meaningful forms of expressiveness for professionals. Finally, the studies suggest that procedural tools with concretely represented relationships can enable productive exploration of form, color, and composition.

4.4.1 Was Para compatible with the skills of manual artists?

The design of Para was partially based on the theory that a direct-manipulation procedural tool would be compatible with manual practice and enable manual artists to leverage existing skills. Two points of evidence support this theory. First, there are clear stylistic similarities between Kim's prior work and the pieces she created with Para, despite being produced with

different tools. The forms and compositions in her abstract paintings mirror her abstract work in Para, and her use of color and blending in her watercolor illustrations corresponds with her Para illustrations. These commonalities suggest that in addition to making use of its procedural affordances, Kim was able to transfer her prior skills and practices to Para. Second, both Kim and participants in the workshop described Para as "intuitive", "enjoyable" and "familiar," in contrast to their experience with textual tools.

Processing and similar programming tools are cool, but you're going to have to pick a number for a thing. Sometimes it's really hard to know what that number is supposed to be. ***It's a lot more obvious to just keep dragging it, and then be like, cool, that one looks good.*** –p3

*I feel like I can start with a blank canvas and then I can find something. A lot of how I work really resonates with this. You can create some parameters and you don't know exactly what it's going to be, and that's great. —****It can build a process or cycle in a very easy to enter and organic way.*** -Kim (speaking about Para)

The intuitive qualities of Para are important because, in addition to being approachable for learners, intuitive tools support specific creative practices. People who think through movement, intuition, and visual impression require computational tools that validate intuitive and relational mindsets in order to be personally expressive (Turkle and Papert, 1992). In line with this, Kim and many of the workshop participants stated that intuition played an important role in their art. This suggests that direct-manipulation procedural tools offer two ways to extend manual practices: by enabling intuitive manipulation of procedural relationships and by providing procedural techniques that are aligned with both the affordances and interaction paradigms of conventional graphic art tools.

4.4.2 Did Para offer meaningful creative opportunities?

A perceived trade-off in tool design is that the easier tools are to use, the less expressive they become. However, the results of the studies demonstrate that Para's accessibility *did not* prohibit expressive creation by professionals. Although Para contained a minimal number of procedural concepts, the artwork people created contained procedural forms and patterns comparable to those created with textual tools. The sketches produced in the workshop (fig 4.8 d) demonstrated radial distributions comparable in complexity and appearance to those created in Processing (fig 4.8 a-b).

Kim's illustration demonstrates parametric modulation in color, scale, and rotation across repeated forms, and her illustrative and abstract work contains generative forms and patterns. Parametric variation, generative form, and complexity via repetition are considered to be primary affordances of applying programming to art and design (Reas et al., 2010).

Para also offered new creative opportunities by facilitating new processes for making art. Kim described how the ability to rapidly produce and manipulate complex compositions facilitated exploration and iteration. She also felt that the generativity in Para led to new ways of thinking about the creative process, describing it as "a way of expanding one's creative mind". Finally, Kim identified unique creative opportunities in combining procedural exploration with manual control, which she noted were unique to Para. *I really liked being able to quickly get a lot of complexity, and then go back and work with the color relationship . . . there was a certain ease to it. There was this combination of not knowing what you could get, but also going back and forth.*

Overall study results and participant artwork demonstrate how a small number of reconfigurable procedural concepts can support new processes and aesthetics through procedural exploration of form, color, and composition, as well as support new ways of thinking about the creative process itself.

4.4.3 What are the limitations of dynamic direct manipulation for manual artists?

Despite the intuitive qualities of dynamic direct-manipulation tools like Para, they pose some limitations compared to programming languages. As I established in the analysis of prior dynamic direct manipulation systems, textual descriptions and expressions are sometimes better suited to describing precise or complex relationships when compared with images and direct manipulation. This was apparent when in Para, people sometimes struggled with interpreting and controlling complex geometric relationships through selection and dragging. People also became frustrated attempting to make precise patterns through direct selection, or in managing the organization of particularly complex compositions. In comparison to Para, and direct-manipulation software in general, textual programming tools can better support numerical evaluation and accuracy. Furthermore, computational abstraction as expressed through textual tools can greatly aid in complex organizational tasks. As Kim used Para to create increasingly complex compositions, she struggled to organize

and keep track of the numerous procedural relationships in a drawing. Effective organizational strategies, like other aspects of programming, are skills that programmers often develop over time. However, there are also numerous examples of programming interfaces and support tools that aid in the process of organizing complex programs, and assisting developers in analyzing and debugging their code and managing variations of a program. As dynamic direct manipulation tools like Para grow in their capacity to support the creation of complex procedural works, it is essential to also develop tools that enable artists to organize and manage the relationships that make up the work.

Another potential limitation of Para is its lack of an external representation. The absence of a representational programming language in a procedural tool can be a double-edged sword. The results of the studies strongly suggest that Para's direct manipulation played a large part in making the system inviting and intuitive. At the same time, the lack of an external representation prevented participants working with a "readable" description of their work. Like debugging, reading a program that describing one's work can be a powerful intellectual tool. It enables artists to reflect on the relationships in their work and consider ways of revising, refining, or abstracting these relationships. An external representation is also useful in communicating and sharing ideas with other people. Given the confidence people expressed when using Para, direct manipulation of procedural relationships may provide a way to make the challenge of learning abstract representational tools more approachable. With further development, tools like Para could assist manual artists in the process of understanding computational abstraction and benefiting from its application to their art.

5

Dynamic Brushes: Representational Programming for Manual Drawing

Para demonstrated how procedural tools can be meaningful for manual artists when these tools are compatible with manual practices. Yet, manual art creation is a broad, multi-dimensional domain. Para focused on the issue of integrating concrete engagement with procedural creation, but did not address many other aspects of manual practice. The interviews with manual artists highlighted the value artists place on manual drawing itself. In particular, artists focused on the unique forms of personal style and variation that emerge from the physical act of drawing, the ability of drawing to generate new ideas, and the speed at which drawing enables them to translate ideas to reality.

Discussions with procedural artists also suggested alternative creative opportunities in procedural art. Primarily, they demonstrated the power of designing one's own procedural tools for specific creative tasks. The interviews established how procedural artists use tool development as a way to extend their own practice and as a mechanism for collaborating with manual artists. Furthermore, procedural tool development demonstrated the overall power of programming to create functionality that is reusable by different people in different contexts. Collaborations between manual and procedural artists are one important way of integrating drawing and programming. Arguably many manual artists who lack opportunities to collaborate, could still benefit from procedural tools that are specific to their drawing practice.

The unique creative opportunities of manual drawing and the power of being able to develop personal procedural tools motivated the development of Dynamic Brushes: a system that enables manual artists to extend their

drawing practice with procedural behaviors of their own design. This chapter describes the design and development of Dynamic Brushes, a system that integrates a direct-manipulation bitmap drawing application with a visual programming environment for authoring procedural brushes. The creation of Dynamic Brushes required the development of a computational model that enables expressive and understandable descriptions of procedural drawing behaviors and incorporates data from manual drawing. It also necessitated the development of an interface that integrated procedural creation and manual creation, and the development of an interaction set that enabled artists to transition between programming and artwork creation.

This chapter begins with a description of the design goals of Dynamic Brushes. This is followed by an outline of the Dynamic Brushes programming model and software interface, combined with a description of the iterative design process that led to this design. The expressiveness of Dynamic Brushes is demonstrated through a series of sample brush behaviors and applications. Finally, this chapter outlines the evaluation of the system through an extended study with two professional manual artists. The results of this evaluation are analyzed within context of Dynamic Brushes' original design goals.

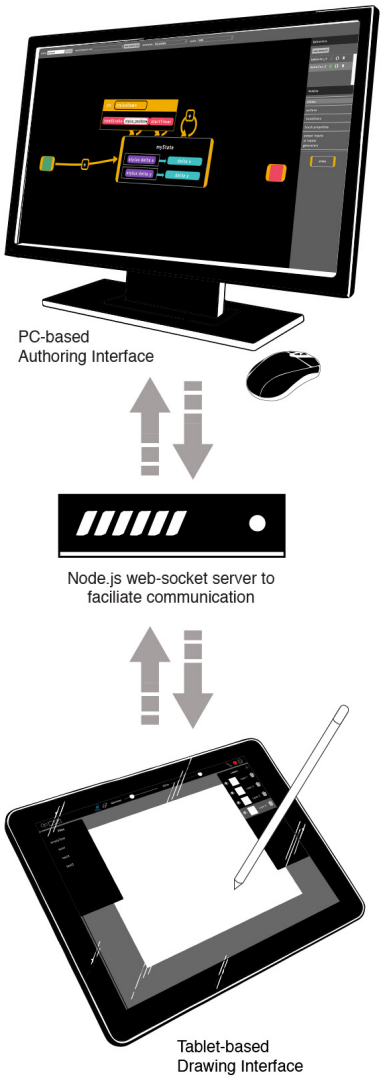

Figure 5.1: Dynamic Brushes is a multi-modal system, divided between a PC application for authoring brush behaviors and a tablet-based application for creating drawings with these behaviors. A web-socket server manages communication between the two interfaces.

5.1 Design Goals

Dynamic Brushes was motivated by a primary goal: creating a programming system that was compatible with manual drawing, and two supporting goals: developing the system in a way that was learnable for people who were new to programming, and retaining high degrees of artistic expressiveness. These goals correspond with the meaningful properties of manual and procedural practices revealed in the prior interviews with professional artists and observations of artists at work. Manual artists value the stylistic diversity, manual variation, and creative exploration that manual drawing enables. They are interested in the opportunities of procedural tools to enable introspection into their practice, or support new forms of expression, but are hesitant to use procedural tools which are opaque, present high learning thresholds, and restrict the ability for manual improvisation and variation. Similarly, procedural artists value procedural tools for their ability to support dynamic artifacts and processes, and the opportunity they provide to develop both personal and public creative tools. However, even many procedural artists value opportunities to integrate manual control and organic variation into procedural artwork.

The remainder of this section outlines a series of design criteria that respond to these specific opportunities and challenges of procedural and manual practices.

Compatibility with manual practice

- **Support procedural manipulation of manual drawing data:** The programming model should be designed to create behaviors that respond to physical properties of manual drawing, temporal information about a drawing or mark, and analysis of the geometric properties of a drawing.
- **Be compatible with the "liveness" of drawing:** The system should perform operations and produce visual output in real-time. It should also enable artists to make procedural edits in the process of drawing.
- **Privilege manual creation and programming equally:** The programming model should contain features that facilitate drawing by making it easy to set and dynamically change stylistic and geometric properties of a brush. The system interface should contain elements designed explicitly for manual drawing.
- **Preserve the personal expressiveness of individual artists:** The system should enable drawings created with the same procedural functionality by different people to reflect unique styles of different artists.
- **Appeal to manual artists:** Artists who do not use programming in their practice should be interested in using the system.

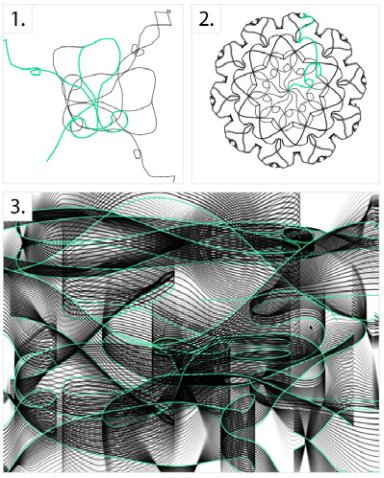

Figure 5.2: Samples of prototype procedural brushes. Green line indicates the path of the manual stroke and black lines are procedural strokes. 1: Mirroring behavior with 1 level of recursion. 2: Radial-symmetrical behavior with 3 levels of recursion. 3: Exponential-decay behavior where original stroke is distorted along its inflection points.

Expressiveness

- **Support extended practice:** Professional artists should be able to use the system for an extended period without reaching a creative "ceiling". The system should enable experienced artists to create complex procedural functionality.
- **Provide access to primary procedural techniques:** The system should enable artists to apply procedural techniques comparable to those used by professional procedural artists working with conventional procedural tools and programming languages.
- **Support visual diversity:** Behaviors created with the system should support different styles of illustration.

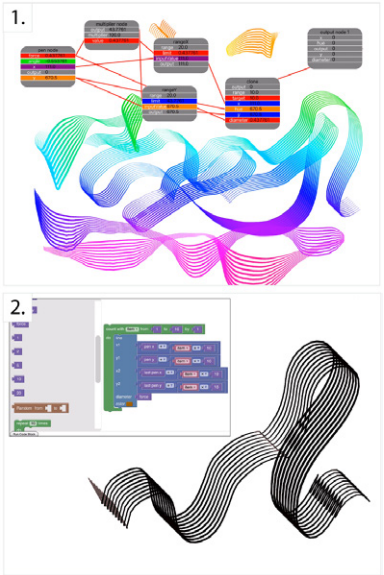

Figure 5.3: Functional prototypes of behavior creation systems using existing visual-programming paradigms. 1: Dataflow prototype. 2: Block-based prototype.

Learnability

- **Support a wide range of expressiveness, with a minimal number of procedural concepts and structures:** To reduce learning thresholds, the programming model of Dynamic Brushes should require fewer components and control structures for creating procedural artwork compared to textual creative coding libraries.

- **Support discovery and tinkerability:** Artists should be able to independently discover and experiment with many aspects of the programming language.

- **Provide accessible starting points**: People who are using the system for the first time should be able to quickly get started creating work.

- **Enable interesting results from simple programs:** In addition to supporting complex outcomes through extended use, the system should allow for compelling artwork to be produced with simple behaviors. As artists incrementally learn new aspects of the system, they should be able to access new creative opportunities.

5.2 Design Process

Development of Dynamic Brushes was first motivated by the development and evaluation of Para, and through interviews with artists. Construction of the Dynamic Brushes system began by prototyping a series of sample brush behaviors using existing programming languages (fig 5.2) to explore concrete applications of the proposed system. This was followed by the prototyping of a general procedural model that would allow for the creation of similar behaviors and simultaneously prototyping static and interactive interfaces for expressing this model. Para focused on object-to-object relationships, which could be represented through visual icons on top of the artwork. Conversely, sample Dynamic Brushes behaviors responded to changes over space *and* time. Representing temporal behaviors exclusively in the context of a 2D canvas would be challenging and potentially misleading. This led to exploring external representations, in addition to the drawing canvas.

The Dynamic Brushes programming model and interface were created simultaneously. While the interface was in development, feedback was solicited from professional interaction designers, and early versions of the system were tested in informal sessions with artists. A key aspect of

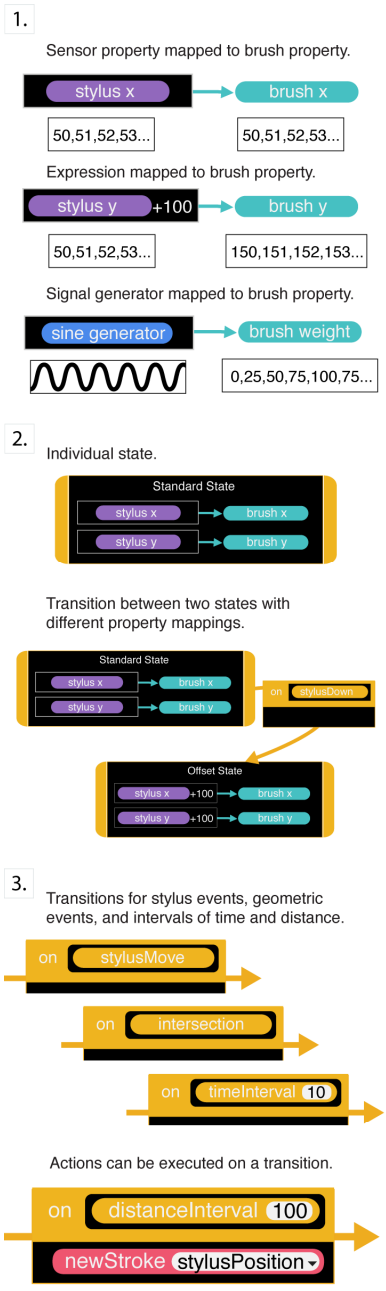

Figure 5.4: Components of the Dynamic Brushes Programming Model. 1.Property Mappings. 2.States. 3.Transitions.

informing the Dynamic Brushes programming model involved understanding how it should align or diverge from existing languages. As a result, two initial functional prototypes were created with dataflow and block-based paradigms (fig: 5.3).

The dataflow approach enabled simple declarative mappings between sensor data and drawing output and the implicit looping of data flow languages made it easy to map input to multiple brush outputs. But there were also issues with dataflow. Dataflow languages traditionally do not provide control over the program's state; data is automatically piped through the system as soon as it is available (Johnston et al., 2004). Yet early sample brush behaviors demonstrated the utility of changing state on discrete events. Dataflow was not well suited to this functionality. The block-based prototype was imperative and therefore better suited to describing state-changes. However, it also made the process of describing and maintaining relationships between data and output more complex in comparision to the declarative structure in the dataflow prototype. Prior systems in UI behavior programming and data visualization demonstrated computational models that combined imperative and declarative paradigms (Aish, 2012; Oney et al., 2014; Victor, 2013b). The experiments with existing paradigms, research into hybrid declarative-imperative tools, and an iterative design process collectively resulted in a final programming model of states, event transitions, and constraints.

Position Props.	**Description**
x,y	Absolute cartesian position.
radius,theta	Absolute polar position, relative to the brush origin.
delta x,y	Relative position, i.e. distance the brush should move from its current position.
Geometric Props.	**Description**
scale x,y	Scale of the segment relative to the stroke's origin.
rotation	Rotation of the segment relative to the stroke's origin.
Stylistic Props.	**Description**
diameter	Stroke width in pixels.
hue	Hue of the stroke color.
saturation	Saturation of the stroke color.
lightness	Lightness of the stroke color.
alpha	opacity of the stroke.
Hierarchical Props.	**Description**
spawn index	index of instantiation relative to all other instances spawned by the brush's parent.
sibling count	Total number of instances spawned by the brush's parent.
spawn level	The number of connections between the brush instance and the root brush (i.e. if *brush a* spawns *brush b* which spawns *brush c*, the level of *brush c* is 2.)

Table 5.1: All brush properties.

5.3 Programming Model

Programming in Dynamic Brushes is structured around the creation of brushes: interactive objects that procedurally transform manual input from the artist to draw visual brush strokes on the canvas. Drawing is initiated whenever the brush registers a change in its position. When a change occurs, the brush moves to the new position and draws a segment from its previous position to its current position. In addition to position, each brush object has properties that define the style and geometry of its strokes (table 5.1). Style and geometric properties are passive—changes to them do not update the drawing. Instead they are accessed and rendered whenever position changes.

The Dynamic Brushes programming language enables artists to generate brush objects, and describe how and when brush properties should change. At its core, the programming model is based on a finite state-machine with three primary mechanisms to describe brush behavior: property mappings, states, and event-driven transitions.

Property mappings are continuous relationships that assign brush properties to reference data (fig 5.4-1). Mappings can be created between any brush property, including position, scale, line weight, opacity, and color and a reference value. References range from stylus data, mathematical expressions, signal generators, UI component values, or some combination of these. Table 5.2 lists all possible reference types for property mappings. Similar to the constraints in Para, property mappings are declarative and brush properties are automatically updated as the reference value changes.

Property mappings are stored within *states* (fig 5.4-2). When a state is entered, it activates all property mappings within it. As long as the state remains active, property mappings are automatically updated as their reference changes. When a transition to another state occurs, the former state becomes inactive. Each brush instance can only have one active state at a time. In addition to standard states, the model contains two specialized states, a setup state that each behavior starts in when first initialized, and a die state. When the die state is transitioned to it destroys the brush instance and removes it from memory. Setup and die states cannot contain property mappings.

