Improving K-12 STEM Education Outcomes through Technological Integration

Michael J. Urban
Bemidji State University, USA

David A. Falvo
Walden University, USA

A volume in the Advances in Early Childhood and K-12 Education (AECKE) Book Series

Information Science REFERENCE
An Imprint of IGI Global

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Names: Urban, Michael J., 1977- editor of compilation. | Falvo, David A., 1960- editor of compilation.
Title: Improving K-12 STEM education outcomes through technological integration / Michael J. Urban and David A. Falvo, editors.
Description: Hershey, PA : Information Science Reference, [2016] | Includes bibliographical references and index.
Identifiers: LCCN 2015037582| ISBN 9781466696167 (hardcover) | ISBN 9781466696174 (ebook)
Subjects: LCSH: Science--Study and teaching--Technological innovations. | Educational technology.
Classification: LCC LB1585 .I47 2016 | DDC 372.35/044--dc23 LC record available at http://lccn.loc.gov/2015037582

This book is published in the IGI Global book series Advances in Early Childhood and K-12 Education (AECKE) (ISSN: 2329-5929; eISSN: 2329-5937)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 15
Computer Programming in Elementary and Middle School:
Connections across Content

Danielle Boyd Harlow
University of California – Santa Barbara, USA

Charlotte Hill
University of California – Santa Barbara, USA

Hilary Dwyer
University of California – Santa Barbara, USA

Ashley Iveland
University of California – Santa Barbara, USA

Alexandria K. Hansen
University of California – Santa Barbara, USA

Anne E. Leak
University of California – Santa Barbara, USA

Diana M. Franklin
University of Chicago, USA

ABSTRACT

Computing has impacted almost all aspects of life, making it increasingly important for the next generation to understand how to develop and use software. Yet, a lack of research on how children learn computer science and an already impacted elementary school schedule has meant that very few children have the opportunity to learn computer science prior to high school. This chapter introduces literature on teaching computer programming to elementary and middle school, highlights three studies that span elementary and middle school, and discusses how programming can be integrated into other content areas and address national standards.

INTRODUCTION

To become the next generation of innovators, today's children must learn to create with technology. To create new technology or innovate on existing technology increasingly requires learning computer programming skills. Computer science in K-12 classrooms is rapidly expanding as technology, programming environments designed for children, and computer science curricula become available. Both the

DOI: 10.4018/978-1-4666-9616-7.ch015

International Society for Technology in Education (ISTE, 2007) and the Computer Science Teachers Association (CSTA) have developed standards relevant to computer programming and computer use.

Teaching computer programming to elementary and middle school children has been facilitated by the development of graphical programming environments that allow children to create programs by dragging and dropping commands (represented as images of blocks) onto a screen, lowering the cognitive barrier to programming (e.g., Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) and increasing novices' interest and excitement in programming (e.g., Malan & Leitner, 2007). These programming environments allow younger students to access computer science ideas without first learning complicated syntax and formatting – attributes of traditional programming. Despite these developments in programming environments and curricula, as well as nationwide efforts to include computer programming at all levels, the vast majority of children do not learn programming in schools prior to high school.

Computer science, when it is taught to younger children, is often limited to outreach events, summer camps, and projects with parents. These informal types of learning experiences have shown success, but they impact only self-selected students, often those already interested in computers and those from middle and upper class families (e.g., Margolis & Fisher, 2003). Integrating computer science into elementary and middle school classrooms, rather than only out-of-school environments is important for addressing equity issues. The current demographics in computer science and computer engineering are indicative of a larger access gap in technology fields. Currently, female students receive a slight majority of all undergraduate degrees nationally, yet they represent only 14.5% of Computer Science degrees awarded. Latino/a students earn only 6.5% of undergraduate computer science degrees while representing 14% of the national population (Computing Research Association, 2013). The disparity in interests and experience with computer programming must be addressed early in students' education. In fact, research indicates that students' reported interest in pursuing a career in science and engineering areas as 8th graders is a strong predictor of whether or not they will pursue a science career (Tai, Liu, Maltese, & Fan, 2006). This means that students' experiences prior to 8th grade are important to recruiting students to careers into science careers, including computer science.

To reach all students, computer science needs to be integrated into the elementary school classroom at the elementary and middle school levels. Yet, integrating computer science into the school day is challenging, in part, because the school day is already full, making adding an additional topic difficult. To be widely integrated into the elementary and middle school curriculum, computer programming activities must address the content standards in mathematics, literacy, and science that teachers are held accountable for in their K-12 classes.

The United States recently made significant changes to educational standards to ensure the nation's graduating students are college and career ready and equipped with the skills and knowledge necessary to be competitive on a global scale. These new standards include the Common Core State Standards (CCSS) for English Language Arts and Mathematics (National Governors Association Center for Best Practices & Council of Chief State & School Officers, 2010) and the Next Generation Science Standards (NGSS) (NGSS Lead States, 2013). All focus on practices of the discipline along with important disciplinary ideas. One important aspect of the NGSS is that, for the first time, engineering design is included in standards. Engineering design is the process by which engineers develop innovative solutions to problems. The process involves understanding the problem, generating ideas, selecting an idea based on multiple constraints, and improving the idea, a process consistent with computer programming.

This shift in educational standards provides an opportunity to consider how computer programming can support students' learning as they work to meet standards in other content areas. Rather than teach-

ing computer programming as a separate subject entirely, it can be integrated into traditional classroom subjects, such as reading, mathematics or science. For example, students can create digital book reports by programming a story that includes character development or program imagined alternate endings to stories, supporting language learning. Students can also create programs that draw shapes of specified angles and numbers of sides, supporting mathematics learning. To support science learning, students can program stories that require the use of specific science content, such as interdependent relationships in an ecosystem. Integrating computer programming with other content areas is not only desirable from a scheduling perspective, but also from a student learning perspective. When computer science is integrated with other content areas, students learn that programming can be useful across disciplines. Computer programming allows teachers and their students to explore concepts across the curriculum and creatively communicate learning, providing children with opportunities to use content in novel ways such as creating learning experiences for other children (e.g., Kafai, 2006).

Computer programming is not only useful for supporting other content areas or even for the types of thinking it encourages. Learning to program allows children to create software that can be viewed and used by others. Kafai and Burke (2014) argue that the value of programming is that provides a mechanism for children to express themselves and connects them to networks of digital communities. That is, programming provides students an opportunity to participate in the construction of digital spaces and resources. This idea resonates with Papert and Harel's (1991) theory that people learn best when creating things that can be seen or used by others. Resnick (2006), the developer of Scratch, one of the most popular programming environments for children, expanded on why learning through making is so powerful. He described that the design cycle of trying out and testing ideas "can be seen as a type of play: children play out their ideas with each new creation. In design activities, as in play, children test boundaries, experiment with ideas, and explore what's possible" (Resnick, 2006, p. 196).

This chapter overviews three projects designed to help elementary and middle school students learn programming and computational thinking using block-based programming environments. Following this overview, considerations are discussed to assess the suitability of programming languages and environments for use with elementary and middle school students.

BACKGROUND

Computational Thinking

One of the eight practices included in the NGSS is "mathematics and computational thinking," an indicator of the importance of computational thinking and computers in the field of science. Broadly, computational thinking means to consider how to formulate a problem so that a computer can solve it. While the articulation of the NGSS practice largely focuses on the mathematics part of this practice and, thus far, has not described computational thinking in much detail, the Computer Science Teachers' Association (CSTA) has developed a set of standards that defines this practice and describes how it relates to other disciplines. In developing the standards, they used a definition of computational thinking reported by Barr and Stephenson (2011) and developed at workshops collaboratively hosted by CSTA and NSTA:

CT [Computational Thinking] is an approach to solving problems in a way that can be implemented with a computer. …CT is a problem-solving methodology that can be automated and transferred and applied

across subjects. The power of computational thinking is that it applies to every other type of reasoning. It enables all kinds of things to get done: quantum mechanics, advanced biology, human-computer systems, development of useful computational tools. (Barr & Stephenson, 2011, p. 51)

The term computational thinking was first used by Papert (1996) and further defined by Wing (2008a, 2008b). Seiter and Foreman (2013) later articulated a learning progression for computational thinking based on their observations of what children were able to do at each grade level in elementary school. They found that children in grades three and four were able to develop many of the components of computational thinking necessary for digital story-telling, including animating characters and creating conversations. Brennan and Resnick (2013) developed a framework for computational thinking that added practices and perspectives to concepts. According to their model, computational thinking concepts include sequences, loops, events, parallelism, conditionals, operators, and data; practices include being incremental and iterative, testing and debugging, abstracting and modularizing; and perspectives include questioning, connecting, and questioning. Grover and Pea (2013) point out the computational thinking overlaps with other disciplinary ways of thinking such as engineering, mathematical, and design thinking, and extends these ways of thinking in ways specific to computing.

