Extending Manual Drawing Practices with Artist-Centric Programming Tools

Jennifer Jacobs[1], **Joel Brandt**[2,†], **Radomír Měch**[2], **Mitchel Resnick**[1]
[1]MIT Media Lab [2]Adobe Research
{jacobsj,mres}@media.mit.edu jbrandt@snap.com[†] rmech@adobe.com

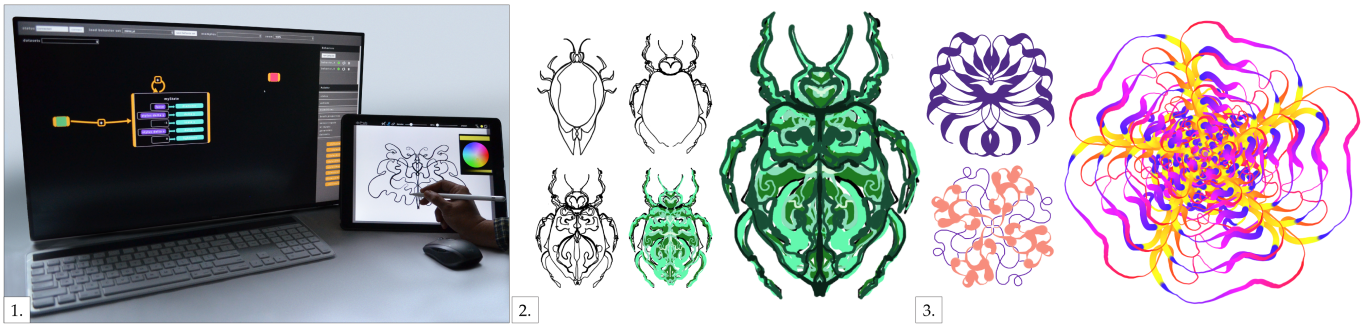

Figure 1. 1: Dynamic Brushes programming and drawing interfaces. 2-3: The system enables artists to create procedural drawing tools that extend manual drawing. Here, an artist has created tools that replicate and transform manual strokes through variation in rotation, reflection and style.

ABSTRACT
Procedural art, or art made with programming, suggests opportunities to extend traditional arts like painting and drawing; however, this potential is limited by tools that conflict with manual practices. Programming languages present learning barriers and manual drawing input is not a first class primitive in common programming models. We hypothesize that by developing programming languages and environments that align with how manual artists work, we can build procedural systems that *enhance*, rather than *displace*, manual art. To explore this, we developed Dynamic Brushes, a programming and drawing environment motivated by interviews with artists. Dynamic Brushes enables the creation of ad-hoc drawing tools that transform stylus inputs to procedural patterns. Applications range from transforming individual strokes to behaviors that draw multiple strokes simultaneously, respond to temporal events, and leverage external data. Results from an extended evaluation with artists provide guidelines for learnable, expressive systems that blend manual and procedural creation.

Author Keywords
Procedural art; generative art; programming

ACM Classification Keywords
D.2.6 Programming Environments

CHI 2018, April 21–26, 2018, Montréal, QC, Canada.

ACM ISBN 978-1-4503-5620-6/18/04 ...$15.00.
http://doi.org/10.1145/3173574.3174164

INTRODUCTION
Throughout human history, artists have used their hands to express themselves. Traditional arts like painting rely on the expressive power of manual tools to preserve traces of human movement and gesture [42]. Manual processes like drawing enable artists to quickly translate their ideas to reality [4]. Manual manipulation allows artists to develop knowledge and skills through physical practice [46, 56]. Computers have displaced some forms of manual production and altered others, however; they have also provided alternative artistic opportunities. Computer programming has given rise to *procedural art*: artworks described as a series of instruction and executed by a machine [53]. Procedural creation allows artists to manage complex structures, automate processes, and generalize and reuse operations [41, 52]. Computational tools support exploration and experimentation because their processes can be revised and iterated upon without loss of quality or affordance [40] and the act of programming enables reflection on aesthetic relationships and process [32].

Procedural and manual art practices have different qualities, yet they are *not* mutually exclusive. Experienced programmers build software that procedurally transforms manual drawings [33, 35, 45] and collaborations between artists and computer scientists demonstrate how algorithms from computational geometry [16] and machine learning [15] can be adapted to blend manual and generative designs. Yet the building blocks of procedural art are often poorly aligned with the processes of manual artists. Programming languages for art [10, 37, 50] use abstractions from general-purpose textual languages [51]. As a result, they have high learning thresholds for new programmers [31]. Because manual drawing inputs are not first-class primitives in most programming languages,

[†]Now at Snap Inc.

Constants	A constant numeric value.
Stylus Inputs	Data from the stylus including force, x and y position and delta, angle, heading, and speed.
UI Inputs	References to the drawing interface sliders and color-picker.
Generators	Functions that return a sequence of values between 0 and 100 corresponding with waveform or distribution—modulation with a sine function, or a random sequence of values.
Brush Properties	Properties of a brush, including position, geometry, and style.
Brush Hierarchy	Hierarchical properties relative to parent including spawn index and sibling count.

Table 1. Mapping Input Types

creating procedures that respond to manual drawing requires significant effort and expertise. Furthermore, programming environments present manual artists with unfamiliar interfaces and interactions that can prevent artists from incorporating their existing skills [23]. The creative opportunities of combining manual and procedural art, and the challenges manual artists encounter when attempting to do so led to our research question: *How can we develop programming environments that support the integration of procedural and manual art?*

We hypothesized that, by developing programming languages and environments that align with manual practices, we can build procedural systems that *enhance and extend* manual art. To explore this, we interviewed professional artists and examined manual and procedural practice. Interviews revealed the importance manual artists place on tools that preserve manual variation and style. Observations demonstrated how procedural artists incorporated manual expression in their work or collaborated with manual artists, by building *ad-hoc interactive drawing tools* that procedurally transformed manual drawings. Our observations led us to develop an approachable and expressive environment for *personal drawing-tool creation* that empowers manual artists to create procedural tools that support their established practices.

Prior research that aims to broaden participation in procedural art focuses on simplified textual languages [22, 48, 50], visual programming [13, 14, 17], or direct-manipulation of procedural and parametric relationships [21, 28, 63]. Instead, we build on systems that integrate representational programming and direct-manipulation interfaces [1, 9, 20, 24, 39], but with the new objective of tool creation. We make the following contributions: **First**, we introduce Dynamic Brushes, an integrated visual programming and stylus-based drawing environment. **Second**, we demonstrate the Dynamic Brushes programming model, which enables the creation of numerous tool behaviors through a small number of primitives and operations. The model combines declarative property mappings, states, and event-driven transitions to enable applications that range from procedural transformation of individual strokes to behaviors that draw multiple strokes simultaneously, respond to temporal events, and leverage external inputs and data. **Third**, we demonstrate how procedural tool creation can extend manual practices, support reflection, and foster agency through an extended evaluation with professional artists. **Fourth**, we provide insights for developing learnable and expressive procedural tools that are compatible with manual creation.

BACKGROUND

We built on personal experience in manual and procedural art through 9 interviews with professional artists. Within our subjects, 5 worked primarily with manual tools and 4 used programming. Interviews lasted 30 minutes to 2 hours. From analysis of interview transcripts we distilled themes on manual and procedural *learning*, *process*, and *expression*, which we describe in relation to learning and creativity research.

