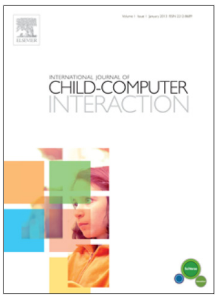


# How block-based, text-based, and hybrid block/text modalities shape novice programming practices

David Weintrop, Uri Wilensky

PII: S2212-8689(17)30031-4
DOI: https://doi.org/10.1016/j.ijcci.2018.04.005
Reference: IJCCI 103

To appear in: *International Journal of Child-Computer Interaction*

Received date : 10 February 2017
Revised date : 9 April 2018
Accepted date : 30 April 2018

# How Block-based, Text-based, and Hybrid Block/Text Modalities Shape Novice Programming Practices

David Weintrop[a] and Uri Wilensky[b]

[a] Teaching & Learning, Policy & Leadership
College of Education
College of Information Studies
University of Maryland
3942 Campus Dr. Suite 2226D
College Park, MD 207421427
weintrop@umd.edu

[b] Center for Connected Learning and Computer-Based Modeling
Learning Sciences and Computer Science
Northwestern University
2120 Campus Dr.
Evanston, IL, USA, 60208
uri@northwestern.edu

*Abstract*
There is growing diversity in the design of introductory programming environments. Where once all novices learned to program in conventional text-based languages, today, there exists a growing ecosystem of approaches to programming including graphical, tangible, and scaffolded text environments. To date, relatively little work has explored the relationship between the design of novice programming environments and the programming practices they engender in their users. This paper seeks to shed light on this dimension of learning to program through the careful analysis of novice programmers' experiences learning with a hybrid blocks/text programming environment. Specifically, this paper is concerned with how novices leverage the various affordances designed into programming environments and programming languages to support their early efforts to author programs. We explore this relationship through the construct of modality using data from a study conducted in a high school computer science classroom in which students spent five weeks working in blocks-based, text-based, and hybrid blocks/text programming environments. This paper uses a detailed vignette of a novice writing a program in the hybrid environment as a way to characterize emerging programming practices, then presents analyses of programming trends from the full study population to speak to the generality of the practices identified in the vignette. The analyses focus not only on characterizing authoring strategies but also on identifying patterns in novices' help-seeking behaviors. By focusing on how modality influences novices' emerging programming practices, this paper contributes to our understanding of the relationship between programming environment and learning, illuminating the role of design in shaping introductory programming experiences.

*Keywords*
Design; Modality; Programming Environments; Computer Science Education; Block-based Programming

## 1. Introduction

The last decade has seen a rapid growth in the number of ways young learners can engage in the act of programming. This includes new programming environments, toys that feature programming and other computing ideas, and online campaigns designed to introduce computer science to large numbers of learners around the world. Across these contexts, graphical block-based programming has a growing presence, spanning both formal (classroom) and informal settings [1]. Recently, a wave of blended and dual-modality programming environments has emerged that integrate affordances of the block-based programming approach into conventional text-based programming interfaces, further diversifying the introductory computing landscape. Despite this growth in the variety of introductory programming environments, relatively little is known about the relationship between modality, programming interface design, and the impact they have on learners' emerging programming practices. We use the term modality to capture both the representational infrastructure used as well as the set of interactions the interface supports. Understanding the effects of emerging programming modalities for novices is critical to the larger goal of making computing education effective and accessible to all learners. This paper seeks to address this gap in the literature by answering the following research question:

*How does modality affect learners' emerging programming practices?*

To answer this question, we use data from a 5-week quasi-experimental study conducted in a public high school in a Midwestern American city. Our analysis includes an illustrative vignette of one novice programming in a hybrid blocks/text environment with special attention being paid to practices that were uniquely afforded by the hybrid modality. The paper then looks at programming practices across the full set of participants (n = 90) who learned to program in either a block-based, text-based, or hybrid environment, using log data collected from the programming environments to identify macro trends and how they differ by modality. The paper concludes with a discussion of the implications of this work with respect to modality and the design of introductory programming environments.

## 2. Related Work

This section focuses on work, both theoretical and empirical, upon which our evaluation of novice programming environments relies. The goal of this section is to present the foundational ideas that underpin our approach as well as to highlight the need for the proposed work on programming practices as a complement to existing scholarship.

### 2.1 On Modality

Given a semantics, we use the term *modality* to capture how one interacts with and composes within that semantics. Such interactions are defined and supported by the presentation and capabilities of the semantics itself, be they visual, expressive, or tangible. In looking at the interactions enabled by the presentation of a semantics, modality is not a characteristic of a representational system alone but also captures the relationship between the representation, its interface, and how one uses it. This conceptualization of modality is similar to the notion of affordance [2,3] in that it captures a characteristic of the interaction between an actor and the thing being acted upon, in this case, a representational system. Our choice for the word modality is intended to link the interactional characteristics and capabilities with the way in which the semantics exists, i.e. its modal qualities. In the literature, the term "modality" is often used to capture human sensor modes (visual, auditory, etc.), as can be seen with the *modality effect*, which describes the different outcomes of presenting information visually versus through an auditory mode [4]. We view our use of the term modality as consistent with this usages in that the modality label captures the relationship between the form in which something is presented and how it shapes the way one interacts with it. Our use of the term modality differs from this usage in that the modes we are concerned with are not defined by human sensory-motor capabilities but instead the modes in which an idea is expressed or interacted with.

Our conceptualization of modality is similar to other terms used to describe the design of learning environments and representational systems but captures a distinct dimension of the learning interaction. In this work, we use the term *representation* to describe the set of symbols used to present concepts to the user akin to how Palmer [5] and later Kaput [6] use of the term. In this work, the programming language serves as the representation of interest. The term *interface* is intended to capture the presentation of the representation, specifically how programming commands are presented (e.g. as blocks with colors and notches intended to convey additional meaning). The data presented in this work are drawn from a block-based interface, text-based interface, and hybrid block-text interface, all of which use the same programming language. The interface, and thus the representation, are a part of the larger *environment*, which includes the canvas upon which block-based programs are constructed, the block palette where the set of available commands are collected, as well as the stage where programs are visually executed. This nomenclature for describing the components of the environment is drawn from the Scratch community [7]. These three terms (representation, interface, and environment) all describe this system as it is, independent of the user. The introduction of the term modality is intended to give us a label for the set of action enabled by these aspects of the system which is useful given our ultimate focus on how features of the system affect the user. Saying block-based modality thus captures the interface and representation as well as the suite of actions enabled by it (e.g. browsing the block palette or dragging-and-dropping commands). Our motivation in making this shift is to explicitly include the set of possible interaction and their influence on the user as part of our analytic focus. In doing so, the term recognizes the foundational role the user plays in bringing an interface to life and categorizing its impact. This shift from interface to interface-plus-actor also motivates the specific analysis presented in this work, where the block-based environment is analyzed through user actions as opposed to independent of them.

Our conceptualization of modality is akin to an interface metaphor [8]. However, given the larger focus of this line of work on cognitive and perceptual outcomes of design on the learner, we have adopted the term modality to shift focus to the space between the user and the system, rather than prioritizing the system and secondarily the user (as a term like interface metaphor might connote). Further, the construct of modality gives us a language for comparing interaction patterns enabled by different interfaces in a consistent way, thus avoiding needing to compare across labels (e.g. comparing the block-based interface metaphor with a text-based language). Throughout this paper, we use the term interface when describing the environment independent of how it will be used and modality when discussing the environment and user in conjunction.

