Supporting Visual Artists in Programming through Direct Inspection and Control of Program Execution

Jingyi Li[1], **Joel Brandt**[2], **Radomír Měch**[2], **Maneesh Agrawala**[1], **Jennifer Jacobs**[3]
[1]Stanford University [2]Adobe Research [3]University of California Santa Barbara
{jingyili, maneesh}@cs.stanford.edu, {jobrandt, rmech}@adobe.com, jmjacobs@ucsb.edu

ABSTRACT
Programming offers new opportunities for visual art creation, but understanding and manipulating the abstract representations that make programming powerful can pose challenges for artists who are accustomed to manual tools and concrete visual interaction. We hypothesize that we can reduce these barriers through programming environments *that link state to visual artwork output*. We created Demystified Dynamic Brushes (DDB), a tool that bidirectionally links code, numerical data, and artwork across the programming interface and the execution environment—i.e., the artist's in-progress artwork. DDB automatically records stylus input as artists draw, and stores a history of brush state and output in relation to the input. This structure enables artists to *inspect* current and past numerical input, state, and output and *control program execution* through the direct selection of visual geometric elements in the drawing canvas. An observational study suggests that artists engage in program inspection when they can visually access geometric state information on the drawing canvas in the process of manual drawing.

Author Keywords
Creativity support tools; Visual art; Programming

CCS Concepts
•**Human-centered computing** → **Human computer interaction (HCI);** *User interface design;*

INTRODUCTION
The relevance and power of programming for visual artists have led to the development of *creative coding systems*: symbolic programming languages for art and design [29]. Creative coding systems integrate the expressiveness of general-purpose programming languages with the vocabulary of art and design through domain-specific abstractions for geometry and visual style [30]. Creative coding enables forms of visual art not possible with traditional tools, yet it poses challenges for visual artists [16, 30]. Although specialized for art and design applications, creative coding languages impose the same workflows as programming languages for general purpose software development—they require understanding abstract representations and working in a highly structured manner. Using such languages requires artists to adopt approaches from software development, but current creative coding systems lack the functionality to aid artists in understanding and adapting these abstractions and workflows [39].

CHI'20, April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04…$15.00
DOI: https://doi.org/10.1145/3313831.3376765

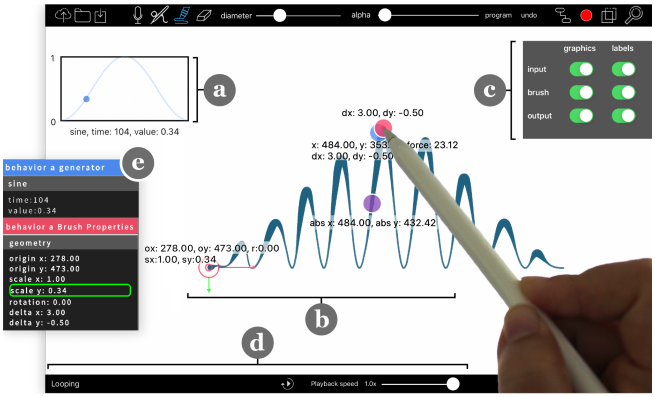


Figure 1. DDB links visual inspectors in the drawing/execution environment (this figure) with numerical inspectors in the programming environment (Figure 2). The drawing environment features (a) visualizations of synthetic input streams called generators, (b) iconography on top of the artwork that displays geometric properties, (c) inspection toggles, and (d) controls for looping manual input. Here, artist selects a point on their drawing which updates the program state in both the visual and numerical inspectors (e, shown in greater detail in Figure 2).

The challenges of understanding programming manifest in specific ways for visual artists. Artistic production is often an iterative process where ideation and execution are tightly coupled [5], and non-objective *exploration* enables artists to react and plan while in the process of producing finished work [3]. In comparison, traditional forms of software development emphasize linear working structures with predefined goals [11]. Working with creative coding systems often requires a similarly structured process, which can be challenging for artists who create by reacting to changes [38]. Visual artists are also drawn to *visual* and *manual* tools and media, ranging from paint and brushes to digital styluses and direct-manipulation software. These media allow artists to process information through observing and manipulating concrete graphic elements [9]. In comparison, symbolic programming requires manipulating a representational *description* of the work [38]. Abstract, representational tools are incredibly powerful, but they can limit epistemic action and thinking through embodied engagement [18]—forms of cognition that are often central to visual art practice.

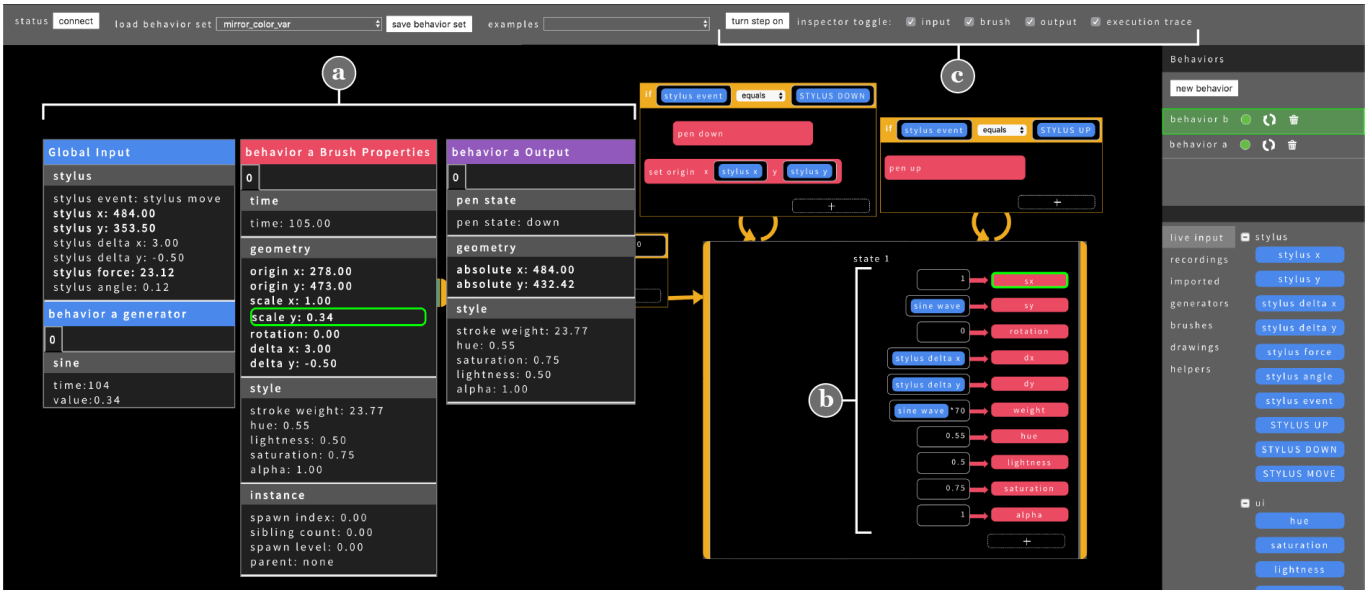


Figure 2. Continuing from Figure 1, the DDB programming environment features (a) numerical inspectors for input, brush state, and output, (b) the program code, and (c) controls for stepping through program execution and toggling the numerical inspectors on and off.