Mapping Datatypes	**Description**
Constants	A constant numeric value.
Sensors	Incoming data from a sensor. Currently the system supports data from the stylus including force, x and y position and delta, angle, and speed.
UI Values	Values generated from the sliders and color-picker in the drawing interface.
Generators	Datatypes that produce a sequence of values corresponding to a given function—modulation with a sine function, or a random sequence of values.
Brush Properties	References the current properties of a brush.

Table 5.2: Reference types for property mappings.

Movement between states is managed by event-driven *transitions*. Transitions are unidirectional connections that begin at one state, and end at a second state. Transitions can be triggered by stylus events, geometric events, and intervals of time (table: 5.3). Each transition can contain a list of actions that are executed once, in order, when the transition occurs. The number of actions in Dynamic Brushes is much smaller than the number of brush properties or reference types. Actions can be used to initialize new strokes, control timers or to initialize new brushes (table: 5.5). Where applicable, both transitions and actions can accept arguments that determine their behavior.

When a behavior is initialized, it generates a brush-instance. In object-oriented programming terms, behaviors are classes and brushes are class-objects. When first created, all behaviors automatically generate one brush instance. Each time a behavior is edited by the artist, all instances are

Event Name	Description
onComplete	Triggered immediately after a state is entered and all mappings are activated. This is the default event when a transition is first created.
stylusDown stylusUp stylusMove	Triggered once when the stylus touches the tablet, is lifted from the tablet, or repeatedly as the stylus moves across the tablet, respectively.
stylusMoveBy stylusXMoveBy stylusYMoveBy	Triggered when the stylus moves by an interval of pixels designated by the event argument. stylusX and stylusY MoveBy events measure horizontal and vertical distance respectively. stylusMoveBy measures euclidean distance.
timeInterval	Triggered every time the number of seconds specified by the argument elapses.
distanceInterval	Triggered every time the brush has traversed a distance interval specified by the event argument. Useful in cases when brushes move independent of stylus movement.
intersection	Triggered every time the brush intersects with a stroke already drawn on the canvas.

Table 5.3: All transition event types in Dynamic Brushes.

"refreshed", meaning they are destroyed and re-generated by the system. Behaviors can be switched to inactive mode, which prevents the system from automatically generating an instance.

Object-oriented programming can be challenging for new programmers to understand (Kay, 1993). Therefore, the auto-initialization structure in Dynamic Brushes enables artists to begin creating without explicitly having to create brush instances. As artists gain experience with the behavior model, they have the option of leveraging some of the power of object instantiation by creating behaviors that spawn multiple brush instances (see section: 5.5.3). The next section describes how the Dynamic Brushes interface enables artists to create and apply behaviors.

5.4 Interface

The Dynamic Brushes interface is split between two applications: a PC-based *authoring* interface, and a tablet-based *drawing* interface (fig 5.1). The drawing interface runs on an iPad Pro and is implemented in Swift and Objective C. The authoring interface is implemented in JavaScript and uses a modified version of the jsPlumb flowchart toolkit [1] to display the behavior diagrams, and the CodeMirror text editor to display expressions in the property mappings [2]. All behavior processing is handled in the drawing application. Communication between the two applications is facilitated via a Node.js websocket server.

[1] jsPlumb. `jsplumbtoolkit.com`. Accessed July 21st 2017.

[2] CodeMirror. `codemirror.net`. Accessed July 21st 2017.

The authoring interface enables artists to create and edit brush behaviors (fig 5.5 1). It is divided into three sections, a horizontal toolbar, a side panel, and the authoring canvas in the center. The toolbar contains a menu

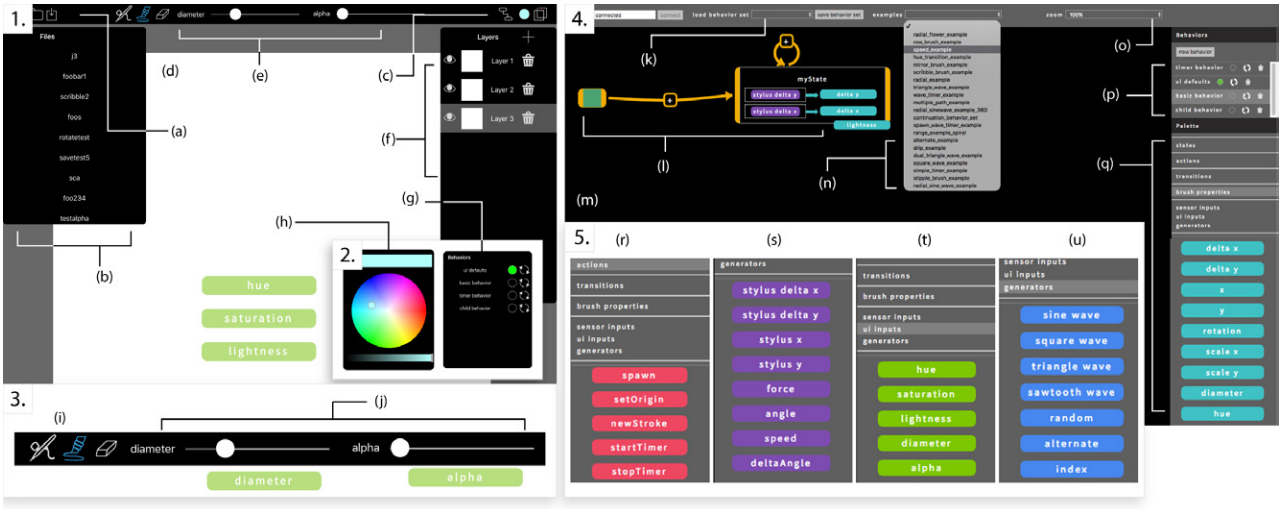

Figure 5.5: Dynamic Brushes interface.

1. Tablet drawing interface.
   (a) Drawing save, load and export menu.
   (b) Load menu.
   (c) Behavior, color-picker and layer panel toggle.
   (d) Drawing canvas.
   (e) Drawing toolbar.
   (f) Layer panel.
2. Detail of drawing interface panels.
   (g) Behavior selection panel.
   (h) Color-picker panel with corresponding UI behavior properties.
3. Detail of drawing toolbar.
   (i) Airbrush, pen, and erase buttons.
   (j) Brush diameter and alpha sliders, with corresponding UI behavior properties.
4. PC Authoring interface.
   (k) Behavior set load/save menu.
   (l) Behavior state diagram.
   (m) Behavior canvas.
   (n) Example menu.
   (o) Canvas zoom control.
   (p) Behavior selection panel.
   (q) Behavior palette with brush properties visible.
5. Detail of behavior palette property menus.
   (r) Detail of action palette.
   (s) Detail of sensor palette.
   (t) Detail of UI palette.
   (u) Detail of generator palette.

for saving and loading behavior sets (k), a menu of example behaviors (n), and control for zooming on the main canvas (o). The panel contains a list of the current behaviors (p). When a behavior is selected, its state diagram is displayed in the canvas for editing. The behavior panel also contains buttons to activate, deactivate, or reset a behavior (fig 5.7 top).

Below the behavior selection panel is the behavior palette. From here, artists can select and drag behavior components onto the canvas. Figure 5.5-5 shows a detail of each sub-menu of the palette and its corresponding components, with type distinguished by color. Similar to palettes in block-based languages, the Dynamic Brushes palette is designed to make it easy for people to view and explore all possible behavior components without consulting an external reference. When hovered over, each component displays a textual or animated tool-tip explaining its use (fig 5.6). Artists compose behaviors in the canvas using components from the palette. Transitions between states can be created by dragging from anywhere on the yellow borders of the state and dropping the resulting connection on top of another state. An arrow indicates the direction of the transition. Each transition has a plus icon that, when clicked, opens a panel with the transition event. By default all transitions are created with a *complete* event, which triggers the transition as soon as the state is entered. Artists can change the transition event by dragging in a different event from the palette. The drawing interface enables artists to apply behaviors created in the authoring interface to the creation of artwork (fig 5.5-1). The drawing interface was designed to be familiar to people with experience in digital illustration and contains analogs of basic UI controls from bitmap drawing software. The drawing canvas (d) can be rotated and scaled using two fingers, and layers can be added and removed using the layer panel

Generator Type	Description
Sine Wave	Returns values that smoothly oscillate between 0 and 100.
Square Wave	Returns values that are either 0 or 100.
Triangle Wave	Returns values that increase and decrease at a constant rate from 0 to 100.
Sawtooth Wave	Returns values that increase at a constant rate to 100 then drop sharply to 0.
Random	Returns uniform random values.
Alternate	Returns a series of alternating values as described by the generator argument (e.g. 0,50,0,75…).
Brush X Buffer Brush Y Buffer Brush Diameter Buffer	Brushes maintains a record of their past x, y, and diameter values. Buffer generators return those values in order of their generation. Buffers can be modified to access the parent's record with a boolean argument.

Table 5.4: Reference types for property mappings.

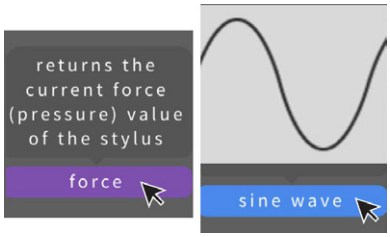

Figure 5.6: Tool-tips displayed when artist hovers over a behavior component. Left: textual tool-tip describing a sensor property. Right: Animated tool-tip illustrating the signal returned by a generator.

(f). The layer panel can be toggled with two other panels (fig 5.5-2) a color-picker (h) or behavior selection menu (g), which displays all behaviors currently available from the authoring interface. The toolbar (fig 5.5-3) contains UI controls for adjusting diameter and alpha brush parameters (j), and selecting between airbrush, pen, and erase drawing modes (i).

5.5 Authoring Behaviors

Collectively, the Dynamic Brushes programming model, visual programming language environment and stylus-based drawing environment, support different approaches to art creation, and different outcomes. This section outlines three categories of behaviors that can be created with the system: simple behaviors, automated behaviors, and spawning behaviors. The creative affordances of category are illustrated through sample applications.

5.5.1 Simple Behaviors

Behaviors are created using the *new behavior* button in the behavior selection panel (fig 5.7-bottom). Each behavior comes default with a green setup state and a red die state. Artists can add additional states by dragging them out from the palette and labeling them. Dynamic Brushes was designed so that new programmers could start by experimenting with behaviors from the example menu. however the system also contains scaffolds for helping people to create new behaviors from scratch. When a behavior is created, artists can select from a menu templates that are pre-populated with states, transitions and property mappings (fig: 5.7-bottom). This structure avoids presenting new programmers with a blank canvas and suggests starting points for common design patterns.

Figure 5.7: Detail of the behavior selection panel. Top: list of current behaviors with options for deleting, refreshing, or toggling the active state of each behavior. Bottom: Menu of behavior templates that providing starting points for common behavior design patterns.

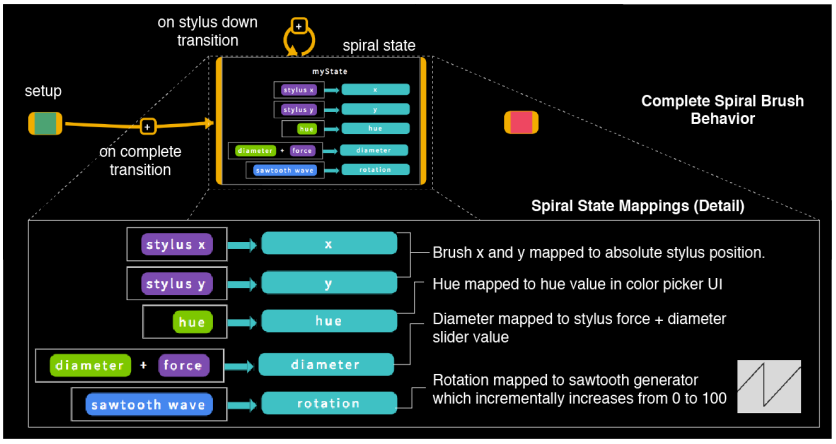

Figure 5.8: Basic brush behavior.

Selecting the basic template creates a behavior with one state with two property mappings that map the brush x and y position to the stylus x and y position. The default state is transitioned to immediately from the setup state when the behavior is initialized with a complete transition. A second transition, activated on a *stylusDown* event transitions back to the default behavior, calling the *newStroke* action in the process. Transitions that start and end at the same state are referred to as looping transitions.

This behavior creates a brush that mimics the functionality of a traditional bitmap brush; the brush follows the path of the stylus, and creates a new stroke every time the stylus is lifted and touched again to the canvas. Although extremely basic, this simple structure can be used to produce a variety of effects with a few additional property mappings. Property mappings can be added to a state by dragging any brush property from the palette and dropping it into the lower portion of a state. This adds the property, along with a corresponding box that accepts both text input and dropped-in reference types. A left-to-right arrow connects the two box to the brush property, indicating the direction of the mapping.

Dragging the hue, alpha, and diameter properties into the state and assigning them constant numeric values enables control over the color, opacity, and line-thickness of the brush. Artists can enact dynamic control over these properties with *UI references*. UI references return the current value of the alpha and diameter sliders, and the color picker in the drawing interface. Mapping the diameter, alpha, and hue properties in the simple behavior to corresponding UI references enables the brush line-thickness, opacity, and color to be controlled by these interface elements. This control can be modified further to create relationships between the UI components and the stylus by combining multiple reference types with simple mathematical expressions. Mapping the brush diameter to

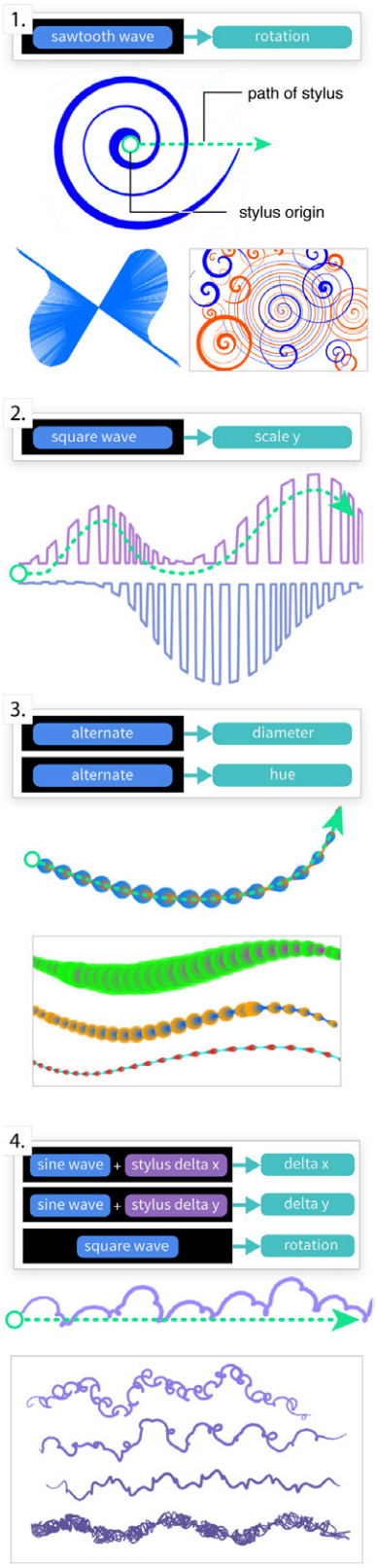

Figure 5.9: Variations on the basic brush behavior produced by mapping brush properties to generators.

$stylus\ force + UI\ diameter$ results in a stroke weight that is modulated by a combination of the position of the diameter slider and the pressure the artist exerts on the stylus. This effect can be controlled further by specifying a more complex expression.

Table 5.5: All actions in Dynamic Brushes.

Action Name	Description
startTimer stopTimer	Starts and stops the brush's internal timer, which is used when triggering timerInterval events.
newStroke	Instructs the brush to end the current stroke and begin a new one at the origin point specified in the action argument (stylus position, parent position, or canvas origin).
setOrigin	Similar to newStroke, it resets the origin point for the brush to the position specified in the action argument. Useful when specifying the brush position by vector rather than absolute position.
spawn	Instantiates a set number of brushes of of a given behavior as specified in the action arguments.

Dynamic Brushes enables artists to easily construct behaviors that resemble standard digital bitmap brushes, but the system's real power lies in the ability to produce functionality that goes beyond non-procedural digital tools. The default brush behavior can quickly be converted to a brush that creates a spiral pattern through the use of one additional mapping on the rotation property. Strokes in Dynamic Brushes are represented as a series of segments generated between the current position and previous position of the brush as it moves across the canvas. Each stroke has an origin that corresponds with the starting point of its first segment. The brush rotation property specifies the degree to which the current segment should be rotated relative to the origin of the stroke. Changing the value of the rotation property as the brush position changes results in transformations to the stroke path. Mapping the rotation property to a *sawtooth* generator results in the rotation repeatedly increasing to 100, and then dropping to zero. This produces a brush that draws a spiral pattern as the artist moves away from the origin of the line (fig: 5.9-1). The rate at which the artist moves the stylus controls the tightness of the spiral. This basic generator mapping structure can be applied to any brush property to produce brushes that modulate in color, position, and scale as the artist draws across the canvas (fig 5.9). Signal generators are common in music synthesizers and audio applications (Jordà et al., 2007). Dynamic Brushes demonstrates how they can also serve as powerful, yet simple tools for augmenting manual drawing with automated variations in the drawing's geometry and appearance.

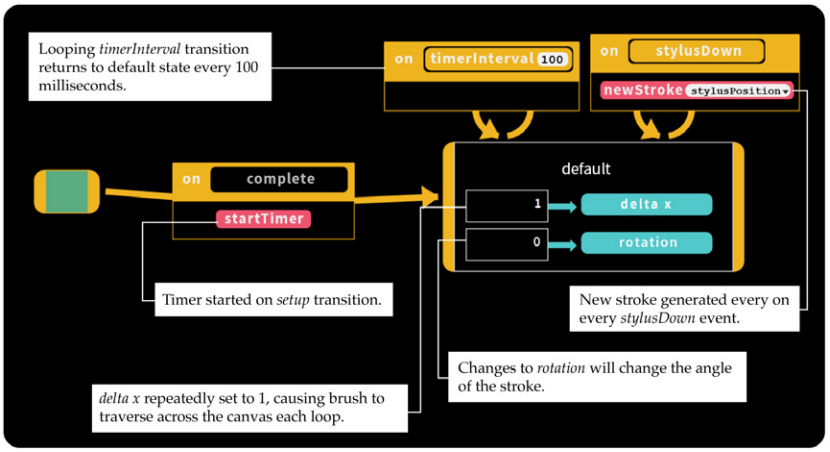

Figure 5.10: Basic timer behavior template. This template can be used to create brushes that draw automatically.

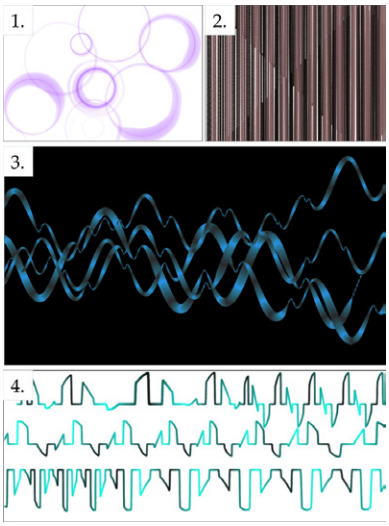

Figure 5.11: Brushes created from the timer template. 1. Automatically drawn circles with manual stroke diameter modulation based on stylus force. 2. Automated cross-hatching. 3-4. Automatically drawn wave-forms with spacing, amplitude, and stroke diameter modulated by the stylus position and force.

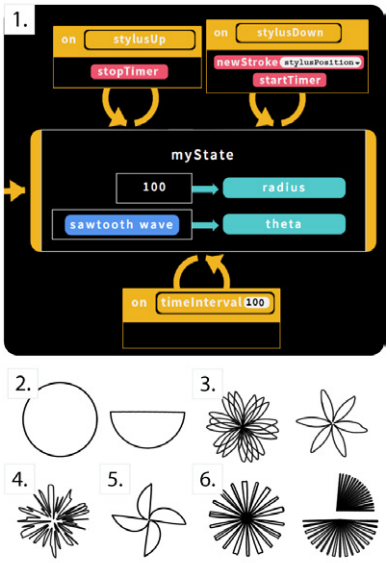

Figure 5.12: Radial forms drawn with timer-driven behaviors. 1: Behavior that draws a circle with radius 100 wherever the stylus is pressed. 2: Circle drawn with behavior, and half circle created by mapping theta to $(sawtooth\ wave \div 2)$. 3-6: Variations created by mapping the radius property to different generators.

