Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments

David Weintrop
UChicago STEM Education
University of Chicago
dweintrop@uchicago.edu

Uri Wilensky
Center for Connected Learning and Computer-based Modeling
Northwestern University
uri@northwestern.edu

ABSTRACT
The last ten years have seen a proliferation of introductory programming environments designed for learners across the K-12 spectrum. These environments include visual block-based tools, text-based languages designed for novices, and, increasingly, hybrid environments that blend features of block-based and text-based programming. This paper presents results from a quasi-experimental study investigating the affordances of a hybrid block/text programming environment relative to comparable block-based and textual versions in an introductory high school computer science class. The analysis reveals the hybrid environment demonstrates characteristics of both ancestors while outperforming the block-based and text-based versions in certain dimensions. This paper contributes to our understanding of the design of introductory programming environments and the design challenge of creating and evaluating novel representations for learning.

Author Keywords
Block-based programming; K-12 Education; Programming Environments; Design

ACM Classification Keywords
D.1.7 Visual Programming; D.2.2 Design Tools and Techniques; K.3.2 Computer Science Education

INTRODCUTION
The last ten years has seen a proliferation of introductory programming environments. Led by the popularity of tools like Scratch and Alice, visual block-based programming is increasingly becoming the modality of choice for programming environments designed for novices of all ages. A recent review of coding environments for children included 19 drag-and-drop tools among the 24 environments reviewed for learners under the age of eight, and 28 drag-and-drop environments out of the 47 total reviewed environments [10]. Further, we expect this trend to continue as a growing number of libraries are making it easy to develop environments that incorporate a block-based programming interface [12]. This growth in popularity can be seen both in informal environments as well as in classrooms where a growing number of curricula, like Exploring Computer Science [29] and the Beauty and Joy of Computing [13] utilize block-based programming.

Until recently, block-based and text-based programming environments have been distinct. An environment used either one modality or the other. As a result, learners trying to migrate from a block-based environment to a more conventional text-based programming language had few environmental supports to facilitate the transition. Multiple approaches have been developed to mitigate this transition cost. One approach is pedagogical, relying on teachers to assist learners in moving between modalities. An alternative approach sees this challenge as a design question, asking: how can we design programming environments to scaffold learners in moving from block-based to text-based programming? Two strategies have been employed in response to the design solution to this challenge: dual modality environment that support both block-based and text-based programming and hybrid block/text environments that try and blend features of both modality to create a best-of-both world tool. This paper explores one implementation of a hybrid block/text programming environment and answers the following research question:

How does a hybrid block/text programming environment perform compared to isomorphic block-based and text-based environments with respect to the conceptual learning of programming concepts and learners' attitudes and perceptions of the field of computer science?

To answer this question, we conducted a 15 week, quasi-experimental study in which high school students in an introductory computer science classroom used either a block-based, text-based, or hybrid block/text programming environment. Data were collected to understand how these three programming modalities impacted learners' experiences. This paper presents the results of this study, specifically focusing on how students in the hybrid condition performed relative to peers in the block-based and text-based environments.

IDC '17, June 27-30, 2017, Stanford, CA, USA

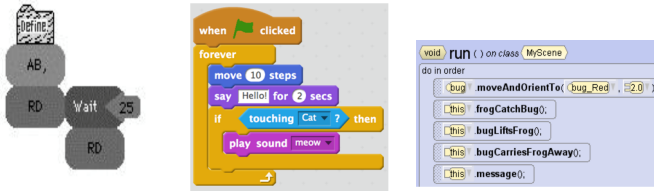
ACM 978-1-4503-4921-5/17/06…$15.00
http://dx.doi.org/10.1145/3078072.3079715

This paper begins with a review of prior work, specifically looking at empirical and design work related to block-based programming environments. Next, we introduce Pencil.cc, the multi-modality programming environment created to study the stated research question. A methods section follows, which details the study design and provides information on the setting and participants. Findings from the study are then presented, including results on outcomes of attitudinal surveys and content assessments. The paper concludes with a discussion section that presents implications of this work.

PRIOR WORK

Block-based programming

Block-based programming (Figure 1) is a programming modality that is becoming increasingly widespread in introductory computing contexts. We use the term modality to capture the representational infrastructure used to depict the program, as well as the various forms of interactions supported by the representation and its presentation. Block-based programming environments leverage a programming-primitive-as-puzzle-piece metaphor that provides visual cues to the user about how and where commands can be used. Composing programs in these environments takes the form of dragging blocks into a canvas and snapping them together to form scripts. If two blocks cannot be joined to form a valid syntactic statement, the environment prevents them from snapping together, thus preventing syntax errors but retaining the practice of assembling programs instruction-by-instruction. Along with using block shape to denote usage, there are other visual cues to help programmers, including color coding by conceptual use and nesting of blocks to denote scope [20,32]. Early versions of this interlocking blocks approach include LogoBlocks [4] and BridgeTalk [5] which helped formulate the programming approach which has since grown to be used in dozens of applications across a variety of domains [10]. The growing ecosystem of block-based tools speaks to the need for more critical research around the affordances and drawbacks of the modality [30,38].



(A) LogoBlocks (B) Scratch (C) Alice

Figure 1. Three examples of block-based programming tools.

