

A Study of Editor Features in a Creative Coding Classroom

Andrew McNutt
University of Chicago
Chicago, IL, USA

Anton Outkine
University of Chicago
Chicago, IL, USA

Ravi Chugh
University of Chicago
Chicago, IL, USA

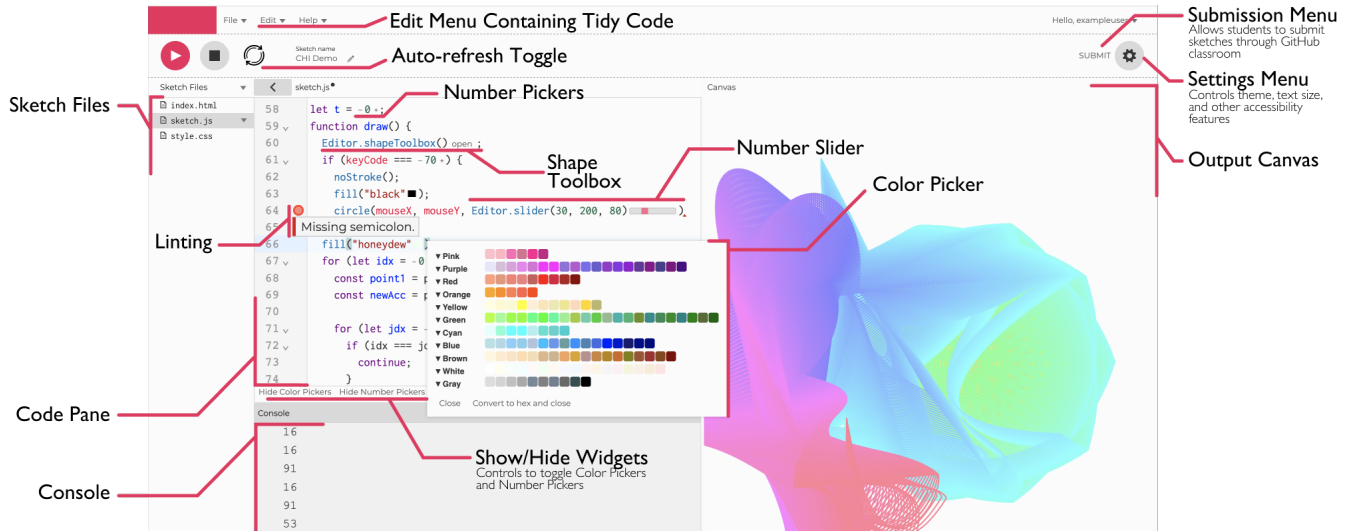


Figure 1: This modified p5 editor (dubbed **p5/y2**) was used in a creative coding course to study how students use and perceive various editor features including standard ones, such as linting and auto-formatting (“Tidy Code”), as well as more experimental features, such as live reloading (“Auto-refresh”) and a toolbox for bidirectionally manipulating shapes.

ABSTRACT

Creative coding is a rapidly expanding domain for both artistic expression and computational education. Numerous libraries and IDEs support creative coding, however there has been little consideration of how the environments themselves might be designed to serve these twin goals. To investigate this gap, we implemented and used an experimental editor to teach a sequence of college and high-school creative coding courses. In the first year, we conducted a log analysis of student work ($n=39$) and surveys regarding prospective features ($n=25$). These guided our implementation of common enhancements (e.g. color pickers) as well as uncommon ones (e.g. bidirectional shape editing). In the second year, we studied the effects of these features through logging ($n=39+$) and survey ($n=23$) studies. Reflecting on the results, we identify opportunities to improve creativity- and novice-focused IDEs and highlight tensions in their design—as in tools that augment artistry or efficiency but may be perceived as hindering learning.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions.acm.org.

CHI '23, April 23–28, 2023, Hamburg, Germany

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9421-5/23/04...\$15.00
<https://doi.org/10.1145/3544548.3580683>

CCS CONCEPTS

• **Human-centered computing** → *Human computer interaction (HCI)*; • **Software and its engineering** → *Integrated and visual development environments*.

KEYWORDS

Creative coding, Code editors, p5, Introductory programming

ACM Reference Format:

Andrew 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 (CHI '23)*, April 23–28, 2023, Hamburg, Germany. ACM, New York, NY, USA, 42 pages. <https://doi.org/10.1145/3544548.3580683>

1 INTRODUCTION

Creative coding is a rapidly expanding computational domain. It generally refers to programming work that “blur(s) the distinction between art and design and science and engineering” [66], encompassing pursuits such as generative art, embedded computing, audio editing, performative live programming, and countless others. Many libraries and languages have arisen to support this programming genre. Some are tuned to domain-specific purposes—such as Orca [55] or Tracery [25] which support creating procedural music and Twitter bots, respectively. Others simplify the process of many common artistic tasks (such as drawing and interactivity) without specializing in a specific area—as in openFrameworks [68] or Processing [87]. Among these general-purpose tools, those in

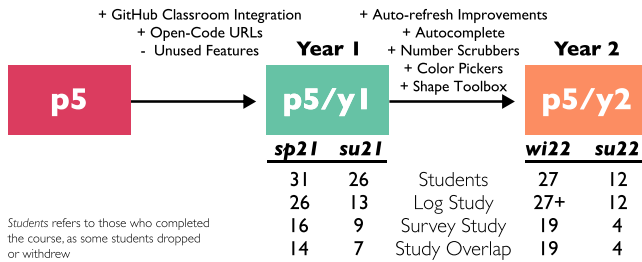


Figure 2: We modified the p5 editor before each year of a creative coding course. We conducted studies to observe student usage and perception of existing and modified features.

the Processing family—such as Processing itself and p5.js [1]—are particularly well known, having attracted large and active communities, exemplified by the prevalence of artist- and novice-focused educational media, like the Coding Train [92].

Beyond the potential for creative or artistic expression, this genre of work has long been embraced as means by which to teach introductory programming [42, 66, 83, 105]—an approach often referred to as *media computation* [40] within CS departments—as it may be easier for students to engage with material that interests them [8], and creative or artistic tasks may be more engaging to students [70] not invested in the more common CS Ed topics. Greenberg et al. [37] argue that creative coding-based introductions to computer science are more appealing to women, and create a more inclusive environment than traditional introductory CS curricula.

Despite the potential utility for both artistic expression and learning to code, there has been relatively little consideration of how to enhance creative coding environments to facilitate these goals. Following a trend exemplified by the development environment bundled with the Processing library, a number of creative coding toolchains come with their own environments, which are often tailored specifically for artists in their domain, as in Orca [55] or Tweakable [5]. For example, the p5 editor—a browser-based editor maintained by the p5 community that acts as a gateway to coding for often non-technical users—is intentionally simple, and has limited “features and frills” to make it easier to jump right into coding [3]. By definition, however, this guiding principle forgoes potential benefits of many standard IDE features (such as autocomplete), standard GUI features (such as color pickers), and more experimental features explored in research communities (such as bidirectional editing).

To shed light on the gap between creative coding tools and their goals for users, this work considers the following questions: *How might we refine and enhance standard tools to extend the creative reach of novices? What sorts of features do novices perceive to be beneficial in a creative coding environment?* We consider these questions in the setting of the p5 editor because of its ubiquity [66] in creative coding contexts, as well as for its simple and mostly standard form, which may inform the design of enhancements to more general-purpose programming tools.

Paper Summary. We conducted a series of studies that considered how students use a modified version of the p5 code editor in an introductory programming and creative coding course at a private research university in the US. Fig. 2 displays an overview of our

work, involving four course offerings spanning two academic years taught to college students (*sp21*, *wi22*) and to high-school students (*su21*, *su22*). While our studies are situated in a classroom, our work is not about pedagogy per se—rather, we focus on understanding the needs and perceptions of creative coding novices as exhibited across the length of a full programming course.

We used a modified version of the p5 editor (referred to as **p5/y1**) in the first-year courses (*sp21* and *su21*) and ran two studies. The first was a log analysis based on capturing code executions during the course ($n=39$). The second was a long-form survey that sought to understand student opinions and expectations about existing and hypothetical editor features ($n=25$).

The results of the first-year studies revealed opportunities to improve existing editor features as well as interest in several hypothetical features—including direct manipulation widgets for modifying colors and a bidirectional shape drawing system. Thus, we further augmented the p5 editor (**p5/y2**) in the second-year offerings (*wi22* and *su22*). Analogous to the studies in the first year, we monitored student behavior ($n=39+$) through an anonymized tracking scheme,¹ and solicited their opinions through an abbreviated version of our previous survey ($n=23$).

We identify several key themes based on the results of the studies.

T1 Simple static analysis seen as supportive. Tools supporting basic automated formatting and analysis—such as code “tidying” or linting—are well received by our novices. However, impolite [104] designs (which are those that do not respect user agency or act in an otherwise irritating manner) can lead to frustration.

T2 Overeager evaluation can overwhelm. Live programming can give immediate feedback on code changes—potentially beneficial for tightening art-making cycles—but it often does so too quickly or in an irritating manner.

T3 Students appreciate avoiding clutter. Like all programmers, novice creative coders are sensitive to inherent tradeoffs between minimal and feature-rich coding environments.

T4 Useful features may be “too useful.” Students were receptive to integrating art-specific and other sophisticated tools into their programming environment. Yet such features can inspire skepticism—even by novices—about their effect on learning.

Next, we situate our study within related work (Sec. 2), and then we describe our creative coding course (Sec. 3). After describing our methodology (Sec. 4), we analyze the results and consider our primary themes (Sec. 5). Through these studies we identify design implications for subsequent creativity- and novice-focused IDEs.

2 RELATED WORK

This paper investigates how to integrate advanced editor techniques into tools focused on novices and creative purposes based on observation and analysis of novice programmers. Given the broad range of related works, we frame the discussion around our primary design decisions: to start with the p5 editor and its existing feature set (Sec. 2.1), to add a suite of more advanced features (Sec. 2.2), and to evaluate these adaptations in a classroom setting (Sec. 2.3).

¹ We did not collect user identifiers in the Year 2 log study. Thus, “39+” indicates that the log data includes students who did not complete the course.

2.1 Creative Coding Environments

Creative coding is a multifarious category of work encompassing diverse approaches and topics. One common element is the use of editing environments that have been customized to address the particular domain of consideration.

One prominent example is p5.js [1] which (like its predecessor Processing [87]) can be used as a standalone library, but is made substantially more approachable by novices through the availability of a simple development environment specific to doing work with that library. The p5 editor [3] simulates a simple web server in the browser by combining each of the files in a “sketch” (synonymous with program in this context) and executing them as a standalone web page in an isolated component. The existing feature set in this editor is a particularly intriguing object for study for several reasons. First, it is lightweight, web-based, and supports cloud-based saves and shares—a good fit for an introductory programming class, as it does not have potentially intimidating baggage of a heavyweight IDE. Second, the text editor (based on CodeMirror [45]) supports a number of contemporary IDE features—such as linting [57] and auto-formatting [15]—that make our findings potentially generalizable. Furthermore, it contains a live-reloading (“auto-refresh”) feature being actively researched in programming-language user-interface communities [88, 90]. By considering (versions of) the existing, relatively standard p5 editor, our formative study aimed to understand which features were important before pursuing more drastic changes within the scope of this work and beyond.

In addition to these more general editors, there are a variety of tools that focus on more limited domains. For instance, ShaderToy [84] provides a browser-based editor for creating and sharing shaders and prominently features procedural and generative visual art. Like creative coding in general, these editing environments are not limited to the graphical domain. Tweakable [5] and Orca [55] provide environments for creating programmatically generated music, based on node-and-wire composition and 2D livecoding, respectively. HappyBrackets [34] more closely reflects our approach to enhancing creativity by augmenting a standard IDE, although it is centered on using IoT devices for musical composition. MakeCode [11] has an editing environment that contains synchronized block and text representations of code with a focus on creating games for microcontroller-based devices. Most similar to our work, p5.fab [95] modifies the p5 editor to support digital fabrication.

Mitchell and Bown [77] studied the needs of creative coders through a lab-based study, highlighting the value of visualizing program state, supporting best practices and short iteration cycles, and assisting exploration. Our findings are closely related to theirs, but located within a classroom and conducted on a longer time-scale—following Frich et al.’s [35] call for more studies to evaluate extant tools in their *in-vivo* usage context. This scale and scope informs our different, but complementary, set of themes.

Creative coding IDEs, and other such tools, target users at an intriguing intersection: many are relatively inexperienced but are strongly motivated to use these systems effectively. Lessons learned from studying users of these systems (e.g. students in a creative coding classroom) may translate to other venues with non-technical high-engagement users. Such populations occur widely and include spreadsheet users [14], tinkerers [17, 21], and artists more generally.

2.2 Advanced Editor Features

Many works have experimented with new ways to augment conventional text-based code editors with more interactive capabilities. Among the plethora of such features, we chose several to consider in this work. In the first year of our study, students had access to a *live-reloading* feature in the existing p5 editor. In the second year, we implemented domain-specific *graphical widgets* (namely, color picker and number sliders), and *bidirectional shape drawing*—among many other advanced features being actively researched—because they are closely aligned with the concerns of creative coding, were well-received in the first-year survey, and were feasible to implement given finite resources. We discuss these features below.

Tanimoto [96, 97] describes programming affordances on a *liveness* spectrum, relating to the degree of agency that users express in the execution of code. These range from the familiar edit-run cycle to predictive execution, with the always-executing style of live-reloading in the p5 editor falling in the middle. Immediate feedback evidently has rich educational utility, as in Python Tutor [39], and the sprawling number of systems its design informs [38]. Omnicode [58] takes a “Display all the values” approach to help novices understand and debug code. They find that the always-on strategy is useful for these purposes, which agrees with Kramer et al.’s [63] findings that live programming helps users fix bugs more quickly than a traditional edit-run cycle. Huang et al. [52] also found that live programming helps students perform some tasks more quickly, but in their study learning outcomes remained unchanged.

Augmenting text with *graphical* representations can provide a more natural way to specify code than textual input. The complexities of these vary from simple inline widgets, such as sliders and color pickers, to more complex designs. Graphite [81] explores a notion of palettes which allow for domain-specific editors, such as for color and regular expressions, surfaced through autocompletestyle menus. Barista [61] integrates interactive structured visual representations inline with code. Andersen et al. [9] build on this premise by formalizing how GUIs might be integrated directly into Racket code. Several features we added to the p5 editor follow the hybrid textual-plus-visual approach found in these works, targeting our specific domain and audience, novice creative coding.

Some systems *bidirectionally* synchronize code and GUI manipulations: changes made to either the source text or corresponding graphical output are reflected in the other [4, 47, 48]. While this has been most prominently used to create parametric drawings [44, 49], both ours and previous works suggest potential value for novices as well. For instance, Hundhausen et al. [53] find that this type of bidirectional development has educational utility and promotes skill transfer to text-based languages and environments. Contrastingly, Do et al. [29] utilize a mixed text-and-direct manipulation approach to teach an Hour of Code course to 5th and 6th graders using a JavaScript-like language. Yet, they did not find as rich an educational benefit, but argue that further development is necessary to situate this UI paradigm in creative-educational contexts. Our results tentatively suggest that this approach can be useful—in terms of student usage and perception. However, further study is needed to understand the effect it has on learning.

2.3 Classroom Studies

Computer science education researchers have studied the potential benefits—regarding gender diversity, retention, and learning outcomes—of emphasizing computing with media in introductory programming courses [41, 42, 94]. Our work provides a step toward understanding the role that programming *tools*—as opposed to curricular design—might play in creative-educational settings.

Despite this work taking place in a classroom, however, our aims in this paper are not focused on measuring the pedagogical impact of individual editor features. Instead, we pursue an understanding of the needs and perceptions of novice creative coders, and our classroom setting allows us to engage with such users on the time scale of an introductory programming course. As Weintrop and Wilensky [103] argue, “it is critical that we conduct studies ... analyzing tools not from the perspective of those who have already mastered the content, but instead from the perspective of the learners who the tools is designed for.” Our work embraces this approach, deriving guidelines for IDE design based on perceived utility of different features “to better inform educators on how to best utilize them in their classrooms ... [and] provide a roadmap for the improvement of these tools moving forward” [103].

While we have specifically opted for a text-based environment, block-based environments are notable for their frequent use with younger learners. In a study with high school students in a formal classroom setting, Weintrop and Wilensky [103] found that students (i) considered it easier to read programs as blocks rather than as text, (ii) liked the visual cues offered by blocks (though this preference diminished over time), (iii) found blocks easier to compose (via drag-and-drop), and (iv) liked how the interface organized blocks into related functionality and helped serve as memory aid. Their study also identified several perceived limitations of blocks: that they are potentially less powerful, slower and more verbose, and inauthentic—in the sense of not “doing the same kinds of things they will do in ‘real life’ outside of the environment in which learning takes place” [91]. Notably, even though novice high school students appreciate the pedagogic value of blocks, they still perceive them as inauthentic. This comports with our findings about skepticism about unfamiliar or advanced features among novices.

A variety of works have employed similar logging studies to ours (often referred to as learning analytics [56]). Some such works use in-course logging studies as a means by which to analyze student progression through assignments, which are described in multiple surveys [54, 56]. Helminen et al. [46] used a similar environment as our own to understand the types of errors students encountered in an introductory Python course. Vihavainen et al. [101] conduct a key-level logging study of novice coding behavior, although they seek to understand student behavior rather than IDE design for novices. Our work is related to these, but we are less interested in understanding issues like student progress through assignments than the hindrances they encounter in the UI generally.

3 COURSE DESCRIPTION

Our course aimed to teach basic computing skills (e.g. variables, iteration, and function decomposition) to students with little-to-no programming experience in the context of creative coding. Learning to program typically also requires learning many surrounding

skills, such as facility with command-line interfaces. To eliminate such possibly intimidating setup difficulties—and allow tighter integration between our web-based instructional texts and the venue where work was to be done—we decided to centralize all student work within the online p5 editor.

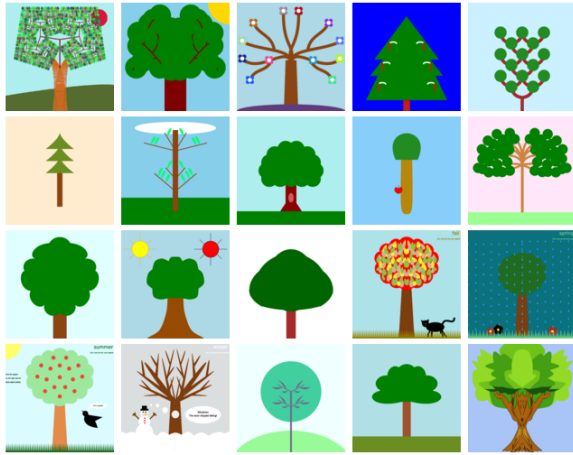
Following common practices in creative coding courses [66] and tutorials (such as from Khan Academy [6] and Happy Coding [106]), we used JavaScript and the p5.js library as the primary learning mediums, although the basics of web programming with HTML and CSS were also introduced. p5 exposes a variety of drawing and interaction methods as primitive functions (such as `rect` and `circle`) which the programmer combines either to make static or dynamic compositions. While p5 can be used to fully manipulate the native DOM, the majority of coding occurs inside a simplified environment focused on HTML-canvas manipulation.

We taught four editions of the course, referred to chronologically as *sp21*, *su21*, *wi22*, and *su22* (summarized in Fig. 3). The *sp21* and *wi22* editions were “full” 10-week college courses, offered from within a computer science department but cross-listed with media arts. Students had broad academic interests: more than 20 different degree programs were represented by the 58 students (see the appendix for a breakdown). The *su21* and *su22* editions were intensive 3-week versions taught over the summer to high school students. A fifth version of the course was taught during the summer of 2021, but was dropped from our analysis because participation was too small to meaningfully analyze. Required coursework consisted of graded individual homeworks, collected but ungraded exercises, and, in the full editions, an individual self-designed project. Taking into account differences in assignments and course material, *su21* and *su22* were roughly two-thirds of *sp21* or *wi22*. Lectures in *sp21* and *su21* were delivered remotely over Zoom. The first three weeks of *wi22* were also taught remotely, with the remaining weeks conducted in a hybrid format (during which students more often joined via Zoom than in person). The *su22* edition was taught entirely in person and—with more in-class time (Fig. 3)—included more required group work on practice exercises than other editions.

While the course was designed for those with limited experience, we observed high levels of self-reported prior experience in each edition. These varying levels color some of our observations. Based on our experience teaching them, the high school students in *su21* may have been over-confident in their description of their prior experience. However, those in *su22* did seem to have non-trivial

Edition	Course Details			Student Details	
	Sessions <small>live coding lectures</small>	Session Length	Session Period	Students who finished the course	Self-reported Experience
<i>sp21</i>	23	50 min	10 weeks	31 <small>including an auditor</small>	64.5%
<i>su21</i>	13	150 including 1-2 short breaks	3	26	84.6%
<i>wi22</i>	23	50	10	27	48.1%
<i>su22</i>	13	270 not including lunch break	3	12	100%
<i>total</i>				96	69.8%

Figure 3: Course details by edition. Experience was found by a pre-course survey that asked “How much programming experience do you have?” We coded answers into no experience, some (having taken less than a college-level course), or high otherwise. We merge the latter two levels here.



Submissions from *wi22* students who opted-in for public release.

Figure 4: One assignment in each course involved designing a tree, which exhibits horizontal axial symmetry.

prior experience, perhaps due to a selection bias caused by the course being offered in-person at our university. Despite higher self-reported prior experience than we expected, in our experience teaching we found that this experience did not necessarily lead to overwhelming mastery of the basic introductory material covered in the course. Therefore, we believe it is fair to view our students, as a group, to be novices.

The progression of assignments was designed to employ fundamental programming concepts (e.g. variables, function abstraction and decomposition, loops, arrays, and objects) for various media computation tasks (e.g. vector graphics drawing, animation, image manipulation, and basic web development). Aiming to serve the twin goals of teaching programming and fostering its use for creative expression, most assignments were open-ended (as opposed to being prescribed with easily-testable specifications). For example, one early assignment asked students to make judicious use of variables and arithmetic expressions to implement a symmetric tree drawing of their own design. Fig. 4 shows a sample of student submissions from *wi22* for this tree assignment. Additional assignments are described in the appendix, and the full course materials are available online at cs111.org.