Here, computational thinking is considered to be a set of problem solving processes and concepts that relate to solving problems both on the computer and in everyday life. Table 1 describes five ideas related to computational thinking that are appropriate for elementary and middle school students and focused on in the three studies discussed in this chapter. These ideas have applications not only in programming, but in how we solve problems everyday.

Many of these ideas can be taught either on or off the computer. CS Unplugged (Bell, Witten, & Fellows, 2009), for example, is a set of activities designed to teach computational thinking independent of computer programming. This chapter focuses primarily on on-computer computer programming tasks.

Table 1. Elementary and middle school computational thinking ideas with related examples from everyday problem solving and programming

Computational Thinking Idea	Everyday Example	Programming Example
Sequencing: Creating an ordered list of instructions to complete a task.	Explaining to someone how to get from one place to another.	Putting commands in the correct order so that the computer accomplishes a specific task.
Breaking down actions: Breaking an event into smaller parts.	Recognizing that events (e.g., "get dressed") involve several smaller steps (e.g., "put socks on, put shoes on").	Recognizing that to make a character move across a screen requires combinations of turning and moving forward.
Looping: Repeating a set of instructions multiple times.	When clapping, you put your hands together and then apart repeatedly.	Repeating a piece of code a specific number of times. For example, to draw a square, you can move and turn 90 degrees four times in a row.
Event-driven Programming: Identifying how one circumstance triggers another.	Many everyday actions are triggered by events. If it rains (a triggering event), one might open an umbrella.	Creating a code that occurs in response to another event. For example, when the space bar is pressed (triggering event), a bat moves across the screen (reaction).
Message Passing: Coordinating actions across code or characters	Messages in everyday interactions can be heard or seen. For example, a traffic light turning red indicates that cars should stop.	Programming one object to send a message after an action. Programming another object to act when that message is received. For example, one character says, "Hello" and broadcasts a message. Another character says, "How are you?" after receiving the message.

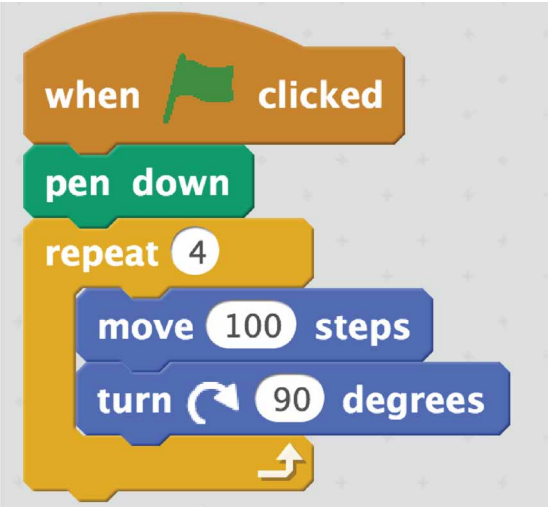
Learning Computer Programming

Computer programming is the process of creating code that tells the computer what to do. This requires not only the ways of thinking described above, but knowledge of a computer language or programming environment. Children have been programming since at least the early 1980's when Seymour Papert (1980) introduced Logo, the first educational programming language designed specifically for children. In Logo, children programmed an image of a turtle to draw using simple commands like "FORWARD 100" which directed the turtle to move forward 100 steps. Papert described how students could learn mathematics and physics through programming, and proposed that children at any grade level could learn to program in the classroom. Papert's work set the stage for the next decades of computer science education research.

Even with simpler languages like Logo, novice programmers of all ages must learn the structure and execution of the programming language, names of commands, and other syntactical and formatting rules. Kelleher and Pausch (2005) noted that those new to programming "must also learn rigid syntax and rigid commands that have seemingly arbitrary or perhaps confusing names" (p. 83). In response, block-based languages were developed to reduce the need to remember codes and syntax, enabling students to focus on programming. Block-based languages and programming environments provide a set of programming commands in graphical blocks that users drag onto the screen and snap together like puzzle pieces to create scripts (see Figure 1 for example of a script). These scripts often control characters or other images that can be used to create a game, tell a story, or more.

Block-based programming languages have gained popularity in recent years, particularly in online applications for iPads and other tablets. Block-based languages are a natural fit for tablets since typing is minimal and dragging and dropping is the primary way of interacting with the environment. These apps and web-based programs range from programming environments that allow users to create their own games and digital stories to more structured game or tutorial-environments designed to teach computer programming. These block-based languages are often inspired by Scratch, a block-based programming language and environment developed by MIT in 2003 (Resnick et al, 2009).

Figure 1. Scratch program (right) to draw a square with sides of length 100. The equivalent Logo program would be a single line of code reading, "repeat 4 [forward 100 right 90]".



In classrooms, teachers must structure the time students spend coding in some way. This may be an unstructured as allotting time to work on independent projects or through more organized curricular tasks. Existing coding curricula range from standards to guide teachers in designing their own lessons, lesson plans for teachers to follow, to online tutorials that guide students through the material. The Computer Science Teaching Association (CSTA) offers general standards for teachers to follow when incorporating programming into their K-8 classes (Seehorn et al., 2011). ScratchEd provides lesson plans for teaching coding with Scratch and an online community where educators can help each other with the material (ScratchEd, 2014). Other curricula, such as the ones provided by code.org, consist of interactive tutorials that students can use either in the classroom with the support of their teacher or independently. Introduced in 2011, Computer Science Education Week (Hawthorne, 2011) has grown in popularity in recent years, and now multiple platforms provide short tutorials or projects every December for students to try out programming.

Whether designed by classroom teachers or part of existing curricula, programming activities developed for students using block-based programming environments typically take two main forms: (1) open-ended and (2) pre-populated. With open-ended programming activities, there are no existing scripts (short pieces of code) provided to students. Instead, there is a blank coding area. Students are provided with the programming equivalent of a blank sheet of paper. In contrast, pre-populated lessons included some sprites (programmable objects or characters) on the screen when the activity is opened and these sprites have some pre-coded scripts. Pre-populated activities may engage students in games (such as programming a character to move through a maze) or challenges to complete or fix an existing project.

Computer programming, even when using block-based programming environments depend on children knowing what the words used as commands mean. In the early 1980's significant work was done to understand the role of natural language in learning programming with mixed results. Some found that using natural language aided programming. Others found that using words that had alternate everyday meanings caused confusion (Wright & Cockburn, 2005). Since programming languages are malleable, if children do not understand the words used for commands or other aspects of the programming environment, new words can be used. As a result, most studies of language in computer science education have focused on the *programming language* and how to make these languages accessible to novices. However, when learning programming, children need to understand not only the words used as commands, but also the language that surrounds computer science and instructional context – which may include the teachers' spoken instructions, explanations, and questions, or written language on worksheets or instructions. Children learn programming in a language-rich context that includes much more than just the commands in the programing language (Dwyer et al., 2015).

Further, since the first programming languages for children were built and tested, there has been a significant shift in students' backgrounds and experience. Census data indicates that between 1980, when Logo was introduced, and 2010, there has been a 158% increase in the number of people who speak a language other than English at home. In contrast, the total US population only increased by 38% (Ryan, 2013). This change in demographics has significant impacts for schools and for the complexity of the relationship between learning language and learning computer programming because many children are learning programming in English-speaking classrooms while still learning English.

The following section describes three studies of elementary and middle school children learning computer programming in ways that augmented other content areas and improve students' access to computer science thinking and practices.