5.5.2 *Automatic Behaviors*

The basic behavior in Dynamic Brushes results in procedural brushes that respond to stylus movement, however procedural tools also offer the opportunity to describe actions that operate independently from the actions of a human artist. Dynamic Brushes supports the creation of behaviors that operate automatically through time-driven behaviors.

Each brush has an internal timer that can be switched on and off using the *startTimer* and *stopTimer* actions. The system also contains a *timerInterval* event, which is triggered when an interval of the specified number of milliseconds has passed. The combination of timers and timerInterval events enables animated brushes that change state automatically as the timer advances. The timer template in the behavior menu provides a starting point for creating animated brushes. Like the basic behavior template, the timer template has a complete transition from the setup state to a default state and a looping stylusDown transition that calls the newStroke action. The timer template calls the startTimer action in the setup transition. It also contains a second looping transition on the default state that is triggered every 100 milliseconds on a timerInterval event (fig 5.10).

The default state sets the brush position through the delta x property. Unlike absolute position, delta properties specify the relative distance the brush should move when the mapping is active. As a result, whenever a delta mapping is activated, any value other than zero will result in a change in position. The looping timerInterval transition continually activates these mappings, producing a brush that automatically draws a line across screen at a rate of one pixel every 100 milliseconds. Every time the artist touches the stylus to the canvas, the current stroke stops drawing, and a new one begins at the stylus position.

Variations on the timer structure enable the production of regular geometric shapes and patterns. Adding in a mapping that sets the rotation allows for lines to be drawn at different angles. Moving the startTimer action to the stylusDown transition, and adding a looping stylusUp transition with a stopTimer action enables the artist to start and stop each line by pressing and raising the stylus, allowing for the rapid creation of grids and cross-hatch effects (fig 5.11-2). Mapping a sine wave, triangle wave, or square wave generator to either the delta x or absolute x property will produce a regular oscillating line in a variety of patterns (fig 5.11-3,4).

Pairing timer-driven behaviors with polar coordinate mappings produces regular radial forms. By keeping the polar radius of the brush constant and mapping the polar angle of to a sawtooth wave, it's possible to create perfect circles. Mapping the radius to different generators results in pinwheel, star, and scribble-like radial patterns (fig 5.12).

Dynamic Brushes was developed to integrate procedural and manual creation, and timer-driven behaviors may appear to eliminate manual input. However, it's also possible to use these behaviors to automate some properties and manually adjust others. This enables artists to incorporate manual variation into regular forms and patterns. Mapping the brush radius to the UI-diameter slider in the circle drawing behavior enables the artist to dynamically control the radius of each circle while it's being drawn. Similarly, mapping brush diameter to stylus force enables the artist to change the line thickness by pressing harder or softer on the stylus (fig 5.11-1). When multiple properties are mapped to UI reference values, the artist can use both hands to change the appearance of the stroke as it's being drawn. Stylus properties can also be used to tweak the appearance of animated brushes. For example, modifying the delta y mapping reference to an expression that adds the stylus delta to the value of the sine generator enables the artist to change the depth of each wave in the pattern by moving the stylus up and down on the canvas. This results in a pattern with even spacing on the x-axis and variation on the y-axis. Collectively, these examples demonstrate how timer-driven behaviors enable artists to combine procedural regularity with manual changes.

Despite the variety of different effects they produce, all of the behaviors presented thus far have contained only one state. Adding additional states to the timer behavior creates opportunities for new patterns and effects. Rectangles, squares and regular polygons can be produced with two states: the original default state, and a second "rotate" state. A single mapping in the rotate state sets the rotation of the brush to its current rotation plus some constant [3]. A timerInterval transition from the default state to the rotate state and a complete transition back from rotate to default produces

[3] Rotation, like all passive brush properties ranges from from 0 to 100, 100 being equivalent to 360 degrees. Because all generators return values between 0 and 100, this conversion makes it more straightforward to map generators to brush properties and makes it easier to write expressions.

a behavior where the brush repeatedly draws a straight line, and rotates itself, until it has transcribed a polygon. Changing the reference value of the rotation mapping will change the number of sides of the polygon. This approach demonstrates how Dynamic Brushes can approximate the geometric drawing capabilities of textual procedural tools like Processing, but in a manner that, by default, enables the artist to specify the position of the geometric shapes by pointing with the stylus. Coincidentally, the method for drawing regular geometric shapes in Dynamic Brushes is similar in structure to the approach in the Logo programming language.

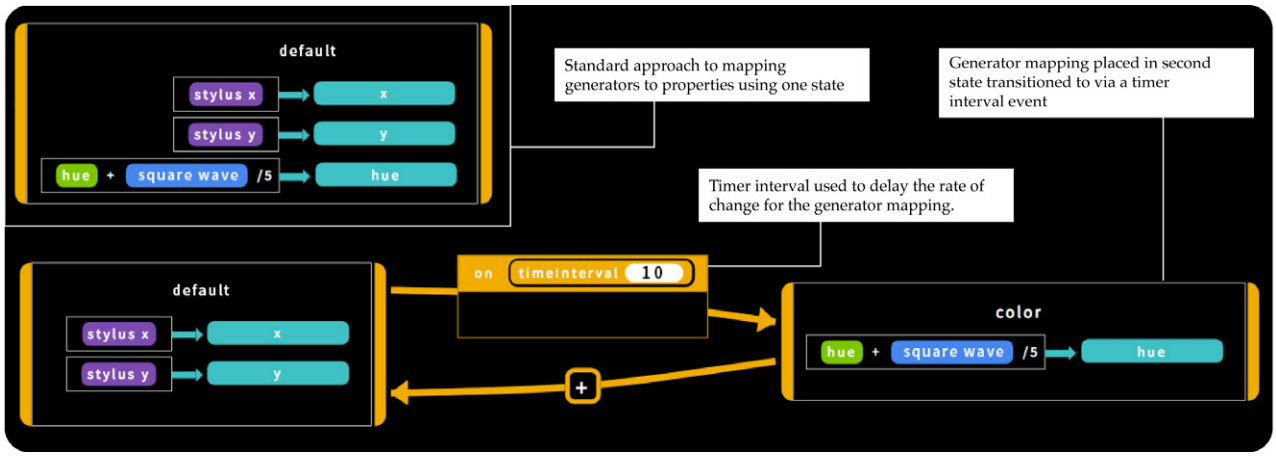

Figure 5.13: Comparison of single state behavior with generator mapping, and behavior using a *timerInterval* transition between two states to control the rate of updates from the generator.

Multiple states and timer intervals can also be used to control the rate of change for brush properties that are mapped to generators. Figure 5.14-1,a demonstrates a brush that switches between red and blue colors as it draws using a square wave generator mapped to the hue property.

The current Dynamic Brushes interface does not enable the artist to adjust the frequency of the generators directly. However, multiple states can allow the artist to adjust the rate at which generators are activated, and thus control the rate of color change for the behavior. Moving the square wave-to-hue mapping to a second state, transitioned to on a *timerInterval* event makes it possible to slow down the rate of color change by increasing the duration of the interval (fig 5.13). This timer-driven approach enables fine-grain control of rates of change, while removing the need for additional interface mechanisms for adjusting generators.

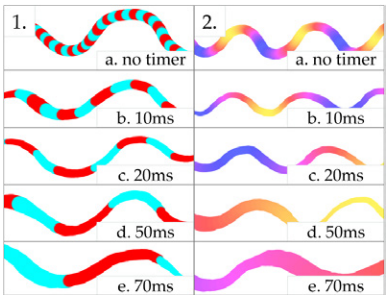

Figure 5.14: Variations produced by using a *timerInterval* transition with a generator. 1.a, 2.a: Brushes that map a square wave or sine wave generator to hue using a single state. 1.b-e, 2.b-e: Brushes that move the generator mapping to a second state, transitioned to on a *timerInterval* event. Longer timer intervals result in greater distance between each color transition.

5.5.3 Spawning Behaviors

When compared to textual procedural art systems, one potential limitation of all of the brush behaviors described up to this point is that they do

not allow multiple drawing actions to be performed simultaneously. This limitation can be partially addressed by manually creating and activating multiple behaviors with different property mappings. This approach works well for up to two or three brushes, but doesn't scale when attempting to draw five, ten, or fifty strokes at the same time, because each additional stroke requires an additional behavior. This is where the *spawn* action has value. Calling *spawn* enables a brush to generate a specific number of instances of any other behavior (including its own). This enables the creation of behaviors that generate multiple brushes in response to a single input. Use of the spawn method is facilitated through a design pattern with two behaviors. One behavior, the "parent", acts as a mechanism for spawning instances of a second behavior, the "child." The behavior-creation menu's *parent* and *child* templates provide starting points for this pattern. The child behavior consists of a single default state with two transitions. The first goes from setup to default on a complete event, and calls the newStroke action in the process. The second transition goes from default to the die state on a stylusUp event. The parent behavior consists of a single default state and two transitions: a complete transition from setup and a looping transition that occurs on a stylus-down event and calls the spawn action. Spawn actions have two arguments, a dropdown menu that lists all current behaviors by name and a box for numeric entry. These arguments allow the artist to specify the behavior that should be spawned and the number of brush instances that should be created.

The combination of the parent and child behaviors results in 10 brush instances being generated on each stylus-down event and also removed on each stylus-up event. To enable variation among procedurally spawned brushes, Dynamic Brushes stores brush instances in a hierarchical data structure, and enables brushes to reference information about their position in the hierarchy (table 5.1-Hierarchical Props.) The *spawn index* property returns the brush's index of instantiation relative to all other instances (or siblings) spawned by the brush's parent. Using this property as a multiplier in a mapping of child behavior enables each brush spawned from that behavior to return a different value for the mapping. Figure 5.15-1 demonstrates forms of artwork produced with this approach using a brush that draws multiple lines in a vertical row, parallel to the path of the stylus. Each spawned brush's y position is mapped to $stylus\ force \times (stylus\ y + spawn\ index)$. Figure 5.15-3 demonstrates a similar approach where the parent spawns five child instances. For each child, rotation is mapped to $spawn\ index \times 100 \div 5$. This produces a pattern with radial symmetry. Another hierarchical property, sibling count returns the total number of brushes spawned by the parent. Dividing the property of the spawn index by that of the sibling count in the radial behavior enables the spawn-count argument in the parent behavior to be

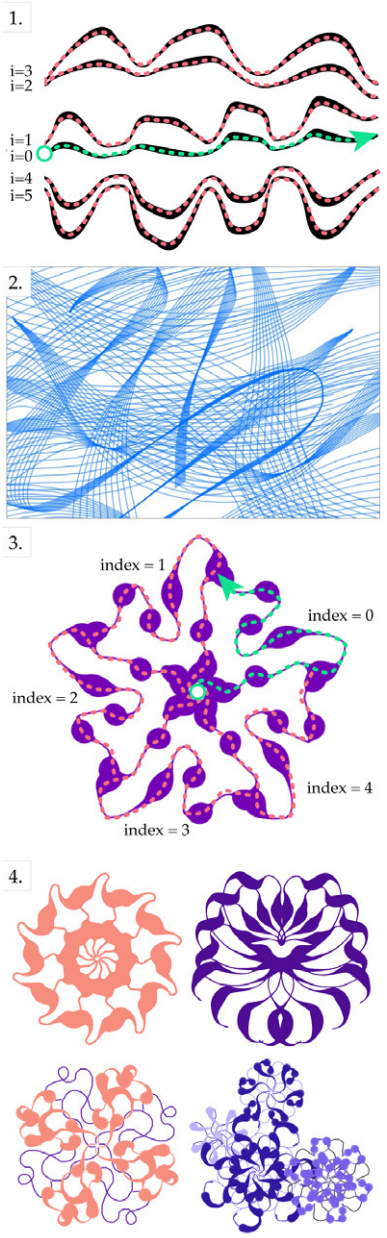

Figure 5.15: Designs created with spawning behaviors. 1. Row brush with vertical waves produced in response to stylus pressure (green-dashed line indicates the path of the stylus). 2. Variation of the row brush behavior. 3. Radial design produced through spawing and iterative rotation. 4. Variations of the radial behavior.

dynamically increased while preserving the radial symmetry of the pattern (fig 5.15-4).

Procedural instantiation can be combined with timer-driven behaviors to produce brushes that generate multiple brush instances that draw independent of the stylus. A “drip brush” (fig 5.16) can be created by pairing the pattern for parent-child spawning with a child behavior that draws on a timerInterval loop. As the parent instance draws, it spawns one child instance on a timerInterval. When spawned, the child immediately transitions to a default state, which maps the y delta to 1 on each timerInterval loop. After 1500 milliseconds, the child behavior transitions to the “drop” state. This state is identical to the default state, except it also maps the brush diameter to a sine-wave generator. After 500 milliseconds, the brush transitions to the die state. The result is that each child brush draws a line with a drip-like appearance by first increasing the diameter gradually and then rapidly decreasing it right before the line ends.

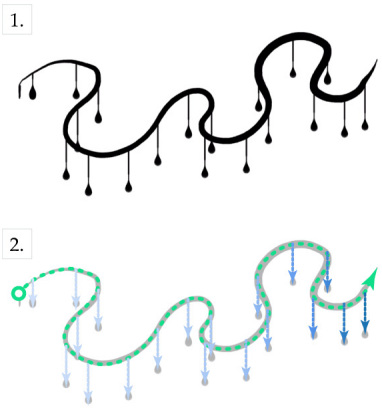

Figure 5.16: Drip Brush behavior. 1. Sample drawing. 2. Visualization of stylus path in green and the path of the spawned child brushes in blue (lighter blue indicates the brush was spawned at an earlier in time).

The combination of procedural instantiation and timers also enables the production of art that resembles work created by professional procedural artists with textual programming languages. Figure 5.19 compares a procedural illustration created in openFrameworks by the artist Zach Lieberman to an illustration created in Dynamic Brushes. The Dynamic Brushes piece is created by extending the drip behavior. Rather than spawing a child brush on a timeInterval event, the modified parent behavior (fig 5.18) calls spawn on a distanceInterval event. This ensures regular spacing between each child brush. The child brush (fig 5.17) contains three states. The first state, “delay,” is transitioned to from setup on a complete event and sets the brush lightness to 90. On a stylusUp event, the delay state transitions to the default state, which loops on a timerInterval to automatically draw a horizontal line.

The child behavior uses the technique described in section 5.5.2 to gradually change the lightness of the child brush from light grey to black as it draws across the canvas. Finally, the default state features two transitions to the die state. One occurs on an intersection event, which takes place if the line intersects with the white parent line. The other occurs on a distance event, once the line has drawn across the other canvas or beyond it. The result is a wavy line drawn manually with the stylus, with multiple horizontal lines that extend from it and appear to fade into the distance.

Collectively these authoring examples demonstrate how the Dynamic Brushes model supports a high degree of procedural expressiveness with a small number of procedural concepts and properties. This is underscored by the fact that Dynamic Brushes can reproduce forms and patterns similar

to those created with textual procedural tools, through a model with fewer procedural components, and with the addition of manual control.

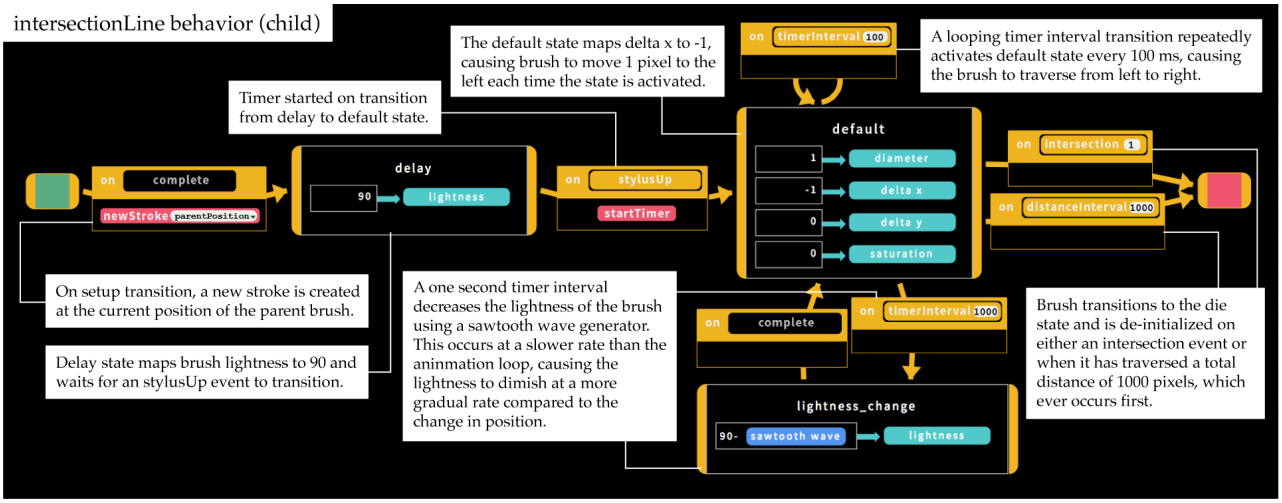

Figure 5.17: Intersection-line child behavior.

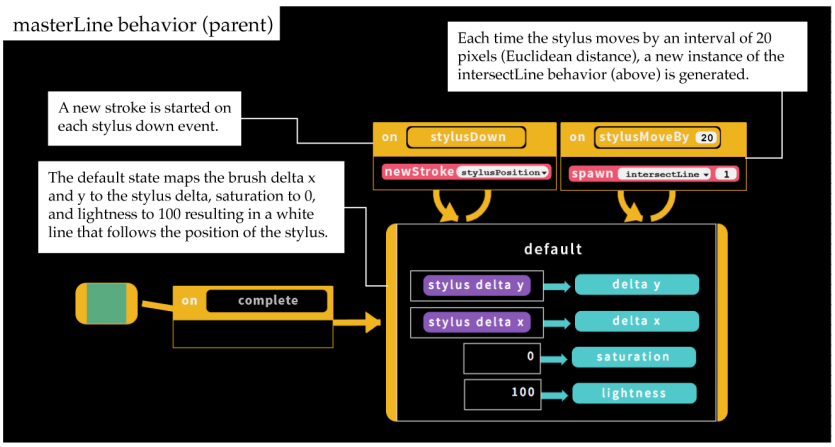

Figure 5.18: Master line behavior, which spawns the intersection-line child behavior.

5.6 *Evaluation*

The evaluation structure for Dynamic Brushes built on the depth-based study methods piloted with Para. Similar to Para, the Dynamic Brushes study aimed to understand the expressiveness of the system in an open-ended setting and its relevance to professional manual artists. The evaluation also examined the learnability of the system. Unlike Para, Dynamic Brushes featured an external programming representation. Furthermore

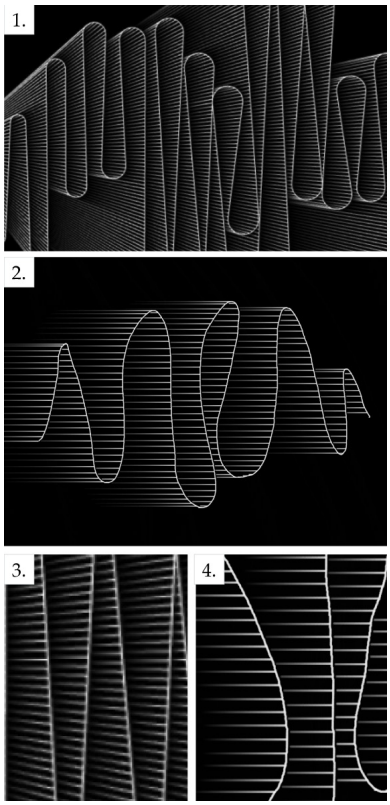

Figure 5.19: Comparison of work by Zach Lieberman in openFrameworks (1), to work from in Dynamic Brushes (2). 3. Closeup of Lieberman's work. 4. Closeup of Dynamic Brushes work.

while Para's procedural functionality built on the conventions of non-procedural vector graphics software, the Dynamic Brushes programming language introduced artists to entirely new ways of thinking about creating and manipulating digital brushes. Therefore, the evaluation of Dynamic Brushes looked at how artists learned and struggled with the system, and in doing so, provided insights about ways future systems might help artists learn representational tools. I developed a set of evaluation criteria based on my original design objectives:

Compatibility with manual practice: Are artists able to make procedural modifications while engaged in the process of drawing? Do the drawings produced by artists show evidence of both procedural and manual creation? Do manual artists view the system as relevant and useful to their practice?