We aid visual artists in leveraging the opportunities of creative coding. Our analysis of the ways artists think and work with concrete visual media lead to our hypothesis: We can support visual artists in understanding creative coding languages with environments that provide explicit links between *manually manipulating visual artwork* and *observing program functionality and state*. To explore this hypothesis, we developed a new creative coding programming environment that provides graphic representations of manual input, program state, and program output, overlaid on artwork in the drawing canvas (Figure 1). By selecting these graphic elements or the artwork itself, artists can inspect abstract data and symbolic code as well as control program execution.

Our work extends the Dynamic Brushes programming language [16]—a visual creative coding system that is designed around manual stylus input—with an inspection and execution control environment. The inspection environment spans across a programming interface and a drawing environment; numerical data shown on the program inspector have corresponding concrete visualizations on the drawing canvas. Artists can access information about the program while they are drawing, and highlight relevant code and data by touching artwork on the canvas. Artists can control the program execution by manually stepping through state transitions and data mappings, or by rapidly looping input data through the system to quickly see the effects of their program changes. Overall, the combination of looping and direct-selection features allows artists to rapidly transition from drawing to inspection and back at arbitrary points in the art creation process. Our specific contributions are as follows:

1) We present a system that connects abstract data, state, and code in a symbolic programming environment with visualizations of input, state, and the corresponding artwork output in a direct-manipulation execution environment called Demystified Dynamic Brushes (DDB).

2) We demonstrate how displaying input, state, and output on the canvas allows artists to inspect program state while drawing. We also show how looping manual input data enables artists to observe the effects of code changes at a rate that corresponds with the speed of manual creation.

3) Our observational evaluation of DDB reveals that artists analyze programs when state is visually shown in the drawing canvas and displayed in a format that aligns with the speed of manual drawing. We present design principles for programming support tools for artists that deviate from assumptions that traditional software debugging tools make.

BACKGROUND AND RELATED WORK

Our research is informed by the opportunities and challenges of current creative coding tools, prior work in end-user programming and debugging, and key practices in visual art.

Opportunities and Challenges of Creative Coding

Programming offers many powerful opportunities for visual artists. General purpose programming languages enable visual artists to apply computational automation [7], abstraction [26], simulation [34], and interaction [25] to the creation of aesthetic visual content. Recognition of the creative applications of programming has led to the development of *creative coding systems*: textual languages like openFrameworks [23], Processing [31] and p5.js [27], as well as visual ones like Max [1], vvvv [40], and Grasshopper [36]. These languages provide artists with domain-specific abstractions for generating and transforming images, video, geometry, and style [30] while retaining access to general purpose programming operations, such as loops, lists, conditionals and functions [32]. The combination of art-oriented abstractions and general purpose programming makes creative coding tools extremely expressive; however, artists new to programming must learn many unfamiliar concepts before making work [17]. Furthermore, a lack of general programming knowledge can limit aesthetic range

because artists can resort to making minor modifications to established algorithms rather than writing new programs [41]. Our objective is to address challenges artists face when learning and understanding code through automatically visualizing program state and input data in relation to the artwork, and by letting artists control their program execution in the drawing environment.

Supporting Program Comprehension

Effective programming in any domain is dependent on comprehension [8]; programmers must interpret the vocabulary of the programming language, understand program execution, and identify program state [39]. Existing programming environments for software development, web scripting, and data visualization aid in this process by connecting output to code. For example, *Theses* uses always-on visualizations to proactively prevent code misconceptions [21] and *Whyline* [19] enables programmers to debug in the execution environment by selecting from questions about program output. *Telescope* [13], *Dinah* [4], and *Vega*'s in-situ visualizations [14] let programmers select graphic elements to highlight the corresponding example code. To help programmers quickly debug in interactive scenarios, *Timelapse* records and replays interactive inputs [4]. DDB builds on these support features from general software development with its own forms of output-based inspection, input recording, and playback that function across a visual programming language and a direct-manipulation drawing environment. We use reified [2] graphic representations of manual input, brushes, and stroke output to communicate program input and state in the drawing environment; users can select these elements to inspect program state.

Programming support tools for artists remain relatively unexplored. Processing [33] contains stepping and breakpoint features that mirror general-purpose software debugging tools. Yet stepping alone can pose challenges, even for experienced programmers [22]. DDB is in part inspired by Victor's ideas for adapting Processing for visually-oriented programmers through visualizations of relevant state and control flow [39]. Some creative coding languages contain features for probing signal data [6] and visualizing data structure [20]; however, these features are only available when manipulating the representational programming language. Direct manipulation programming systems like Sketch-n-Sketch [12] let artists manipulate abstract vector graphics with widgets in the execution environment. In contrast, DDB provides state information and execution control features that can be activated while manipulating symbolic code *and* while engaged in manual drawing in the direct manipulation execution environment.

Manual Manipulation and Exploration in Visual Art

Manual expression is both a means of production and a mechanism for cognition in visual art. Artists create work and process information through concrete visual tools and media (e.g., paint and paintbrushes or direct manipulation software) [28]. Many artists solve problems and think through alternatives by manually manipulating graphic elements [9]. Manual engagement also aids artists, who often approach work from a playful, non-objective mindset, in generating and refining ideas [37]. Artists therefore rely on tools that enable exploring alternatives [5], reactive creation [38], and direct engagement with materials [15] to determine the form of the final artwork. In exploration, the speed of manual expression (e.g., sketching) is critical in helping artists guide their iterations [3].

A variety of software tools seek to aid artists and designers in exploratory practices while programming. *Juxtapose* [10] supports iterative interaction design through parallel source editing and parameter tuning, and *Gemini* [43] enables the parallel editing and merging of generative patterns. More generally, Resnick and Rosenbaum [35] describe how *tinkerable* programming can facilitate continuous exploration through reconfigurable components. These works highlight how quick experimentation helps artists and designers generate ideas and refine their work while programming.

Other research in lowering barriers to creative coding for manual artists have focused on using direct manipulation to describe procedural relationships (i.e., constraints, duplication, and inheritance) in data visualization [42, 24] and generative art [17]. These systems eliminate the need to code with a symbolic language, but are often less computationally expressive than symbolic creative coding systems. We do not bypass symbolic coding, but rather develop features that let artists inspect their code and abstract data in more familiar representations—concrete visualizations on top of their drawings.

DESIGN GOALS

To inform the design of our inspection and execution control environment, we combined established principles in computer programming support [39] and manual interaction [5, 9, 18] with observations from a formative study on how artists currently approach program comprehension.

