Programming By Scaffolded Demonstration with Perpend

Angela Bi*
University of California, Berkeley
Berkeley, California, USA
angelabi2003@berkeley.edu

Eric Rawn*
Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California, USA
erawn@berkeley.edu

Justin Lubin
University of California, Berkeley
Berkeley, California, USA
justinlubin@berkeley.edu

Sarah E. Chasins
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
schasins@cs.berkeley.edu

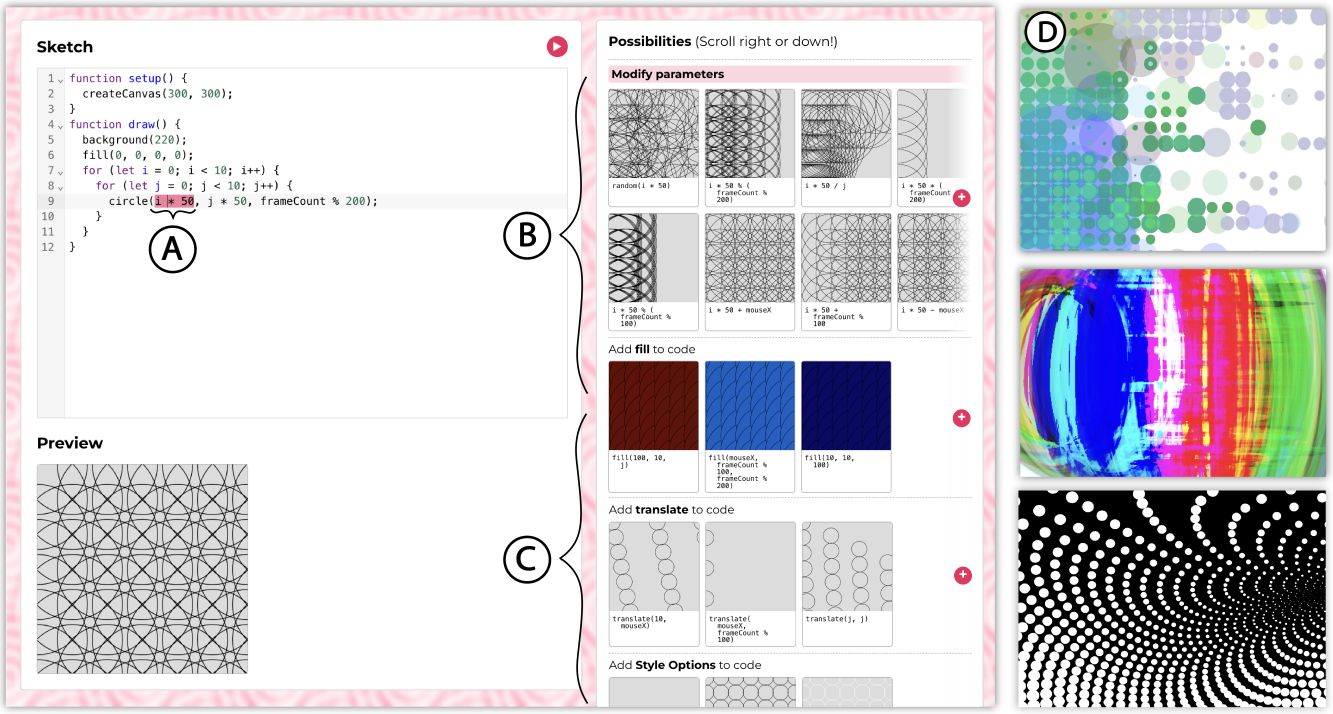


Figure 1: *Perpend* is a `p5.js` editor to support users in *understanding the material possibilities of their program.* Perpend displays the visual output of many possible modifications to the user's program at the cursor location. We call this approach ***Programming By Scaffolded Demonstration*** because it scaffolds a user in navigating their program's space of possibility by displaying a range of valid expression modifications and their outputs. Selecting a pane in the gallery makes the corresponding modification to the user's program. **A.** Perpend highlights the part of the program for which modifications are currently displayed. This is selected using the cursor. **B.** A gallery displays a range of parameter modifications to the program AST at the selected region, displaying the visual results above the textual change. **C.** Perpend also displays the results of new statements appended after the currently selected line in the user's program. **D.** Artwork created with *Perpend*.

*Both authors contributed equally to this research.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

CHI '26, Barcelona, Spain

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2278-3/26/04
https://doi.org/10.1145/3772318.3791327

Abstract

Output-centric programming paradigms such as Direct Manipulation Programming, Programming By Demonstration, and Programming By Example enable users to author programs by constructing an intended output. However, sometimes the purpose of a programming interaction is to *discover* an "intended output" in the first place (e.g., exploratory data analysis, improvisational creative coding, early-stage prototyping). We argue that one role for output-centric programming here is scaffolding the user in demonstrating their next program editing step by selecting among possible modifications to their current program. We call this *Programming By Scaffolded Demonstration* (PBSD). To explore PBSD, we built Perpend, a programming environment for `p5.js`. In a user study with nine artists, we juxtapose Perpend with an existing Direct Manipulation editor, exploring how participants used Perpend to situate themselves within a space of possible programs, shift focus between program text and visual output, and shape their exploration by modifying their program structure.

CCS Concepts

• **Human-centered computing** → *Interactive systems and tools*; *Human computer interaction (HCI)*; • **Software and its engineering** → *Development frameworks and environments*; • **Applied computing** → **Media arts**.

Keywords

End-User Programming, Creative Coding, Output-Centric Programming, Direct Manipulation Programming, Processing, P5.js

ACM Reference Format:

Angela Bi, Eric Rawn, Justin Lubin, and Sarah E. Chasins. 2026. Programming By Scaffolded Demonstration with Perpend. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26), April 13–17, 2026, Barcelona, Spain.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3772318.3791327

1 Introduction

Output-centric programming paradigms such as Direct Manipulation Programming [14, 30, 59], Programming By Demonstration [1, 27, 33], and Programming By Example [35, 36, 97] create programs that are consistent with a user-provided target output. These outputs take the form of, respectively, direct manipulation edits to an artifact (often a graphical image), demonstrations of a procedure, or a set of input-output examples. However, in some situations (especially common in domains like data science [43, 87] and creative practice [66, 93]), users do not always have a target output in mind. In these cases, the purpose of the programming work is to determine such an output. This style of programming has been discussed in a variety of ways: Kery et al. describe it as "exploratory" programming, where "the practice of designing the goal at the same time as experimenting in code is a defining feature" [8]; Brandt et al. discuss the closely related concept of "opportunistic programming", or programming to "prototype, ideate, and discover" [9]; Shaw phrases the idea as: "Much software is developed to discover what it should do, not to satisfy a prior specification" [82]. In each case, the work of programming is coextensive with the work of finding out what the programmer intends the final outcome to be. This raises a central question we address in this paper: *What might be the role of output-centric programming when users do not have a target output in mind?*

We rely on the ideas of anthropologist Tim Ingold [37, 38] to propose an answer: *helping the user understand the space of nearby possible programs given their programming language.* As we discuss in Section 2, our key insight is to view programming languages as *materials* as understood by Ingold: something to be followed and responded to rather than something to force into a predefined form. To explore this idea, we introduce a programming interaction, *Programming By Scaffolded Demonstration* (PBSD), so called because it scaffolds a user in understanding the possibilities of their programming language *as a material* by displaying a range of valid programs reachable from small textual edits and their outputs. If, following Ingold, the role of a user's intentions is not to serve as a specification on the final output (but, rather, as an evolving sense of direction), then the system's aim should not be to align with a hypothesized target output that may or may not exist in the user's mind. Neither, on this picture, should a PBSD system impose metrics or dimensions on which to evaluate generated outputs (e.g., diversity, complexity, interest etc.). Rather, the aim of a PBSD system would be to help a user in exploring their program as a material: the space of programs to which the user can possibly go next with small textual edits, as determined by the design of their programming language.

To explore PBSD in practice, we developed Perpend, an editor for the creative coding library `p5.js` (Processing) [77]. Perpend displays (i) a range of small program modifications and (ii) the visual outputs generated by applying those modifications to the user's program, then running the modified programs. We juxtapose Perpend with an existing Direct Manipulation editor for `p5.js` to understand their different and, as we found, complementary roles. In our user study, we asked nine expert `p5.js` programmers to use each interface for ∼45 minutes on work of their choosing, with a ∼30 minute interview about their experiences. We found that Perpend helped participants expand their sense of possibility across different moments of creative "momentum," work across their textual program and visual output, and expand their own Processing vocabularies. In summary, we present the following contributions:

(1) **Programming by Scaffolded Demonstration**—a new output-centric program construction approach that does not require users to provide desired outputs.
(2) Perpend, a programming environment that instantiates Programming by Scaffolded Demonstration for `p5.js`, which presents galleries of paired code edits and output previews.
(3) A qualitative analysis that puts Perpend in conversation with an existing Direct Manipulation creative coding interface [64], with findings discussing how programming systems can better accommodate a programming as a *manual* practice and programming languages as *materials*.

2 Background and Key Idea: Programming Languages as Materials

This work explores how programming interfaces might support users in what Tim Ingold terms "following [their] materials." In this

section, we describe how we used Ingold's ideas to propose one way a programming language might be thought of as a "material" in this sense: as the space of programs reachable from individual textual modifications given the structure of the programming language. This perspective will be the foundation for defining *Programming by Scaffolded Demonstration*.

Materials. Ingold argues that we should think of a practitioner's intentions and their engagement with the world as interdependent and mutually-informing [37, 38]. In other words, when the purpose of practitioners' work is to discover an intended outcome through the work itself, their intentions may not function as specifications on the final result. Rather, practitioners' intentions motivate engaging with parts of their world *as a material*; that is, as something to be followed and responded to rather than something to force into a predefined form. ("Material," for Ingold, does not indicate a class of things, but names the parts of the world with which a practitioner has this relationship of following and responding.) Practitioners' engagement with their *materials*, in turn, shapes their intentions over the course of the work. The role of a practitioner's intentions, on this view, would then be as a sense of *direction*—continually evolving through the act of navigation—rather than a specification of a final destination. Ingold provides a helpful example: a woodcutter may intend to cut timber with an axe, but the task of splitting the timber is not one of imposing an abstract imagined shape onto passive wood. Rather, the woodcutter "follows the grain" of the wood, adjusting and developing their intentions with each swing in response to the variations of the wood as a *material*.

Itineration. In Ingold's words, following a material "is a matter not of iteration but of itineration" [37]. "Itineration" is the essential *improvisation* work required to respond to the world *as a material*—we develop (and re-develop) our intentions for the work in order to continually adjust to the contours of our materials. From an observer's perspective, the carpenter might appear to be repeating the same motion in splitting the timber (i.e., iteration). But for the carpenter themselves, "who is obliged to follow the material and respond to its singularities" [37], splitting the wood is *itineration*: they continually improvise the next step in response to the last and "as a preparation for the one following" in response to the variations and idiosyncrasies of the material at hand.

Key Idea. But do practitioners engage programming languages *as materials* like they do wood or clay? Do programming languages have a "grain" to be "followed"? In this paper we argue *yes*, and that one way to think about the material possibilities of a program is *what the programming language makes possible*. The space of possibilities for a given program is the set of modifications that are valid given the design of the programming language. The role of an output-directed programming system in "following the grain" of a program would then be to help the user answer the question: "Given my programming language, where can I go next with my program?" A system could do this by showing the user a range of possible small modifications to their current program and displaying the resulting outputs. If our interpretation of Ingold is right, we might expect that helping a user understand the space of possibilities around their program (understanding their programming language as a material) could change how they discover what they intend (how they *itinerate*).

We introduce *Programming by Scaffolded Demonstration* as an approach to output-centric programming systems design which supports a user in understanding their programming language as a material. It does so through generating and displaying a range of small textual edits *given what the programming language makes possible*, rather than mutating a program in order to search for a *target output* of some kind. The motivating question for PBSD that we address in this paper is then: How might we use output-centric programming and program generation techniques to bring users *closer* to their programming language *as a material*, rather than abstract away from it by directing their focus to only inputs and outputs?

3 Related Work

3.1 Working with Materials

In HCI literature, Benabdallah and Peek [7] discuss Simondon's notion of "technical materiality," urging that "systems research should support … the *conditions* of production and not just the realization of computational designs." Due to Ingold's significant inheritance of Simondon's arguments, we see our work exploring precisely this suggestion in the case of programming languages and tools. Other engagements with Ingold in HCI have investigated fabrication among maker communities [94] and explored fabrication systems in which "the maker grows with a set of materials and practices, becoming attuned to their properties or forces and generating knowledge that carries through their practice" [15]. We continue this important line of thought by considering the materiality of programming languages themselves.