Learning

Manual artists learn through physical practice. Interviewed artists described how tactile feedback and physical engagement improved their understanding of proportion and composition and helped refine their aesthetic. The importance of physical learning is well established. Gestural expression and epistemic action enable distinct forms of understanding not possible through verbal or auditory forms of learning [30]. Art instructors often encourage students to learn technique through direct material engagement when visual demonstration and verbal instruction are insufficient [46]. This suggests that tools that limit manual engagement, like many programming platforms may hinder the learning of manual artists. 4 of 5 manual artists we spoke with were interested in programming, yet many encountered difficulties when learning it. 2 artists were particularly frustrated, stating that after investing significant time and effort, they were still only able to produce trivial outcomes. This experience aligns with the sharp learning thresholds encountered by learners of general-purpose programming languages. [43]. These frustrations also suggest a disconnect between the primitives of common programming models, which focus on manipulation of simple geometric forms, and the concerns of manual artists who, in our interviews, emphasized stylistic and aesthetic diversity.

Process

Every artist, regardless of discipline, emphasized the value of working quickly and continuously. The importance of continuous production corresponds with Cziksentmihalyi's concept of *flow*, an invested and active creative state [12]. Manual and procedural artists relied on different kinds of expertise to achieve flow. Procedural artists developed an understanding of technical approaches and design patterns, which enabled them to rapidly respond to a variety of ideas and applications. Conversely, manual artists emphasized building confidence in their gestures and instincts through manual practice, which led to the ability to create work intuitively. These different pathways to flow connect to different styles of learning, but they also correspond with affordances of manual and procedural tools. Traditional manual tools and direct-manipulation environments enable artists to sense what to try and when and intuit which tool to use [40]. By contrast, the breadth of many programming languages requires even expert programmers to regularly stop and consult references [7].

Programming is often portrayed as an exclusively analytic domain [11, 19], which can create the assumption that it is unappealing for people with intuitive processes. However, our interviews suggest that many manual artists are interested in both intuitive *and* analytic practices. Manual artists described how they also valued formal, abstract, or quantifiable representations of their work. One described how he referenced formal design principles when drawing, while another described efforts to quantify her process by documenting the distance travelled by her pen and her drawing execution time.

Position (active)	
x,y	Absolute Cartesian position.
radius,theta	Absolute polar position, relative to the brush origin.
delta x,y	Relative Cartesian position (position vector).
Geometry (passive)	
scale x,y	Scale of the segment relative to the stroke's origin.
rotation	Rotation of the segment relative to the stroke's origin.
Style (passive)	
diameter	Stroke width in pixels.
hue, saturation, lightness	Brush color in HSL.
alpha	Opacity of the stroke.

Table 2. Brush outputs. Brushes draw whenever mappings to active outputs are updated. Passive outputs are referenced when drawing occurs

Computers offer a bridge between concrete and formal domains by converting symbols to concrete artifacts [60]. This suggests that the right computational interface could enable manual artists to bridge analytic and intuitive practices.

Expression

Procedural and manual tools produce different aesthetics. Manual tools preserve evidence of the human hand, whereas procedural tools often diminish manual traces. Mumford argued that the aesthetic subtleties created by different artists produce "significant impulses that can come forth in no other way" [42]. Interestingly, artists from both procedural and manual domains expressed similar dissatisfaction with the flat, predictable quality of computational generativity and the loss of manual style when using procedural tools. Prior human-computer interaction (HCI) research in integrating craft and computation demonstrates how re-designing existing computational tools to accommodate traditional forms of making can provide access to digital fabrication [66] or embedded computing [8], while producing artifacts that retain manual aesthetics and traditional materials. Redesigning programming environments to preserve manual interaction could enable similar outcomes in procedural art.

Many procedural artists we spoke with created custom *tools* to incorporate manual expression into procedural work or foster collaborations with manual artists. These tools ranged from polished programming libraries and software to ad-hoc scripts written while making art. Levin argues that artists are innovative tool developers because they are well connected to the needs of other artists and invested in experimentation and exploration [34]. The procedural artists we spoke to often created tools as a byproduct of producing other artifacts, by generalizing and repackaging their code for personal or public use. This suggests that tool development is a natural affordance of programming. Providing more approachable forms of procedural tool creation making may enable more artists to create procedural functionality that aligns with their practices.

Design Goals

We used the themes from our interviews to generate design goals for procedural tools that support manual practice:

Learnable for new programmers: The system should allow people to make interesting things with simple programs and scaffold learning through multiple entry points. Artists should be able to independently explore procedural functionality.

Compatibility with manual practice: The programming model should emphasize procedural manipulation of manual drawing data. The interface should support manual and procedural interaction equally. The system should enable continuous drawing.

Balanced support for procedural and manual expression: The system should provide access to primary procedural affordances while preserving the manual styles of different artists.

Motivating Example

To illustrate potential applications of our system, we draw on our interview themes and design goals to describe an example artist who would benefit from our approach. Taylor is an illustrator and calligrapher. Through extensive practice, she's developed a fluid drawing style. Her work is characterized by symmetric repeating designs with organic curves. She works by sketching elements of a pattern with a pencil and then uses software to copy and repeat her hand-drawn forms. Doing so requires her to guess at how her drawings might look when digitally transformed. Taylor wants a tool that automatically repeats and transform her hand-drawn strokes *while she draws* and *in the manner she specifies*. Such a tool would allow her to rapidly generate ornate patterns and explore variations, without sacrificing her drawing style.

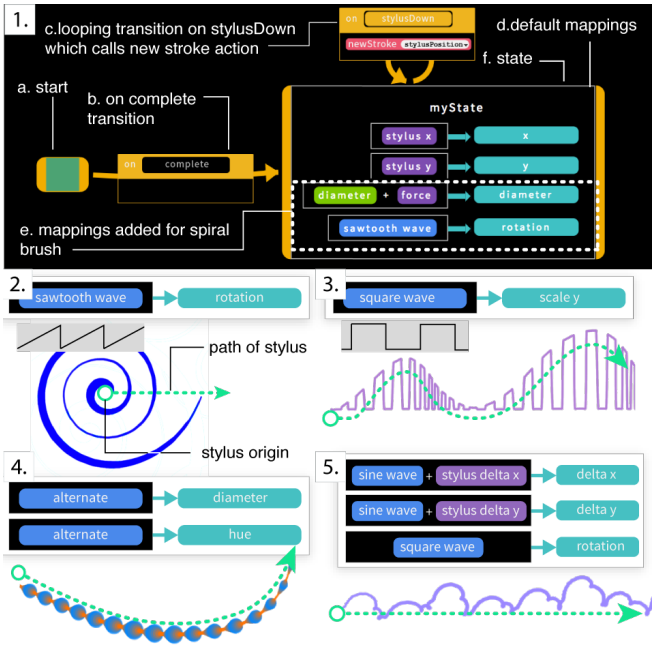

Figure 2. Single-state brushes: 1. Simple template with added mappings to generate spiral. 2-5. Additional single-state brushes.

SYSTEM DESCRIPTION

Dynamic Brushes (DB) is an integrated programming and drawing environment for creating personal drawing tools that translate manual input to procedural forms and patterns. DB was developed through an iterative process. We used an analysis of drawing techniques in procedural and manual artwork to create sample brushes with general-purpose languages. These samples informed the development of our programming model and interface. We tested and revised our system through informal sessions with artists and feedback from professional interaction designers. In the remainder of this section we describe the DB programming model and interface and illustrate its use through sample applications.