Formative Study

We compared the methods of 2 visual artists without programming experience to those of 2 computer science graduate students who were experienced using traditional software debugging tools. None of the participants were familiar with Dynamic Brushes. Participants used an early version of DDB with only the numerical inspectors on the programming interface and a basic execution step-through feature, baselines chosen since they resembled features from general-purpose software debugging tools. Participants worked with a brush program that mapped `sawtooth*360` to rotation, creating a spiral. We asked participants to reason out loud about how the program was producing the spiral, and observed their interactions with the system that facilitated their reasoning.

Design Goals

Our formative study observations provided preliminary insight for how we could apply principles from computer programming support and manual interaction to our specific domain of manual artists. In contrast to the CS graduate students who spent their time reading numbers in the program inspector to mentally calculate the spiral, artists made small program changes and then redrew strokes to guide their understanding of the program. We detail observations alongside our design goals below:

(1) Automatically show manual input in relation to program state and output: The artists, who were new to programming but familiar with digital drawing software, struggled to understand the difference between stylus input and brush behavior, since programming allows arbitrary mappings of input. This suggests that interface elements that enable continuous, visual inspection of input, state, and output data could help artists understand how their inputs translate to numbers that map to program state and produce visual output [39].

(2) Enable epistemic action: Artists reasoned about program functionality while drawing, observing how changes in stylus position, speed, and pressure affected the artwork. This process suggests that artists think about program functionality through manual drawing, a notion that aligns with the process of epistemic action [18, 9], and lead us to speculate that programming environments could support artists in directly manipulating input and artwork to facilitate their reasoning about programs.

(3) Provide visual and interactive representations of program behavior: Artists tried to understand program behavior by examining drawing output and avoided looking at the symbolic program unless directed. Artists stated they focused on the drawing because they were unaccustomed to programming. This aligns with past observations that while visual artists are familiar with concrete visual media, many are new to reading and manipulating numerical data [28]. Therefore, we believe that interactive representations in the drawing itself could aid artists in engaging with symbolic programming by providing explicit links to numerical state and code.

(4) Enable rapid transitions from exploring to authoring: Artists explored alternatives by making code changes, creating a new canvas layer, and then drawing, which created significant delays between editing and observing results. Visual artists generally work quickly, iteratively, and reactively [3]. This suggests that programming tools that integrate manual drawing require features that let artists edit code, observe results, and return to manual drawing in rapid succession.

SYSTEM DESCRIPTION

DDB is a programming and drawing environment designed to support artists in understanding relationships between manual drawing input (drawing), programs that operate on that input (brushes), and the visual output of the programs (strokes) in the Dynamic Brushes programming system.

The Dynamic Brushes symbolic programming model, described in detail by Jacobs et al. [16], is organized around creating *brushes*— procedural drawing behaviors that respond to manual drawing. To program in Dynamic Brushes, artists create *mappings* from stylus input to geometric and stylistic *brush properties*. Mappings are organized in states and are activated and deactivated through event-driven *state transitions* that can be triggered by stylus input or temporal events. The combination of mappings and transitions determine brush state. The Dynamic Brushes interface is divided between a visual programming environment on a PC where artists write brush programs, and a direct-manipulation drawing system (the execution environment) on an iPad Pro where artists draw with the stylus and see the resultant brush output.

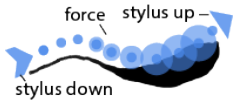
Programming in Dynamic Brushes requires understanding (a) individual mappings, (b) state transitions, and (c) how combinations of brush properties generate visual output on the canvas. Furthermore, because Dynamic Brushes treats stylus data as generic input that can be mapped to any brush property or transition, artists must also understand the decoupling between manual input and brush state. DDB extends Dynamic Brushes with a linked programming and execution environment that allows artists to select elements of their artwork in order to inspect program execution during manual drawing. DDB provides execution information in three categories (input, brush state, and output) and across two modalities: *numerically in relation to code* in the programming interface (Figure 2), and *geometrically in relation to visual artwork* in the drawing canvas (Figure 1).

Numerical Inspection

DDB automatically displays real-time numerical readouts for all program data, enabling artists compare numerical representations of input, state, and output as they work (Figure 2a). As the artist draws with the stylus, the input inspector shows numeric readouts of the stylus's relative and absolute position, force, angle, and event state. If stylus input or other data are present in a brush's mappings, the brush inspector will also update these parameters, including geometric properties (i.e., scale, rotation, and position) and stylistic ones (i.e., stroke weight, hue, saturation, and lightness). The output inspector displays the final position of a brush stroke, which is calculated from applying the geometric brush properties, as well as event state and stylistic properties.

The inspectors are color-coded to the corresponding programming primitives in the Dynamic Brushes language (brush mappings: red, input blocks: blue, output stroke data: purple). Hovering over a brush property in the code will reveal its range of accepted numerical inputs (e.g., [0, 360] for rotation, [0, 1] for hue, $[-\infty, \infty]$ for position). To reduce information overload, we focus only on the active brush instance. If a brush has multiple behaviors, the data of the currently selected behavior will populate the inspectors. Each inspector has tabs to switch between multiple instances of a behavior, such as in the case where one brush spawns multiple children brushes.

Visual Inspection



In addition to the inspection features in the programming environment, DDB displays selected information about the program input, brush state, and output adjacent to the artwork on the drawing interface (Figure 1b). For stylus input, DDB displays current and previous stylus position and force, and stylus up and down events (inline figure). These displays reveal the stylus state regardless of program functionality and in a form that is decoupled from brush output.

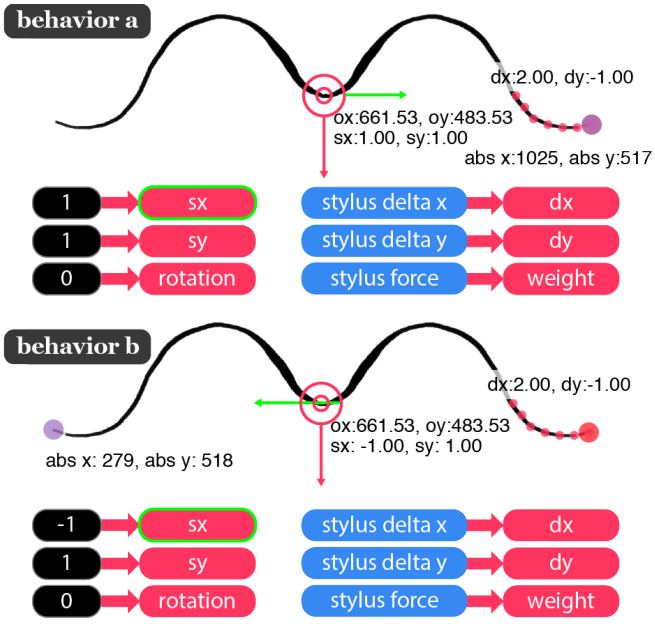
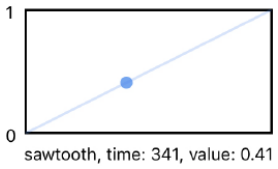
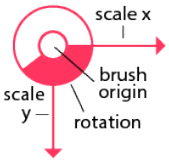


Figure 3. Visual inspectors for brush and output overlaid on artwork generated by two brushes, and each brush's program. Top, brush behavior A follows the user's stylus (right-ending stroke); the red brush position dot and purple output position dot overlap. Bottom, behavior B reflects the user's stylus horizontally (left-ending stroke); the purple output dot is opposite the red brush position dot. The *scale-x* property is highlighted both in the mapping and brush icon. (Note that the input visual inspector is turned off.)