### 2.2 Block-based Programming

Block-based programming interfaces leverage a programming-primitive-as-puzzle-piece metaphor that provide visual cues to the user about how and where commands can be used as their means of constraining program composition [9]. Programming in these environments takes the form of dragging blocks onto a canvas and snapping them together to form scripts. If two blocks cannot be joined to form a valid syntactic statement, the interface prevents them from snapping together, thus preventing syntax errors but retaining the practice of assembling programs instruction-by-instruction. Early versions of this interlocking approach include LogoBlocks [10] and BridgeTalk [11] which helped formulate the programming approach which has since grown to be used in dozens of applications. Alice [12] and Scratch [13] are more recent implementations that have achieved widespread use. Figure 1 shows programs written in these four blocks based programming environments.

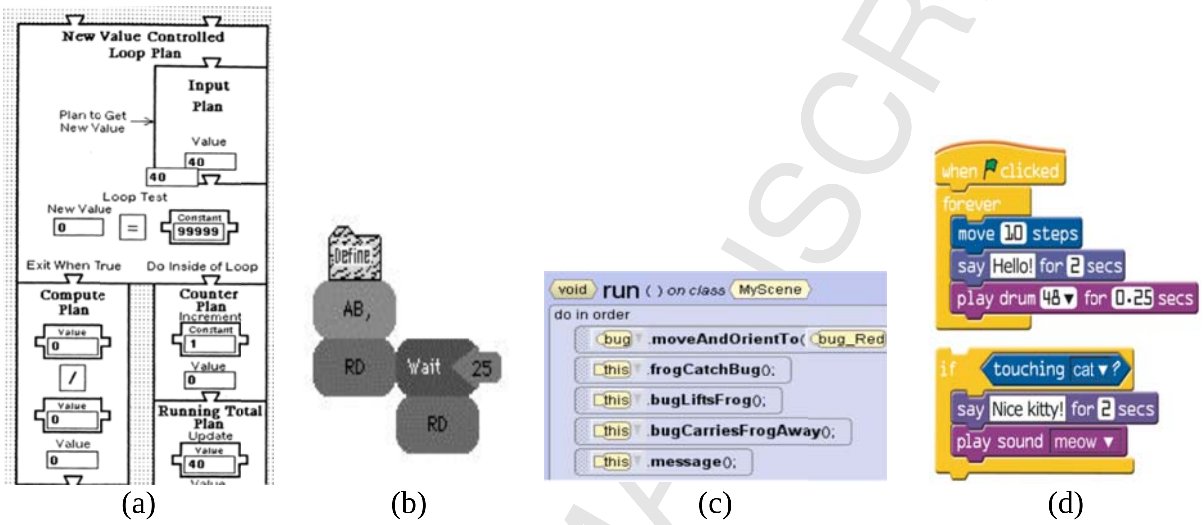


**Figure 1.** Four example block-based programming environments: (a) BridgeTalk, (b) LogoBlocks, (c) Alice, and (d) Scratch.

In addition to being used in more conventional computer science contexts, a growing number of environments have adopted the block-based programming approach to lower the barrier to programming across a variety of domains ranging from mobile app development with MIT App Inventor [14] to modeling and simulation tools like DeltaTick [15] and NetTango [16] to game-based learning environments like RoboBuilder [17]. Further, a growing number of libraries are being developed that make it easy to develop application-specific or task-specific block-based languages [18,19].

The last few years have also seen the emergence of new types of editors that further explore and utilize modality as a design mechanism for supporting novice programmers. Dual-modality environments like Pencil Code [20] and Tiled Grace [21] give learners the ability to move back and forth between block-based and text-based versions of their programs, a strategy found to be utilized by novices while learning to program [22,23]. Other approaches, notably Greenfoot’s frame-based editor [24], present hybrid blocks/text interfaces that blend features of block-based composition along with features of conventional text editors to create new ways for novices to author and edit programs. The growing role of block-based, hybrid, and dual-modality programming environments in introductory computing contexts speaks to the need for more critical research around the affordances and drawbacks of these approaches [25,26].

### 2.3 Evaluating Block-based Programming Environments

To date, most evaluations of block-based programming have focused on Scratch and Alice, as these two environments have the widest use in contemporary K-12 computer science education. Scratch from its inception, was focused on younger learners and informal environments [13] and has been successful at generating excitement and engagement with programming among novice programmers [27–30]. Ben-Ari and colleagues conducted a number of studies on the use of Scratch for teaching computer science and concluded that Scratch could successfully be used to introduce learners to central computer science concepts [31] but that it could also lead to potentially undesirable practices like totally bottom-up programming and a tendency towards extremely fine-grained programming [32]. There is also work showing that block-based programming, in conjunction with effective pedagogy can serve as an effective way to develop important computational thinking skills and prepare learners for future computer science instruction [33]. Other work has found the block-based programming is a developmentally appropriate way to introduce younger learners at different ages to the practice of programming [34].

Compared to Scratch, the Alice programming environment has a longer history of serving as the focal programming environment in introductory programming courses. Much of the motivation for using Alice in courses is based on findings that Alice is more inviting and engaging than text-based alternatives and improves student retention in CS departments [35–37], however, other educators have been less successful in replicating these results and have struggled in transitioning learners from Alice to conventional text-based environments [38,39]. This challenge was addressed with the release of Alice 3 and its inclusion of supports to transition learners to the Java programming language [40].

A growing body of research is conducting systematic comparisons of block-based and text-based environments. A study comparing students answering multiple choice programming questions found that students perform better on questions asked with a block-based representation compared to the text-based equivalent [41]. Other studies investigating learning outcomes in blocks versus text environments found little difference in learning outcomes but did report that students completed activities in block-based environments at a faster rate [42].

## 2.4 Blending Block-based and Text-based Programming

As block-based programming matures and more is known about the affordances and drawbacks of the interface, a growing number of environments are blending features of block-based and text-based programming in a single environment. Two main strategies have been taken to accomplish this: dual-modality environments and hybrid block/text environments. Dual-modality programming environments support both block-based and text-based programming, allowing the user to move back-and-forth between the two modalities as they work. Examples of such environments include Pencil Code [20], Tiled Grace [43], and BlockEditor [22]. Empirical evaluations of dual-modality environments show how such an approach can support learners with differing level of prior experience and support just-in-time scaffolds when the user needs it [22,23].

Unlike dual-modality environments which support both modalities, hybrid block/text environments blend features of block-based interfaces with conventional text-based interfaces to create a new modality distinct from both blocks and text but sharing characteristics of both [44]. We classify these as new modalities as they enable new interaction patterns and potentially provide additional cognitive or perceptual supports for the user distinct from other modalities. There are a growing number of environments that live in this space, each of which offer unique interactions and visual displays. One recent example of this type of environment is Greenfoot's Frame-based Editor [45]. 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" [24]. 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 [46]. A second example of Hybrid block/text programming environment can be seen with GP, which is inspired by Scratch and seeks to address the drawbacks of block-based interfaces as programs become larger and more complicated by incorporating text layouts and keyboard-driven compositional mechanisms [47]. Other hybrid environments include DrawBridge [48] which supports a gradual transition and Pencil.cc, which is the focus of this paper and described in detail below.

**2.5 Theoretical Framing**

Before continuing with the design of the study and findings, we now present the theoretical framing that informs both why we are pursuing the stated research question as well as the methods and analytic techniques employed to answer it. In conducting this work, we draw on a number of theoretical lenses which inform how we think about the role design plays in how learners interact with different tools and representations. Central to our theoretical approach is the view that the tools and representations serve as resources that learners draw upon and, in turn, shape both the practices that develop as well as the conceptual understandings that emerge. This perspective draws on Norman and Hutchins' theory of distributed cognition [49,50] as well as Noss and Hoyles' construct of Webbing [51]. Distributed cognition "extends the reach of what is considered cognitive beyond the individual to encompass interactions between people and with resources and materials in the environment" [52]. Through this lens, the tools and representations are included within the bounds of the cognitive activity, meaning they do not influence cognition, but instead are a fundamental component of it. In bringing this perspective to a learning task, the design of the tools and representational infrastructure used become foundational to how one learns, what one learns, and the practices the learner develops [53]. When we use the term learning in this context, we refer to both conceptual learning as well as the practices associated with the content.

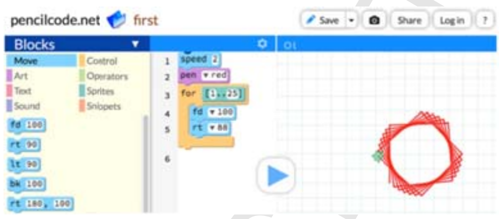
The representational infrastructure used within a domain is not fixed, but instead, it is a designed system that can change over time, with new, more powerful and accessible systems supplanting traditional expressive approaches [54]. This is particularly true with respect to computing due to the rapidly changing fields of programming language design and human-computer interaction. This malleability is important to recognize given the emergence of new representations and the design challenges and opportunities that accompany them. To investigate the shifting of representational systems, we draw on Noss and Hoyles' construct of Webbing. Webbing characterizes the relationship between representation and learning by capturing the "structures that learners draw upon *and reconstruct* for support – in ways that they choose as appropriate for their struggle to construct meaning" [51]. In this work, we use Webbing as a way to map features of the modality to specific practices observed in learners. Through this theoretic lens, the features of the learning environment serve as a suite of resources the learner can draw on throughout their meaning-making process and the development of practices they find useful. As such, the design of components of the cognitive system both define and shape the practices that emerge and learning that can occur. Recognizing the integral and interrelated ways that representation, interface, environment, and modality shape the conceptual and mechanical dimensions of learning to program both motivates this work and informs our approach to analyzing it.

**3. Methods**

### 3.1 Meet Pencil.cc

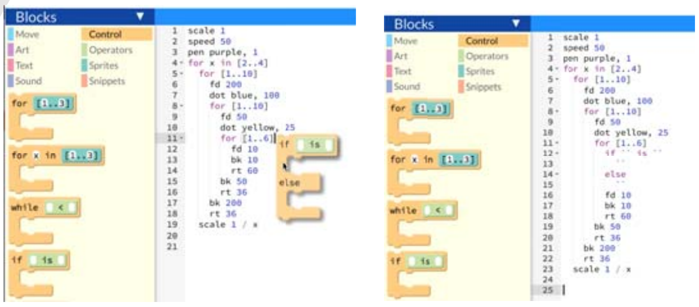
Pencil.cc is a customized version of the Pencil Code programming environment [20] created for this study. In the original Pencil Code, users are free to move back-and-forth between the block-based and text-based interfaces. With Pencil.cc, this ability has been removed, instead, users see a single interface: block-based, text-based, or a hybrid interface that blends features of both the block-based and text-based versions of the interface. The blocks interface of Pencil.cc (Figure 2a) features many of the defining features of block-based interfaces, 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 replaces the blocks palette and canvas with a basic text editor that includes basic programming supports like highlighting, automatic formatting, and syntax checking. Programming in the text interface matches conventional text-based programming editors where authoring a program is done by typing in one character at a time.

The hybrid interface 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 shows Pencil.cc's hybrid interface. This hybrid approach was informed by earlier findings on the design of block-based interfaces, including features that learners found to be useful as well as perceived drawbacks of the block-based programming approach [55]. Specifically, the hybrid design retains the browsable blocks library, supports drag-and-drop composition, and provides pre-fabricated commands. At the same time, the text-based editor present in the hybrid interface addresses some of the drawbacks identified by learners with block-based environments, such as perceived inauthenticity and issues with authoring speed and expressive power [55]. Aside from the programming interface, all other features of the three modes of Pencil.cc are the same, meaning the capabilities and expressive power of the environments are equivalent; anything that can be done in one interface can also be achieved in the other two.



```
speed 2
pen red
for [1..25]
  fd 100
  rt 88
```

(a)



```
scale 1
speed 50
pen purple, 1
for x in [2..4]
  for [1..10]
    fd 200
    dot blue, 100
    for [1..10]
      fd 50
      dot yellow, 25
      for [1..6]
        fd 10
        bk 10
        rt 60
      bk 50
      rt 36
    bk 200
    rt 36
  scale 1 / x
```

```
scale 1
speed 50
pen purple, 1
for x in [2..4]
  for [1..10]
    fd 200
    dot blue, 100
    for [1..10]
      fd 50
      dot yellow, 25
      for [1..6]
        if `` is ``
          ``
        else
          ``
        fd 10
        bk 10
        rt 60
      bk 50
      rt 36
    bk 200
    rt 36
  scale 1 / x
```

(b)

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

### 3.2 Study Design and Data Collection Strategy

This study has a quasi-experimental design, following three high school introductory programming classes for the first 5 weeks of a year-long Introduction to Programming course.

Each of the classes used a different variant of Pencil.cc. One class used a block-based version, the second used a text-based version, and the third used the hybrid blocks/text interface described above. The study began on the first day of school and collected a variety of quantitative and qualitative data. This paper relies on two primary data sources: semi-structured clinical interviews conducted in the final week of the study and computational logs generated throughout the study. For the one-on-one student interviews, the researcher sat alongside students, first asking them questions about their experiences in the class and having them describe projects they completed. In the second half of the interview, students were asked to think-aloud as they worked through activities designed to elicit specific types of thinking around computer science concepts. The computational log data was gathered by Pencil.cc which recorded information about a student's program every time a student clicked the run button. A total of 134,444 Pencil.cc events were collected from the students across the three conditions. Further details about the study design and data collection strategy can be found in [56].

## 3.3 Setting and Participants

This study was conducted at a large, urban, public high school in an American Midwestern city, serving almost 4,000 students. The school is a selective enrollment institution, meaning students have to meet academic standards to attend, although steps are taken by the district to ensure a diverse student population comprised of learners from across the city. The student body is racially diverse and a majority of the students in the school (58.6%) come from economically disadvantaged households.

The experiment was conducted in an existing Introduction to Programming elective course. 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. Relative to the larger student body, White students were overrepresented and Hispanic students were slightly underrepresented, with the other racial groups roughly matching the larger school demographics. 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 recruitment for the courses was beyond the control of the researchers. Of the students participating in the study, almost half (47%) speak a language other than English in their households. The same teacher taught all three sections of the course, allowing us to control for teacher effects.

## 4. Novice Programming Practices

Our analysis of novice programming practices is divided into two sections. First, we present a detailed vignette from an interview with one student from the Hybrid condition. In the next section, we use the computational data collected to link practices and trends found in the vignette to the full set of Hybrid participants and to learners who worked in the other modalities. Note, this analysis is focused specifically on programming practices; an analysis of learning and attitudinal outcomes from this study can be found in [44,56,57].

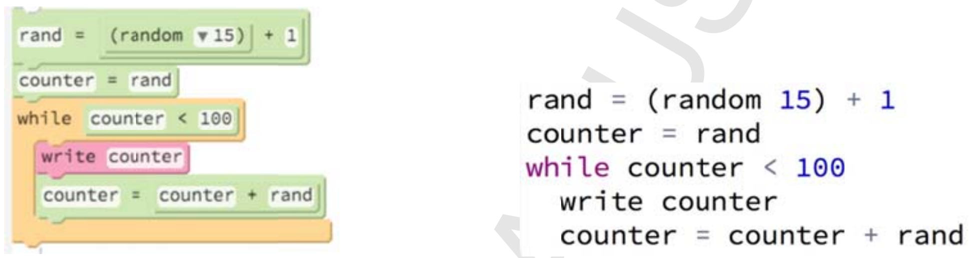
### 4.1 A Vignette of a Novice Programming in a Hybrid Blocks/Text Environment

This vignette is intended to serve as a representative case study to understand how novices navigate and utilize features present in an introductory programming environment, focusing on unique interaction patterns and distinct practices supported by the blocks-text modality. This interview was characteristic of all of the Hybrid interviews and selected because it serves as a nice example of the breadth of supports and challenges learners encountered. Not every moment of the session is present, but instead, an effort was made to provide enough detail to present a clear picture of how the session progressed, with an emphasis placed on key moments and interesting interactions.

The vignette follows a student as he writes a program in response to this prompt:

*Can you write a program that picks a random number less than 15 and then prints out every multiple of that number that is less than 100? So, for example, if your program picked the number 11, it would print out 11, 22, 33, 44, 55, 66, 77, 88, and 99. If it picked the number 2, it would print out 2, 4, 6, 8, 10, 12, 14 and so on, up until 100.*

This specific programming challenge was selected for a number of reasons, including the fact that it is a manageable size, asks students to use a number of programming concepts (e.g. variables and iteration), supports numerous solutions, and includes a number of natural pitfalls. While there are many ways to achieve the desired behavior, the most common approach, and the one taken in the vignette below follows the logic shown in Figure 3.



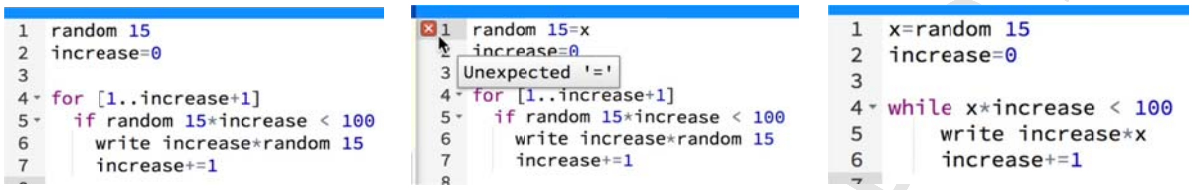
```
rand = (random ▾15) + 1
counter = rand
while counter < 100
  write counter
  counter = counter + rand
```

```
rand = (random 15) + 1
counter = rand
while counter < 100
  write counter
  counter = counter + rand
```

(a) (b)

**Figure 3.** A block-based (a) and text-based (b) solution to the interview prompt

This vignette is of a student had not reported any prior programming experience before the course. After hearing the prompt, the student starts by clicking through the various categories in the blocks palette: `move`, `control` (where he scrolls up and down through the blocks), `text`, then finally `operators`, where he scrolls up, before finally dragging the `random` block out of the palette and into his program. He then puts the cursor next to the default value of `6` and replaces it with `15`. After asking a clarifying question, he then hits the return button, moving his cursor to a blank line, and types: `increase=0`. He then returns to the blocks palette, clicks through a few more categories before opening the `control` group and dragging a `for` block onto the canvas. The `for` block provides a template for the looping structure (as can be seen in Figure 2b). After adding the block to his program, the student deletes the placeholder inside the `for` loop, causing the editor to display a red x next to the loop's definition. This denotes there is an error in the code, in this case, because there is nothing inside the `for` loop. The student hovers his mouse over the error, sees the message `'Unexpected Terminator`,' pauses for a second, then clicks on the Text category of the palette and drags a `write` block into the `for` loop, which causes the red x to disappear. The `write` block has the default argument of `'hello.'`, which the student deletes and replaces with `increase*` and then highlights the first line of the program (`random 15`), copies it and then pastes it after the `*`, giving him the line: `write increase*random 15`. The student next adds a conditional statement to his program by typing the line `if random 15*increase < 100`. He types this command from memory, not using the block palette or other environmental supports for help. He then makes a few more modifications to his program using the keyboard, including changing the code that controls the loop and adding a line to increment his variable. At this point, he has been programming for two minutes and 48 seconds and has written the program shown in Figure 4a.



```
1  random 15
2  increase=0
3
4  for [1..increase+1]
5    if random 15*increase < 100
6      write increase*random 15
7      increase+=1
```

(a)

```
1  random 15=x
2  increase=0
3  Unexpected '='
4  for [1..increase+1]
5    if random 15*increase < 100
6      write increase*random 15
7      increase+=1
8
```

(b)

```
1  x=random 15
2  increase=0
3
4  while x*increase < 100
5      write increase*x
6      increase+=1
```

(c)

**Figure 4.** The Hybrid student's first run program (a), an error message displayed by the editor (b), and his final program (c).

At this point, the student pauses and the interviewer asks him what he would like to do next. After a pause, the student responds "*I'm going to set this to a variable because I'm not sure if this `random 15`* (clicks and drags to highlight the `random 15` in line 5) *is the same as this `random 15`* (hovers his mouse of line 1). *I'm just going to set it to* `x`." He then adds the characters `=x` to the end of line one, so the line reads: `random 15=x`, which causes the editor to display a red x next to the first line. The student sees this and says: "*wait, what's this*?" and hovers over the x, revealing the errors message shown in Figure 4b. The student reads this message, pauses for a second, then adds a second = to the line so the first line of the program reads `random 15==x`[1]. This addition to the program resolves the compile-time syntax error and the red x disappears.

Upon running the program, he gets an error saying `'x is not defined'`. He pauses, thinks for a minute, then deletes the `==x` at the end of the line and then adds `x=` in front of the `random 15` command. This fixes the error, and he hits run again. His program then prints out the character `0` and then stops. To try and figure out what is happening in his program, the student changes the `+1` in line 4 to `+2` and reruns the program. It now prints out `0` on the first line and then `2` on the second line, which gives him some insight into the behavior. He says "*I'd also like to see what `x` is*" and then puts the cursor at the end of line 1 and types in `write x`. He then asks "*that is it, right?*" and drags a `write` block into the program, compares the syntax of the newly added command with the command he just typed and says "*yeah*" before deleting the `write` command he just drag-and-dropped into his program. The student then reruns the program again, now with a debug statement, the output is: `2 0 10`, which causes the student to say "*that's strange. Oh*" and changes the `random 15` command that is still in line 6 to use the variable `x`. He then runs the program again, which produces: `6 0 6`, which, based on the student's facial expression, was what he was expecting to see. A moment after running this program the student has a revelation. "*Oh, I know, I know what to do, I think a `while` loop would be much better.*" He then highlights the `for` loop definition in line 6, deletes it all at once, types in `while`, and then copies and pastes the condition out of the `if` statement, producing the program showing in Figure 4c. "*Yeah, this ought to work better.*" The student then runs the program, which

[1] In Pencil.cc, the double equals sign (==) can be used to compare the equality of two objects whereas a single equals sign (=) is used for assignment. The use of the double equals sign in line one is syntactically valid but not what the student intended.

demonstrates the correct behavior. In authoring this program, the student spent a total of 9 minutes and 8 seconds, ran his program 9 times, and encountered two syntax errors.

### 4.1.1 Hybrid Vignette Discussion

In this vignette, we see the student leverage affordances common to block-based programming environments but also demonstrates conventional text-based author practices. The first interesting thing to note from this vignette is how the student moved back and forth between dragging in commands from the palette and typing them directly into his program. This could be seen right from the beginning in that the first command (`random 6`) in the program was added via dragging the block onto the editor and the second command (`increase=0`) was typed. For the third command (a `for` block) he goes back to the palette to again drag-and-drop a command into his program, which he then modifies with the keyboard (deleting the placeholder for the nested statement). This shows the hybrid interface supports two modes of composition and that this novice fluidly moved between them while composing his program. In the vignette, we also see the learner clicking through categories in search of commands, showing that the browsability of the block-based interface has been retained.

Another noteworthy feature in this vignette is the presence of in-editor feedback for syntax errors and the student responses. Twice during this activity, the student introduced an error; in both cases, it was while he was typing in commands with the keyboard. One of these errors happened when the student typed in the line: `random 15=x`. What is interesting about this is that the student's next move was to add a second `=`, producing the line `random 15==x`, which is still incorrect, but is no longer a compile-time error, which gives the impression that the error has been corrected. This interaction is noteworthy as this type of error would be very difficult to make in a block-based interface where more constraints are placed on how commands can be assembled. Further, the approach of tinkering with commands by adding and removing characters in hopes of resolving compile-time errors is largely absent from our block-based interview data.

The introduction of syntax errors was frequently observed in both the Text and Hybrid modalities, but in the Hybrid case, there are additional supports provided by the interface that can help address this. One clear example occurred when the student wanted to add a debugging statement to his program, he typed in the line `write x` and then dragged out a `write` block and placed it below the line he just typed to check the syntax. Upon seeing that what he had typed matched what appeared when he dropped the `write` block, he deleted the second command and continued working. This pattern of using the blocks as a way to check syntax keyed in was observed in all four of the hybrid interviews conducted and reported as a frequent strategy used by students throughout the 5-week study. In this capacity, the blocks were not serving as a means for remembering what is possible, or a way to author new portions of a program, but instead serving as a way for students to double-check to make sure they were doing things correctly. This pattern was unexpected and reveals one type of support novice programmers need: in-editor scaffolds to quickly verify syntax.

A final thing to note from this vignette is that the student utilized a number of common text-editing techniques: notably, copy-and-pasting lines of code to move them around and highlighting blocks of text either to denote something to the interviewer or to delete portions of the program. Seeing the student make these types of moves is not particularly surprising as high school students are usually comfortable with text manipulation. This is noteworthy in that the Blocks modality does not give the student the ability to do this type of character-by-character highlighting or easily support copy-and-pasting sets of commands and thus was not observed in the interviews with students from the Blocks condition of the study.

## 4.2 Programming Practices Across All Participants

Having provided a qualitative description of what it looks like for a novice to program in the Hybrid modality and how he blended conventional block-based and text-based programming practices, we now investigate emerging programming patterns and practices across the full set of participants using the computational log data collected during the study.

### 4.2.1 Running Programs

The run event is captured every time a student runs their program in Pencil.cc A total of 76,110 run events were captured by the logging system during the 5-week study. We focus on run events in this section as it serves as a proxy for one dimension of emerging learner programming practices; that of the speed of completion of programs, reliance on program output for evaluation and reflection, and a measure of the iterative program development process. Calculating the average number of runs per student by condition shows the Blocks students running a program an average of 733 times, Hybrid students running their programs an average of 1,073 times, and Text students running their programs 742.9 times on average. An ANOVA calculation of the average number of runs per student in each condition shows there to be a statistically significant difference between the condition $F(2,89) = 8.71$, $p < .001$. A Tukey HSD post hoc analysis shows that students in the Hybrid condition ran their programs significantly more often than the other conditions (compared to Blocks $p < .001$, compared to Text $p = .003$), while there was no difference in the number of runs between Blocks and Text students ($p = .86$). This pattern of students in the Hybrid condition running their programs more frequently than the other two conditions is relatively consistent across the 5-week curriculum, as can be seen in Figure 5, which shows the average number of runs by students per assignment, ordered chronologically from the first assignment (Quilt) to the summative Final Project. Figure 5 also shows the concept covered in each assignment, to help situate the activities within the larger curriculum[2]. The full curriculum can be found in [56].

[2] We report independent ANOVA calculations in Figure 5 rather than a single model due to the shifting nature of assignments across the curriculum. Assignments varied by concept, form of output (textual vs. graphical), and constraints (narrow vs. open). These difference led us to favor independent tests, providing more clearer insight into where difference emerged.

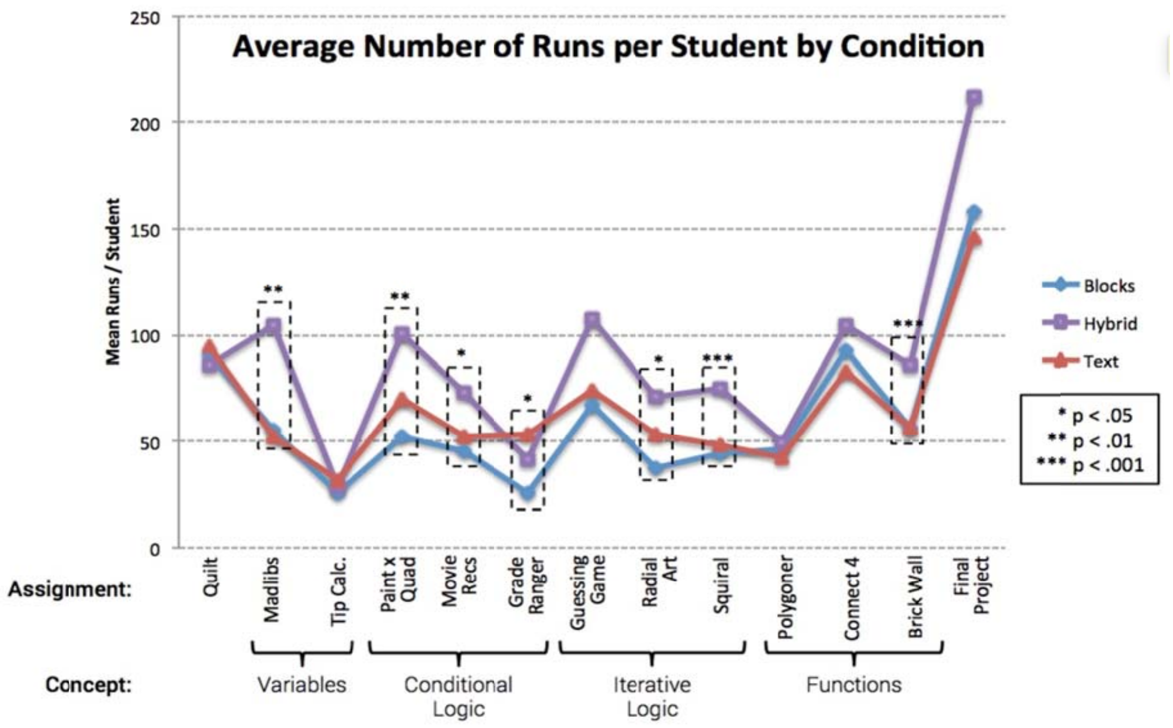


**Figure 5**. The average number of runs by students for each project chronologically, broken down by condition.

Our explanation for this pattern, and the reason we are focusing on it in this analysis is that it is linked to how students in the Hybrid condition used practices tied to both the blocks and text modalities. In the vignette above, four of the nine runs that occurred show the student checking to see if his program is achieving the desired behavior, two of the runs were him trying to correct a syntax error, two more were him running his program to try and debug his solution after having added a write command for debugging purposes, and finally, he reran his program one time without making any changes to see if behavior changed. As will be explained below, this distribution of runs shows how the hybrid modality supported both block-based and text-based practices in learners and explains the high average run frequency shown in the aggregate data.

Two of the nine runs in the Hybrid condition were driven by the student trying to resolve a syntax error, a situation that rarely occurred in the Blocks condition but was common in the Text condition. These runs happened in quick succession as they involved the student making small changes (usually only a few characters in their program). Using the timestamps recorded for runs, we can see how often students ran their programs in quick succession (defined as within 5 seconds of the previous run) after making a small change to their program (thus excluding reruns of an unaltered program). Students in the Blocks condition made small changes and quickly reran their programs an average of 6.8 times per assignment, compared to 12.2 times for Hybrid students and 14.1 times for learners in the Text condition, a difference that is statistically significant ($F(2, 89) = 4.94$, $p < .01$). So here we see the Hybrid condition demonstrating a pattern similar to learners in the Text condition.

At the same time, the Blocks modality makes it easy for students to quickly add commands to their program because dragging-and-dropping is faster than typing in commands one character at a time. As a result, on average, students in the Blocks condition produced programs that were longer in length than their Text and Hybrid peers. On 10 of the 13 assignments in the 5-week curriculum the Blocks students produced the longest programs on average, with students in the Hybrid condition producing the longest programs in the other three assignments. Running an ANOVA calculation for each of the assignments, four were found to have statistically significant differences across conditions at the $p < .05$ level: Tip Calculator ($F(2, 82) = 4.78$, $p = .01$) , Grade Ranger ($F(2, 71) = 5.26$, $p = .01$), Radial Art ($F = (2, 83) = 3.51$, $p = .03$) and Connect 4 ($F(2, 87) = 2.90$, $p = .05$). In all but the Connect 4 assignment, the Blocks condition students produced the longest programs and the Text students had the shortest programs. The assignments with the greatest stratification of program length focused on conditional logic (Paint by Quad, Movie Recommendation Engine, and Grand Ranger) and the last two assignments from the functions portion of the course (Connect 4 and Brick Wall). The variance in the conditional logic assignments is similar to what was seen in the runs-by-assignment analysis (Figure 5), but that pattern does not continue with the iterative logic assignments or the functions assignments. This variation in the Connect 4 and Brick Wall assignments may come from the fact that those two assignments were by far the most difficult in that they asked students to incorporate logic from previous parts of the course and required the most amount of code to accomplish relative to the other assignments[3]. The fact that we see a difference in conditional logic is another piece of evidence towards the larger trend of modality affecting students' learning and use of those constructs [41,58]. In this case, we are using program length as a rough proxy for ease of composition given that all conditions had the same time on task. The fact that programs can be assembled more easily contributes to students running their programs more often to check the correctness of their program, which, again, leads to students in the hybrid condition, on average, running their programs more frequently than their text-based or block-based peers.

Collectively, these data show how the Hybrid interface has the ease-of-composition of the block-based modality, which makes it easy to quickly add commands to the program to see if they work. At the same time, it also allows for syntax errors, due to the lack of constraints on how and where commands can be added. In this case, the blended Hybrid interface results in a summative behavior (i.e. students do both) as opposed to reductive outcome (i.e. the Hybrid interface relieves the user from having to do certain things).

## 5. Discussion

This paper investigates how programming practices are shaped by modality, specifically looking at practices novices developed while working in a hybrid blocks/text modality. Using a variety of methods and data sources, this paper reveals characteristics of programming in a hybrid modality and how practices adopted draw from both of the source modalities. Here we discuss implications of the finding that interface shapes modality and modality shapes emerging programming practices.

### 5.1 Modality Matters

In the vignette, we see how one student used various compositional strategies and techniques to write a functioning program. The student used the blocks palette for syntactic help and to browse the set of available blocks for sources of inspiration. While at the same time, he typed in commands from memory and used copy-and-paste text editing moves characteristic of

[3] The Grade Ranger and Movie Recommendation Engine assignments' numbers are inflated due to the amount of text included in the assignment.

programming in a conventional text-based programming interface. Further, the student employed practices unique to the hybrid blocks/text modality when he used the drag-and-drop feature of the blocks as a way to check the syntax of commands he had typed in. Given our conceptualization of modality as affordances of the representation and the interactions they support, this diverse set of uses highlights the way modality shapes programming practices. Drawing on our theoretical framework and taking a distributed condition lens to this environment, we can see the modality and features of the interface influence the act of programming. The existence of the Blocks palette means students do not need to recall specifics about what is possible in the language from memory nor memorize syntactic detail, as the environment itself *knows* this information. In other words, the knowledge of what is possible and how and where commands can be used are encompassed by the modality and thus need not be committed to a learner's memory. Likewise, the set of supports the hybrid interface provided was used in a variety of ways, showing it is not a single use that the design pushes learners towards, but instead, a suite of resources, or *Webbing* using Noss & Hoyles (1996) terminology, that the interface presents and learners use. The conclusion to be drawn from this analysis is a recognition that the resources provided by a specific interface shape modality and that modality impacts novices emerging programming practices.

**5.2 Modality and the Design of Learning Environments**

A second contribution of this work is a demonstration of how modality is malleable and how interface design, along with characteristics of the environment and representation, can change both what users are able to do and how they are able to do it. This perspective opens the door to the larger enterprise of creating new modalities through the revision of existing forms as well as the creation of entirely new ways of expressing ideas and interacting with representational systems. This is akin to Wilensky and Papert's notion of restructuration [54,59], the term they use to describe shifts in representational infrastructure where one set of representational forms is replaced with another. The hybrid interface presented in this study is one example of designing a new modality. In creating this specific blended interface, we sought to retain block-based features identified as useful while also incorporating strengths (real or perceived) of conventional text-based programming. The result was a modality distinct from the two that it drew from and producing unique programming practices in the learner. While the data in this paper do not allow us to claim the hybrid modality is superior, they do represent a successful demonstration of the creation of a new modality and the effects it can have on novice programming practices. This can be seen in how new, unique practices emerged, such as the dragging of blocks into the text area to verify syntax. Likewise, the fact that students in the Hybrid condition utilized productive block-based strategies (such as browsing the categories) as well as incorporated useful text-based moves (like copy-and-pasting chunks of text) show how modality can facilitate and promote productive programming practices in novices. This work, alongside complementary studies looking into learning and attitudinal outcomes of students working in the Hybrid modality [43] show the promise of this line of design work.

A final thing to note about modality and design is the recognition that through design we can give agency to learners and scaffold novices at various points along the learning trajectory. In designing an environment with a rich webbing of various supports, the learner can be in control of their own learning, deciding for him or herself how to proceed. For example, in the vignette, we saw how the student was able to type out an `if` statement from memory, but used the drag-and-drop feature to add a `for` loop, suggesting he had knowledge of the syntax of one of these concepts but not the other. This shows the learning environment meeting the learner at their current developmental level, providing supports when and where they were needed. The larger take away from this work is not just that modality shapes learners' experiences with content, but that the design and evaluation of modalities can and should be an active area of research, with computer science and programming environments leading the way.

## 6. Conclusion

This paper contributes to our understanding of the relationship between the design of introductory programming environments and the programming practices they engender. Using a detailed vignette and data from a five-week study, we show how modality affects novice programmers' emerging programming practices. In doing so, we develop the notion of modality as a means for describing the relationship between an interface and learner and highlight modality as one possible design dimension that can be used to support novices in having early programming successes. This work is intended to complement other work focused on conceptual learning and attitudinal and engagement outcomes and help us think through the relationship between design and learning, specifically as it related to programming. Given the increased role of computer science and the growing number of introductory environments being developed and used in classrooms, having a complete picture of how these design choices impact novices is essential. The ultimate goal of this line of inquiry is that it will help shape the next generation of introductory computer science learning environments, and in doing so, shape the next generation of computationally literate students.

## 7. References

[1] C. Duncan, T. Bell, S. Tanimoto, Should Your 8-year-old Learn Coding?, in: Proc. 9th Workshop Prim. Second. Comput. Educ., ACM, New York, NY, USA, 2014: pp. 60–69.
[2] J.J. Gibson, The ecological approach to visual perception, Psychology Press, 1986.
[3] D.A. Norman, The design of everyday things, Doubleday, New York, 1990.
[4] P. Ginns, Meta-analysis of the modality effect, Learn. Instr. 15 (2005) 313–331. doi:10.1016/j.learninstruc.2005.07.001.
[5] S.E. Palmer, Fundamental aspects of cognitive representation, in: E. Rosch, B.B. Lloyd (Eds.), Cogn. Categ., Lawrence Erlbaum Associates, Hillsdale, N.J., 1978: pp. 259–303.
[6] J.J. Kaput, Towards a Theory of Symbol, in: C. Janvier (Ed.), Probl. Represent. Teach. Learn. Math., Lawrence Erlbaum Associates, Hillsdale, NJ, 1987: p. 159.
[7] J.H. Maloney, M. Resnick, N. Rusk, B. Silverman, E. Eastmond, The Scratch programming language and environment, ACM Trans. Comput. Educ. TOCE. 10 (2010) 16.
[8] T.D. Erickson, Working with interface metaphors, in: Read. Human–Computer Interact., Elsevier, 1995: pp. 147–151.
[9] D. Bau, J. Gray, C. Kelleher, J. Sheldon, F. Turbak, Learnable programming: blocks and beyond, Commun. ACM. 60 (2017) 72–80.
[10] A. Begel, LogoBlocks: A graphical programming language for interacting with the world., Electrical Engineering and Computer Science Department. MIT, 1996.
[11] J. Bonar, B.W. Liffick, A visual programming language for novices, in: S.K. Chang (Ed.), Princ. Vis. Program. Syst., Prentice-Hall, Inc., 1987.
[12] S. Cooper, W. Dann, R. Pausch, Alice: a 3-D tool for introductory programming concepts, J. Comput. Sci. Coll. 15 (2000) 107–116.
[13] M. Resnick, B. Silverman, Y. Kafai, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, Scratch: Programming for all, Commun. ACM. 52 (2009) 60.

[14] D. Wolber, H. Abelson, E. Spertus, L. Looney, App Inventor: Create Your Own Android Apps, O’Reilly Media, Sebastopol, Calif, 2011.
[15] M.H. Wilkerson-Jerde, U. Wilensky, Restructuring Change, Interpreting Changes: The DeltaTick Modeling and Analysis Toolkit, in: J. Clayson, I. Kalas (Eds.), Proc. Constr. 2010 Conf., Paris, France, 2010.
[16] M.S. Horn, U. Wilensky, NetTango: A mash-up of NetLogo and Tern, in: Moher T Chair Pinkard N Discussant Syst. Collide Chall. Oppor. Learn. Technol. Mashups, Vancouver, British Columbia, 2012.
[17] D. Weintrop, U. Wilensky, RoboBuilder: A program-to-play constructionist video game, in: C. Kynigos, J. Clayson, N. Yiannoutsou (Eds.), Proc. Constr. 2012 Conf., Athens, Greece, 2012.
[18] N. Fraser, Blockly, Google, https://developers.google.com/blockly/, 2013.
[19] R.V. Roque, OpenBlocks: An extendable framework for graphical block programming systems, Master’s Thesis, Massachusetts Institute of Technology, 2007.
[20] D. Bau, D.A. Bau, M. Dawson, C.S. Pickens, Pencil Code: Block Code for a Text World, in: Proc. 14th Int. Conf. Interact. Des. Child., ACM, New York, NY, USA, 2015: pp. 445–448.
[21] M. Homer, J. Noble, Combining Tiled and Textual Views of Code, in: IEEE Work. Conf. Softw. Vis. VISSOFT, IEEE, Victoria, BC, 2014: pp. 1–10.
[22] Y. Matsuzawa, T. Ohata, M. Sugiura, S. Sakai, Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment, in: Proc. 46th ACM Tech. Symp. Comput. Sci. Educ., ACM Press, 2015: pp. 185–190.
[23] D. Weintrop, N. Holbert, From blocks to text and back: Programming patterns in a dual-modality environment, in: Proc. 48th ACM Tech. Symp. Comput. Sci. Educ., ACM, New York, NY, USA, 2017.
[24] M. Kölling, N.C.C. Brown, A. Altadmri, Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming, in: Proc. Workshop Prim. Second. Comput. Educ., ACM, New York, NY, USA, 2015: pp. 29–38.
[25] R.B. Shapiro, M. Ahrens, Beyond Blocks: Syntax and Semantics, Commun ACM. 59 (2016) 39–41.
[26] D. Weintrop, U. Wilensky, The challenges of studying blocks-based programming environments, in: 2015 IEEE Blocks Workshop Blocks Beyond, 2015: pp. 5–7.
[27] D.J. Malan, H.H. Leitner, Scratch for budding computer scientists, in: ACM SIGCSE Bull., ACM, 2007: pp. 223–227.
[28] J.H. Maloney, K. Peppler, Y. Kafai, M. Resnick, N. Rusk, Programming by choice: Urban youth learning programming with Scratch, ACM SIGCSE Bull. 40 (2008) 367–371.
[29] B. Tangney, E. Oldham, C. Conneely, S. Barrett, J. Lawlor, Pedagogy and processes for a computer programming outreach workshop—The bridge to college model, Educ. IEEE Trans. On. 53 (2010) 53–60.
[30] A. Wilson, D.C. Moffat, Evaluating Scratch to introduce younger schoolchildren to programming, Proc. 22nd Annu. Psychol. Program. Interest Group Univ. Carlos III Madr. Leganés Spain. (2010).
[31] O. Meerbaum-Salant, M. Armoni, M.M. Ben-Ari, Learning computer science concepts with Scratch, in: Proc. Sixth Int. Workshop Comput. Educ. Res., 2010: pp. 69–76.

[32] O. Meerbaum-Salant, M. Armoni, M. Ben-Ari, Habits of programming in Scratch, in: Proc. 16th Annu. Jt. Conf. Innov. Technol. Comput. Sci. Educ., ACM, Darmstadt, Germany, 2011: pp. 168–172.
[33] S. Grover, R. Pea, S. Cooper, Designing for deeper learning in a blended computer science course for middle school students, Comput. Sci. Educ. 25 (2015) 199–237.
[34] D. Franklin, G. Skifstad, R. Rolock, I. Mehrotra, V. Ding, A. Hansen, D. Weintrop, D. Harlow, Using Upper-Elementary Student Performance to Understand Conceptual Sequencing in a Blocks-based Curriculum, in: Proc. 2017 ACM SIGCSE Tech. Symp. Comput. Sci. Educ., ACM, New York, NY, USA, 2017: pp. 231–236.
[35] K. Johnsgard, J. McDonald, Using Alice in Overview Courses to Improve Success Rates in Programming I, in: IEEE 21st Conf. Softw. Eng. Educ. Train. 2008 CSEET 08, 2008: pp. 129–136.
[36] B. Moskal, D. Lurie, S. Cooper, Evaluating the effectiveness of a new instructional approach, in: Proc. 35th SIGCSE Tech. Symp. Comput. Sci. Educ., 2004: pp. 75–79.
[37] P. Mullins, D. Whitfield, M. Conlon, Using Alice 2.0 as a first language, J. Comput. Sci. Coll. 24 (2009) 136–143.
[38] D.C. Cliburn, Student opinions of Alice in CS1, in: Front. Educ. Conf. 2008 FIE 2008 38th Annu., IEEE, 2008: p. T3B–1.
[39] K. Powers, S. Ecott, L.M. Hirshfield, Through the looking glass: teaching CS0 with Alice, ACM SIGCSE Bull. 39 (2007) 213–217.
[40] W. Dann, S. Cooper, R. Pausch, Learning to Program with Alice, Prentice Hall Press, 2011.
[41] D. Weintrop, U. Wilensky, Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs, in: Proc. Elev. Annu. Int. Conf. Int. Comput. Educ. Res., ACM, New York, NY, USA, 2015: pp. 101–110.
[42] T.W. Price, T. Barnes, Comparing Textual and Block Interfaces in a Novice Programming Environment, in: ACM Press, 2015: pp. 91–99.
[43] M. Homer, J. Noble, Lessons in Combining Block-based and Textual Programming, J. Vis. Lang. Sentient Syst. 3 (2017) 22–39. doi:10.18293/VLSS2017.
[44] D. Weintrop, U. Wilensky, Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments, in: Proc. 2017 Conf. Interact. Des. Child., ACM, New York, NY, USA, 2017: pp. 183–192.
[45] M. Kölling, N.C.C. Brown, A. Altadmri, Frame-Based Editing, J. Vis. Lang. Sentient Syst. 3 (2017) 40–67.
[46] T.W. Price, N.C. Brown, D. Lipovac, T. Barnes, M. Kölling, Evaluation of a Frame-based Programming Editor, in: Proc. 2016 ACM Conf. Int. Comput. Educ. Res., ACM, 2016: pp. 33–42.
[47] J. Mönig, Y. Ohshima, J. Maloney, Blocks at your fingertips: Blurring the line between blocks and text in GP, in: 2015 IEEE Blocks Workshop Blocks Beyond, 2015: pp. 51–53.
[48] A. Stead, A.F. Blackwell, Learning Syntax as Notational Expertise when using DrawBridge, in: Proc. Psychol. Program. Interest Group Annu. Conf. PPIG 2014, University of Sussex, 2014: pp. 41–52.
[49] E. Hutchins, How a cockpit remembers its speeds, Cogn. Sci. 19 (1995) 265–288.
[50] D.A. Norman, Things that make us smart: Defending human attributes in the age of the machine, Basic Books, 1993.
[51] R. Noss, C. Hoyles, Windows on mathematical meanings: Learning cultures and computers, Kluwer, Dordrecht, 1996.

[52] J. Hollan, E. Hutchins, D. Kirsh, Distributed cognition: toward a new foundation for human-computer interaction research, ACM Trans. Comput.-Hum. Interact. TOCHI. 7 (2000) 174–196.
[53] J. Kaput, R. Noss, C. Hoyles, Developing new notations for a learnable mathematics in the computational era, Handb. Int. Res. Math. Educ. (2002) 51–75.
[54] U. Wilensky, S. Papert, Restructurations: Reformulating knowledge disciplines through new representational forms, in: J. Clayson, I. Kallas (Eds.), Proc. Constr. 2010 Conf., Paris, France, 2010.
[55] D. Weintrop, U. Wilensky, To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming, in: Proc. 14th Int. Conf. Interact. Des. Child., ACM, New York, NY, USA, 2015: pp. 199–208.
[56] D. Weintrop, Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms, Ph.D. Dissertation, Northwestern University, 2016.
[57] D. Weintrop, U. Wilensky, Comparing Blocks-based and Text-based Programming in High School Computer Science Classrooms, ACM Trans. Comput. Educ. TOCE. (In Press).
[58] C.M. Lewis, How programming environment shapes perception, learning and goals: Logo vs. Scratch, in: Proc. 41st ACM Tech. Symp. Comput. Sci. Educ., New York, NY, 2010: pp. 346–350.
[59] U. Wilensky, A. Papert, B. Sherin, A.A. DiSessa, A. Kay, S. Turkle, Center for Learning and Computation-Based Knowledge (CLiCK), Proposal to the National Science Foundation - Science of Learning Center., 2005.