Expressiveness: Are artists able to create a wide variety of different procedural behaviors with the system? Does artwork created by the system show evidence of procedural aesthetics and techniques found in work created by conventional procedural tools? Are artists able to produce drawings that are stylistically different and that reflect their personal style?

Learnability: Do the size and scope of procedural concepts, properties, and functions feel approachable for manual artists? Are artists able to engage in independent exploration and experimentation while using the system? Are artists able to articulate how the procedural components of the system are functioning? Does using the system assist artists in understanding general procedural concepts?

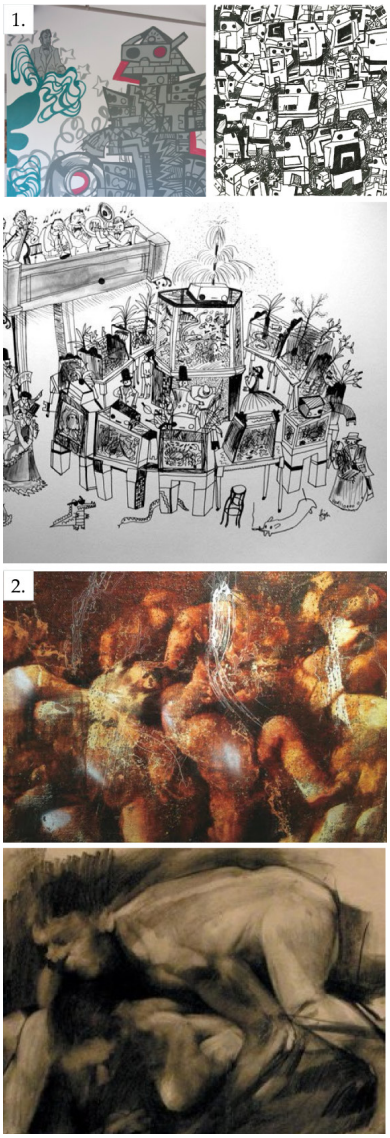

Figure 5.20: Prior work by artists in the study. 1. Illustrations by Fish McGill. 2. Paintings by Ben Tritt.

5.6.1 Study methodology

I commissioned two professional artists to use Dynamic Brushes for two weeks. Each artist was asked to produce a minimum of three pieces of completed artwork with the system during that time. In order to understand how well Dynamic Brushes supported different aesthetics and processes, I selected two manual artists with different art styles and approaches. Fish McGill was a professional illustrator and member of the faculty at a fine-arts college, where he taught illustration and graphic design. Fish primarily created illustrative artwork of animals, people, and other characters. He had extensive experience in physical and digital illustration and maintained a practice of daily sketching in a physical sketchbook. Much of Fish's subject matter emerged from his daily life with his wife and children. Fish had some prior experience in web programming, and

had collaborated with procedural artists to create work in Processing. He stressed the challenges of these collaborations in our initial conversations. Fish was extremely interested in finding ways to incorporate programming in to his drawing process.

The second artist, Ben Tritt, was a professional painter. Ben was trained in oil painting, and later worked with digital painting tools and physical paint. Ben's subject matter ranged from stylized portraits and landscapes to abstract compositions; however, his abstract work often emerged from referencing photos or objects in the real world. In our conversations, he emphasized the importance of having precise control over the quality of his brush and the way it laid down paint or pixels on the canvas. Prior to the study, Ben regularly drew with a digital stylus and enjoyed the freedom and flexibility offered by digital painting tools. While he saw great potential in digital tools to extend his practice, he desired greater control than he had over digital and physical painting tools. He had no prior experience with programming, but had an understanding of the basics of parametric design and felt it might offer a pathway for greater control over digital painting tools.

One week before the study, I met with each artist for two hours for an introductory interview. I also used this meeting to observe and record the artists' painting and drawing process with their preferred tools. This provided the opportunity to make initial adjustments to Dynamic Brushes to support Ben and Fish's preferred ways of working. Over the course of the study, I conducted six individual meetings with Fish and five with Ben. These meetings occurred at an interval of every 2-3 days. The first three meetings lasted 2 hours each, and were divided between training sessions in the Dynamic Brushes programming language and authoring interface, and periods where I observed the artists working with the system and answered their questions. The first training session covered the basics of the interface and creating single-state behaviors. The second training session covered multi-state behaviors and timers. The final training session introduced artists to the spawn action. In later meetings I reviewed the artwork each artist produced and discussed their experiences. Throughout the study, I used both artist's feedback to improve Dynamic Brushes' functionality. At the conclusion of the study, after meeting individually with both artists to discuss their pieces, I conducted one additional meeting in the form of a joint discussion where the artists shared their artwork with each other and compared
their experiences.

I collected data through recorded discussions, written surveys, and participant artwork. Surveys contained attitudinal questions relating to my evalu-

ation criteria, using 5-point Likert scales, with 5 as the optimal response. I did not use experience sampling because the Para study demonstrated how this approach could be disruptive to the creative process. Instead, I collected automated backups of in-progress participant artwork and code and used this information to inform subsequent discussions with the artists. Similar to the Para study, I allowed the artists to incorporate other digital tools in their work with Dynamic Brushes. In the results, I distinguish between portions of artwork produced with Dynamic Brushes and those produced by other means. Survey results, discussion transcripts, and artwork were analyzed with respect to my evaluation criteria. The majority of the results are qualitative, triangulated from open-ended survey responses and group or individual discussions.

5.7 Evaluation Results

The artists found the behavior model to be challenging to learn at first, but both were gradually able to use the system in an exploratory and experimental fashion. Ben produced six finished pieces, and Fish produced upwards of ten. Both artists also created animations with some of the imagery produced in Dynamic Brushes. Stylistically speaking, Ben focused on painterly portraits and Fish created repeating patterns and illustrations. The artists felt that Dynamic Brushes offered opportunities for greater flexibility with digital tools and extensions of their prior practice, and could potentially support collaborative practice and reflection on artistic process.

5.7.1 Learnability

Ben and Fish faced several challenges learning the authoring environment of Dynamic Brushes. Initially, both struggled with understanding specific concepts within the programming model. Fish quickly grasped the principles behind constraints and transitions but took longer to understand spawning and using multiple behaviors. He initially focused on trying to understand all aspects of the programming language; however after several meetings, he transitioned to exploratory use of the system, and decided to focus more on making artwork and less on trying to understand how each component of a behavior functioned. Fish described how working with examples facilitated this transition:

Before that, I was just thinking, "I need to have much more control over this. I need to understand things more." Whereas, I don't think that when I'm

in other pieces of software... In [Dynamic Brushes] I think so much of the experience comes from refining the brushes to do the thing that you want. So you do have to ramp up, you do need to learn a little bit more about how the back end of it works so you can have the type of marks that you wanna make ... After yesterday, I was like, "Alright, you know enough, and now you have some examples. Now, just try doing what you normally do with things, where you make some stuff and combine it with other things you already know how to do."

Fish described how approaching the tool with a perspective similar to how he approached non-procedural software made his experience using Dynamic Brushes more enjoyable overall. He also noted that the expectation to produce finished artwork with the system motivated him to focus more on outcomes than on building an understanding of the programming language as the study progressed.

Unlike Fish, who had worked with web programming before, Ben had no prior programming experience and had greater difficulty than Fish understanding some of the procedural concepts of Dynamic Brushes. In particular, he initially struggled with understanding the semantic difference between transitions and property mappings, and we devoted a significant time during our second meeting to talking about these concepts. Early on, Ben was frustrated, in part because he attributed his lack of understanding as a failure to grasp what he assumed were obvious concepts. Despite Ben's difficulties, he also experienced key moments of insight while using the tool. He described the experience of grasping the concept of property mappings:

"At first my brain didn't really understand being able to use force for multiple things ... what would that mean? It's very abstract. And then when I felt it, it's like "Oh, wow! The force could actually control two things or even three things." It could change the lightness AND the diameter."

Ben talked about how moments of insight like these made working with Dynamic Brushes particularly compelling and distinguished the software from other digital painting systems he used in the past:

When I'm using [Dynamic Brushes] and something new happens, I feel my brain light up. I feel like "Ah!" There's a little hint of something that is like the part of this that can be addictive. I wanna stretch those things out and find them.

The programming model and authoring environment, while supporting exploration and experimentation was not intuitive for either artist. Con-

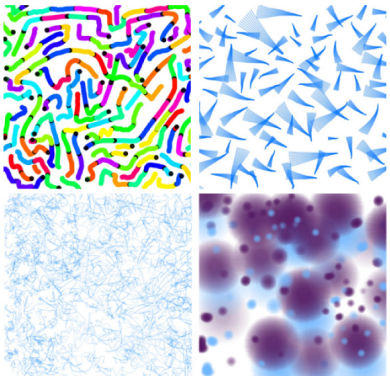

Figure 5.21: Patterns created by Fish McGill with Dynamic Brushes. Clockwise from top-left: color alternation using stylusMoveBy transition, fan pattern using spawn action and iterative rotation of spawned children, scribble pattern by modulating delta x and y with a sine generator, two-tone pattern produced with "stereo-brush"—two behaviors that draw simultaneously, one with an offset.

Figure 5.22: Extension of prior work with Dynamic Brushes. Top: hand-drawn composition notebook illustrations by Fish McGill, produced prior to the study. Bottom: Iteration of the series with background patterns produced in Dynamic Brushes.

versely, both artists described the drawing environment as familiar and easy to use. In particular, they appreciated its simplicity. Ben specifically commented on how it enabled him to focus on the task of painting:

One of the things that I really like about [the drawing interface] is the fact there's much less stuff . . . It's annoying when you have all these icons and all these things. You just want to sit in front of the canvas and not be confronted with that. When I'm painting, I'm just painting.

In addition to being simple and accessible, both artists felt that the familiarity of the drawing interface improved their comfort using the system as a whole and their confidence when exploring the behavior authoring interface. As Fish put it:

Definitely, [the drawing interface] is still the more comfortable area for me, and [the programming interface] less so, but I've become more interested in it the more I use it.

Overall, both artists relied more on starting with an example behavior and adjusting it to suit their objectives than creating behaviors from scratch; however, they also explored creating their own unique behaviors. In working with the authoring environment, the terminology of properties and transitions significantly affected their understanding. We discussed ways in which property functionality could be communicated visually, and I added animated tool-tips that visualized generator functionality (fig 5.6). These depictions helped convey functionality more immediately than a verbal description.

5.7.2 Expressiveness

Each artist produced multiple finished pieces over the course of the study in addition to numerous sketches and experiments. The pieces created by each artist reflected their styles and approaches prior to working with Dynamic Brushes. Fish started by experimenting with a variety of behaviors to generate textures and patterns (fig 5.21) and then began incorporating these patterns into illustrations and adding typography in Adobe Illustrator. In the process, he built on themes and subject matter from his prior work. For example, before the study, Fish had created a series of illustrated composition notebooks. He used Dynamic Brushes to iterate on this series by using a scribble-like brush behavior to create backings for a new set of notebook illustrations (fig 5.22). He also created an animated notebook illustration that progressively layered a series of patterns.

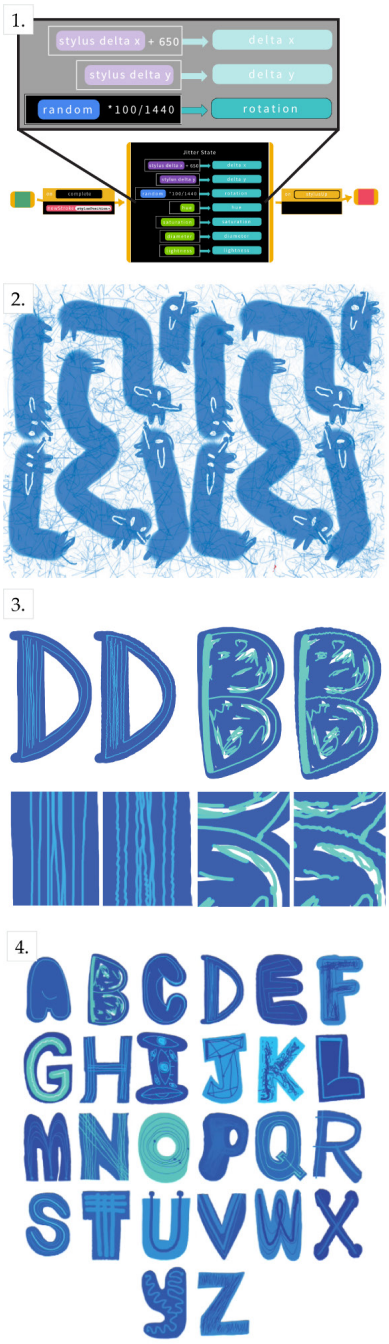

Figure 5.23: Repeating illustrations by Fish McGill. 1. Fish's behavior for the "fraternal twin brush". 2. Frame from animation produced with the behavior. 3. Sample from alphabet series designed with fraternal twin brush (Bottom shows a closeup of the variation in line quality between each copy.) 4. Entire alphabet.

Towards the end of the study, Fish focused on creating illustrations with a behavior set that created multiple marks offset from the position of the stylus. The behavior built from the spawning template and consisted of a parent brush that followed the position of the stylus, while spawning one or more child brushes that were progressively offset on the x axis. Fish first used this behavior to create repeating patterns. He later modified it to incorporate a random jitter in the line of the child brush (fig 5.23). The results were two simultaneously produced drawings with identical forms but different line qualities, one smooth and one ragged. Fish named this brush the "fraternal twin brush" and characterized it as a way to draw as one artist at two different periods in their life, comparing it to how the style of the American cartoonist Charles Schulz's style changed as he aged:

The other line looks like "young" Charles Schulz drew this one, and "old" Charles Schulz drew that one. That's what I was thinking about. What if I could draw as another [artist], but during two periods of their life.

Unlike Fish, who primarily focused on graphic illustrations, Ben applied Dynamic Brushes to the creation of painterly portraits (fig 5.24). This was consistent with his prior practice in abstracted oil portraiture. Ben was interested in using Dynamic Brushes to gain precise control of brush texture and appearance. Because the software did not provide procedural control over the bitmap brush texture, Ben focused initially on behaviors that tuned the relationship between pressure and brush diameter and alpha. Ben later requested that we use one of our discussions to work together to construct a brush that produced the kind of response he sought. We eventually built a behavior that used two states to create a brush that continued drawing for one second after the stylus was lifted from the canvas, with the drawing occurring at a proportional speed and angle based on where the stylus was before being lifted (fig 5.25). Using this brush, the artist could simulate something resembling "throwing the paint" by drawing with a flicking-motion. We were not able to tune the behavior to his exact specifications, yet Ben was excited about the potential to partially simulate physics of a real brush:

The idea of being able to [digitally] throw paints, it kind of opened my mind to a type of thinking that I haven't experienced anywhere else. If you could continue to refine that so you can have a lot more control over it with even the slight micro movements, that could be very, very exciting.

Overall, Ben and Fish focused at different levels of abstraction when creating brushes. Fish 's brushes structured the composition of his work by drawing multiple strokes in different geometries and styles across the entire canvas, whereas Ben focused on brushes that modified the quality of

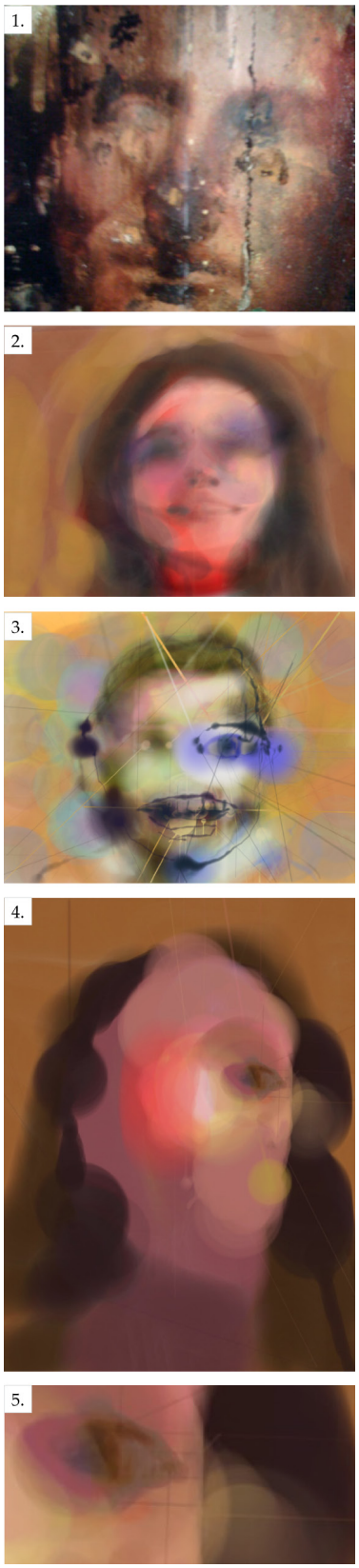

Figure 5.24: Portraiture by Ben Tritt. 1. Prior portrait created in oil. 2-4. Portraits drawn in Dynamic Brushes with the "continuation" behavior. 5. Closeup of the effects of the continuation brush.

the stroke itself. Fish 's approach was much more aligned with the original application objectives of Dynamic Brushes; therefore, he often had more success when creating and applying brushes than Ben. I made several revisions to Dynamic Brushes to support Ben 's approach, including modifying the response curve for alpha and diameter for fine-grain control, and modifications to the graphics back-end for improved fidelity when creating artwork with multiple semi-transparent layers. I also added additional sensor reference properties, including stylus speed and stylus heading to define the direction of the stylus vector in polar coordinates. The speed property was effective in achieving a degree of physical brush simulation, but much of Ben 's interest centered on variable control over the brush mark itself, including its texture and the hardness of the top and bottom edges. The Dynamic Brushes programming model could potentially support texture and hardness properties, but incorporating this degree of control into the system would have required restructuring the bitmap texture-mapping functionality of the system, something that was not feasible in the time frame of the study.

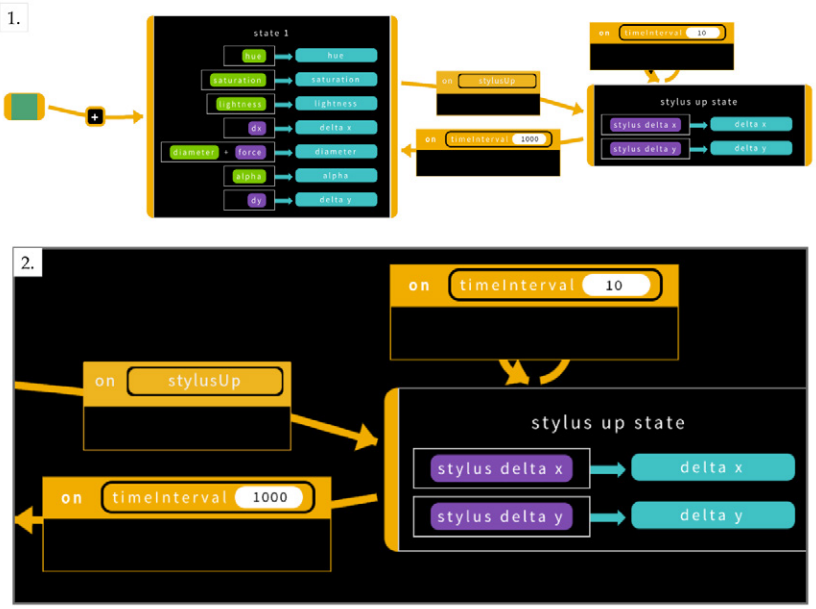

Figure 5.25: "Continuation" behavior used by Ben Tritt for portraits. 1. Full behavior. 2. Detail of transition to continuation state.