Although DDB centers around stylus input, artists may also use synthetic input streams in the form of *generators* (i.e., random, sine, sawtooth, square, and triangle waves) for greater expressive power to drive brush functionality independent of stylus input. However, unlike stylus input, which has a natural geometric mapping in the canvas, graphic or geometric visualizations of generator data do not correspond to canvas geometry. To avoid chances misleading artists, we visualize a generator's current value as a point in a graph (inline figure) in the corner of the canvas (Figure 1a), which lets artists view changes in the generator value alongside changes in the artwork.



Similar to stylus input, DDB displays brush state through a dynamic icon that shows the brush's geometric properties (inline figure). The brush icon *reifies* the brush state by converting abstract concepts into concrete objects [2]—in this case, the brush origin, position, scale, and rotation. In the Dynamic Brushes programming language, these properties determine a series of transformations that are used in calculating how the output stroke is drawn. The brush icon updates to reflect the numerical values of its properties– the icon is positioned at the current brush origin, its axis lines scale to the *scale-x* or *scale-y* values, and the rotation arc sweeps across the circle as the angle changes from 0 to 360.



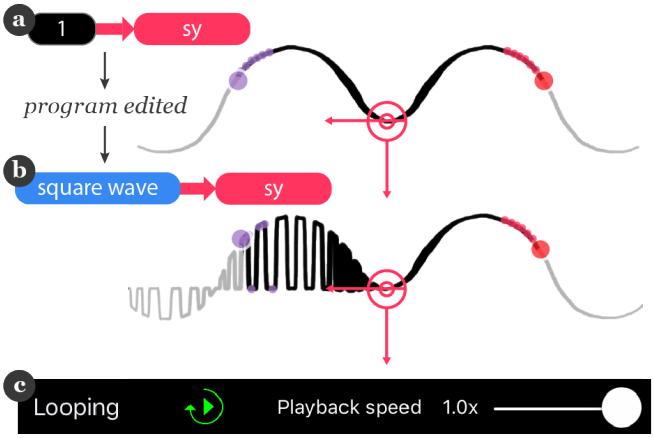


Figure 4. DDB loops an artist's last stroke, updating the artwork as they modify the brush program. (a) Continuing from Figure 3, the system loops through behavior B's input. (b) The artist now maps a *square wave* to *scale y*, which automatically changes the stroke. (Note that the grey lines illustrate future positions and do not appear in the system.) (c) The looping toolbar on the bottom of the drawing interface allows artists to enter and exit looping mode and adjust playback speed.

Trails of points show past brush positions (i.e. before geometric transformation) and computed output positions. Visual inspectors also include labels displaying their numeric values; both visual inspectors and labels may be individually toggled on and off (Figure 1c).

DDB's visual inspection elements are aligned with principles of epistemic action. Because they are lightweight and do not obscure the underlying artwork, the visualizations allow artists to access state information as they draw, rather than pausing and searching through the numerical inspectors for a desired value. Furthermore, the combination of inspection elements for stylus input, brush state, and stroke output enables artists to observe cases where these values converge or diverge. In contrast to other creative coding systems, which display a blank canvas when a program malfunctions, DDB presents visual feedback even if the program does not generate any strokes.

Looping

A challenge all programmers face is the delay between modifying a program and observing the effects of that change [21]. For programs that require extensive manual input, like those in Dynamic Brushes, this is exacerbated by having to make a change on the programming interface with their keyboard and mouse, picking up their stylus, and shifting to drawing on the iPad Pro to observe the change. DDB's *looping* functionality addresses this issue by automatically recording each stylus gesture as the artist draws. At any point, the artist can loop the most recent gesture through the system while making edits to the program (Figure 4). This design ensures that artists' brush programs will still work as intended because each looped segment contains all the events required for a complete state transition through the program. Looping enables artists to adjust their brush program and observe how different mappings and properties result in different outputs, while holding the input constant. Upon each loop completion or after a program edit, DDB erases strokes generated by the

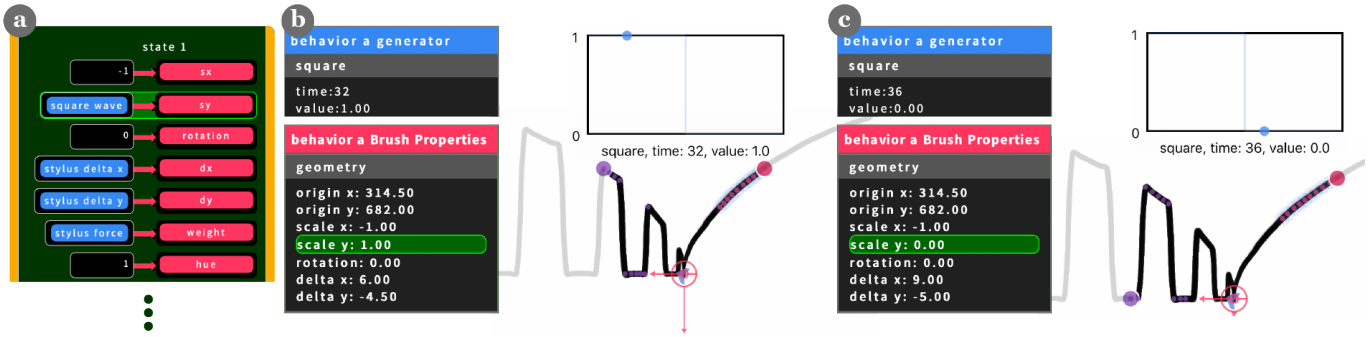


Figure 5. Artists can toggle *step mode* to manually follow their program's execution trace. (a) The brush program highlights the current transition or mapping while stepping. (b) Each step through of a mapping also updates the corresponding property in the numerical inspectors. After stepping through all the mappings, the artwork and visual inspectors update. Here, the square wave's value is 1, which means the output's y position is the same as the brush's y position. (c) After stepping through, the square wave value changes to 0. Now the output's y position is 0, which is at the brush origin.