Although we work with Ingold in this paper, Schön [80] and Suchman [88] present ideas which have much in common, but with markedly different emphases. Schön's "reflective conversation with materials" [80] is described in terms of an "experiment" through which a practitioner develops a better theory of the design situation: the reflective practitioner "must learn by reflection on the situation's resistance that his hypothesis is inadequate, and in what way, or that his framing of the problem is inadequate, and in what way" [81, 153]. Ingold might then categorize Schön's "conversation with materials" as a picture of "iteration," in contrast to his view of "itineration," the latter of which emphasizes the essential improvisation in responding to the unique features of the work rather than the gradual refinement of a specification through repeated experimentation. Suchman, by contrast, does seem to argue for an *itinerant* position, but just has little to say about materials themselves—her focus is on understanding *plans* as "resources" for attending to materials [88]. She includes an illustrative example of navigating rapids in a canoe: "The purpose of the plan in this case is not to get your canoe through the rapids, but rather to orient you in such a way that you can obtain the best possible position from which to use those embodied skills on which, in the final analysis, your success depends" [88].

Ingold's conception of materiality is offered as an alternative to other discussions which bring attention "not to the vitality of materials but to the agency of objects" [37]—namely Latour's Actor-Network Theory [46], but might also include more recent examples

in HCI [23, 92] and Cognitive Science [57]. While we personally found Ingold's interest in skilled practices on *making* with materials to be the most illuminating for our work here, other perspectives on materiality have certainly been found useful elsewhere in HCI.

In domain-focused studies, HCI researchers have also studied the relationship practitioners have with their materials: ceramic [67], clay [15], wood [16], digital drawing [40], and creative software [93]. We hope to add to this conversation, studying how Processing artists use their programming language as a material in creative practice.

3.2 Output-Centric Programming

A variety of programming systems—such as Direct Manipulation Programming, Programming By Example, and Programming By Demonstration—center program outputs in the program construction process. We call such systems *output-centric programming systems*. To date, output-centric programming systems enable users to construct programs by having the user provide intermediate, partial, or full target outputs.

Direct Manipulation Programming [14] augments traditional programming languages with the ability to *directly manipulate* [83] outputs. Typically, Direct Manipulation Programming systems aim to automatically alter the user's program in response to users' direct manipulation actions using techniques like program synthesis [14, 30, 31], a mapping between GUI actions and specific program transformations they should trigger [25, 31, 32, 44, 54, 64, 99, 108], bidirectional evaluation [59, 106], and model-view-update for live programming [71]. Researchers have successfully applied Direct Manipulation Programming systems to graphic creation (e.g., SVG) [14, 18, 30, 31, 64, 106], data analysis and visualization [44, 99, 108], document preparation [25, 59], and even general-purpose programming [31, 71]. Programming By Example systems generate programs that satisfy provided sets of input-output examples, dating back to the seminal THESYS system in 1977 [89]. Since then, the programming languages community has developed many Programming By Example synthesis techniques [1, 6, 20, 21, 27, 33, 34, 49, 55, 65, 68, 69, 72, 84, 90, 102], and, in recent years, the HCI community has developed novel interactions with such systems [17, 60, 74, 96, 104, 105, 107]. Programming By Demonstration systems generate programs that generalize one or more actions performed by the user. For example, the seminal SMARTedit system [47] generalizes repetitive text edits, and CoScipter [50], Ringer [5], Rousillon [13], and ScrapeViz [45] generate programs for web automation tasks. Programming By Demonstration has also been applied broadly in the domain of robotics [4], as well as in other areas of HCI like making "mashups" [53, 97] and spreadsheet programming [12].

In all of these tools' respective settings, users struggle *not* with knowing an intended output, but with constructing a program that produces that output. In this work we seek to extend output-centric programming to contexts where the situation is reversed, and the user's primary difficulty is not in writing a program suitable to a specification, but in working with a program to discover what they intend their program to do. Therefore, we do not argue that Programming by Scaffolded Demonstration competes with existing output-centric approaches, or that it is applicable to the same domains in which those approaches have found success. Rather, we offer Programming by Scaffolded Demonstration as an extension of this line of work to situations where users face the question of how they might discover what is possible to accomplish with a program, and, through doing so, what they might wish to program.

3.3 Example and Design Galleries

Previous work in *example galleries* and *design galleries* has focused on helping users search for relevant examples, tune numerical parameters in their programs, and explore natural language image generation. For example, researchers have created such systems to create galleries of stylistically-relevant webpages [48, 78] and to synthesize example API usage snippets [24] While these works recognize, as we do, that understanding many different possibilities can be valuable for a user, these systems focus on finding and displaying relevant *examples* of the work of others from existing corpi. Similarly, Subbaraman et al. [86] discuss the "collecting' behaviors of users on the OpenProcessing platform, using the gallery to visually navigate through examples of work from other users. In our work, we're interested in helping the user explore the possibilities *of their own program*.

Parametric generation of designs has been explored in computer graphics [58], 3D modeling [91], parametric design [103], and data science [98]. Like interactive numerical sliders or color pickers (as we see in our study), these systems help users explore a variety of constants that can be used for a given argument—e.g., values ranging between 0 and 1 used as a smoothness parameter. In contrast to our system, these tools do not explore possible *expressions*, only constants. Recent work has also used a design gallery approach for natural language image generation Dreamsheets [2]. Parametric design works share our focus on supporting exploration, but their interactions revolve around defining particular, fixed axes on which a design can be permuted—then permuting the designs by selecting from a set of numerical constants or selecting from a set of related natural language texts.

In a data science context, Voyager [98] uses a design gallery much like the parametric design tools described above, but with the fixed axes defined *in advance* by Voyager. Specifically, the items that can be permuted are: (i) what data attribute is mapped to the `x` and `y` axes, (ii) features of marks added in the visualization, and (iii) faceting of the graph by rows and columns. The tool automatically generates a data structure representing a range of choices for each of these axes, and the resulting visualizations are displayed in a gallery. Another tool that uses fixed axes (this one specifically for algorithmic art) is GenerativeGI [22]. Like Voyager, GenerativeGI uses a fixed set of axes chosen by the tool's designers, but it uses a much larger set of axes. For example, one of the author-written axes offers the option to use either `'edgy'` or `'curves'` as the `'flow-field-type'`. The authors provided a separate set of axes for each of 15 different art "techniques." Although GenerativeGI does not show a gallery (instead the end user provides the fitness function that the tool should optimize for, and GenerativeGI outputs the highest-scoring variant), the overall fixed-axis approach is similar. When the tools described in this section explore possible values for a given axis, they essentially substitute constants—numerical constants, strings from a set of strings, data attribute names from a set of attribute names—into a configuration file. In contrast, a PBSD

system uses the design of an existing programming language to modify and extend arbitrary and complex programs. With Perpend, for example, users can insert new expressions into any JavaScript program: writing new functions, using Perpend inside the function body; writing loops, then use Perpend in the loop body; and introducing not a fixed number literal or string literal, but an expression that uses the user's own variables (as we saw in our user study). As we discuss in Section 7.2.1, a key reflection from our study is that the design of a user's general-purpose programming language (thought of as a material) already decides dimensions upon which valid expressions can be composed, and revealing those dimensions to the user without imposing additional constraints can be a valuable aim of programming interface design.

Edgeworth [70] generates a gallery of visualization variants for use as the answers for multiple choice exam questions. The input is the correct visualization, and the output is a set of similar but ultimately incorrect visualizations. After visualization generation, the user sees a set of visualizations, then picks the ones they want to use in their multiple-choice question. Although each visualization was generated by a Penrose [101] program, the user does not see or directly interact with the programs. The *single-shot* visualization generation process applies randomly selected mutations across the correct visualization in order to generate new programs, which then generate new visualizations. Our goal with PBSD was to study an interaction loop for authoring programs, where a system would provide the user with a set of modified programs and their outputs, and the user would be able to edit their program both manually and by selecting a displayed modification from the system. While Edgeworth uses program variations to produce a gallery, it does not offer this program authoring loop.

3.4 Creative Coding

Digital drawing systems have explored how to support users in writing programs which complement their manual practices of digital drawing [39, 40, 51] and specifying procedural behavior of visual elements through direct manipulation [29, 41, 42, 100]. Jacobs et al. [39] explore the relationship between programming digital brushes *procedurally* and the *manual* use of those brushes; as we discuss in Section 7.2.1, we extend this distinction to the activity of programming itself, showing how procedural and manual aspects arise in authoring programs which generate visual images in addition to serving as tools for use in later creation.

Node-based `p5.js` environments include Stamper [11] for program composition, Spellburst [3] for program generation via natural language prompting, and Quickpose, for version control [76]. Juxtapose [28] supported interaction designers to experiment with program alternatives for interface development. While such systems aim to support what Ingold would call "iteration" [37] via a node-based composition environment, Perpend aims to support users in understanding the range of possible edits to their program text through visualization, leaving the `p5.js` composition environment otherwise unchanged.

Domain-Specific Language (DSL) contributions in creative coding include danceON [73], which supports motion-tracked visualizations, and P5fab for 3D fabrication [85]. Because Perpend makes an orthogonal contribution, we could imagine Perpend being used in tandem with these domain-specific interventions.

Generative artists' discussions of experimentation and discovery especially resonate with Ingold's idea of *following materials*. Hobbes describes the process of narrowing a "messy idea" from a multi-dimensional space of possibilities into a final artwork through sampling and exploration of the space of possible outputs from any single program [95]. Maclean discusses the importance of a "bricolage" style of programming in creative coding [61], characterized by a tight loop of action and observation, similar to Schön's idea of a "conversation" with materials [80]. These authors both emphasize an ongoing *responsiveness* for practitioners of generative art. With Ingold in mind, we suggest that, among other things, generative artists are responsive to their program and programming language *as a material*, which is to say, in terms of how the textual program may be edited and the consequences of that change on the output of the program. While Hobbes focuses on how generative artists understand the space of possible outputs generated by individual programs, in this work we consider how programming systems can support practitioners in exploring their space of possible *programs*.

Li et al. explore the values and motivations of visual artists [52]. We see PBSD as aligned with their call for software tools which do not "hinder how artists traditionally learn through manual engagement with materials"—in our case, the material is the programming language.

Recently, a system featured in a study by McNutt et al. [64] introduced many direct manipulation features for a `p5.js` web editor for educational purposes. Because we juxtapose Perpend with this system in our user study, we defer to Section 5 for a longer discussion of the differences between these two systems.

4 System Design and Implementation

4.1 System Design

4.1.1 Gallery. Our implementation of the gallery view is a scrollable grid. The first row displays program edits that can be inserted at the user's current cursor position. The following rows display edits that insert function calls one line above or one line below the user's current cursor position. Each row corresponds to a single function; each item in a row uses different arguments for the function. By visualizing a row of possible patches with the same API call, we aimed to help the user develop an intuition about the possibilities of their current program at the cursor's position.

4.1.2 User Control of Patch Location. The gallery refreshes patches whenever a user clicks a location in the program text with their cursor. Rather than introduce a separate interaction, we wanted the gallery to ambiently update as the user navigated their program. With the current system and representative sketches from our study, edits display within the range of 440 milliseconds to 560 milliseconds after a click or selection.

The user's cursor location controls which subtree of the program's Abstract Syntax Tree (AST) will be altered by Perpend's candidate modifications. Some patches replace the AST subtree associated with the particular expression in which the user has placed their cursor. Other patches leave that AST subtree untouched but insert a new subtree as a neighboring branch of the AST.