Although Dynamic Brushes enabled artists to create animated brush behaviors, the system was designed to produce static artwork. Despite this, Ben and Fish created animated works in the final week of the study. They accomplished this by creating multiple variations of a drawing in Dynamic Brushes and using additional software to compose these variations into an animation. Fish built on the idea of animating Dynamic Brushes drawings to create a moving alphabet. He used his fraternal twin brush to simul-

taneously create two variations of each letter and then animated them to flip back and forth between the variations (fig 5.23-3). He outlined his approach to animation as follows:

I noticed you can draw in stereo, I can draw things that are mirrored, I can draw things in parallel. In the past, I've done animation where you'll do, say, a title sequence, and you write whatever the title is on the screen, and then you trace over that, and then you trace over that again, and you give this illusion of animation. Whereas with this tool, I could do two to three at once and then do that process. And It's like, "Okay, well, that's something that I've done before," but this is much more suitable and enjoyable to draw with.

5.7.3 Compatibility with manual practice

Ben and Fish distilled their manual process down to a small number of simple techniques. Ben described how all paintings are built from "very simple things", combined in "very complex ways". He viewed all paintings as compositions of spots, edges, and transparency. Similarly, Fish described how every drawing he created was comprised of three line styles:

There's three separate lines that I usually make every type of drawing with. Everything is a zigzag, a wiggly line, or up and down. When I'm drawing anywhere, I always think of those lines as the tools I'm working with.

Ben and Fish 's general descriptions of painting and drawing resonate with their different approaches in Dynamic Brushes. Ben viewed painting as the recombination and manipulation of texture and transparency. Similarly in Dynamic Brushes, he worked at a textural level and focused on how Dynamic Brushes could be used to control the quality of the mark itself. Fish viewed drawings as compositions of different line styles. In Dynamic Brushes, therefore, he worked at a compositional level and created brushes that produced variations in line appearance and geometry. Many of the signals produced with the generators directly aligned with Fish 's illustrations of the line styles in his drawing. In line with this, Fish used generators extensively throughout the study and Ben did not. When asked about the similarity between generators and his line styles, Fish said it was a "fun coincidence" and described how comparing generators to the way he thought about drawing helped him understand the overall application of generators.

Ben and Fish had different reactions to the dual interface structure of Dynamic Brushes. Both artists saw value in having different interfaces for

artwork creation and programming; however, unlike Ben, Fish stated that he preferred both interfaces on the same device to enhance the portability of the system. Furthermore, while Ben was satisfied with primarily controlling all programming functionality in the authoring interface, Fish made it clear that he wanted more control over the functionality of the behavior while he was engaged in drawing, for example by toggling behaviors on and off and being able to modify some of the parameters or expressions of a brush without having to switch to a different interface. The addition of UI properties in the authoring interface and the behavior-control panel in the drawing interface addressed some of Fish 's concerns.

Fish used the authoring interface more than Ben during the study. However Ben 's preference for a separation between programming and drawing also connected to his belief in the different thought processes involved in tool creation and art creation. Ben viewed tool creation as intrinsic to the artistic process, and acknowledged that other creators might prefer to tweak tools while creating art, or even simultaneously engage in tool building and art making. For him, however, art creation began once he had familiarized himself with a tool to the degree that it became "an archetype in his brain". Following that point, he wanted to work exclusively in an intuitive, tacit, and non-verbal mindset:

When you're painting, the idea of having to do anything other than paint is horrible. As soon as I actually take my brush and start making marks, I can't talk. I guess, it's a process that requires connections in parts of the brain that are demanding.

Although Fish disliked working on two separate devices by himself, he found the two-device setup effective when collaborating with another person. Fish involved his three-year-old son in some of his experiments with Dynamic Brushes. In these scenarios, one person drew on the tablet, while the other simultaneously modified the behavior by changing the values of the expression or by dragging in new property mappings. From this experience, Fish talked about how he could see Dynamic Brushes being used as a collaborative tool in performance-based artwork, where one artist illustrated and the other programmed in real time.

Fish 's use of Dynamic Brushes to create animation also demonstrated an extension of his manual practice. He described how creating multiple images simultaneously in Dynamic Brushes built on prior approaches to animation and iteration:

I was borrowing from something I'd done in the past, but trying it in this tool because it was so customizable. I could make [the drawings] look a specific

way or an unpredictable way, and that was something I really enjoyed about this. I do like to work iteratively, too. That's not anything new.

5.7.4 Reflection

Both artists reflected on broader aspects of artwork creation and the implications of a system like Dynamic Brushes. Prior to the study, Ben and Fish believed that tool creation was an important component of artistic practice. While working with Dynamic Brushes, they reflected on how digital tools opened or restricted opportunities for tool creation. Fish described how working with Dynamic Brushes made him less satisfied with conventional digital tools:

I'm a little frustrated now when I open Photoshop. I'm like, "Wait a second. I can't make brushes like I was doing in there." I've become attached to [Dynamic Brushes]. It feels more like a sketchbook or a drawing experience than I've had in other tools because it's combining just enough of the good stuff with the more complex, robust stuff.

Furthermore, reflecting on the tool creation process led Fish to generate ideas for new tools or new ways the system might function. In this quote, Fish described how he arrived at the idea of behaviors that functioned as erasers rather than brushes:

Everything is just a variation of adding or subtracting a pixel. Everything is based off of that. And then I could see, well, all of these tools are just based on that one thing, but I'd never thought of that before. I had some distance from the program when I used this tool. And that got me on this kick of, "Wow! I wish the eraser tool could do some of the stuff that my brushes are doing."

Ben used his experience with applying Dynamic Brushes to animation to question the general application of all digital art tools. Specifically, he wondered if it was sufficient for digital tools to simulate physical media with greater flexibility, for example to reverse changes, add or modify layers, or tweak the image properties, or if it would be more meaningful to apply digital tools to the creation of artifacts that would not be possible with "analog" media:

I was thinking, what are the things about [Dynamic Brushes] that justify it, in and of itself, as a digital product? And animation is probably the most obvious, right? Just being able to observe all of those stages.

He went on to describe how the emphasis of Dynamic Brushes on creating brushes that reflected the artistic process seemed to naturally have implications for supporting general forms of artistic reflection. As a result he argued that Dynamic Brushes should have a built-in recording mechanism to directly facilitate reflection:

A recording process should be, I think, part of the medium . . . This is the most obvious affordance of this whole thing. You can have the work reflect the thinking process in a way that obviously, in analog, you never could.

Finally, Ben touched on the power of providing artists with options when working with tools, but also stressed the importance of imposing restrictions and constraints:

I think that the tools become a lens that a person sees the world through. Having too many options available is almost disturbing because then you have no way to narrow these things down, not enough is also not good.

Ben and Fish encountered challenges when learning the Dynamic Brushes programming language, but the system also enabled them to create in new ways, and build on their existing work. Most significantly, working in Dynamic Brushes impacted their general expectations and perceptions towards digital tools, and encouraged them to reflect on how procedural tools might reshape art practices in the future. The next chapter opens with an analysis of the results of the Dynamic Brushes study in comparison to the results of the Para study.

6

Discussion

This chapter discusses the implications of Para and Dynamic Brushes in the broader context of developing tools for combining manual and procedural creation. The chapter is divided into two sections. The first section compares the affordances, opportunities, and limitations of representational programming and dynamic direct manipulation as tools for manual art. The second section revisits the research questions posed in the introduction and outlines general design principles for supporting learning, expressive manual creation, and diverse perspectives in procedural art.

6.1 The Opportunities and Tradeoffs of Para and Dynamic Brushes

Dynamic Brushes and Para were developed and evaluated with similar methods, yet they enable different ways of working and support different outcomes. These differences are in large part due to the fact that Para integrates procedural manipulation with concrete artwork, whereas Dynamic Brushes uses an external programming representation. A comparison of both systems provides a comparison of the trade-offs of these approaches to manual art.

6.1.1 Matching procedural models to manual practice

Para and Dynamic Brushes have two different procedural models. Para's model is entirely declarative and structured around constraints, lists, and declarative duplication. Dynamic Brushes uses a combination of declarative and imperative structures, combining constraints, states, and

event-driven transitions. Two questions are important to address when examining these procedural models as mechanisms for artistic production. First, how expressive is each model when applied to procedural art creation? Second, to what degree does each model align with the objectives of manual artists?

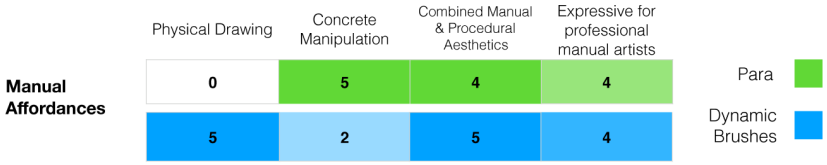

Figure 6.1: Comparison of the manual affordances of Para (green) and Dynamic Brushes (blue), drawn from analysis of artist feedback, artwork, and surveys. The darker the color, the greater each system supports a given practice. Numbers are assigned for clarity but do not indicate numerical ratings by artists.

How expressive are Para and Dynamic Brushes' models when applied to art creation?

The models used in Para and Dynamic Brushes were developed for different applications and direct comparisons can not be made between all aspects of the systems. They are similar, however, with respect to their use of constraints to control stylistic and geometric properties of artwork. Para 's constraints enable dynamic updates and iterative variation among large collections of artwork, based on a range of pre-defined distribution functions. In its current form, Para does not allow for the creation of arbitrary constraint functions, but this is more a limitation of the direct-manipulation interface than the model itself. Para 's model enables constraints between different properties, for example, mapping the rotation of one shape to the hue of another, and also supports chaining multiple constraints in sequence; however, these properties are more applicable to supporting flexible and systematic updates than to generating different artistic effects or aesthetics. Another limitation of Para 's model lies in its lack of conditionals; an artist cannot specify a constraint that negatively affects objects in some cases but not others.

The Dynamic Brushes model allows for constraints between geometric and stylistic brush properties and sensor data, user interface elements, function generators, other brush properties, or some combination of these. In addition, it allows for collections of constraints, or states, to be rendered active and inactive based on discrete geometric or time-based events. Although it does not support arbitrary conditionals, the Dynamic Brushes model does enable conditional application of constraints through transition events.

The computational expressiveness enabled by conditional events in Dynamic Brushes supported several concrete artistic applications. As demonstrated in the sample applications, they supported control of the rate of change of a generator to produce different variations in color and form and

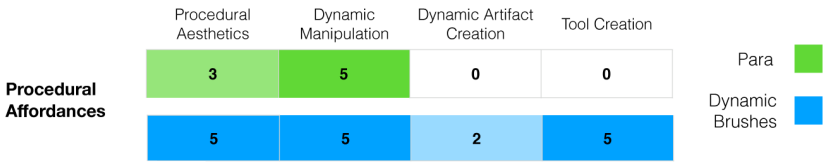

Figure 6.2: Comparison of the procedural affordances of Para (green) and Dynamic Brushes (blue), drawn from analysis of artist feedback, artwork, and surveys. The darker the color, the greater each system supports a given practice.

also supported autonomous strokes that terminated after a certain distance or intersection event. Conditional events also had applications in the artist study. For example, Ben used them in his continuation brush behavior to change the brush parameters after the stylus was lifted. Furthermore, in working with Para, several opportunities arose in which conditionals would have been useful. These included conditional addition or removal of objects in a duplicator based on an evaluation of their properties, and conditional application of a constraint to a list of objects based on the geometry or style of the objects themselves.

The state-machine structure of Dynamic Brushes also supported a wider class of procedural art forms and functions compared to Para. Although it lacked many of the control structures of a textual language like Processing or openFrameworks, Dynamic Brushes still enabled the creation of interactive behaviors that resembled those created with these textual tools. It also enabled the creation of self-similar forms and recursive behaviors, something that is not currently possible in Para. However, as Schachman demonstrates, recursion is not necessarily dependent on an external programming representation (2012). Overall, Dynamic Brushes could be considered an alternative model for creating procedural art similar to that produced through advanced use of textual tools, whereas Para offers a way to extend the expressiveness and flexibility of non-procedural software for digital graphic art.

One limitation of the Dynamic Brushes model that emerged during its evaluation was its lack of variables. Variables would have provided a convenient method to systematically replicate and update values across different property mappings. Moreover, in the case of generative behaviors, variables would have enabled the same randomly generated values to be accessed in different mappings, and event or method arguments. Variables do not conflict with the Dynamic Brushes state machine model; however, effective use of variables requires an understanding of additional computational concepts, including data storage and symbolic naming. Therefore, addition of variables to the Dynamic Brushes language would have to be weighed against the additional learning challenges they would create.

To what degree do the models of Para and Dynamic Brushes align objectives of manual artists?
The background interviews and evaluation discussions established that manual artists, while deeply invested in concrete practices, also think about their processes in systematic and formal ways. This suggests that the concept of programming as a mechanism for formal representation does not inherently conflict with manual art creation. Para and Dynamic Brushes provide an opportunity to examine how two specific programming models, declarative constraints and state machines, may align or conflict with both the mindsets and the objectives of manual artists.

At a basic level, Para 's constraints were more intuitive to manual artists than Dynamic Brushes ' state machine. Artists described Para as familiar, intuitive, and reminiscent of traditional tools and media. Conversely, artists considered the Dynamic Brushes Drawing interface to be familiar but required time to learn the authoring environment and programming model. At this point, it 's unclear if the intuitive quality of Para in comparison with Dynamic Brushes demonstrated its alignment with the mindsets of manual artists themselves or was a reflection of the intuitive nature of declarative constraints in general, especially when compared to state machines.

Some aesthetic relationships were much easier for manual artists to express in Para than in Dynamic Brushes. The most significant of these was the ability to describe a range. In both studies, situations arose where artists wanted to create a constraint that targeted a specific range of values, for example, mapping hue exclusively to shades of blue or constraining rotation between 0 and 45 degrees. In Dynamic Brushes, artists struggled to do this for two reasons. First, the model contained no built-in mapping function and defining a range of values in a property mapping required the artists to write a relatively complex mathematical expression. Second, because the range of possible values in a mapping was expressed numerically rather than visually, generating the desired range was often unintuitive. For example, one artist repeatedly experimented with a brush design before discovering the correct expression that would constrain hue values to shades of blue. Conversely, Para 's model enabled the visual expression of ranges by default. To produce a desired range, all an artist had to do was create a list-based constraint or duplicator and then manually set the property of the reference objects to the desired minimum and maximum values. These values could be set through direct manipulation transformation tools or UI components, and in the process the artist received immediate visual feedback on the results of interpolating across the selected range.

It is also important to examine Dynamic Brushes ' and Para 's models with respect to the types of manual art they were designed around— bitmap drawing and vector graphics. As mathematically defined forms, vector graphics have the benefit of unambiguous high-level geometric and stylistic properties, which are well suited to procedural representation and manipulation. Bitmap images also have discrete properties, but these are most commonly represented as collections of pixels or distributions of numerical data, like a histogram. Procedural manipulation of bitmaps generally requires artists to deal with lower-level representations than vectors; therefore, vector graphics present a lower bar than bitmap drawing for providing artists with access to straight-forward, intuitive forms of procedural manipulation. That said, vector graphics can conflict with manual drawing. Bitmap drawing environments often are compatible with or specifically designed to support manual drawing with a stylus, and therefore, more closely reflect physical painting and sketching. Vector editing systems, while highly expressive for many forms of art and design, often emphasize drawing regular shapes or creating Bezier curves. This reduces expressiveness when it comes to gestural forms of drawing. The limitations of vector graphics were reflected both in the evaluations of Para and subsequent interviews with manual artists who felt restricted by the regular forms and clean lines. This feedback was one of the primary factors motivating the development of Dynamic Brushes around stylus-based bitmap drawing. The Dynamic Brushes model, while more complex and less intuitive than Para, was designed to extend drawing with a stylus, and therefore, did a better job of preserving the variation and stylistic diversity of traditional forms of manual drawing.

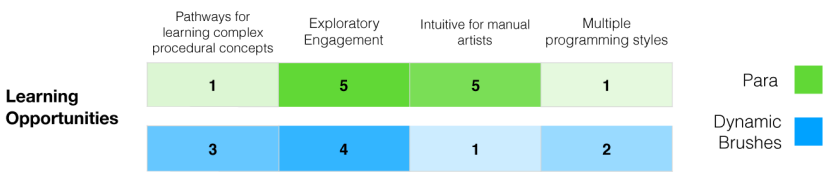

Figure 6.3: Comparison of the learning affordances of Para (green) and Dynamic Brushes (blue), drawn from analysis of artist feedback, artwork, and surveys. The darker the color, the greater each system supports a given practice.

Para and Dynamic Brushes suggest several different pathways for matching procedural models to manual art. There are situations in which artists are either disinterested or unable to invest effort to learn programming. In such cases, systems that contain a small set of procedural constructs that are compatible with a specific digital art domain can provide people with access to many of the opportunities of procedural art without requiring them to learn an entirely new model. The trade-off is that this approach will have a lower ceiling in terms of computational expressiveness than tools with representational procedural descriptions and will likely provide artists with less procedural control over some of the finer points of manual

expression. Alternatively, if the goal is to support intuitive entry and still provide opportunities for relatively complex procedural control, it may be productive to develop systems that augment procedural relationships that can be defined exclusively through direct manipulation along with concepts that benefit from some external representation—for example complex mathematical expressions, variables, or imperative sequences. Such tools will require more learning on the part of the artist, but they also may provide opportunities to scaffold that learning by enabling artists to experiment with complex procedural structures through direct manipulation. This idea is in line with Kay 's theory that direct manipulation might aid in the learning process of representational systems by helping people to develop mental maps of abstract concepts (1990). Para suggests a starting point for both of these pathways.

A third pathway, suggested by Dynamic Brushes, is to design a model that takes advantage of complex procedural concepts and structures to enable expressive procedural manipulation of the core properties of a given manual practice. In the case of Dynamic Brushes, this involved developing a model that enabled artists to control line style and geometry in response to gestural drawing data and discrete events over time and space. A variant of Dynamic Brushes that targeted painting might focus instead on controlling brush texture and hardness in response to stylus physics. This pathway has the advantage of providing artists with a wide range of computational expressiveness and a diversity of outcomes, while remaining relevant to the aesthetics of the original manual practice. The tradeoff is that more complex procedural structures will require additional learning support. It is also unlikely that this approach will produce models that reflect the way manual artists already think, because the process of manipulating procedural representations of line, form, and color is fundamentally different from physical control. Yet it is important to point out that programming and manual art are different mediums, and more importantly, artists are individuals who vary widely in how they think about their practice. The goal of building a universal procedural model that is highly intuitive and highly expressive for all manual artists ignores this reality. Some forms of computational expressiveness will inevitably require that artists learn new models and mindsets. Given adequate support, learning new mindsets can also offer creative opportunities in itself.

Overall, procedural systems for manual artists that aim to provide advanced computational capacity should emphasize selecting procedural functionality that is compatible with aesthetics and practices of a specific domain of manual creation. It also should focus on scaffolding the learning process. I discuss ways of doing this in the following subsection. In addition, if sufficiently expressive, such tools may offer artists the opportunity

to develop new procedural structures and systems that directly reflect their personal practices and ways of thinking. This is one of the most powerful creative opportunities offered by programming.