4 METHODS

We now describe the studies that ran alongside each offering of our creative coding course and provide summary statistics. See appendix for survey instruments, ethics statement, and other materials.

4.1 Year 1: Editions *sp21* and *su21* with *p5/y1*

In the first year of our two-year formative study, we deployed the *p5* editor mostly as is. We added a couple features to support course logistics, but we did not add any new programming affordances.

4.1.1 Custom Features in *p5/y1*. Our initial fork added two features in support of teaching the class online. The first enabled students to submit assignments from within the editor to GitHub repositories as pull requests. Course staff then provided feedback

and grading on these pull requests, merging them once complete. The second mechanism allowed students to click any code example in the online course materials to open the code directly in the editor (without intervening copy-pastes or file-saves). We removed features which were either not relevant to the class or would have negatively affected the course design (such as project sharing).

4.1.2 (Per-User) Log Study. We ran a study in the first two course offerings (*sp21* and *su21*) to collect information about the coding behavior of students who opted-in to participate. For these students, we captured the state of each sketch on every execution, save, submission, and structure edit (e.g. find and replace) throughout the course. Logs were sent to a cloud-based server which only recorded events generated by study participants. This ensured that all students experienced the same level of network traffic regardless of study involvement and thus did not penalize participants.

Student consent (and parental consent for students under 18) was sought prior to the course as part of a pre-course on-boarding process, which was also used to gather GitHub identification for submitting assignments and gauge prior experience levels (Fig. 3). Students were not compensated for their involvement in the log study as participation did not modify the course experience. Students were able to retract consent at any time during the course. Logs were not analyzed during the course. Although relatively coarse-grained, the logged events capture overall trends and patterns in the use of basic editor features. In contrast, a key-level log study (as in Vihavainen et al.'s [101] study of novices' first weeks with an IDE) might have enabled more detailed observations at increased cost, both in terms of data collection and analysis, without clearly supporting our research questions about feature usage.

During this study we collected ~0.5 million logged actions spread across ~5500 sessions, which we define as periods of interactivity with ≤ 15 minutes between any two actions. On average sessions lasted $\mu=23.3$ minutes with a standard deviation of $\sigma=38.9$ minutes (listed as $\mu \pm \sigma$ hereafter). Our analysis of error frequency did not shed light on our research questions, but we provide summary statistics about observed run-time errors in the appendix.

4.1.3 (Long-Format) Feature Survey. After both *sp21* and *su21*, students were invited to take part in an online survey soliciting their experiences using *p5/y1* and opinions about various features. They were asked about a series of features (Fig. 6), each presented as a static image with a paragraph of descriptive text. The feature progression was bookended by free-text questions on more general topics, such as debugging and code organization.

A total of 25 *sp21* and *su21* students participated in the survey. Participation in the log study was not a prerequisite. Our survey tool did not report the working time, but based on piloting, we believe that the survey took 20-40 minutes to complete. Participants were paid \$30 for completing the survey. Demographic data was not collected beyond a self-reported experience level.

For each feature, the survey asked both free-text questions and Likert-item style rating questions (how "Useful" is the feature, and how "Often" would they use it). The number of surveyed features (17) was rather high, and we did not randomize their order. So, to help calibrate ratings, at the end of the progression a table summarizing the features asked for additional Likert-item style numerical ratings (how "Interested" they were in each feature). We found

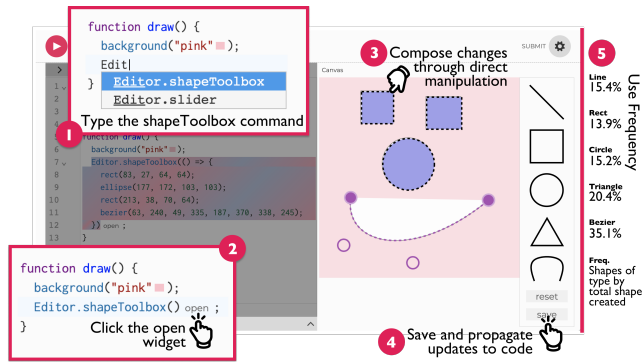


Figure 5: The Shape Toolbox is used to create simple compositions of shape-drawing code through simple GUI actions. (1)-(4): Interaction workflow. (5): Frequency of shape usage.

slightly negative correlations (Spearman's r) with presentation order and rating: Useful: $r=-0.117$, Interested: $r=-0.118$, and Often: $r=-0.133$ with $p \leq 0.015$. These metrics exhibited good agreement: $r=0.817$ (Useful/Often), $r=0.635$ (Useful/Interested), $r=0.655$ (Often/Interested) with $p < 0.001$. Thus calibrated, we focus only on perceived Usefulness. This selection is further informed by the technology acceptance model [27], which suggests that users' perceived usefulness is indicative of subsequent usage, and is thus a more helpful quality than estimated Interest or frequency of use. We found that only in-context docs was statistically significantly ($r=0.308$, $p < 0.01$) correlated with self-reported experience. This slightly negative correlation is also reflected in the qualitative comments about that feature (see Fig. 9 and Sec. 5.3).

We included a mixture of features for general computation (such as "Interactive Value Inspector" and "Linked Copy-and-Paste") as well as creative coding (such as "Coding by Drawing Tools" and "Canvas Ruler") that would be understandable based on their experience in the course. The complete list of surveyed features is shown in Fig. 6 and described in the appendix. Other features, such as notebook-style programming or multi-canvas editors (such as Stamper [20]) were considered, but not included—we believed that textual descriptions or static renderings would be unlikely to give effective motivation for their utility, and students may incorrectly forecast their experience of such unfamiliar features. A key limitation of our survey is the simple static presentation of features. Participant perceptions might have differed if they had watched video demonstrations or been able to experiment with the features.

4.2 Year 2: Editions *wi22* and *su22* with *p5/y2*

After gleaning student predilections in our formative Year 1 studies, we modified our editor to investigate these stated preferences. We used *p5/y2* in *wi22* and *su22*, during which we ran two more studies. Whereas we customized *p5/y1* to improve course logistics, our changes in *p5/y2* were motivated by the first year results.

4.2.1 Custom Features in *p5/y2*. We implemented the top-3 unimplemented features from the survey (see Fig. 6 or Fig. 13 in the appendix): Color Pickers, Autocomplete, and Shape Toolbox (which

was a synthesis of Coding by Drawing Tools and Directly Manipulate Shapes). Given limited resources, we forwent the Canvas Ruler (the next most-preferred feature) because there is a simple workaround for identifying positions (e.g. console logging the `mouseX` and `mouseY` on mouse movement), although we intend to address it in future editions. Several highly-rated features (e.g. Time Travel Slider, *p5* State Displays, and Interactive Value Inspector) were more speculative and thus deemed beyond the scope of this work. We made two further modifications based on observations, namely, adjusting Auto-refresh and adding Number Sliders (which are common in interactive documents [99]).

Two of these new features are accessed by calling special functions. `Editor.slider(min, max, value)` renders a Number Slider (—) for value in the range from min to max (see Fig. 1), with an optional fourth step argument to override the default continuous-dragging behavior. One alternative design is to store the metadata (range and step) in special comments, for example, as in Juxtapose [43]. Such an approach warrants comparison to our chosen design in future work. However, we elected to use the function call API to mimic a common *p5* function for creating dynamic sliders, which students already learned (`createSlider` [2]).

The most novel feature introduced in *p5/y2*, the Shape Toolbox, is also accessed through a function call-based workflow. The user calls `Editor.shapeToolbox()` and clicks a button to open the shape-drawing GUI. The text area is then disabled and a simple drawing toolbox is overlaid atop the output window. Using the Toolbox, users create compositions in the output pane by adding, translating, rotating, and scaling shapes (including primitive shapes and Bezier segments) with direct manipulation. Once satisfied, users click "save" to update the code—shape commands are called in the body of a function passed to `Editor.shapeToolbox`. Fig. 5 depicts this workflow. Translation back to the code is achieved through a simple template matching method allowed by a one-to-one mapping between drawn elements and lines of code. We decided against always displaying the shape-drawing GUI for two reasons. First, not all calls to shape drawing functions can have GUIs—this is the subject of research on bidirectional editing. At one of the end the spectrum is a very simple approach that maintains a top-level "scratchpad" function (where all new shape-drawing calls would be added), and at the other end are heavyweight and expressive techniques in prior work [44, 47, 49]—the former would be more restrictive than our chosen approach, and the latter beyond the scope of this work. Second, shape-drawing is only one aspect of our creative coding tasks; our approach displays the GUI only when the user explicitly opts to use it.

4.2.2 (Aggregate-Use) Log Study. We collected anonymized usage of *p5/y2* features during *wi22* and *su22*. Logs were collected through a customized version of Umami [22], a self-hosted privacy-minded tracker. We elected not to capture full-sketch snapshots in this study because our planned analyses for the second year did not require them. We believed lighter weight instrumentation would allow us to capture more fine-grained usage patterns, such as the length of sessions. Finally, this reconfiguration to full anonymity gave us leeway to collect data on all course participants, rather than just those who opted-in.

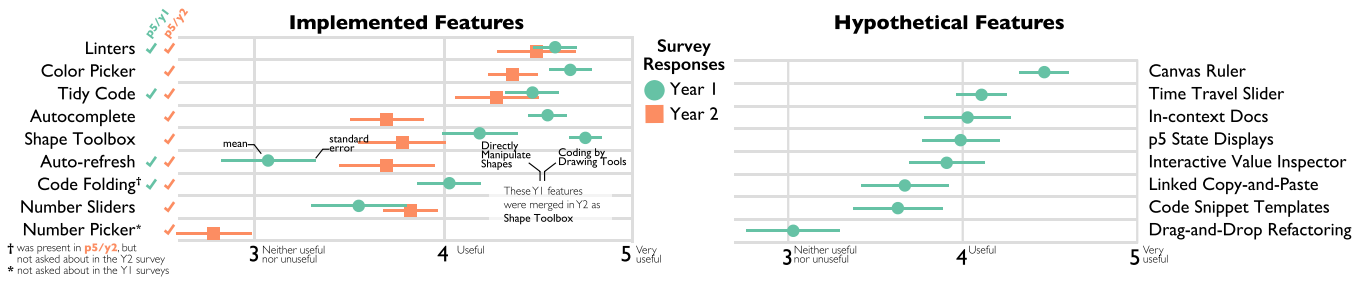


Figure 6: The features surveyed across both years and how “Useful” they were deemed to be on a 5-point Likert scale.

Through this process we collected ~1.2 million events across ~6730 sessions (defined as before). Due to a configuration error (present only in *wi22*), events were not collected with unique session identifiers, although we were able to reconstruct 75.4% of the sessions—the remainder are excluded from analyses requiring specific session information. While the incomplete data is unfortunate, it still provides a more detailed picture of activity than in Year 1, which saw 68% log study participation across *sp21* and *su21*. Within this reduced sample, sessions lasted $\mu=24.1 \pm 38.0$ minutes.

4.2.3 (Short-Format) Feature Survey. Near the end of *wi22*, students were invited to take an abbreviated version of the Year 1 survey (Sec. 4.1.3), containing only features that were added or improved upon in *p5/y2*. Following the structure of the previous survey, we asked about frequency of use and Usefulness for features one at a time, followed by a summary table asking about Interest and a suite of reflection questions. Participants were compensated with extra credit roughly equivalent to 1% of the final course grade.

A total of 23 students participated in the survey. Our survey provider did not measure time taken to respond, but based on piloting we believe that the survey took 10–15 minutes to complete. Presentation order was not correlated with any of our metrics ($p=0.779$ – 0.939). Again, the ratings exhibited reasonable agreement: $r=0.727$ (Useful/Often), $r=0.607$ (Useful/Interested), $r=0.693$ (Often/Interested) with $p<0.001$. As with Year 1, we focus only on Usefulness in the body of the text (see the appendix for the others). We found prior experience to be statistically significantly ($p<0.01$) correlated with only a single feature, auto-refresh, for which there was a somewhat negative correlation ($r=-0.487$).

5 ANALYSIS

We now reflect on the features, connecting them to the themes summarized in Sec. 1, denoted T1 through T4. We consider features implemented in both *p5/y1* and *p5/y2* (Sec. 5.1), followed by those added in *p5/y2* (Sec. 5.2), and then introduce concerns that cut across multiple features (Sec. 5.3 and Sec. 5.4). Our analysis draws on data from the survey studies (summarized in Fig. 6) and the log studies as appropriate. Participants from the *sp21*, *su21*, *wi22*, and *su22* surveys are referred to as A1–16, B1–9, C1–19, and D1–4, respectively, and are colored by year.

5.1 Features in Both *p5/y1* and *p5/y2*

We begin by considering features present in both editor versions.

5.1.1 Linting. This static analysis tool eagerly executes after small code edits, checking simple syntactic assertions akin to spell check for code. It was well received in both years and was mostly seen as helpful, although sometimes impolite.

Students found linting to be “very helpful” (A5,7,17,20, C10,16, D1, 2) and “very useful” (A19, C2,4,6,12,18, D4), because it “saves time and energy” (A4) and shows “where I needed to go to fix simple bugs” (A16). C10 believed that debugging “would be way more annoying without it” because “it’s not always obvious what you did wrong” (D4). Whereas 86.0% of executions in Year 1 passed lint, in Year 2 (where we had visibility into all lint runs) code passed 13.5% of lint runs (which happened after most small text edits). This may indicate that students address lint errors before running code as a simple integrity check, or that the analyses are executed too early; however, student comments seem to indicate the former. Unlike other features, students were incentivized to attend to it, as the absence of lint errors was a small part of homework grades (98.7% and 97.2% of submissions in Years 1 and 2, respectively, passed lint).

Beyond code style, linting can provide opportunities to expose novices to other best practices. For example, CSSLint [26] (used in *p5/y2*) explained that the `*` selector is considered bad practice because it is inefficient. Indeed, C3 felt that linting “trained me to think and type in a certain way”, and A5 observed that it could be “a nice way to point out when I am making stylistic errors (instead of [Tidy Code] just magically fixing all of them for me).” Utilizing this well-received channel for introducing programming features and practices is an opportunity for future IDE design. T1

Participants also offered ideas to improve the feature. Because the editor eagerly ran the linter, “the yellow line warning[s] often exist all the time. It annoys me” (B4). Instead, some students would have preferred not to see lint errors “until I finish typing” (A13) or “before finishing a line of code” (B5)—the mechanics of exactly when and how to display errors for incomplete code will require careful design (as considered, e.g. in Hazel [79, 80]). Others expressed a desire for more nuance—“acknowledging the difference between ‘This Must Be Changed To Have Nice Code™’ and ‘hey, maybe consider changing this thing!’” (A5)—and control—being able to “ignore/exit out of a warning” (A3). Poorly-received default choices and persistent errors can repel users. As an extreme example, one *wi22* student decided to use Replit [89], rather than *p5/y2*, for their final project because too many (CSSLint) errors seemed irrelevant or unclear how to fix. Linters integrated with editors in this way do not offer mechanisms to override general advice or to indicate that the user knows what they are doing. This is impolite computing [104]: it

forgoes user agency and generally is perceived as a pest. Avoiding these pitfalls is important to leverage the instructive opportunities offered by the well-received, static analysis-informed tools. **T1**

5.1.2 Tidy Code. Auto-formatters provide on-demand code restyling without semantic modification, and are common in professional coding workflows [86]. We often encouraged the use of this tool—called Tidy Code in the p5 editor—in lectures, but we did not incentivize its usage in grading. It was invoked manually (from the top menu bar or keyboard shortcut) rather than being executed on every save.

Like linting, this feature was generally well received. Students found auto-formatting to be “*super useful*” (**A15**) and “*very satisfying*” (**A2**). The formatting choices were not always appreciated, however. Whereas **C15** “*only rarely preferred my own organization*”, **A12** felt the results “*appeared less organized, such as having irregular line breaks*” and **A10** “*worried it would mess up my organization*.” We observed that students in Year 1 often (needlessly) invoked auto-formatting twice in a row. In particular there was a probability of 16.15% and 8.65% (in *sp21* and *su21*) of auto-formatted code being auto-formatted again right away—with similar behavior observed for saves (see appendix for details). This suggests that providing clear code-state signals (analogous to linting’s visual indicators) may reduce needless anxiety-motivated saves and tidying. The presence of this behavior in Year 1 suggests it was likely repeated in Year 2; however, the aforementioned configuration error prevented us from collecting auto-formatting usage. While simple indicators may seem to be trivial UI modifications, we suggest that it will impact the perception and understanding of such features.

Several students would have liked the feature to be customizable, rather than enforcing a fixed set of “*preferences that should not be forced by tidy code*” (**C16**). Indeed, some students would have liked auto-formatting better “*if it was a little configurable*” (**A16**); for example, “*if there [were] multiple common/standard rulesets there could be a way to choose which you want to follow*” (**A5**). Furthermore, it “*would be helpful to be able to specify which block of code to tidy*” (**A13**). Thus, extending well-chosen defaults with ways to selectively customize style preferences—a notion which has been referred to as “*code style sheets*” [69]—could further increase the politeness of this feature and thus its utility. **T1**

Like the teachable moments offered by linting, **B8** felt they “*Learnt a lot about code organization using this feature!*” As implemented, however, the results of auto-formatting are updated in the code box without explanation. Better would be for the editor to “*show you what you are doing ‘incorrectly’*” (**C19**), for example, using visual highlights and annotations to explain the differences—which could also serve as scaffolding to introduce version control tools.

5.1.3 Auto-refresh. This feature re-executes code upon text edits—a workflow demonstrating “*level-3 liveness*” [96, 97]. Auto-refresh was present in both **p5/y1** (inherited from the original editor) and in **p5/y2** (where it was modified). In principle, live feedback would seem particularly helpful in a creative coding context as programs are often updated with small graphical adjustments, and thus well matched with a short edit-run cycle. It was also well matched with our setting: the Normalized Programming State model [23] suggests that spending longer periods of time in syntactically unknown states (such as when the code has not been executed in a while) is

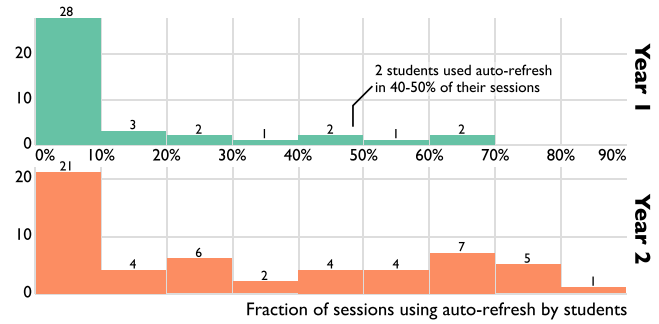


Figure 7: Histograms of the fraction of sessions where a student used auto-refresh any amount. 58.7% and 5.1% of students never used auto-refresh in Years 1 and 2 respectively.

negatively correlated with program success. This needs to be balanced with the cognitive load [54] caused by repeated executions.

Auto-refresh in **p5/y1** did not achieve a fruitful balance. As indicated in Fig. 7, only a handful of participants regularly used it and most students used it rarely, if at all. The survey responses color this imbalance. Whereas **A9** “*used this all the time and loved it*”, finding it “*way easier than clicking the ‘play’ button all the time*”, others felt that the keyboard hotkey was sufficient (**A16**, **B1,8**, **D1**).

More important than convenience were differing views on the fundamental interaction model itself. **B7** appreciated the ability “*to see what I was creating as I coded*”, finding it useful even though “*error messages that kept popping up got in the way a little*”, while others found the errors “*very distracting*” (**A5,6**). Participants felt the feature was “*running incomplete code unintentionally*” (**B6**) and “*when you don’t want it to*” (**B1**). Instead, some students felt robbed of their agency over their code, desiring “*to be the boss of when my code reran*” (**A5**) and in “*control my own pace*” (**B2**), only running the code when “*I know I have something that I want to see*” (**A13**). This suggests that, while spending too much time in syntactically invalid states may be detrimental [23], spending too little time is also problematic. Developing a careful understanding of the tradeoffs is an important avenue for future live programming work. **T2**

For the purposes of this work, we made only simple changes to auto-refresh in **p5/y2** based on our observations from the first year. We increased the refresh delay from 400ms to 1s, and, more importantly, in the event that executed code had lint errors—a proxy for run-time errors—the editor did *not* refresh the canvas, instead indicating that it was “*stale*.” Thus, in the (many) cases when edits are incomplete or erroneous, the canvas remains visually stable.

The modified auto-refresh was modestly better received, with its Usefulness increasing from $\mu=3.1$ to $\mu=3.7$. In addition, per Fig. 7, it was used more often—although we note that auto-refresh was demonstrated more at the beginning of *wi22* and *su22* than in prior editions. A one-sided t-test indicates that students in Year 2 used auto-refresh significantly ($p<0.001$) more often. Yet, the overall balance remained far from perfect. Some participants were “*stressed*” at “*all the errors that pop up as I implement new things*” (**C15**) and “*before I got to fix them*” (**C4**). These negative views seemed more likely to come from those with prior experience ($r=-0.487$, $p<0.001$), which may suggest that expectations are set by experience with

tools exhibiting a different execution cadence. Others, however, found it “*very useful for certain exercises that needed lots of small adjustments*” (C3) and “*very helpful when using trial and error*” (C16). Overall, we observed no significant changes in user behavior after modifying auto-refresh, despite the improved perception of the feature. This again underscores that designing UIs to be polite (or at least not irritating) is critical to their usage.

5.2 Features Only in p5/y2

Next, we consider the features that were added in p5/y2. While Year 2 survey responses are based on hands-on experience with the features in p5/y2, Year 1 responses are based on descriptions in the survey and experience with other tools. Feature use in wi22 is shown in Fig. 8.