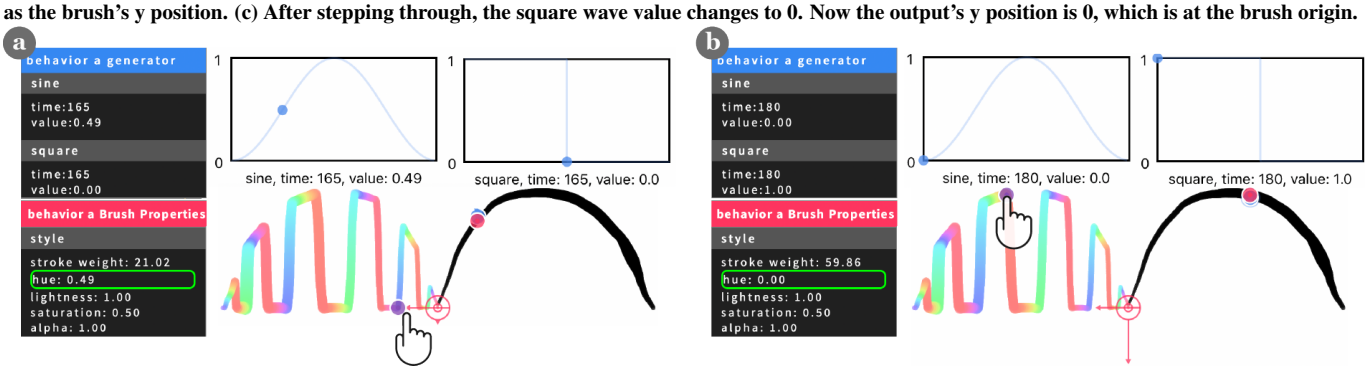


Figure 6. Using DDB's direct-selection feature to understand the numerical values behind colors. (a) An artist uses their finger to touch a point on the stroke that is blue, which jumps the program state to the point where the stroke was drawn and refreshes the numerical and visual inspectors. The hue value for blue is 0.49. (b) The artist now touches a red point, which again updates the program state, revealing a hue value of 0 for red.

previous loop and restarts the loop. We designed looping to enable artists to focus their attention on exploring and modifying their programs by quickly observing how their drawings changed, rather than laboriously setting recording start and end points or repeatedly generating manual input.

Stepping

In addition to observing changes in visual output in response to program edits, artists also often need to understand the effects of a series of program operations. Similar to controlled debugging practices in software development, artists require fine-grained control of program execution to debug complex drawing behaviors. To support this, DDB provides stepping functionality. When step mode is activated, the system will re-initialize all active brush behaviors and queue up the last stylus stroke as input. On a key press, the system will advance through each stage of the program, including initialization, event evaluation, state transition, and mapping evaluation. In the programming environment, DDB highlights the current transition, state, or mapping as it is executed (Figure 5). On the canvas, drawings and visual inspectors incrementally update on each step. Stepping is more complex to use than looping, but enables a more granular form of program inspection. Artists can use stepping to track how a brush property changes in response to an input, or to understand how the series of geometric brush transformations affect output location.

Linked Direct Selection

Artists may also want to examine program state at specific points within a drawing as they work. DDB supports this through the ability to directly select points along the drawing to inspect program execution. When artists touch visual inspection elements, they highlight the corresponding row for the property in the numerical inspectors and in the code (and vice-versa), which makes it easier to identify and focus on a specific parameter during debugging. When they touch any point in their artwork, DDB will jump the program state back to the point in its execution that corresponds to drawing that stroke and accordingly update the numerical and visual inspectors (Figure 6). We chose selection by touch to be in line with interactions artists were familiar with in manual and direct manipulation tools. By seeing the numerical values and code that made specific parts of their artwork, artists can retrieve and inspect "happy accidents" encountered while exploring, helping them transition back to authorship to recreate their desired (though sometimes unintentionally discovered) outputs.

Implementation

We developed DDB as an extension to the Dynamic Brushes programming language because it enabled us to explore ways of supporting inspection and execution control for programming environments that act on manual drawing input. We added functionality that automatically records stylus events, position, and force inputs as artists draw, and segments this data into individual stylus gestures. At a regular interval, as the system receives input, it stores the corresponding state

for all brushes and output in relation to the input. When artists engage in real-time drawing, or loop through previously recorded input, the system returns the current time for the input (either in real time or recorded) which is used to access the corresponding brush state and output data. When artists select an arbitrary point on the artwork to inspect, the system determines the time at which the selected stroke segment was drawn and uses that time to access the corresponding brush state and input.

EXPLORING AND MODIFYING PROGRAMS WITH DDB

Collectively, DDB's feature set enables artists to inspect program input, state, and output, and control program execution while engaged in both drawing and programming. To concretely demonstrate how DDB's features support visual and interactive inspection, epistemic action, and rapid transition from authoring to exploring, we present the workflow of prototypal visual artist at work with DDB.

Jo is an experienced digital painter who has signed up for a Dynamic Brushes workshop. After an initial tutorial on the language, Jo opens an example program, a mirror brush, which reflects what she draws with the stylus across the y axis, and is also pen pressure sensitive (inline figure). To figure out what makes the brush pressure sensitive, Jo clicks on *stroke weight* in the brush inspector, which highlights in the code (Figure 2b) the mapping between *stylus force* and *stroke weight*.

Jo now wants to figure out what makes the mirror brush "mirror" her stylus input. She notices the mirror brush is composed of two different brush behaviors, each of which shows different data in the geometric inspector. Looking at the reified brush icon for each behavior, she sees the x-axes are pointing in opposite directions (Figure 3). She touches the x-axis on the brush icon for behavior B, which highlights it neon green and correspondingly highlights the *scale x* property in the numerical inspector, which reads -1. The visual difference between the two icons reveals the reason for the mirroring effect—the two behaviors are identical, except behavior B flips the x-axis.

Now that Jo understands how the brush works, she wants to make modifications. She presses a button on the drawing interface to turn on *looping* (Figure 4c, and 1d in context). Jo uses this feature to quickly explore new ideas for mappings on the programming interface, without needing to pick up her stylus and redraw her strokes after every change (Figure 4a). To explore, she drags in different generators to different brush properties. Jo stops looping when she drags in a *square wave generator* to the *scale-y* property because this produces an interesting result (Figure 4b).