6.1.2 Integrating artwork and representation

The procedural models for Para and Dynamic Brushes were influenced by the interfaces through which they are expressed. The design of each model is based on core interface and interaction goals set out at the start of each project. With Para, the goal was to enable artists to create procedural art exclusively through the direct manipulation of artwork. As a result, Para contains no external procedural representation. The procedural functionality in Para is restricted to concepts that can be visualized directly on-top of vector artwork and then manipulated through established direct-manipulation tools and interface components. For Dynamic Brushes, the original goal was to create a unified system for artists to both create and apply drawing tools. Because tools represent a mechanism to produce artwork, rather than relationships between artwork, the system was designed with an interface for tool creation that was separate from the artwork itself. As a result, Dynamic Brushes contains a procedural representation that is separate from the artwork. The decision to move to an external representation in Dynamic Brushes following the development of Para is not considered evidence of the expressive limits of procedural direct manipulation in comparison to external representations. Like their respective procedural models, the interfaces and representational structure of Para and Dynamic Brushes is a reflection of each system 's original goals and the envisioned needs of artists using each system. Rather than advocate for the superiority of either approach, comparison of the two systems provides an opportunity to examine tradeoffs in separating or integrating procedural and manual interfaces in terms of learning, creation, organization, and reflection.

As a starting point, it is useful to unpack the different design approaches of each system with regards to creating simple interactions and interfaces. The lack of an external representation in Para reduced the overall complexity of the interface. Compared to Dynamic Brushes, this reduced the amount of initial learning to operate the system. However, moving the vast majority of Dynamic Brushes ' procedural functionality to a separate interface also allowed for simplification of the Dynamic Brushes drawing interface. The benefits of this simplicity were remarked upon by the artists multiple times during the Dynamic Brushes study. Furthermore, there is evidence that Para 's integrated interface became challenging when work-

ing with large numbers of lists and constraints. Although Para 's panel for document structure contained a mechanism to visualize constraint structure, it often was insufficient when artists used the system for complex compositions. By comparison, Dynamic Brushes ' flowchart-like representation of procedural functionality was capable of representing both simple expressions and complex behaviors.

Organizational challenges are not unique to procedural art tools. Non-procedural digital tools also can make it challenging for artists to manage complex artwork, parse document structure, and make relevant selections (Perteneder et al., 2015; Gogia, 2016). Furthermore prior work suggests that borrowing computational organizational structures and leveraging external representations can alleviate some of the challenges involved in managing and selecting complex digital artwork (Xia et al., 2016, 2017). Tools like Para that emphasize direct manipulation of procedural relationships may still benefit from an external representation for organizing complex artwork.

Another side effect of integrating artwork and representation is that the process expands the roles that the artwork itself can play. In Para, vector objects could constitute *artwork* for example, components of the finished composition; *control mechanisms*, for example, a means of specifying the value for a constraint on other objects; or occupy *both roles simultaneously*. The idea of artwork as interface suggests the opportunity for artists to develop their own interface components as they create procedural compositions. In practice, however, artists I worked with did not take advantage of this opportunity. In studies with Para, I never observed people creating vector objects to exclusively act as control elements even though this technique was demonstrated in training sessions. Furthermore, one artist actually found the idea of using arbitrary objects to control the appearance of separate collections of artwork confusing. She was able to use lists most effectively when I restructured them to support internal constraints.

Dynamic Brushes had fewer opportunities for artists to create and customize interface components. Artists could customize how the values of built-in interface elements, for example, the color picker and the sliders, affected brush behavior, but it did not enable artists to manipulate behaviors by manipulating artwork. It was apparent that the artists valued the ability to control behaviors through built in UI elements and considered it useful to provide them with additional flexibility in the layout and structure of these elements. Yet there was no indication that artists felt that the structure of the artwork itself should dictate the behavior of the brush. It is possible that enabling artists to use artwork as an interface may

over-complicate the already challenging task of integrating procedural concepts into manual artwork. It may be most useful to provide flexible control over UI elements like sliders, knobs, and graphs, that allow artists to "physically" and visually tweak procedural properties. Hartmann et al.'s Juxtapose system provides an example of how this approach is effective in the domain of interface and interaction development (2008). Alternatively, artwork as interface may be a more relevant or obvious concept when creating interactive artifacts. A good example of this is the Apparatus system, which is aimed at creating interactive diagrams (Schachman, 2015).

Whether textual or graphical, code can simultaneously serve as a functional object and as a record of ideas and process. The ability to review and reflect on this record can provide powerful opportunities for learning and insight, both for the people who created the code, and for people who read code created by others. Although the artists who worked with Para described how the tool enabled new forms of exploration and expression, there was no evidence that Para provided them with a broader understanding of programming or digital functionality. Yet both Ben and Fish articulated specific insights that emerged from working with Dynamic Brushes, including Ben's description of how all procedural mappings were arbitrary, and Fish's realization that all behaviors essentially boiled down to different mechanisms for adding pixels to the canvas. These moments of reflection demonstrate the potential of programming to generate new ideas, and to encourage new ways of seeing.

The potential of external representations to encourage learning leads to a significant limitation of the Dynamic Brushes interface. In both learning and creating, effective programming environments must provide feedback to the programmer about how the program is actually operating. As Victor puts it: "if a programmer cannot see what a program is doing, she can 't understand it." Victor (2012b). In its current iteration, Dynamic Brushes presents creators with the challenge of both designing brushes and creating art with them, yet it provides little information about how a brush is actually operating while in use. The comparison of the learning curve of Para to Dynamic Brushes makes the depth of this challenge very apparent and suggests that procedural tools that incorporate an external representation must also contain structures for visualizing and communicating program functionality.

Fortunately, observing artists work with Dynamic Brushes provided insight into what this feedback structure might look like. **First**, the system should enable inspection of brush properties. On the most basic level, this would involve numerical displays of the current property values of a brush when it is in operation. However, a better approach would be to display these

values graphically, in a manner similar to generator visualizations. **Second**, the system should indicate the current state of an operating brush, both in the authoring interface and the drawing interface. This could be as simple as highlighting the current state block in the authoring interface and displaying the name of the current state at the top of the canvas in the drawing interface. Because some state transitions occur instantaneously, the system should also indicate the previous state and potentially possible future states. **Third**, because behaviors of Dynamic Brushes are time-based, the system should provide artists with the ability to visualize brush functionality as it changes over time. To achieve this, the system could enable the artist to record their use of a brush, and then enable them to view this recording step-by-step in conjunction with a visualization of changes in state and property values in the behavior. The ability to record and simulate execution is also important because it is often difficult to simultaneously operate and debug a system in real time. Overall providing feedback on the functionality of a tool or behavior is essential, not just in supporting learning, but in furthering the creative objectives of the artists who use the system.

6.1.3 Audiences and outcomes

Building effective systems is dependent on a good understanding of the intended audience and a clear envisioning of the expected outcomes of the system. Broadly, Para and Dynamic Brushes were developed to support the practices of manual artists and the creation of artwork that combined procedural and manual imagery and aesthetics. These objectives speak to a wide range of potential audiences and outcomes. This section compares the original objectives of each system, with respect to the kinds of people who were actually invested in using the system within the studies, and the types of artifacts they created.

Audience

Para 's targeted audience was illustrators, graphic artists, and other 2D visual artists. The system was intended to support the production of 2D vector artwork. Dynamic Brushes was intended to be used by people who create art by drawing with a pen, pencil, paintbrush, or stylus, and it was expected that people would produce static bitmap drawings with the system. Study results demonstrate how both systems were appealing to people who were representative of the intended audience. It is important to recognize that all artists in this research had prior interest, if not expertise, in procedural art and programming. This research cannot speak to Para 's or Dynamic Brushes' appeal to people with no prior interest in

procedural creation; however, this was never an objective. The studies do demonstrate specific procedural applications that were appealing to different kinds of manual artists, for example, color-blending experimentation and physical paintbrush simulation for painters, and compositional repetition and variation for illustrators. These applications provide insight about future pathways for developing procedural tools for specific manual domains. In addition, they suggest ways of communicating the affordances and benefits of procedural tools to manual artists who are unfamiliar with procedural art altogether.

Another way of describing the intended audience of Para and Dynamic Brushes is: "manual artists who are interested in procedural art, but deterred by the challenges of learning programming". Some might question if this audience even exists. If procedural art is "art created through programming", why would it be of interest to artists who are not interested in learning programming to begin with?

This question presents a fallacy, because assumes that the reason manual artists avoid learning programming is because they are uninterested in its applications. In reality, artists may avoid learning programming for many reasons, including restrictions that existing programming systems pose on established practices, cultural and social barriers in traditional computer science education, and the perception that some people lack the intellectual capacity to learn programming. The question also is only valid if programming is framed exclusively as the process of writing a series of instructions and hitting compile— a process that is directly in conflict with manual creation. Yet this description of programming, while accurate for some aspects of the practice, does nothing to convey programming 's creative or intellectual opportunities. Programming also is a mechanism to organize and manipulate complex systems, a means to construct dynamic and generative artifacts, and a tool to support procedural and abstract forms of thinking. Interviews with artists and depth-based studies demonstrate some overlap among the creative, intellectual, and reflective properties of programming and the perspectives and objectives of manual artists. The appeal of the alternative procedural structures of Para and Dynamic Brushes for artists who previously struggled with programming reaffirms the capacity of computation to support a broad range of mindsets, processes, and values. Furthermore, it demonstrates the substantial creative opportunities of developing procedural systems for people who don't fit the stereotypical depiction of a programmer.

Outcomes

Para and Dynamic Brushes were both designed to create 2D static artwork. This objective was designed to reflect established forms of art in both

manual and procedural domains, such as pen and ink drawings, digital illustrations, and procedurally generated prints. Each system also was designed so that artwork that was created using it could be incorporated into works created with other tools. Para enabled illustrations to be exported as SVG files and imported into other vector editing tools like Illustrator and Inkscape. Likewise, Dynamic Brushes enabled drawings to be exported as high-resolution transparent PNG images and imported into bitmap tools like Photoshop or GIMP. Across the studies, artists successfully used both approaches and created work exclusively in Para or Dynamic Brushes or created work using the tools to generate assets that were further transformed in third-party software. This demonstrates how Para and Dynamic Brushes can function both as self-contained 2D art creation tools and as components of a multi-tool workflow.

There also were applications of Para and Dynamic Brushes that emerged beyond their original design objectives. Both artists in the Dynamic Brushes study used the tool to create animations, and one artist used it to create assets for an interactive web-page. Although none of the Para artists used the system to produce animations, the systems ' ability to fluidly control change in multiple objects by manipulating a single element of a composition provided a strong suggestion of animation-based applications. The fact that both Para and Dynamic Brushes encourage the creation of motion-based artifacts is not incidental. Rather, it speaks to the potential of procedural tools to encourage liveness, movement and responsiveness even when these outcomes are not directly supported by a system. Future work could examine how dynamic tool creation and dynamic direct manipulation might broaden the practice and diversity of participation in performative, animated, and interactive forms of procedural art.

6.2 Recommendations for Broadening Practice and Participation in Procedural Art

Dynamic Brushes and Para were developed to explore three research questions. This section revisits each of these questions and outlines design principles and recommendations for supporting manual artists in expressive, accessible, and learnable forms of procedural art creation.

6.2.1 How can we create procedural art systems that are engaging, accessible and expressive for people with experience in manual art?

Develop procedural systems that incorporate interface and interaction paradigms from manual tools and practices.
Rather than introduce manual artists to entirely unfamiliar interfaces, Para and Dynamic Brushes extended interface paradigms from manual tools. This approach demonstrated several important benefits. **First**, it aligned procedural functionality with prior skills and practices of manual artists, enabling them to leverage established skills and working styles. **Second**, introducing manual artists to procedural creation through familiar interfaces can encourage exploration into unfamiliar functionality. This was reflected by Fish 's description of how the familiarity of the Dynamic Brushes drawing interface instilled confidence when transitioning to the authoring environment. **Third**, using established manual art tools as a foundation can guide system designers in selecting procedural models and interaction structures that are compatible with specific manual practices, as demonstrated by the design methodologies of Para and Dynamic Brushes. This approach also has the benefit of making it easy to integrate other components from manual tools in response to the needs of the artist, such as when opacity control and blend mode was rapidly integrated into Para. However, designers must also be careful in choosing to extend digital interfaces and interactions. Some digital art tools, like vector editing, can be restrictive for certain types of manual art. Designers should take care to extend digital interfaces that preserve the kinds of manual expressiveness important to their target audience.

Iterate on system functionality with the goal of supporting specific artistic applications.
When attempting to develop expressive procedural tools, designers can optimize for several different kinds of expressiveness. These include computational power, the ability to produce a wide variety of artifacts and outcomes, and the ability to produce diverse works in style and appearance. As Para and Dynamic Brushes indicate, traditional computational expressiveness is meaningless for manual artists if not directly linked to the ability to create compelling artwork. Therefore, system designers should evaluate and refine the procedural functionality of a tool by examining the kinds of artifacts and aesthetics it supports.

The process of refining a procedural tool can be supported through sample applications. Silver 's (2014) concept of a sample project space is particularly relevant here. Examples that are too similar will result in a system that supports a limited range of outcomes, whereas examples that are visually

impressive and extremely complex to create will produce a system that is challenging to learn. Instead it is important to inform the design process through a variety of different examples that demonstrate simple and complex approaches to creating different styles and outcomes.

Recognize the importance of depth-based evaluation.
In the process of developing two novel software systems, this research demonstrated an evaluation methodology where professional artists provide feedback on a system and contribute to its development by making finished artwork. This depth-based approach presents many challenges. It requires developing systems that are robust enough to support extended professional practice, and it also involves additional engineering work during the study itself. It also requires the participation of artists who work in non-procedural domains and are willing to develop artwork with an experimental tool and document and discuss their experiences while doing so. Despite these challenges, this research demonstrates how depth-based engagement provides unique opportunities and produces forms of knowledge not offered by other forms of evaluation.

In research where the objective is to understand the learnability and the expressiveness of a tool, it is essential to observe people in different stages of learning and practice. The extended study period enabled artists to spend sufficient time learning the systems and master many aspects of the tool. The effects of these extended learning periods were made apparent through the major improvements in the sophistication of artwork created in the second week of the depth-based studies compared to work produced in the first week. The work from the second week of each study also more closely resembled each artists' personal style than work from the first week. Similarly the feedback artists provided about the relevance and challenges of each system reflected different levels of learning. Artists began by focusing on challenges in the interface and struggles with procedural concepts, and gradually transitioned to questions about the broader artistic implications of the system, and reflections on the relationship between programming and art production.

Depth-based studies also reveal limitations of a system that only emerge through extended use. The Para study demonstrated how the system required alternative organizational tools once the artist's artwork reached a certain level of complexity. A short-term study may highlight initial learning challenges but will not demonstrate how tools enable artists to develop expertise, nor demonstrate the kinds of opportunities and limitations that emerge when practitioners have developed a degree of expertise.

Another advantage of the depth-based approach is that it can reveal new

creative opportunities that were not originally considered by the system designers. This was most clearly demonstrated in the animation and motion-graphic work that emerged at the end of the Dynamic Brushes studies. Furthermore, a focus on depth of engagement rather than breadth, enables researchers and developers to rapidly iterate on the design of a system in response to the objectives and needs of the people using the tool. This process improves alignment of tools with specific art applications and adds to the satisfaction artists can experience when collaborating with programmers.

Depth-based evaluations result in a multi-dimensional understanding of how a tool performs during extended use. Evaluating how people learn, apply, and reflect on a creative tool over time is an essential component of developing effective, engaging, and expressive tools for application in communities of practice.

6.2.2 How can procedural art systems contribute to the process of learning and understanding representational tools?

As chapters two and three demonstrate, applying programming to art creation is challenging. This difficulty can be described as two distinct problems. The first is that existing languages and systems provide abstractions that are not effective or expressive for a given domain of manual art. Strategies for designing effective abstractions for manual art are addressed in section 6.1.1. The second problem is that programming itself is difficult to learn. Manual artists, like many people, lack experience with core computational concepts and ways of thinking. In part, this is because programming is not taught as core subject in most primary education in the way that writing, math, and science, and to a lesser extent—art, are taught. Not all artists need to learn programming, but many are interested in incorporating procedural techniques in their work. Strategies for designing effective procedural art tools for artists without prior programming experience should therefore include support for the critical processes of learning and understanding aspects of programming.

Enable complexity and diversity through recombination of a select number of expressive building blocks.
Complex programming syntax conventions and lengthy APIs are particularly problematic for some artists because of the steep learning curve they produce. When developing computational tools for young people, Resnick and Silverman avoid the threshold issue altogether by arguing that “a little bit of programming goes a long way” and focusing on tools

that provide children with a small set of basic computational elements that support a wide variety of applications (2005). In many ways Para and Dynamic Brushes reflect a similar approach. Each system is built around small number of procedural concepts that enable complexity and diverse outcomes. In the case of Para, this approach reduced the learning threshold in comparison to text-based procedural tools, while still supporting meaningful creative outcomes for professionals. In Dynamic Brushes, the focus on property mappings, states, and transitions enabled both simple and complex behaviors to be created from the same basic building blocks. This meant that artists were not required to learn numerous additional computational concepts in order to explore increasingly complex behaviors.

Simple computational building blocks can offer a great deal of expressive power, yet selecting the correct building blocks is a complex process. The previous sections describe strategies for developing meaningful and approachable computational models for manual artists. These strategies are most effective when applied through a process of iterative development and evaluation. An inevitable part of the iterative design process is that it often involves trying out things that end up not working. Prior versions of Para and Dynamic Brushes contain procedural functionality that proved to be ineffective when actually tested with people, and was subsequently discarded. Within the broader context of research involving human-computer interaction, discarding functionality can be anathema when accompanied by the pressure to publish. Despite this, nothing reveals the learning challenges of a tool more quickly and clearly than actually observing people interacting with it.

Introduce tools alongside other structures that support learning
Tools, valuable as they are, are only one component to support learning. People learn in in different ways (Turkle and Papert, 1992), and effective learning environments should provide a variety of resources and opportunities for engagement, instruction, and participation.Templates and examples for representational programming can enable artists to quickly begin creating projects and explore the applications of a tool without having to understand all of the tool 's functionality. It is also important to identify the obscure computational concepts in a system and to structure instruction sessions that introduce these concepts one at a time and let people experiment in the presence of a facilitator. Introducing too many aspects of a system at once can result in people becoming avoid becoming overwhelmed and demotivated. As a side note, when working to identify non-obvious computational concepts, it is helpful to consult with people who are inexperienced programmers. State machines can seem like an obvious concept to software engineers, but are often foreign to others.

It's also important to provide opportunities for learners to reflect and discuss their experience with other people. Incidentally, the regular interview sessions with artists over the course of the study provided a space for this reflection. At one point, Fish described how the conversations played an important role in his learning process:

It was like meta reflection. You would point out things that I wasn't necessarily seeing.

Fish later compared our conversations to his experience of creating work in a community of artists. He described how the ability to show work to others and receive feedback yielded insights or revealed new aspects of a piece that would not have occurred working in isolation. Learning via dialog and reflection was a positive byproduct of the Dynamic Brushes evaluation methodology. To better support artists' learning in the future, successive iterations of this methodology should seek to actively support this process.

In considering other ways to support procedural learning, it also is important to reflect on how the progress of successful learning is evaluated. Neither artist in the Dynamic Brushes study emerged with complete understanding of the system. This was not considered a failure. Because learning abstraction takes time (Kay, 1993) it was assumed at the beginning of the study that artists would not learn all procedural aspects of Dynamic Brushes. More importantly, the study focused on other measures to indicate successful learning. Observing people applying previously unfamiliar procedural concepts to personally meaningful projects or expressing enthusiasm for continuing to work with a procedural tool are important measures of a positive computational learning experience. As Brennan and Resnick demonstrate, compelling projects, intrinsic motivation, and an understanding of the relevance of programming to personal goals can often be better indicators of learning and engagement than evidence of computational thinking alone (2012).