Programming Model

The DB programming model is organized around the creation of *brushes*—bitmap drawing tools that respond to external inputs and events. Each brush is an instance of a *behavior*—a class definition that describes brush functionality. Behaviors are composed of three components: *property mappings*, *states*, and *transitions*. **Property mappings** are declarative relationships between an *input*— incoming data, and an *output*—a brush property that is automatically updated when the input changes (fig 2-1, e). To support personal expressiveness in procedural artwork and preserve manual processes, the DB model uses stylus data as the primary input. Additional inputs include values generated by user-interface components and function generators (table: 1) which enable artists to manipulate how other brush properties are modified in response to stylus actions. Outputs include brush position, visual style, and geometric transformations (table: 2). Collections of mappings are organized within **states** and become active when the state is entered (fig 2-1, f). DB contains two specialized states: start, which serves as a setup action, and end, which destroys the brush when entered. States are activated and deactivated via event-driven, unidirectional **transitions** (fig 2-1,b). Transitions are triggered by stylus, spatial, geometric, and temporal events. Transitions can include actions— discrete methods executed when the transition occurs (fig 2-1, c).

In developing the DB programming language, our objective was to create a model that blended procedural generativity with manual data and events, allowing different artists to create different styles with the same brushes. We chose the combination of mappings, states, and transitions because this set provided an expressive representation for this integration while also offering a way to lower learning thresholds. Constraints are easy to use for simple relationships, yet can be confusing when chained together in complex relationships [43]. State machines offer a powerful way to describe reactive systems [3], and the combination of constraints and states has been demonstrated to simplify the creation of constraint-based interactive User Interface (UI) behavior [47]. Our system contributes to prior work through a new constraints-and-states programming model that enables artists to integrate procedural and manual art creation, while avoiding complex constraint-based programming.

Interface

The DB interface is comprised of two linked applications: a PC visual programming environment for behavior creation and a direct-manipulation tablet drawing interface where behaviors are initialized as brushes and used to create artwork (fig 1-1) [1]. The drawing interface is an iOS application running on an iPad Pro. The authoring interface is implemented in JavaScript using jsPlumb and CodeMirror [2] Communication between the applications is managed by a Node.js server.

To support continuous drawing, the DB interface preserves conventions from traditional stylus-based drawing tools, including a drawing canvas, layer panel, color-picker, property adjustment sliders and buttons for different pen modes. We also designed the drawing interface to be supported by, rather than dependent on, the programming interface, meaning that artists can begin work exclusively in the drawing interface with example brushes and transition to the programming interface when developing new brush behaviors.The programming interface includes an example menu, palette of behavior primitives, and a central scripting area. We developed DB as a visual language to support exploratory learning. Like block-languages [54], the DB palette provides access to primitives without an external reference. In addition, the environment is *live*, meaning as a behavior is edited, any active brushes in the drawing interface are automatically updated.

Artists program by dragging mapping inputs, outputs, and states from the palette into the scripting area. Transitions are created by dragging from the yellow handles of one state into another state. Each transition contains a panel with its event trigger. When initialized, transitions are triggered via an *on-complete* event, which occurs automatically after the state is entered. The event can be changed by dropping in a different event from the palette. Actions are also stored in the palette and can be added to a transition by dropping them in the box beneath the event. Events and actions that accept arguments contain a drop-down menu or numeric entry box. Artists can create behaviors in DB from scratch. To make starting programming approachable, however, we also included learning scaffolds in the form of example brushes and templates. Templates enable artists to explore procedural concepts incrementally by providing starting points for common design patterns in DB. The following sections demonstrate sample brushes that build on primary design patterns.

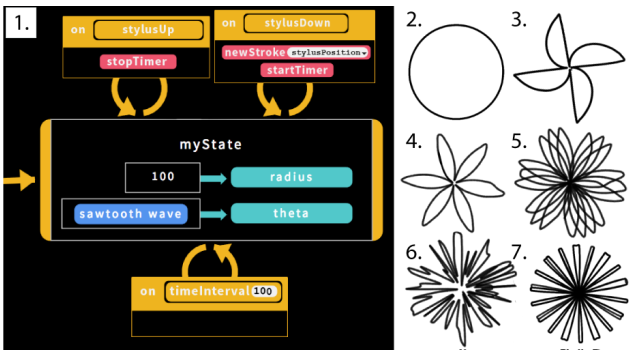

Figure 3. Automated regular forms: 1. Timer-driven behavior that draws regular forms through polar mappings. 2-7 Radial forms produced through different generator mappings to the radius output.

Simple Brushes

DB can be applied to the creation of simple programs with compelling effects. *Simple brushes* are derived from single state behaviors and create effects solely through property mappings. The *simple template* demonstrates this design pattern (fig 2-1). It contains a state ("default") that maps the stylus x and y inputs to the brush x and y outputs (fig 2-1, d). The default state is activated upon initialization through an *on-complete* transition (fig 2-1, b). A second looping transition, activated on a stylusDown generates a new stroke each time the stylus is touched to the canvas by calling the *new stroke* action in the transition (fig 2-1, c). This structure produces functionality comparable to a traditional bitmap brush, and provides a jumping off point for more complex functionality.

[1] See github.com/pixelmaid/DynamicBrushes for source.

[2] jsplumbtoolkit.com, codemirror.net. Accessed July 21st 2017.

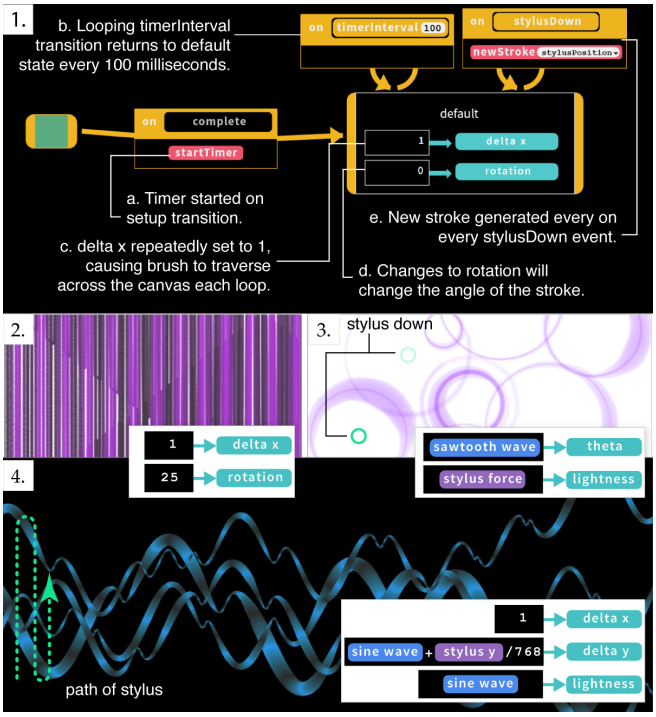

Figure 4. Automated Brushes: 1. Timer template, 2. Crosshatch brush, 3. Circle brush with diameter modulated by stylus force, 4. Wave brush with amplitude modulated by stylus y.

By mapping the rotation property to a *sawtooth* generator, the simple template can be modified to produce a brush that automatically draws spirals (fig 2-1, e). Brush strokes in DB are represented as a series of segments. The brush rotation property specifies the degree to which the current segment should be rotated relative to the stroke origin. Therefore, a generator input that linearly increases the rotation property as the brush position changes will incrementally shift the brush's heading and produce a spiraling stroke as the artist draws. Signal generators are common in music synthesizers [26, 13]. We adapt them in DB to augment manually executed strokes with periodic variation in color, position, and scale (fig 2).