METHODS

Participants and Study Designs

To illustrate the possibilities of computer programming across grade levels, this chapter describes three separate studies. These were all "entry" experiences for students, meaning that all three projects assumed students began with no programming background. Students did not continue from one experience into another. Two were conducted in elementary classrooms and involved elementary school teachers, and one was an outreach program taught exclusively by computer scientists and undergraduate and graduate students. In one study the students learned the computer programming skills necessary to draw geometric shapes. In the other two studies, the students learned the computer programming skills necessary to create digital stories. For the digital stories, students programmed characters and scenes that interacted with one another. We discuss the participants and data collection for the three projects beginning with the youngest students. Table 2 summarizes information about participants and curricula for the three studies.

The participants in the first study presented were 20 third grade students (eight and nine years old) in one elementary school classroom in California and their teacher. Underrepresented ethnic groups (mostly Latino/a) constituted 31.5% of the student population. Nearly one-fifth (19%) were identified as English Language learners. The classroom was evenly divided across gender. The teacher and his students were part of a project using XO laptops, a computer developed as part of the One Laptop per Child (OLPC) project. The students were video recorded once a week for one hour during class time devoted to using the XO's. The analysis focused on class periods in which the teacher chose to use TurtleArt, a simplified block-based programming environment that was included as one of the many applications available on the computer. The teacher used TurtleArt for teaching geometry, and also allowed students free time

Table 2. Overview of participants, curricular structure, programming environment, and goals of studies. N/A indicates that these data were not collected for the indicated study.

	Lower Elem. Study	Upper Elem. Study	Middle School Study
Grade Level	3rd	4th-6th	Entering 6th-8th
Participants (N)	20	1,250	120
Gender	50% female, 50% male	50% female, 50% male	76% female, 24%male
Ethnicity	N/A	N/A	64% Hispanic, 36% other
English Language Learners	19%	2%-82% (depending on school)	N/A
Qualified for free or reduced lunch	24%	4%-100% (depending on school)	N/A
Study Duration	1 year	3 years	3 years
Programming Environment	TurtleArt	LaPlaya	Scratch
Time spent on CS	1 hr/wk (4 hrs on Turtle Art)	16 hours	30 hrs (of a 60 hr camp)
Final Project	Drawing shapes	Digital Story	Digital Story
Computer Science Goals	Sequencing, Looping, Spatial Reasoning	Sequencing, Breaking Down Actions, Initialization, Event-Driven Programming	Sequencing, Event-Driven Programming, Initialization, Message Passing

to explore the programming platform and to make drawings of their choosing. Unlike the other studies that will be described in this chapter, the focus of this study was exploratory and focused on what children did when using TurtleArt, not as a means to inform future instruction. The data analysis was designed to identify how ideas moved across the classroom with particular attention to new ideas and outputs expressed by the students. Videos of class periods using TurtleArt were event-mapped (Collins & Green, 1992) and episodes in which children shared ideas were transcribed and coded based on verbal and non-verbal interactions between students and their peers, teachers, and laptops. See Harlow and Leak (2014), for more details.

The second study focused on upper elementary school students (grades four through six, ages nine through eleven). The study participants included over 1,250 fourth through sixth graders in schools across California. The first year of implementation included 15 classrooms with over 400 students, and in the second year included more than 35 classrooms with over 850 students. Classrooms of students were selected to represent California's diverse population, with a particular effort to include schools with large populations of underrepresented ethnic minorities and English Language Learners. Participating schools ranged from 2% to 82% English Language learners and from 4% to 100% students qualifying for free and reduced lunch, a proxy for socioeconomic status. This research followed a design-based methodology, a way to research learning in the context of a complex learning system. The research on learning is both informed by the context and informs the design of curriculum, pedagogy, and contexts (Barab & Squire, 2004). In this case the research on student learning informed the design of the programming environment, curricular tasks and goals, and pedagogical training of participating teachers. Teachers attended a summer workshop to learn the curriculum and then taught approximately one 45-minute lesson per week for sixteen weeks in their classrooms. The curricular activities consisted of scaffolded pre-populated projects and a cumulative open-ended digital story project. Extensive qualitative data were collected both years at schools within driving distance of the host university. This included classroom observations, interviews with students informally during activities and more formally after they completed a module, online and written student work, and exit interviews with local teachers. At the schools further than 100 miles away, written work, periodic snapshots of all computer projects, responses to online surveys and assessment questions were collected. Analysis included coding of qualitative data and quantitative analysis of the periodic snapshots of computer projects and focused on instances when students or teachers were frustrated or completed projects in unexpected ways. This allowed us to understand how children learned computer science and to identify unintentional barriers to learning or insufficient scaffolding and adjust the curriculum or programming environment in response (e.g., Dwyer et al., 2015; Franklin et al., 2015; Hill, Dwyer, Martinez, Harlow, & Franklin, 2015).

The third study presented focused on middle school students. The participants of this study were enrolled in a two-week day camp (six hours/day) taught once each summer for three summers. Approximately 40 students attended the camp each summer on the campus of a local university. The students were primarily Latino/a (64%), female (78%), and entering grades six through eight. Unlike the other two projects in which the computer science lessons were taught primarily by classroom teachers, the middle school camp was taught by project staff which consisted of three university faculty members (two in computer science and one in Chicana/o studies), three graduate students (from Chicana/o studies, education, and computer science departments), and six undergraduates (mostly computer science). Also unlike the other two studies where all students in a classroom participated, these participants were self-selected. Data collected on the middle school project included quantitative pre/post assessments of student interest, field notes on students' requests for assistance and completed student projects with

a focus on assessing student learning and the effectiveness of the project. Analysis focused on student interest and their learning about computer science as a result of the summer camp (Franklin, Conrad, Aldana, & Hough, 2011, Franklin et. al., 2013).

Programming Environments

The three studies used different, but related, block-based programming environments: TurtleArt, LaPlaya, and Scratch. The youngest group of students (grade three) used TurtleArt (e.g., Bontá, Papert, & Silverman, 2010; Pilco, 1990) to create drawings (see Figure 2). TurtleArt is a derivative of Logo. In Logo, children command a turtle on the screen to move and draw by typing commands such as Forward 100 and Right 90. TurtleArt uses these same commands, but the commands are displayed as two-dimensional blocks that are dragged and dropped onto a programming area and snap together like puzzle pieces to create a script or code. The interface is similar to the Scratch interface (Resnick et al, 2009) used by older children, but TurtleArt has a smaller scope and number of available commands. TurtleArt commands are limited to those commands that direct the turtle to move and draw. Unlike Scratch programs, which are capable of including multiple interacting programmable entities (sprites), TurtleArt programs typically command only a single programmable agent (the turtle), simplifying the process.

The upper elementary school students (grades four through six) used a modified version of Scratch, called LaPlaya (see Figure 3). LaPlaya differs from Scratch in that the developer can select the commands that will be visible to the user to limit distractions and reduce the number of commands available to the students. It also adds the functionality of hiding scripts (pieces of code) so that, when students are learning, they can focus only on the sprite of interest. These features of LaPlaya allow for pre-populated projects in which students can focus only on parts of the full project and assist in implementing lessons in the context of short lab periods (see Hill et al., 2015).