To better understand how the square wave affects her drawing, Jo toggles on *step through* mode in the programming interface (Figure 2c). As she steps through her code, Jo pays attention to how the output changes alongside the square-wave visualization. She notices that when the square wave is 1, the output follows her stylus input, but when its value is 0, the brush icon's y axis is substantially shortened and the output jumps back to the location of the brush origin (Figure 5). She reasons

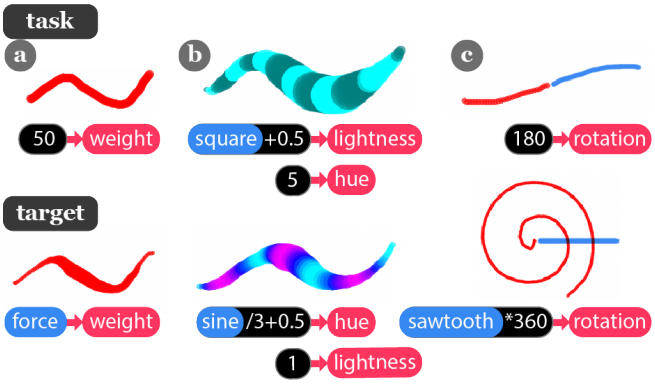
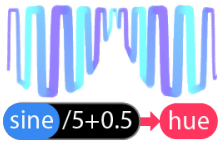


Figure 7. The three study tasks participants completed. (a) Modifying a constant weight to be mapped to stylus force. (b) Modifying a brush whose visual characteristics depended on the lightness property to depend on the hue property. (c) Modifying a brush with a constant rotation to create a spiral. (Note the blue line shows the stylus input path.)

that the *square wave* → *scale y* mapping causes her output to draw at the *stylus y* position when the square wave's value is 1, and then the *brush origin* position when its value is 0.



Jo wants to also experiment with the stylistic properties of her brush—specifically, she would like an automatic color change as she draws. To first understand the numerical range of input, she hovers over the *hue* property in her program, which displays a tool tip from 0 to 1. She drags a sawtooth wave into the hue, but looking at the output, doesn't like the abrupt jump in color. By looking at the sawtooth wave visualization, she notices it linearly increases from 0 to 1 before suddenly jumping back to 0, so she switches the mapping to a sine wave, which has a more gradual transition back down. She drags in a sine wave to the hue, which draws a rainbow stroke. Jo decides she wants to restrict the color to cooler hues. She uses the *linked direct selection* feature and touches a blue part of the stroke, jumping her program state (Figure 6a). Jo notices that the hue value here is 0.49. She then touches a red part of the stroke and observes that the hue value at this point is 0.0 (Figure 6b). Jo returns to looping mode, and starts scaling the sine wave by dividing it by a constant until she gets a range she likes, and then adds 0.5 to it to bring the hue range to the blues (inline figure). She can construct the correct mathematical expressions to apply to the sine wave, having seen the target values through linked direct selection.

EVALUATION

In evaluating DDB, we sought to understand how visual artists relied on and transitioned between the inspection features on the programming and drawing interfaces when learning the Dynamic Brushes programming language. We conducted an observational study where we incrementally introduced DDB's support features while participants made a series of modifications to existing brushes to achieve demonstrated target brushes. We also observed artists as they made open-ended variations to a brush program of their choice. We primarily focused on collecting qualitative data including how participants approached making variations, what inspection features they

used or not, and how and when they transitioned between the programming and drawing environments.

Procedure

We recruited 5 professional visual artists (2 male, 3 female) with little to no prior programming experience through a university fine arts department. We introduced participants to DDB's support features and the Dynamic Brushes programming language by guiding them through the process of creating a sample brush. We then gave participants three structured tasks (Figure 7) where they had to modify an example brush, and observed how they relied or didn't rely on the features we introduced. Finally, participants had 15 minutes of open exploration to make any desired modifications to an example brush of their choice. Each study session lasted approximately two hours. We chose the three tasks to probe participants' understandings of property mappings and the difference between live input and constant numeric values (Figure 7a), and how abstract input data affects stylistic (Figure 7b) and geometric (Figure 7c) properties. We constrained tasks to editing data-bindings for brush properties (rather than creating states and transitions) to focus on participants' abilities to modify a brush's properties from input data.

DDB's support features are designed to work in tandem with different creative scenarios, such as initial learning, focused problem solving, and open exploration. We chose a qualitative methodology so we could observe how participants used the support features across these different scenarios, over longer periods of time, and in a holistic fashion that aligns with aspects of real-world use.

Results

Participants relied heavily on features that matched to their existing workflows—they frequently used looping and focused on the visual inspection elements. They relied on the numerical inspection in rare cases when solving specific technical challenges, or not at all. We detail the results for each feature below, presenting results in the form of representative quotes and anecdotes due to the qualitative nature of our evaluation.

Visual Inspection: All participants referenced the visual inspection elements shown alongside their drawings when thinking aloud during the study. For example, P2 said the visualizations made him *think spatially rather than mathematically, so it was about the locations and less about numbers and multiplications.* Three participants highlighted the rotation inspection indicator as particularly helpful in their reasoning about brush state. P3 pointed out it helped her understand the scale of her input data, looking to see *if the rotation arc was full* so she could *make big enough changes—when it wasn't full, that's when I realized I needed to multiply [the sawtooth wave in Task C].*

Four participants said the synchronization between the generator visualizations and the brush position indicator was helpful. Successfully completing Task C, P3 said, *I knew the sawtooth wave one was right since it only went up in one direction as I drew, just like how the spiral goes in one direction, while the other generators went both up and down.* However, in some cases, the generator waveform visualizations lead participants to make incorrect assumptions since they expected the waveforms to geometrically align with the types of geometric transformations they had to create. P2, who used a sine wave for Task C, said, *I was caught up in the idea that the spiral needed to rotate. The straight line of the sawtooth wave didn't match with the spiral's curved shape in my head.*

Four of the participants also used the numerical labels that accompanied the drawing visualizations to make decisions when modifying their programs. P4 said, *I preferred the labels to laser-focusing on the tiny little boxes [in the programming interface].* However, P5—a traditional painter—toggled off the on-canvas visualizations, stating, *When I see these numbers, I'm reminded of the programmatic aspects of my drawing and not the artistic aspects of it, so I feel like my attention is being pulled away.*

Numerical Inspection: Participants almost universally avoided looking at the real-time numerical print outs in the programming interface. When asked, they described being overwhelmed and intimidated by the amount of numerical information and its refresh rate. Looking at the numerical inspectors required a context-shift—P4, who looked to understand why her hue value wasn't changing, said, *This is helpful, but painful. I love it, but you have to focus, which takes you out of the drawing zone.*

Looping: Most participants used looping to understand the range of a mapping—such as changing the hue value from 0 (red) to 0.5 (blue) to 1 (back to red, since hue is a color wheel) and observing the changes while the system redrew their input. They also used looping to quickly experiment with mathematical modifiers. However, P5 avoided using looping, saying he *valued the kinesthetic experience* of manual drawing, and that looping his inputs made him fixate on *past mistakes.*

Stepping: Stepping through the program execution trace was never used outside of instruction. When asked, participants said they forgot about the feature, or would use it in detailed scenarios but found it at odds with the rapid nature of manual drawing. P1 said, *I didn't think stepping was super useful for me, but my personality is fast-paced and quick—I gravitated towards looping, which let me see my mistakes faster.*

Linked Direct Selection and Highlighted Visualizations: We observed one participant (P2) touching a brush which mapped hue to a sine wave during Task B. He wanted to understand which values of the sine wave corresponded to which colors, and looked at how the generator visualization changed values as he touched different colors on the stroke. Although participants found the highlights across the programming and drawing interfaces useful in guiding their attention to their actively edited parameters when they were triggered during instruction, we did not observe them independently selecting visualization elements to highlight them in the numerical inspector. Overall, participants gravitated towards watching the visualization elements in drawing interface update as the drawing looped or as they manually drew.