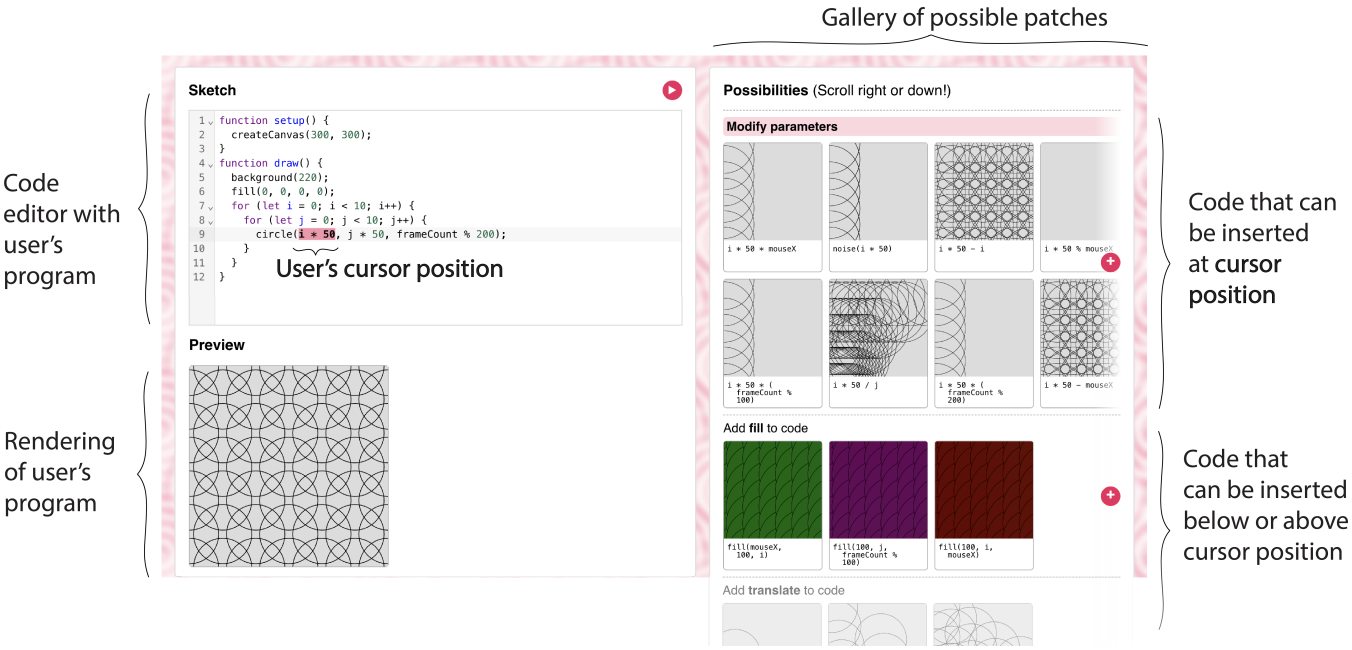


Figure 2: The Perpend Interface On the left side, the interface includes a code editor with the user's program and a preview showing the visual output of the user's current program, similar to the existing `p5.js` web editor. On the right side, the interface contains a gallery-style visualization of possible changes (that we introduce as "patches" in Sections 4.1.2 and 4.1.3) to their program. To invoke and refresh the gallery view, the user clicks on a location in their program text. When the user clicks on an expression within their program (e.g., `i * 50`), each gallery item displays its patch applied to that expression (e.g., `(i * 50) * mouseX`); each gallery item also displays the visual output produced by running the user's program with the patched expression in place.

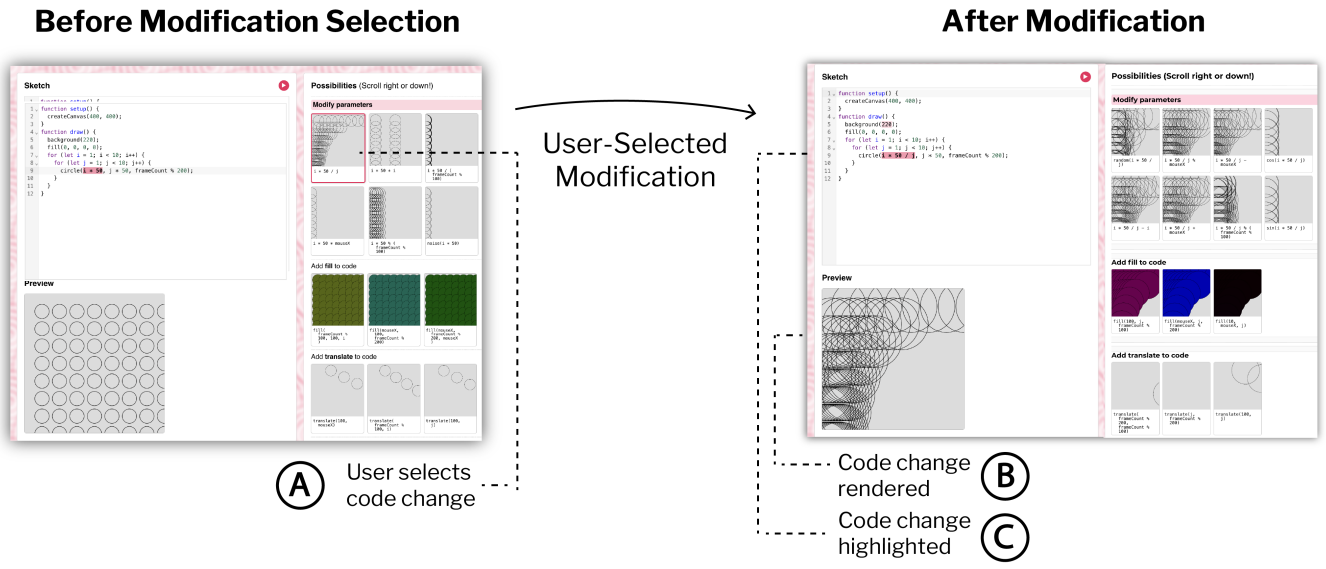


Figure 3: Perpend's Interaction Flow: On the first screen (left), Perpend displays the visual output of possible modifications at the user's current cursor position in their program. When a user selects a modification from the grid of possibilities (**A**), Perpend applies that modification to the user's current program. On the second screen (right), the new program change is highlighted in the user's program (**C**), and the updated program is now displayed in the main render window (**B**).

4.1.3 Generating Patches. Perpend displays single-location, syntactically small patches to the program text. By only changing the program in the area that the user specifies, we hypothesized that the system would help users both more easily understand the effects of a single change and understand the possibilities of their program through the gallery as a whole. In contrast, DM programming tools (e.g., [32]) ensure that program changes correspond to output changes, but this can result in complex changes to multiple locations in the program. We aimed to introduce small changes at a single location, to support users in quickly grasping the relationship between the displayed output, the candidate edit, and their current program.

Each patch is an AST subtree that can either be substituted for an existing subtree of the program AST, or else inserted as a new subtree within the program AST. We chose to limit Perpend to patches that require only a single-location AST manipulation (i.e. ruling out `p5.js` functions like `beginMatrix()`, etc.) and individual arithmetic operators (*, +, etc.). Because we are not trying to predict or suggest an outcome the user intends, but rather trying to visualize possible next edits to the program text at a specific cursor location, we chose small program edits that would result in a runnable program, based on the syntax and semantics of the JavaScript language and `p5.js` library. We also included patch *conditions* that ruled out insertions which would be guaranteed to not visually affect the output, such as adding `fill()` calls when no `fill`-able shapes had been added.

Patches use a range of constant values which would be significant in the context of many Processing API calls (e.g., 0, 1, -1, 100) alongside any user-defined variables in scope at the current cursor location, which can be automatically identified via analysis of the AST. In addition, we included commonly used built-in `p5.js` global variables (e.g., `frameCount`). See the Appendix A for a table of functions, variables, constants, and operators used in Perpend edits.

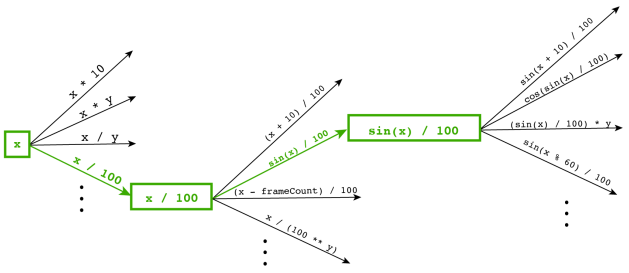


Figure 4: With a Programming By Scaffolded Demonstration (PBSD) system, a user navigates a space of possible program modifications.

4.1.4 Sampling and Displaying Possible Patches. To display programs to the user, Perpend takes a random sample of generated patches and applies them to the user's current program. That is, Perpend either inserts the new AST subtree into the program AST or replaces an existing AST subtree, then executes code generation to acquire a complete JavaScript program. For each patch, Perpend runs the complete, patched program in order to generate the corresponding output.

4.2 Programming By Scaffolded Demonstration in Context

4.2.1 User-Provided Outputs Not Required. As we reviewed in Section 3.2, Programming By Scaffolded Demonstration (PBSD) is situated in a long, rich tradition of output-centric programming tools, including (1) Direct Manipulation Programming, (2) Programming By Example, and (3) Programming By Demonstration. While we see PBSD as an extension to this line of work, there is a key difference between PBSD and these existing approaches: PBSD **does not require that the user provide partial or full desired outputs**. In all other output-centric programming systems, the user must provide a partial or full desired output, which a user may not be able to (or want to) provide.

(1) In **Direct Manipulation Programming**, users use direct manipulation interactions (interaction with canvas elements, menus, etc.) to tweak the program's current output directly. The system returns a corresponding modification to the program itself. If the program p produces output o (that is, if $p() = o$), and then the user tweaks o to make o', then the system generates p' to ensure that $p'() = o'$. For example, a user may drag an element on a visual canvas to change that element's position. The new canvas with the updated element is o', and the programming system generates the corresponding program p' which generates the new canvas with that element's position changed (so that p' generates o'). The user is required to specify the new updated canvas o' themselves as a direct specification of what the new program should generate.
(2) In **Programming By Example**, users provide pairs of input-output examples. The system returns a program p that meets the criteria that for all input-output pairs (i, o) the user provided, $p(i) = o$. For example, the user might provide examples of a desired text transformation to capitalize the first letter of each word, providing input-output pairs (i, o): (cat, Cat), (hat, Hat). The system returns a program which generates each output given the paired input. The user is required to specify the input-output pairs (i, o) themselves as examples of the desired program behavior.
(3) In **Programming By Demonstration**, users provide input-output examples, as in Programming By Example, but they also provide the sequence of steps they take to transform the input into the output, which can make it easier for the system to find a program that matches the input-output pairs. The system ensures that for every demonstration $(i, \text{intermediate}_1, \ldots, \text{intermediate}_n, o)$ the user provides, $p(i) = o$. For example, a user demonstrates how to scrape the total number of publications for a handful of authors in a publication database (i = name, databaseEntryForName, statsTableForEntry, o = numberOfPublications). The system generates a program which produces the correct output for each corresponding input, guiding its synthesis using the intermediate steps. The user is required to specify the input-output pairs and the intermediate steps between $(i, \text{intermediate}_1, \ldots, \text{intermediate}_n, o)$ as demonstration of the desired program behavior.

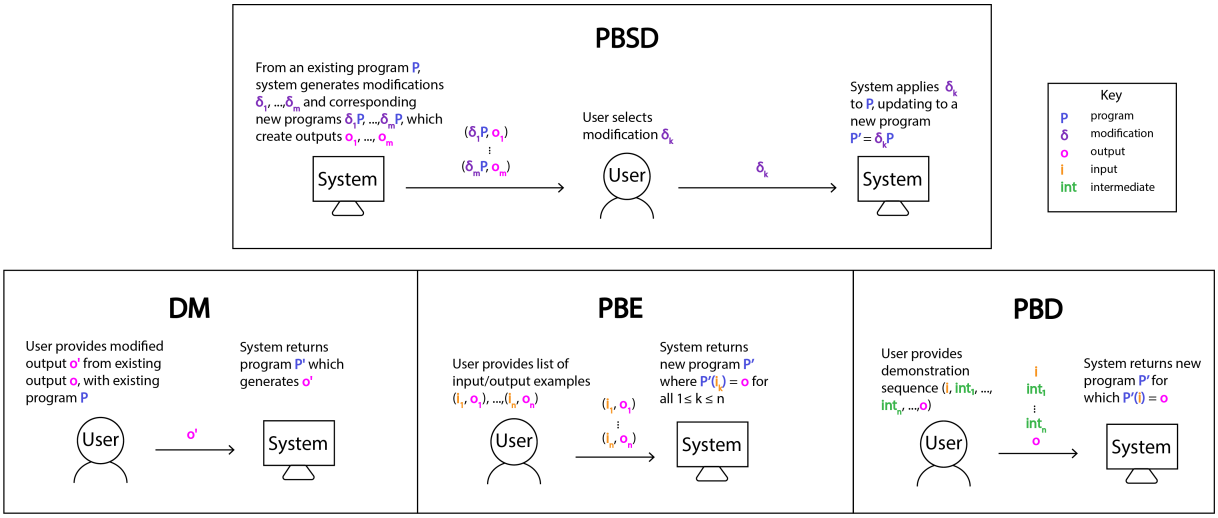


Figure 5: One interaction loop of Programming By Scaffolded Demonstration (PBSD), Direct Manipulation (DM), Programming by Example (PBE), and Programming by Demonstration (PBD). In DM, PBE, and PBD, the user provides some form of output to the system. In PBSD, the user provides a program and selects an output without providing an output themselves.