Figure 2. TurtleArt programming environment used with lower elementary school students

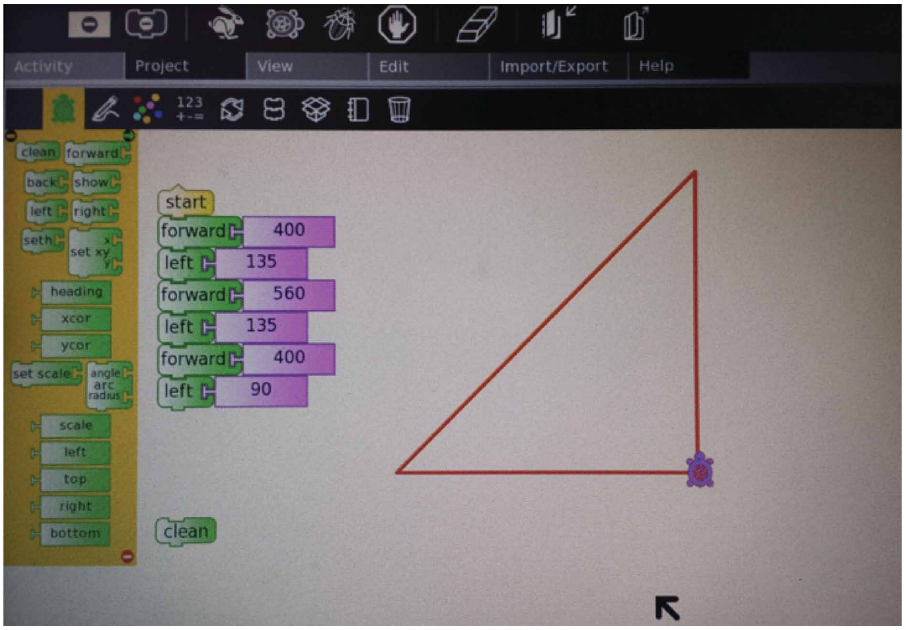
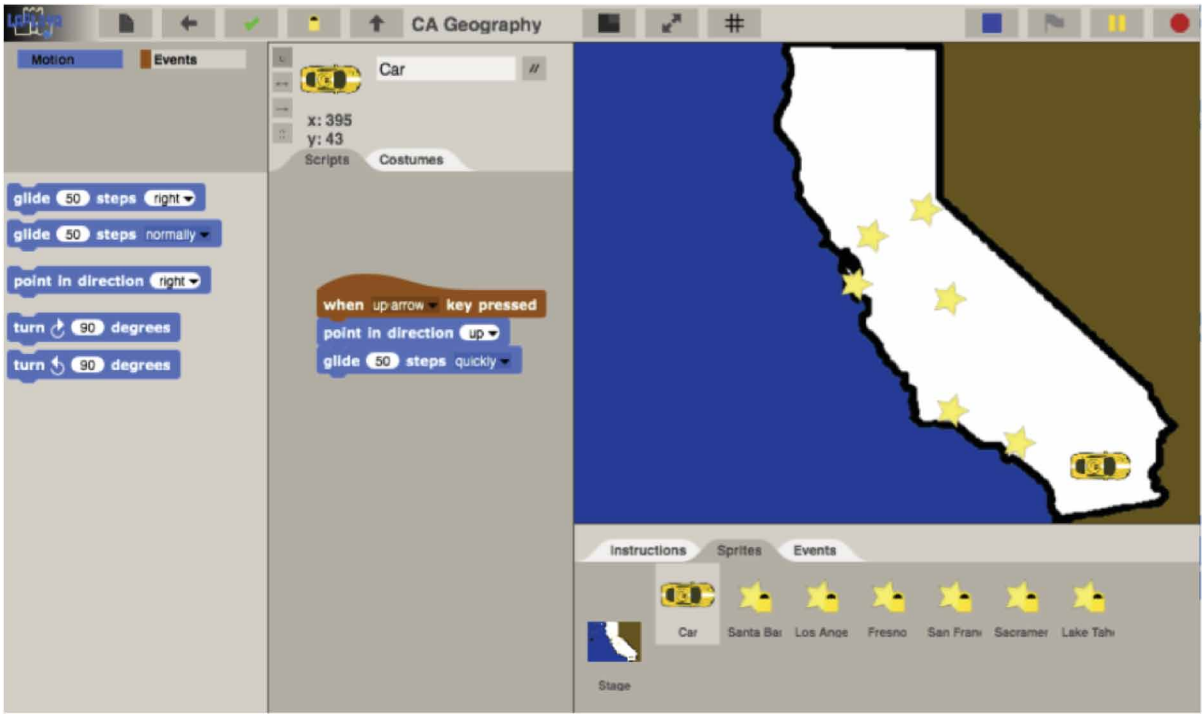


Figure 3. LaPlaya programming environment used with upper elementary school students



The middle school students used the freely available Scratch programming environment (see Figure 4). Scratch includes a selection of blocks (commands) that are divided into categories based on their function. A menu of categories is visible in the upper left of the environment on the left side of the screen. Below this, all the blocks in the selected category are displayed. Students drag these scripts onto the programming area (center area of screen). The lower right area shows the sprites, or programmable objects, used in the program. The scripts displayed are those of the selected sprite. The upper right area of the environment shows the program output.

FINDINGS

Lower Elementary School Students: Drawing Shapes

As described above, in the study of the youngest children, third graders used TurtleArt to create geometric shapes. The teacher and researchers identified TurtleArt as a particularly valuable context for students developing creative ideas and for taking ownership of their ideas, a focus of the teacher's instruction.

In this project, the teacher allowed students time to explore with TurtleArt without instructions. That is, the children were engaged in open-ended programming. During this time, some students tried to draw with specific goals in mind (e.g., tried to draw their name), while others explored what happened when they put blocks together, noticing patterns and shapes (see example in Figure 5). At other times, the

Figure 4. Scratch programming environment used for with middle school students

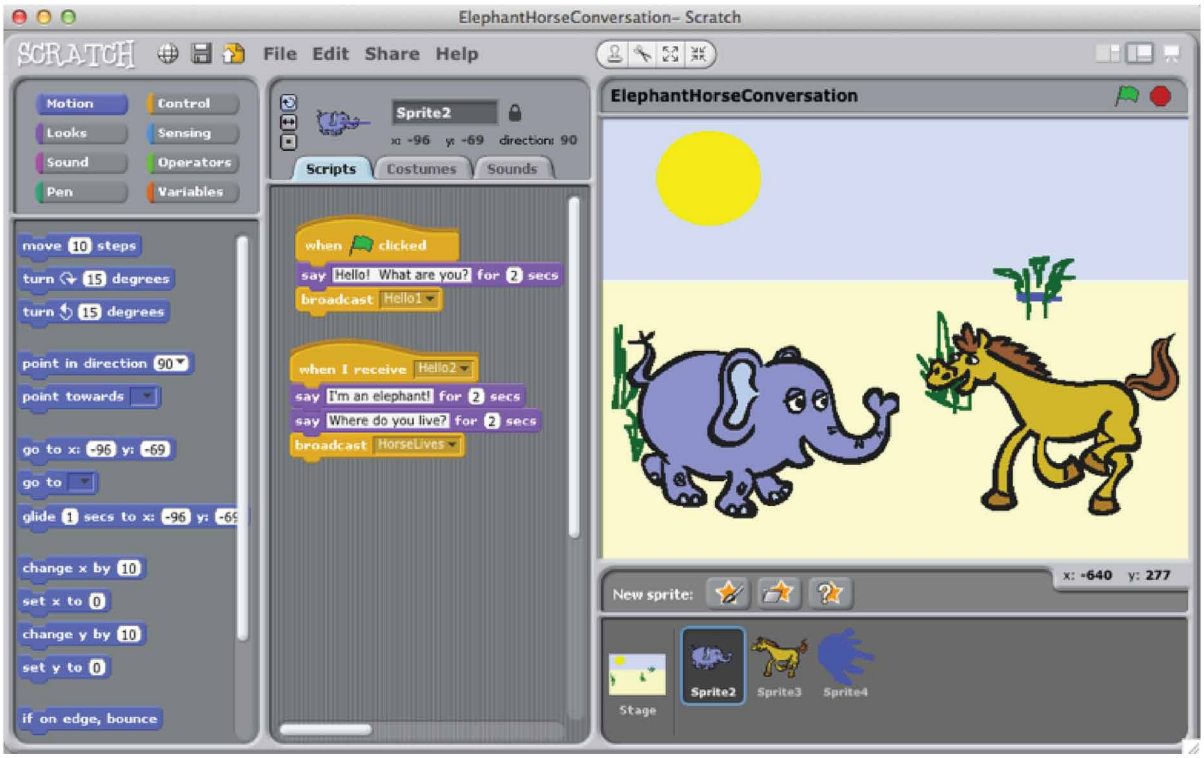
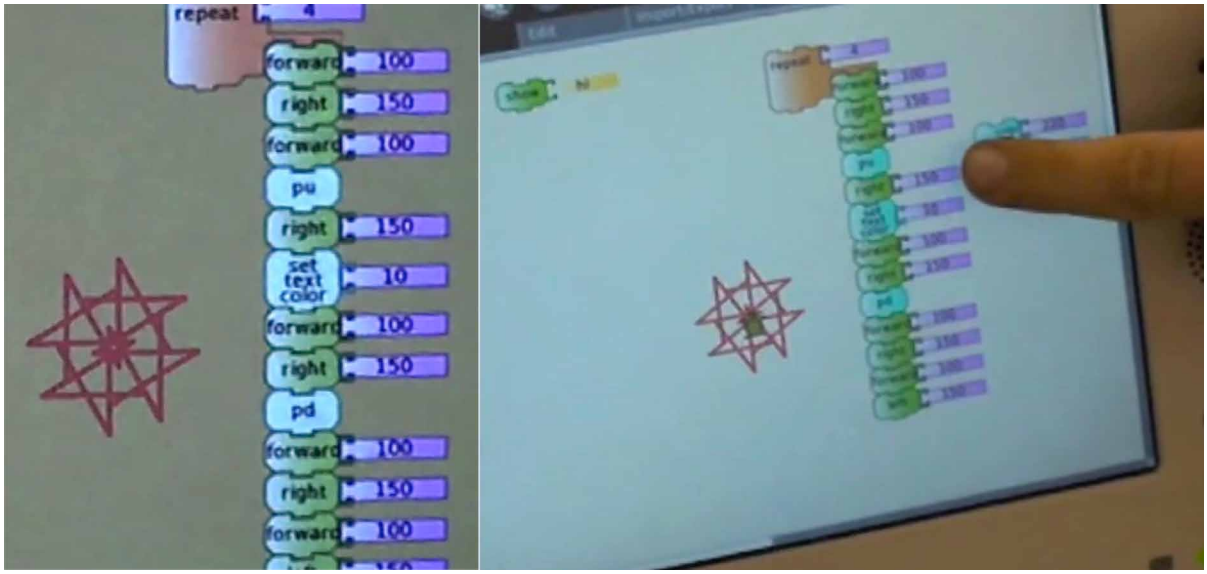