Bridging Block-based to Text-based Programming

With the rise in popularity of the block-based approach to programming, a question of growing importance is how well these tools prepare students for future, text-based programming languages. An emerging body of scholarship is exploring the question of transfer of programming knowledge between block-based and text-based programming. For example, as Alice is often used in undergraduate computer science classrooms, the transition from Alice to Java is an active area of research showing mixed results. A number of studies have reported student difficulties in making the transition [7,14,26], with one study stating: "Students do not seem to naturally make a strong connection between the formal coding process and what they are doing with Alice" [25:213]. In contrast, other researchers have found Alice to be an effective way to introduce learners to programming and have had success transition learners to Java [8,9].

A few studies have been conducted looking at the transition from Scratch to other text-based programming languages. While many of these report only anecdotal evidence, Armoni, Merrbaum-Salant & Ben-Ari [1] conducted a longitudinal study looking at whether students who had taken Scratch programming classes in middle school performed better in a high school, text-based programming course. Overall, the researchers found little quantitative difference in performance on assessments between students who had previous worked with Scratch and those who had not, but were able to find some areas where the Scratch students out-performed their peers (specifically on the concept of looping). Additionally, the authors found qualitative differences between the two populations, with students who had prior Scratch experience reporting higher levels of motivation and self-efficacy. A quasi-experimental study investigating students transition from block-based languages to Java found block-based programming, without pedagogical support, provided relatively little benefit for novices in the transition to a text-based language [33].

Scaffolding the Block-to-Text Transition

Block-to-Text Transformations

A growing number of tools are being designed to address the block-to-text gap, either as new stand-alone tools or add-ons to existing tools. One direction programming designers have pursued is giving users the ability to convert block-based programs into equivalent text-based scripts. This includes environments like PicoBlocks, TurtleArt, and the App Inventor Java Bridge [40]. Other tools provide native language translation, for example, the Blockly toolkit comes with built-in language generators that allow you to convert graphical scripts to equivalent JavaScript, Python, or XML files [12]. Additionally, Blockly is architected in such a way as to make it easy to add additional generators to the library making it extensible for future block-to-text transformations. The DrawBridge project is noteworthy in its effort to bridge block-based and text-based programming by introducing pen-and-paper drawing and program-by-demonstration features into its larger pedagogical strategy [31]. Game authoring has also been used as a context to motivate block-to-text programming as demonstrated by the Flip project [17].

Dual Modality Programming Environments

While the environments listed in the previous section provide a one-way transition from a block-based interface to the textual form, a growing number of tools are providing bi-directional support. Pencil Code [3] provides a two-way transition between blocks and CoffeeScript, JavaScript, and HTML, while tools have also been built for Java [21], Python [2], and Grace [16]. In these environments, the user can move back and forth between block-based and text-based representations of their program. Matsuzawa et al. [21] taught an introductory programming course using an environment that allowed users to program with either a block-based or text-based Java interface and found that over the course of the semester, students systematically transitioned from blocks to text on their own. These findings were replicated in another study that investigated the cause of block-to-text transitions, and found students often used the block-based modality when adding commands to their programs for the first time, revealing one specific way dual modality environments are able to scaffold novice programmers [34].

Hybrid environments

The last section of this literature review looks at the types of environments that are the focus of this work: hybrid block/text environments. These environments blend features of block-based interfaces with conventional text-based editors to create a new modality distinct from both blocks and text. One recent example of this type of environment is Greenfoot's Frame-based Editor. As its creators explain, frame-based editing "maintains some of the graphical representation advantages, discoverability and error avoidance of blocks while providing the flexibility, keyboard-entry capabilities, and readability of text" [19:29–30]. A central design goal of the Frame-based editing approach is to keep the atomic unit of operation a valid node in the program's abstract syntax tree (i.e. you add/edit/delete full commands, akin to working in blocks), but that manipulation of these nodes can be completed with the keyboard and the program presentation retains the visual characteristics of a text-based program. Early analysis of frame-based editing shows the promise of this specific hybrid approach [27]. A second example of a hybrid block/text programming tool can be seen with GP, a tool inspired by Scratch that seeks to address the drawbacks of block-based languages as programs become larger and more complicated by incorporated text layouts and keyboard-driven compositional mechanisms [22]. A third hybrid block/text environment is Pencil.cc, which is the focus of this paper and described in detail below.

MEET PENCIL.CC

Pencil.cc is a customized version of the Pencil Code environment [3] that supports block-based, text-based, and hybrid block/text programming, but is designed such that a given user only has access to one of these programming modalities. Pencil.cc's blocks interface (Figure 2a) features many of the defining features of block-based tools, including the drag-and-drop programming mechanism, a palette of blocks for the user to choose from, and visual cues on how and where blocks can be used. The text version of Pencil.cc presents users with a text editor that includes basic programming supports like highlighting, automatic formatting, and syntax checking.

The hybrid form of Pencil.cc retains the block-palette and the ability to drag-and-drop commands into a program, but replaces the blocks canvas with a text editor. When a user drags a block from the palette onto the text canvas, the block turns into the textual equivalent and is inserted into the program in a syntactically valid way. Thus, the hybrid interface supports both drag-and-drop and keyboard-driven composition. Figure 2b and 2c shows Pencil.cc's hybrid interface. This hybrid approach was informed by earlier findings on the design of block-based tools, including features that learners found to be useful as well as perceived drawbacks of block-based programming tools [36]. Specifically, this hybrid design retains features including a browsable blocks library, drag-and-drop composition, and pre-fabricated commands. At the same time, the text-editor interface tries to address some of the drawbacks identified by learners in block-based tools, such as perceived inauthenticity and issues with block-based environment being less powerful or slower than text-based alternatives.