6.2.3 How can we support different ways of thinking and creating with representational mediums?

The overarching focus of this research was to broaden practice and participation in procedural art by finding ways to support manual artists in procedural creation. Efforts to diversify existing communities of practice offer equivalent benefits for established practitioners and new participants. New practitioners in a community often enable a community as a

whole to broaden perspectives and attitudes, and can lead to new types of artifacts. Diversification is particularly valuable for artists and programmers because they are groups of people who focus on trying new things and pushing the limits of existing structures. This final discussion section describes approaches for fostering a diversity of ways of thinking and creating with representational tools and mediums.

Foster ecosystems of tools

In light of the challenges of learning multiple tools or programming languages and difficulties working with multiple tools, it can be tempting to try to design "one (computational) ring to rule them all"; in other words, a system that enables expressive computational creation across a broad range of domains and applications. This approach is theoretically appealing, but impractical and uninteresting when applied to real-world creation. Computation as a whole is continually evolving and changing. More importantly, *people* are continually changing, and different creators expect different affordances from their tools. The rate at which new creative technologies are being developed and the diversity of individual artistic practices suggests that one master model of interaction or one universal programming language is unlikely to scale for new people and new technologies.

An better approach is to work to foster ecosystems of tools. This requires developing tools that accept, and even embrace the fact that there are trade-offs between different interface paradigms, interaction structures and programming models. Doing so can enable the development of tools that allow people to engage with computational creation at different levels of investment, with different forms of complexity, and in ways that embody different values. Domain-specific systems can enable specific forms of expression which would be diluted or obscured by general purpose tools. Therefore, rather than attempting to optimize tools across numerous unrelated domains and practices, designers should develop tools with respect to domain knowledge in a select number of interconnected fields.

The term ecosystem implies more than a diversity of computational tools. It also involves developing methods to foster relationships between different tools. The design of new tools should demonstrate an awareness of other tools and resources within the ecosystem. This doesn't mean that all software should be able to read and write to all possible file formats. Rather designers should make deliberate choices about the ways in which creators can transfer information in and out of a tool, and how these processes connect to specific applications and workflows. Ecosystems also suggest life and growth, therefore tools must be actively maintained and changed to reflect the emerging needs of people who use them.

It is important to emphasize that building and maintaining an interacting, evolving system is contingent on empowering new people to build their own tools. This can take many different forms. It includes building on efforts to increase diversity in computer science, which can translate to a better representation of different kinds of people in mainstream software engineering. It also creating requires systems that make it feasible for practitioners to make their own tools. At the end of the day, the person with the best understanding of the creative needs of a particular artist is the artist themselves.

Treat artists as collaborators, not users

As previously stated, creating effective tools requires an understanding of the kinds of artifacts they support. Therefore, systems engineers must devote a portion of their time to testing out their tools by actually making things with them. It can be extremely challenging however to effectively transition between tool creation and artifact creation. Despite the interdependence between creating systems and creating artwork, these two practices involve different skills, and require different forms of evaluation. This tension is one of the reasons that evaluating tools with artists is so valuable. As experts in artifact creation, artists can push tools to their limits, apply them in unexpected ways, and reveal concrete affordances and deficiencies.

In the field of Human-Computer-Interaction, it's common to refer to the people who test out a system as "users". In my conversations with Fish, he described why he disliked this term:

It implies dependence. I think the people that use these tools are independent and we have no business calling them that. We have no right to say that we're informing their experience. They create their experience.

Fish 's statement demonstrates how the term user suggests that it is the software that defines a person 's experience, rather than the person themselves. Moreover the term user also reduces a person's creative agency within the context of human-computer interaction research by implying that the only person engaged in a creative act is the researcher or designer of the system. For the purposes of this research, that depiction is inaccurate. Both system designers and artists possess different and potentially complementary forms of expertise, and both are capable of meaningful creative contributions over the course of the research. As the evaluations of Para and Dynamic Brushes demonstrate, the artifacts produced by artists directly shape future applications of the system. Furthermore, in-depth discussions with artists about the process of using a system do more than enable system designers to correct misconceptions or provide technical

insights. They provide system designers with new ideas about making and creativity, ideas that significantly improve the system designers' abilities to develop novel, diverse, and creative technologies.

An alternative way of framing the approach to system development that is outlined in this dissertation, is as a creative collaboration. In this collaboration, the artist and the engineer work together to produce three outcomes: a system, a collection of artwork, and a body of knowledge about how programming can support creative practice. Framing the artist as a collaborator rather than a user is more than a semantic nicety. The ways in which people describe the roles of participants shape their perception of what is and is not expected of them, and therefore, can affect the work they produce and the statements they make. Similarly, the ways engineers and researchers refer to the people who use their tools reflect their view of the abilities and agency of those people, and therefore, affects the choices they make in designing those tools. This generalization extends to reporting results as well, where choice of words can influence how readers perceive the value of different people 's contributions to the work itself.

7

Conclusion

The final chapter of this dissertation reflects on potential future projects and discusses the contributions of this work to the broader field of systems engineering research. The chapter begins by looking at additional domains of artistic practice that might benefit from the approaches of Para and Dynamic Brushes, and proposes alternative design and evaluation methodologies that may further enable the development of expressive computational tools for manual practitioners. This is followed by a discussion of the implications of artificial intelligence for artists. The chapter presents a series of questions that address how we might offer new creative opportunities through artificial intelligence without reducing human artistic agency. The chapter concludes by demonstrating the value of multidisciplinary knowledge in systems engineering and advocating for the importance of supporting people from non-traditional computer science backgrounds in building systems of their own.

7.1 Opportunities for Future Work

In addition to enabling new artifacts and experiences, and contributing new knowledge, another measure of a research project's success is its ability to hint at new pathways for creation and investigation. In their current state, Dynamic Brushes and Para offer meaningful forms of engagement. However, the process of developing and evaluating both systems also suggested opportunities for future research. This section describes three avenues of future work: opportunities for broader deployment and evaluation, developing systems for dynamic artifact production, and artistic production as research.

7.1.1 Breadth-based Evaluation

The depth based studies had many advantages, yet they also had the limitation of reflecting the viewpoints and experiences of a relatively small number of artists. Going forward, it would be valuable to compare the results of these initial evaluations with experiences from a broader community of practitioners.

This breadth-based evaluation could take several forms. One approach would be to conduct in-person workshops with larger groups of artists. This approach would enable the systems to be evaluated by a larger number of people, and hopefully produce a greater variety of finished artwork, without requiring significant amounts of additional engineering. Furthermore, it would provide the opportunity to observe people working with Para and Dynamic Brushes in a group, which would provide opportunities for dialog between artists throughout the creative process, and potentially enable additional forms of collaboration. Workshops could also be structured so that the same group of artists have the opportunity to work with both systems and compare them.

Another opportunity for evaluation with a broader audience would involve making the systems freely available general public. This strategy corresponds with the approach of many prior creative coding tools and frameworks. It's important to point out however, that effective deployment of a new system for public use requires many support structures. Public deployment would require significant engineering efforts to make each system more robust and compatible with different platforms and browsers. In the case of Dynamic Brushes, it would require improving the server architecture to support larger numbers of creators, whereas Para would require better ways of saving and storing people's artwork. Additionally, public deployment requires the creation of online documentation and tutorials, and involves providing channels for people to ask questions and receive assistance. Finally, from a research perspective, public deployment also requires developing mechanisms for collecting information from people online, a process which brings up different concerns about privacy and attribution. Prior examples of "in the wild" evaluation of systems and hardware demonstrate the unique research opportunities of public deployment (Buechley and Hill, 2010; Qi et al., 2016). Yet, this process also requires laborious engineering of many standard software features, and may require a lengthy period of initial development before useful information is generated. Rather than release the system as a standalone piece

of software, an alternative approach is to work with industry partners to incorporate aspects of a system into existing software. Because Para and Dynamic Brushes incorporate interface components from existing digital art and design tools, they have the potential to be incorporated directly into the systems that they extend. This approach would have the benefit of reaching a large audience, because established software platforms already have a large community of practitioners. The challenge of this approach is that existing software platforms have established technical standards and interaction paradigms that can conflict with the functionality of a system developed in research. This is especially true for research that focuses on innovations in interface and interaction design; it's generally more challenging, and potentially more contentious to to adapt the interface of an existing piece of software than to incorporate a new filter or effect.

Both stand-alone deployment or incorporation into existing software pose challenges, but they also offer significant advantages. Releasing a tool to the public offers the opportunity to generate feedback and and artwork from a significantly larger community of artists and gain an understanding of how a system performs in an ecologically valid context. Furthermore, many promising systems developed in research contexts never see the light of day. Investing the effort to continue past the stage of the prototype would allow for a broader range of people to actively benefit from the results of systems engineering research.

7.1.2 Dynamic Artifacts

Although Para and Dynamic Brushes were developed as a tools for creating static artwork, they offer potential starting points for other forms of procedural creation. Parametric constraints are often applied to the design of physical objects, therefore as a parametric tool, Para lends itself to CAD applications. The familiarity of Para's interface for people with experience in vector graphics, combined with the system's emphasis on producing ornate 2D patterns, suggests the opportunity to apply Para to the design of ornamental patterns and designs for physical objects. Textile and clothing design might be a particularly fruitful domain of exploration.

Another potential domain is animation. Both Para and Dynamic Brushes enable dynamic control over multiple visual elements. Furthermore, while designed to support static artwork, the process of working with both tool produces animation-like effects, and in the case of Dynamic Brushes, encourages artists to produce actual animations. These qualities suggest that both systems could be modified to support generative animation

and motion graphics. Migrating either Para or Dynamic Brushes towards animation would provide the opportunity to explore new ways of enabling artists to dynamically control how artwork changes over time. Procedural animation would also pose new challenges and raise new questions with regards to integrating manual and procedural creation. Possible questions include: What interfaces most effectively represent time-based procedural relationships? How can procedural systems enable artists to preview the effect of a procedural relationship over the course of an entire animation? How can procedural models be designed that are both expressive and approachable for people working in traditional forms of animation?

Perhaps the most exciting domains to consider are those that offer the potential for entirely new art-forms. In both procedural and manual art there are subsets of practitioners who engage in live performance. These artists create drawings, paintings, or computationally-produced compositions in real time in front of an audience. The liveness of the Dynamic Brushes programming language, combined with the speed of stylus-based drawing suggest the potential to extend these performance-based art forms, by enabling artists to create live compositions that blend procedural and manual creation. Developing Dynamic Brushes as a performance tool would offer the opportunity to incorporate other forms of sensors or input devices that would enable artists to physically shape the artwork in real time, or experiment with supporting multiple artists working collaboratively. Finally, applying Dynamic Brushes to performance would act as a forcing function to modify the programming representation to the point where artists can re-configure or build new tools without disrupting the flow of the performance itself.

7.1.3 Artistic Production as Research

The design and evaluation methodologies of this dissertation stress the importance of collaboration with artists. Any future research in applying either system to other domains of artistic practice would therefore involve collaborating with artists in sculpture, textile design, animation, and performance. Collaborations with a broader variety of artists would provide the opportunity to further iterate on and improve the depth-based study methodology. Furthermore, additional development of either Para or Dynamic Brushes would likely lead to extended versions of the depth-based approach.

In the Para and Dynamic Brushes studies, artists spent two weeks engaged with each system. This represents a significant amount of time

when compared to other approaches to system evaluation in human-computer-interaction research. Yet in the real world, professional artists often spend months, or even years developing and refining a single piece of work. Developing and evaluating a procedural art system over a similar period of time would offer new research opportunities in comparison to the two-week approach. These include: the opportunity to more thoroughly test the expressive limits of a system, the possibility of performing more extensive iterations on system functionality in response to an artist's creative objectives, and the potential for artists to produce more mature and developed forms of work with the system. When considering possible approaches to long-term studies with creative systems, it may be useful to build off long-term creative methods employed by professional artists. To this end, here are several possible directions for long-term evaluation methodologies that draw from methodologies in art:

- **The 100 Day Research Project:** The 100 Day workshop is a practice developed by the designer Michael Bierut, in which students are required to repeat a creative task every day for 100 days (2011). 100 day projects have since been undertaken by numerous artists and designers across different disciplines as motivational tool and a means to push their practice forward. The *100 Day Research Project* would build on this idea by having a systems researcher and an artist work together to use a prototype system to create one piece every day for 100 days. The artist's role would be to create each piece, while the researcher's role would be to modify the system in response to the work and ideas of the artist. This approach could reduce the pressure to create individual polished works and serve as a forcing function to investigate the full extent of the tool's creative potential.

- **Exhibition as Motivation:** Artists often create work with the intention of showing it to the public, a process which raises both practical and conceptual concerns. These include: identifying the audience for the work, determining the technical requirements for installing the piece, and considering how the work will be preserved or documented following the exhibition. The *Exhibition as Motivation* study method would be aimed at examining how a system performs throughout the process of conceiving of, creating, and exhibiting a work. At the start of the study, the systems researcher and the artist would agree on a general goal for the exhibition and a timeline for completion. This objective would be revisited throughout the course of the study as both the system and the artwork change in response to one another. Because creating work for exhibition generally involves the use of multiple tools and platforms this methodology would aid in understanding how effectively a system performs in conjunction with other technologies and techniques. Fur-

thermore, the exhibition itself would allow the researcher to observe public perceptions of the artwork.

- **Developing Domain Knowledge through Practice:** The previous chapter demonstrated the importance of domain knowledge when developing tools for artists. The approach taken in this dissertation to building domain knowledge involved interviewing a variety of professional artists. However, as these conversations demonstrate, many aspects of manual art knowledge are tacit, meaning they are difficult to express through words and better understood through physical engagement. In order to develop tacit domain knowledge, the *Domain Knowledge through Practice* study methodology would have researchers work under a professional artist directly to learn aspects of their craft. This learning period could occur either prior to system development, or in conjunction with the development process itself. This methodology would increase the time required to develop a system, but would also provide the researcher with a greater understanding of the tactile and physical aspects of the manual practice they seek to extend.

Each of these proposed study methodologies emphasizes the same basic idea: developing effective tools involves making things. The inherent tension between the mindsets involved in artifact creation versus those involved in tool creation can make it tempting to remain siloed in one space or the other. Study methods that push researchers to work in both these spaces will result in better systems and more interesting artwork.

7.2 Creative Programming and Artificial Intelligence

Computation is a constantly changing field. One domain of computation that suggests the potential to significantly reshape the field is artificial intelligence. Online technology and ubiquitous computing enable the generation and organization of extremely large datasets that cover a wide range of human behavior, including products consumed, decisions made, approaches to playing games, ways of accessing information, and creative processes. This data, paired with machine learning algorithms, has been used to develop systems that appear to perform tasks previously only performed by humans. From the wide range of emerging applications of Artificial Intelligence, Artificially intelligent systems designed to produce human-like artwork are of particular relevance to this research. A subfield of machine learning called deep learning enables automated production of illustrations or paintings within a specific style, novel music compositions, and systems that auto-complete sketches and drawings. The rapid ad-

vancement of this technology suggests that it will be possible to automate other aspects of human artistic creation in the future.

The work described in this dissertation is motivated by the argument that computation is a meaningful creative medium for human artists. As a result, many of the contributions of this work focus on ways to mitigate the challenges of applying programming or procedural techniques, rather than eliminating them through machine inference and automation. For some people, the growth of computational systems that autonomously create artwork, suggests that programming itself may eventually become an irrelevant artistic skill. However, the history of art, both computational and manual, and the motivations of many artists strongly suggest that while the *form* of programming is likely to change, advances in artificial intelligence are unlikely to eliminate artists' need or desire to program.

Many artists already incorporate aspects of automation in their work, yet they generally retain some aspect of human creative control. This is because complete automation of a creative process trivializes the end result. If anyone can automatically generate a painting in the style of Van Gogh or a song that sounds like a composition of the Beatles, those artifacts eventually become uninteresting as artwork. As the interviews demonstrate, the role of a computational artist is not just to use technology, but also to twist it, subvert it, and reshape it to the point that it can be used to produce critical, difficult, or unexpected outcomes. Doing so requires the ability to program. Artists may push the edge of forms of production driven by artificial intelligence, and in fact, have done so already. But it is unlikely that artists will ever be satisfied with outcomes from a system with which they have no meaningful involvement.

When considering the implications of artificial intelligence and art, it's also important to recognize that artists almost always care equally about process and outcomes. The artist Memo Akten described how artwork produced with deep learning appealed to him, not for the aesthetics, but for the processes involved in the algorithms themselves:

The aesthetic is interesting. It's trippy, surreal, abstract, psychedelic, painterly, rich in detail. But the novelty is likely to wear off quickly for most people except for the [especially] dedicated. Using new datasets or learning to control the output (so it's not just puppy-slugs) can undoubtedly give it a new breath of life. And there is potential to do very interesting work conceptually pairing datasets with seed images. But it's not the aesthetic that excites me. Instead, like I said recently... the poetry behind the scenes is blowing my mind! (2015)

This comment indicates that one of the engaging aspects of artificially intelligent systems is not what they create, but how they create. Engaging in the crafting of the functionality of these systems can reveal how their functionality aligns with or diverges from the ways in which human artists work, and can reveal patterns in human-designed artifacts that might not be readily apparent. Artificial intelligence and machine learning involve specific approaches to programming and debugging, and as these technologies continue to advance, they will continue to change how people program. Yet no matter the form of programming, pushing the edge of a technology, or understanding how it works require an understanding of the code that comprises it. For artists interested in leveraging these technologies to produce artifacts or explore processes, programming is and will continue to be an important skill.

The more complex, but perhaps more significant response to the question of whether artificial intelligence negates the need for artists to program is that *it is not the right question.* Similar to the introduction of the computer itself, artificial intelligence has the potential to fundamentally change how artists make things. While the scope of this change is still unclear, there are important questions that systems engineers and researchers can begin addressing to maximize the creative potential of this new technology while reducing the likelihood that it will disenfranchise human creators. Possible questions include:

1. What unique, new creative opportunities are offered by artificial intelligence and what kinds of creators will these opportunities appeal to?

2. What new implications for attribution and ownership will result from this form of production, and how can human creators maintain control over their own work at the same time they engage in the sharing of data and information?

3. Humans and machines have different strengths when it comes to making things. How can we build systems that enable both computational agents and human creators to work together in ways that reflect the strength of both parties?

4. What new approaches to learning are necessary to enable artists to apply artificially intelligent forms of creation in personally meaningful and expressive ways?

5. In addition to providing new forms of art creation, how can artificial intelligence support established domains of artistic practice without eliminating forms of engagement that are important to artists?

The systems described in this dissertation do not incorporate machine learning or artificial intelligence; however, the design methods and approaches to evaluation presented are relevant to creative tools regardless of technology. Collectively, they provide a framework for developing tools with emerging technologies that preserve human agency, style, and choice while opening up new domains of computational expression.

7.3 Multidisciplinary Systems Engineering

Ultimately this dissertation is about two topics— building computational tools that enable different kinds of people to make things, and supporting thinking about making in new ways. When this research is presented to software developers or researchers of systems engineering, a version of the following question is often asked:

Do you really believe that artists will want to program?

At first, this question surprised me. Developers and software engineers program on a daily basis and generally recognize the creative power of computation. Why not naturally see the value of extending this power to people in other domains? After reflection, I now believe this question has little to do with a failure to recognize the creative potential of computing. Instead, I believe it mirrors the distinction many engineers see between their work and the work of others. This attitude possibly connects to broad social and cultural perceptions that certain people are suitable for engineering and programming and others are not (Wakabayashi, 2017). Unfortunately, despite a great deal of evidence to the contrary, this perspective also is often internalized by people who don't fit within the stereotypical norms of engineering, or who arrive to programming through unconventional pathways (Margolis et al., 2010; Margolis and Fisher, 2003).

My experience of working with different people in different forms of programming has resulted in a strong belief that anyone has the capacity to program. Moreover, my transition from manual art to systems engineering research has taught me that programming skills are only one aspect of building innovative, effective, and interesting computational systems. Diversity is another. Efforts to increase the representation of minorities and women in computer science is an important step in diversifying the kinds

of computational tools that are created. Encouraging diversity also involves broadening conceptions about the skills, knowledge, and backgrounds that are relevant to system engineering. As the work in this dissertation demonstrates, building a piece of software requires some traditional engineering skills, but it's also dependent on knowledge and experiences in many other fields.