Figure 5. Screen shot of student work exploring patterns and shapes



teacher set specific challenges to target mathematical or computer science concepts or skills. Examples of challenges included drawing squares of different sizes or in specific positions, drawing triangles, and shapes such as houses that were created by combining multiple shapes, tasks that require the computational thinking ideas of sequencing and the mathematical ideas of angles and measurement. Other tasks included drawing a square in the smallest possible number of steps, prompting students to consider looping, and drawing shapes that only use multiples of 360 for the angles, a geometry idea in their math instruction. Guiding instruction with challenges helped students to create new shapes more efficiently.

In the TurtleArt programming environment, the program and resulting drawing are shown in the same area of the screen. This means that the output (drawing) and process (code) are easily and simultaneously visible to an onlooker. This made TurtleArt a useful platform for sharing ideas with their peers. The researchers developed a representation for mapping *how* the ideas changed as they were taken up and developed by other children in the class, identifying changes in the output and changes in the program. Ideas were then mapped based on these codes to explore how they moved between students and evolved over time. In mapping these ideas and their resulting changes in programs and outputs, we were able to identify the context in which ideas moved through the classroom. More efficient programming techniques were usually prompted by questions or comments from the teacher. For example, when the teacher asked a child if he could make the same shape in fewer steps, two children, Cory and Tom (pseudonyms), worked together and figured out how to use looping to make their program more efficient.

Cory: *Look at this! Heather, look at this.*
Mr. Mills: *How many clicks, Cory?*
Cory: *Um, a lot.*
Tom: *No, just tell me*
Cory: *I don't know, I don't remember.*
Tom: *Clean and do it over and then tell many clicks and I can make it do it in one click.*
Cory: *I don't know how to do that*
Tom: (leaning over and pointing to Cory's screen) *repeat. No, you put it there.*
Cory: *Forward, forward, forward, forward. Like that?*
Tom: *Okay good, now forward, click, forward.*
Cory: *Whoa, that's a lot* [*of steps*]
Tom: (In a loud voice) *He did it in one click. He taught himself to do it.*

In contrast, once students developed skills, new uses of these techniques were usually student-driven. For example, a student asked her friend how she had made a triangle and then returned to her own computer to put triangles together and make a flower. Third grade students in this study shared and changed their ideas, using similar programs to create original outputs. In some cases, ideas developed in one class meeting were taken up again and further replicated and evolved in subsequent class periods, sometimes weeks or months later. The shared ideas became part of the classroom knowledge and ideas that the children could access when using TurtleArt. Programming in the TurtleArt environment provided third graders with an opportunity to explore mathematical content in a way that they took ownership of their ideas.

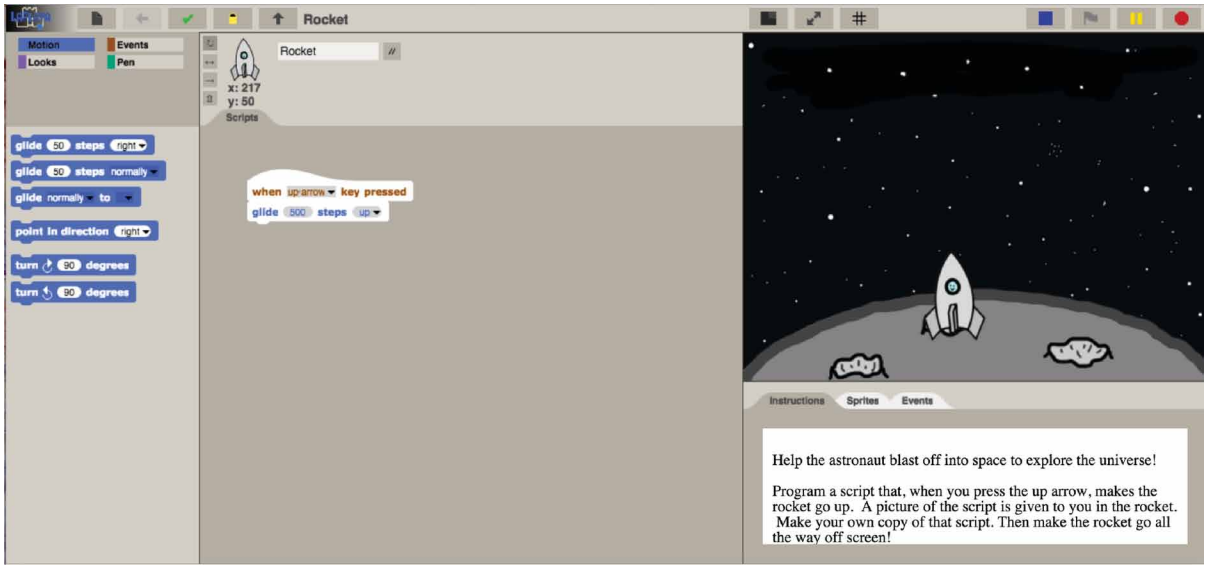
Upper Elementary School Students: Scaffolded Projects and Digital Story Telling

The studies of fourth through sixth graders were part of a larger project, which had two interdependent goals: (1) designing computer science curriculum for upper elementary school students and (2) investigating how upper elementary school students learn computer science. The curriculum, called Kids Engaged in Learning Programming – Computer Science (KELP-CS), is a scaffolded curriculum that emphasizes computational thinking, basic programming, and NGSS engineering design. In the first round of implementation, a minimally modified version of Scratch was used as the programming environment. As students struggled with aspects of Scratch, the interface was further modified. The current version is called LaPlaya (see Hill et al., 2015 for description).

KELP-CS incorporated both on-computer and off-computer activities, but analysis focused primarily on the on-computer activities. In the on-computer skill building activities, students completed a series of small programming projects at their own pace that incrementally became more challenging. The goal of these lessons was for students to learn the skills for programming digital stories (e.g., sequencing, event-based programming, etc.). Most of the on-computer tasks used pre-programmed projects. For example, in one project, students programmed a bear to navigate a maze to reach a honey pot and avoid bushes. The honey pot was programmed to display a congratulatory message when the bear reached it and the bushes programmed to display an error message and send the bear back to the starting point. Another activity tasked students with programming a rocket to move when the arrow keys were pressed (see Figure 6).