Automated Brushes

One primary affordance of programming is the ability to automate processes. DB enables *automated drawing* through mechanisms to control timers and respond to temporal events. Each brush contains an internal timer that can be controlled by the *startTimer* and *stopTimer* actions. Transitions can be activated in response to timers through the *timerInterval* event, which triggers on a specified interval. The *timer template* provides a starting point for using timers to create automated brushes (fig 4-1). The template is similar to the simple template, meaning that the brush will begin to draw whenever and wherever the stylus is touched to the canvas, and will restart with each subsequent stylusDown event. The template also includes a startTimer action in the start transition (fig 4-1, a) and a second looping transition on the default state triggered on a timerInterval of 100 ms (fig 4-1,b). Unlike the simple template, the timer template "default" state contains a single mapping from an constant input to the *delta x* brush property. (fig 4-1, c). Deltas specify relative position by setting the distance the brush should move when the input is updated. The timerInterval transition continually activates the delta mapping, producing a brush that draws a line across the canvas at a one pixel per 100ms.

The timer template provides a starting point for automated drawing of regular patterns. For example, adding a mapping between a constant numerical input and the brush rotation output creates straight lines drawn on an angle. This enables the creation of grids and cross-hatch effects (fig 4-2). Timers also enable the creation of precise geometric shapes. Circles can be drawn automatically with the timer template by mapping radius to a constant, and the polar angle (*theta*) to a generator with a linear increase, like the sawtooth generator (fig: 3-1, 2). Mapping the radius to a generator input results in brushes that draw pinwheels, stars, and fans (fig: 3-3 through 7). Automated shape-drawing brushes create forms like those produced through textual creative coding languages, yet there is a key difference in how these shapes are initialized. Conventional programming languages initialize shapes with multiple function calls and indicate position via textual Cartesian coordinates which can be difficult to intuit. In DB, shapes are initialized and positioned through direct manipulation by touching the stylus in the desired location.

The DB model makes it easy to adjust the behavior of automated brushes with stylus actions and other forms of manual input. Mapping brush diameter to stylus force in the circle-drawing behavior enables the artist to modulate the line thickness along the circle's edge by pressing harder or softer on the stylus (fig 4-3). Mapping stylistic and geometric outputs to stylus and UI inputs allows the artist to use the stylus and their non-dominant hand to manipulate an automated brush, for instance by manually tuning the scale and hue of a regular form as it is being drawn. Stylus input can also be used to introduce variation into regular patterns. Mapping the delta y output to a mathematical expression that adds the sine wave generator to the stylus y delta enables the artist to change the depth of each wave as the brush draws across the canvas by moving the stylus (fig 4-4). Collectively, these examples demonstrate how DB blends manual control and procedural automation.

Spawning Behaviors

Manual art often involves a one-to-one relationship between input and output where for every gesture with a pencil or paintbrush, one mark is drawn. Unlike manual drawing, computers provide the opportunity to run multiple processes simultaneously, and procedural art is often produced through systems with multiple interacting agents. DB preserves this affordance by enabling artists to create brushes that generate *multiple autonomous drawing processes*. Using the *spawn* action enables one brush to generate any number of new brush instances. The spawn action includes two arguments: a dropdown menu that enables the artist to select from a list of existing behaviors, and a text entry input to specify the number of instances of that behavior to generate.

The *parent* and *child* templates provide a design pattern for spawn-based behaviors. The child template consists of a single state with two transitions to the start and end states. The ingoing transition calls the newStroke action. The outgoing transition occurs on stylusUp and destroys the brush. The

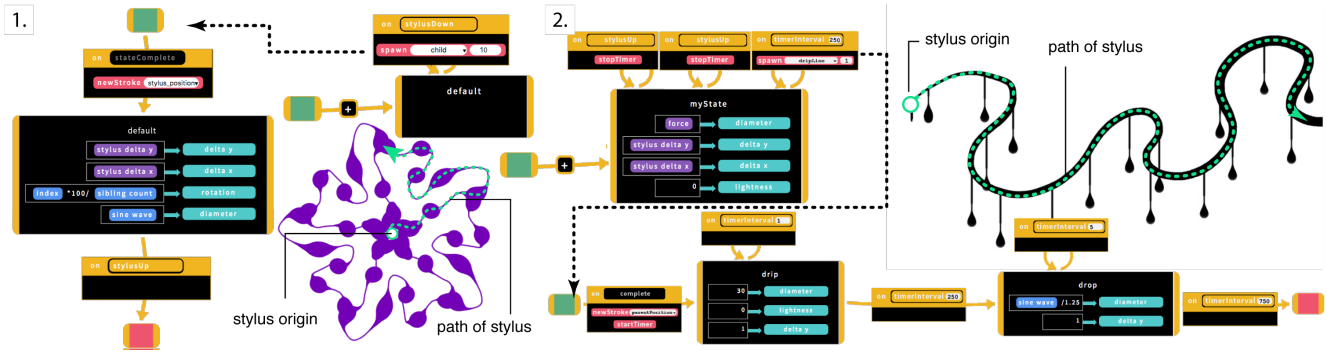

Figure 5. Spawning 1. Radial design produced through spawning and iterative rotation. 2. Drip Brush produced through spawning and timers.

parent brush acts as a mechanism for spawning multiple child brushes on a stylusDown event. In unison, these behaviors create 10 child brushes on each stylus-down event, which are removed on each stylus-up event.

Spawning can be combined with timers to create behaviors that generate multiple automated brushes. Figure 5-2 demonstrates an effect created by pairing the parent spawning template with a child behavior that draws on a timerInterval loop. As the parent brush draws, it spawns one child brush on a regular interval. Each child uses a timerInterval transition to draw a downward vertical line from the parent stroke. After 1500 ms, the child behavior transitions to a "drop" state which, maps the brush diameter to a sine-wave generator, and then in turn, increases then decreases the diameter of the stroke before transitioning to the end state. This creates a dripping effect.

Spawning is powerful enough to support behaviors that mimic professional procedural artwork. Figure 6 compares a procedural illustration by Zach Lieberman created in the C++ openFrameworks library to one created in DB. The DB piece is created by a behavior similar to the drip brush effect where automated child brushes are spawned on a *distance interval* event, ensuring regular spacing. The child brushes draw horizontally until they intersect with the parent line, or draw beyond the edge of the canvas. This is achieved through two transitions to the end state, one that occurs on an intersection event, and the other which occurs on a distance event.

Simple, automated, and spawning behaviors demonstrate how the DB model supports a high degree of procedural expressiveness with a small number of procedural concepts and properties. Furthermore, these applications demonstrate how DB can reproduce forms and patterns similar to those created with textual procedural tools while also adding manual control.

Dynamic Brushes Workflow

To demonstrate a sample workflow with DB, we return to our motivating example and describe how Taylor would use the system to create her desired tool. Taylor begins in the DB programming interface. She uses the basic template to create two new behaviors–"standard" and "mirrored". She adds a mapping in the mirrored behavior, setting the scale x property to -1. When Taylor transitions to the drawing interface and draws a curling line with the stylus, the brushes generated by her behaviors respond, producing a mirrored pattern of her original stroke (fig 1-2). To increase the complexity of design, Taylor creates copies of her first two behaviors named "standard-rotated" and "mirrored-rotated." She adds a rotation mapping to both copies set to 50. Taylor switches to the drawing interface, hides the layer with her first mirrored drawing, and creates a new blank layer to draw on. The combined brushes create a pattern with four strokes, mirrored on both x and y axes. Taylor is not yet satisfied with the result; however, modifying the rotation has given her an idea.