Integrated use of DDB: As the study went on and participants became more familiar with the inspection features, they used them more. P2 said, *At first I somewhat ignored the fea-*

tures to feel more flexible while drawing. But as we created the brushes over time, I needed to know exactly what the numbers were. An hour into the session I felt like I was translating between the world of numbers and artwork, and I could see how the correspondence was happening. This was also reflected in the open ended exploration, where participants started with a desire to play, focusing primarily on the drawing instead of the inspection features. As they experimented and generated ideas, they began trying to understand data relationships and examined the visual inspection elements. Many participants wanted to understand how to manipulate data into the minimum/maximum ranges to get a "dramatic effect."

During the constrained tasks, participants often made incorrect choices about the kinds of input data to use if their edit didn't result in any major *visual changes* in the drawing, even if the data numerically changed. Furthermore, although they understood the terminology for brush properties, participants had difficulty isolating specific properties in their visual effects. For example, many participants expected that lightness (as opposed to hue) would change a brush from blue to purple, or that scaling would impact a brush's rotation.

LIMITATIONS

Our preliminary evaluation structure was limited by the small number of participants and the lack of an explicit comparative baseline (e.g., the original Dynamic Brushes system without DDB's new features). Furthermore, evaluations with additional artists over a period longer than two hours would likely reveal additional insights—particularly regarding the use of stepping and numerical inspection as artists warm up to thinking with numerical data. Inspection features impacted system performance when processing complex input data or large programs which participants sometimes found limiting. These issues are common in systems building research with prototype software, and could be mitigated by optimizing our data storage and retrieval.

DISCUSSION

We designed DDB to explore our hypothesis that programming environments that provide explicit links between *manually manipulating visual artwork* and *observing program functionality and state* could aid visual artists in understanding creative coding languages. Here we examine that hypothesis and discuss the outcomes of our approach with respect to our original design goals. Based on our analysis, we present five design principles for programming support tools for artists below.

Display Numeric Information in Relation to Artwork

All participants were reluctant to use the numeric programming inspectors throughout the study, stating that they were overwhelmed by the volume, rate of change, and numeric quality of the information. Yet all but one participant repeatedly referenced numerical labels adjacent to the visual inspectors in the drawing environment. Despite replicating the numeric data from the programming inspectors, participants did not describe being overwhelmed by the labels—rather, they used them to inform decisions when editing their code. This suggests that visual artists are more likely to engage with numeric program data when it is displayed the context of the artwork and contextualized with graphic signifiers. Overall, in comparison to traditional software debugging tools which provide numeric inspection of program state relative to the code, programming environments for visual artists should display relevant numeric input, state, and output data *in the execution environment in relation to the geometry of the artists' in progress artwork*. The system should also let artists toggle between showing or hiding this information.

Support Epistemic Action on Input and Output

We designed DDB's visual inspectors so artists could examine program state while drawing. The fact that four of the five participants used the visual inspectors while drawing with the stylus indicates that they found the elements to be compatible in their drawing workflows. Participants frequently commented on changes in the visual inspectors while they drew, indicating they could also focus their attention on the program state information conveyed.

Participants largely avoided DDB features that did not create perceivable changes in the artwork. This was particularly apparent for the visual inspector linked highlighting feature. Although highlighting showed relevant code and numeric output in the programming environment, unlike manual drawing or looping, interacting with the visual inspector elements did not update the artwork.

The importance artists placed on combining manual manipulation with *perceivable changes* in output suggests that future programming environments for visual art should enable epistemic action by allowing artists to *manually manipulate artwork, program state, and input data to understand code*. Along these lines, future work could allow artists to manipulate output by adjusting the linked highlighting features in DDB. By manually moving, scaling, and rotating the stylus and generator inspectors, artists could adjust program input and immediately observe the results in the drawing environment. For example, artists could select and position dots along the stylus inspector path to alter the stylus input data (akin to editing control points on a bezier curve). Similarly, by rotating, scaling and moving the brush inspection icon, artists could alter the input to the brush rotation, scaling and origin mappings. Updating brush mappings that manipulate external data with mathematical expressions would pose some implementation challenges, since artists would need to be able to specify the scope of their direct manipulation actions with respect to specific inputs and constants within the mapping expression; however, prior work in bi-directional editing demonstrates ways to approach this in editing vector graphics [12].

Inspect Geometric Data on the Drawing Canvas

DDB's visual inspectors were based on the idea that graphically displaying program input and state in the drawing environment would provide an approachable way for artists to build an understanding of program functionality. Our study observations support this premise—in particular, the combination of the generator and visual brush inspectors aided participants in understanding mappings between abstract input

and brush geometry. Our evaluation also highlighted the challenges of showing visualizations of abstract data alongside geometric artwork. Participants frequently assumed that mappings between generators and brush geometry would translate to effects in the artwork that matched the geometry of the generator waveform. This misconception reveals the broader challenge of aiding artists in understanding applications of *non-geometric* input data, something that is absent from physical and direct-manipulation drawing tools.

Overall, participants' use of the visual inspection features suggests that programming environments for visual artists should enable artists to inspect *geometric* data through dynamic visual representations of geometric input, state, and output on the drawing canvas. These elements should be displayed relative to the corresponding artwork geometry, and update as the artist manually draws with the system. Non-geometric input should be visualized within the drawing environment but *outside the drawing canvas* to distinguish it from the geometric data.

Rapid and Arbitrary Execution Control

All participants in our study explored code variations during both structured and open-ended phases. In the process all but one used looping to observe the effects of code changes in drawing environment but all participants avoided using stepping. There are several potential conclusions to draw from this result. First, participants' reliance on looping indicates that looping speed (loops execute at the speed of the original drawing) and ease (input data is recorded and played back automatically) were well aligned with the speed and playfulness of exploratory manual drawing. Similarly, participants' avoidance of stepping suggests the need to manually activate stepping and the slow display of visual output in the step-through process conflicted with speed and flow exploratory drawing. While it is possible that artists who are more experienced in using Dynamic Brushes would rely more on stepping to debug complex programs than the participants did in our study, executing code line-by-line will always be at odds with rapid free-form drawing. The importance of speed in exploratory visual art creation indicates that rather than enable line-by-line execution traces, *programming environments for visual artists should allow artists to trace execution at arbitrary points in the drawing process*. Future work in DDB might examine ways to combine looping with targeted inspection during program execution. One possibility would be to adapt the linked direct selection functionality to enable artists to specify breakpoints in their code by selecting discrete points on the artwork.