Students also completed a culminating engineering design project (a digital story). As students progressed through the curriculum, they created and revised their final project. The curriculum was implemented in three ways depending on the structure of the specific elementary school: (1) a teacher taught lessons to his or her homeroom class either in the computer lab or on laptops in their classrooms, (2) a teacher taught lessons to students in an entire grade throughout the week as classes visited the computer lab, or (3) a technology specialist taught lessons in the computer lab as classes visited each week.

Figure 6. Screen shot of rocket task



Focus groups were conducted with fourth graders before implementing the computer science curriculum showed that students had foundational computational thinking skills (Dwyer, Hill, Carpenter, Harlow, & Franklin, 2014). These interviews engaged groups of children in off-computer tasks such as writing instructions to draw complex figures, following such instructions, and sorting objects by weight. In the interviews, fourth graders recognized the need for specific instructions, were able to critique peers' instructions, and could create basic algorithms (instructions to solve problems). These findings provided a starting place for building the curriculum. During implementation of KELP-CS, students successfully completed projects related to sequencing, event-driven programming, and message passing, key computational thinking ideas (refer to Table 1) as well as initialization, costume changes, and scene changes. The culminating digital story projects (see Killian et al., in press, for more details) engaged students in engineering design as they iteratively revised what stories or information to share and ways to implement such narratives in through programming.

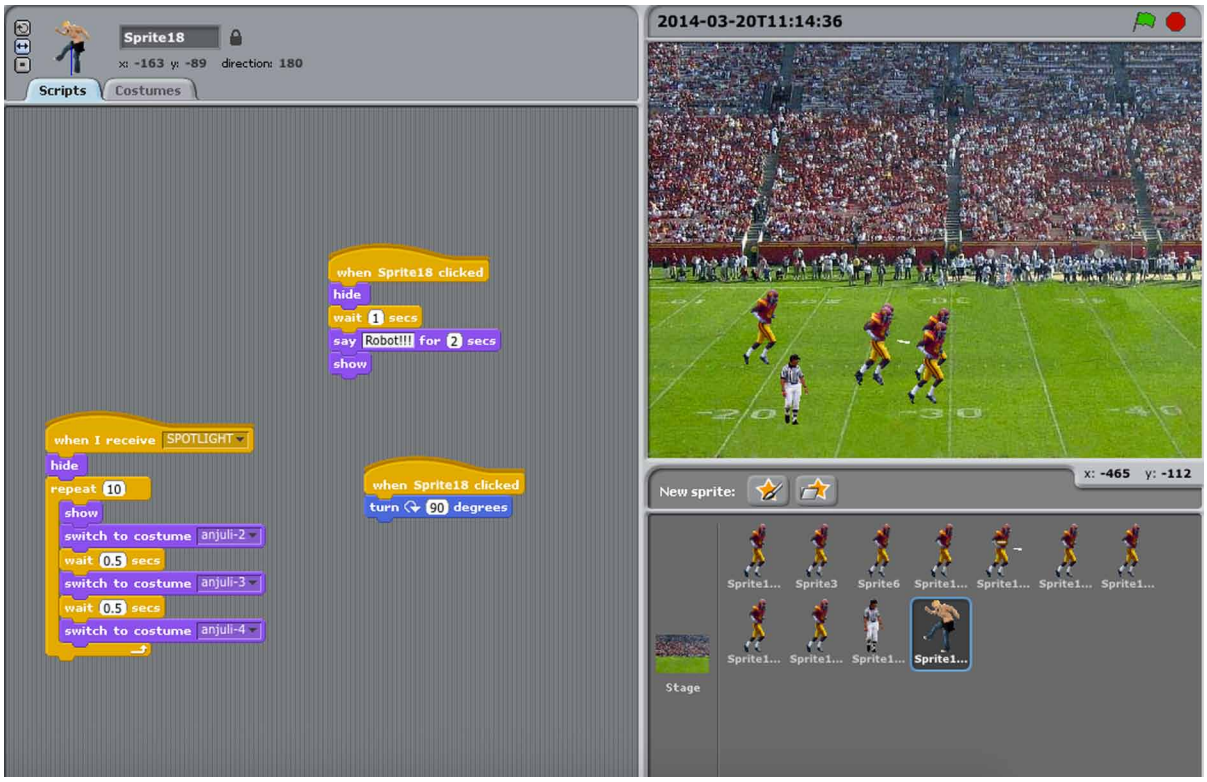
Students' difficulties with the curriculum and the programming environment pointed to barriers to learning programming related to their language skills, mathematical knowledge, as well as complexities of programming. Several target classrooms in this study included populations of near 80% English Language Learners, most of whom spoke Spanish at home. This meant that the majority of students in these classrooms were reading instructions, listening to a teacher, and creating programs in a language they were still learning. This compounded the difficulty of learning the meaning of commands and the specialized language of computer science. Analysis of classroom videos focusing on student difficulties indicate that children struggled to distinguish between commands that had different functions but used words that had similar meanings in English (see Dwyer et al, 2015). For example, the commands "glide" means to move to another location with visible motion and "go to" means to disappear from one location and instantly appear at another. Other language related difficulties included larger concepts like broadcasting messages. In Scratch and LaPlaya, events can be coordinated by sending messages from one sprite to another. Instead of the "send message" command, children tried to use the "say" command, which displays a speech bubble and text above the sprite. Ongoing research on this project is attempting to tease apart the experiences of English Language Learners, as compared to fluent English speakers to better support their learning.

The initial programming environment used with these students (a minimally edited version of Scratch) also required mathematical knowledge above fourth grade level to successfully create even simple digital stories. These ideas included negative numbers, percentages, and the x/y coordinate plane. Identifying these issues led to changes in the programming environment used with the fourth through sixth grade students (Hill et al., 2015).

In addition to the language and mathematics ideas, students struggled with some programming practices. The first was keeping track of and coordinating multiple events and sprites. Both Scratch and LaPlaya display only the scripts associated with a selected sprite. This means that while working on programming one sprite, the student must keep the actions of the others in mind in order to coordinate among them to create a coherent story. Providing students with tools for visually representing their stories (story boards and flow charts) was crucial to their success. Figure 7 shows a screen shot from example student work in which a student programmed a football game. This student uses message passing to control the flow of the story. Notice that only the commands directing one player are visible.

Despite the difficulties of learning to program, students overwhelmingly enjoyed creating computer programs, even when they struggled. Students were asked in focus groups at the end of the curriculum what they liked and what they did not like about the computer programming activities and whether they

Figure 7. Example of a digital story depicting a football game that uses message passing



would recommend it to a friend. Every child answered positively. For example, when asked what they would tell their friends about learning computer programming, Robin responded, "[I would say] It's really good but I warn you one of them is really hard… That's the one where you had to make the dang chicken switch around and that took me like an hour." A second student, Lance, followed up by stating, "But that's what makes it fun. That makes it fun."

Middle School Students: Digital Storytelling

The middle school students were enrolled in an out-of-school summer program to learn about three topics: endangered species, Mayan culture, and computer programming (Franklin, Conrad, Aldana, & Hough, 2011; Franklin et al., 2013). The goal of the summer camp was to increase interest in computer science for female and Latina/o students, groups traditionally marginalized in computer science. The summer program integrated computing education with two themes: Mesoamerican culture and conservation of endangered species. Such topics built upon students' ethnic heritage and interest in endangered species. This interdisciplinary, project-based curriculum was designed with the goals of recruiting students who may not have selected computer or robotics camps, and increasing students' interest, confidence and experience in computer science.

The curriculum focused on the skills necessary to complete a digital story-telling project in Scratch. This culminating project told a Mayan myth with an endangered species from the Mesoamerican geographic region. There were four activities categories: (1) learning about animals and conservation, (2)

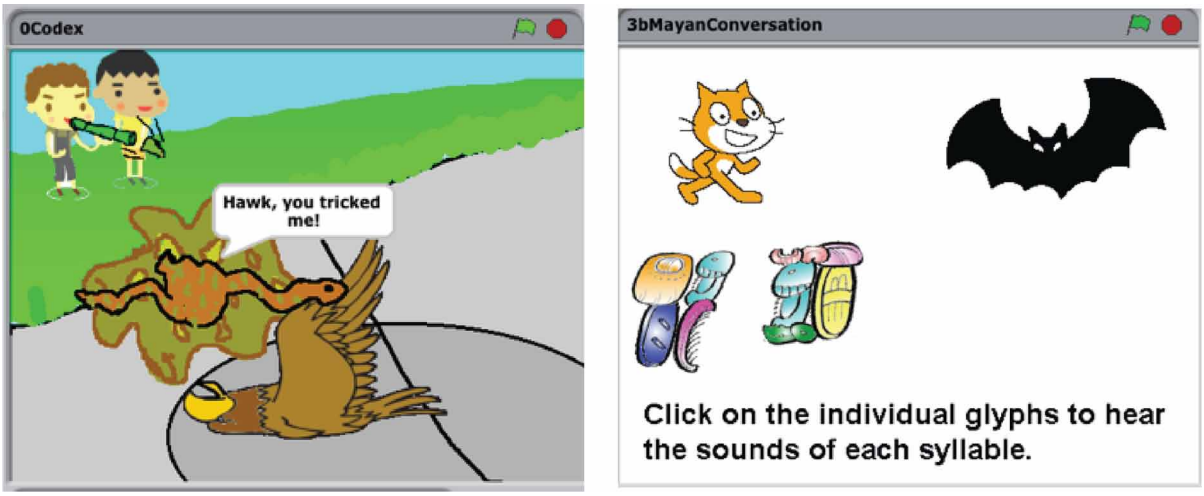
learning about Mesoamerican culture, (3) developing basics skills with art and drawing, and (4) introducing and developing programming skills. Half of each camp day was devoted to computer science projects, followed by two of the other subjects.

Students used the Scratch programming environment to complete their programming projects. Because the culminating project was a digital story, computer science content during week one consisted of lessons that taught skills and concepts needed to create digital stories. Working in pairs, students used Scratch to make a "name poem" acrostic for a selected animal (Wolz, Stone, Pulimood & Pearson, 2010). Each letter described the animal including information such as where it lived, what it ate, and ecological challenges to its survival. Students then created a digital story based on Mayan and Aztec stories. Figure 8 shows two screen shots from student work that are examples of how the summer camp activities integrated computer science and Mayan culture. The screen shot on the right depicts a myth about a snake who convinces a frog to let him eat him (to carry him) and is itself soon after eaten by a hawk. The screen shot on the left is student work from an activity where the students were provided the pictures of glyphs (Mayan writing) and a set of sounds. The students had to figure out which sound went with the glyphs, then program the glyph to say the sound when the glyph was clicked.

Each programming lesson had two parts: a warm-up exercise in which students learned one or more new computer science concepts with substantial scaffolding and support, and a small project in which students applied their new knowledge to a similar, but new, problem. In year one, the computer science topics included sequential execution, message passing, initialization, and the combination of loops and movement (complex animation). Lessons on loops, variables and branches were added in the second year.

Unlike the fourth through sixth graders, language and mathematics skills were not identified as a significant barrier to learning the computer programming. However, the students had some difficulty with the computer science ideas. Over the two years, the project directors made changes to the curriculum such as revising activities to provide more scaffolding and increasing opportunities to assess the development of student ideas. Programming ideas were introduced in targeted tasks and were built on

Figure 8. Screen shot of two examples of student work showing how computer science and Mayan culture were integrated into the summer camp activities



over time. The research on middle school students learning computer science indicated a need to revise the curriculum. The curriculum designers added detailed project checklists that included the objective of the project and the specific elements necessary to consider the project "complete." This provided a visual aid for students to recall lesson directions and expectations. A second change was the addition of guided peer sharing and review of projects, which provided an opportunity to share their work with other students. To facilitate this engagement, curriculum designers created "gallery walks" and scaffolded ways to provide constructive feedback to peers. Students were also taught two types of feedback: feedback about the way the project looked or acted and feedback that related to the computer science concepts used in the implementation of the project.

While the content of the camp included computational thinking and programming, the primary goal of the summer camp was to increase interest in computer science among females and underrepresented ethnic populations. The summer camp successfully increased both boys' and girls' interest as evidenced by pre and post assessments. The majority of the students were drawn to the camp because of the non-computer science topics (Animals and Mesoamerican Culture). In fact, in year one, only 31% of the students indicated that they selected the camp because of the computer science theme and only 22% indicated computer science as a possible career choice. After the camp 65% of the participants indicated computer science as a possible career choice. In the second year, 17% indicated computer science as a career choice before the camp and 44% indicated computer science as a possible career after the camp. That the students were drawn to the camp because of the non-computer science topics, but reported developing an interest in computer science points to the importance of combining computer science with other content areas to attract underrepresented populations.

DISCUSSION AND RECOMMENDATIONS

Connections to Content Standards

Given an appropriate task and coding environment, even elementary school children can become interested in and adept at coding. Further, computer programming can be linked to other content area standards in ways that allow children to develop programming skills while also learning mathematics, literacy, and science. Of the three studies discussed above, two (lower elementary and middle school) preceded the implementation of CCSS and NGSS. Thus, the following discussion of how these two projects addressed new mathematics and literacy standards is retroactive. In contrast, the study of upper elementary school students was conducted after the CCSS and NGSS had been written. The inclusion of engineering design as described in the NGSS was an intentional focus of the project. The connections to the CCSS and NGSS are only provided as an example of how instructors might think about using computer science instructions to link to other content areas. While each of the studies included computer science activities that could address standards in multiple content areas, only one set of standards for each grade level study is discussed.

The third grade students using Turtle Art were introduced to key concepts and skills for computer programming while learning mathematics and literacy that aligns with Common Core State Standards. In mathematics, the TurtleArt programming tasks provided opportunities for students to represent and

solve problems involving addition and subtraction, apply properties of operations as strategies to multiply and divide, draw shapes and their attributes, understand concepts of angles, and draw and identify lines and angles among others. By programming in TurtleArt, they received feedback in the form of drawings. For example, if a student tried to make a drawing of a triangle, but the angles provided for each turn the turtle made did not add up to 180 degrees, the drawing would not look like a triangle.

The curriculum for fourth through sixth grade students intentionally connected to the Next Generation Science Standards. As described above, engineering design is a new requirement for science at all levels of K-12 education. Programming is a natural fit for engineering design. Students completed "engineering design" lessons with the goal of designing, revising, and implementing a digital story. In the design thinking lessons, students were given an engineering problem (e.g., create a digital story). The classroom teacher selected a topic and target audience. After identifying the problem, students brainstormed ideas and collaborated to evaluate these ideas and decide on the best one. They then created storyboards and flowcharts for their digital story. Finally, students worked on programming their digital story by testing and revising iteratively. Also, teachers may have encouraged students to create a story showing a scientific process, demonstrate a science concept, or even create an interactive world where the user could engage in the science. By doing so, students deeply engaged in a core idea within the Next Generation Science Standards. For more details, see Hansen et al. (2015).

As an integral part of the middle school curriculum, students were assigned an endangered animal from the Mesoamerican region and researched their animal, including the environment it lived in and how it was depicted in Mayan art and culture. They used this information to design and program digital stories. This research – conducted prior to programming their digital stories – required students to read fiction and non-fiction texts, thereby engaging in practices and meeting standards included in the Common Core State Standards for English Language Arts. Further, the programming task itself met standards related to writing. In creating digital stories, students established a point of view and either told the story from a narrator's perspective or from the perspective of the characters (sprites in Scratch). They also organized an event sequence. These tasks are related to Common Core Standards for English Language Arts in the seventh grade.

Recommendations

There exist many websites and apps for teaching programming through block-based programming. In addition, there are non-block-based programming environments, robotics kits that include programming, and board games, and toys designed to teach programming. Choosing an appropriate app or web-based programming environment can be challenging, especially for use in schools. The studies discussed in this chapter used block-based programming environments that were scaffolded to be appropriate for the grade level. In designing instruction and researching children's learning, each project described identified the need to balance tasks designed to teach programming skills necessary to accomplish an end goal and open-ended tasks that provided opportunities for students to innovate on the skills they had developed. Across the three studies, the programming language, programming environment, and curriculum all needed to be considered to ensure that they matched with children's physical abilities and backgrounds in reading and mathematics (see also Duncan, Bell, & Tanimoto, 2014). Below, recommendations related to the programming environment and curricula are discussed.