Returning to the programming interface, Taylor deactivates all behaviors except "standard-rotated." She drags index and sibling count generators from the palette (table: 1-brush hierarchy) into the rotation mapping and sets the rotation to index * 100 / sibling count. As a result, each child brush is rotated relative to its index of instantiation. Taylor then modifies the transitions of the standard-rotated behavior to correspond with the child design pattern (fig 5-1). Taylor creates a new behavior from the parent template and sets it to spawn five instances of the "standard-rotated" behavior. Finally, she returns to the drawing interface and adds a new layer. Now, as she draws, a complex radial pattern is produced with lines radiating out from the point at which she first touches the stylus to the canvas (fig 5-1). She explores variations of this behavior structure to produce a variety of radial and mirrored patterns (fig 1-3).

EVALUATION

The evaluation of DB sought to examine the system's expressiveness, learnability, and compatibility with manual art. We used a 2 week study with professional artists to evaluate DB's performance in extended practice. This approach built on prior methodology evaluating creative systems with a small number of participants over an extended period [23]. We selected evaluation criteria that aligned with our design goals:

Compatibility with manual practice: Are manual artists interested and engaged in DB? Do they consider it useful? Can artists make procedural modifications while drawing?

Expressiveness: Do artworks contain procedural and manual aesthetics? Can artists create works in their personal style?

Learnability: Is the system approachable? Can artists explore it independently? Does it aid in learning procedural concepts?

Evaluation Methodology

We commissioned 2 professional artists to use DB for 2 weeks and requested they create a minimum of 3 finished works during that time. We selected the graphic illustrator, Fish McGill

and the painter Ben Tritt. McGill was a professional illustrator and faculty at a fine-arts college. He had extensive experience in physical and digital illustration, some prior experience in web scripting, and had collaborated with procedural artists using Processing. McGill maintained a daily sketching practice, and was interested in incorporating programming with his drawing. Tritt was a professional oil and digital painter. His work included portraits and abstract compositions. Tritt had no prior programming experience but was interested in achieving greater control in digital painting. Both drew regularly with digital styluses. We interviewed McGill and Tritt and observed them at work one week before the study. These pre-interviews informed additional system adjustments, including improved layer transparency and the addition of a color picker and sliders for stroke weight and alpha. We updated the model with mapping inputs that corresponded with these additions.

During the study, we met with each artist separately, every 2-3 days based on their availability. We had 7 meetings with McGill and 5 with Tritt. The first 3 meetings lasted 2 hours apiece and were divided between training the artists and observing their use of DB. The first training covered simple behaviors, the second focused on multi-state behaviors and timers, and the third introduced spawning. In later meetings we reviewed participant artwork and discussed the artists' experiences. In the final meetings, both artists presented their completed works. We concluded with a group discussion.

We collected data through transcribed audio and video recordings of the meetings and observation sessions, written surveys, and participant artwork. Surveys contained attitudinal questions relating to our evaluation criteria, using 5-point Likert scales, with 5 as the optimal response. We instrumented DB to store in-progress artwork and brush behaviors while artists used the system. Artists also wrote short reflections after every working session, which we reviewed prior to each meeting. We allowed the artists to incorporate external software and tools in their work with DB to understand its cross compatibility with other forms of production. In the results, we distinguish between portions of artwork produced with DB and those produced by other means. Surveys, transcripts, observation videos, and artwork were analyzed with respect to our evaluation criteria. The majority of the results are qualitative, drawn from artwork analysis, survey responses, and discussions.

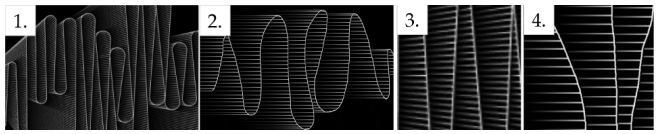

Figure 6. Comparison of Zach Lieberman's work, created with the openFrameworks C++ library (1, 3) vs. artwork in Dynamic Brushes (2,4).

Limitations

We compensated for our small sample size by selecting artists in different mediums and styles. Evaluations with additional artists would likely reveal additional insights. The variety of the results suggests our approach has broad relevance.

Results

The artists found the DB model challenging at first but developed competency and demonstrated a grasp of primary concepts by the end of the study. Tritt produced 6 finished portraits, and McGill produced upwards of 10 illustrations. The artists felt DB provided greater flexibility compared to other digital art tools, though it was better suited to the McGill's illustration than Tritt's painting, and both used motion graphics software to convert their DB drawings to animations.

Learnability: Both artists initially approached DB by trying to understand all aspects of the programming model, and struggled to grasp some concepts at first. McGill understood constraints and transitions in the first week but took longer to understand spawning. By the end of week 1, he transitioned from focusing on learning the model to focusing on making artwork and exploring new procedural concepts in the process. McGill described how this exploratory approach was similar to how he learned other drawing tools, and made his experience with DB more enjoyable. Tritt had greater difficulty learning some of procedural concepts. He initially struggled with the semantic difference between transitions and mappings. In spite of these challenges, Tritt also had moments of insight while using the tool. He described his feeling when he first understood the power of property mappings:

At first my brain didn't really understand being able to use [stylus] force for multiple things . . . what would that mean? ***And then I felt it.*** *The force could actually control two things or even three . . . lightness AND the diameter!*

Tritt said these insights made working with DB "addictive", and that he felt his "brain light up" as he learned the model.

Both artists felt that examples and templates aided in their learning process. In addition the artists immediately gravitated to the drawing interface. Both artists said the familiarity of the drawing interface increased their comfort with the system as a whole, and Tritt felt its simplicity allowed him to focus on painting. For McGill, the familiar drawing interface made him more willing to explore aspects of the programming environment. McGill and Tritt relied on layers as a mechanism to experiment with behaviors, using one layer as a scratch pad for testing and revising a brush, and other layers for drawing artwork once they were satisfied.

Compatibility with manual practice: Tritt and McGill had different reactions to the DB interface. Both saw value in separating programming and drawing; however, McGill disliked having to switch to the programming interface to perform simple adjustments, like modifying the parameters of the brush or switching to a different brush. To compensate, we added a control panel that enabled artists to select and activate different brushes in the drawing interface. These additions addressed some but not all of McGill's concerns. McGill disliked working on two devices by himself but enjoyed the setup when collaborating with another person. When working from home, McGill involved his son in his drawing. In these scenarios, one person drew on the tablet, while the other modified the behavior by changing the values of the expression or by dragging in new property mappings.

Tritt enjoyed the separation created between the two interfaces because it reflected his desire to create tools before or after making art, rather than during production. McGill and Tritt's

different attitudes towards tool creation were reflected in their use of brushes. McGill experimented with many different brushes. He started by creating a variety of textures and patterns with brushes that changed color on an interval, repeated strokes in a fan configuration, or produced a scribble-like pattern (fig 7, 2). In the second week, McGill used the spawning template to create a brush that drew multiple identical marks offset from the position of the stylus. He later modified it to incorporate a random jitter in the line of the child brush so that the resulting effect was two identical drawings with different line qualities. McGill named this brush the "fraternal twin brush," characterizing it as a way to draw as "one artist at two different periods in their life." He drew inspiration from how the cartoonist Charles Schulz's style changed as he aged.