In Pencil Code, users are free to move back-and-forth between the blocks and text modalities. With Pencil.cc, this

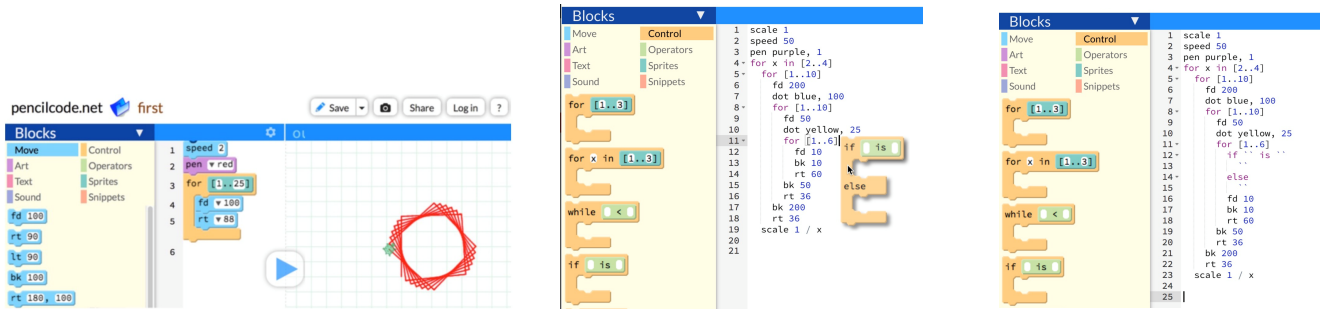


Figure 2. Pencil.cc's interfaces. (a) shows the blocks interface while (b) and (c) show Pencil.cc's hybrid block/text interface. The middle image (b) shows how learners can drag-drop blocks into the text editor; the right image (b) shows the results of this action.

ability has been removed, instead users see either the block-based interface, the text-based interface, or the hybrid interface. Aside from the programming modality, all other features of the three modes of Pencil.cc are the same, making the three environments isomorphic, meaning the capabilities and expressive power are equivalent across modalities; anything that can be done in one mode can also be achieved in the other two. Pencil.cc was chosen as the environment for this study due to its ability to support all three modalities and because it shares many characteristics with widely used introductory environments such as visual execution of programs and accessible language primitives.

METHODS

In this section, we present details of the study conducted to understand outcomes of hybrid block/text programming environments relative to isomorphic block-based and text alternatives. We begin by detailing the design of the study, then present information about the participants and setting.

Study Design and Data Collection Strategy

This study uses a quasi-experimental setup with three high school introductory programming classes. The study follows each classroom for the first 15 weeks of a yearlong introduction to programming course. The 15 weeks were broken down into two phases, a five-week introductory phase, which was immediately followed by a 10-week Java phase. During the introductory phase, each of the three classes used a different Pencil.cc modality (block-based, text-based, or hybrid). This means students only saw one of the three modalities (so a student in the blocks condition, never saw the text or hybrid interface). These three classes constituted the three conditions of the study: Blocks, Text, and Hybrid. Starting in week six, all three classes transition to Java and followed the same curriculum in the same programming environment for the remaining 10 weeks.

The study began on the first day of school with students in the three classes taking Pre attitudinal surveys and content assessments. The Commutative Assessment [37] was used for the content assessment and a modified version of the attitudinal survey from the Georgia Computes project [6] was used for the attitudinal survey with questions added related to students' perceptions of the different modalities. The attitudinal survey and content assessments were administered at three points during the fifteen-week study: the first day of the study (Pre), at the conclusion of the introductory curriculum at the end of week 5 (Mid), and at the conclusion of the study in week 15 (Post). The surveys were administered online during class time on consecutive days. The attitudinal survey took students around 20 minutes to complete and the content assessment took close to 25 minutes. The assessments were given on the same day across all three classes.

The five-week curriculum for the introductory phase of the study was based on the Beauty and Joy of Computing course [13], along with an assortment of other introductory computing activities grounded in the Constructionist programming tradition and the Logo programming language [15,23]. An emphasis of this design was to allow students creative freedom in each assignment. Over the course of the five-week introductory phase, four major conceptual topics were covered: variables, conditional logic, looping logic, and procedures. Throughout the activities, care was taken to blend visually executing programs (like traditional Logo graphics drawing activities) and text-based activities (like asking for and responding to input from the keyboard). The curriculum was largely designed by the first author but the classroom teacher contributed ideas and customized the activities while teaching them. Full versions of the Commutative Assessment, the attitudinal survey, and the five-week curriculum can be found in the appendices of [33].

Setting and Participants

This study was conducted at a large, urban, public high school in a Midwestern city in the United States of America, serving almost 4,000 students. The school is a selective enrollment institution, meaning students have to take an exam and qualify to attend. In this school district, students are selected based on their performance on the admissions test relative to other students from their school (as opposed to all other applicants). As a result, students attend this school from across the city and there is an equal representation of students from under-resourced schools as from schools in more affluent parts of the city. A majority of the students in the school (58.6%) come from economically disadvantaged households.

The computer science course used for the study is an elective class but historically has attracted students from a variety of racial background and been taken by both male and female students. A total of 90 students participated in the study. The self-reported racial breakdown of the participants was: 41% White, 27% Hispanic, 11% Asian, 11% Multiracial, and 10% Black. The classes comprised of students across all four years of high school, with a reported mean age of 17.1 (SD = 1.1 years). The three classes in the study were comprised of 15 female students and 75 male students. This gender disparity is problematic, but as recruitment for the courses was out of the control of the researchers, there was little that could be done to address this. Of the students participating in the study, 47% speak a language other than English in their households.

The experiment was conducted in an existing Introduction to Programming course. Historically, the class spent the entire year teaching students the Java programming language. To accommodate the study, Java instruction began in the sixth week of school, after the conclusion of the five-week curriculum previously discussed. Each class had 30 students and each student was assigned a laptop which they used every class. Students sat in individual desks that were on wheels, that allowed them to move their desks around. The same teacher taught all three sections of the course, allowing us to control for teacher effects.

FINDINGS

The findings section is broken down into two main sections. First, we look at student performance on the content assessment by modality, then we look at attitudinal and perceptual outcomes from this study. While there is much that can be said about the data presented below, the analysis and discussion in the paper is focuses specifically on the Hybrid condition and understanding how it fared relative to the Blocks and Text conditions. A comparison of the Blocks condition to the Text condition can be found in [35]. Throughout this section, only significant statistics are presented; absent values were not significant.

Content Assessment

We begin our findings section by looking at how students performed on the content assessment at the three time points based on the version of Pencil.cc they used during the introductory five weeks of the study. At the outset of the study, the Hybrid condition was not statistically different than either the Blocks class or the Text condition. This means that the three classes are not different from each other with respect to their incoming programming knowledge. Figure 3 shows cumulative scores for students across the three conditions on the Pre, Mid, and Post Commutative Assessment administrations[1].

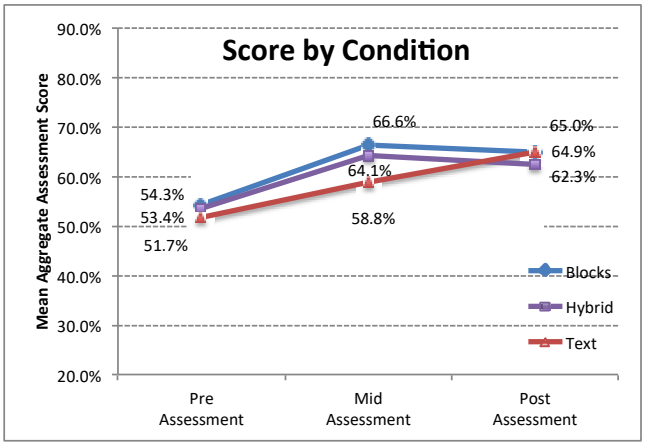


Figure 3. Content scores by condition over time.

The positive slope for all three conditions between the Pre and Mid assessments means that, in aggregate, students in all three classes performed better on the Mid survey than they did on the Pre. Given that this was an introductory class it is not surprising, but still noteworthy and an encouraging sign given that these three conditions cover almost the entirety of the modalities used to introduce learners to programming. For the Hybrid condition, the improvement on test scores from the Pre to the Mid was significant, $t(26) = 6.65$, $p < .001$, $d = .65$. This shows that students in the Hybrid condition did better on the assessment after working in the Hybrid interface of Pencil.cc for five week.

On the Mid assessment, students' performance in the Hybrid condition was significantly different from the Text condition, albeit with a moderate effect size ($t(52) = 2.03$, p-value = 0.04, $d = 0.58$), and not different from the Blocks condition. On the Post assessment, the three conditions converge, resulting in the Hybrid condition not being statistically different from the other two conditions. Looking over the three time points in Figure 3 shows a pattern of students in the Hybrid condition performing more similarly to the Blocks condition than to the Text condition over the course of the 15-week study. This trend can be seen by the fact that the change in the score between the Mid and Post administrations for the Hybrid condition was significantly different than the learning gains observed for the Text condition ($t(48) = 2.88$, $p = .01$, $d = .81$), but not the Blocks condition.

Condition by Concept

This section investigates how students in the Hybrid condition performed compared to the Blocks and Text conditions with respect to specific programming concepts. The Commutative Assessment covers six content areas: Algorithms, Code Comprehension, Conditional Logic, Functions, Iterative Logic, and Variables. If we look at how students performed on each of the six content areas on the Mid assessment, we find that students in the Hybrid condition scored between their Block-based and Text-based peers on half of them (Code Comprehension, Conditional Logic, and Functions,). In two concept areas (Iterative logic and Algorithms) the Hybrid condition scored the highest, while they scored the lowest on Variables questions. None of these differences are statistically significant at the $p < .05$ level. This means with this data, we cannot state that students learned specific concepts better in the Hybrid version of Pencil.cc compared to either the Blocks or Text modalities. Where we do see differences emerge is in students' perceived ease-of-use of these concepts. On the Mid attitudinal assessment, students were asked: "How easy was it to use ___ in Pencil.cc?", where ___ was replaced with the concept under question. The mean responses to the 7-point Likert questions are shown in Figure 4. The higher the score, the easier a student thought it was to use the given concept.

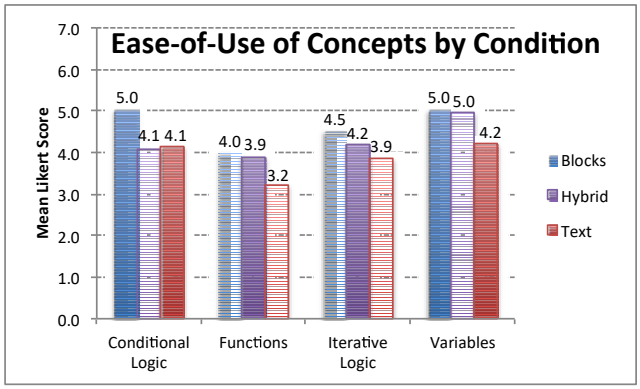


Figure 4. Student reported ease of using concepts in Pencil.cc.

[1] Please note the line charts in this paper to not start at 0.

Two of the four concept areas show a significant difference between Hybrid and another modality. interestingly, in one case the Hybrid condition is similar to the Text condition, and in the other, the Hybrid is similar to Blocks. When asked how easy students found it to use conditional logic, students in the Hybrid condition reported a scores that was significantly lower than their Block-based peers ($U = 212$, $p = .01$)[2], and no different from the Text condition. This means, first, the Hybrid condition found conditional statements significantly harder to use than students in the Blocks condition. Second, students working in the Hybrid condition found conditional logic as difficult to use as students in the Text condition. This suggests the drag-and-drop mechanism provided by the Hybrid interface was not successful in providing the same amount of scaffolding as the full Blocks interface provided.

Running the same analysis on the responses to the question of how easy are variables to use, we find the opposite trend. The Hybrid condition is significantly different from the Text condition ($U = 248.5$, $p = .05$) but no different from the Blocks condition. Here, the drag-and-drop composition mechanism provided by the Hybrid programming environment was successful in making students feel like variables were as easy to use as the Blocks condition.

Looking across the data, one explanation for ease-of-use in the Hybrid condition is that the simpler the post-addition modification is, the easier the construct to be used is perceived. Whereas the use of a conditional statement in the Hybrid condition often included non-trivial post addition edits (i.e. defining the test and the commands to be followed after the test is evaluated, as can be seen by the ``s that need to be replaced by the user in Figure 2c), when adding variables to a program via drag-and-drop, only very straightforward edits are necessary, like entering the name or changing the value. In this way, the addition of variable blocks to a program in the Hybrid condition presented a template that required simple replacements. This finding is useful in that it can provide direction in terms of the design of new hybrid environments and how the addition of different programming constructs is accomplished. One other important thing to note from this analysis is the revelation that the effect of a given hybrid block/text implementation does not necessarily have a uniform benefit across the various constructs that constitute a programming language. This highlights the need for nuanced analysis beyond just aggregated assessment scores.

Attitudes and Authenticity

The attitudinal assessment given to students included a series of 10-point Likert scale questions. In this section, we present data looking at how students in the Hybrid condition responded relative to their block-based and text-based peers with respect to confidence, enjoyment, perceived authenticity of the programming environment, and interest in pursuing the field of computer science.

Confidence in Programming Ability

The first attitudinal dimension we present is students' perceived confidence in their own programming ability. Student responses to the following two Likert scale statements: I will be good at programming (or I am good at programming on the Post survey) and I will do well in this course, which were then averaged together. These questions show an acceptable level of correlation, having Cronbach's α scores of .79 (Pre), .80 (Mid), and .88 (Post) on the surveys, all of which are near the .8 threshold commonly used to denote an acceptable level of reliability.

Looking at how students responded on the confidence questions at the Midpoint of the study (i.e. after working in Pencil.cc for five weeks) we find no difference between the Hybrid condition and either the Blocks condition or the Text condition, as all three groups reported roughly the same average level of confidence. However, after students transitioned to Java, a difference between the conditions emerges. At the conclusion of the study, the Hybrid students reported confidence level was $M = 8.3$ ($SD = 1.9$), compared to Blocks students' average of $M = 7.4$ ($SD = 2.3$) and Text students' $M = 8.3$ ($SD = 1.1$). In other words, the Hybrid students and Text students had a higher level of confidence than the Blocks condition. The Hybrid conditions average confidence score is only significantly higher than the Blocks condition at the .10 level ($U = 274$, $p = .10$) and not significantly different than the Text condition. Taken together, this shows, with respect to confidence, at the conclusion of the study, the Hybrid condition was more similar to the Blocks students and their higher reported level of confidence compared to students who had worked in the Text only interface.

Enjoyment of Programming

The second attitudinal dimension is whether or not students' enjoyment of programming differed based on the modality they used. To calculate a measure of enjoyment, responses to the following three Likert statements from the Pre, Mid, and Post surveys were averaged: I like programming, Programming is Fun, and I am excited about this course. These three questions were found to reliably report the same underlying disposition at all three time points (Pre Cronbach's $\alpha = .84$, Mid Cronbach's $\alpha = .84$, Post Cronbach's $\alpha = .89$). Comparing students' scores at the Mid and Post time points, the Hybrid condition was not significantly different from either the Blocks condition or the Text condition. The conclusion here is that the hybrid version of Pencil.cc did not affect students' enjoyment of the act of programming. This is possibly because there are other more salient features of a programming experience that shape enjoyment (like the activities, setting, or runtime environment) or that modality does not affect enjoyment.

[2] When comparing the conditions to each other, we use a Wilcoxon Rank Sum test (reported as a U statistic). This test is appropriate as the two samples are independent and the underlying data is non-parametric and ordinal in nature.

Programming is Hard

The attitudinal survey included the Likert statement: Programming is Hard. On both the Mid survey and the Post survey, students in the Hybrid condition reported scores significantly lower than the Blocks condition (Mid: U = 488, p = .01; Post: U = 481, p = .04) and not different than the Text condition. This mean, that with respect to perceived difficulty of programming, the Hybrid condition was viewed as the same as the Text condition, suggesting that the Hybrid interface did not succeed in making students feel like programming was easier.

Interest in Future Computer Science

The last attitudinal category looks at whether or not the modality used in the introductory programming environment affected students' interest in future computer science courses. More specifically, students were asked to respond to the statement: I plan to take more computer science courses after this one. Figure 5 shows the average response for students grouped by condition.

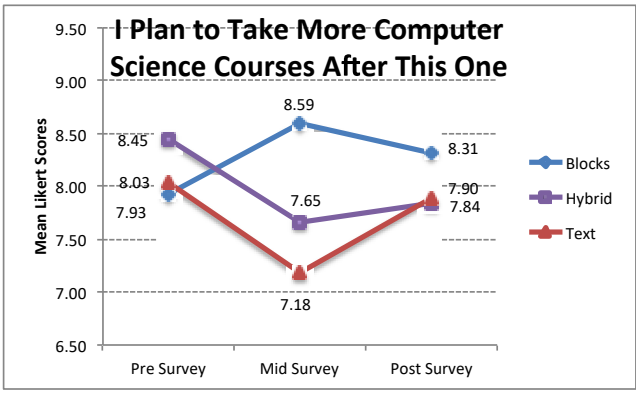


Figure 5. Average responses to the Likert statement: I plan to take more computer science courses after this one.

Like with the other attitudinal findings, these data show student responses in the Hybrid condition being more similar to the Text condition than the Blocks condition. On the Mid survey, the Hybrid condition is significantly different from the Blocks condition (U = 451, p = .05) but not different from the Text condition. On the Post survey, there is no statistically significant difference between the Hybrid group and either the Blocks condition as the reported attitudes converge. Like with the overall scores shown in Figure 3, the three conditions diverge at the midpoint of the study, then converge at its conclusion. However, unlike the content assessments, when asked about interest in future computer science courses, the Hybrid condition's path followed that of the Text condition, rather than the Blocks condition. This is not the desired outcome as the Blocks condition reported a higher level of interest in future computer science learning opportunities.

Perceived Authenticity

One drawback identified in using block-based programming environments with high school-aged learners is the perceived lack of authenticity and a recognized difference between what it looks like to program in block-based languages versus conventional text-based languages [36]. On the Mid and Post attitudinal surveys, students were asked if what they did in the first five weeks of the course was similar to what "real programmers" do. Responses were given on a ten-point Likert scale, with a higher score meaning students agreed more strongly. Figure 6 shows student responses by condition to this prompt.

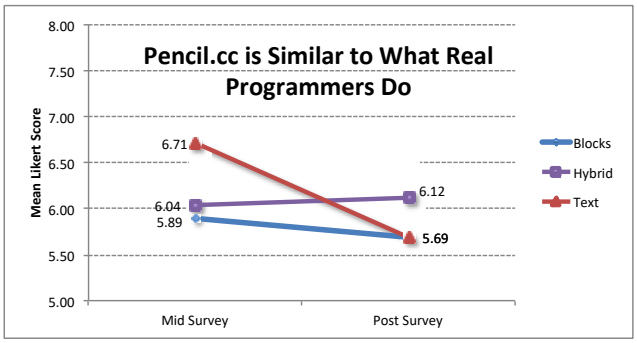


Figure 6. Student responses to the authenticity prompt: Pencil.cc is similar to what real programmers do.

After working in Pencil.cc for five weeks, students in the Text condition viewed the environment as the most similar to what real programmers do, while the Blocks and Hybrid conditions had similar views of the authenticity, which were lower than the Text condition. This suggests that this specific Hybrid implementation was not successful in addressing the authenticity concern previously identified in the literature. However, after spending 10 weeks programming in Java, students' perceptions of the authenticity of Pencil.cc shifted. In both the Text and Blocks conditions, students viewed Pencil.cc as less authentic than they had initially, while the perceived authenticity of Hybrid improved, meaning students thought the Hybrid Pencil.cc environment was slightly more authentic than they had initially. This positive change for the Hybrid condition is significantly different from the negative change for the Text condition (U = 437.5, p = .01) but not the Blocks condition. The Hybrid condition was the only modality with a positive slope, suggesting that after working in Java, students did not see the Hybrid condition as less authentic than they had before. One possible explanation for the Hybrid condition's different outcome stems from the fact that only in that condition do students interact with more than one modality (graphical blocks and text side-by-side), thus possibly suggesting to students that programming is not a uniform activity, but instead, that the act of programming and programming languages and environments can take many shapes and rely on many modalities, interfaces, and technologies.

DISCUSSION

The research question this paper addresses is investigating whether or not it is possible to design a "best-of-both worlds" introductory programming modality that includes features of both block-based and text-based editors. In this

section, we summarize the findings, first looking at how the three conditions fared relative to each other during the 5-week comparative portion of the study, then looking at if and how those differences persisted as students transition to Java. Finally, we step back from this specific study and discuss the larger implications of these findings.

The Hybrid Condition: The First Five Weeks

Over the first five weeks of the study, students in the Hybrid condition showed attitudinal changes that were similar to those observed in the Text condition: little change with respect to confidence or enjoyment of programming and a decrease in interest in taking future computer science courses. When asked about how the introductory environment compared to what real programmers do, the Hybrid students gave responses similar to the Blocks students, which were lower than their Text-based peers. Together, these findings show that the Hybrid environment was not particularly successful with respect to cultivating positive attitudes relative to the other modalities. On the Mid administration of the Commutative Assessment, the Hybrid condition scored between the Blocks and Text students overall, but was closer to the higher score of the Blocks condition than to the Text condition. When asked about the perceived ease-of-use of various programming constructs, the Hybrid students reported that variables and functions were as easy to use as the students in the Blocks condition, but found conditional logic more difficult to use, like the Text condition. Taken together, these results highlight how the Hybrid condition has successfully blended the Blocks and Text modalities, however, was not universally successful in achieving a best-of-both-world outcome, as there were instances where the Hybrid condition was closer to the underperforming modality. There were also a few places where the Hybrid condition is distinct from the other two modalities, suggesting that, along some dimensions, the Hybrid modality is more than the sum of the other two modalities.

The Hybrid Condition: Transitioning to Java

Whereas the Hybrid condition did not seem to produce positive outcomes with respect to attitudinal measures during the first five weeks, things changed after the transition to Java. When asked to reflect on their time in the introductory modality, students in the Hybrid condition reported their time in Pencil.cc as being the most similar to what real programmers do. In the four other attitudinal categories evaluated, the Hybrid condition saw relatively little change, having three categories showing slight increases (enjoyment, perceived difficulty, and interest) and a minor decrease the last (confidence). Students' scores on the Commutative Assessment decreased slightly after working in Java for ten weeks, suggesting the modality was not an outlier with respect to preparation for future text-based learning in a different language.

Modality is Malleable

One of the contributions of this work is providing evidence that modality is malleable and showing that its design is consequential. Unlike many other domains, the representations novices use when being introduced to computer science are not set, but instead are part of an active area of design research. Designing novice programming tools is one instantiation of the larger intellectual pursuit of creating new ways of expressing ideas and interacting with representational systems [18,24,39]. Programming languages provide an especially rich context for this work due to what Papert [23] called the *Protean* nature of computers. Computers provide the ability to introduce layers of abstraction between the way a representational infrastructure is presented to the user and the form those instructions must take in order to be executed by the machine upon which they reside. From this relatively blank canvas, a vast design space emerges for the creation of new modalities. Looking at the three modalities used for this work, we can start to see the various dimensions along which computationally situated modalities can be defined. Visual rendering (color, shape, location on the screen, etc.) serves as a first dimension along which modality design can explore. Likewise, the novelty of the modality and if, how, and when other representational systems are incorporated can differ. For example, when comparing Scratch Jr. [11] and its use of glyphs to Scratch [28] where natural language expressions are used, to Pencil Code [3] which provides visual supports on top of programming keywords, and finally to Logo [23], which is a fully text language, we see a spectrum of how textual language can exist within a programming interface. Figure 7 shows the turn right command as it is represented across these four tools.

(a) (b) (c) (d)

Figure 7. The command to turn right in four modalities: (a) Scratch Jr., (b) Scratch, (c) Pencil Code, and (d) Logo.

These four representations highlight visual differences between modalities. Other design dimensions to be explored in the creation of new modalities include temporal differences (have the representation change over time), auditory components, responsive or interactive modalities where the representation changes based on state or input, or creating tangible dimensions of the modality that are independent or live alongside virtual elements.

As our definition of modality includes interactions, thus designing modalities extends beyond just the visual depiction of the representation. Consideration of modality design also includes various interaction capabilities that influence the mechanics of interaction with and use of the modality. In this work, the difference between dragging-and-dropping blocks on the canvas versus typing commands in character-by-character with the keyboard highlight difference in interaction.

A final important dimension to discuss in terms of the malleability of modalities is to point out that a user need not be pinned to a specific modality. The learners in this study were held to a specific modality for the purpose of the research, not because of design constraints. The growing number of dual modality environments discussed earlier in this paper highlight the potential of this approach. The ability to provide multiple modalities within the same environment further opens the set of possibilities to designers of learning environments and computational tools more broadly. A strength of this approach is that it gives agency to the learner to decide not just how to use the various features of a single modality, but also to choose the modality they want to use.

CONCLUSION

As the presence of programming and computer science in K-12 classrooms grows, it is important that we take time and care to develop and evaluate the environments learners are using. Introductory programming environments that blend features of the block-based and textual modalities are one design approach that is gaining popularity. In this paper, we present one such implementation of a hybrid programming environment and compare it to isomorphic block-based and text-based alternatives to understand the impacts of this approach. In doing so, we have identified places where this hybrid design is effective and other dimensions where more design work needs to be done to better support learners. The hybrid design used in this study shows the potential for this line of work towards the greater goal of developing effective and accessible programming environments. Looking beyond programming environments, the larger contribution of this work is showing how the design choices made in the creation of new modalities and learning environments can produce outcomes similar to either of the source modalities used, as well as unique outcomes distinct from the designs that served as its inspiration. The hope for this work is that by attending to modality and viewing it as a design challenge, we can improve existing and create new ways for learners to engage with the powerful ideas of computing that surround them.

REFERENCES

1. M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari. 2015. From Scratch to “Real” Programming. *ACM Transactions on Computing Education (TOCE)* 14, 4: 25:1-15.
2. A. C. Bart, E. Tilevich, C. A. Shaffer, and D. Kafura. 2015. From interest to usefulness with BlockPy, a block-based, educational environment. In *Blocks and Beyond Workshop, 2015 IEEE*, 87–89.
3. D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens. 2015. Pencil Code: Block Code for a Text World. In *Proceedings of the 14th International Conference on Interaction Design and Children* (IDC ’15), 445–448.
4. A. Begel. 1996. LogoBlocks: A graphical programming language for interacting with the world. Electrical Engineering and Computer Science Department. MIT, Cambridge, MA.
5. J. Bonar and B. W. Liffick. 1987. A visual programming language for novices. In *Principles of Visual Programming Systems*, S. K Chang (ed.). Prentice-Hall, Inc.
6. A. Bruckman, M. Biggers, B. Ericson, T. McKlin, J. Dimond, B. DiSalvo, M. Hewner, L. Ni, and S. Yardi. 2009. Georgia computes!: Improving the computing education pipeline. In *ACM SIGCSE Bulletin*, 86–90.
7. D. C. Cliburn. 2008. Student opinions of Alice in CS1. In *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*, T3B–1.
8. W. Dann, S. Cooper, and B. Ericson. 2009. *Exploring Wonderland: Java Programming Using Alice and Media Computation*. Prentice Hall Press.
9. W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. 2012. Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 141–146.
10. C. Duncan, T. Bell, and S. Tanimoto. 2014. Should Your 8-year-old Learn Coding? In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (WiPSCE ’14), 60–69.
11. L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bontá, and M. Resnick. 2013. Designing ScratchJr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*, 1–10.
12. N. Fraser. 2013. *Blockly*. Google, https://developers.google.com/blockly/.
13. D. Garcia, B. Harvey, and T. Barnes. 2015. The Beauty and Joy of Computing. *ACM Inroads* 6, 4: 71–79.
14. R. Garlick and E. C. Cankaya. 2010. Using Alice in CS1: A quantitative experiment. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, 165–168.
15. B. Harvey. 1997. *Computer science logo style: Beyond programming*. The MIT Press.
16. M. Homer and J. Noble. 2014. Combining Tiled and Textual Views of Code. In *IEEE Working Conference on Software Visualisation (VISSOFT)*, 1–10.
17. K. Howland and J. Good. 2014. Learning to communicate computationally with flip: A bi-modal programming language for game creation. *Computers & Education*.
18. J. Kaput, R. Noss, and C. Hoyles. 2002. Developing new notations for a learnable mathematics in the

computational era. *Handbook of international research in mathematics education*: 51–75.

19. M. Kölling, N. C. C. Brown, and A. Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (WiPSCE '15), 29–38.
20. J. H. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4: 16.
21. Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 185–190.
22. J. Mönig, Y. Ohshima, and J. Maloney. 2015. Blocks at your fingertips: Blurring the line between blocks and text in GP. In *2015 IEEE Blocks and Beyond Workshop*, 51–53.
23. S. Papert. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic books, New York.
24. S. Papert. 2006. Afterword: After how comes What. In *The Cambridge Handbook of the Learning Sciences*, R.K. Sawyer (ed.). Cambridge University Press, 581 – 586.
25. D. Parsons and P. Haden. 2007. Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Procedings of The Twentieth Annual NACCQ Conference*, 209–215.
26. K. Powers, S. Ecott, and L.M. Hirshfield. 2007. Through the looking glass: teaching CS0 with Alice. *ACM SIGCSE Bulletin* 39, 1: 213–217.
27. T. W. Price, N. C. C. Brown, D. Lipovac, T. Barnes, and M. Kölling. 2016. Evaluation of a Frame-based Programming Editor. In *In Proceedings of the 2016 ACM Conference on International Computing Education Research*, 33–42.
28. M. Resnick, B. Silverman, Y. Kafai, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, and J. Silver. 2009. Scratch: Programming for all. *Communications of the ACM* 52, 11: 60.
29. J. J. Ryoo, J. Margolis, C. H. Lee, C. D. Sandoval, and J. Goode. 2013. Democratizing computer science knowledge: transforming the face of computer science through public high school education. *Learning, Media and Technology* 38, 2: 161–181.
30. R. B. Shapiro and M. Ahrens. 2016. Beyond Blocks: Syntax and Semantics. *Commun. ACM* 59, 5: 39–41.
31. A. Stead and A. F. Blackwell. 2014. Learning Syntax as Notational Expertise when using DrawBridge. In *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2014)*, 41–52.
32. M. Tempel. 2013. Blocks Programming. *CSTA Voice* 9, 1.
33. D. Weintrop. 2016. Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms. Northwestern University, Evanston, IL.
34. D. Weintrop and N. Holbert. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '17), 633–638.
35. D. Weintrop and U. Wilensky. In Press. Comparing Blocks-based and Text-based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education (TOCE)*.
36. D. Weintrop and U. Wilensky. 2015. To Block or Not to Block, that is the Question: Students' Perceptions of Blocks-based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (IDC '15), 199–208.
37. D. Weintrop and U. Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (ICER '15), 101–110.
38. D. Weintrop and U. Wilensky. 2015. The challenges of studying blocks-based programming environments. In *2015 IEEE Blocks and Beyond Workshop*, 5–7.
39. U. Wilensky and S. Papert. 2010. Restructurations: Reformulating knowledge disciplines through new representational forms. In *Proceedings of the Constructionism 2010 conference*.
40. 2014. *App Inventory Java Bridge*. Retrieved from https://code.google.com/p/apptomarket/