5.2.1 Shape Toolbox. The most significant addition to p5/y2 was the Shape Toolbox feature that allowed GUI-based specification of primitive shapes using direct manipulation which generated matching code (Fig. 5). The constituent parts of this feature were highly perceived in Year 1: $\mu=4.8\pm0.44$ for Coding by Drawing Tools, and $\mu=4.2\pm1.0$ for Directly Manipulate Shapes. Some students believed it would be “*very beginner friendly*” (A3) and would make work “*a lot easier and faster*” (B7). Others believed it would also reduce errors (A9) and help with debugging (B6).

Help programming curved shapes—such as the trees in Fig. 4—was particularly enticing: “*for bezier curves, changing the input values rarely produced an expected result*” (A12), highlighting a gulf of execution [78]. The process usually involved “*lots of trial and error*” (A3), sometimes resulting in student disengagement: “*Coding the bezier curves manually turned me off of them, and I did not attempt them in my work*” (A14). However, that same student noted “*If I had had a tool like this, I certainly would have used them.*”

Several students in Year 2 embraced the feature. For example, C18 found it “*EXTREMELY helpful, especially when it came to drawing Bezier curves. Every time I had to draw a curve, I used the shape toolbox. I probably would have cried without it.*” C10 mentioned that it “*Was very nice to use it to get approximate coordinates then fine tune them after.*”

Although the feature was “*very useful for beginner projects*” (C2), several students, including C6, “*used them less as time progressed.*” Shape Toolbox was used often for the tree homework (see HW3 in Fig. 8), and use per execution by week was minimal after that assignment, being used in only 2.12% of all (available) sessions. Perhaps because the feature did not have a stable visual presence (as with the auto-refresh button), some students “*completely forgot this existed, but I think it would have been really really useful if I had remembered*” (C4). In addition, although we expected the feature to be used extensively for HW2, in wi22 Editor, shapeToolbox was announced but not demonstrated in class until after the assignment was released. Bezier curves accounted for the majority of invocations (see Fig. 5.5). Toolbox sessions (from open to save) lasted $\mu=22\pm30$ seconds, indicating that it may have been used relatively often to make small graphical adjustments, as opposed to building larger compositions.

Students may have continued to use them later in the course “*if it allowed for some of the shapes that are more complicated*” (C16). Further limiting the utility of the feature, within an invocation shape

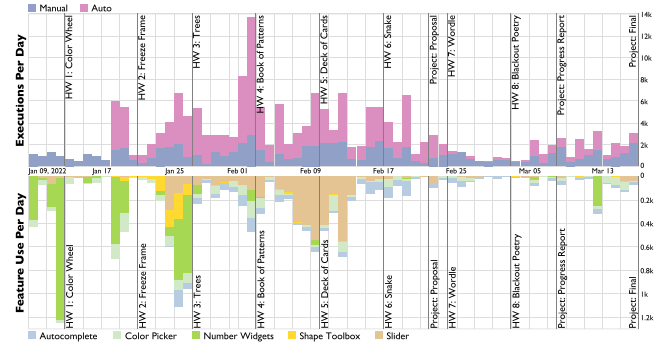


Figure 8: Feature use in wi22 was guided by course content. For instance, autocomplete was demonstrated prior to HW3 and sliders were included in the starter code for HW5.

drawing functions allowed only literals—once a student wanted to use variables and arithmetic expressions, the Toolbox would no longer open. “[C]reating an object without this feature would be better because of the precision” (B5) afforded by variables, expressions, and so on. Thus, the feature ultimately fell short of what students imagined: $\mu=3.8\pm1.2$.

Bidirectional updates are being explored in a growing number of systems (as in Sketch-n-Sketch [49]), but there remain significant technical and UI design challenges to explore, before even considering their value to novices. As predicted by a couple students, more feature-rich bidirectional synchronization would need to reconcile ambiguous graphical interactions (“*There are many parameters and it would be hard to make it so it manipulates them one at a time*” (B5)) and their effect on other parts of the program (“*My only but major concern would be that it doesn’t confuse the other lines of code, and that it may not run the way the programmer wants to use it*” (B6)).

Nevertheless, the experience suggests that even a simple implementation of this very desirable feature was promising. However, as we discuss in Sec. 5.3, many students were skeptical about the effect of this feature on learning. T4

5.2.2 Autocomplete. We enabled a simple autocomplete menu [45] and populated it with p5-specific identifiers (variables and function names), syntax templates (common patterns, like for-loops with holes), and commands for invoking the Shape Toolbox and Number Sliders. Passively supporting learning in this way would seem to be a natural fit for our setting, but some students were leery of it.

Year 1 survey respondents anticipated autocomplete positively ($\mu=4.6\pm0.5$), believing it would help in several ways. For example, to “*increase speed and productivity when coding*” (B9) and “*make it faster to get debugging done*” (B6). In addition, A13 believed autocomplete would encourage better code style: “*not having dynamic autocomplete incentivizes me to write non-descriptive function names and variables for the sake of efficiency.*” Participants also believed autocomplete would help “*discover new features*” (B5), “*expose us to new things we didn’t know existed*” (B7), and provide “*an idea of what to write or what could be written*” (B4). These beliefs are in line with how professional programmers use autocomplete to debug and explore APIs [71].

However, the experienced reality of **p5/y2** fell short ($\mu=3.7\pm1.0$) of anticipation. Autocomplete was used in only 12% of sessions (with 33.6% selections being templates), although it was used progressively less as **wi22** and **su22** went on. While this relative infrequency of use may be related to the simple implementation (which did not include embedded documentation or other common guidance features) or the emphasis later in the course on web programming (the DOM was not thoroughly reflected in the autocomplete suggestions), this trend appears to agree with how Vihavainen et al. [101] observed novice usage of autocomplete. They note that 27.3% of novices initially used autocomplete to create a particular command (Java's system print), which decreased to 1.64% after a week of use.

This appears to suggest that autocomplete can serve as a vehicle for teaching: it is *"a useful guide until I was able to type certain things in by memory"* (C13). Some perceived the ability to *"stop memorizing certain code"* (A9) as a benefit, while others thought *"it's a give and take"* (C7) and might hinder *"programmers' knowledge about commands and their forms in the long run"* (B8). We return to this concern about the effect on learning in Sec. 5.3. Beyond these hesitations, it is unclear why more students did not engage with the feature, although some noted that it can be *"annoying when you already know what you want"* (C15)—which suggests that the clutter **T3** or cognitive noise **T2** may be a factor. Given this diversity of opinion, we suggest that configurability is important to designing such features politely, as some students (such as D4) wanted to be able to turn off autocomplete (to limit its disturbances).

5.2.3 Color Pickers. Integrating a color picker into an editor for creative coding was perceived as very useful in the Year 1 surveys ($\mu=4.7\pm0.56$). Whereas A4, probably like many students, did not pick colors as much in the second half of the course, A6 said *"I had a color picker tab open for every single assignment."* Based on this enthusiasm, we implemented a modal color picker dialog box in **p5/y2** (Fig. 1), following a sentiment from A13 that such a design *"would probably be more helpful than in the code to prevent clutter."* **T3** Note that a similar color picker was recently added to the p5 editor, highlighting the value of this feature. However, ours supports more color formats, namely, web color names—driven by a participant suggestion (B9)—and RGB values as numeric lists.

After experiencing color pickers, students viewed them positively ($\mu=4.4\pm0.65$), with the **wi22** students using them frequently in the first half of the course, as depicted in Fig. 8. They helped make the process *"more efficient"* (A9), *"[taking] out the hassle"* (C6) of *"open[ing] up another program"* (A2). Color pickers may foster creativity, as they could *"let me pick some irregular colors"* (B2).

Several participants also voiced support for the idea, suggested in the survey prompt, for an eyedropper tool. Others suggested additional features inspired by drawing programs like Illustrator, such as grid (D1), zoom (C15), *"better proportions"* (A3), or a way to *"group lines and shapes and move them all at once"* (B7). Such lightweight and familiar tools from creativity domains are natural enhancements—as long as they are not impolitely imposed **T1**—that we intend to investigate in the future.

5.2.4 Number Pickers and Sliders. "Scrubbers" [100], which allow direct manipulation of numeric values by dragging, are often touted in live programming systems and interactive documents [99] as being representative of the value of those environments. Despite

the overlap between live programming's close connection to the visual domain and the interests of creative coding, the clutter **T3** and lack of control **T2** brought on by these features impeded adoption.

Some Year 1 students were positive about hover-based Number Sliders, believing they would allow them to *"experiment with the code more quickly"* (B6) and *"more efficiently"* (B9). However, some worried that *"it could make the editor look more crowded"* (A3), while A5 noted *"I would rather just do it myself."*

Nevertheless, we added Number Sliders to **p5/y2**, which appear (per Sec. 4.2.1) inline via `Editor.slider(min, max, value)`, as well as Number Pickers (Fig. 1), which are buttons surrounding each number literal that allow it to be incremented and decremented (`-600+`). (Such small modifications explain the large absolute number of Number Picker events in Fig. 8.) Students found these additions could be a *"quick, helpful way to make sure my assignments didn't break at a larger scale"* (C6), as was the case for HW5 (cf. Fig. 8). Although scrubbers were perhaps most useful toward the latter stages of a task, *"when I'm playing around with my final result"* (C17), C13 felt they *"allowed me to tap into my creativity."*

Yet, per Fig. 6, the feature was not so highly rated. A recurring theme is that scrubbers—in various configurations—felt *"messy"* (B4, C9), *"disrupted the look of the code"* (C4) **T3** or were just generally unnecessary (B10). A14 felt that the transitory changes would be confusing, and hard to maintain a model of different parameter configurations. **T2** Others wanted more refinement in the numeric type, such as limiting it to numbers *"divisible by five"* (A5). While these features are typically well-used in graphical applications like Figma, it seems that this type of feature is *"trying to solve or better a process that needs no help"* (B10). While there is evident overlap between our domain with other artistic settings, not every translated feature will match the interests of learners.

5.3 On Skepticism

Next, we grapple with the perception that some tools take away learning opportunities that may be needed to *"become a good programmer."* **T4** Several features were perceived as making things too easy for novices. Fig. 9 summarizes how "skeptics" worried about different features. These perceptions are valuable: as the technology acceptance model [27] and related theories highlight, *perceived usefulness is a central part of whether a system is ultimately used.*

Several students worried how syntax templates (see appendix) and autocomplete balanced the tradeoff between augmenting their abilities and enfeebling their development of skills. Whereas A5 was *"not sure if it actually matters"* to practice memorizing names and function signatures, C17 weighed the tradeoff according to the goals of the student: *"I wouldn't consider it a horrible thing for those who don't want to go into coding professionally/too much"*—implying that a more serious programmer might indeed miss out on practicing an important skill. A16 thought such features *"might reduce some of the learning by doing that you get when coding, so I'm not sure if it's great for a class. I learn through my coding mistakes and this would reduce the number of mistakes, so a mixed bag."*

There were similar concerns about in-editor documentation (also discussed in the appendix). For example, A15 said that *"new coders need to learn the process of going into the manual."* A4 reconciled the aforementioned tradeoff as follows: *"However, depending on*

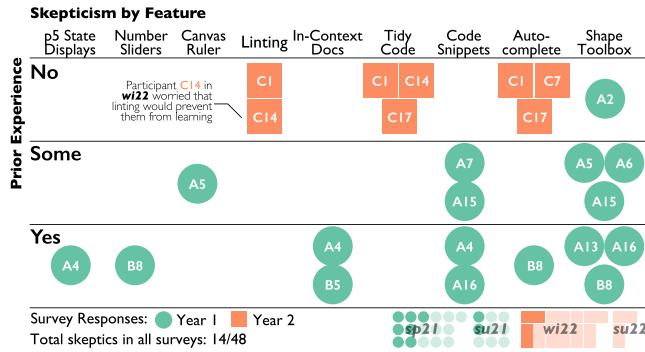


Figure 9: Some skeptical survey respondents worried that some features would deleteriously affect learning.

the specific goal of this course, if it is to focus more on the creative coding aspect and not necessarily ‘become a good programmer’ then in-context docs would be awessommeee.”

Some of these concerns might be ameliorated by introducing a notion of documentation or autocomplete levels (in a similar style as DrRacket’s language levels [73]), which gradually adjust what information is available as new concepts are introduced. However, without sufficient signaling, students might construct mental models of the information present in the feature and then dismiss all subsequent configurations.

Conflicting views over the Shape Toolbox in Year 1 were most striking. “This feature would be so useful and allow for more creative opportunities especially for beginner coders” (A10). It would also be a “useful learning tool” (A3) by allowing students to “see how the code changes in order to learn how certain parts of the code are working” (B9). But many students were skeptical. This “feels like cheating!” (A6) and “saves way too much work for the new learners” (A15). “It’s way too useful but can hinder with the learning process of basics of coding. As a student, I won’t want this but as a programmer who knows the basics, it’s a nice feature” (B8). “I think some of these features while helpful would have discouraged learning. Some of the most rewarding parts was sweating through inconvenient parts” (A2). As shown in Fig. 9, some of this hesitation was self-censorship by students with little or no prior experience.

However, among Year 2 participants (all of whom had access to the feature), there were no skeptics of the feature. Perhaps the idea of others having improved tools is jarring, while students who are given improved tools simply worry about the plenty of challenging learning left to do. We view this situation as akin to giving students calculators in a math class: they help with specific classes of tasks that, once simplified, enable learning about richer topics.

Students seem to construct a naive model of what makes a good programmer, suggested above as being someone who has memorized the entire language and does not depend on digital assistants or developer-experience tools, thereby dismissing behaviors besides this as being inauthentic. We suggest that reorganizing and reforming this model is part of the value that classroom-based computing education offers, as it can help to offer a thicker model of what is authentic [91]. Enhancements to novice-oriented IDEs such may also help to dispel these notions if they are *perceived* as realistic tools rather than as something akin to training wheels.

5.4 On Creativity

Finally, we consider the role of creativity in our editor. While there exists little agreement on what creativity means in HCI research [35], we found that students espoused two clear views on how tools might help them creatively: automating tasks that impede of creativity and helping explore unknown functionality.

For students, creativity often appeared to be something which typical coding tasks stood in the way of; obstacles that some technical interventions could ameliorate. Shape Toolbox was emblematic of this style of reduction. For instance, A10 believed that such a tool would “allow for more creative opportunities especially for beginner coders,” and B9 believed that it “could help with planning ideas for art projects and increase creativity.” As noted in Sec. 5.2.1 students in p5/y2 embraced this feature and appeared to use it to reduce the tedium required to precisely locate shapes, thereby making greater room for artistic expression. Others highlighted the value that tools that reduced tedious tasks, such as picking individual coordinates through a ruler (e.g. A3 and A9) or identifying which lines corresponded to which components of the image (B9). Summarizing this view, C16 observed that “I think what this editor did well as an art tool was streamlining certain common processes.”

Beyond reducing tedium were opportunities for exploration, which were manifested both as moments of play (A8) or fun (A4, C6), as well as discovering new functionality. For instance A12 believed that “comprehensive documentation would have allowed me to be both more creative,” a view which was confirmed by C17, who believed that such features help do “things I don’t yet know how to do by myself” and thereby “help me be more creative.” A9 believed that surfacing program state might encourage reflection and discovery, such as by seeing that “circle has a round stroke cap, so it might make me wonder what other shapes the stroke cap could have.” A14 believed that an assistant that made artistic suggestions might be well received, “[f]or instance, if I’m editing text, and I was given suggestions for font, color, etc.” Similarly, A4 noted that it would be useful receive suggestions to help inspire their designs, for example, through “videos on youtube, images, articles.” Some features, such as number sliders, were highlighted as being only valuable “when I’m playing around with my final result” (C17), but that they “encourage a lot of experimentation and creativity” (A4). These observations align with prior work, which highlighted the value of providing assistance in exploring the space of possible designs in creative coding contexts [77] and in creativity support tools generally [93].

Whether a student’s primary purpose was closer to coding or to making art was an additional source of skepticism to those in Sec. 5.3. Again, regarding Shape Toolbox: “This would be great, but would reduce the amount of time figuring out the code. This would make it more an explicit art tool, and less a ‘make art with code’ tool” (A16). A5 similarly noted that “feels a little too much like draw-ing for my taste” and took the class “with the primary goal of getting better at programming so I’d want to do things the code-y way”. Similarly, C17 felt that it “hinder[ed] the process of creative discovery— including trial and error”, however this was mostly not an issue as they sought to be “to be more accurate than creative” in this course. While it is natural to want tools to be familiar, we believe that new authoring paradigms (e.g. bidirectional programming) should be viewed as complementary rather than antagonistic.

6 DISCUSSION

This paper explored the observed behaviors and surveyed perceptions of novice programmers in a creative coding course. To wrap up, we recap the main themes, reflect on the connection between our work and other domains, describe limitations of our studies, and offer avenues for future work.

6.1 Recap: Themes

In our analysis, we chose four recurring themes to highlight.

T1 Static Analyses. We observed that simple static analyses were seen as supportive of a variety of types of work—notable given that error messages sometimes are obstacles in introductory settings [16]. Polite lightweight assistants that respect user agency, like those expressed through linting or auto-formatting, can be a helpful platform on which to learn and test new skills with confidence. On the other hand, A5 noted that they “*would also probably prefer to do things by hand*” rather than use advanced features because there lacked visual indicators of a particular action’s effect—highlighting the importance of clear effect-forecasting for feature trust.

T2 Liveness. We saw that overeager evaluation can overwhelm and stress users through distracting updates that are unsynchronized with their expected edit-run cadence. Live programming offers enticing benefits for novice and creative contexts (e.g. feedback immediacy or a closeness of mapping between code and graphics). Yet, these interaction challenges for non-expert settings are not yet thoroughly understood, leaving open questions about how to blend user control with system eagerness in a profitable way that maintains an experience level-attuned sense of agency.

T3 Clutter. We noted that amateurs are mindful of how the editing space can become overwhelming if too much visual noise or unfamiliar forms of interaction are introduced. For instance, students are aware that individual features (e.g. Number Scrubbers, lint errors, and autocomplete menus) can break their flow. We highlight the difficulty and importance of developing design guidelines that can aid the development of novel features within these constraints.

T4 Skepticism. Finally, we discussed how user perceptions of a feature can inspire skepticism about its propriety in learning environments. Year 1 students believed that the Shape Toolbox would impede learning; however, those who used it in Year 2 did not share that concern, instead viewing it as a convenience. Year 2 students also saw knowledge assistants such as autocomplete as detrimental to their development as programmers. We believe it is valuable future work to better understand what types of features and knowledge assistants are likely to be viewed as detrimental.

6.2 Connections to Other Domains

Next we reflect on how our findings may apply more broadly.

6.2.1 Programming Pedagogy. Our work is merely situated within a classroom; we do not seek to make claims about the learning effects of the features we studied—this is an important, separable direction for future work. Yet, some of our themes may carry over to pedagogically-minded editors in more general learning contexts.

We suggest that skepticism **T4** about features perceived as being too useful, such as autocomplete, may continue to be prevalent in learning contexts. Such concerns might be circumvented by emphasizing tools that help correct, rather than help complete, such as how linting **T1** can identify an error while also providing justification and explanation for that error. We also note the value of having a programming environment that is perceived as being approachable (A3,5, B1). Furthermore, tools having not “*got in the way*” (C16) or otherwise cluttering **T3** the display in unhelpful ways seem intuitively valuable, particularly in learning contexts. Similarly, live execution **T2** may be beneficial in non-visual contexts as it promotes immediate feedback, such as by rerunning a test suite dynamically, as in Jest’s watch mode [32] or Huang et al.’s use of projection boxes [52] in a classroom to expose live values.

Finally, like others before us [36, 70, 105], we found that a curriculum centered around media-art topics—as opposed to more abstract content often found in intro CS courses—invited a broad range of students who might not otherwise study CS in a formal setting (the appendix lists majors represented in the courses).

6.2.2 Other Domains. Editors specialized to a given domain can make adaptations that aid that context. In this work we focused on creative coding and designed affordances specific to this domain, however our findings might be applied in related contexts. We highlight the value of bidirectional editing, linting, and designing editors with their effects on creativity in mind.

Bidirectional synchronization of code and effect (such as in our Shape Toolbox) seems to be an especially valuable approach in domains that have a prominent visual component. This has been explored by Asai et al. [10] as a mechanism to clean and synthesize data for statistical modeling, as well by DeLine [28] and Wu et al. [107] for data science tasks such as modeling and analysis. We suggest such synchronization might be usefully applied to other visualization contexts (like preparing charts for presentation), as well as other creative coding contexts. Such interfaces may potentially reduce tedium in certain tasks and, more fundamentally, may provide opportunities for learning about the domain, for example, demonstrating how to achieve a particular effect using code.

Next, we highlight that linters (or other static analysis tools **T1**) can provide a straightforward channel for introducing newcomers to basic principles and best practices of a particular domain. While they have already been explored in some contexts—such as for spreadsheets [13] and visualizations [51, 74]—additional fields such as data science [75] and music editors might integrate these concepts as well in order to surface best practices, such as highlighting statistical fallacies, helping guide usage with unusual tools (e.g. Orca [55]), or surfacing accidental discordance or inaudible components in music editors. As discussed, such ambient assistants should be designed in a polite manner (e.g., through granular dismissal of advice) to avoid being irritating and then dismissed.

While most technical tasks require some amount of creativity, we argue that features in editors in creativity centered-domains should be constructed in order to align specifically with goals of either reducing tedium or aiding in exploration. Barke et al. [12] observe a similar pattern of exploration vs. acceleration in use of the AI-powered code assistant Copilot for traditional, non-creative software development tasks, suggesting overlap in editor features that

support creative coding and coding more generally. Compton [24] argues for IDEs with features that are valuable unto themselves—for example, for being playful or thought-provoking—rather than their use as a means to end. Non-productivity focused techniques may be useful in creative coding contexts more generally, perhaps as a design advisor as A4 suggested. These additions may drive unexpected patterns of usage, leading to new types of discovery through play—which might even be valuable in technical domains like data science or visualization [102]. At the same time, such interventions may inspire skepticism T4 about their authenticity if they are perceived as too whimsical or unrealistic.