Unlike McGill, Tritt primarily iterated on one type of brush. He began the study with an interest in precisely controlling the texture and "flow" of the brush. As a result, he initially worked to finely tune the relationship between stylus force, diameter, and alpha. Tritt requested assistance in modifying the brush to simulate qualities of a physical paintbrush. With assistance, he created a brush that continued drawing a stroke for one second after the stylus was lifted from the canvas at a speed and angle relative to the vector and speed of the stylus. Using this brush, Tritt attempted to simulate the physics of paint as the brush was flicked across the canvas; however, he was not able to refine the effect to the desired level. We added additional stylus inputs to improve Tritt's experience including stylus speed and heading. The speed property was effective in achieving a degree of physical brush simulation, but much of Tritt's interest centered on variable control over the hardness of the top and bottom edges of the brush.

Expressiveness: McGill used the repeating patterns he created in the first week as backgrounds and textures for other illustrations he created in DB. In the process, he built on themes from his prior work. He extended a series of illustrated composition notebooks he had produced prior to the study by using the scribble brush to create patterns for a new set of notebooks, and by adding typography in Illustrator (fig 7,1-2). In the second week, he used his twin-brush behavior to create a series of repeating illustrations. He described how he tweaked the distance and positioning between each brush so that the repeats were less obvious. After adding the jitter effect, he used the fraternal twin brush to design a typeface, with alternating smooth and rough letters (fig 7, 4-6). Tritt primarily focused on using the flicking brush to create painterly portraits (fig 8). This aligned with his prior work in oil portraiture.

Both Tritt and McGill created animated works in the final week of the study. McGill approached this by creating multiple variations of a drawing with his twin brush and using additional software to compose these variations into an animation. He described how DB improved the process of creating hand-drawn animations:

[Animations are] something I've done before," but with [DB] it's much more suitable and enjoyable to draw with.

McGill also converted his alphabet into a series of animated letters. He used his fraternal twin brush to simultaneously create two variations of each lette,r and then animated them to flip back and forth between the variations (fig 7-4). Tritt also created animations, but unlike McGill, Tritt's animations reflected the progression of a painting through different brush iterations and different stages in the study. He described how the animations stemmed from a desire to document how the procedural aspects of the painting changed over time.

DISCUSSION

We motivated DB with the idea that tool creation could help artists combine manual and procedural expression. Here we discuss the outcomes of our approach by examining how the DB model aligned with different types of manual practice; the ways in which the DB interface supported learning, creation, and reflection; and the potential for tool making to encourage critical analysis and agency in software production.

Aligning Procedural Models with Practice

Both artists created personal brushes during the study, yet McGill was more satisfied with the results of his brushes than Tritt. In part, this may reflect McGill's prior experience with web programming; however, it also suggests that DB's model was better aligned with McGill's process. Tritt described how his paintings were made up of "very simple things," combined in "very complex ways." His manual process involved skillful manipulation of spots, edges, and transparency. In contrast, McGill said his drawings were all comprised of three line types: "zigzags," "wiggly lines," and an "up and down."

McGill's line types are evocative of DB's generators, which he used extensively in his brushes. His success demonstrates how DB's model is well suited to manipulating manual art at the level of form and composition. By comparison, Tritt's description of spots and edges reflected his desire to control texture and transparency at a lower level. DB's current brush properties are not effective at providing precise control over mark texture and edge quality. It is possible that the same state-based model, with different mapping primitives could support Tritt's painterly applications. A future study that tests a version of DB with higher-level representations of stylus physics and lower-level representations of bitmap textures would provide a starting point for testing this idea. Furthermore, the demonstration of the current limitations of DB is useful in and of itself, by contributing to the understanding of how procedural tools might better support a variety of manual practices. Our failure to anticipate the limitations of DB primitives for painting suggests that future design frameworks for procedural tools should also examine how the level of abstraction of the programming model aligns with the types of control and mark-making exhibited in a given domain.

Informing Decisions and Supporting Reflections

The evaluation demonstrated how, in 2 weeks, artists were able to develop sufficient competency in DB to produce polished outcomes. Furthermore it showed how DB helped to foster an understanding of general procedural concepts, such as the arbitrary nature of constraints. Despite these successes, evidence suggests that improvements in the way DB visualizes programs and communicates procedural functionality may improve artists' abilities to learn and apply the system.

Figure 7. McGill's artwork. 1. Prior illustrations with pen and ink, 2. Continuation of pen and ink series with patterns drawn in dynamic brushes and typography from Illustrator, 3. Repeating pattern drawn with "scribble" and "stereo" brushes, 4-5. Alphabet produced with "fraternal twin" brushes

Communication of DB programming model functionality could be revised to be easier to understand and more compatible with manual practice. Both McGill and Tritt initially struggled with understanding the functionality of some inputs. They also disliked having to read textual explanations while working. Tritt described how he transitioned to a non-verbal mindset when painting. We revised generator tool-tips to display animated wave-forms rather than text, which both artists found easier to understand. The generator revision suggests procedural systems that communicate primitive functionality primarily through icons and motion, rather than text, could be easier for manual artists to interpret while they are engaged in drawing. A similar approach could help artists to better predict the effects of their edits. At times we noticed artists using trial and error to set the desired value for a mapping. This challenge could be mitigated through a mechanism for inspecting the current value of mapping inputs and outputs, and with visualizations demonstrating how the outputs change over time. When possible, these values could be communicated visually.

Finally, DB could improve by aiding artists in documenting and examining brush execution. It is not incidental that both artists created animations that reflected different iterations of a brush. Tritt described how DB's brush creation naturally had implications for more general forms of reflection. His animations stemmed from his desire to document his process:

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

Modifying DB to include functionality to record, visualize and simulate brush execution could both aid in the reflection process. In addition, it could offer a method for debugging brushes by enabling artists to step through the execution of a brush in response to pre-recorded input data. Finally, a recording mechanism would provide the opportunity to formally extend dynamic brushes as a tool for animation.

Integrating Tool and Art Making

By design, DB encourages artists to integrate art production and tool production. Tool production involves different mindsets and approaches than artwork creation; therefore, we were unsure how manual artists would react to their integration. The procedural artists we interviewed described how building effective tools required them to think in general terms and consider multiple scenarios that could interfere with making specific artifacts. Our own experiences simultaneously building DB and creating sample artwork highlighted similar tensions. Despite these tensions, our evaluation demonstrated that while procedural tool production is challenging, it is also a process that manual artists can enjoy and benefit from. McGill's use of DB to extend his prior illustration work and his development of procedural behaviors that reflected his ideas about artistic practice demonstrate how integrating tool making and manual drawing can connect new forms of expression with established practices and values. Furthermore, feedback from McGill and Tritt in our final discussion suggests that tool production may also foster critical evaluation of digital software conventions and encourage artists to consider alternatives. McGill described how working with DB made him less satisfied with Photoshop:

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." [DB] feels more like a drawing experience than other tools because it's combining just enough of the good stuff with more complex, robust stuff.*

Tritt went even further, questioning the overall justification for digital tools. His use of DB led him to wonder if instead of simulating physical media, digital tools were perhaps more powerful when applied to forms of creation not possible with "analog" media. Overall, both artists left the study with a strong desire to reshape the digital tools they used.

Technological advances suggest opportunities for democratized software development [65], yet many communities are still poorly served by software created by others [2, 27]. Roque demonstrated how creative programming workshops helped people who were underrepresented in technology production to develop both technological empowerment and community connections [57] Our research suggests that relevant and approachable tools for *software development*, targeted at groups of people who are underrepresented in technology production, may contribute to addressing forms of technological inequity.