In contrast, users of Programming By *Scaffolded* Demonstration *select between* possible intermediate values by selecting possible modifications (See Fig. 4). At each step, a PBSD system generates a range of possible program modifications and their outputs as candidates for possible next steps (a *system-provided* analogue to the *user-provided* intermediate$_k$ steps in Programming by Demonstration, as shown above). The user then selects among those options, and the selected candidate becomes the user's new current program and output, and the interaction loop starts again. *In this way, PBSD systems do not require a user to provide intermediate$_k$ or any other user-specified output o or o′ themselves.* In Perpend's case, users are only required to specify a location in the program text at which to introduce modifications, and a gallery of corresponding possible modifications and their outputs is displayed (although the user remains free, of course, to edit the program text themselves).

More formally, in PBSD the user provides a work-in-progress program P, and the system, given the structure of the programming language, provides a set of small textual modifications $\delta_1, \ldots, \delta_m$ with corresponding updated programs $\delta_1 P, \ldots, \delta_m P$ and generated outputs $o_1, \ldots, o_m$. The user then selects among these modification-output pairs—say, (δ_k, o_k)—to arrive at a new program $P' = \delta_k P$ (such that $P'() = o_k$). This process can be repeated as desired and interleaved with arbitrary program edits. PBSD can therefore provide an output-centric programming experience even when the user cannot provide a desired output themselves. PBSD thus complements the wide variety of existing output-centric approaches that excel when outputs are known ahead of time, supporting a different group of users rather than competing with existing approaches.

4.2.2 Program vs. Output "Continuity." Perpend offers different guarantees than other output-centric programming tools. For example, in Direct Manipulation programming systems, the system guarantees that running the program will always produce the same output that the user created via direct manipulation; the program can change arbitrarily to meet this goal. We might think of the *program output* as "continuous" from one modification to the next in this case, in that the user can understand the new output as the result of a small (or single) modification to the previous output. But because the resulting program can change in arbitrary ways to produce the new output, the *program text* would not always be continuous, in this sense.[1] In contrast, with Perpend, the possible modifications displayed to the user are syntactically small program transformations. Perpend therefore provides a "continuity" of the *program text*, while allowing drastic changes in the visual output. Because the goal of PBSD systems is to help users understand their possible next steps in editing their *program text*, we limited the displayed modifications to the smallest textual modifications which would still produce complete programs: modifying and adding to individual arithmetic expressions, and adding individual API calls. As we discuss in Section 7.1, Perpend's small, local edits seemed to support participants in maintaining a sense of *program-continuity* (and, perhaps as a result, program understanding) as they edited their programs through the interface.

5 User Study Design

To understand the role of Perpend as a Programming by Scaffolded Demonstration tool in users' programming and artistic practice, we asked nine Processing artists to use Perpend in a user study. Because we argue that a key difference between Perpend and Direct Manipulation programming systems is that Perpend generates many contextual, possible modified program outputs in contrast to a user manually constructing a new target output, we juxtaposed Perpend with an existing direct manipulation `p5.js` interface [64] (referred to as DMEditor here) to understand their differences in

[1] Some systems, like Sketch-n-Sketch [14, 30, 32], attempt to make these potentially-large edits as logical as possible by preserving the structure of the original program to the greatest extent possible, but even sensible and readable program edits can result in large textual differences to the program.

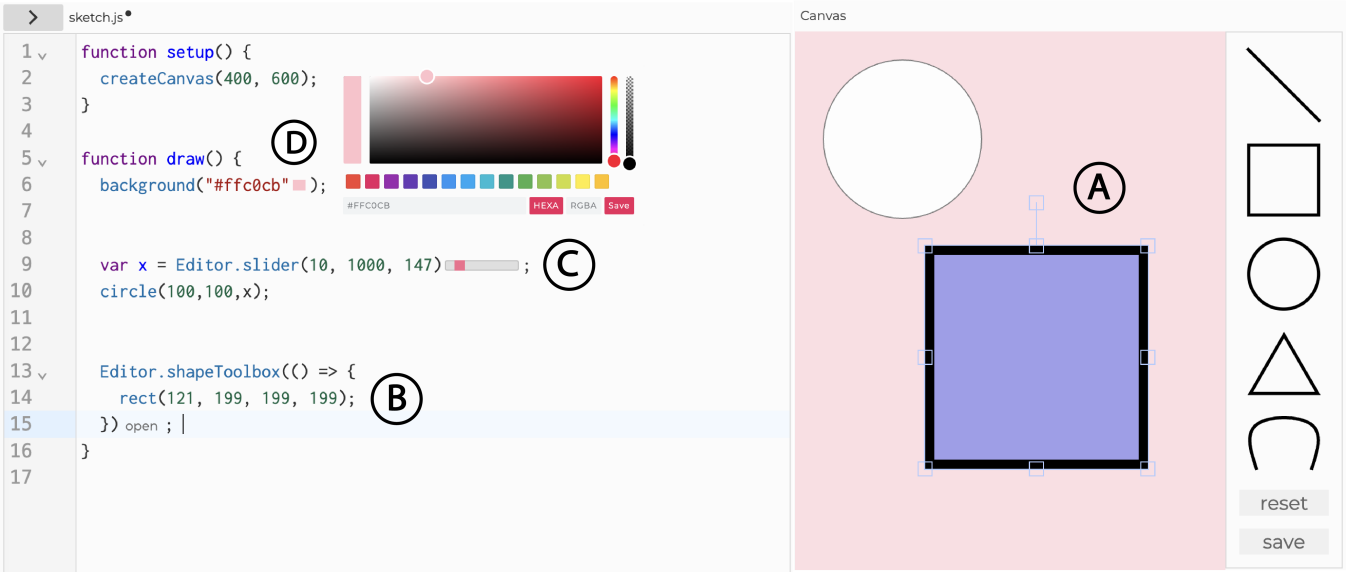


Figure 6: The Direct Manipulation `p5.js` interface (DMEditor) we juxtaposed with Perpend in our user study. A. DMEditor includes a direct manipulation `shapeToolbox`, which enables users to place and modify vector graphics onto the canvas which are automatically updated within the program text (B). **C**. DM's `slider` abstraction, which can replace any numerical value. **D**. DM also includes a direct manipulation color picker.

practice. Besides the inclusion of formative research, this study aligns well with the principles of *Comparative Structured Observation* [56], especially in our emphasis on participant reflection between the two interfaces and our aim to understand how PBSD and Direct Manipulation Programming can be valuable concepts for the design of programming systems rather than experimentally validating which design was better suited to a pre-defined task.

5.0.1 DMEditor. DMEditor includes three main direct manipulation features: a direct manipulation `shapeToolbox`, which the user instantiates by writing `Editor.shapeToolBox()` and clicking the "open" button, and enables them to place and modify vector graphics onto the canvas which are automatically updated within the program text; a `Slider` abstraction, which the user instantiates by `Editor.slider(start, end, current)` and can replace any numerical value; and a direct manipulation color picker we denote `ColorPicker`, which displays either a continuous color range or a preset selection of colors such as "orange" and "red." Aside from these features, the editor includes a text editor pane and a live preview of the program's output.

5.0.2 Participants. We recruited participants from social media (Reddit and Instagram) in addition to academic email lists. A screening survey was used to select for participants who had experience with Processing and identified as artists or creatives. On a scale of 1-10, participants self-reported a mean of 5.3 for their Processing experience (min 3, max 8), and on a scale of 1-5 their mean self-reported programming experience was 3.5 (min 3, max 4). All participants reported using Processing for art practice, interaction design, or design work. See Appendix B for additional participant details. Perpend is designed to help practitioners understand the space of small modifications to their program text, and not primarily intended as an education tool for novice users to learn either programming or *p5.js* specifically. We therefore selected users with only significant programming experience and prioritized users with p5.js expertise. See Appendix B for participant's self-rated programming experience and creative coding background. In a counter-balanced design, participants used each interface for ~45 min, and described their experiences in a semi-structured ~30 min interview at the end. We asked participants about their experiences with both interfaces (they were not told which interface had been made by the researchers until the conclusion of the study), and to contextualize what they had made in the session. Many decided to vocalize observations or experiences during their working sessions, although they were not required to do this. Participants were given the choice to start a new project or continue existing work, but were otherwise given no direction for how they should use either interface besides a small tutorial in each interface's functionality. Participants were compensated for their time with a 60 USD gift card at the conclusion of the study session. Interviews were recorded on Zoom and transcribed for analysis. Figure 8 shows participants' answers to the screening survey. Columns 2 and 3 report measures of self-reported `p5.js` and programming expertise, following the question design in [19]. The last two columns' questions were open response and similar answers were grouped into categories (e.g., "Art," "artistic practice," and "art practice" were grouped into "Art practice").

5.0.3 Analysis. We analyzed the data through a *reflexive thematic analysis* (RTA) [10]. After transcribing the interviews we generated an initial set of codes. Because the recordings also included participants' programs and their programming behaviors, we continually consulted the video recordings themselves to contextualize verbal statements within the programs and programming behavior

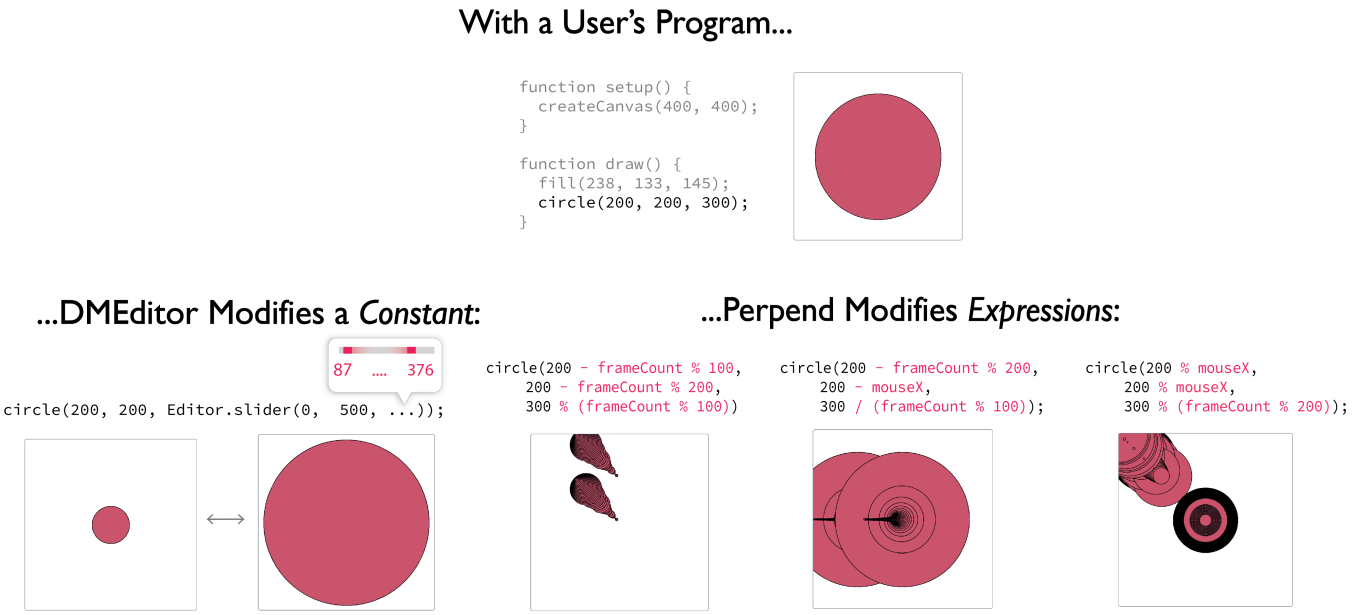


Figure 7: Illustration of the Different Functionalities of DMEditor and Perpend. Top: A user's program which includes a `circle()` call which they can explore with either tool. **Left**: DMEditor's `Slider` enables the user to change the third parameter of `circle()`, the radius, through direct manipulation, changing the value of a numeric constant. **Right**: When a user places their cursor on `circle()`, Perpend displays possible *expression* modifications to each parameter field, utilizing arithmetic expressions (`/` and `%`) and `p5.js` global variables (`frameCount` and `mouseX` **).**

those statements occurred in tandem with. We then began iteratively developing themes while returning to the data to recode and recontextualize. Because our analysis included how participants used the features of both interfaces and the programs they wrote alongside an interview, we developed our initial themes from either codes from the transcription or codes associated with programming behavior. Part of our iteration on themes identified connections between these, and we generated new themes which captured both.

6 Findings

6.1 Characterizing User Stories

Before exploring our more analytic themes, we begin with a brief characterization of user stories as a way to describe the interfaces in more concrete detail.

6.1.1 DMEditor. All participants used `shapeToolBox` in some form. P6 used `shapeToolBox` for the majority of their session, creating and moving a large number of circles and bezier curves. Other participants (P1, P2, P9) used `shapeToolBox` at the beginning of their sessions to create initial arrangements of three to five different shapes, which they utilized in their later (textual) program development. P4 and P7, who created simple drawing interfaces, and P3, who worked on their code from Perpend, used `shapeToolBox` the least, only using `shapeToolBox` once or twice in the session.