6.3 Limitations and Future Work

As described throughout, our study had a number of limitations. These included data collection errors (such as the configuration error in *wi22*) and the relative simplicity of the survey. For instance, our use of static images—as opposed to videos or interactive prototypes—limited our ability to accurately explore reactions to proposed features. However, the use of non-interactive stimuli (following Kery et al. [60]) allowed respondents to project their own beliefs about the features and ignore potentially distracting low-level bugs or stylistic issues. Further, we only implemented a subset of all designs we identified, so we cannot make inferences about what features would be most valuable in general. Instead, we focus only on the observed themes and interactions with implemented features. This approach was a coarse and inexpensive way to identify and explore some potentially fruitful features, however not all such features were necessarily identified nor considered. Future work could implement more of the identified features—and also augment our observations with lab studies—to better understand the effects of particular features. Furthermore, whereas our work investigated how novices *perceive* the utility of various editor features, subsequent work should also investigate their *pedagogical* effects on learning outcomes—one notable point for comparison is that Oviatt et al. [82] found novel interfaces can hinder learning.

The biases of our particular student populations may not be reflective of a more general student population, however the views of the college-aged (*sp21*, *wi22*) students seem aligned with those of the high-school students (*su22*). In addition, they are in agreement with those of *su21* students who, because of pandemic era-distancing, attended from around the world and thus drawn from a substantially different population. Our own biases were likely projected onto the students in teaching this material, and different instructors may have inspired different responses in students. To this end, student perceptions are likely reflective of the context and content of the work they were asked to do. For instance, the open-ended nature of many assignments likely shaped student opinions of the features we asked about, which may have been different under more structured programming tasks. In future work we would like to reexamine our findings by teaching the course to and soliciting feedback from students from other institutions, age groups, and backgrounds.

Students were generally positive about the editor being online and the way in which our feedback and submission systems were integrated (A3,5), with B1 noting that they were especially beginner-friendly. Nevertheless, the choice to use a web tool had limitations. Students with inconsistent internet connections struggled with the

online environment (prompting B6 to suggest an offline mode), while others had computers that were unable to handle the computational weight of a larger web application (which made some students hesitant to explore some editor features). For instance, A2 noted that they hesitated to use auto-refresh because “*my computer was already very slow and I didn’t want my code to crash while it was running.*” These concerns were particularly prominent during the fully online *sp21* and *su21* editions. While in-person teaching has resumed (as in *wi22* and *su22*), that consideration of how to build novice-oriented tools that support those with limited internet connectivity or less powerful computers should not cease.

While our target population in this work was students, in future work we wish to understand what features instructors see as valuable or concerning in such a setting. Similarly, it would be useful to consider whether these user interface patterns are applicable to professional artists working in creative coding spaces—questions which are closely connected to Li et al.’s [67] study of the tools that artists make for themselves. Of particular relevance are artist-designed custom coding environments used for teaching and artistic practice (such as Field [30]).

In sum, creative coding has been, and continues to be, fertile soil for HCI research. We believe that studying the problems users in these creative domains face is valuable unto itself, and is ever more relevant as creative coding becomes an increasingly common way to introduce computing and to make art.

ACKNOWLEDGMENTS

We are grateful to those who made our courses possible, including the course staff (Brian Hempel, Angela Liu, and Bhakti Shah), Kevin Workman for allowing us to incorporate his Happy Coding tutorials, and the p5 community and developers for building such useful tools. We thank Lilian Huang, Shriram Krishnamurthi, Elsie Lee-Robbins, Justin Lubin, and the anonymous reviewers for their helpful commentary. Finally, we thank our students, without whom this work could not have taken place. This work was supported in part by the University of Chicago College Innovation Fund.

REFERENCES

- [1] 2021. p5.js. <https://p5js.org/>. Accessed 9/21/21.
- [2] 2021. p5.js: createSlider. <https://p5js.org/reference/#/p5/createSlider>. Accessed 9/17/21.
- [3] 2021. p5.js editor. <https://github.com/processing/p5.js-web-editor>. Accessed 9/17/21.
- [4] 2021. Utopia. <https://github.com/concrete-utopia/utopia>.
- [5] 2022. Tweakable: an online programming environment for audio and video. <https://tweakable.org/>. Accessed 8/25/22.
- [6] Khan Academy. 2021. Computer Programming. <https://www.khanacademy.org/computing/computer-programming>. Accessed 4/3/2022.
- [7] Abdulaziz Alaboudi and Thomas D LaToza. 2021. Edit-Run Behavior in Programming and Debugging. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–10. <https://doi.org/10.1109/VL/HCC51201.2021.9576170>
- [8] Susan A Ambrose, Michael W Bridges, Michele DiPietro, Marsha C Lovett, and Marie K Norman. 2010. *How Learning Works: Seven Research-based Principles for Smart Teaching*. John Wiley & Sons, New York.
- [9] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding interactive visual syntax to textual code. *Proceedings of the ACM on Programming Languages (OOPSLA)* 4 (2020), 1–28.
- [10] Kentaro Asai, Tsukasa Fukusato, and Takeo Igarashi. 2020. Integrated Development Environment with Interactive Scatter Plot for Examining Statistical Modeling. In *SIGCHI Conference on Human Factors in Computing Systems*. 1–7.
- [11] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michal Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: Embedded Programming for

- Education, in Blocks and TypeScript. In *ACM SIGPLAN Workshop on SPLASH-E*. ACM, 7–12. <https://doi.org/10.1145/3358711.3361630>
- [12] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. *arXiv preprint arXiv:2206.15000* (2022).
- [13] Daniel W Barowy, Emery D Berger, and Benjamin Zorn. 2018. ExceLint: automatically finding spreadsheet formula errors. *Proceedings of the ACM on Programming Languages (OOPSLA)* 2 (2018), 1–26.
- [14] Lyn Bartram, Michael Correll, and Melanie Tory. 2022. Untidy Data: The Unreasonable Effectiveness of Tables. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2022), 686–696. <https://doi.org/10.1109/TVCG.2021.3114830>
- [15] beautify web. 2021. JSBeautify. <https://github.com/beautify-web/js-beautify>. Accessed 4/3/2022.
- [16] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Working Group Reports on Innovation and Technology in Computer Science Education, ITICSE-WGR*. ACM, 177–210. <https://doi.org/10.1145/3344429.3372508>
- [17] Laura Beckwith, Cory Kissinger, Margaret M. Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan F. Blackwell, and Curtis R. Cook. 2006. Tinkering and gender in end-user programmers' debugging. In *SIGCHI Conference on Human Factors in Computing Systems*. ACM, 231–240. <https://doi.org/10.1145/1124772.1124808>
- [18] Andrew Bragdon, Steven P. Reiss, Robert C. Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. 2010. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *International Conference on Software Engineering (ICSE)*. ACM, 455–464. <https://doi.org/10.1145/1806799.1806866>
- [19] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *SIGCHI Conference on Human Factors in Computing Systems*. 513–522.
- [20] Cameron Burgess, Dan Lockton, Maayan Albert, and Daniel Cardoso Llach. 2020. Stamper: An Artboard-Oriented Creative Coding Environment. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. ACM, 1–9. <https://doi.org/10.1145/3334480.3382994>
- [21] Margaret M. Burnett, Anicia Peters, Charles Hill, and Noha Elarief. 2016. Finding Gender-Inclusiveness Software Issues with GenderMag: A Field Investigation. In *SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2586–2598. <https://doi.org/10.1145/2858036.2858274>
- [22] Mike Cao. 2021. Umami. <https://umami.is/>. Accessed 4/3/2022.
- [23] Adam S Carter, Christopher D Hundhausen, and Olusola Adesope. 2015. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM, 141–150. <https://doi.org/10.1145/2787622.2787710>
- [24] Kate Compton. 2021. Conversation Starter: Imagining Autotelic IDEs. In *CEUR Workshop Proceedings*, Vol. 3217. CEUR-WS.
- [25] Kate Compton, Ben Kybartas, and Michael Mateas. 2015. Tracery: An Author-Focused Generative Text Tool. In *International Conference on Interactive Digital Storytelling*. Springer, 154–161. https://doi.org/10.1007/978-3-319-27036-4_14
- [26] CSSLint. 2021. CSSLint. <https://github.com/CSSLint/csslint>. Accessed 4/3/2022.
- [27] Fred D Davis. 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly* (1989), 319–340.
- [28] Robert A DeLine. 2021. Glinda: Supporting data science with live programming, GUIs and a Domain-specific Language. In *SIGCHI Conference on Human Factors in Computing Systems*. 1–11.
- [29] Quan Do, Kiersten Campbell, Emmie Hine, Dzung Pham, Alex Taylor, Iris Howley, and Daniel W Barowy. 2019. Evaluating ProDirect Manipulation in Hour of Code. In *ACM SIGPLAN Symposium on SPLASH-E*. 25–35. <https://doi.org/10.1145/3358711.3361623>
- [30] Marc Downie and Paul Kaiser. 2021. Field. <http://openendedgroup.com/field/>.
- [31] Jonathan Edwards. 2005. Subtext: Uncovering the Simplicity of Programming. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. 505–518. <https://doi.org/10.1145/1094811.1094851>
- [32] Facebook. 2022. Jest CLI Options. <https://jestjs.io/docs/cli>. Accessed 11/15/20.
- [33] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2017. Autofolding for Source Code Summarization. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1095–1109. <https://doi.org/10.1109/TSE.2017.2664836>
- [34] Angelo Fraietta, Oliver Bown, Sam Ferguson, Sam Gillespie, and Liam Bray. 2019. Rapid composition for networked devices: HappyBrackets. *Computer Music Journal* 43, 2–3 (2019), 89–108.
- [35] Jonas Frich, Michael Mose Biskjaer, and Peter Dalsgaard. 2018. Twenty years of creativity research in human-computer interaction: Current state and future directions. In *Proceedings of the 2018 Designing Interactive Systems Conference*. 1235–1257.
- [36] Ira Greenberg. 2007. *Processing: creative coding and computational art*. Apress.
- [37] Ira Greenberg, Deepak Kumar, and Dianna Xu. 2012. Creative Coding and Visual Portfolios for CS1. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. 247–252. <https://doi.org/10.1145/2157136.2157214>
- [38] Philip Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *ACM Symposium on User Interface Software and Technology (UIST)*. <https://doi.org/10.1145/3472749.3474819>
- [39] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 579–584. <https://doi.org/10.1145/2445196.2445368>
- [40] Mark Guzdial. 2004–. Media Computation Teachers Website. <http://coweb.cc.gatech.edu/mediaComp-teach>.
- [41] Mark Guzdial. 2013. Exploring Hypotheses about Media Computation. In *ACM Conference on International Computing Education Research (ICER)*.
- [42] Mark Guzdial and Andrea Forte. 2005. Design Process for a Non-Majors Computing Course. *ACM SIGCSE Bulletin* 37, 1 (2005), 361–365. <https://doi.org/10.1145/1047344.1047468>
- [43] 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 *ACM Symposium on User Interface Software and Technology (UIST)*. 91–100. <https://doi.org/10.1145/1449715.1449732>
- [44] Baku Hasimoto. 2021. Glisp. <https://github.com/baku89/glisp>.
- [45] Marijn Haverbeke et al. 2021. Code Mirror 6. <https://codemirror.net/6/>. Accessed 4/3/22.
- [46] Juha Helminen, Petri Ihantola, and Ville Karavirta. 2013. Recording and Analyzing In-Browser Programming Sessions. In *Koli Calling International Conference on Computing Education Research*. 13–22. <https://doi.org/10.1145/2526968.2526970>
- [47] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *ACM Symposium on User Interface Software and Technology (UIST)*. 379–390. <https://doi.org/10.1145/2984511.2984575>
- [48] Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. In *European Conference on Object-Oriented Programming, ECOOP (LIPICs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:29. <https://doi.org/10.4230/LIPICs.ECOOP.2022.16>
- [49] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *ACM Symposium on User Interface Software and Technology (UIST)*. 281–292. <https://doi.org/10.1145/3332165.3347925>
- [50] T Dean Hendrix, James H Cross, Larry A Barowski, and Karl S Mathias. 1998. Visual Support for Incremental Abstraction and Refinement in Ada 95. *SIGAda Annual International Conference on Ada Technology* 18, 6 (1998), 142–147. <https://doi.org/10.1145/289524.289568>
- [51] Aspen K Hopkins, Michael Correll, and Arvind Satyanarayan. 2020. VisualLint: Sketchy in situ annotations of chart construction errors. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 219–228.
- [52] Ruanqianqian (Lisa) Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. 2022. Investigating the Impact of Using a Live Programming Environment in a CS1 Course. In *ACM Technical Symposium on Computer Science Education (SIGCSE) (SIGCSE 2022)*. Association for Computing Machinery, 495–501. <https://doi.org/10.1145/3478431.3499305>
- [53] Christopher D Hundhausen, Sean F Farley, and Jonathan L Brown. 2009. Can Direct Manipulation Lower the Barriers to Computer Programming and Promote Transfer of Training? An Experimental Study. *ACM Transactions on Computer-Human Interaction (TOCHI)* 16, 3 (2009), 1–40. <https://doi.org/10.1145/1592440.1592442>
- [54] Christopher David Hundhausen, Daniel M Olivares, and Adam S Carter. 2017. IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda. *ACM Transactions on Computing Education (TOCE)* 17, 3 (2017), 1–26. <https://doi.org/10.1145/3105759>
- [55] hundredrabbits. 2021. Orca. <https://github.com/hundredrabbits/Orca>. Accessed 9/21/21.
- [56] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, et al. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. *Proceedings of the 2015 ITICSE on Working Group Reports* (2015), 41–63. <https://doi.org/10.1145/2858796.2858798>
- [57] jshint. 2021. JSHint. <https://github.com/jshint/jshint>. Accessed 4/3/2022.
- [58] Hyeonsu Kang and Philip J Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *ACM Symposium on User Interface Software and Technology (UIST)*. 737–745. <https://doi.org/10.1145/3126594.3126632>
- [59] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *CHI*, Vol. 10. <https://doi.org/10.1145/3025453.3025626>

- [60] 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 *ACM Symposium on User Interface Software and Technology (UIST)*. 140–151.
- [61] Amy J Ko and Brad A Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *SIGCHI Conference on Human Factors in Computing Systems*. 387–396.
- [62] Masatomo Kobayashi and Takeo Igarashi. 2007. Boomerang: Suspendable Drag-and-Drop Interactions Based on a Throw-and-Catch Metaphor. In *ACM Symposium on User Interface Software and Technology (UIST)*. 187–190. <https://doi.org/10.1145/1294211.1294243>
- [63] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How Live Coding Affects Developers' Coding Behavior. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 5–8. <https://doi.org/10.1109/VLHCC.2014.6883013>
- [64] Yun Young Lee, Nicholas Chen, and Ralph E Johnson. 2013. Drag-and-drop Refactoring: Intuitive and Efficient Program Transformation. In *International Conference on Software Engineering (ICSE)*. IEEE, 23–32. <https://doi.org/10.1109/ICSE.2013.6606548>
- [65] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1–7. <https://doi.org/10.1145/3313831.3376494>
- [66] Golan Levin and Tega Brain. 2021. *Code as Creative Medium: A Handbook for Computational Art and Design*. MIT.
- [67] Jingyi Li, Sonia Hashim, and Jennifer Jacobs. 2021. What We Can Learn From Visual Artists About Software Development. In *SIGCHI Conference on Human Factors in Computing Systems*. 1–14. <https://doi.org/10.1145/3411764.3445682>
- [68] Zach Lieberman. 2020. openFrameworks. <https://openframeworks.cc/ofBook/chapters/foreword.html>. Accessed 9/21/21.
- [69] Justin Lubin and Ravi Chugh. 2020. Type-Directed Program Transformations for the Working Functional Programmer. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
- [70] Mihaela Malita and Ethel Schuster. 2020. From Drawing to Coding: Teaching Programming with Processing. *Journal of Computing Sciences in Colleges* 35, 8 (April 2020), 245–246. <https://doi.org/10.5555/3417639.3417663>
- [71] Mariana Mărașoiu, Luke Church, and Alan Blackwell. 2015. An empirical investigation of code completion usage by professional software developers. In *Psychology of Programming Interest Group (PPIG 2015)*. 59–68.
- [72] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. 499–504. <https://doi.org/10.1145/1953163.1953308>
- [73] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: on Novices' Interactions with Error Messages. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2011, part of SPLASH '11*. 3–18. <https://doi.org/10.1145/2048237.2048241>
- [74] Andrew McNutt, Gordon Kindlmann, and Michael Correll. 2020. Surfacing Visualization Mirages. In *SIGCHI Conference on Human Factors in Computing Systems*. 1–16.
- [75] Andrew M. McNutt, Chenglong Wang, Rob DeLine, and Steven M. Drucker. 2023. On the Design of AI-powered Code Assistants for Notebooks. In *SIGCHI Conference on Human Factors in Computing Systems*. To Appear.
- [76] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. 2017. Micro-Versioning Tool to Support Experimentation in Exploratory Programming. In *SIGCHI Conference on Human Factors in Computing Systems*. 6208–6219. <https://doi.org/10.1145/3025453.3025597>
- [77] Mark C Mitchell and Oliver Bown. 2013. Towards a creativity support tool in processing: understanding the needs of creative coders. In *Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration*. 143–146.
- [78] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.
- [79] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages (POPL)* 3, Article 14 (2019), 32 pages. <https://doi.org/10.1145/3290327>
- [80] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. Association for Computing Machinery, 86–99. <https://doi.org/10.1145/3009837.3009900>
- [81] Cyrus Omar, Young Seok Yoon, Thomas D LaToza, and Brad A Myers. 2012. Active Code Completion. In *International Conference on Software Engineering (ICSE)*. IEEE, 859–869. <https://doi.org/10.1109/ICSE.2012.6227133>
- [82] Sharon Oviatt, Alex Arthur, and Julia Cohen. 2006. Quiet interfaces that help students think. In *ACM Symposium on User Interface Software and Technology (UIST)*. 191–200.
- [83] Kylie Peppler and Yasmin Kafai. 2009. Creative Coding: Programming for Personal Expression. *International Conference on Computer Supported Collaborative Learning (CSCSL)* 30 (2009), 7.
- [84] Inigo Quilez and Pol Jeremias. 2013. ShaderToy. <https://www.shadertoy.com/>. Accessed 9/7/2022.
- [85] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (2019), 9. <https://doi.org/10.22152/programming-journal.org/2019/3/9>
- [86] Olli Rautiainen. 2020. How to write better code with linting, formatting, and analysis tools. <https://www.eficode.com/blog/how-to-write-better-code-with-tools>. Accessed 4/4/2022.
- [87] Casey Reas and Ben Fry. 2007. *Processing: a programming handbook for visual designers and artists*. Mit Press.
- [88] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2019), Issue 1. <https://doi.org/10.22152/programming-journal.org/2019/3/1>
- [89] Replit. 2021. Replit. <https://replit.com/>. Accessed 4/3/2022.
- [90] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live Coding: A Review of the Literature. In *ACM Conference on Innovation and Technology in Computer Science Education*, Vol. 1. 164–170. <https://doi.org/10.1145/3430665.3456382>
- [91] David Williamson Shaffer and Mitchel Resnick. 1999. "Thick" Authenticity: New Media and Authentic Learning. *Journal of Interactive Learning Research* 10, 2 (December 1999), 195–215.
- [92] Daniel Shiffman. 2021. Coding Train. <https://thecodingtrain.com/>.
- [93] Ben Shneiderman. 2007. Creativity support tools: accelerating discovery and innovation. *Commun. ACM* 50, 12 (2007), 20–32.
- [94] Beth Simon, Päivi Kinnunen, Leo Porter, and Dov Zazkis. 2010. Experience Report: CS1 for Majors with Media Computation. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.
- [95] Blair Subbaraman and Nadya Peek. 2022. p5.fab: Direct Control of Digital Fabrication Machines from a Creative Coding Environment. In *Designing Interactive Systems Conference*. 1148–1161.
- [96] Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing* 1, 2 (June 1990), 127–139. [https://doi.org/10.1016/S1045-926X\(05\)80012-6](https://doi.org/10.1016/S1045-926X(05)80012-6)
- [97] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *Workshop on Live Programming, LIVE*. IEEE, 31–34. <https://doi.org/10.1109/LIVE.2013.6617346>
- [98] Michael Toomim, Andrew Begel, and Susan L Graham. 2004. Managing Duplicated Code with Linked Editing. In *Symposium on Visual Languages-Human Centric Computing (VL/HCC)*. IEEE, 173–180. <https://doi.org/10.1109/VLHCC.2004.35>
- [99] Bret Victor. 2011. Explorable Explanations. <http://worrydream.com/ExplorableExplanations/>.
- [100] Bret Victor. 2011. Scrubbing Calculator. <http://worrydream.com/ScrubbingCalculator/>.
- [101] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How Novices Tackle their First Lines of Code in an IDE: Analysis of Programming Session Traces. In *Koli Calling International Conference on Computing Education Research*. 109–116. <https://doi.org/10.1145/2674683.2674692>
- [102] Nathalie Vladis, Aspen Hopkins, and Arvind Satyanarayan. 2020. Data Crafting: Exploring Data through Craft and Play. IEEE VIS Workshop on Data Vis Activities to Facilitate Learning, Reflecting, Discussing, and Designing.
- [103] David Weintrop and Uri Wilensky. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. In *International Conference on Interaction Design and Children (IDC)*.
- [104] Brian Whitworth. 2005. Polite Computing. *Behaviour & Information Technology* 24, 5 (2005), 353–363. <https://doi.org/10.1080/01449290512331333700>
- [105] Zoe J Wood, Paul Muhl, and Katelyn Hicks. 2016. Computational Art: Introducing High School Students to Computing via Art. In *ACM Technical Symposium on Computing Science Education*. 261–266. <https://doi.org/10.1145/2839509.2844614>
- [106] Kevin Workman. 2021. Happy Coding Tutorials. <https://happycoding.io/>. Accessed 4/3/2022.
- [107] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 152–165. <https://doi.org/10.1145/3379337.3415851>
- [108] YoungSeok Yoon and Brad A Myers. 2015. Supporting Selective Undo in a Code Editor. In *International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE, 223–233. <https://doi.org/10.1109/ICSE.2015.43>

APPENDIX

In this appendix we provide supplementary material that fell outside the scope of the main content of the paper.