Inspect Across Different Manual Inputs

Artists' reactions to the looping functionality also reveal insights about handling input recording and playback. In traditional debugging tools, programmers need to test interactive systems on consistent input behavior to achieve predictable results. However, in programs that accept manual input, the opposite is true. Manual drawing is expressive because it reflects variations across different drawing gestures—therefore, when combining programming and manual drawing, artists need to understand how programs will function across these different variations. This was evident in the study when artists opted to test changes in the code not by looping but by redrawing strokes with different gestures and speeds, and when one participant described not wanting to use looping because it made him focus on the mistakes of a single gesture. These observations suggest that, unlike traditional debugging tools that support recording and playback of a single interaction input, *programming tools for artists should support examining program performance across a variety of manual inputs.*

CONCLUSION

Motivated by the opportunities of programming for visual art and the challenges programming presents for visual artists, we created DDB, a tool that bidirectionally links code, numerical data, and artwork across the programming interface and the artist's in-progress artwork. DDB provides artists with visual and numerical inspection, looping and step-through of manual input, and linked selection between artwork and program state. Our evaluation showed that artists benefit from program inspection that is compatible with manual drawing and displays state in the context of the artwork. We see future opportunities in developing visual inspection elements that enable artists to manipulate input and code. Overall, this work expands how programming environments can support different kinds of programmers in understanding and authoring code.

ACKNOWLEDGEMENTS

Many thanks to the artists and computer scientists who participated in our studies. Special thanks to Evan Strasnick and Will Crichton for detailed feedback on our programming model and system functionality.

REFERENCES

[1] Cycling '74. 2016. Max. (2016). http://cycling74.com/products/max.

[2] Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '00)*. ACM, New York, NY, USA, 102–109. DOI: http://dx.doi.org/10.1145/345513.345267

[3] John Berger. 2008. Drawing. In *Selected Essays of John Berger*, Geoff Dyer (Ed.). Knopf Doubleday Publishing Group.

[4] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 473–484. DOI: http://dx.doi.org/10.1145/2501988.2502050

[5] Ellen Do and Mark D. Gross. 2007. Environments for Creativity: A Lab for Making Things. In *Proceedings of the 6th ACM SIGCHI Conference on Creativity & Cognition (C&C '07)*. ACM.

[6] Max Documentation. 2019. Max Signal Probing. (2019). https://docs.cycling74.com/max7/vignettes/signal_probe.

[7] Benjamin Jotham Fry. 2004. *Computational information design*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[8] David J. Gilmore. 1991. Models of debugging. *Acta Psychologica* 78, 1 (1991), 151–172. DOI: http://dx.doi.org/https://doi.org/10.1016/0001-6918(91)90009-O

[9] M. D. Gross. 2009. Visual Languages and Visual Thinking: Sketch Based Interaction and Modeling. In *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling (SBIM '09)*. ACM, New York, NY, USA, 7–11. DOI: http://dx.doi.org/10.1145/1572741.1572743

[10] 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

[11] B. Harvey. 1991. Symbolic Programming vs. the A.P. Curriculum. *The Computing Teacher* 56 (February 1991), 27–29.

[12] 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

[13] Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 233–245. DOI: http://dx.doi.org/10.1145/2984511.2984570

[14] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 532, 12 pages. DOI: http://dx.doi.org/10.1145/3173574.3174106

[15] Tim Ingold. 2010. The textility of making. *Cambridge Journal of Economics* 34 (01 2010), 91–102. DOI: http://dx.doi.org/10.1093/cje/bep042

[16] Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. 2018. Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 590, 13 pages. DOI: http://dx.doi.org/10.1145/3173574.3174164

[17] 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

[18] 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, 140–149. DOI: http://dx.doi.org/10.1145/1142405.1142429

[19] Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 301–310. DOI: http://dx.doi.org/10.1145/1368088.1368130

[20] Mode Lab. 2015. What is a Data Tree? In *The Grasshopper Primer* (3rd ed.). https://modelab.gitbooks.io/grasshopper-primer/1-foundations/1-5/2_what-is-a-data-tree.html.

[21] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490. DOI: http://dx.doi.org/10.1145/2556288.2557409

[22] Henry Lieberman. 1984. Steps Toward Better Debugging Tools for LISP. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 247–255. DOI: http://dx.doi.org/10.1145/800055.802041

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

[24] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernand Kerr, and John Stasko. 2018. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *To Appear. Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/3173574.3173797

[25] Rune Madsen. 2019. Introduction. In *Programming Design Systems*. https://programmingdesignsystems.com/introduction.

[26] Michael Mateas. 2005. Procedural literacy: Educating the new media practitioner. *On the Horizon* 13 (06 2005), 101–111. DOI: http://dx.doi.org/10.1108/10748120510608133

[27] Lauren McCarthy. 2016. P5.js. (2016). http://p5js.org/.

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

[29] Matt Pearson. 2011. *Generative Art*. Manning Publications Co., Greenwich, CT, USA.

[30] Casey Reas. 2004. The Language of Computers. In *Creative Code*, John Maeda and Red Burns (Eds.). Thames & Hudson, London, United Kingdom, 44.

[31] C. Reas and B. Fry. 2004. Processing. (2004). http://processing.org.

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

[33] C. Reas and B. Fry. 2019. Processing Environment. (2019). https://processing.org/reference/environment/.

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

[35] 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.

[36] David Rutten. 2007. Grasshopper. http://www.grasshopper3d.com. (2007).

[37] Toby Schachman. 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)*. ACM, New York, NY, USA, 1–10. DOI:http://dx.doi.org/10.1145/2384592.2384594

[38] Bret Victor. 2011. Dynamic Pictures. (2011). http://worrydream.com/DynamicPicturesMotivation.

[39] B. Victor. 2012. Learnable Programing: Designing a programming system for understanding programs. (2012). http://worrydream.com/LearnableProgramming/

[40] vvvv group. 2017. vvvv. (2017). https://vvvv.org/.

[41] Marius Watz. 2012. The Algorithm Thought Police. http://mariuswatz.com/mwatztumblrcom/the-algorithm-thought-police.html/. (2012).

[42] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, De Araujo, and Daniel Widgor. 2018. DataInk: Direct and Creative Data-Oriented Drawing. In *To Appear. Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/3173574.3173797

[43] Loutfouz Zaman, Wolfgang Stuerzlinger, Christian Neugebauer, Rob Woodbury, Maher Elkhaldi, Naghmi Shireen, and Michael Terry. 2015. GEM-NI: A System for Creating and Managing Alternatives In Generative Design. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 1201–1210. DOI:http://dx.doi.org/10.1145/2702123.2702398