An understanding of the principles and process of interaction design enables the creation of systems that are not merely usable, but pleasurable, engaging, and compelling. Knowledge of interaction design is also relevant to systems engineers who work in collaboration with designers. It can help them understand why a designer may push for features that seem incidental to the functionality of the system and are laborious to implement.

Teaching experience is also valuable. Engineers who have an understanding of theories of learning or who have taught others can be better at recognizing the kinds of learning challenges a system may present and better positioned to develop ways of scaffolding learning. Teaching can provide engineers with empathy for the people who use their tools. It can also dispel unconstructive attitudes towards learners by demonstrating that new creators, while inexperienced, still posses strong design objectives, stylistic preferences, and personal values.

The capacity to investigate the attitudes and values of different communities and individuals is also an important skill. Specifically, experience with methods of social science research can help engineers understand general values, practices, and attitudes of communities who use their systems. Practice in conducting in-person interviews can facilitate dialog with individual creators and domain experts, and provide invaluable insight for the improvement of a system.

Most importantly, in addition to people who arrive at systems engineering through established forms of computer science education, the field of computer science could also provide support for people who arrive from other domains. Moving from working as a domain practitioner to creating systems for that domain is an incredibly challenging process in itself. Unfortunately, this process often is made more challenging by existing standards and silos in computer science education and hiring practices, which can limit participation to a few highly structured, traditional pathways. Future computational innovation is directly dependent on engaging people who are not traditional computer scientists as systems engineers. People who work in the arts, journalism, civic engagement, biology, and other fields are uniquely positioned to develop effective, unexpected, and domain-shaping tools for those fields. It is not coincidence that many of

the most popular platforms for creative coding were developed by people who went to art school. Not all practitioners need to create systems; however, for creators who demonstrate an interest in building their own computational tools, every effort should be made to support and encourage them along the way.

Bibliography

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

Aish, R. (2012). DesignScript: origins, explanation, illustration. In *Computational Design Modelling*, pages 1–8. Springer.

Akten, M. (2015). #Deepdream is blowing my mind. `http://medium.com/@memoakten/deepdream-is-blowing-my-mind-6a2c8669c698`.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

Amershi, S., Cakmak, M., Knox, W. B., and Kulesza, T. (2014). Power to the people: The role of humans in interactive machine learning. *AI Magazine*, 35(4):105–120.

Arduino (2017). Arduino introduction. `http://www.arduino.cc/en/Guide/Introduction`.

Autodesk (2016). Project dreamcatcher. `http://autodeskresearch.com/projects/dreamcatcher`.

Azzarello, N. (2017). Stefanie pender teaches a robot the art of glassmaking. `http://http://designboom.com/art/stefanie-pender-robot-glassmaking-autodesk-04-17-2017/`.

Bader, C., Patrick, W. G., Kolb, D., Hays, S. G., Keating, S., Sharma, S., Dikovsky, D., Belocon, B., Weaver, J. C., Silver, P. A., et al. (2016). Grown, printed, and biologically augmented: An additively manufactured microfluidic wearable, functionally templated for synthetic microbes. *3D Printing and Additive Manufacturing*, 3(2):79–89.

Benjamin, W. (1968). *Illuminations*. Houghton Mifflin Harcourt.

Berger, J. (1972). *Ways of Seeing*. Penguin Books Limited.

Berger, J. (2008). *Selected Essays of John Berger*. Vintage International. Knopf Doubleday Publishing Group.

Bierut, M. (2011). Five years of 100 days. `http://designobserver.com/feature/five-years-of-100-days`.

Blackwell, A. F. (2014). Palimpsest: A layered language for exploratory image processing. *Journal of Visual Languages Computing*, 25(5):545 – 571.

Blauvelt, G., Wrensch, T., and Eisenberg, M. (1999). Integrating craft materials and computation. In *Proceedings of the 3rd Conference on Creativity & Cognition*, C&C '99, pages 50–56, New York, NY, USA. ACM.

Bontá, P., Papert, A., and Silverman, B. (2010). Turtle, art, turtleart. In *Proc. of Constructionism 2010 Conference*.

Bouzit, M., Burdea, G., Popescu, G., and Boian, R. (2002). The rutgers master ii-new design force-feedback glove. *IEEE/ASME Transactions on Mechatronics*, 7(2):256–263.

Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. (2010). Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 513–522, New York, NY, USA. ACM.

Brennan, K. and Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, pages 1–25.

Buechley, L. and Eisenberg, M. (2008). The LilyPad Arduino: Toward wearable engineering for everyone. *IEEE Pervasive Computing*, 7(2):12–15.

Buechley, L. and Hill, B. M. (2010). LilyPad in the wild: How hardware's long tail is supporting new engineering and design communities. In *Proceedings of the 8th ACM Conference on Designing Interactive Systems*, DIS '10, pages 199–207, New York, NY, USA. ACM.

Chaudhuri, S. and Koltun, V. (2010). Data-driven suggestions for creativity support in 3d modeling. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 183:1–183:10, New York, NY, USA. ACM.

Choi, I. and Follmer, S. (2016). Wolverine: A wearable haptic interface for grasping in VR. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16 Adjunct, pages 117–119, New York, NY, USA. ACM.

Cooper, S., Dann, W., and Pausch, R. (2000). Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in*

Colleges, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges.

Crutzen, C. and Kotkamp, E. (2008). *Object Orientation*. EBSCO ebook academic collection. MIT Press.

Cypher, A. and Halbert, D. (1993). *Watch what I Do: Programming by Demonstration*. MIT Press.

DiGiano, C. and Eisenberg, M. (1995). Self-disclosing design tools: A gentle introduction to end-user programming. In *Proceedings of the 1st Conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques*, DIS '95, pages 189–197, New York, NY, USA. ACM.

Dixon, M. and Fogarty, J. (2010). Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1525–1534, New York, NY, USA. ACM.

Do, E. and Gross, M. D. (2007). Environments for creativity: A lab for making things. In *Proceedings of the 6th ACM SIGCHI Conference on Creativity & Cognition*, C&C '07. ACM.

Efrat, T. A., Mizrahi, M., and Zoran, A. (2016). The hybrid bricolage: Bridging parametric design with craft through algorithmic modularity. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5984–5995, New York, NY, USA. ACM.

Faaborg, A. and Lieberman, H. (2006). A goal-oriented web browser. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 751–760, New York, NY, USA. ACM.

Gogia, S. (2016). *Insight: interactive machine learning for complex graphics selection*. PhD thesis, Massachusetts Institute of Technology.

Goodman, N. (1968). *Languages of Art: An Approach to a Theory of Symbols*. Bobbs-Merrill.

Guerrero, P., Bernstein, G., Li, W., and Mitra, N. J. (2016). Patex: Exploring pattern variations. *ACM Trans. Graph.*, 35(4):48:1–48:13.

Guzdial, M. (2003). A media computation course for non-majors. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '03, pages 104–108, New York, NY, USA. ACM.

Hartmann, B., Wu, L., Collins, K., and Klemmer, S. R. (2007). Programming by a sample: Rapidly creating web applications with d.mix. In *Proceed-*

ings of the 20th Annual ACM Symposium on User Interface Software and Technology, UIST '07, pages 241–250, New York, NY, USA. ACM.

Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S. R. (2008). Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, pages 91–100, New York, NY, USA. ACM.

Harvey, B. (1991). Symbolic programming vs. the a.p. curriculum. *The Computing Teacher*, 56:27–29.

Hoarau, R. and Conversy, S. (2012). Augmenting the scope of interactions with implicit and explicit graphical structures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1937–1946, New York, NY, USA. ACM.

Hughes, T. (2017). Robot's personality comes to life in museum exhibit. `http://cmu.edu/piper/stories/2017/february/robot-comes-to-life.html`.

Industries, M. (2015). Thingiverse customizer. `http://www.thingiverse.com/apps/customizer`.

Jacobs, J. and Buechley, L. (2013). Codeable objects: Computational design and digital fabrication for novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1589–1598, New York, NY, USA. ACM.

Jacobs, J., Resnick, M., and Buechley, L. (2014). Dresscode: supporting youth in computational design and making. In *Constructionism*, Vienna, Austria.

Johnston, W. M., Hanna, J. R. P., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34.

Jordà, S., Geiger, G., Alonso, M., and Kaltenbrunner, M. (2007). The reactable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, TEI '07, New York, NY, USA. ACM.

Kalogerakis, E., Chaudhuri, S., Koller, D., and Koltun, V. (2012a). A probabilistic model for component-based shape synthesis. *ACM Trans. Graph.*, 31(4):55:1–55:11.

Kalogerakis, E., Nowrouzezahrai, D., Breslav, S., and Hertzmann, A. (2012b).

Learning hatching for pen-and-ink illustration of surfaces. *ACM Trans. Graph.*, 31(1):1:1–1:17.

Kay, A. and Goldberg, A. (1977). Personal dynamic media. *Computer*, 10(3):31–41.

Kay, A. C. (1990). *User Interface: A Personal View*. Addison-Wesley.

Kay, A. C. (1993). The early history of smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 69–95, New York, NY, USA. ACM.

Kazi, R. H., Chevalier, F., Grossman, T., and Fitzmaurice, G. (2014). Kitty: Sketching dynamic and interactive illustrations. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, New York, NY, USA. ACM.

Kazi, R. H., Grossman, T., Umetani, N., and Fitzmaurice, G. (2016). Skuid: Sketching dynamic drawings using the principles of 2d animation. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, pages 84:1–84:1, New York, NY, USA. ACM.

Kim, N. W., Schweickart, E., Liu, Z., Dontcheva, M., Li, W., Popovic, J., and Pfister, H. (2017). Data-driven guides: Supporting expressive design for information graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):491–500.

Klemmer, S. R., Hartmann, B., and Takayama, L. (2006). How bodies matter: Five themes for interaction design. In *Proceedings of the 6th Conference on Designing Interactive Systems*, DIS '06, New York, NY, USA. ACM.

Ko, A. J., Myers, B. A., and Aung, H. H. (2004). Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 199–206, Washington, DC, USA. IEEE Computer Society.

Krainin, J. and Lawrence, M. R. (1990). Recorded lecture by steve jobs. `http://www.youtube.com/watch?v=ob_GX50Za6c`.

Landay, J. A. and Myers, B. A. (1995). Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 43–50, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Lévi-Strauss, C. (1966). *The Savage Mind*. Nature of Human Society. University of Chicago Press, Chicago, IL, USA.

Levin, G. (2000). *Painterly Interfaces for Audiovisual Performance*. PhD thesis, Massachusetts Institute of Technology.

Levin, G. (2003). Essay for creative code.

Levin, G. (2015). *EYEO: Converge to Inspire. 2011-2015*.

Lieberman, H. (1993). Mondrian: A teachable graphical editor. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 144–, New York, NY, USA. ACM.

Lieberman, Z. (2014a). Foreword. In *ofBook, a collaboratively written book about openFrameworks*. `http://openframeworks.cc/ofBook/chapters/foreword.html`.

Lieberman, Z. (2014b). Philosophy. In *ofBook, a collaboratively written book about openFrameworks*. `http://openframeworks.cc/ofBook/chapters/of_philosophy.html`.

Maeda, J. (1999). *Design by Numbers*. MIT Press.

Malan, K. and Halland, K. (2004). Examples that can do harm in learning programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 83–87, New York, NY, USA. ACM.

Maleki, M. M., Woodbury, R. F., and Neustaedter, C. (2014). Liveness, localization and lookahead: Interaction elements for parametric design. In *Proceedings of the 2014 Conference on Designing Interactive Systems*, DIS '14, pages 805–814, New York, NY, USA. ACM.

Maloney, J. H. and Smith, R. B. (1995). Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*. ACM.

Margolis, J., Estrella, R., Goode, J., Holme, J., and Nao, K. (2010). *Stuck in the Shallow End: Education, Race, and Computing*. MIT Press.

Margolis, J. and Fisher, A. (2003). *Unlocking the Clubhouse: Women in Computing*. MIT Press.

Marks, J., Andalman, B., Beardsley, P. A., Freeman, W., Gibson, S., Hodgins, J., Kang, T., Mirtich, B., Pfister, H., Ruml, W., Ryall, K., Seims, J., and Shieber, S. (1997). Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 389–400, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

McCullough, M. (1996). *Abstracting Craft: The Practiced Digital Hand*. The MIT Press, Cambridge, Massachusetts.

McMullan, T. (2017). Life drawing and machine learning: An interview with artist anna ridler. http://alphr.com/go/1006623.

Mitchell, W. (1990). *The Logic of Architecture: Design, Computation, and Cognition*. Cognition special issues. MIT Press, Cambridge, MA, USA.

Motzenbecker, D. (2017). Fast drawing for everyone. http://blog.google:443/topics/machine-learning/fast-drawing-everyone.

Mumford, L. (1952). *Art and Technics*. Bampton lectures in America. Columbia University Press.

Myers, B., Hudson, S. E., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28.

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123.

Nadin, M. (1997). *The Civilization of Illiteracy*. Dresden University Press.

Nakagaki, K., Vink, L., Counts, J., Windham, D., Leithinger, D., Follmer, S., and Ishii, H. (2016). Materiable: Rendering dynamic material properties in response to direct physical touch with shape changing interfaces. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 2764–2772, New York, NY, USA. ACM.

Needleman, C. (1979). *The work of craft: an inquiry into the nature of crafts and craftsmanship*. Arkana.

Norman, D. (2002). *The Design of Everyday Things*. Basic Books.

Obrist, H. U. (2013). *Do it: The Compendium*. Independent Curators International.

O'Donovan, P., Agarwala, A., and Hertzmann, A. (2015). Designscape: Design with interactive layout suggestions. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1221–1224, New York, NY, USA. ACM.

Oehlberg, L., Willett, W., and Mackay, W. E. (2015). Patterns of physical design remixing in online maker communities. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, New York, NY, USA. ACM.

Oney, S., Myers, B., and Brandt, J. (2012). Constraintjs: Programming interactive behaviors for the web by integrating constraints and states. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, New York, NY, USA. ACM.

Oney, S., Myers, B., and Brandt, J. (2014). Interstate: A language and environment for expressing interface behavior. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, New York, NY, USA. ACM.

Ozenc, F. K., Kim, M., Zimmerman, J., Oney, S., and Myers, B. (2010). How to support designers in getting hold of the immaterial material of software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, New York, NY, USA. ACM.

Papert, S. (1980). *Mindstorms: children, computers, and powerful ideas*. Basic Books.

Perteneder, F., Bresler, M., Grossauer, E.-M., Leong, J., and Haller, M. (2015). cluster: Smart clustering of free-hand sketches on large interactive surfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 37–46, New York, NY, USA. ACM.

Pirolli, P. L. and Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie*, 39(2):240.

Puckette, M. (1988). The patcher. In *Proceedings of the 1988 International Computer Music Conference. San Francisco*. International Computer Music Association.

Qi, J. et al. (2016). *Paper electronics: circuits on paper for learning and self-expression*. PhD thesis, Massachusetts Institute of Technology.

Reas, C. and Fry, B. (2007). *The Processing Handbook*. MIT Press, Cambridge, Massachusetts, USA.

Reas, C., McWilliams, C., and LUST (2010). *Form and Code*. Princeton Architectural Press, New York, NY, USA.

Reichardt, J. (1969). *Cybernetic serendipity: the computer and the arts*. Praeger.

Renkl, A., Stark, R., Gruber, H., and Mandl, H. (1998). Learning from worked-out examples: The effects of example variability and elicited self-explanations. *Contemporary Educational Psychology*, 23(1):90 – 108.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). Scratch: Programming for all. *Commun. ACM*, 52(11).

Resnick, M. and Rosenbaum, E. (2013). Designing for tinkerability. In Honey, M. and Kanter, D., editors, *Design Make Play: Growing the Next Generation of STEM Innovators*. Routledge.

Resnick, M. and Silverman, B. (2005). Some reflections on designing construction kits for kids. In *Proceedings of the 2005 Conference on Interaction Design and Children*, IDC '05, New York, NY, USA. ACM.

Roedl, D., Bardzell, S., and Bardzell, J. (2015). Sustainable making? balancing optimism and criticism in hci discourse. *ACM Trans. Comput.-Hum. Interact.*, 22(3).

Rosenkrantz, J. and Louis-Rosenberg, J. (2015). Nervous system tools. `http://n-e-r-v-o-u-s.com/tools`.

Schachman, T. (2012). Alternative programming interfaces for alternative programmers. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, New York, NY, USA. ACM.

Schachman, T. (2015). Apparatus: a hybrid graphics editor / programming environment for creating interactive diagrams. In *Strange Loop*.

Schön, D. (1992). Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems*, 5(1):3 – 14. Artificial Intelligence in Design Conference 1991 Special Issue.

Schön, D. and Bennett, J. (1996). Bringing design to software. chapter Reflective Conversation with Materials. ACM, New York, NY, USA.

Scratch (2017). Scratch website. `http://scratch.mit.edu`.

Sennett, R. (2008). *The Craftsman*. Yale University Press.

Shneiderman, B. (1993). *Direct manipulation: a step beyond programming languages*. Human/computer interaction. Ablex Publishing Company.

Silver, J., of Technology. Department of Architecture. Program in Media Arts, M. I., and Sciences (2014). *Lens X Block: World as Construction Kit*.

Sims, K. (1994). Evolving virtual creatures. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 15–22, New York, NY, USA. ACM.

Smith, D. C. (1975). *Pygmalion: A Creative Programming Environment*. PhD thesis, Stanford, CA, USA.

Sterling, B. (2015). Design fiction: Sputniko!, âĂIJtranceflora, amyâĂŹs glowing silkâĂİ. `http://wired.com/beyond-the-beyond/2015/04/design-fiction-sputniko-tranceflora-amys-glowing-silk/`.

Sutherland, I. E. (1964). Sketchpad a man-machine graphical communication system. *Transactions of the Society for Computer Simulation*, 2(5):R–3–R–20.

Taivalsaari, A. (1996). On the notion of inheritance. *ACM Computing Surveys (CSUR)*, 28(3):438–479.

Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N., and Měch, R. (2012). Learning design patterns with bayesian grammar induction. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 63–74, New York, NY, USA. ACM.

Torres, C. and Paulos, E. (2015). Metamorphe: Designing expressive 3d models for digital fabrication. In *Proceedings of the 2015 ACM SIGCHI Conference on Creativity and Cognition*, C&C '15, New York, NY, USA. ACM.

Turkle, S. and Papert, S. (1992). Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1).

Victor, B. (2011). Dynamic pictures. `http://worrydream.com/DynamicPicturesMotivation`.

Victor, B. (2012a). Inventing on principle. In *Proc. of the Canadian University Software Engineering Conference*.

Victor, B. (2012b). Learnable programming: Designing a programming system for understanding programs. `http://worrydream.com/LearnableProgramming`.

Victor, B. (2012c). Stop drawing dead fish. In *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12.

Victor, B. (2013a). Drawing dynamic data visualizations addendum. `http://worrydream.com/DrawingDynamicVisualizationsTalkAddendum`.

Victor, B. (2013b). Drawing dynamic data visualizations (talk). `http://vimeo.com/66085662`.

Wakabayashi, D. (2017). Contentious memo strikes nerve inside google and out. *The New York Times*. `http://www.nytimes.com/2017/08/08/technology/google-engineer-fired-gender-memo.html`.

Weintrop, D. and Wilensky, U. (2015). To block or not to block, that is the question: Students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*, IDC '15, pages 199–208, New York, NY, USA. ACM.

Xia, H., Araujo, B., Grossman, T., and Wigdor, D. (2016). Object-oriented drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 4610–4621, New York, NY, USA. ACM.

Xia, H., Araujo, B., and Wigdor, D. (2017). Collection objects: Enabling fluid formation and manipulation of aggregate selections. In *Proceedings of*

the 2017 CHI Conference on Human Factors in Computing Systems, CHI '17, pages 5592–5604, New York, NY, USA. ACM.