- In Sec. A we make several notes about the course design and other ancillary details.
- In Sec. B we provide additional details about several of our studies.

A THE CREATIVE CODING COURSE

Here we provide additional context for our course. In Fig. 11, we show the schedule for the 3-week *su21* edition ([link to course site](#)). This edition featured only a sequence of homeworks and exercises, and did not include the self-guided project found in the “full” editions of course (*sp21*, *wi22*). A similar version of the course was also taught as *su22*. There were six individual homework assignments in *su21*, the first five of which appeared in all course editions:

1. **Color Wheel.** Recreate a given red-yellow-blue color wheel. (*function calls, color and shape-drawing APIs, trigonometric expressions*)
2. **Freeze Frame.** Pick a static frame from the “StoryBots: Shapes” video and recreate it. (*function calls, color and shape-drawing APIs*)
3. **Trees.** Use variables and arithmetic expressions to implement a symmetric tree drawing. (*variables, arithmetic, curves*)
4. **Book of Patterns.** Implement several 2-dimensional grid patterns—stripes, polka dots, checks, plaid, chevron, harlequin, argyle, and honeycomb—inspired by the designs in *My First Book of Patterns*. (*nested loops*)
5. **Snake.** Make a simple version of the classic snake game. Starter code was provided with function stubs for different aspects of a simple model-view-controller architecture (*mutable variables, arrays, objects, animation, mouse and keyboard events*)
6. **Subway Font.** Rewrite a webpage using a “font” that resembles the signage of the New York City subway. As shown on the cover of *Subway*, some letters are rendered white-on-black and others are set atop colored circles. (*HTML, CSS, DOM API, dictionaries*)

As highlighted in the main text, we designed our course primarily for college students with little-to-no programming experience who were not planning to major in computer science. In *sp21*, 4 out of the 31 students were undeclared, and among the remaining 27 students 14 different programs of study were represented. In *wi22*, 10 out of the 27 students were undeclared, and among the remaining 17 students 12 different programs of study were represented. All told, students from 23 different departments participated in the course (Fig. 12).

Based on both our study and pre-course on-boarding surveys, students self-reported high levels of prior experience (as highlighted in Fig. 3) in each edition of the course. In *sp21*, students who had previously completed computer science courses at the university—in a couple cases many such courses—were mistakenly allowed to enroll. This enrollment issue was fixed for *wi22*, but still nearly half of the students (who completed the course) self-reported prior experience through self-study, courses in high school, and from other university departments or institutions. In *su21* and *su22*, enrollment was unrestricted (the high-school students were not already associated with the university), and a large majority of these students reported prior experience. In any case, the different levels among our student populations helps color some of the observations in the main text.

B ADDITIONAL STUDY DETAILS

In this section we present aspects of our studies that did not fit in the main text: the ethics statement for our studies, additional results, followed by descriptions of the hypothetical features from our Year 1 survey that were not implemented in Year 2.

B.1 Ethics Statement

All studies were reviewed and determined to be exempt by our university’s institutional review board. We did not collect demographic data, because it was not a core aspect of our investigation. Although we designed and taught these courses with an eye towards the associated studies, we believe the course materials we developed and delivered (through lectures, office hours, and online discussions) were minimally affected by the presence of these studies and our use of custom versions of the p5 editor.

B.2 Additional Results

Here we list a series of one-off results that were observed. Then in Sec. B.2.1 include an analysis of the code folding feature (original part of the analysis in Sec. 5.1). Finally we provide an additional analysis of the the auto-refresh feature in Sec. B.2.2.

- In Fig. 13, we show an alternative depiction of the results from both years of our survey which includes metrics other than the one used in the main text (namely Usefulness).
- In Fig. 14 we provide a simple summary of the volume of code executions across all three editions along with assignment due dates, which highlights that execution volume tended to be higher for earlier graphic-only assignments (compared to later assignments which involved interactivity or HTML/CSS).
- In Fig. 15 we provide a related graphic showing execution history for *sp21* and *su21*, along with the relative error rates by day. In Year 1, our logging scheme did not include a mechanism for explicitly collecting run-time errors, so they were reconstructed post-course by running each logged sketch for 10 seconds and collecting all errors generated during that period. This approach may exclude errors students saw, such as those generated through interaction with the sketch or through randomness. On average each session had $\mu=7.27\pm32.8$ errors, with outliers excluded. Within our reduced sample from Year 2, sessions exhibited $\mu=30.7\pm95.8$ errors, again with outliers excluded. A one-sided t-test indicated that there were significantly more mean errors per session in Year

2 ($p < 0.001$). This increase is likely due to the new collection method, which captured errors witnessed by the user rather than just reconstructed errors.

- In Fig. 16 we provide a figure showing the bi-gram action sequence probability of actions in Year 1.
- We then provide tables in Fig. 6 for Fig. 17 and Fig. 18.

B.2.1 Code Folding. This simple feature, common to most modern editors [33], allows functions and other blocks of code to be collapsed and later expanded. This feature was generally well liked as it made code “*feel more organized*” (A9), while helping users avoid “*being overwhelmed*” (A2) and making things “*look neater and less intimidating*” (A1). This has the organizational benefit that it is “*easier to find specific chunks of code*” (A1), which, as noted by A13 and A16, reduces the amount of scrolling—these are well-understood benefits of this feature [50]. Being able to organize and navigate code are important concerns for novice creative coders. T3

However, the feature was not universally appreciated. Whereas B8 found that code folding “*Helped a lot while debugging and re-organization!*”, A9 asked “*when debugging, what if the problem is in one of the lines of code that are hidden?*” A number of participants noted that they simply did not use it or did not find it helpful. Some students only invoked it accidentally (A10), while others found it confusing (A5,14) because it did not clearly communicate what code was to be folded. Code folding, or other interface-based code organization tools, seem especially valuable in this context as most sketches typically involved only a single file (e.g. *sp21* and *wi22* final projects had a median of 1 JavaScript file). As the file structure abstraction for code organization is underused, there is opportunity for interface-based abstraction.

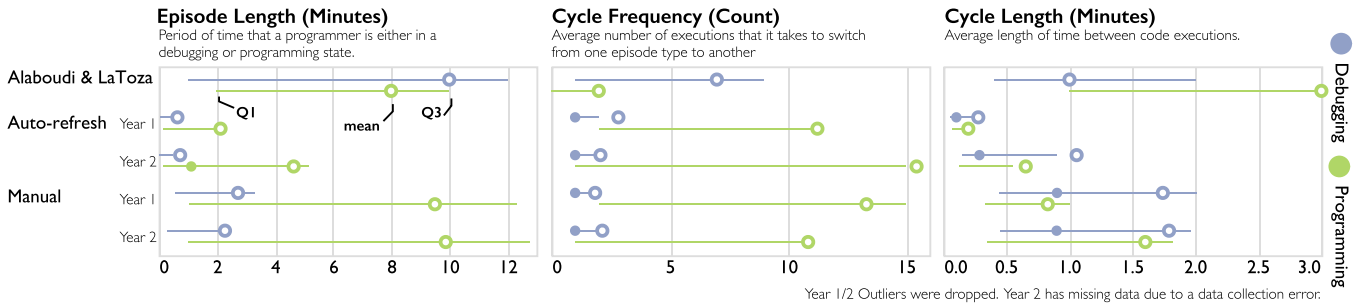


Figure 10: Comparing how students shifted between debugging and programming states (using different execution styles) against a baseline of professional programmers [7].

B.2.2 Live Coding. In addition to the analysis of the auto-refresh feature considered in the main text (Sec. 5.1.3), we also sought to understand how student edit-run behavior compared to that of professional programmers. Fig. 10 shows how auto-refresh usage affected the length, size, and frequency of edit-run cycles with regard to debugging versus programming states adopting the metrics used by Alaboudi and LaToza [7], who studied how professional programmers shift between debugging and programming states during edit-run cycles in their own work. A salient difference from the baseline was that the number of executions to transition from a programming to a debugging state (and vice versa) was shorter for our students. This is likely informed by the domain: the professionals were working on projects such as Firefox and Curl, which likely have a different execution cadence than the graphic-oriented work conducted in creative coding. The programming episode length was similar for the professionals and those using manual execution—although debugging episodes for the latter group were much shorter, suggesting that the errors were much less complex for our students. However, given the differences in expertise and domain between these groups, it is difficult to identify a primary cause of the changes.

The key observation is that the usage patterns exhibited by our students were not the same as those of professionals, but were not fundamentally dissimilar. This suggests the potential transferability of our observations about novices to more experienced users.

Using auto-refresh does not appear to have an effect on cycle frequency, although it seems to be associated with shorter episodes and cycle lengths. This coheres with our expectation of auto-refresh, as it triggers executions more quickly than one might with manual execution, but suggests a certain consistency related to task.

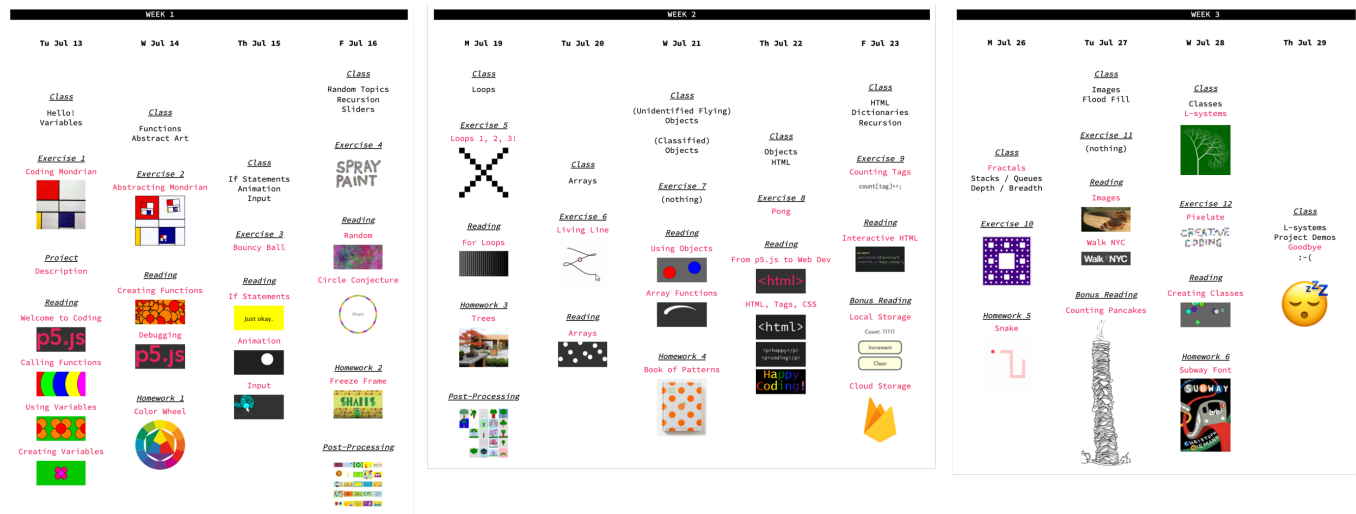


Figure 11: Schedule for su21 edition of the course. Readings and their corresponding images were adapted from Workman's HappyCoding tutorials [106].

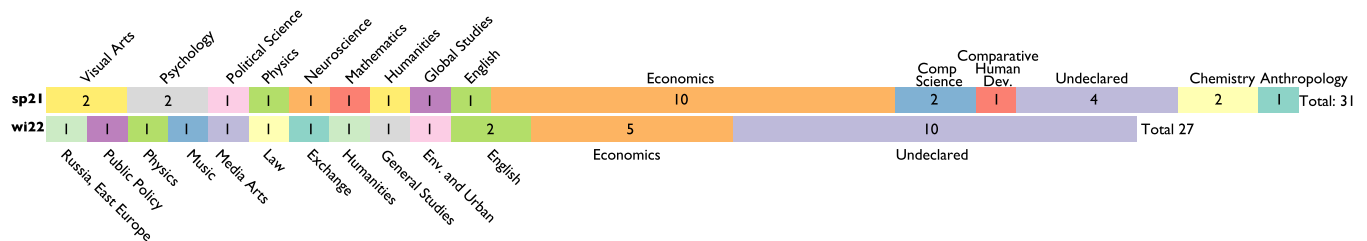


Figure 12: Home departments of students who completed the sp21 (top) and wi22 (bottom) courses.

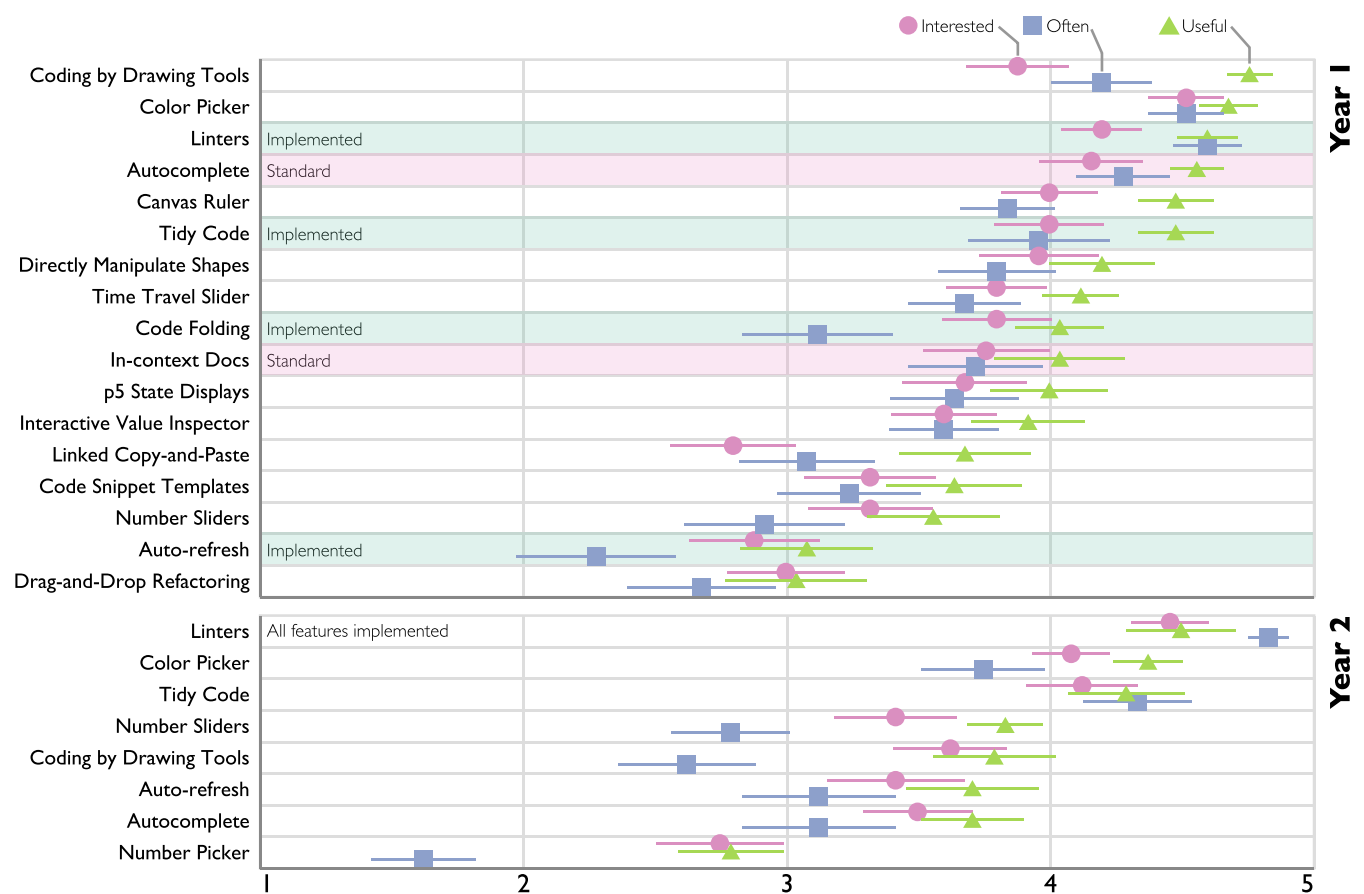


Figure 13: Participants in both the long (Year 1) and short (Year 2) survey were asked about a variety of features and rated each of them on how **Interested** they were in it, how **Often** they would use it, and how **Useful** they thought it was. Most features considered are non-standard, however several were implemented in our editor or standard (but not implemented). It is notable that although some of the most commonly instrumented features are not necessarily predictive of perceived utility.

Average Executions per Student

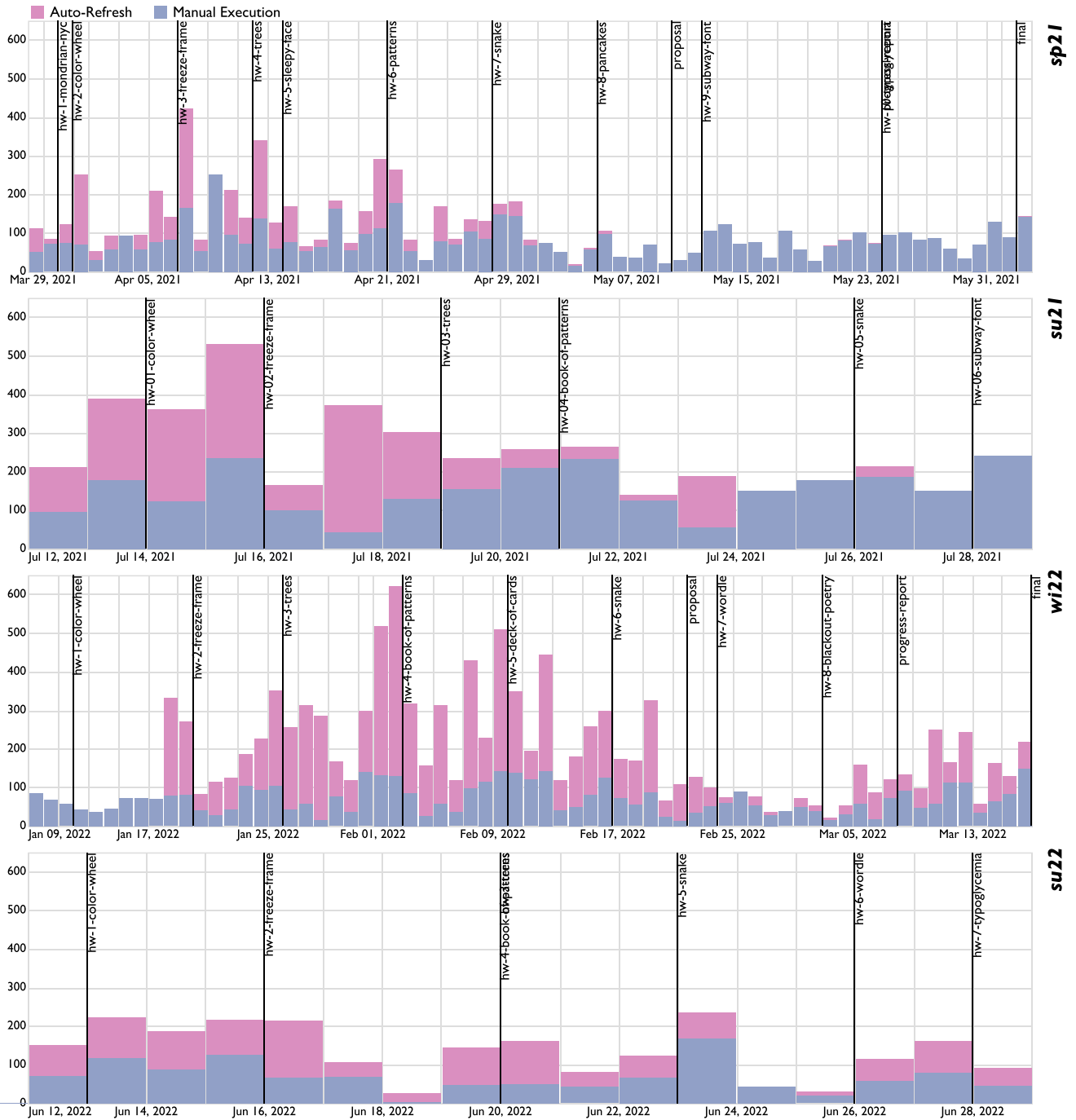


Figure 14: A summary of executions for each of the course editions show auto-refresh versus manual execution.

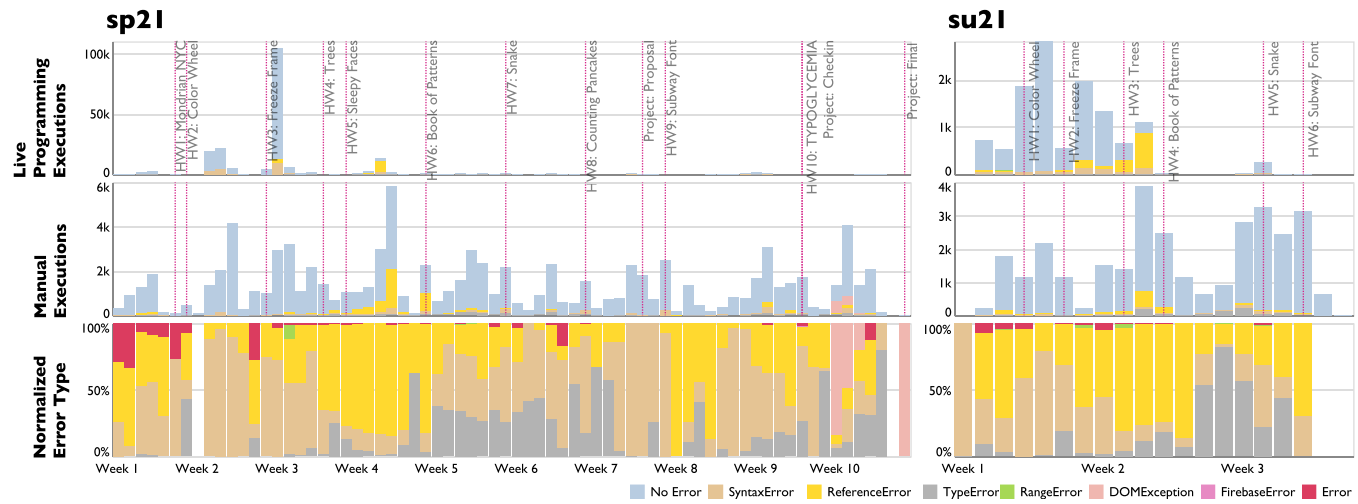


Figure 15: Errors over time in the first year of courses. Outliers have not been dropped. Our research questions are generally not motivated by analysis of error types, as they were typically driven by course material, rather than by interface design.