RELATED WORK

Our research is inspired by creative coding tools, learnable programming systems, and procedural direct manipulation.

Creative Coding Frameworks

Our research was galvanized by the study of specialized textual programming libraries for procedural and interactive art [50, 37, 6]. These libraries are powerful and expressive [51, 36], yet they retain many of challenges of general-purpose programming languages. Visual programming can address some of the challenges of textual programming while retaining expressiveness [44]. We noted that visual languages for artists often use a dataflow representation. Programs are structured as directed graphs that filter and operate on three-dimensional models, audio, images, and video [13, 49, 14, 17]. Dataflow systems

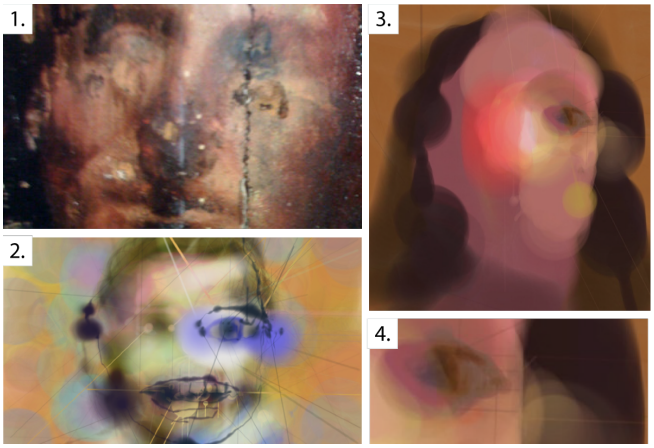

Figure 8. Tritt's Art. 1. Prior work in oil, 2-3. Portraits drawn in Dynamic Brushes with "flicking" brush, 4. Closeup of "flicking" effect.

are expressive, but often ill-matched for describing cyclic systems [25]. We therefore chose a visual state-based paradigm for DB so that artists could create behaviors that cycle through different functionality on discrete drawing events.

Integrated programming models and interfaces can support specific applications and workflows. In the space of visual design, DressCode [24], PIM [39], and DesignScript [1] integrate textual programming, with linked direct-manipulation environments to simultaneously support different design approaches through different interfaces. Loi et al. developed a programming model for 2D texture generation with three levels of operators targeted at different portions of the texture-generation process [38]. Juxtapose improves the process of iterative interaction design through parallel source editing and tuning of parameters with sliders [18]. Interstate simplifies user-interface programming through a combination of declarative constraints, state machines, and a live, visual notation [47]. Reactable [26] supports real-time procedural music performance through a flow-controlled programming language with physical blocks representing modular synthesizer components. We also sought to leverage domain knowledge to coordinate the development of a programming model and programming environment uniquely suited to the needs of manual artists. Therefore, we focused on programming abstractions that preserved the expressiveness of manual drawing and embedded these abstractions in an interface with aspects that resembled conventional digital drawing software.

Learnable Programming Languages

In developing DB, our goal was to enable artists to achieve practical ends while simultaneously engaging in new forms of thinking and learning. To achieve this goal, we examined prior research on supporting new programmers in creative forms of learning. When creating learnable creative languages for young people, Resnick and Silverman argue that *a little bit of programming goes a long way*, and they focus on systems with a minimal number of carefully selected programming concepts. Designing programming tools that are *tinkerable* can facilitate continuous exploration and ease of entry by providing access to computational concepts through re-configurable modular components [55]. The Scratch programming language demonstrates how this approach provides easy entry points, while also supporting a diversity of projects and applications [54]. To provide artists with an approachable, tinkerable platform for procedural art, DB is structured on small number of programming primitives that can be recombined in different ways.

Procedural Direct Manipulation

Visual programming languages reduce some of the challenges of programming, but they still require artists and designers to work through an abstract description rather than a concrete artifact [61]. An alternative approach involves creating and manipulating procedural relationships through direct manipulation. SketchPad demonstrated parametric relationships that could be described by selecting and manipulating geometric shapes [59]. More recently, Victor demonstrated sample systems with applications in game development, interactive animation, and data visualization [62, 63, 64]. Kitty [28] and Skuid [29] reduced the challenge of creating interactive motion graphics and animation through direct manipulation of a relational graph that is superimposed on the artwork and Apparatus [58] enables the creation of interactive diagrams through an integration of a direct manipulation editor and declarative textual expressions. In two-dimensional graphic art, Hoarau and Conversy demonstrate graphic design tools that enable designers to indicate dependencies between object properties and use master objects to perform global updates [21]. Pampliset represents parameter values for bitmap transformations as image layers, which can be retroactively adjusted to propagate changes across a composition [5]. Para supports procedural art through the combination of visually represented constraints, lists, and declarative duplication [23]. Like these prior systems, DB relies on the convenience and brevity of declarative constraints to describe mappings between external data and drawing output; however, our overall programming model is fundamentally different. Procedural direct manipulation is well suited to forms of design that emphasize relationships between visual entities. Our goal involved the more abstract task of tool development. We therefore incorporated an external programming representation into DB that was better suited to representing abstract interactive behaviors.

CONCLUSION AND FUTURE WORK

Motivated by conversations with artists, we created Dynamic Brushes, a visual programming and drawing environment for blending manual and procedural production through personal tool creation. Our evaluation demonstrated that tool development can be engaging for manual artists while providing opportunities to extend their established practice and style. We see future research opportunities in developing mechanisms to support artists in visualizing, recording and inspecting drawing behaviors, and experimenting with different brush primitives to support procedural control over brush textures and physics. Overall, we are excited about the potential of domain-specific programming languages to foster broader participation in creative system development.

ACKNOWLEDGMENTS

Thanks to the Dynamic Medium and Lifelong Kindergarten research groups, and Golan Levin. Special thanks to Fish McGill and Ben Tritt.