All participants used `ColorPickers` to toggle the color of their sketch's background or shapes at least once. For some of these participants (P1, P2, P8, P9), this only took a few moments of deciding on a color using the tool. P4 and P7, who wrote programs for a simple drawing interface, spent more time with `ColorPicker` to change and experiment with the color of their brushes. P6 used `ColorPicker` the most out of all participants, matching colors to a reference image and later to adjust the colors of gradients.

Participants (P1, P2, P4, P7, P8) used `Sliders` to control numerical parameters of shape functions such as the `radius` of `circle()` and the start and end of `line()`. Most participants used `Slider` at least once in their code; P6 only used `Slider` once to test it out, P4 and P7 used `Slider` to adjust the parameters of one function, and P1 and P8 used `Sliders` the most, creating a collection of at least five `Sliders` and using them for various functions.

6.1.2 Perpend. All participants used the gallery at least once to accept modifications. P1, P3, and P8 spent 15-20 minutes of their session scanning through and accepting Perpend edits; P7, who continued working on their drawing interface from DMEditor, used Perpend's modifications the least, only accepting a modification once to modify their brush; all other participants fell somewhere in between.

Some participants (P2, P4, P5, P6) spent more time in the beginning of their session working on their program text before scanning and accepting modifications from Perpend; we noticed that these participants also accepted more modifications from the "Modify Parameters" row as opposed to accepting modifications that added new function calls. Other participants (P1, P3, P8) accepted more modifications that added functions early on in their session, resulting in longer series of successive function calls.

P1, P2, P4, and P7 spent 10–15 minutes of their session scrolling through the rows of Perpend's gallery and examining the visual

outcome of Perpend's modifications closely, either before accepting a given modification or after accepting a modification, in order to explore it further. A few participants (P1, P2) requested more edits by clicking the "+" button to see more entries in a specific row, but scrolling the gallery was a more popular interaction.

6.2 "Dialing In" vs. "A Sense of Possible Directions to Move"

Many participants found DMEditor most helpful for moments of "dialing-in … refin[ing]" (P1) and "honing in" (P3) on small visual or program goals; on the other hand, many participants found Perpend most helpful for "figuring out what can be played around with" (P8) and "expand[ing] my immediate sense of possible directions to move." (P1)

6.2.1 "Dialing In": DMEditor Helped Participants Narrow Possibilities. Many participants found DMEditor most useful for "refin[ing]"(P1) and "honing in"(P3) on a specific visual or program intention. For P1, sliders were especially important for small, visual intentions for their program, which they called "micro-goals" (P1). They expressed that these "micro-goals," such as selecting a specific color, were best accomplished by using DMEditor's tools to find the parameters (and their combinations) to produce the intended visual result: "I'm trying to combine my understanding of what's happening with my eyes and my understanding of the system" (P1). For P2, using DMEditor required planning ahead—formulating a specific intention for both the visual output (e.g., changing the size of a circle) and the program (e.g., creating a slider for the third parameter of `circle()`) in order to effectively utilize those tools: "I'm thinking really hard about what I want to do, because unless I think about it really hard I won't be able to come up with what functions I'm supposed to use" (P2). Because DMEditor best supported exploration along pre-defined axes, participants found DMEditor features most helpful in moments where they had clear visual or program specifications.

6.2.2 "A Sense of Possible Directions to Move": How Participants used Perpend to Discover New Possibilities . In contrast to DMEditor, Perpend's range of displayed modifications widened participants' sense of what was possible in their program. In this section, we explore how Perpend shaped users' intentions at three different moments of creative direction: when participants had a clear direction, seeing Perpend's possibilities sometimes steered them to unexpected places; when participants had questions about a specific section or line of their program, they structured their programs to bound their exploration with Perpend to those areas; and when participants did not have a sense of creative momentum at all, they looked to Perpend to find new paths.

Surprise: Participants Followed Perpend's Displayed Possibilities in Unexpected Directions. Sometimes, even when participants had a sense of creative direction while using Perpend, they changed course due to a displayed possibility they saw. P1, remarking on Perpend's gallery, said, "these are so cool that I have to click on them ... even though that's not really what I have in mind." Even when P3 thought they "may have screwed up and it's not doing what I think it's doing," they added "that's maybe the beauty of it, that these actually look pretty interesting ... These are kinda great, actually." P5, reflecting on their experience with Perpend's gallery, said that the gallery encouraged them to "try out new functions and new ways to put things together." P9 found Perpend's gallery "so much cooler" since the gallery visualizes a *space* of "multiple options on the side" as opposed to `p5.js`'s web editor, where code "compiles at the time that you change your code." Perpend's gallery widened participants' sense of how possible API calls and expressions could change the output of their program, and they sometimes steered away from existing directions.

Bounded Exploration. Some participants (P1, P2, P4) had intentions that surfaced questions about specific lines or regions of their program, and they used Perpend to explore these questions within the bounds of that part of the program. P1, for example, while clicking into a `fill()` call within a `for` loop with an index variable `i`, said, "The third parameter, the blue, I'd like to edit the blue, see what ideas [Perpend] has. Sick. This is exactly what I'm talking about. `10` over `i`. Brilliant!". When P4 looked at their animated arrangement of `circle()` calls in their `for` loop, they wanted the circles to "interact more," so they clicked one of the position parameters of their `circle()` function. Once they saw an option that visualized the result of adding `random()` around the parameter `j * 30`, they said, "Ooh! randomness. That's a good idea." When participants did not have specific a visual output in mind but had a region of interest in their program, they could use Perpend to explore the possibilities of their program in a targeted way.

Participants (P1, P2, P4) actively sought out moments of bounded exploration by writing functions with placeholder parameters to see what was possible to express with them. For example, P2 wrote `drawFish(100,100,100)` and moved their cursor onto the function name and its parameters in order to bound their exploration with Perpend. P2 described using Perpend in this way as "giving [Perpend] a hook into what [they] wanted it to change, as opposed to changing the entire thing" and that using Perpend "felt very additive" to their existing programming practice, adding a "measured amount of craziness" (P2) to explore within the bounds they expressed with their program structure. In these ways, participants could articulate what they intended to explore into the structure of their program, bounding Perpend to display the possible edits of a function call written in order to explore it with Perpend.

Perpend Helped Find New Paths when the Current One Ran Out. When participants (P1, P2, P7, P8) struggled to find direction, Perpend gave them new questions to pose—it helped participants "pick up momentum" (P1) when they had difficulty seeing where they might go next. By "just clicking" (P8) around their program text, Perpend's continually updating gallery moved P8 to "a good starting point" that they could "start building from" (P8). In P2's words, seeing Perpend's "visualization of the space of options of where I can go next" made "[their modifications on the program] feel more like choices, and less like things that happened because [they] failed to articulate [themselves]" (P2). Perpend's gallery provided a visualization of possible modifications so participants could better understand the space of where they might go next, helping them understand what they might *intend* to express by showing them what would be *possible* to express. Perpend displayed valid modifications given the design of the language to give participants a sense of what might be possible, which, when they had temporarily

run out of creative "momentum" (P1), gave them new directions in which to continue their work.

6.3 Shifting Focus Between The Program and Its Output

Some participants (P1, P2, P3, P6) discussed two different ways of working with `p5.js`: First, in a *visual-to-program* style, where the practitioner has a visual output in mind that they seek to instantiate with changes to their program, and second, in a *program-to-visual* style, where the practitioner modifies their program to explore its emergent visual result. P2 described this distinction as working "programmatically" (working with the program to understand its possibilities) vs. "thinking like someone who draws" (working with the visual output to form intentions for the program). In this section, we describe first how DMEditor seemed especially suited for this *visual-to-program* style, finding programs which created target visual outputs. By contrast, Perpend seemed to support both styles, and additionally seemed to support working between them more easily.

6.3.1 DMEditor Helped Participants Use a Visual-to-Program Style. DMEditor's `ColorPicker` and `shapeToolBox` encouraged a visual-to-program style, which was well-suited to tasks such as matching sky and cloud colors to a reference image (P6), or drawing a fish by arranging a circle and triangle in the appropriate relative positions and sizes (P2) in `shapeToolBox`. What we call working *visual-to-program*, P2 described as "thinking representationally" (P2). As P2 said sarcastically, "I draw [a] circle... I draw [a] triangle... [It looks] like [a] fish now" (P2). Similarly, P6 described how `shapeToolBox` encouraged them to view the code as an afterthought: "I would say I almost didn't pay attention to the code … because at that point it was just a bunch of numbers and I could just move [the shape directly in `shapeToolBox`]."

6.3.2 Perpend Supported Participants in Both Visual-to-Program and Program-to-Visual Activities. In contrast to DMEditor, Perpend supported participants both when focusing on program output, "working backwards" (P3) to program text (*visual-to-program*) and when focusing on modifying their program to see the visual result (*program-to-visual*).

How Participants Used Perpend in a Visual-To-Program Style. Many participants (P1, P2, P3, P4, P7) accepted modifications from Perpend's gallery based on visual output, and "work[ed] backwards" (P3) to understand how the program modifications produced that visual change. When clicking on the `fill()` function, P7 tested one modification, saying "This [modification] kind of works though. I think it's color-coded based on what frame we're on? It's actually kind of helpful." After accepting modifications to their `circle()` function, P7 looked at the result and said "these are quite interesting. I think it's this rotate that's doing something strange with the `log()`. Oh, and there's a `mouseX` involved, so it should depend on where I am." Reflecting on this visual-to-program interaction, P3 pointed out the work required to regain an understanding of their program after they had accepted a modification: "… You're just trying to look for interesting aesthetics. And you can work backwards from the interesting aesthetics, like 'That's great, how did the system that created it get [to the visual output]?'"

How Participants Used Perpend in a Program-to-Visual Style. Some participants (P1, P2, P4) explored intentions for their program text with Perpend's gallery. These intentions could be expressed through Perpend by clicking on a specific parameter such as "the *x*-position of the `circle` function" (P4) and "[seeing] what [Perpend is] changing" (P4), or "The third parameter, the blue, I'd like to edit the blue, see what ideas [Perpend] has" (P1). Another notable example of this was when P2 made the `drawFish(x, y, width, height)` function and clicked on it to bound the edits that would appear in the gallery. In each case, Perpend helped users explore questions originating in the program text through the display of visual possibilities.

6.3.3 How Perpend Supported A Synergy between The Two Styles. While the previous sections discussed *visual-to-program* and *program-to-visual* separately, our findings also suggested that Perpend supported participants in more flexibly moving between these styles in the course of their work. By displaying the *visual* results of small *programming* changes, Perpend helped P2 use `p5.js` "like a drawing tool" (P2) and "a programming tool" (P2) simultaneously: "… I have both interests [in] being a … visual artist and also a programmatic—'What will happen if I use math in this way?'—[artist] … and sometimes I have trouble meshing those two things because I think [p5.js] interactions are built for doing it one way or the other. I like how [Perpend] encouraged me to be able to do both." P2 mentioned that "[With Perpend,] I still feel like I'm making those visual artistic choices" (P2) with their textual programming without "getting lost in a documentation page trying to figure out what order parameters are supposed to be in" (P2). Many participants (P1, P2, P4, P5, P8, P9) mentioned the usefulness of Perpend's gallery in selecting modifications from visual interest, so they could, as P4 put it, "see visually, 'Oh! I like that. Let me do that'" (P4). Characterizing this interest as visual *in contrast to an interest in the program* would be misleading, however, because participants seemed to be interested in the visual outputs *because they corresponded to interesting programs*. In contrast to DMEditor, where "what you see [is] what you get" (P1), Perpend's gallery of edits prompted questions about the underlying programs which generated the displayed visual outputs: "You can work backwards from the interesting aesthetics, like 'That's great, how did [Perpend] get [to the visual output]?'" (P3). For P3, "interesting aesthetics" were ultimately in service of exploring the possibilities of the *program*: "There's not only the space of what your code is actually outputting, there's the space of what's possible with a very small change, and you also want to explore that. And that's something that [Perpend] can do really well." In contrast to thinking about `p5.js` as a purely visual medium—"This can just be like Illustrator but 10 thousand times slower … that's fine, but that's not the best way to use the tool" (P1)—the unique aspects of `p5.js` as a procedural medium were exemplified for P1 by Perpend's gallery of visually-discontinuous, but program-continuous possibilities: "I see [Perpend's gallery] and I'm like 'Yes, take me to that weird flying triangle'" (P1).

6.3.4 Shifting Between the Output and the Program Required Maintaining A Working Understanding of the Program. While some participants enjoyed focusing on the displayed outputs of Perpend over the code modifications themselves (Section 6.3.2), participants (P1, P3, P4, P6, P8) still required a working understanding of their

program in their explorations. P1 said "I feel like there's a limit to how many suggestions I can accept before I lose my mental model." P4 emphasized this point by discussing how manually writing a program was directly tied to their working understanding: "It seems like the process of being able to create the code and decide the arguments yourself is pretty important to maintaining this mental model in your mind of, 'this is where everything is.' It's only possible because I'm the one who originally put [the code there]" (P4). Although these participants still chose to explore visual possibilities and integrate modifications into their programs with Perpend, they emphasized a need for a working understanding of Perpend's program modifications before continuing.

6.3.5 *Participants Tweaked and Reused Perpend's Modifications; DMEditor Distanced Participants From Their Program.*

DMEditor Distanced Participants From Their Programs. Participants (P2, P5, P6) sometimes struggled to connect their work in DMEditor back to their program text, as DMEditor seemed to encourage working *only* in terms of the visual output. P6 said, "[`shapeToolBox`] took away the association of what's on the screen and what's [in the code]." P2 created a visual representation of a fish using `shapeToolBox` but, when they tried to animate the fish in their program, ran into problems creating a suitable abstraction. They struggled to associate the visual representation with the parameters in the corresponding program text in order to abstract the numbers in `circle()`'s and `triangle()`'s fields into function parameters. P5 created a bezier curve petal in `shapeToolBox` but failed to rotate the petal around its center because they struggled to understand the eight fields in the `bezier()` function and their relationship to the orientation and position of the curve overall. While DMEditor enabled participants to create new programs through Direct Manipulation, the distance between the visual interface and the generated code gave some participants difficulty in utilizing the generated program text.

Participants Tweaked, Reused, Repurposed Perpend Modifications. We observed that many participants (P2, P5, P6, P8) altered, repeatedly used, and ultimately repurposed constructs they first discovered in the Perpend gallery. Some participants discussed how Perpend's modifications expanded their `p5.js` vocabulary with new concepts and constructs. P6 described how seeing Perpend's modifications in the context of their program gave them an understanding of how various expressions (i.e. sinusoid functions, temporal variables like `frameCount`, and modular arithmetic) could be applied elsewhere in their program. They said this differed from using an example found on the internet because Perpend was displaying a modification *to their current program*, and so they could more easily apply "the concept" to other places. Specifically, they pointed to the importance of seeing the visual result of using a `sin()` function around *an existing expression they had selected within their current program*—not understanding what `sin()` does abstractly in the general case, but understanding how it transformed an element of their own program: "Because it's not just the code, it's the intention behind choosing to put something within a `sin()`" (P6). P2 mentioned how Perpend broadened their *p5.js* vocabulary, analogous to other resources for learning about `p5.js`: "So sometimes I want [`p5.js`] to visually do something. I don't really know how to do that, so I look for example code...and I'll have the same feeling of like, 'What the heck does that do? Let me just copy and paste that in, and then see what it does to mine.' And [Perpend] feels like that, but faster …" Overall, Perpend's gallery of possible modifications to participants' programs helped some participants incorporate Perpend's modifications as new additions to their creative programming pallette.

7 Discussion

7.1 Output-Directed Programming Without User-Provided Output Specifications

7.1.1 *Programming by Scaffolded Demonstration in Other Domains.*

While Perpend was designed for creative coding with *p5.js*, Ingold argues that understanding the possibilities of one's materials is not unique to artistic domains, and is rather a part of any domain of skilled making. But where might PBSD be most useful? Reflecting on our study findings, we propose four characteristics for domains which might be well-suited for PBSD systems:

(1) **Domains in Which Users Write Programs to *Discover*.** As we discussed in the introduction, PBSD aims to support output-centric programming in contexts where the task of writing a program is "to discover what it should do" [82]. We might expect, then, that PBSD might be most applicable to domains in which users author programs in order to explore what they intend for those programs to do. Domains such as exploratory data science [8], software prototyping [9], and web and interface design [28] are promising candidates. Additionally, we might expect domains where users can easily express the intended outputs of their program but have difficulty expressing the program itself (domains to which existing output-centric approaches are well-suited) to be less-promising domains for PBSD systems, as we see PBSD extending the range of applications of output-centric programming rather than to compete with existing approaches. As we found in our comparison to DMEditor, however, some domains may fall somewhere in between. Users in these domains may be best supported with a range of output-oriented ways to author programs—PBSD being one of them—and moving flexibly between approaches as their work changes.

(2) **Informative Visual Outputs.** One reason we chose to develop Perpend for *p5.js* is that users are already creating visual artifacts as their program output. For Perpend's gallery of possibilities, the user's ability to quickly scan outputs visually seemed essential. We would predict that successful PBSD systems in future domains will require ways to quickly display informative views of a program's output so that users can understand their possible next steps at a glance. This does not mean, however, that PBSD is only suited to programming situations which already produce visual images. Future work might explore how to build expressive visualizations of outputs so that users can quickly read and select from many options. For example, for programs that compose transformations on objects like tables or dictionaries, we could show visualizations of the transformed objects. For programs that return single values (e.g., numeric values, string values), we could display the outputs in a sortable,

filterable, or searchable table rather than gallery view. For programs with location outputs, we could display outputs on a map instead of a gallery. Effective ways to visually communicate changes in program output are likely to be highly specific to a particular domain and use-case, and designing them would be an exciting question for future work. Future HCI research might also investigate new programming languages and interfaces for users to specify bespoke output visualizations for their unique situations, tailoring the visualizations in a PBSD system to their personal requirements.

(3) **Complex Languages.** One key finding from our user study (Section 6.2.2) was that users could leverage the features of a powerful language like JavaScript to structure Perpend's gallery. Through user-defined functions, for loops, and local variables, participants changed the material possibilities of their programs—that is, changed what potential next programs were available through small textual edits. We think PBSD is especially suited to contexts where users are building complex programs, and pre-defined axes to compare and generate programs fall short. For example, in a hypothetical PBSD system in a data transformation and visualization, we might build within a language that has control flow, user-defined abstraction, and complex data processing (such as R [75]) rather than implement a system with a config-style specification grammar like Vega-Lite [79]. Crucially, these language features would enable users to use the structure of their program text as their implicit dimensions for exploration, rather than relying on predefined ones. Because the question a PBSD system is designed to answer is not "what program produces a desired (or even desirable) output?" but rather "what does my language make possible for my current program?", we might imagine a PBSD system to be most useful in data cleaning and transformation stages, rather than visualization, where the question "what visualizations are possible given my data?" can lead to perceptually deceptive graphics without more structured approaches [62]. In this context, a PBSD system might display multiple visualizations of table transformations to convey the effect of a modification on their data. As we saw in Perpend, we might expect users to engage with such a gallery of modifications as *possibilities* which orient users to what program modifications and constructs are available to them to apply and tweak themselves, not as *suggestions* for literal changes to the program which the user accepts verbatim. Interestingly, we expect a PBSD tool can be useful in a complex language *even without covering the whole language*. For example, programmers write React within larger JavaScript programs. A PBSD system to support React programmers might trigger only when the user is editing the JSX markup (for context, React components return JSX markup); when the programmer clicks into the business logic above, the PBSD window would disappear. Patches in this PBSD system could be limited to modifying the JSX (including adding or modifying React components). The gallery would display screenshots of the modified webpages.

7.1.2 Designing for Itineration, Rather than Iteration. Output-directed systems have often focused on building ways for users to express procedural behavior in radically different ways than writing a textual program. By contrast, we argue that Perpend was successful as an interface for "itineration" [37] because it focused on using visual outputs to deepen participants' engagement with the program text. In other words, by displaying the edits the programming language made possible in the user's program, Perpend attempted to show users "the grain" of the language, without aligning with a specification the user had in mind.

We presented *Programming by Scaffolded Demonstration* in Section 4.2 as a new approach to programming systems design, but the questions we have for PBSD systems also change accordingly. If the questions Direct Manipulation Programming systems designers ask are "What representations of the work does a user already know how to provide, and how can a DM system synthesize a program from them?" and "What new interactions are possible when users can specify procedural behavior through visual manipulation?", PBSD designers might ask "How can a PBSD system better situate the user in the possibilities of their language?"

These questions are not necessarily in conflict. In fact, some of our participants (P2, P6, P8) requested using both tools in a single system because they were each valuable for separate reasons—to use Ingold's vocabulary, DMEditor for "iteration" and Perpend for "itineration" [37], or, in the words of our participants, "dialing in" and navigating via a "sense of possible directions to move." Because DMEditor and PBSD serve different roles, we envision programming systems which include both, supporting users in complementary moments of their work.

7.1.3 Supporting Users in Leveraging Program Structure To Guide Exploration. Because Perpend displayed modifications only at the cursor's program location, users intentionally bounded their exploration to specific regions and authored parts of their programs to direct Perpend's gallery (Section 6.2.2). Future PBSD systems should consider how they might better leverage program structure to enable finer-grained focusing by the user. Perpend used the current cursor location, but future systems might experiment with interactions such as dragging over a region, highlighting multiple disjoint parts of the user's program, or selecting based on annotations like indentation and comments.

Another way to do this, we propose, is to consider programming language design decisions which enable this finer-grained focusing. For example, in `p5.js`, `fill()` calls set a global color for all shapes that are drawn afterwards until another color is set. This presented difficulty for P3 and P5 when they only wanted to change the color of a single shape. By changing the language, for example adding an optional color parameter to be passed to a shape directly, or a `with` construct which could specify a context, users might have more flexibility in how they bound Perpend's exploration. Language design decisions like these could open possibilities for users to direct a PBSD system more precisely by writing their program in different ways.

7.1.4 Supporting Users in Expanding Their Own Vocabulary of Program Modifications. One surprise from our study was how some participants utilized the modifications displayed by Perpend elsewhere in their programs. Beyond understanding a single program

modification in a sufficient way to continue working, some participants added those modifications to their own working vocabulary in Processing, as P6 did with `sin()` and `frameCount` in Section 6.3.5. We have already discussed how future work could support users in understanding modifications selected from Perpend, but PBSD could go further: better supporting users in adding new tools into their own programming toolboxes. Two potential strengths of Perpend in this respect were, first, in displaying the textual program change immediately below the visual output, so that users could understand the textual change which would lead to that image; and second, in changing the program only at the cursor position, so that users could understand the consequences of the change *in their program* and *at that specific location in the program.* In Section 6.3.5, P2 described Perpend as an easier alternative to inserting examples from the internet into their program. Rather than displaying an entirely different program, of which a user like P2 is only seeking a small part to integrate into their own work, Perpend displays *the user's current program* with only the single modification. We think this aspect of Perpend played a crucial role for P6, where Perpend helped them incorporate `sin()` into their vocabulary by showing them *how and why* they might use it: "Because it's not just the code, it's the intention behind choosing to put something within a `sin()`."

7.1.5 Supporting Perspective-Shifting. As we found in Section 6.3.1, participants found DMEditor most useful for working in what we called a *visual-to-program* style, in some cases shifting focus away from the program significantly. By contrast, participants using Perpend seemed to keep both their program and its visual output in focus during their work (Section 6.3.2). Rather than aim to support users composing programs through outputs *instead of* working in the program text (as many output-centric systems do), we intended Perpend to help users understand their outputs *and the corresponding programs* as two crucial halves of the programming work. As we discussed in Section 6.3.3, for many of our participants, a "visual[ly]" (P2) interesting image was not in practice separable from an *image generated by an interesting program.* By displaying simple modifications at one place in the program, both in terms of the textual change and the generated visual output, Perpend seemed to support participants in working between both styles more easily. For domains like Processing, where both the program and the program output are meaningful parts of the work, output-directed programming systems need to support this perspective-shifting.

7.2 Understanding and Designing For Programming as a Material

7.2.1 Supporting Programming as a Manual Practice. Jacobs et al [40] distinguish between *manual* practices, such as drawing or sculpting, and *procedural* practices, such as programming. *Manual* practices, on this picture, are "intuitive" (non-theoretical) and learned through "physical practice" and "direct material engagement." *Procedural* practices are "analytic" and accomplished through theory or formal knowledge. A key argument the authors make with their system, *Dynamic Brushes*, is that "… many manual artists are interested in both" and that "the right computational interface could enable manual artists to bridge analytic and intuitive practices." In our study we found evidence for the complementary claim: that programming (a "procedural" practice) is in some aspects a *manual* practice, and that the right interfaces (i.e., Perpend) can support *programmers* as well in both the "analytic" and "intuitive" aspects of their work. To see how programming can be a "manual" practice, we turned back to Ingold. He gives us one way to think about the manual/procedural distinction in terms of a practitioner's expertise and their "feel" for their craft:

> The carpenter who has a feel for what he is doing is one who can bring the many concurrent variations with which he must engage more or less into phase with one another. This calls for continual correction, in response to an ongoing perceptual monitoring of the task as it unfolds. — Tim Ingold, *The Textility of Making* [37]

Reflecting on our conversations with participants, many highlighted this "feel" for their `p5.js` programs. Ingold helps us see that developing such a "feel" might be an important part of their *manual* expertise. P8 told us that their work often consists in "see[ing] where [their program] goes" as they are "carving" and "shaping" the program's behavior improvisationally. They prioritized "just getting [a program] up" into the editor and "figuring out how to work with it once it's up there" without a "solid idea of what I want [my `p5.js` compositions] to look like in the end." P2 described their *itinerant* work in `p5.js` rather explicitly: "I actually might want an idea that's different from what I'm envisioning, but is a result of what I tried to articulate?" (P2). P1 told us how Perpend prompted them to consider the ways that an analytical ideal of working with Processing was unrealistic, and reflected on how programming with Processing felt like following a *material*: "[Perpend] sort of crushes this idea I have of the platonic, perfect Processing person who is able to manifest their vision through code perfectly with no major mistakes. They're just like, 'I know exactly what … sort of style [or] color thing I want' versus the person who is like, 'I admit that the world is unknowable, and art is unknowable and what it may make me feel is in the hands of others'" (P1).

In each case, their expertise is not only in mastering a set of logical rules, but also in knowing how to attend to the programming language as a material and to follow its "grain"—in having a *feel* for the program. In this light, we argue that programming should be seen as also having meaningfully manual dimension, and that this manual aspect of programming can and should be supported by programming tools.

Tool Design for Programming as a Manual Practice. If programming is a manual practice, how might tools recognize and support this? One idea came from P4, discussing how they had occasionally paused development to regain understanding while using Perpend. They described how the manual practice of programming supported their working understanding: "Maintaining [a] mental model [of the code is] … only possible because I'm the one who originally put that [code there]" (P4). Why did this happen? One explanation is that Perpend *interrupted their rhythm.* Because Perpend modified the user's code directly, the normal rhythm of *manually typing the program change* never happened. If, as we argue, we should see programming as partly a *manual* practice, then P4's skill in maintaining a "feel" for the program could be a crucial way they

maintain a working understanding. This resonated with a participant quote from McNutt et al. [63]: "in the arts, there's a real value to slowing down and taking the hood off things. Because it lets you ask critical questions, [such as] if you're truly engaging with it as a medium." If part of the "rhythm" of programming is learning to see the local possibilities and consequences of a program change *as one is typing it*, then it is not surprising that P4 needed to pause: *Perpend had interrupted their rhythm.*

Thinking about programming as a *manual* practice prompts us to ask how we might study, support, or simply not interrupt a programmer's sense of "rhythm", their "ongoing perceptual monitoring" and "continual correction" as they work. If part of the expertise of a programmer is in developing a "feel," in addition to formal knowledge or logical reasoning, programming tools need to be aware of how they impact both.

7.2.2 Program(mming Language)s as Materials. We have argued that programming is an engagement with a material, namely a programming *language*. The goal of Perpend (and a PBSD system in general) is to support users in understanding the "grain" of this material, or the space of programs reachable from small textual edits, which is determined by the structure of the programming language. In this sense, Perpend aims to bring users' awareness to their programming languages as *spaces of possibility*—with all of their contours, rough edges, and well-worn grooves. In the case of programming languages like JavaScript and `p5.js`, the question of their "grain"—what they make difficult, easy, possible, or necessary to say—is a question of the choices of a language designer. Supporting users in seeing and following these grains would then also entail making more visible that programming languages are not a neutral substrate, but are designed, and could be designed otherwise.

But drawing a such a stark divide between the user's program and their programming language would be misleading. We argue that programmers are *already* designing their language, and therefore, they are already changing their material. An example comes from Section 6.2.2, when P2 wrote a new function in order to bound Perpend's gallery to specific parameters: "… I even found myself writing `fill()`-something, so I could see what other things [Perpend] could give me for `fill()`, but I could [also] put `drawFish()` [a custom function] and click on it and see what it could give me for `drawFish()`. I think that was me saying, "I want possibilities for this specifically" (P2). By creating a new abstraction, P2 *added a new contour to their programming language* in order to direct Perpend's gallery.[2] While Perpend focused on helping participants understand the programming *language* as a material, we argue now that *programs themselves already shape that material.* When programmers "follow the material," they are thus shaping that material as well.

As tool designers, understanding a program as a material might suggest building tools which better leverage users' programs as *already* articulated spaces of possibility. Rather than asking how interfaces can provide alternative representations to compose programs aside from the program text, we might ask how programming tools can better respond to, and be shaped by, the structure of the user's program itself. The user's program, in this sense, is already a rich articulation of where they wish to go.

[2]Loretta Rose Guarino, in her survey on abstraction: "We can view every large programming project as involving the design, use, and implementation of a special-purpose programming language." [26]

8 Limitations and Future Work

The primary limitations for this study were the duration of the study session and only having a single session per participant. Unfortunately two hours was the most we felt comfortable asking participants to schedule without burden, but it was very short from the perspective of observing artmaking. Future work should observe tool use over a longer continuous duration. Secondly, we were also limited in the number of sessions. Running multiple observations with each participant would allow us to see if participants responded differently to DM and Perpend as they became more comfortable with them. While of course more participants would help us see a more diverse set of artists, we do not expect artists to be a homogeneous group, nor would we expect any sample to be representative of artists as a whole. That being said, studying a greater number of participants would have let us observe more perspectives and more ways of using Perpend.

While Perpend was not designed to help novice programmers learn Javascript or *p5.js*, future PBSD systems might explore how to help novices discover new features of their languages or refine fluency with existing concepts. As we saw in Section 6.3.5, Perpend seemed to support some participants in deepening their understanding of *p5.js* constructs. For users unfamiliar with more basic programming constructs however (e.g., control flow, abstraction, variables), displayed edits to introduce those constructs may not be very effective if they require significant modification of a user's program at multiple places in the program text. On the other hand, supporting expert users for long-term engagement with PBSD systems might be best accomplished by supporting users to author their own (and perhaps more complex) rewrite rules and support user-defined palettes of insertions from existing code in a user's corpus.

9 Conclusion

This work explored the role of output-centric programming for domains where the user writes a program "to discover what it should do, not to satisfy a prior specification" [82]. We found that one answer is to support the user's interaction with the programming language as a *material*: as a space of possibilities to be followed, rather than a substance onto which a prior intention is imposed. We introduced *Programming By Scaffolded Demonstration* to help users navigate this space of possibility opened by their language, enabling users to compose a program through selection of valid textual modifications and their resulting outputs. While other output-directed systems often seek to mediate between a user and their textual program, our study on Perpend showed that output-directed systems can instead support the programming text itself as already a rich site of surprise and exploration for both the program and its output. We found that Perpend supported users in taking up their own programs as spaces of possibility, modifying their program structure to change the space of possible programs which Perpend displayed. By displaying small modifications to the program *text* given the structure of their language, Perpend helped users engage

with their programs not just as efficient means to a desired output, but as an expression through a specific material—*a language*—with its own syntactic and semantic contours.

Acknowledgments

We thank the reviewers for their suggestions and insights. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1752814 and is supported in part by NSF Grants 2243822, FW-HTF 2129008 and CA-HDR 2033558, as well as by gifts from Google, G-Research, Adobe, and Microsoft. Chasins is a Chan Zuckerberg Biohub Investigator.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044 (CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 934–950.

[2] Shm Garanganao Almeda, J.D. Zamfirescu-Pereira, Kyu Won Kim, Pradeep Mani Rathnam, and Bjoern Hartmann. 2024. Prompting for Discovery: Flexible Sense-Making for AI Art-Making with Dreamsheets. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–17. doi:10.1145/3613904.3642858

[3] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 100, 22 pages. doi:10.1145/3586183.3606719

[4] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A Survey of Robot Learning from Demonstration. *Robotics and Autonomous Systems* 57, 5 (May 2009), 469–483. doi:10.1016/j.robot.2008.10.024

[5] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 748–764. doi:10.1145/2983990.2984020

[6] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 168:1–168:27. doi:10.1145/3360594

[7] Gabrielle Benabdallah and Nadya Peek. 2024. Technical Mentality: Principles for HCI Research and Practice. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–14. doi:10.1145/3613904.3642720

[8] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, 25–29. doi:10.1109/vlhcc.2017.8103446

[9] J. Brandt, P.J. Guo, J. Lewenstein, S.R. Klemmer, and M. Dontcheva. 2009. Writing Code to Prototype, Ideate, and Discover. *IEEE Software* 26, 5 (Sept. 2009), 18–24. doi:10.1109/ms.2009.147

[10] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. doi:10.1191/1478088706qp063oa

[11] Cameron Burgess, Dan Lockton, Maayan Albert, and Daniel Cardoso Llach. 2020. Stamper: An Artboard-Oriented Creative Coding Environment. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–9. doi:10.1145/3334480.3382994

[12] Kerry Shih-Ping Chang and Brad A. Myers. 2017. Gneiss: Spreadsheet Programming Using Structured Web Service Data. *Journal of Visual Languages & Computing* 39 (April 2017), 41–50. doi:10.1016/j.jvlc.2016.07.004

[13] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. doi:10.1145/3242587.3242661

[14] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 341–354. doi:10.1145/2908080.2908103

[15] Laura Devendorf and Kimiko Ryokai. 2015. Being the Machine: Reconfiguring Agency and Control in Hybrid Fabrication. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, Seoul Republic of Korea, 2477–2486. doi:10.1145/2702123.2702547

[16] Kristin N. Dew and Daniela K. Rosner. 2018. Lessons from the Woodshop: Cultivating Design with Living Materials. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–12. doi:10.1145/3173574.3174159

[17] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376442

[18] Conal M. Elliott. 2007. Tangible Functional Programming. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 59–70. doi:10.1145/1291151.1291163

[19] Janet Feigenspan, Christian Kastner, Jorg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring Programming Experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, Passau, Germany, 73–82. doi:10.1109/icpc.2012.6240511

[20] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 599–612. doi:10.1145/3093333.3009851

[21] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. doi:10.1145/2737924.2737977

[22] Erik M. Fredericks, Jared M. Moore, and Abigail C. Diller. 2024. GenerativeGI: creating generative art with genetic improvement. *Automated Software Engineering* 31, 1 (May 2024), 23. doi:10.1007/s10515-024-00414-3

[23] Verena Fuchsberger, Martin Murer, and Manfred Tscheligi. 2013. Materials, Materiality, and Media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Paris France, 2853–2862. doi:10.1145/2470654.2481395

[24] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API Usage Examples at Scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–12. doi:10.1145/3173574.3174154

[25] Camille Gobert and Michel Beaudouin-Lafon. 2022. I-LaTeX : Manipulating Transitional Representations between LaTeX Code and Generated Documents. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3491102.3517494

[26] Loretta Rose Guarino. 1978. *The Evolution of Abstraction in Programming Languages*. Technical Report. Carnegie Mellon University.

[27] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/1926385.1926423

[28] 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 Press, Monterey, CA, USA, 91. doi:10.1145/1449715.1449732

[29] Sonia Hashim, Tobias Höllerer, and Jennifer Jacobs. 2023. Drawing Transforms: A Unifying Interaction Primitive to Procedurally Manipulate Graphics across Style, Space, and Time. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–15. doi:10.1145/3544548.3580642

[30] 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)*. Association for Computing Machinery, New York, NY, USA, 379–390. doi:10.1145/2984511.2984575

[31] Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:29. doi:10.4230/LIPIcs.ECOOP.2022.16

[32] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. doi:10.1145/3332165.3347925

[33] Martin Hofmann and Emanuel Kitzelmann. 2010. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and*

Program Manipulation (PEPM '10). Association for Computing Machinery, New York, NY, USA, 93–100. doi:10.1145/1706356.1706375

[34] Qiantan Hong and Alex Aiken. 2024. Recursive Program Synthesis Using Paramorphisms. *Proc. ACM Program. Lang.* 8, PLDI (June 2024), 151:102–151:125. doi:10.1145/3656381

[35] Darris Hupp and Robert C. Miller. 2007. Smart Bookmarks: Automatic Retroactive Macro Recording on the Web. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. Association for Computing Machinery, New York, NY, USA, 81–90. doi:10.1145/1294211.1294226

[36] David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. Association for Computing Machinery, New York, NY, USA, 125–134. doi:10.1145/1166253.1166274

[37] T. Ingold. 2010. The Textility of Making. *Cambridge Journal of Economics* 34, 1 (Jan. 2010), 91–102. doi:10.1093/cje/bep042

[38] Tim Ingold. 2013. *Making: Anthropology, Archaeology, Art and Architecture*. Routledge, London ; New York.