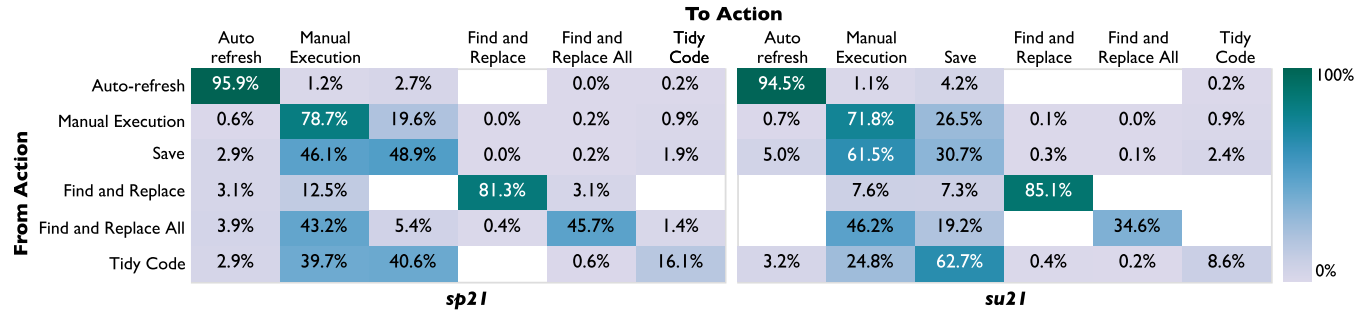


Figure 16: The bi-gram action sequence probability in Year 1 shows the rate at which a given action is followed by another particular action. We do not show *wi22* because we mistakenly did not collect Tidy Code executions.

Feature Name	rating mean	σ	Q1	Q3
Auto-refresh	3.08	1.26	3	4.00
Autocomplete	4.56	0.51	4	5.00
Canvas Ruler	4.48	0.71	4	5.00
Code Folding	4.04	0.84	3	5.00
Code Snippet Templates	3.64	1.29	3	5.00
Coding by Drawing Tools	4.76	0.44	5	5.00
Color Picker	4.68	0.56	4	5.00
Directly Manipulate Shapes	4.20	1.00	4	5.00
Drag-and-Drop Refactoring	3.04	1.34	2	4.00
In-context Docs	4.04	1.24	4	5.00
Interactive Value Inspector	3.92	1.08	4	4.00
Linked Copy-and-Paste	3.68	1.25	3	5.00
Linters	4.60	0.58	4	5.00
Number Sliders	3.56	1.26	3	5.00
Tidy Code	4.48	0.71	4	5.00
Time Travel Slider	4.12	0.73	4	5.00
p5 State Displays	4.00	1.12	3	5.00

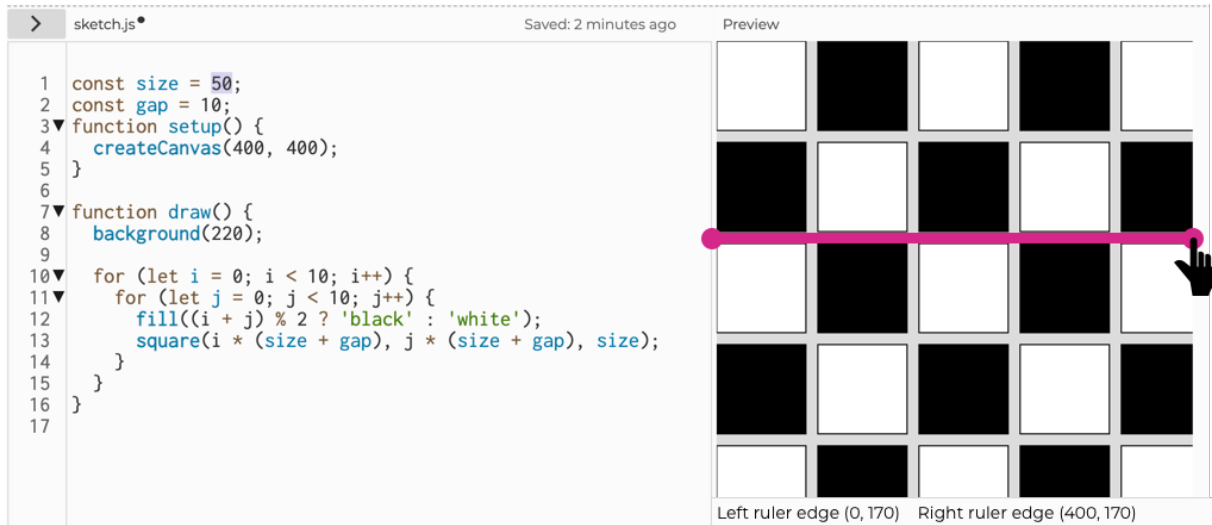
Figure 17: The computed values for Fig. 13, the survey results relating to Usefulness from Year 1.

Feature Name	rating mean	σ	Q1	Q3
Auto-refresh	3.71	1.23	3	5.00
Autocomplete	3.71	0.95	3	4.00
Coding by Drawing Tools	3.79	1.14	3	5.00
Color Picker	4.38	0.65	4	5.00
Linters	4.50	1.02	4	5.00
Number Picker	2.79	0.98	2	3.25
Number Sliders	3.83	0.70	3	4.00
Tidy Code	4.29	1.08	4	5.00

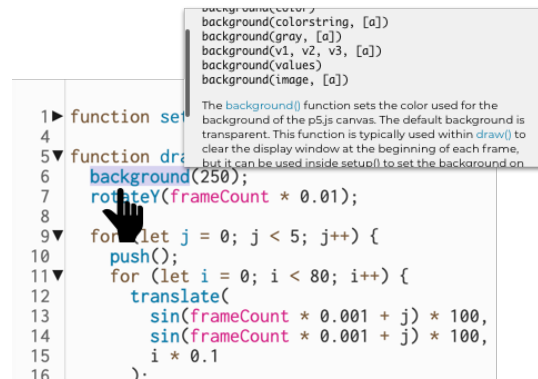
Figure 18: The computed values for Fig. 13, the survey results relating to Usefulness from Year 2.

B.3 Hypothetical Features

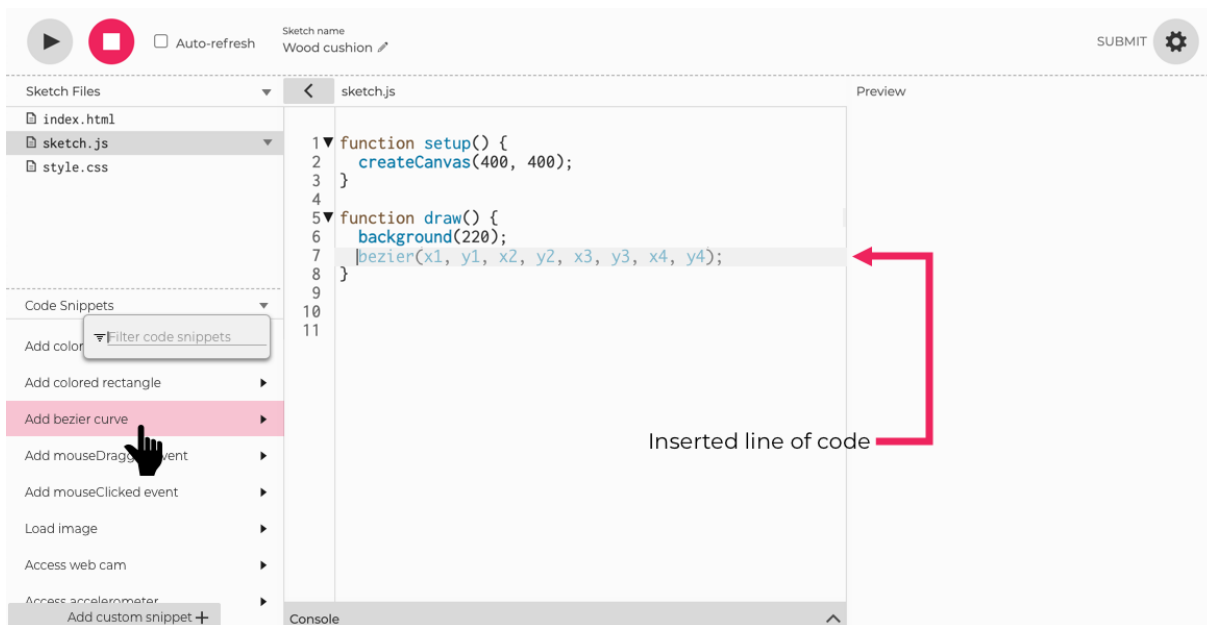
Here we return to the hypothetical features asked about in Year 1 surveys but not implemented in p5/y2. Because responses were based only on a brief description and static image, we limit our discussion of each feature.



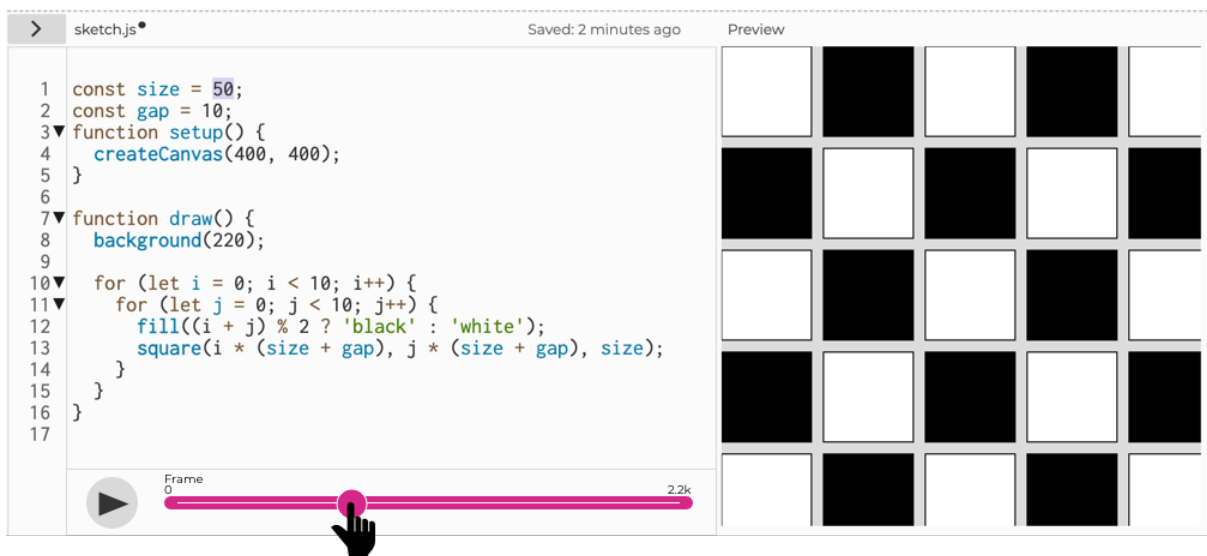
Canvas Ruler. As mentioned earlier, the Canvas Ruler was widely viewed as a useful tool to add for creative coding—however, B8 felt “it would take away the fun of mouseX and mouseY!” Several additional suggestions were made, such as being able to “measure angles, so a ruler and compass.” (A16) In future editions of the course we intend to return to this feature, as it seems like a natural next step. The primary concern in implementing such an addition would be that it does not clutter the interface T3, and perhaps, per commentary on Syntax Templates, be controllable with a keyboard.



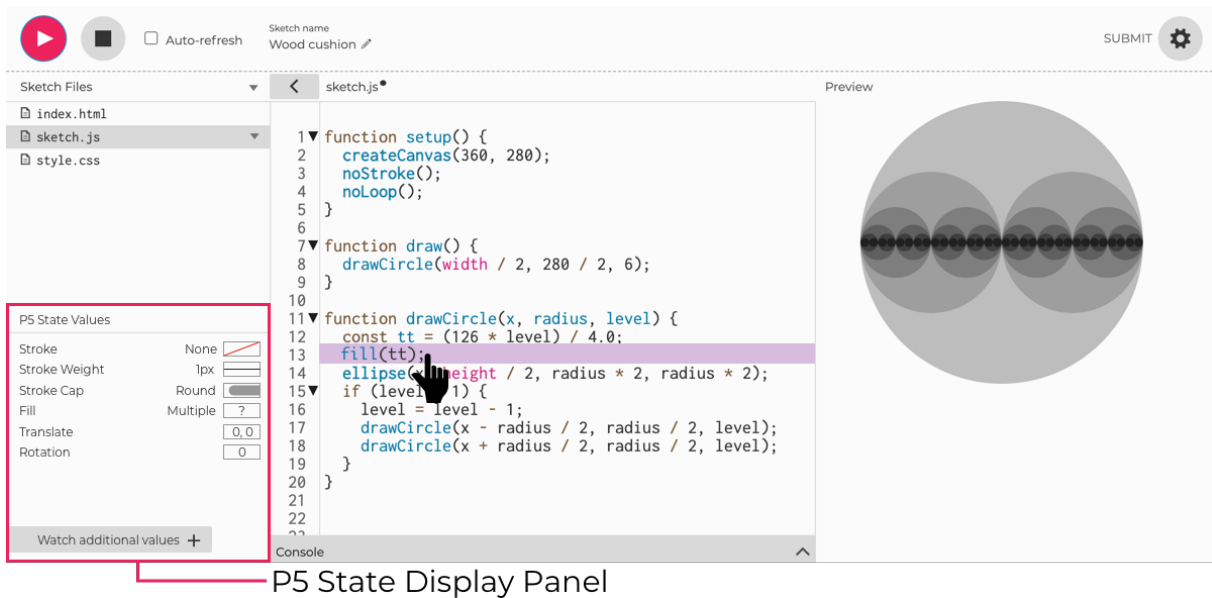
In-context Docs. Many full-featured editors (e.g. VS Code) include relevant documentation about language features and user-defined variables as a tooltip. As with autocomplete, B4 believed In-context Docs would be helpful because “gives me an idea of what to [write].” Several participants echoed this sentiment, believing that it “would have drastically widened my skill set” (A14). On the other hand, A4 was “actually a little torn by [it] because I think googling and traveling to the reference is really important. It may start off as inconvenient but just becomes more natural with practice”—which is in line with our observations about student skepticism. T4 Among the quantitative ratings from the surveys in the first year, this feature was the only one that had a statistically significant relationship ($p < 0.01$) with self reported experience was in-context docs, in particular exhibiting a negative correlation ($r = -0.308$). It is possible that an alternative presentation of this feature (perhaps in the search-based style of Blueprint [19]) might elicit more positive responses, however, based on these results we believe that users might be similarly skeptical, although exploration of such responses could be usefully explored in future work.



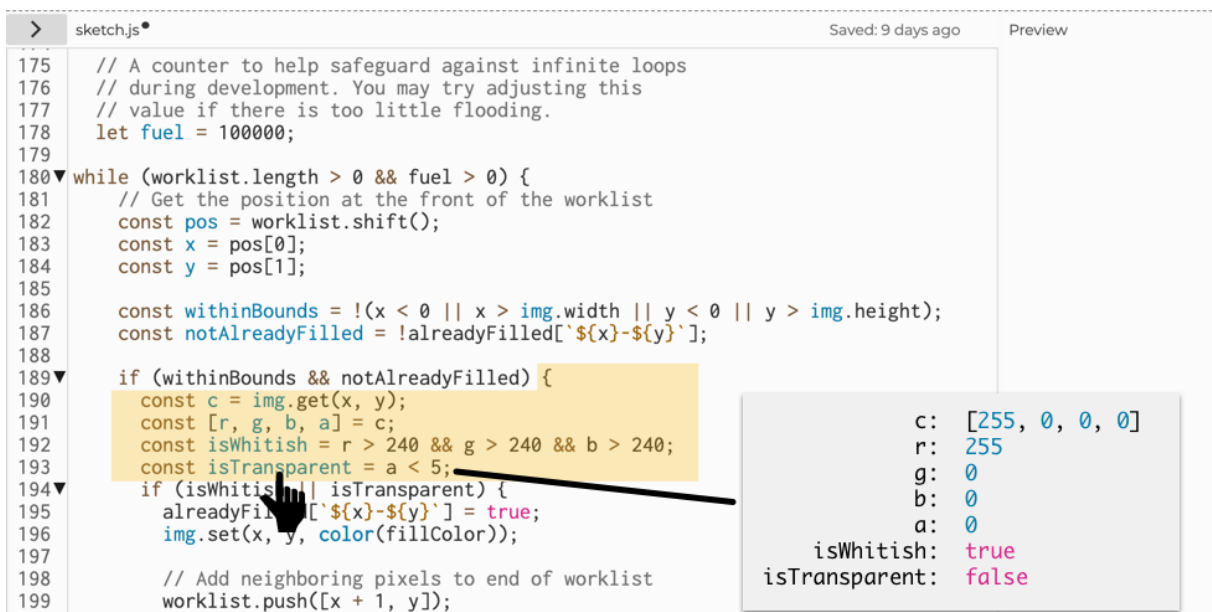
Syntax Templates. The syntax templates we implement in our autocomplete were similar to our proposed Code Snippet Templates, however the latter feature was docked (in the manner of Google Colab's Code Snippet library), and thus required mouse clicks, which may have dampened enthusiasm for the feature: “I think if there were keyboard shortcuts for these then I would use them extensively” (A13). Some thought these features would be an “easy way to get students started with no experience” (A24), but as discussed in Sec. 6 others were skeptical. We still believe this feature would be valuable to implement in the future, possibly integrated into the autocomplete, in order to keep the interface tidy and unencumbered. ^{T3}



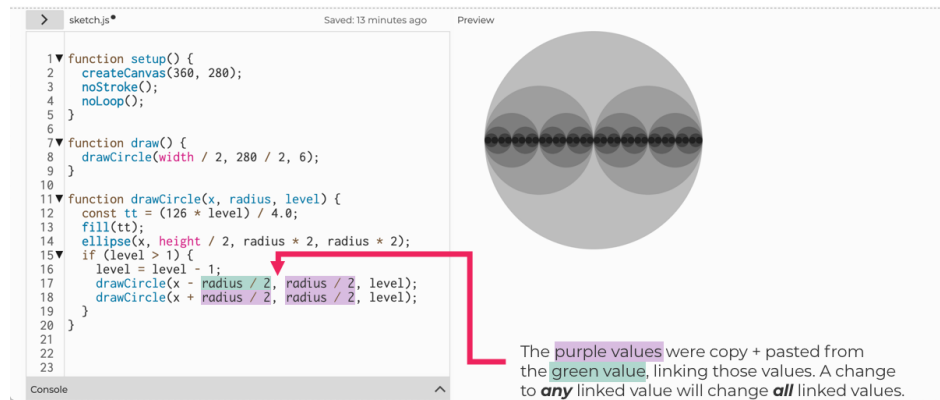
Time Travel Slider. This proposed feature would allow the state of the code execution to be paused and rewind in order to support debugging tasks—which a majority of respondents either understood as a GUI-based shortcut for p5's `frameRate` setting (which specifies how many times per second the draw loop is called) or as a mechanism for version control, both of which, while interesting, are not the feature we intended. While several students expressed enthusiasm for this latter idea (indicating the potential utility of a Variolite-style [59] or other selective undos, such as that of Yoon and Myers [108] or Mikami et al.'s [76] Micro-Versioning), this did not yield coherent feedback, beyond confusion about unfamiliar features.



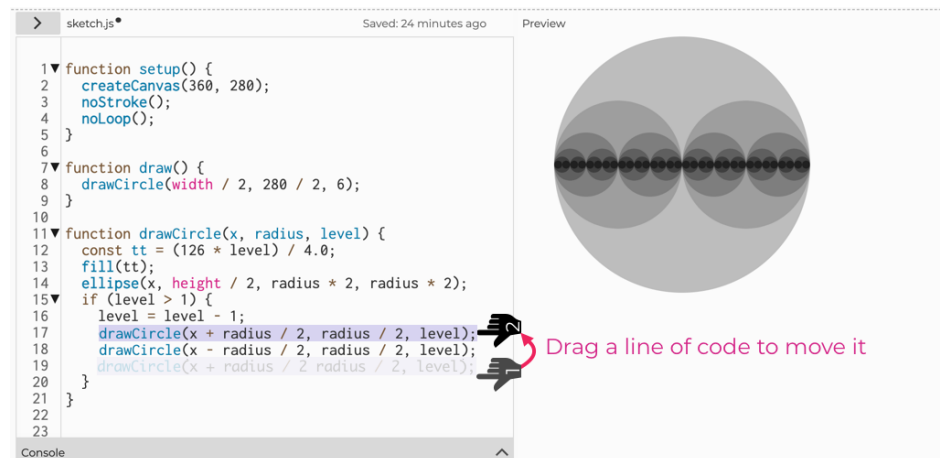
p5 State Displays. A display of current values for common library variables, such as `strokeWidth` and `fill` color, at particular lines of code—similar in spirit to object value displays in creativity tools like Illustrator. Some students were enthusiastic about this feature, noting that it “*would be extremely useful to be able to see all this information in one place*” (B7), while others felt it might enrich creativity by showing what options are available (A9). Others were less enthusiastic, noting that it would be “*a little redundant*” (A1) with running the code, or that it would be tedious (A7) compared to simply writing code.



Interactive Value Inspector. A growing thread of research allows users to inspect the current value of variables at various lines of code on demand, such as in Lerner’s Project Boxes [65] or Kang and Guo’s [58] “DISPLAY ALL THE VALUES!” approach to novice coding in Omnicode, as well through live probes [85]. In this feature we proposed a Projection Box style feature that included a customizable inspector. Students were generally enthusiastic about this feature, noting that it would be helpful for beginners (B3,8) as it would make “*loop definitions*” (A16) and debugging (A1,10). Yet some worried that the implementation might be overwhelming (A5) or distracting (A6), or would not substantially improve over console. Log-based debugging (A14). In addition one skeptical student believed that it might “*make the coder (especially early learner) to be lazy*” (A15), and prevent them from learning good debugging skills (T4).



Linked Copy-and-Paste. Vihavainen et al. [101] note that novices tend to make heavy use of copy-paste, so a natural point of enhancement then would be to embed variable-style abstraction into copy-paste itself. This idea has been discussed in research works previously [31, 98], however is not typically seen in this style of editor. Some students thought this would be helpful, by “*facilitat[ing] better organizational practices*” (A12) or in niche situations (A7,13). However, most others were apprehensive about the feature’s value. Some noted that it seemed to be a more oblique version of creating a variable (A1,16). Some thought that what was already in the editor sufficiently addressed any tasks linked copy-and-paste might accomplish, through regular copy-paste (A9) and Find & Replace (B8). A13 argued that a Sublime-style multi-cursor selection would be more flexible and preferable. We note that multi-cursor support was enabled in our editor (as part of CodeMirror), although students were not explicitly made aware of this functionality. Others still simply thought it would not be useful, and would “*creat[e] mess for me*” (B4) or otherwise be confusing (A10, B5). ^{T3}



Drag-and-Drop Refactoring. Clicking and dragging values to create arguments, variables, and other functions in a technique that has been previously explored to useful effect [64]. In this feature we proposed a simple version of this feature, however our presentation lacked the nuance of the presentations used by Lee et al. [64], which may have led feature being rated lowest. Although a few respondents were intrigued, some said they would prefer copy and paste (A12, B1,5,6,8), most were disinterested. For example: “*I personally don’t like dragging and dropping things because there is room for dragging and dropping into the wrong section especially if your computer is slow. I don’t think copy paste was too time consuming and encourages greater accuracy*” (A2); several others shared these views about efficiency and accuracy. In addition, there were concerns about the usability of the feature: “*Clicking and dragging is not an ergonomic motion on a laptop touchpad*” (A6). We highlight this as an especially valuable concern, as dragging may not be an accessible motion for some users, although something like Kobayashi and Igarashi’s suspendable drag-and-drop interactions [62] may usefully address these concerns.

Other Suggested Features. Beyond the hypothetical features we presented, some respondents suggested ideas like scratchpads or selective execution contexts similar to some of the ideas expressed in Code Bubbles [18] or Jupyter notebooks. Others suggested course-specific affordances, such as hints relevant to the assignment or integration of the assignment directly into the editor. This has a similar flavor as DrRacket’s language levels, and Marceau et al. [72] briefly sketched out learner-attuned error messaging levels. This is similar to Interactive Tutor Systems [53] which integrate curriculum and course work into a single environment. While this level of integration can be helpful, it may undermine the utility of an in-class instruction model because such interfaces are naturally self- rather than group-paced, although that should be investigated in future work.

C SURVEY INSTRUMENT FOR INITIAL SURVEY – YEAR 1 (*sp21*, *su21*)

C.1 Page 1: Consent to Participate in a Research Study

Research Project Title: Post-Course Survey of Students in Creative Coding (2021)

Principal Investigator: Ravi Chugh

Graduate Student: Andrew McNutt

IRB Protocol: IRB21-1062

This form is designed for students younger than 18 years of age who took the Creative Coding Pre-College Immersion class in Summer 2021 and their parents, respectively referred to as “you” and “your child” below. You (or your child) is being asked to take part in a research study. This form has important information about the reason for doing this study, what we will ask you (or your child) to do, and the way we would like to use information about you (or your child) if you choose to allow yourself (or your child) to be in the study.

Purpose of Research Study: You are (or your child is) being asked to participate in a research study regarding the usability of editors for creative coding. In our recently completed course we used an in-browser editor that was slightly modified from the publicly available p5 editor. We are interested in understanding what editor features might be useful to someone learning to code (particularly in the context of a creative coding course) or otherwise making digital art works. Ultimately, this research may be published and presented at scientific conferences to improve the community’s knowledge about editors for creative coding, and may be used to improve the editor used in future iterations of our course.

Participation Procedures and Activities: The full extent of the procedure will involve completing this survey. We anticipate that completion of this survey will take up to 60 minutes. Due to the difficulty of determining credit for partial completion, no compensation will be provided for partial completion. At the end of the form you (or your child) will provide a student id and preferred email address, and you (or your child) will receive a \$30 Amazon gift card for participating.

Consent and Assent Process: If you are (or your child is) 18 years or older, you (or your child) can provide the consent required to opt-in to the study. If you are (or your child is) under 18 years of age, you can give your assent (or you can give your parental consent) to join the study. For students under 18 years of age, participation in this study requires both consent from a parent as well as assent from the student.

Risks/Discomforts of Being in this Study: The risks to your participation in the survey are those associated with basic computer tasks, including boredom, fatigue, or mild stress. Benefits of Being in this Study The only benefit to you (or your child) is the learning experience from participating in a research study. The benefit to society is the contribution to scientific knowledge.

Confidentiality of Data and Limits to Confidentiality: Any reports and presentations about the findings from this study will not include your (or your child’s) name or any other identifying information.

Use of Your Research Data: We will never share the data beyond the University of Chicago research team. However, an analysis of the data may be analyzed and published in scientific conference proceedings or journal articles. The free-text responses provided to any portion of this survey may be quoted in part or in whole in this publication. We will remove any information from the analysis that could identify you (or your child) before providing the analysis for publication.

Voluntary Participation and Right to Refuse or Withdraw: Participation in this study is voluntary. The decision to participate in this study is entirely up to you and your child. You (or your child) may refuse to take part in the study at any time without prejudice or penalties and will not result in any loss of benefits to which you (or your child) are otherwise entitled.

Mandatory Reporting of Child Abuse or Neglect: The research study staff are mandated reporters and are required to report suspected child abuse or neglect to the Illinois Department of Child and Family Services. For more information, please see the University policy: <https://tinyurl.com/mr26uazn>

Contact Information for Research Questions and Participation: If you have questions or concerns about the study, you can contact the researchers at:

Principal Investigator
Ravi Chugh,
Associate Professor
John Crerar Library
University of Chicago
5730 S Ellis Ave
Chicago, IL 60637
Email: rchugh@uchicago.edu

Graduate Student
Andrew McNutt,
PhD student
John Crerar Library

University of Chicago
5730 S Ellis Ave
Chicago, IL 60637
Email: mcnuttt@uchicago.edu

If you have any questions about your rights as a participant in this research, feel you have been harmed, or wish to discuss other study-related concerns with someone who is not part of the research team, you can contact the University of Chicago Social and Behavioral Sciences Institutional Review Board (IRB) Office by phone at (773) 702-2915, or by email at sbs-irb@uchicago.edu.

Parental Consent

- (1) Parent full name (Last, First)
- (2) Parent Email address
- (3) I have read and understood this consent form. Yes ☐ no ☐
- (4) I am a parent and give consent for my child, under 18 years of age, to participate in this study. Yes ☐ no ☐

Student Assent

- (1) Student full name (Last, First)
- (2) Student Email address
- (3) Student GitHub username (same as used for homework submission in this class)
- (4) Student CNetID (the username before your @uchicago.edu email address)
- (5) I have read and understood this consent form. Yes ☐ no ☐
- (6) I am a student, under 18 years of age, and give assent to participate in this study. Yes ☐ no ☐

C.2 Page 2: Introduction and Reflection

In this section, we'll ask you some questions about your programming background, and to reflect on your experience during the course.

- (1) **Pre-Course Experience.** How much programming experience did you have prior to taking the course?
- (2) **Post-Course Confidence.** How confident do you feel in your programming skills after taking this course? Have they improved?
- (3) **Challenges.** What aspect of coding or learning to program gave you the most trouble? As a way to help organize your thinking, consider the assignment that you had the most difficulty with. Could the editor have done anything to help you with that?
- (4) **Debugging.** Think about the experience of debugging. How did you go about doing it? If you used `console.log` to help debug, did you find it helpful? Did you use any other strategies? Is there anything about it or the debugging process that you wish could have been different?
- (5) **Error Messages.** Think about the error messages you encountered (inline in the code box, in the console area under the code box, in the browser console, or elsewhere). Were they useful? How did you deal with them? Do you wish they were presented differently?
- (6) **Code Organization.** How did you go about organizing your code? For instance, how did you decide where to place variables, create functions? Was there ever a point when your organizational scheme ran into problems, if so how did you handle it? Is there anything the editor could have done to help you during these organizational tasks?
- (7) **Freeze Frame Homework.** Think about the freeze frame assignment (or any other time during the course when you needed to repeatedly edit and re-run the code in order to get particular positions or other values to achieve a desired effect). Is there anything the editor could have done to help you get your image to be just right?
- (8) **External Tools.** It's natural to use other tools as part of the programming process, such as color eye droppers or p5's online documentation. Do you think it would be useful to integrate these tools as part of the editor? What other tools can you imagine wanting to be part of your in-editor coding workflow?
- (9) **Desired Features.** What sorts of editor features might have allowed you to be more effective in your coding? What sorts of editor features might have allowed you to be more creative?

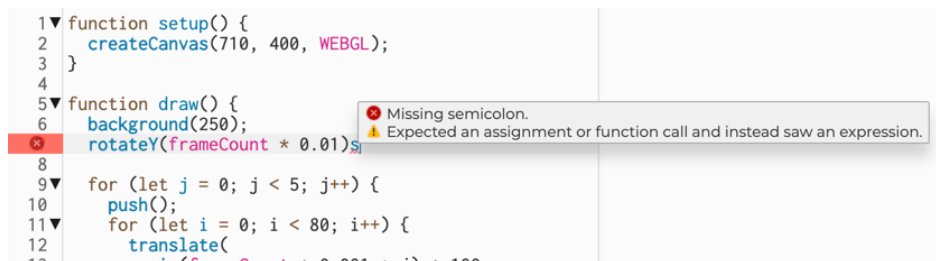
C.3 Pages 3-18: Editor Features

In this section, we'll ask for your thoughts and opinions about some features that appeared in the editor as well as some hypothetical features that we may implement for future iterations of the course.

- (1) **Autocomplete** Imagine an editor feature which provides autocomplete suggestions as you type. This would be akin to the predictive text feature found in many messaging applications, but would be sensitive to variables you've created and functions available from imported libraries. While this feature appears in some other editors it did not appear in our editor.

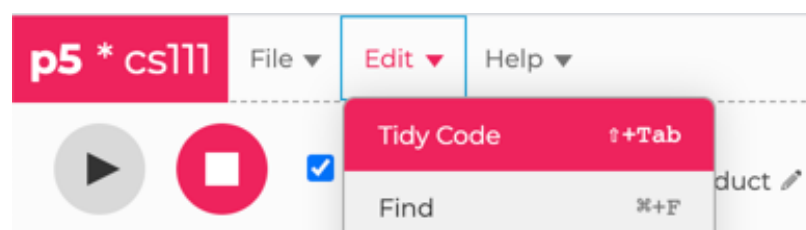


- (a) Do you think this would be useful? (1) Not Very Useful ○, (2) ○, (3) Neither useful nor useless ○, (4) ○, Very Useful ○
 (b) How often do you think you would use this feature? (1) Never ○, (2) ○, (3) Occasionally ○, (4) ○, All the time ○
 (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (2) **Linters** Our editor featured a tool called a “linter” that surfaced stylistic or coding errors through on-screen alerts, as in the image below. This tool is analogous to spell- and grammar-checkers in standard word processors. The particular linter used in our editor, called JSHint, tends not to give many warnings for stylistic errors. Other available linters give many more warnings for stylistic errors.



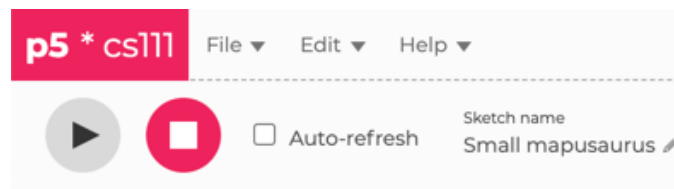
Lint error and a Lint warning

- (a) Do you think this is useful? (1) Not Very Useful ○, (2) ○, (3) Neither useful nor useless ○, (4) ○, Very Useful ○
 (b) How often do you think you used this feature? (1) Never ○, (2) ○, (3) Occasionally ○, (4) ○, All the time ○
 (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (3) **Tidy Code** Our editor featured a button called “Tidy Code” which automatically reorganized your code. This feature is sometimes seen in other editors and is more commonly known as an “auto-formatter”. These can typically be configured to enforce a particular coding style.

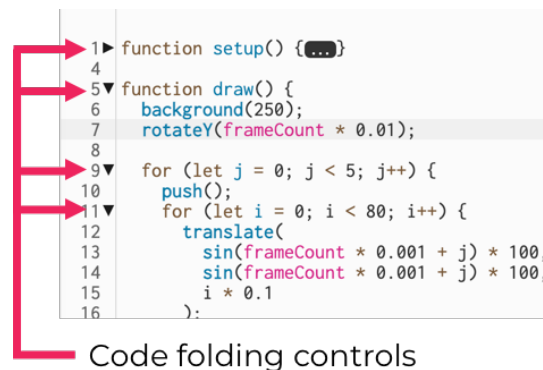


- (a) Do you think this is useful? (1) Not Very Useful ○, (2) ○, (3) Neither useful nor useless ○, (4) ○, Very Useful ○
 (b) How often do you think you used this feature? (1) Never ○, (2) ○, (3) Occasionally ○, (4) ○, All the time ○
 (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?

- (4) **Auto-refresh** There is a feature in our editor called “Auto-refresh.” When selected, it re-runs your code every time you finish typing (or sometimes before). This enables small update cycles as you code.



- (a) Do you think this is useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you used this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (5) **Code Folding** There is a feature in our editor, and many other editors, called “code folding” as in the screenshot below. This allows you to collapse certain sections of code, such as functions and loops. The “folded” code is still there and can be referenced from other places, but it’s temporarily hidden and replaced with “...”.

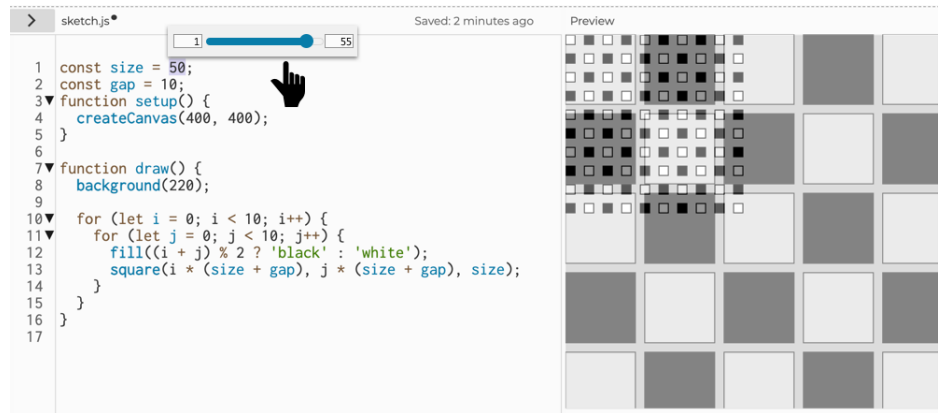


- (a) Do you think this is useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you used this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (6) **Canvas Ruler** Imagine a feature which allows you to place a draggable ruler into the drawing side of the editor. You can use it as a way to visually identify screen coordinates. This feature might involve a way to display the current direction and placement of the coordinate origin, especially with regard to translation and rotation functions.

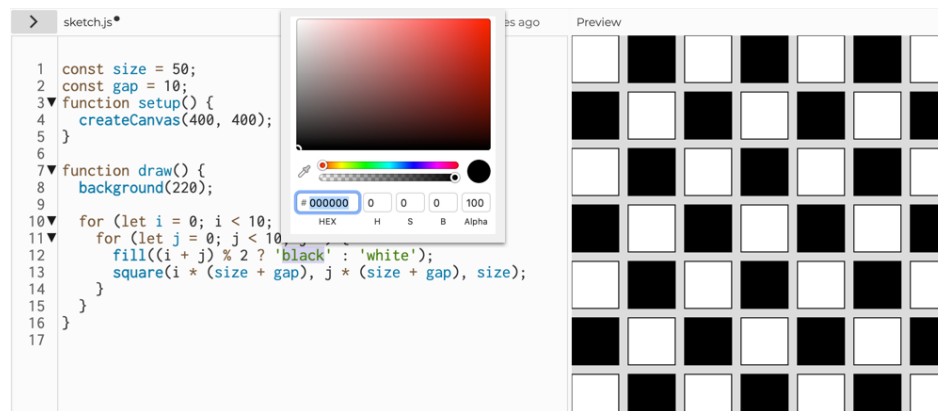


- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?

- (7) **Number Sliders** Imagine a feature which allows you to modify the numeric values in the code without typing or re-running the program (such as in the image below). With this feature, you click a value of interest and then drag a slider that appears above it to change it. The canvas is continuously re-rendered as you drag the slider. This would be similar to using p5's slider function, but, rather than just changing the value in the running code, it would also modify the text of the code.

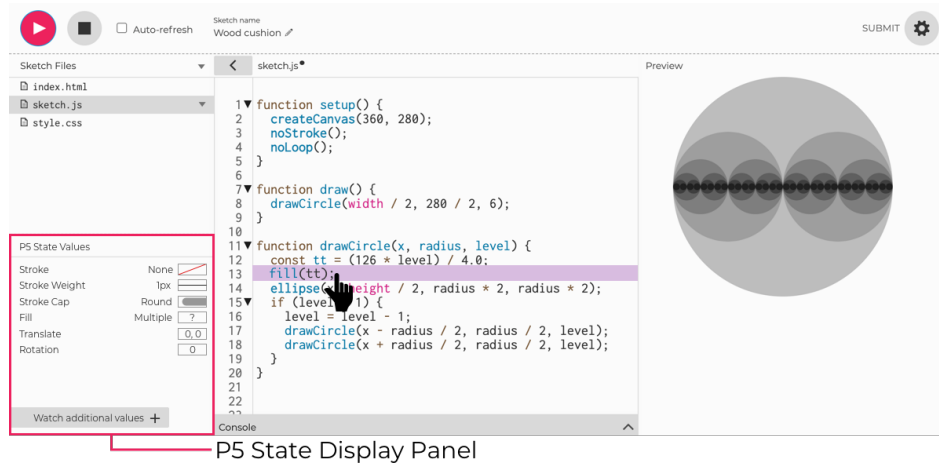


- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor unuseful ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (8) **Color Picker** Imagine having a color picker integrated into the editor. When selecting color values in the code, the color picker could appear on hover (as in the image below) to modify the value, or the tool could be docked into the bottom of the editor (allowing it to be always on). This could include pre-configured or document-based palettes, as in Illustrator or Photoshop.



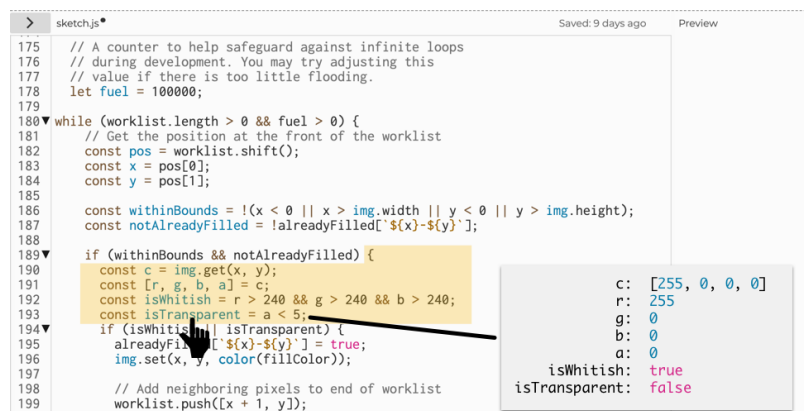
- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor unuseful ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?

- (9) **p5 State Displays** In p5 it is common to set values for variables such as `strokeWidth` (which describes the width of subsequently drawn lines), or `fill` (which describes the interior color of subsequently drawn shape). These are examples of "state variables". There are a variety of such variables in p5, however (in contrast with digital drawing tools like Photoshop), these variables are not displayed anywhere in the editor. Imagine a feature where all of the relevant state values are shown, such that when you move the text cursor to a line in your code, the display shows the state values at that point in time. This would allow you to evaluate if your drawing tools are configured as you want them to be.



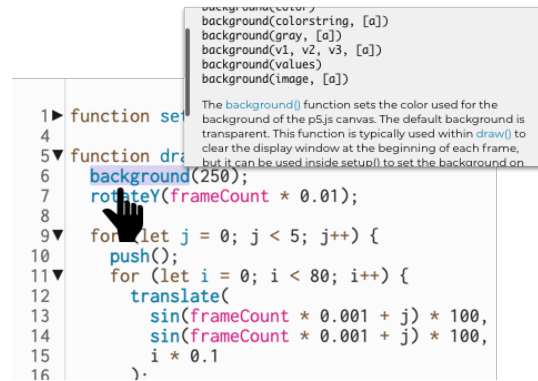
P5 State Display Panel

- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (10) **Interactive Value Inspector** Imagine a feature which allows you to see the value of the current program execution by hovering over chunks of the code. It would provide similar information as when inserting `console.log` statements into your code, but instead you would extract that same information through hovering. In contrast with "p5 State Displays" (which only shows p5 state variables like `fill` and `strokeWidth`) this feature would allow you to see both state variables as well as the value of all variables, including ones you've defined. This information could be presented through a tooltip (as in the below image) or through a docked panel.



- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?

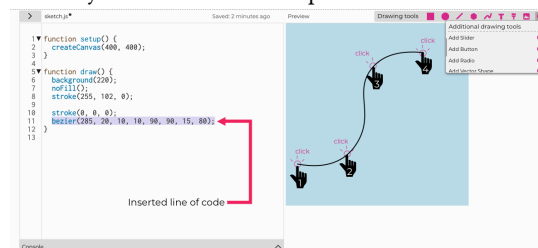
- (11) **In-context Docs** Imagine an editor feature which gives you access to the documentation while you are writing code. This might involve a tooltip that appears on hover (as in the image below) which describes the usage of a particular function. It could also involve showing the description in a dedicated pane on the side. While such features appear in some other editors it did not appear in our editor.



- (a) Do you think this would be useful? (1) Not Very Useful ○, (2) ○, (3) Neither useful nor useless ○, (4) ○, Very Useful ○
 (b) How often do you think you would use this feature? (1) Never ○, (2) ○, (3) Occasionally ○, (4) ○, All the time ○
 (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (12) **Code Snippet Templates** Imagine a feature which allows you to paste in common code snippets from a list. After clicking one of the desired options (such as in the image below) a piece of code achieving that functionality will be added to your code. These snippets could include small structures, such as for-loops, or larger structures, such as particular API uses or classes. This feature sometimes appears in other coding systems, but was not implemented in our system.

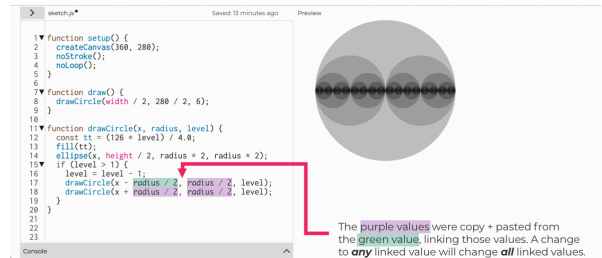


- (a) Do you think this would be useful? (1) Not Very Useful ○, (2) ○, (3) Neither useful nor useless ○, (4) ○, Very Useful ○
 (b) How often do you think you would use this feature? (1) Never ○, (2) ○, (3) Occasionally ○, (4) ○, All the time ○
 (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (13) **Coding by Drawing Tools** Imagine an editor feature which allows you to fill out the arguments to particular functions graphically. For instance, you might indicate to the editor that you are interested in drawing a bezier curve, and then draw each of the vertices in the curve directly on the editor, just as you would in a GUI-based tool like Illustrator, which in turn inserts a corresponding line of bezier command in your code. Unlike in the previous feature, which just inserted code templates, this feature allows you to specify the values of the inserted code with your mouse on the output canvas.

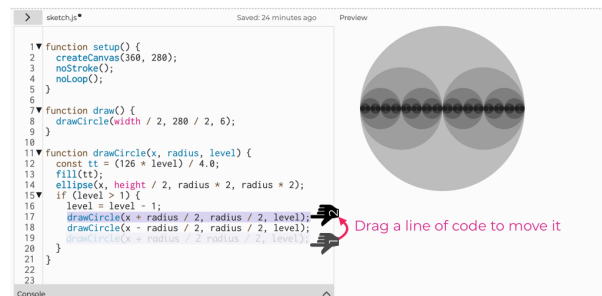


- (a) Do you think this would be useful? (1) Not Very Useful ○, (2) ○, (3) Neither useful nor useless ○, (4) ○, Very Useful ○
 (b) How often do you think you would use this feature? (1) Never ○, (2) ○, (3) Occasionally ○, (4) ○, All the time ○
 (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?

- (14) **Linked Copy-and-Paste** A common abstraction mechanism that we used in class is to create variables or functions rather than copy-pasting chunks of code. While variables and functions are a useful form of computational thinking, there are other ways to approach this task. Imagine a feature which keeps track of your copy-and-pastes: whenever you edit a value you've copied and pasted, all pieces of code which were copied are also changed. This special linked copy-paste can be selectively turned on and off so that you can make edits without changing all copies.



- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (15) **Drag-and-Drop Refactoring** Refactoring is the process of changing the way a piece of code is organized such that the functionality remains the same, but the code is easier to work with. You probably did this during the course by making a variable to capture repeated code or by creating a function to represent some repeated functionality. Imagine a feature which allows you to click and drag values to create arguments, variables, and functions. This might allow you to reorder lines of code by clicking and dragging them, or to highlight a series of repeated values and drag them to automatically create a new variable.

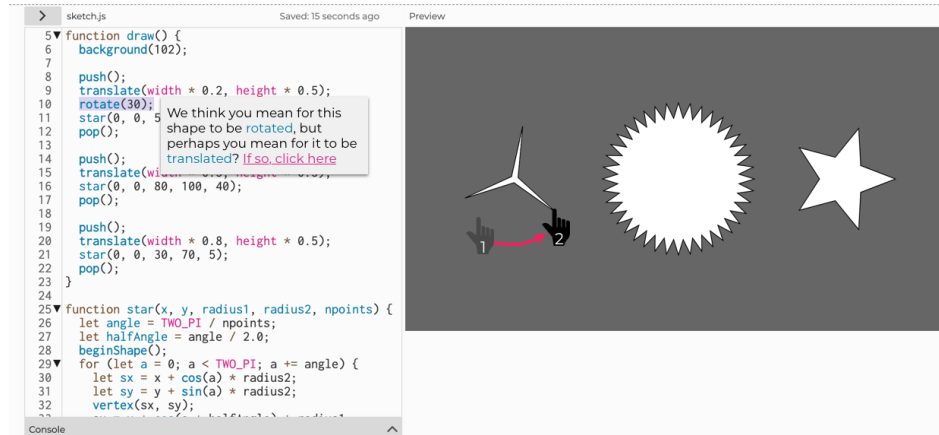


- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?
- (16) **Time Travel Slider** Imagine an editor feature which allows you to go back to earlier points in time of your code's execution. With such a feature you'd press Play, as normal, and watch your code execute. If there was an intermediate state you were curious about you can pause the execution and go back (by dragging a slider) to an earlier state of the canvas. Once you are finished inspecting you can resume execution without rerunning the code. This would allow you to inspect how your code was adding shapes to the canvas over time.



- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor useless ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?

- (17) **Directly Manipulate Shape Attributes on Canvas** Imagine being able to edit the output canvas and have that change the corresponding JavaScript code. This would involve making changes to specific values graphically, such as changing the size of circle or end points of lines by dragging them to a desired position. This differs from the functionality of the previously described "Code by Drawing Tools" feature; that one allowed clicking and dragging to add new shapes to the code, whereas this one allows clicking and dragging to dynamically update and modify existing shapes in the code.



- (a) Do you think this would be useful? (1) Not Very Useful ☐, (2) ☐, (3) Neither useful nor unuseful ☐, (4) ☐, Very Useful ☐
- (b) How often do you think you would use this feature? (1) Never ☐, (2) ☐, (3) Occasionally ☐, (4) ☐, All the time ☐
- (c) Why or why not? Is there any way you would like to modify this feature to make it more useful?

C.4 Page 19: Wrap up

We just worked through a variety of potential editor features individually. To wrap up, we will recap those features and consider them collectively.

- (1) **Feature Review.** Reflect on the features we've just been considering. Which of these hypothetical features we considered are you most excited about? (hover over (?) for details)
 - **Autocomplete** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Linters** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Tidy Code** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Auto-refresh** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Code Folding** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Canvas Ruler** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Number Sliders** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Color Picker** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **p5 State Displays** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Interactive Value Inspector** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **In-context Docs** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Code Snippet Templates** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Coding by Drawing Tools** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Linked Copy-and-Paste** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Drag-and-Drop Refactoring** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Time Travel Slider** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Directly Manipulate Shape Attributes on Canvas** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
- (2) **Course Experience.** Would any of these features have made the course easier for you? Which would you have ignored? Would any of them have helped you learn to program more easily?
- (3) **Challenges.** In the first part of the survey we asked you to think about the assignment that gave you the most difficulty. Thinking about that assignment again, please describe in detail how some of these features may have changed your experience with that particular assignment.
- (4) **Course Project.** Thinking about your course project, do you think any of these editor enhancements would have helped you reach your goals either more quickly or more expressively? Why or why not? Are there any particular editor features that would have made the process easier?
- (5) **Creativity Tools.** If you have experience with creativity tools, such as Illustrator or Photoshop, are there any features you'd like to have as part of a coding editor? If so, what are they and how would they help you accomplish your goals?
- (6) **Suggestions.** Do you have any ideas for editor features (beyond those you might have suggested in other places in this survey)?
- (7) **Miscellaneous.** Is there anything else you would like us to know? Any additional feedback you'd like to share about the editor, or any other technical aspect of the course?

C.5 Page 20: Parting Questions

We've now reached the end of the survey! Thanks for participating! Just two final questions.

- (1) What is your CNetID? (It's the thing at the beginning of your @uchicago.edu email address.)
- (2) What email address would you like your Amazon gift card delivered to?

D SURVEY INSTRUMENT FOR FOLLOW-UP SURVEY — YEAR 2 (*wi22, su22*)

D.1 Page 1: Consent to Participate in a Research Study

Study Number: IRB22-0158

Study Title: Post-Course Survey of Students CS11111 (Winter 2022)

Researcher: Ravi Chugh

Graduate Student: Andrew McNutt

Description: We are researchers at the University of Chicago doing a research study about the usability of editors for creative coding. In CS11111 WI22 we used an in-browser editor that featured a number of enhancements and augmentations to the publicly available p5 editor. We are interested in understanding what editor features are useful for someone learning to code (particularly in the context of a creative coding course) or otherwise making digital art works. To facilitate this inquiry we are asking you to complete a survey. This survey does not include questions about personal or sensitive information. Participation should take about 10-20 minutes. Your participation is voluntary. You are eligible to participate in this survey because you are enrolled in CS11111 WI22, which is the sole criteria for eligibility.

Incentives: In return for your participation, you will receive a small amount of extra credit, roughly equivalent to 1 exercise (1

Risks and Benefits: The risks to your participation in the survey are those associated with basic computer tasks, including boredom, fatigue, or mild stress. The only benefit to you is the learning experience from participating in a research study. The benefit to society is the contribution to scientific knowledge.

Confidentiality: Ultimately, this research may be published and presented at scientific conferences to improve the community's knowledge about editors for creative coding, and may be used to improve the editor used in future iterations of our course. Any reports and presentations about the findings from this study will not include your name or any other information that could identify you. If you decide to withdraw from this study, any data already collected will be destroyed.

Use of Your Research Data: We will never share the data beyond the University of Chicago research team. However, an analysis of the data may be analyzed and published in scientific conference proceedings or journal articles. The free-text responses you provide to any portion of this survey may be quoted in part or in whole in this publication. We will remove any information from the analysis that could identify you before providing the analysis for publication.

Voluntary Participation and Right to Refuse or Withdraw: Participation in this study is voluntary. The decision to participate in this study is entirely up to you. You may refuse to take part in the study at any time without prejudice or penalties to you and will not result in any loss of benefits to which you are otherwise entitled.

Contact Information for Research Questions and Participation: If you have questions or concerns about the study, you can contact the researchers at

Principal Investigator

Ravi Chugh,

Associate Professor

John Crerar Library

University of Chicago

5730 S Ellis Ave

Chicago, IL 60637

Email: rchugh@uchicago.edu

Graduate Student

Andrew McNutt,

PhD student

John Crerar Library

University of Chicago

5730 S Ellis Ave

Chicago, IL 60637

Email: mcnutt@uchicago.edu

If you have any questions about your rights as a participant in this research, feel you have been harmed, or wish to discuss other study-related concerns with someone who is not part of the research team, you can contact the University of Chicago Institutional Review Board (IRB) Office by phone at (773) 702-2915, or by email at sbs-irb@uchicago.edu.

Consent: Participation is voluntary. Refusal to participate or withdrawing from the research will involve no penalty or loss of benefits to which you might otherwise be entitled. By clicking "Agree" below, you confirm that you have read the consent form, are at least 18 years old, and agree to participate in the research. Please print or save a copy of this page for your records.

(1) Full name

(2) GitHub username (same as used for homework submission in this class)

(3) Student Id (the username before your @uchicago.edu email address)

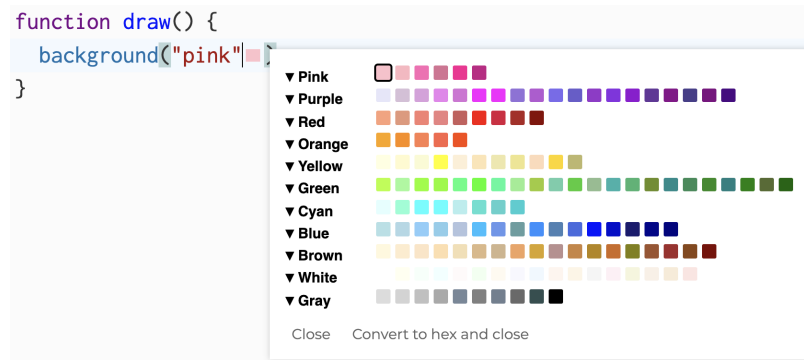
(4) I am a student, 18 years of age or older, I have read and understood this consent form, and give consent to participate in this study.

Yes ☐ no ☐

D.2 Page 2: Feature Questions

The editor we used in the course included several notable features. We are interested in understanding your use and perception of these features.

(1) Feature: Color Pickers



- (a) How often did you use Color Pickers? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Color Pickers are useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Color Pickers? For instance: How did you feel they affected your learning? Is there any way you would modify them to make them more useful?

(2) Feature: Number Pickers

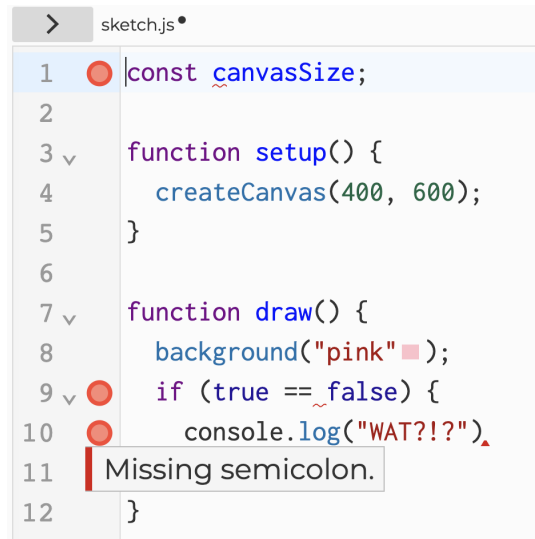
```
function setup() {
  createCanvas(-403 +, -600 +);
}
```

- (a) How often did you use Number Pickers? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Number Pickers are useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Number Pickers? For instance: How did you feel they affected your learning? Is there any way you would modify them to make them more useful?

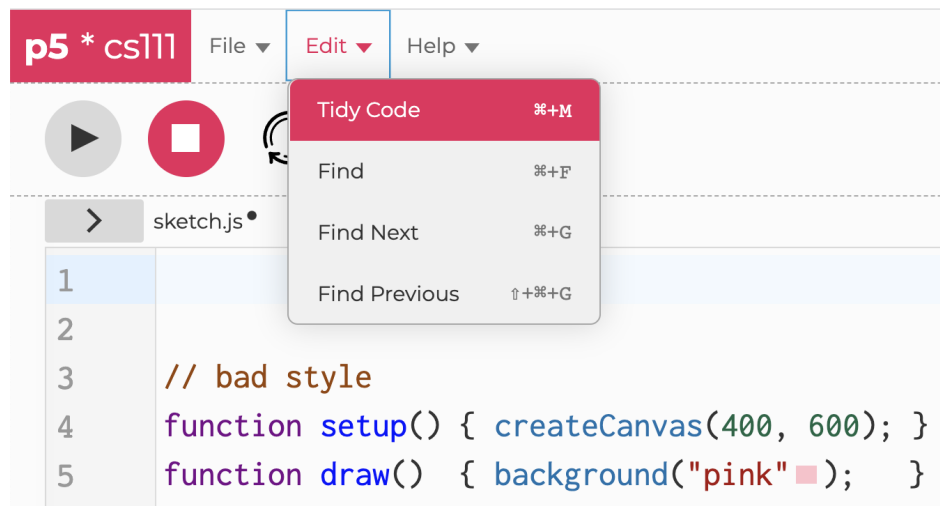
(3) Feature: Number Sliders



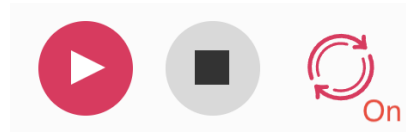
- (a) How often did you use Number Sliders? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Number Sliders are useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Number Sliders? For instance: How did you feel they affected your learning? Is there any way you would modify them to make them more useful?

(4) Feature: Linting

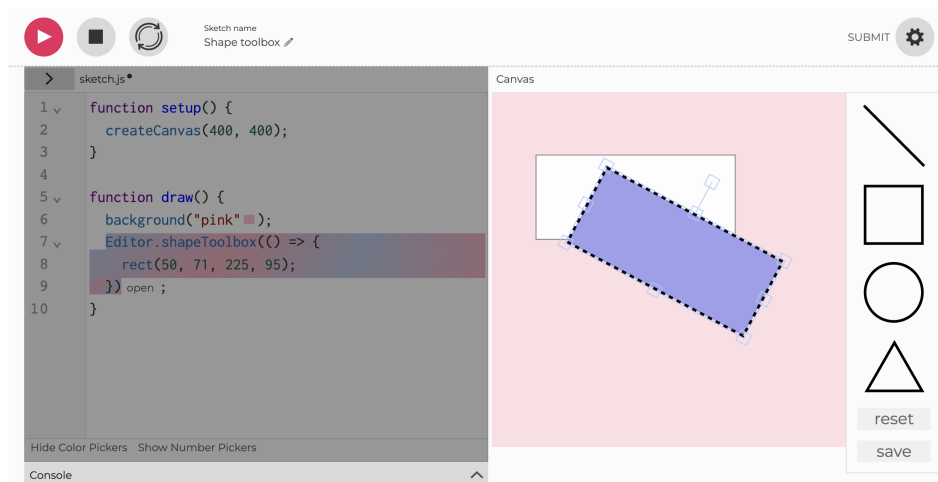
- (a) How often did you use Linting? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Linting is useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Linting? For instance: How did you feel it affected your learning? Is there any way you would modify it to make it more useful?

(5) Feature: Tidy Code

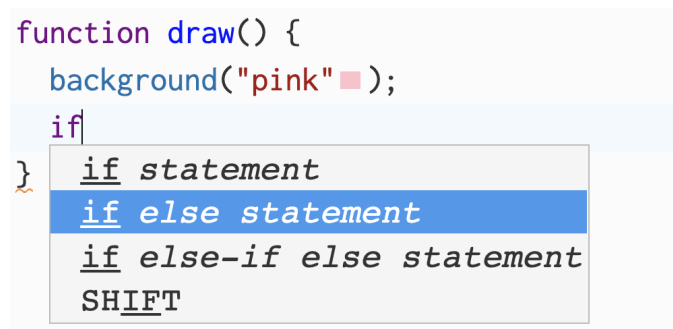
- (a) How often did you use Tidy Code? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Tidy Code is useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Tidy Code? For instance: How did you feel it affected your learning? Is there any way you would modify it to make it more useful?

(6) Feature: Auto-Refresh

- (a) How often did you use Auto-Refresh? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Auto-Refresh is useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Auto-Refresh? For instance: How did you feel it affected your learning? Is there any way you would modify it to make it more useful?

(7) Feature: Shape Toolbox

- (a) How often did you use Shape Toolbox? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Shape Toolbox is useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Shape Toolbox? For instance: How did you feel it affected your learning? Is there any way you would modify it to make it more useful?

(8) Feature: Autocomplete

- (a) How often did you use Autocomplete? ☐ Never, ☐ Once in a while, ☐ Occasionally, ☐ Frequently, ☐ All the time
- (b) Do you think Autocomplete is useful? ☐ Not very useful, ☐ Not useful, ☐ Neither useful nor unuseful, ☐ Useful, ☐ Very Useful
- (c) Do you have any comments about Autocomplete? For instance: How did you feel it affected your learning? Is there any way you would modify it to make it more useful?

D.3 Page: Reflection Questions

Next, we'd like you to reflect on a few additional aspects of the course, how they might be improved, and ways in which the editor might be modified to meet those challenges.

- (1) **Feature Review** Reflect on the features we've just been considering. Which of the features we considered are you most excited about?
 - **Color Pickers** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Number Pickers** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Number Sliders** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Linting** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Tidy Code** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Auto-Refresh** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Shape Toolbox** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
 - **Autocomplete** Very Disinterested ☐, Disinterested ☐, Neutral ☐, Interested ☐, Very Interested ☐
- (2) **Effect on Learning** While it is hard to compare with something you didn't do, how do you think your experience in the course would have been different had these features not been part of the editor? Do you feel you would have learned more or less than you did?
- (3) **Art Tools** Now that you've learned some programming for creative coding, how does that affect your perspective of art making? How might a code editor help or hinder the art making process?
- (4) **Challenges** What aspect of coding or learning to program gave you the most trouble? As a way to help organize your thinking, consider the assignment that you had the most difficulty with. Could the editor have done anything to help you with that?
- (5) **External Tools** It's natural to use other tools as part of the programming process, such as color eye droppers or p5's online documentation. Do you think it would be useful to integrate these tools as part of the editor? What other tools can you imagine wanting to be part of your in-editor coding workflow?
- (6) **Desired Features** What sorts of editor features might have allowed you to be more effective in your coding? What sorts of editor features might have allowed you to be more creative?
- (7) **Miscellaneous** Is there anything else you would like us to know? Any additional feedback you'd like to share about the editor, or any other technical aspect of the course?