REFERENCES

1. Robert Aish. 2012. DesignScript: origins, explanation, illustration. In *Computational Design Modelling*. Springer, 1–8.
2. Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine Bias:There 's software used across the country to predict future criminals. And it 's biased against blacks. *ProPublica* (May 2016). https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.
3. C. Appert and M. Beaudouin-Lafon. 2008. SwingStates: Adding State Machines to Java and the Swing Toolkit. *Softw. Pract. Exper.* 38, 11 (Sept. 2008), 1149–1182. DOI: http://dx.doi.org/10.1002/spe.v38:11
4. John Berger. 2008. Drawing. In *Selected Essays of John Berger*, Geoff Dyer (Ed.). Knopf Doubleday Publishing Group. https://books.google.com/books?id=dYIUiSrKVccC
5. Alan F. Blackwell. 2014. Palimpsest: A layered language for exploratory image processing. *Journal of Visual Languages Computing* 25, 5 (2014), 545 – 571. DOI: http://dx.doi.org/10.1016/j.jvlc.2014.07.001
6. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. DOI: http://dx.doi.org/10.1109/TVCG.2011.185
7. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 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)*. ACM, New York, NY, USA, 513–522. DOI: http://dx.doi.org/10.1145/1753326.1753402
8. Leah Buechley and Benjamin Mako Hill. 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)*. ACM, New York, NY, USA, 199–207. DOI: http://dx.doi.org/10.1145/1858171.1858206
9. Ravi Chugh, Jacob Albers, and Mitchell Spradlin. 2015. Program Synthesis for Direct Manipulation Interfaces. *CoRR* abs/1507.02988 (2015). http://arxiv.org/abs/1507.02988
10. Cinder. 2017. Cinder:About. (2017). https://libcinder.org/about.
11. Cecile Crutzen and Erna Kotkamp. 2008. *Object Orientation*. MIT Press.
12. M. Csikszentmihalyi. 2009. *Flow: The Psychology of Optimal Experience*. Harper Collins.
13. Cycling-74. 2016. Max. (2016). http://cycling74.com/products/max.
14. Scott Davidson. 2007. Grasshopper. http://www.grasshopper3d.com. (2007).
15. Nettrice Gaskins. 2017. Machine Drawing: Shantell Martin and the Algorist. (2017). http://magazine.art21.org/2017/07/06/machine-drawing-shantell-martin-and-the-algorist.
16. Emily Gobielle and Theo Watson. 2010. Here to There: Poster Series for Children. (2010). http://design-io.com/projects/HereToThere/.
17. Experimental Media Research Group. 2004. NodeBox. (2004). http://www.nodebox.net.
18. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 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)*. ACM, New York, NY, USA, 91–100. http://doi.acm.org/10.1145/1449715.1449732
19. B. Harvey. 1991. Symbolic Programming vs. the A.P. Curriculum. *The Computing Teacher* 56 (February 1991), 27–29.
20. Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 379–390. DOI: http://dx.doi.org/10.1145/2984511.2984575
21. Raphaël Hoarau and Stéphane Conversy. 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)*. ACM, New York, NY, USA, 1937–1946. DOI: http://dx.doi.org/10.1145/2207676.2208337
22. Jennifer Jacobs and Leah Buechley. 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)*. ACM, New York, NY, USA, 1589–1598.
23. Jennifer Jacobs, Sumit Gogia, Radomír Měch, and Joel R. Brandt. 2017. Supporting Expressive Procedural Art Creation Through Direct Manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6330–6341. DOI: http://dx.doi.org/10.1145/3025453.3025927
24. Jennifer Jacobs, Mitchel Resnick, and Leah Buechley. 2014. Dresscode: supporting youth in computational design and making. In *Constructionism*. Vienna, Austria.
25. Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *Comput. Surveys* 36, 1 (March 2004), 1–34. DOI:http://dx.doi.org/10.1145/1013208.1013209

26. Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. 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)*. ACM, New York, NY, USA.

27. Matthew Kay, Cynthia Matuszek, and Sean A. Munson. 2015. Unequal Representation and Gender Stereotypes in Image Search Results for Occupations. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3819–3828. DOI: http://dx.doi.org/10.1145/2702123.2702520

28. Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: Sketching Dynamic and Interactive Illustrations. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 11.

29. Rubaiat Habib Kazi, Tovi Grossman, Nobuyuki Umetani, and George Fitzmaurice. 2016. SKUID: Sketching Dynamic Drawings Using the Principles of 2D Animation. In *ACM SIGGRAPH 2016 Talks (SIGGRAPH '16)*. ACM, New York, NY, USA, Article 84, 1 pages. DOI:http://dx.doi.org/10.1145/2897839.2927410

30. Scott R. Klemmer, Björn Hartmann, and Leila Takayama. 2006. How Bodies Matter: Five Themes for Interaction Design. In *Proceedings of the 6th Conference on Designing Interactive Systems (DIS '06)*. ACM, New York, NY, USA.

31. Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC '04)*. IEEE Computer Society, Washington, DC, USA, 199–206. http://dx.doi.org/10.1109/VLHCC.2004.47

32. Golan Levin. 2003. Essay for Creative Code. (2003). http://www.flong.com/texts/essays/essay_creative_code

33. Golan Levin. 2010. Yellowtail. (2010). http://flong.com/projects/yellowtail.

34. Golan Levin. 2015. Foreword: For Us, By Us. In *EYEO: Converge to Inspire. 2011-2015*.

35. Zach Lieberman. 2009. Drawn. (2009). http://thesystemis.com/projects/drawn/.

36. Zach Lieberman. 2014. Philosophy. In *ofBook, a collaboratively written book about openFrameworks*. http://openframeworks.cc/ofBook/chapters/of_philosophy.html.

37. Z. Lieberman, T. Watson, and A. Castro. 2015. openFrameworks. (2015). http://openframeworks.cc/about.

38. Hugo Loi, Thomas Hurtut, Romain Vergne, and Joelle Thollot. 2017. Programmable 2D Arrangements for Element Texture Design. *ACM Trans. Graph.* 36, 3, Article 27 (May 2017), 17 pages. DOI: http://dx.doi.org/10.1145/2983617

39. Maryam M. Maleki, Robert F. Woodbury, and Carman Neustaedter. 2014. Liveness, Localization and Lookahead: Interaction Elements for Parametric Design. In *Proceedings of the 2014 Conference on Designing Interactive Systems (DIS '14)*. ACM, New York, NY, USA.

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

41. W.J. Mitchell. 1990. *The Logic of Architecture: Design, Computation, and Cognition*. MIT Press, Cambridge, MA, USA.

42. L. Mumford. 1952. *Art and Technics*. Columbia University Press.

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

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

45. Erik Natzke. 2012. Cloud Art Process. (2012). http://vimeo.com/69323991.

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

47. Stephen Oney, Brad Myers, and Joel Brandt. 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)*. ACM, New York, NY, USA.

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

49. Miller Puckette. 1988. The patcher. In *Proceedings of the 1988 International Computer Music Conference. San Francisco*. International Computer Music Association.

50. C. Reas and B. Fry. 2004. Processing. (2004). http://processing.org.

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

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

53. Jasia Reichardt. 1969. *Cybernetic serendipity: the computer and the arts*. Praeger.

54. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009).

55. M. Resnick and E.O. Rosenbaum. 2013. Designing for Tinkerability. In *Design Make Play: Growing the Next Generation of STEM Innovators*, M. Honey and D. Kanter (Eds.). Routledge.
56. David Roedl, Shaowen Bardzell, and Jeffrey Bardzell. 2015. Sustainable Making? Balancing Optimism and Criticism in HCI Discourse. *ACM Trans. Comput.-Hum. Interact.* 22, 3 (June 2015).
57. R. Roque, K. Lin, and R. Liuzzi. 2016. "I'm not just a mom": Parents developing multiple roles in creative computing.. In *12th International Conference of the Learning Sciences*.
58. Toby Schachman. 2015. Apparatus: a hybrid graphics editor / programming environment for creating interactive diagrams. In *Strange Loop*.
59. Ivan E. Sutherland. 1964. Sketchpad a Man-Machine Graphical Communication System. *Transactions of the Society for Computer Simulation* 2, 5 (1964), R–3–R–20. DOI:http://dx.doi.org/10.1177/003754976400200514
60. S. Turkle and S. Papert. 1992. Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior* 11, 1 (March 1992).
61. B. Victor. 2011. Dynamic Pictures. (2011). http://worrydream.com/DynamicPicturesMotivation.
62. B. Victor. 2012a. Inventing on Principle. In *Proc. of the Canadian University Software Engineering Conference*.
63. B. Victor. 2012b. Stop Drawing Dead Fish. In *ACM SIGGRAPH 2012 Talks (SIGGRAPH '12)*.
64. B. Victor. 2013. Drawing Dynamic Data Visualizations (Talk). (2013). http://vimeo.com/66085662.
65. E. von Hippel. 2005. *Democratizing Innovation*. MIT Press.
66. Amit Zoran and Joseph A. Paradiso. 2013. FreeD: A Freehand Digital Sculpting Tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA.