# Starting from Scratch: Outcomes of Early Computer Science Learning Experiences and Implications for What Comes Next

David Weintrop
University of Maryland
College Park, MD, USA
weintrop@umd.edu

Alexandria K. Hansen
UC Santa Barbara
Santa Barbara, CA, USA
akhansen@ucsb.edu

Danielle B. Harlow
UC Santa Barbara
Santa Barbara, CA, USA
dharlow@education.ucsb.edu

Diana Franklin
University of Chicago
Chicago, IL, USA
dmfraklin@uchicago.edu

## ABSTRACT

Visual block-based programming environments (VBBPEs) such as Scratch and Alice are increasingly being used in introductory computer science lessons across elementary school grades. These environments, and the curricula that accompany them, are designed to be developmentally-appropriate and engaging for younger learners but may introduce challenges for future computer science educators. Using the final projects of 4th, 5th, and 6th grade students who completed an introductory curriculum using a VBBPE, this paper focuses on patterns that show success within the context of VBBPEs but could pose potential challenges for teachers of follow-up computer science instruction. This paper focuses on three specific strategies observed in learners' projects: (1) wait blocks being used to manage program execution, (2) the use of event-based programming strategies to produce parallel outcomes, and (3) the coupling of taught concepts to curricular presentation. For each of these outcomes, we present data on how the course materials supported them, what learners achieved while enacting them, and the implications the strategy poses for future educators. We then discuss possible design and pedagogical responses. The contribution of this work is that it identifies early computer science learning strategies, contextualizes them within developmentally-appropriate environments, and discusses their implications with respect to future pedagogy. This paper advances our understanding of the role of VBBPEs in introductory computing and their place within the larger K-12 computer science trajectory.

## CCS CONCEPTS

• Social and professional topics → Professional topics → Computing Education


*ICER '18, August 13–15, 2018, Espoo, Finland*
 ACM ISBN 978-1-4503-5628-2/18/08…$15.00
https://doi.org/10.1145/3230977.3230988

## KEYWORDS

Elementary Computer Science Education; Introductory Computing Curricula; Block-based programming; Learning

**ACM Reference format:**

D. Weintrop, A. K. Hansen, D. B. Harlow, and D. Franklin. 2018. In Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18). ACM, New York, NY, USA, 21-29. DOI: https://doi.org/10.1145/3230977.3230988

## 1 INTRODUCTION

The call to bring computer science (CS) to all learners has reached a roar as districts, states, and countries around the world are increasingly making CS part of the school experience for learners across the K-12 spectrum. While there exists a diversity of languages, programming environments, and curricula for the oldest K-12 learners, a narrower set of introductory experiences exist for younger students. In elementary school (grades K-8, ages 5-13), CS instruction is largely being taught using visual block-based programming environments (VBBPEs) like Scratch [39] and Alice [9]. Curricula including Creative Computing [6] and the K-8 code.org materials utilize VBBPEs. VBBPEs are popular due to the affordances they provide young learners. Transitioning learners from introductory learning experiences with VBBPEs to more conventional text-based programming environments poses challenges to educators and curriculum designers. The very features of VBBPEs that allow novice learners to be successful may present challenges to future educators, requiring them to more closely consider learners' previous experiences and potentially modify instructional strategies to support learners as they progress.

It is these considerations that we explore in this paper, specifically with the goal of understanding outcomes of using VBBPEs with elementary learners and how this decision should inform subsequent curriculum design and pedagogy. More specifically, we answer the following research questions:

> *What are examples of strategies that learners develop through introductory experiences with VBBPEs that future educators should be aware of? How and when might these strategies differ from what is taught in subsequent classes where text-based programming languages are used?*

To begin to answer these questions, we draw on data from a classroom implementation of a VBBPE and accompanying curriculum. Specifically, we present data on three outcomes of students learning CS with a VBBPE that have potential implications for future instruction: 1) wait blocks being used to manage program execution in two distinct ways, (2) the use of event-based programming strategies to produce parallel outcomes, and (3) the coupling of taught concepts to curricular presentation. For each of these strategies, we present data on how the course materials supported these outcomes, what learners achieved through enacting them, and discuss potential design and pedagogical responses. The goal of this work is to advance our understanding of what learners are able to achieve in a developmentally-appropriate introductory course with a VBBPE and to consider what the implications of these outcomes are for future CS instruction that moves beyond VBBPEs. This paper begins with a review of relevant work before presenting the LaPlaya environment and KELP-CS curriculum. We then present the three strategies and discuss implications of this work.

## 2 PRIOR WORK

### 2.1 Visual Block-based Programming Environments

In this paper, we use the term visual block-based programming environment (VBBPE) to capture the set of programming tools that introduce learners to programming through a block-based interface and have a visual execution environment (e.g. sprites on a stage). This type of environment is exemplified by Scratch [39], Alice [9], and Pencil Code [3]. Numerous widely-used introductory environments do not meet this definition of a VBBPE as they only include some of the defining VBBPE features, like MIT App Inventor's [54] use of block-based programming and Greenfoot's use of sprite-like actors [26]. In this section, we discuss three key features of VBBPEs that are pertinent to this study.

The first key component of VBBPEs is the use of a block-based programming interface that leverages a programming-primitive-as-puzzle-piece metaphor to provide visual cues to the user about how and where blocks can be used [4, 30]. Users compose programs in these environments by 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 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 use, there are other visual cues to help programmers, including color coding by conceptual use and nesting of blocks to denote scope [30, 47, 53].

A second key characteristic of VBBPEs is the notion of a Sprite – an on-screen, two-dimensional character that follows programming instructions defined by the user. The sprite can be viewed as the modern incarnation of Logo's turtle [36]. In discussing the development and role of the Turtle, Papert invokes Piaget's notion of a mother structure – an intellectual construct from which concepts can be created. In the Turtle, Papert saw the embodiment of differential geometry in a way that could be anthropomorphized by the learner [35]. While the Sprite can still be used towards these mathematical ends, increasingly it's role is as a computational mother structure, i.e. a means to develop foundational computational ideas. As we will argue in this paper, whereas the path one follows using the turtle as the means to express differential geometry concepts has been mapped [1], it is less clear what path one follows when moving from sprite-driven programming towards more advanced computational ideas that may not be executed visually.

A third central feature of VBBPEs is their support for open-ended and exploratory programming activities. This feature draws directly from the Constructionist design principle of being "discovery rich" [35]. Scratch and other VBBPEs accomplish this by providing an accessible and intuitive set of programming blocks but little in the way of constraints with respect to how they can or should be used. Through designing a platform for open-ended exploratory activities, VBBPEs do not prescribe specific practices, instead supporting an epistemological pluralism [49] that does not favor one specific program approach or one type of project.

### 2.2 Computer Science in Elementary School

In the last decade, bringing CS to K-8 has become more widespread, facilitated by programming tools designed for young learners [12, 25]. Early work on programming as a means for learning conducted by Papert and colleagues with the Logo language found that programming was accessible to younger learners and could serve as a powerful learning practice [20, 34, 36]. Following these successes, much of the curricular and programming environment design effort has employed Constructionist design principles, foregrounding learning-by-doing and learner-directed activities. This can be seen in growing library of curricula designed for elementary learning, including: Creative Computing [6], Foundations for Advancing Computational Thinking [17], Animal Tlatoque [13], and the KELP-CS curriculum [23]. There are also growing online communities where classroom activities designed for elementary students are curated and shared, like the ScratchEd website and the CS for All Consortium, which includes over 100 organizations that self-identify as content providers for elementary learners. Code.org also offers nine distinct CS courses for students across grades K-8 (ages 5-13), including both conventional computer-based curricula as well as offline activities based on the CS unplugged curriculum [5] and computing activities designed for science classrooms based on Project GUTS [38]. Collectively, these resources capture part of the quickly expanding ecosystem of ways that CS is being introduced into elementary education.

### 2.3 Research on Learning In VBBPEs

A growing body of research is investigating how block-based programming shapes learners' conceptual understanding of CS concepts and emerging programming strategies. For example, researchers have documented a number of 'habits' of programming learners develop while working in block-based tools, such as an emphasis on bottom-up programming where

learners focus on using specific blocks [33]. Other strategies investigated include documenting how learners at different ages design for their audiences [19] and debugging strategies and the requisite knowledge to implement them [28]. Further work has documented programming strategies specific to VBBPEs, looking at how the scaffolds present in the environment support unique patterns of interaction [52]. Likewise, a growing body of research is documenting how novices learn with block-based tools; identifying misconceptions learners may develop in VBBPEs and developmentally-appropriate content for learners [15, 16, 42]. For example, research looking at learners' emerging understanding of the initialization of state and variables in VBBPEs identified four distinct conceptual components of the topic (e.g. when to initialize) and showed how they are differentially manifested in VBBPEs compared to conventional text-based languages [14]. The findings presented herein build on and complement this work by continuing to fill in our understanding of what it means for young learners to develop foundational understandings of computational ideas in VBBPEs. Likewise, our analysis considers if and how ideas and strategies developed in VBBPEs do or do not relate to future instruction and learning in conventional text-based languages.

## 3 METHODS

The work presented in this paper is part of a larger, design-based research study focusing on the creation of elementary CS classroom materials. We begin this section by presenting the LaPlaya VBBPE and KELP-CS curriculum. We then present details on the participants and study design before concluding the section discussing the data collected and analytic approach used.

### 3.1 Materials

LaPlaya (Fig. 1) is a VBBPE built on top of the Snap! programming environment [22]. Like Scratch, students program via a drag-and-drop interaction, producing scripts of blocks to control on-screen sprites. LaPlaya is designed to support both guided and open-ended exploration for upper elementary school students (grades 4-6; ages 8-12). To help make programming more accessible to younger learners, LaPlaya includes a number of unique pedagogical scaffolds. For example, when introducing new concepts, students are provided with pre-programmed and locked scripts (Fig. 2a) at the beginning of the new activity. These scripts are visible and accompanied by text descriptions to serve as examples. LaPlaya also includes white, inert scripts (Fig. 2b) that are not executable and serve as templates of how blocks can be used to accomplish a desired outcome.

LaPlaya's blocks were also modified with respect to the original Snap! language in order to remove more advanced mathematical concepts such as percentages, negative numbers, and decimals for our younger students (ages 9-10). In addition, to support learners at varying reading levels, LaPlaya has an audio, read-aloud function so task instructions can be heard. This is particularly important for English language learners. For additional information about LaPlaya and the modifications made to make it more accessible to novice learners, see [21, 23].

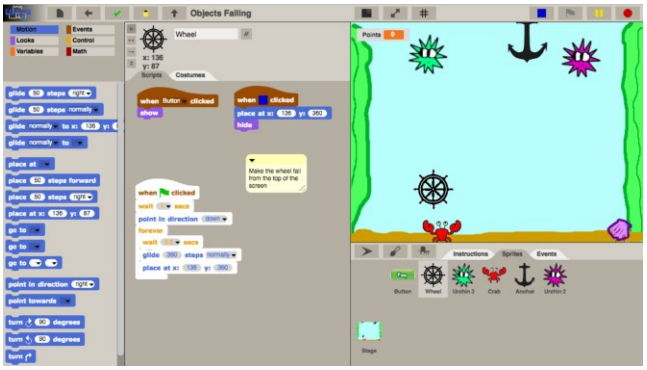


**Figure 1. The LaPlaya programming environment.**


**(a)** **(b)**

**Figure 2. LaPlaya scaffolds: (a) Predefined, locked scripts with textual hints and (b) inert scripts used as templates.**

The KELP-CS curriculum was designed for the LaPlaya programming environment with the goal of providing a developmentally-appropriate introduction to foundational CS concepts. KELP-CS consists of a predefined sequence of modules comprised of activities that gradually introduce CS concepts and the associated blocks to students as they progress. KELP-CS includes both these structured tasks with specific conceptual learning objectives as well as an open-ended play area with the module's full set of blocks to keep more advanced learners engaged and provide a space for learner-directed exploration throughout the curriculum. The KELP-CS curriculum has two main types of activities: 1) On-computer assignments and 2) "unplugged" activities completed away from a computer. On-computer activities consisted of small, incremental tasks designed to move students to higher levels of programming sophistication. Unplugged activities were modeled after CS-Unplugged [5] with the goal of connecting computing to students everyday lives. Modules culminate with open-ended projects.

KELP-CS features two main curricular modules, each designed around a different theme: 1) Digital Storytelling and 2) Game Design. Each module is designed to be completed in approximately 15-18 hours of instruction. The modules are meant to be completed sequentially. The first Module (Digital Storytelling) covers the following concepts: Sequencing, Breaking down actions, Events, Initialization, Animating sprites, and Changing scenes. The second module (Game Design) continues with Broadcasting messages, Loops, Conditional logic, and Variables. Both modules culminated with an open-ended programming activity, allowing the learner to employ the concepts learned throughout the unit.

### 3.2 Participants and Study Design

The data for this paper are drawn from one school located in the Southwestern United States where 4th, 5th, and 6th grade students (ages 9-12) worked through the KELP-CS curriculum over the course of two consecutive school years. In the first year, two classes in each grade, totaling 44 fourth graders, 48 fifth graders and 43 sixth graders, completed the first KELP-CS module (Digital Story Telling). The second year consisted of a single fifth grade class of 18 students who completed the second KELP-CS module (Game Design). This resulted in a total of 135 unique students over the two years. The school where the study was conducted is racially diverse (54% White, 35% Hispanic or Latino, 5% Asian, and 3% Black or African American) with approximately 31% of students coming from economically disadvantaged households and 16% of students schoolwide designated as English-language learners.

Each year, students spent roughly 1 hour per week for 15 weeks working in the KELP-CS curriculum. Each class session was observed by researchers and video recorded for later analysis. Additionally, the LaPlaya VBBPE was modified so as to automatically collect student projects, which serves as the primary data source for the analysis presented in this paper.

### 3.3 Analytic Approach

The analysis presented in this paper focuses on the 135 summative programs authored by the participants in this study. Our decision to focus on student-authored programs stems from the constructivist learning orientation and the constructionist design philosophy we bring to this work. Constructivist learning theory posits that new knowledge is built through the processes of assimilation and accommodation with learners' existing knowledge. In Constructionist learning environments, that understanding is manifest through the artifacts built by learners, which in this case, are the programs authored [34]. As such, we use these constructed artifacts as a means to gain insight into learners' emerging understanding of CS concepts and look at how concepts are used within the larger program to understand emerging programming strategies.

To analyze this data, each program was statically analyzed using a custom-written script to catalog its contents with respect to type and frequency of blocks used. Next, we undertook a grounded-theory approach [44] with one researcher analyzing each program individually looking for evidence of strategies, patterns, or emerging programming strategies. This initial set of strategies was presented to the larger research team, who further refined the defining characteristics of each strategy and created a qualitative coding manual to describe the usages. The coding manual was then applied to the full set of programs to understand the frequency of each pattern. This approach allows us to situate each pattern within the curriculum as a way to help us understand potential implications with respect to pedagogy and future CS learning.

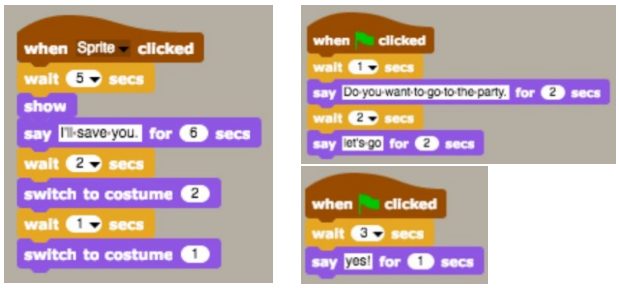
## 4 FINDINGS

This section presents three outcomes of learners as expressed in their culminating projects following the completion of modules in KELP-CS. The analysis focuses on emerging learner strategies that are developmentally-appropriate and productive within the context of the KELP-CS curriculum and have potential implications for how teachers of subsequent classes that use traditional programming languages design instruction. As such, educators should be aware of these strategies and think about how best to productively utilize them to scaffold and support learners as they progress in their CS learning careers. For each strategy, we first present student data demonstrating the strategy in use and documenting its frequency across the participant pool. We then link the strategy to features of KELP-CS or LaPlaya and discuss how the environment productively supports the strategy as well as future challenges that may emerge. We conclude each section discussing ways future educators can respond to it.

It bears repeating that the goal of this work is to identify strategies students develop in VBBPEs that are *different from* those conventionally used in non-VBBPE introductory CS instruction. This is not meant to imply that VBBPEs are inappropriate for introductory computing contexts, instead, we seek to advance our understanding of how best to support learners as they progress along a CS learning trajectory.

### 4.1 Managing Execution with Wait Blocks

A frequent goal of programs in VBBPE is to coordinate a series of on-screen events such as the speed at which a sprite dances or how and when two sprites interact. Achieving this coordination requires the learner to define specific instructions in their programs. One common way to achieve these behaviors is to manually control the speed of the execution of scripts using the wait block. Seventy-four participants (54.8%) used the wait block at least once in their final projects, while 13 (9.6%) used it more than 10 times. In our analysis, we found this strategy employed in two distinct forms: intra-sprite delays and inter-sprite synchronization, both of which are accomplished using the wait block.




(a) (b)

**Figure 3. Two examples of students using wait block to control the execution of their programs**

4.1.1 *Intra-script Delays.* As part of their final projects, students often wanted to slow down the execution of a script for a single sprite and used the wait block to accomplish this goal. Fig. 3a shows a student project that demonstrates an intra-script delay. In this case, the wait blocks added between the costume change blocks are used for this purpose. The result is that the sprite's appearance changes at the specified rate. This strategy was used

as both a storytelling mechanism and a way to animate sprites. Of the 135 projects analyzed, 58 of them (43%) utilized this strategy to control the speed of execution within a script.

*4.1.2 Inter-Sprite Synchronization with Wait Blocks.* The second use of the wait block to control program execution was to coordinate the timing of actions between sprites. Fig. 3b shows two scripts from a student project that implements a conversation between two sprites. Because the synchronization spans two sprites, the scripts that define this synchronization cannot be viewed on screen at the same time. This means authoring and debugging programs that use this strategy relies heavily on interpreting the outcome of the visual execution environment. Across the full set of participants, 37.8% of final projects included this type of intra-sprite coordination.

*4.1.3 Introductory Benefits and Potential Future Challenges.* In these two examples, we see how young learners take advantage of the access that VBBPEs give to manipulate how and when scripts are run. Through the use of the wait block, novice programmers were able to create animations and achieve synchronization through parallelization of their programs, both of which are outcomes that would come much later in a conventional, non-VBBPE-based CS instructional sequence. However, these strategies do introduce potential mismatches with future CS instruction. For example, in manipulating execution speed with the wait block, learners both manually control the rate of execution of a program and use that control to slow down the rate of execution. These are reasonable for this context as doing so makes animations clearer and sets the pace of sprite interactions such that the user has time to interpret what is happening on the screen.

However, the characteristic of wanting to control the speed of execution is rarely a goal in early text-based instruction found in K-12 classrooms. High school courses taught in languages like Java, Python, or JavaScript rarely include animations or ask students to control the rate at which things happen, instead, the focus is on non-temporal aspects of programming (like algorithms, sequencing, state, etc.). Further, when time is considered in most introductory text-based programming instruction, the goal is to *speed up* execution time, not slow it down. These strategies suggest to learners that the speed at which computers perform tasks can be easily predicted or controlled.

Additionally, the use of wait blocks to manually and explicitly control timing to achieve parallelization is quite distinct from the parallel programming approaches students might encounter early in text-based programming instruction. This fact can be seen in the design of many VBBPEs directly as many include message passing and broadcasting features to achieve parallel outcomes. Previous work looking at how parallel outcomes are achieved found that students were substantially less likely to use this mechanism than the simpler wait blocks [19]. This finding is replicated by this work as only 16 students used this feature of the VBBPE. Because of the age of the students, the simplicity of wait blocks, and the relative predictability when on a single machine, it is appropriate that students solve problems with wait blocks.

4.1.4 *Considerations for Future Educators and Designers.* Educators teaching a class comprised of students that recently completed a course using a VBBPE should be aware of the strategies their students may have developed related to the wait block and other temporal blocks, such as say for. Learners may begin to think that speed is a characteristic of the computer that is meant to be programmatically manipulated, alongside aspects like sequential flow and program state. While manually controlling when instructions are evaluated or focusing on speed or timing of a piece of code is an authentic programming strategy, temporal characteristics of programming (such as optimization or parallel computing) usually occur much later in CS instruction. In terms of how this affects educators, the first step is raising awareness of the difference between programming *when* an event occurs versus *how* it occurs. As more students enter second and third computing curricula with prior experience in VBBPEs, teachers may want to include explicit instruction on the temporal dimension of the programs being authored and attend to students' potential desire to pursue solutions that seek to manipulate the speed at which instructions execute as a means to achieve a desired outcome.

Future educators should also be aware of the implication of learners coordinating parallel execution with wait blocks. Using wait block to control the behavior of the program connotes the idea that each object has an internal clock that controls how and when it operates, and that there is a shared universal clock on which they can rely for timing. When learners do eventually encounter synchronization in parallel systems, they will need to be explicitly taught about absolute timing and the assumptions that can and cannot be made based on the technologies and tools being used. This again ties back to the larger theme of deemphasizing *when* commands execute, instead focusing learners' attention on *how* they are used.

## 4.2 Coordination with Event-based Programming

Visual block-based programming environments often employ an event-based programming approach. In this paradigm, to run a program, you associate a sequence of blocks with an action, be it clicking the green flag (akin to a start button), waiting for an in-program event (like receiving a message), or binding a script to a key press. Events are an intuitive and accessible way to engage novices and younger learners with programming and were very common in student final projects. Students used an average of 13.5 events blocks (*SD* 20.8) per project, with 27 students using more than 20 event blocks, and 6 students defining more than 50 events.

Event-driven programming makes it easy to create interactive programs. It also gives the programmer direct control over how and when behaviors in their programs are run. In addition, it allows a programmer to think and program separately about what should happen at different points in the program, reducing the length of any piece of code. In this way, it helps achieve the low-threshold to programming sought by the designers of VBBPEs and contributes to the engagement and enjoyment of the environment [30].

*4.2.1 Introductory Benefits and Potential Future Challenges.* The inclusion of a suite of event blocks (such as when key pressed and when sprite clicked) gives the learner a number of intuitive hooks for inserting programmed behavior. This approach is different

from many general-purpose programming languages, in which early instruction often focuses on a single main function (often called main) that is called that begins the serial execution of the program. Event-based programming provides a pair of introductory benefits that may turn into potential challenges for future educators.

The first outcome of introducing learners to programming in event-based VBBPEs is tied to the fact that event-based programming environments are inherently parallel. In VBBPEs, multiple sprites can operate in parallel in response to the same event, or a single sprite can perform two tasks in parallel in response to the same event. These parallel programming capabilities are present in the text-based languages that learners may transition to but students are unlikely to encounter these features until later in their CS education paths. Looking across the full set of projects, 106 of the 135 student-created projects utilized concurrency by having multiple scripts linked to the same event.

A second potential outcome from learning to program in event-driven programming is developing habits that are unique to the event-based paradigm and do not have natural analogs in conventional text-based programming languages. For example, students can bind multiple scripts to the same event for the same sprite *even though the events are not intended to execute in parallel.* Fig. 4 depicts an example of this found in a final project showing four of the 12 scripts the student defined for the when left arrow key pressed event of a single sprite. In composing these blocks, the learner directly mapped an event with multiple actions. Conceptually, this both makes sense and is an intuitive approach to achieving a behavior such as making multiple things happen after a single key press. However, this also circumvents the need to define the steps of the program sequentially in a single script. While this is a functional solution, it is not how the same outcome would be achieved in a non-event-based context. A total of 36 final projects included parallel implementations of serial behaviors, which suggests this is a relatively common occurrence and something educators should be made aware of. This distinction is meaningful because if all the commands shown in Fig. 4 were moved into a single script, the numerical values in the wait block would need to change, meaning the shift is not just reorganizing commands, but the underlying logic needs to be modified as well.

*4.2.2 Considerations for Future Educators and Designers.* The program shown in Fig. 4 is one example of the more general outcome of learners developing programming strategies that

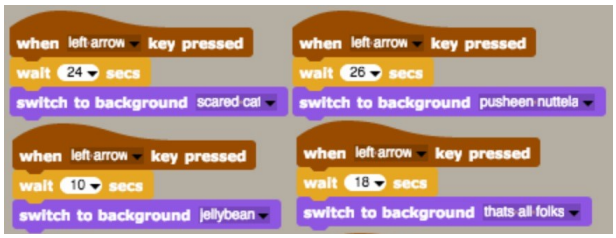



**Figure 4. Four of the 12 when left arrow key pressed scripts defined in one students' final project.**

leverage features of event-based programming. This is to be expected of novices with little prior experience and shows how they take advantage of affordances present in VBBPEs. This finding suggests that teachers of more advanced courses should be aware of and prepared to help students move from the parallel thinking supported by events toward the linear, sequential ordering of commands imposed by the languages used in later instruction.

## 4.3 Coupling Concepts with Specific Contexts

The strategy used to introduce new CS concepts to elementary learners in the KELP-CS curriculum was to situate the concept in a specific context and provide scaffolds to facilitate learners in writing a program to use the concept in a specific way. This strategy serves as the first step in the Use-Modify-Create pedagogical strategy common to introductory computational thinking instruction [27]. KELP-CS had novices first *Use* a new concept, then provided opportunities to incorporate the concept into later programs, either in the same role but different context (*Modify*) or in new ways altogether (*Create*). For example, to introduce learners to conditional logic (the if block), the KELP-CS curriculum helped learners create a maze game, using conditional logic to make sure the player's sprite did not walk through any walls or touch any of the obstacles along the way. In a different game, the if block was used to detect when two sprites touched. In both cases, an if block with a touching block inside it was used to implement collision detection in the game. Fig. 5a shows how these blocks were first introduced to learners through LaPlaya's inert blocks feature. The thing to note about this example is the pattern of nesting the if block inside a forever loop and using the touching block as the test condition of the if block.

To understand the relationship between the way concepts were introduced in the KELP-CS curriculum and how they were used in students' open-ended final project, we developed an analytic coding scheme to situate the use of a concept within the Use-Modify-Create trajectory. In this analytic scheme, concepts can be presented in students' final projects at four levels of sophistication delineated based on their similarity to how the concept was introduced in the curriculum. First, the concept could be absent (as was the case for conditional logic in 6 of the 18 Module 2 projects). Second, the *Use* level of sophistication describes instances where learners use the concept in their final projects in the exact same role as it was used within the lesson. Meaning the use of the forever, if and touching blocks matches the structure shown in Fig. 5a.

The third level of expertise is illustrated by Fig. 5b. In this case, the student applied conditional logic in the same way (i.e. as a mechanism for detecting collisions) and programmed it with the same general structure (an if block nested in a forever block with a touching block as its test condition), but *Modified* its application to integrate other concepts such as a score, sprite placement, timing, and visibility. Two-thirds of the final projects (12 projects) demonstrated the ability to incorporate conditional logic in their programs in a role similar to how it was used in the curriculum. Finally, three students used conditional logic in a different way (beyond a mechanism for detecting collisions), demonstrating a *Create* level of understanding. These data show


(a) (b)

**Figure 5. (a) The first conditional logic template and (b) an example of how it was incorporated into a final project.**

how the Use-Modify-Create progression implemented in KELP-CS is developmentally-appropriate and helped learners at different levels, but also suggests the opportunity for educators to adjust pedagogical strategies to effectively build on these early successes.

*4.3.1 Introductory Benefits and Potential Future Challenges.* Firstly, it is important to note that in introductory contexts, any application of the concept in a learner-authored program should be viewed as a success. The strategy of students using code exactly as it was taught to them or as it exists in another project can help novices have programming successes early in their exposure to CS content. This form of remixing is a central strategy to computational thinking [7] and a common practice of programmers at all levels of expertise. Likewise, CS educators have argued that it is important to introduce concepts in context to help learners see the relevance and applicability of what is being learned [8, 41]. However, there is also research showing that students are more successful in learning when extraneous material (i.e. context) is absent, arguing the decontextualized presentation is more effective as it decreases the cognitive load associated with the learning task [32, 45]. Based on these seemingly contradictory findings, we came to the same conclusion as Guzdial [18], who concluded: "The only way to achieve decontextualized knowledge is to teach beyond a single context."

As such, in the KELP-CS curriculum, concepts were taught in multiple contexts. In the case of conditional logic in Module 2, that meant using the if block to create different styles of games. The goal in showing these two different applications of conditional logic was to help learners understand the underlying concept and see how it can be used in two distinct ways. However, the results show that it is unclear the degree to which the two contexts helped. In particular, it could have provided two concrete ways of using the concepts as opposed to providing a generalized understanding of the concept of conditional logic.

*4.3.2 Considerations for Future Educators and Designers.* While the Use-Modify-Create approach was pedagogically productive, our analysis suggests it does pose a potential challenge for future CS educators. By teaching concepts situated in a specific context, there is potential that the concept gets coupled to that particular use (e.g. for building games).

The data presented above, and the characterization of usages within the Use-Modify-Create progression, shows learners using the concept at all four levels of sophistication. There are two direct implications of this for future educators. First, educators should be aware of how concepts were presented and situated in early lessons so as to be able to present new and complementary uses of concepts. When a concept is presented in a new context, educators should also try and link the new presentation with prior contexts and implement bridging and hugging strategies suggested by research [11, 37].

Second, these data reinforce the fact that full understanding of a concept may require several courses in which concepts are taught in a variety of ways – educators targeting second or third experiences should not consider that "conditionals have already been covered" and assume full understanding. For a subsequent Scratch curriculum, in the case of conditional logic, this may mean using the construct for something other than collision detection or using it outside of a loop. Examples include comparing numerical values as part of a score-keeping mechanism or comparing x or y positions on the screen as to coordinate a dance across multiple sprites. Further, this is especially true when the new context is in a different language, modality, or environment. Doing so will further help students build conceptual bridges between the different forms of programming they will see throughout future CS instruction.

## 5 DISCUSSION

### 5.1 Preparing Expert Computer Science Teachers

One of the many challenges faced by districts and schools is recruiting, training, and retaining capable CS teachers. As the demand for CS across K-12 grows, so, too, does the need to bring new teachers into the discipline. One of the goals of this work is to show that when it comes to supporting learners as they progress through the K-12 CS trajectory, teachers should be aware of both previous and future CS courses to best support learners.

This paper documents three specific strategies that teachers charged with moving learners on to the next step in their CS careers should be prepared for. The strategies span curriculum design (defining new contexts to situate content), CS content knowledge (situations where learners focus on temporal aspects of execution), and CS pedagogical content knowledge (how different paradigms affect programming strategies). Collectively, this highlights the challenge that new CS teachers face. We make this point at the same time that one of the prevailing approaches to training new CS teachers is to provide them with a full suite of classroom materials (lessons, environments, assessments, etc.) to make it as easy as possible for them to get up and running quickly in their new discipline. While this is a prudent strategy given the immediate demand, the work presented above gives pause to the view that such an approach is sufficient for training teachers to support learners across the K-12 spectrum. Instead, our hope is this all-in-one approach for teachers serves as only the first step in the career-spanning undertaking of learning how best to support students in learning CS.

### 5.2 Giving Agency to the Learner

A second discussion point that relates to this work speaks to an emerging trend in the continually shifting landscape of introductory programming. Throughout this paper introductory programming environments have been treated as relatively static entities, i.e. they present a single interface and only support one form of interaction. In the case of VBBPEs, that means block-based programming in a sprite-driven context. This characterization is becoming less-and-less accurate with the emergence of new and more flexible programming environments. For example, dual-programming environments such as Pencil Code [3] and Tiled Grace [24] allow the learner to seamlessly transition back and forth between block-based and text-based modalities. Research is finding that this approach is useful for novice programmers [31, 50].

While the LaPlaya programming environment did provide a scaffolded programming interface that expanded as learners' knowledge and confidence grew in the form of introducing new blocks and categories, this shift was determined by the curriculum rather than the learner. It is easy to imagine what a learner-directed version of this form of scaffolding could look like where the learners themselves are free to decide how and when they want to see more blocks or simplify the programming environment. Scratch offers features similar to this in the form of extensions and Microwrolds [48]. Likewise, it is easy to imagine a curriculum that is designed to give the learner more agency in deciding how and when they progress, a feature that has been implemented by numerous online learning tools. The challenge with this approach is figuring out how to support individual agency while still ensuring shared content coverage across the classroom in order to ensure all learners are suitably prepared for future learning opportunities.

### 5.3 Choosing the Right Tools and Curricula for Your Classroom

One of the challenges of teaching CS is choosing the right curricula and programming languages and environments for your classroom. This is especially challenging for teachers with little or no prior CS experience. As this work highlights, features of the programming environment and the chosen curriculum both shape leaners emerging understandings of CS concepts and the programming strategies they develop. The challenge of picking the best tools and curricula for the classroom is magnified by the lack of a consensus on sequencing of CS concepts, guidelines for how in depth to cover concepts at different grade levels, and agreement in the community about what programming languages to use for instruction. These are problems that are actively being addressed both in the CS education research community [40, 43] as well as through wide-scale community initiatives to provide guidance to states, standards writers, and curriculum designers [10, 55]. When creating and choosing a curriculum and then deciding what environment (or environments) to accompany it in the classroom, it is important to make sure the two are aligned. At the same time, figuring out when, how or even if it is necessary to transition learners from environments that prioritize accessibility versus computational power and broad applicability, is another line ongoing work [2, 11]. There is also a growing body of research investigating VBBPEs and their affordances and drawbacks for elementary CS classrooms [23] and high school classrooms [51]. As this work progresses, hopefully, clarity will emerge as to how best to match environments with curricula and support teachers in effectively bringing them into their classrooms.

It is also important to note, that throughout this work, we have focused on the goal of conceptual learning of CS concepts. While this is an important goal, it is not always the focus of introductory classrooms, nor should it be. A second goal for introductory CS curricula, and one where VBBPEs have historically excelled, is getting learners excited about the field and changing their perceptions about what CS is and who can be a computer scientist [29, 39]. Getting learners interested, excited, and engaged with CS is potentially more important than conceptual learning, especially for young learners as it can change students views of potential future educational goals [46]. The take away from this discussion is the importance of having clear goals as an educator, and aligning those goals with the tools and curricula you choose. One contribution of this work is providing a deeper understanding of the consequences of this decision, especially if the goal for the course is to prepare students for future computer science learning.

## 6 CONCLUSION

Whereas computer science was once a subject reserved for the final years of high school and beyond, the subject has a growing presence across the K-12 spectrum. In response to the need for computing education in earlier grades, a growing ecosystem of novice programming environments and curricula has emerged. Increasingly, educators and curriculum designers are turning to VBBPEs to serve as the way novices are introduced to programming. While these environments have excelled in informal spaces, their transition to formal classrooms, and their use to prepare all learners for future CS instruction is not without its challenges. In this work, we sought to identify programming strat4gies novices developed while working in VBBPEs that are distinct from what is typically taught in text-based languages and then consider their implications for educators. In particular, using data from student-authored, open-ended summative projects, we show how novices' uses of the wait block, their coordination of execution through events, and the coupling of concepts to the contexts in which they were first introduced all have potential implications for future instruction. The contribution of this work is to deepen our understanding of the use of VBBPEs in the classroom early in learners' CS careers, especially when the goal is preparation for future CS instruction. In doing so, we shed light on open challenges we face as educators and advance the goal of creating effective, accessible CS learning experiences and bringing CS to all learners.

## ACKNOWLEDGMENTS

This work is supported by the National Science Foundation Awards CNS-1240985 and CNS-1738758. We would also like to thank all of the teachers, students, and schools involved in this project.

## REFERENCES

[1] Abelson, H. and diSessa, A.A. 1986. *Turtle geometry: The computer as a medium for exploring mathematics*. The MIT Press.

[2] Armoni, M., Meerbaum-Salant, O. and Ben-Ari, M. 2015. From Scratch to "Real" Programming. *ACM Transactions on Computing Education (TOCE)*. 14, 4 (2015), 25:1–15.

[3] Bau, D., Bau, D.A., Dawson, M. and Pickens, C.S. 2015. Pencil Code: Block Code for a Text World. *Proc. of the 14th International Conference on Interaction Design and Children* (New York, NY, USA, 2015), 445–448.

[4] Bau, D., Gray, J., Kelleher, C., Sheldon, J. and Turbak, F. 2017. Learnable programming: blocks and beyond. *Comm. of the ACM*. 60, 6 (2017), 72–80.

[5] Bell, T.C., Witten, I.H. and Fellows, M.R. 1998. *Computer Science Unplugged: Off-line activities and games for all ages*. Citeseer.

[6] Brennan, K. 2013. Learning computing through creating and connecting. *Computer*. 46, 9 (2013), 52–59.

[7] Brennan, K. and Resnick, M. 2012. New frameworks for studying and assessing the development of computational thinking. Paper Presented at the American Education Researchers Association Conference. (Vancouver, Canada, 2012).

[8] Cooper, S. and Cunningham, S. 2010. Teaching computer science in context. *ACM Inroads*. 1, 1 (2010), 5–8.

[9] Cooper, S., Dann, W. and Pausch, R. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*. 15, 5 (2000), 107–116.

[10] CSTA Standards Task Force 2016. K–12 Computer Science Standards.

[11] Dann, W., Cosgrove, D., Slater, D., Culyba, D. and Cooper, S. 2012. Mediated transfer: Alice 3 to Java. *Proc. of the 43rd ACM SIGCSE Technical Symposium on Computer Science Education* (2012), 141–146.

[12] Duncan, C., Bell, T. and Tanimoto, S. 2014. Should Your 8-year-old Learn Coding? *Proc. of the 9th Workshop in Primary and Secondary Computing Education* (New York, NY, USA, 2014), 60–69.

[13] Franklin, D., Conrad, P., Aldana, G. and Hough, S. 2011. Animal tlatoque: attracting middle school students to computing through culturally-relevant themes. *Proc. of the 42nd ACM technical symposium on Computer science education* (2011), 453–458.

[14] Franklin, D., Hill, C., Dwyer, H., Hansen, A., Iveland, A. and Harlow, D. 2016. Initialization in Scratch: Seeking Knowledge Transfer. *Proc. of the 47th ACM Technical Symposium on Computing Science Education* (2016), 217–222.

[15] Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D. and Harlow, D. 2017. Using Upper-Elementary Student Performance to Understand Conceptual Sequencing in a Blocks-based Curriculum. *Proc. of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2017), 231–236.

[16] Grover, S. and Basu, S. 2017. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. *Proc. of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (New York, NY, 2017), 267–272.

[17] Grover, S., Pea, R. and Cooper, S. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*. 25, 2 (Apr. 2015), 199–237.

[18] Guzdial, M. 2010. Does contextualized computing education help? *ACM Inroads*. 1, 4 (2010), 4–6.

[19] Hansen, A., Iveland, A., Carlin, C., Harlow, D. and Franklin, D. 2016. User-Centered Design in Block-Based Programming: Developmental & Pedagogical Considerations for Children. Proc. of the 15th International Conference on Interaction Design and Children (2016), 147–156.

[20] Harel, I. and Papert, S. 1990. Software design as a learning environment. *Interactive Learning Environments*. 1, 1 (1990), 1–32.

[21] Harlow, D., Dwyer, H., Hansen, A., Iveland, A. and Franklin, D. Accepted. Ecological design based research in computer science education: Affordances and effectivities for elementary school students. *Cognition and Instruction*.

[22] Harvey, B. and Mönig, J. 2010. Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists? *Proc. of Constructionism 2010 Conference* (Paris, France, 2010), 1–10.

[23] Hill, C., Dwyer, H., Martinez, T., Harlow, D. and Franklin, D. 2015. Floors and Flexibility: Designing a programming environment for 4th-6th grade classrooms. *Proc. of the 46th ACM Technical Symposium on Computer Science Education* (2015), 546–551.

[24] Homer, M. and Noble, J. 2014. Combining Tiled and Textual Views of Code. *IEEE Working Conference on Software Visualisation* (BC, CA 2014), 1–10.

[25] Kelleher, C. and Pausch, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*. 37, 2 (2005), 83–137.

[26] Kölling, M. 2010. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*. 10, 4 (2010), 14.

[27] Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J. and Werner, L. 2011. Computational thinking for youth in practice. *ACM Inroads*. 2, 1 (2011), 32–37.

[28] Lewis, C.M. 2012. The Importance of Students' Attention to Program State: A Case Study of Debugging Behavior. *Proc. of the 9th Annual International Conference on International Computing Education Research* (New York, NY, USA, 2012), 127–134.

[29] Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M. and Rusk, N. 2008. Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin*. 40, 1 (2008), 367–371.

[30] Maloney, J.H., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E. 2010. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*. 10, 4 (2010), 16.

[31] Matsuzawa, Y., Ohata, T., Sugiura, M. and Sakai, S. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment. *Proc. of the 46th ACM Technical Symposium on Computer Science Education* (2015), 185–190.

[32] Mayer, R.E. 2002. Multimedia learning. *Psychology of learning and motivation*. 41, (2002), 85–139.

[33] Meerbaum-Salant, O., Armoni, M. and Ben-Ari, M. 2011. Habits of programming in Scratch. *Proc. of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany, 2011), 168–172.

[34] Papert, S. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic books.

[35] Papert, S. 1988. The conservation of Piaget: The computer as grist for the constructivist mill. *Constructivism in the computer age*. Lawrence Erlbaum. 3–13.

[36] Papert, S., Watt, D., diSessa, A. and Weir, S. 1979. *Final report of the Brookline Logo Project: Project summary and data analysis (Logo Memo 53)*. MIT Logo Group.

[37] Perkins, D.N. and Salomon, G. 1988. Teaching for transfer. *Educational leadership*. 46, 1 (1988), 22–32.

[38] Project GUTS: 2016. *http://www.projectguts.org/*. Accessed: 2017-04-10.

[39] Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E. and Silver, J. 2009. Scratch: Programming for all. *Comm. of the ACM*. 52, 11 (2009), 60.

[40] Rich, K., Strickland, C. and Franklin, D. 2017. A Literature Review through the Lens of Computer Science Learning Goals Theorized and Explored in Research. Proc. of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (Seattle, Wa., 2017).

[41] Rich, L., Perry, H. and Guzdial, M. 2004. A CS1 course designed to address interests of women. *ACM SIGCSE Bulletin* (2004), 190–194.

[42] Seiter, L. and Foreman, B. 2013. Modeling the Learning Progressions of Computational Thinking of Primary Grade Students. *Proc. of the 9th Annual ACM Conference on International Computing Education Research* (New York, NY, USA, 2013), 59–66.

[43] Stefik, A. and Hanenberg, S. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. *Proc. of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (New York, NY, USA, 2014), 283–299.

[44] Strauss, A. and Corbin, J. 1994. Grounded Theory Methodology: An Overview. *Strategies of Qualitative Inquiry*. Sage Publications, Inc. 158–183.

[45] Sweller, J. and Chandler, P. 1994. Why some material is difficult to learn. *Cognition and instruction*. 12, 3 (1994), 185–233.

[46] Tai, R.H., Liu, C.Q., Maltese, A.V. and Fan, X. 2006. Career choice: Enhanced: Planning early for. *Science*. 312, 26 (2006).

[47] Tempel, M. 2013. Blocks Programming. *CSTA Voice*. 9, 1 (2013).

[48] Tsur, M. and Rusk, N. 2018. Scratch Microworlds: Designing Project-Based Introductions to Coding. (2018), 894–899.

[49] Turkle, S. and Papert, S. 1990. Epistemological pluralism: Styles and voices within the computer culture. *SIGNS: : Journal of Women in Culture and Society*. 16, 1 (1990), 128–157.

[50] Weintrop, D. and Holbert, N. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. *Proc. of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2017), 633–638.

[51] Weintrop, D. and Wilensky, U. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education (TOCE)*. 18, 1 (Oct. 2017), 3.

[52] Weintrop, D. and Wilensky, U. 2018. How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction*. (May 2018).

[53] Weintrop, D. and Wilensky, U. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. *Proc. of the 14th International Conference on Interaction Design and Children* (New York, NY, USA, 2015), 199–208.

[54] Wolber, D., Abelson, H., Spertus, E. and Looney, L. 2011. *App Inventor: Create Your Own Android Apps*. O'Reilly Media.

[55] 2016. K–12 Computer Science Framework.