[39] Jennifer Jacobs. 2017. *Dynamic Drawing: Broadening Practice and Participation in Procedural Art*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[40] 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*. ACM, Montreal QC Canada, 1–13. doi:10.1145/3173574.3174164

[41] 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*. ACM, Honolulu Hawaii USA, 395–405. doi:10.1145/2642918.2647375

[42] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014. Draco: Bringing Life to Illustrations with Kinetic Textures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Toronto Ontario Canada, 351–360. doi:10.1145/2556288.2556987

[43] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–11. doi:10.1145/3173574.3173748

[44] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. doi:10.1145/3379337.3415842

[45] Rebecca Krosnick and Steve Oney. 2024. ScrapeViz: Hierarchical Representations for Web Scraping Macros. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, USA, 300–305. doi:10.1109/VL/HCC60511.2024.00040

[46] Bruno Latour. 2005. *Reassembling the Social: An Introduction to Actor-Network-Theory*. Oxford University Press, UK. doi:10.1093/oso/9780199256044.001.0001

[47] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (Oct. 2003), 111–156. doi:10.1023/A:1025671410623

[48] Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen Brafman, and Scott R. Klemmer. 2010. Designing with Interactive Example Galleries. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems - CHI '10*. ACM Press, Atlanta, Georgia, USA, 2257. doi:10.1145/1753326.1753667

[49] Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 70:2048–70:2078. doi:10.1145/3571263

[50] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. Association for Computing Machinery, New York, NY, USA, 1719–1728. doi:10.1145/1357054.1357323

[51] Jingyi Li, Joel Brandt, Radomír Mech, Maneesh Agrawala, and Jennifer Jacobs. 2020. Supporting Visual Artists in Programming through Direct Inspection and Control of Program Execution. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–12. doi:10.1145/3313831.3376765

[52] Jingyi Li, Sonia Hashim, and Jennifer Jacobs. 2021. What We Can Learn From Visual Artists About Software Development. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3411764.3445682

[53] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-User Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. Association for Computing Machinery, New York, NY, USA, 97–106. doi:10.1145/1502650.1502667

[54] Richard Lin, Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Bjoern Hartmann. 2021. Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, New York, NY, USA, 1039–1049. doi:10.1145/3472749.3474804

[55] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 109:1–109:29. doi:10.1145/3408991

[56] Wendy E. Mackay and Joanna McGrenere. 2025. Comparative Structured Observation. *ACM Trans. Comput.-Hum. Interact.* 32, 2 (April 2025), 14:1–14:27. doi:10.1145/3711838

[57] Lambros Malafouris. 2013. *How Things Shape the Mind: A Theory of Material Engagement*. MIT Press, Boston, Massachussetts, USA. doi:10.7551/mitpress/9476.001.0001

[58] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. 1997. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., USA, 389–400. doi:10.1145/258734.258887

[59] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 127:1–127:28. doi:10.1145/3276497

[60] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 291–301. doi:10.1145/2807442.2807459

[61] Jon McCormack and Mark d'Inverno (Eds.). 2012. *Computers and Creativity*. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/978-3-642-31727-9

[62] Andrew McNutt, Gordon Kindlmann, and Michael Correll. 2020. Surfacing Visualization Mirages. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3313831.3376420

[63] Andrew Michael McNutt, Sam Cohen, and Ravi Chugh. 2025. Slowness, Politics, and Joy: Values That Guide Technology Choices in Creative Coding Classrooms. doi:10.31219/osf.io/83z94_v1

[64] Andrew M Mcnutt, Anton Outkine, and Ravi Chugh. 2023. A Study of Editor Features in a Creative Coding Classroom. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–15. doi:10.1145/3544548.3580683

[65] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 21:1–21:29. doi:10.1145/3498682

[66] Nick Montfort. 2016. *Exploratory Programming for the Arts and Humanities*. The MIT Press, Cambridge, Massachusetts.

[67] Hedieh Moradi, Long N Nguyen, Quyen-Anh Valentina Nguyen, and Cesar Torres. 2022. Glaze Epochs: Understanding Lifelong Material Relationships within Ceramics Studios. In *Sixteenth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '22)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3490149.3501310

[68] Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. 2023. Program Synthesis Using Example Propagation. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, Michael Hanus and Daniela Inclezan (Eds.). Springer Nature Switzerland, Cham, 20–36. doi:10.1007/978-3-031-24841-2_2

[69] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. 2022. Predictive Synthesis of API-centric Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 40–49. doi:10.1145/3520312.3534866

[70] Wode Ni, Sam Estep, Hwei-Shin Harriman, Kenneth R. Koedinger, and Joshua Sunshine. 2024. Edgeworth: Efficient and Scalable Authoring of Visual Thinking Activities. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale*. ACM, Atlanta GA USA, 98–109. doi:10.1145/3657604.3662034

[71] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. doi:10.1145/3453483.3454059

[72] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. doi:10.1145/2737924.2738007

[73] William Christopher Payne, Yoav Bergner, Mary Etta West, Carlie Charp, R. Benjamin Benjamin Shapiro, Danielle Albers Szafir, Edd V. Taylor, and Kayla DesPortes. 2021. danceON: Culturally Responsive Creative Computing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–16. doi:10.1145/3411764.3445149

[74] Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 159:1–159:30. doi:10.1145/3428227

[75] R Core Team. 2023. R: A Language and Environment for Statistical Computing.

[76] Eric Rawn, Jingyi Li, Eric Paulos, and Sarah E. Chasins. 2023. Understanding Version Control as Material Interaction with Quickpose. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–18. doi:10.1145/3544548.3581394

[77] Casey Reas and Ben Fry. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, Cambridge, Mass.

[78] Daniel Ritchie, Ankita Arvind Kejriwal, and Scott R. Klemmer. 2011. D.Tour: Style-Based Exploration of Design Example Galleries. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology - UIST '11*. ACM Press, Santa Barbara, California, USA, 165. doi:10.1145/2047196.2047216

[79] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. doi:10.1109/TVCG.2016.2599030

[80] D. A. Schön. 1992. Designing as Reflective Conversation with the Materials of a Design Situation. *Knowledge-Based Systems* 5, 1 (March 1992), 3–14. doi:10.1016/0950-7051(92)90020-G

[81] Donald A. Schön. 2017. *The Reflective Practitioner: How Professionals Think in Action*. Routledge, United States.

[82] Mary Shaw. 2020. Myths and Mythconceptions: What Does It Mean to Be a Programming Language, Anyhow? *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020), 1–44. doi:10.1145/3480947

[83] Ben Shneiderman. 1982. The Future of Interactive Systems and the Emergence of Direct Manipulation†. *Behaviour & Information Technology* 1, 3 (July 1982), 237–256. doi:10.1080/01449298208914450

[84] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 326–340. doi:10.1145/2908080.2908102

[85] Blair Subbaraman and Nadya Peek. 2022. P5.Fab: Direct Control of Digital Fabrication Machines from a Creative Coding Environment. In *Designing Interactive Systems Conference*. ACM, Virtual Event Australia, 1148–1161. doi:10.1145/3532106.3533496

[86] Blair Subbaraman, Shenna Shim, and Nadya Peek. 2023. Forking a Sketch: How the OpenProcessing Community Uses Remixing to Collect, Annotate, Tune, and Extend Creative Code. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*. ACM, Pittsburgh PA USA, 326–342. doi:10.1145/3563657.3595969

[87] Krishna Subramanian, Ilya Zubarev, Simon Völker, and Jan Borchers. 2019. Supporting Data Workers To Perform Exploratory Programming. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems (CHI EA '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3290607.3313027

[88] Lucy A. Suchman. 1987. *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge University Press, UK.

[89] Phillip D. Summers. 1977. A Methodology for LISP Program Construction from Examples. *J. ACM* 24, 1 (Jan. 1977), 161–175. doi:10.1145/321992.322002

[90] Jerry Swan, Krzysztof Krawiec, and Zoltan A. Kocsis. 2019. Stochastic Synthesis of Recursive Functions Made Easy with Bananas, Lenses, Envelopes and Barbed Wire. *Genetic Programming and Evolvable Machines* 20, 3 (Sept. 2019), 327–350. doi:10.1007/s10710-019-09347-3

[91] Jerry O. Talton, Daniel Gibson, Lingfeng Yang, Pat Hanrahan, and Vladlen Koltun. 2009. Exploratory Modeling with Collaborative Design Spaces. In *ACM SIGGRAPH Asia 2009 Papers (SIGGRAPH Asia '09)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/1661412.1618513

[92] Jakob Tholander, Chiara Rossitto, Asreen Rostami, Yoshio Ishiguro, Takashi Miyaki, and Jun Rekimoto. 2021. Design in Action: Unpacking the Artists' Role in Performance-Led Research. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–13. doi:10.1145/3411764.3445056

[93] Martin Tricaud and Michel Beaudouin-Lafon. 2023. Revisiting Creative Behaviour as an Epistemic Process: Lessons from 12 Computational Artists and Designers. In *Proceedings of the 35th Australian Computer-Human Interaction Conference*. ACM, Wellington New Zealand, 175–190. doi:10.1145/3638380.3638395

[94] Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. 2021. Tools, Tricks, and Hacks: Exploring Novel Digital Fabrication Workflows on #PlotterTwitter. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–15. doi:10.1145/3411764.3445653

[95] Tyler Hobbs. 2025. Direction, Reaction, and Messy Ideas. https://www.tylerxhobbs.com/words/direction-reaction-and-messy-ideas

[96] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3411764.3445249

[97] Jeffrey Wong and Jason I. Hong. 2007. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. Association for Computing Machinery, New York, NY, USA, 1435–1444. doi:10.1145/1240624.1240842

[98] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, Denver Colorado USA, 2648–2659. doi:10.1145/3025453.3025768

[99] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 152–165. doi:10.1145/3379337.3415851

[100] Haijun Xia, Bruno Araujo, Tovi Grossman, and Daniel Wigdor. 2016. Object-Oriented Drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, San Jose California USA, 4610–4621. doi:10.1145/2858036.2858075

[101] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. Graph.* 39, 4 (Aug. 2020), 144:144:1–144:144:16. doi:10.1145/3386569.3392375

[102] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 141:860–141:883. doi:10.1145/3591255

[103] 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*. ACM, Seoul Republic of Korea, 1201–1210. doi:10.1145/2702123.2702398

[104] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3411764.3445646

[105] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. doi:10.1145/3379337.3415900

[106] Xing Zhang, Ruifeng Xie, Guanchen Guo, Xiao He, Tao Zan, and Zhenjiang Hu. 2024. Fusing Direct Manipulations into Functional Programs. *Reproduction Package for Article 'Fusing Direct Manipulations into Functional Programs'* 8, POPL (Jan. 2024), 41:1211–41:1238. doi:10.1145/3632883

[107] Zhanhui Zhou, Man To Tang, Qiping Pan, Shangyin Tan, Xinyu Wang, and Tianyi Zhang. 2022. INTENT: Interactive Tensor Transformation Synthesis. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3526113.3545653

[108] Parker Ziegler, Justin Lubin, and Sarah E. Chasins. 2025. Fast Direct Manipulation Programming with Patch-Reconciliation Correspondence. *Reproduction Package for Article "Fast Direct Manipulation Programming with Patch-Reconciliation Correspondence"* 9, PLDI (June 2025), 175:699–175:724. doi:10.1145/3729278

A Functions, Variables, and Operators used in Perpend Modifications

Shapes, lines, and related functions	`scale, rotate, stroke, noStroke, ellipse, fill, noFill, translate, line, quad, triangle, erase, noErase`
Sinusoids and numerical distributions	`sin, cos, tan, noise, random`
Variables	`frameCount, mouseX, mouseY, 100, 200`, User-defined variables
Arithmetic Operators	`*, +, -, /, %`

Figure 8: A list of functions, variables and operators used in Perpend's patches.

B Participant Programming Background and *p5.js* Experience

	Self-Rated Processing Experience 1 (Beginner) 10 (Expert)	**Self-Rated Programming Experience** 1 (Beginner) 5 (Expert)	**What do you use Processing/`p5.js` for?**	**What types of media do you make with Processing/p5.js?**
P1	7	3	Art practice, graphic design, interactive tool programming	Printed textiles, 2D and 3D art, interactive interfaces
P2	6	4	Art practice, teaching, for fun	2D art, animated art, interactive art
P3	4	3	Generative art	2D art
P4	3	4	2D Art	2D art, interactive interfaces
P5	4	4	Art practice	Interactive interfaces
P6	4	3	Art practice, school assignments	2D animated art, interactive art
P7	8	3	Art practice, school assignments	2D art, interactive art
P8	7	4	Art practice	2D art
P9	5	4	Design work, interaction design	Sound, interactive art, 2D art

Figure 9: While our participants used `p5.js` for a diversity of use cases, each had significant experience with both `p5.js` and programming in general.