Mathematics and Language Demands of Programming Environments

Programming environments for block-based languages include the visual interfaces that students interact with to create and view their programs. It includes the block palette, which may either display all available command blocks or a menu that includes categories of types of command blocks. The interface also includes a space where students construct scripts or groups of blocks arranged in sequence and a place to see the output (the "stage"). There may also be an area where sprites (programmable objects) can be selected and edited. This is a complex learning space. Educators need to consider the reading and mathematics requirements for students to use the programming environment. While the middle school project used the full version of Scratch at the time of the study, the two projects for elementary school used programming environments that simplified the Scratch programming environment.

Math skills ended up being an important consideration for selection of a programming environment because not all children in the fourth grade classrooms had the requisite math skills to be comfortable with the Scratch environment. They had not yet learned coordinate planes, percentages, relative angles or negative numbers. These ideas caused problems when, for example, children attempted to set the size of a sprite to be smaller or larger than the default. To do so, students select or type a percentage or decimal values of the original size. Moving and placing sprites required understanding the x and y coordinates and the coordinate plane. As a result, the project team made changes to the programming environment. However, an equally effective solution would be to select a programming environment that does not require these skills.

Language skills is a second important factor to consider. The programming language is comprised of the set of blocks or commands that are used to create programs. Block-based programming languages do not require much typing. However, while there are notable exceptions such as ScratchJr (Flannery et al., 2013), most require children to be able to read and develop new meanings for words that they may associate with other definitions. Words like "glide" and "go to" or "broadcast" and "say" may have similar definitions in everyday interactions, but result in different actions in a program. These subtle differences were difficult for students.

Because the skill levels and experiences of classes may vary considerably, it is important for educators to evaluate any programming environment and the programming language for the pre-requisite mathematics and language skills. They should choose a programming environment that does not include unnecessary barriers to programming and they should be prepared to teach the mathematics or language skills required. In the studies described above, this was handled in both ways. The language for the lower elementary students required students to understand angles, a topic they had yet to learn in third grade. The classroom teacher chose to couple programming activities with mathematics lessons so that, as the students learned about angles and geometric shapes, they could apply this learning in their programming activities. In the upper elementary school classroom, the interface was designed to remove many barriers to learning programming, so that participating teachers would not be required to adjust the teaching of other content areas.

Curriculum and Instruction

In all cases discussed in this chapter, students learned programming skills in order to accomplish creating something of their own design. The youngest children created drawings of geometric shapes while the upper elementary and middle school students created digital stories. The *creation* of something, not just

the learning of skills, was a central component. All three projects included both skill building directed exercises and open-ended activities that allowed more freedom and creativity and in all three cases, teachers and project staff grappled with achieving an appropriate balance that sufficiently scaffolded students' progress while also allowing students freedom to choose how they would use the programming. This was handled in different ways. In the elementary school classroom, much of the programming time was unstructured and students were encouraged to share what they learned with each other. The programming lessons were coupled with mathematics lessons and students were encouraged to practice their new mathematics learning. The teacher interrupted students for mini-lessons, often highlighting a new block or programming skill discovered by a student in the class. In the upper elementary school classrooms, structured skill-building activities taught programming skills that were later applied in larger open-ended projects. The programming skills learned in the structured activities could be used in their open-ended project, but the structured activities and open-ended project were separate tasks. At the middle school level, students learned the skills in the process of developing larger open-ended projects.

Structured tasks in which all students are working on the same project at the same time simplified the teachers' responsibility. Students faced similar difficulties to each other throughout the tasks. This meant that teachers only had to answer a handful of questions and they could leverage the expertise of children who finished early to assist those who were struggling. It also meant that the end goal was known. Open-ended tasks, in contrast, provided students with opportunities to follow their own interests and students sometimes had questions that teachers did not know how to solve. Further, every child could have a different question. In these cases, teachers needed to be comfortable learning along with the students or placing the responsibility to figure things out on the students.

The layout of the computer lab or classroom where instruction took place had implications for how classes were managed. Visual cues such as a post-it notes on the computer to signal to the teacher that they needed assistance allowed students to continue working. Moving young children to the front of the room (away from computers) or asking them to turn monitors off facilitated focus on group discussion or teachers' instructions. For more specific classroom management strategies see Franklin et al. (2015).

For the middle school students, the most important goal was to interest students in computer programming. In the study discussed here, this was done by integrating computer science with Mayan culture and endangered animals, topics of interest to the target populations. Hence, the focus was not on ecology or cultural studies as an end, or even technical skills in computer science, but rather the topics were used as a means to interest students in computing. The same approach was used with the upper elementary classrooms. Initially, all tasks were related to fourth grade science or social studies curriculum. In some cases, this was problematic. Requiring students to remember or look up ideas from other content areas slowed students down and distracted from the programming. As a result, the tasks were revised to provide all necessary information so that they were useful across a variety of classrooms. The level of integration of other content should be carefully considered to support learning in both computer science and the other content area.

Finally, as has been discussed earlier, access and equity are important to consider. Children have varied prior experiences that can lead to a range of comfort with technology. Failure to attend to these varied experiences may make children that are less comfortable with technology think that they are less capable that their peers who have had significantly more experience with computers.

FUTURE RESEARCH DIRECTIONS

There is scant research focused on how elementary and middle school children learn computer science, especially in school settings. Thus, there are many areas of needed research. One area is on identifying prior knowledge that can be built on to teach computational thinking. This is vital for developing appropriate curriculum and programming environments for this age group. Once prior knowledge is identified, research on learning on how children develop ideas in this area over time and how this learning is related to developmental level is critical. Much work has been done to identify learning progressions in science which can be a model for computer science education. A third area is how computer science learning at the elementary and middle school contributes to success in other learning contexts – both concurrently and to learning contexts at a later time. Research on how computer science knowledge enhances other content areas is important, as is research on what can be learned in elementary and middle school that will transfer to better success in computer science classes in high school and post-secondary education. Fourth, increasing the workforce diversity is a particular concern in computer science. Research is needed to understand how to support English language learners, students with special needs, students who lack access to technology, and female students in learning computer programming and computational thinking and developing and sustaining an interest in computer science.

As has been argued in this chapter, to reach all students, computer science education needs to be accessible to all children. This means, computer programming should be taught in classrooms, not only in out-of-school contexts. School environments differ from out-of-school environments in that all children participate, not just self-selected students, teachers are held accountable for addressing state and national standards, and classroom elementary and middle school teachers may neither be experts in computer science nor have sufficient free time to learn computer science at the level of out-of-school providers. How to support schools, teachers, and students to engage in meaningful computer science activities is an area of needed research. Related to this, future research should also focus on the knowledge that teachers need to support student learning of computer programming and mechanisms for providing professional development for practicing teachers and integrating this knowledge into teacher education programs for pre-service teachers.

CONCLUSION

Even from a young age, children are capable of learning computational thinking and programming. Further, learning computer programming allows them to innovate with technology, not just use technology. However, students' prior experiences with technology vary greatly. Even for those who have extensive experience using technology, transitioning from using technology to developing their own drawings, digital stories, or games through computer programming involve new skills. Many programming environments exist for teaching programming. Educators need to consider carefully the requirements of the environments and the prior experiences of their students ensure success. Programming tasks can connect children to each other and to a larger audience. As such, computer programming is a powerful tool for children. As Kafai and Burke note, "Learning to code ultimately manifests its worth when it increases an individual's capacity to participate in today's digital publics" (Kafai & Burke, 2014, p. 9).

REFERENCES

Barab, S., & Squire, K. (2004). Design-based research: Putting a stake in the ground. *Journal of the Learning Sciences*, *13*(1), 1–14. doi:10.1207/s15327809jls1301_1

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *ACM Inroads*, *2*(1), 48–54. doi:10.1145/1929887.1929905

Bell, T., Witten, I. H., & Fellows, M. (2009). *Computer science unplugged*. Retrieved February 10, 2015, from http://csunplugged.org

Bontá, P., Papert, A., & Silverman, B. (2010). Turtle, art, TurtleArt. In Proceedings of Constructionism. Paris, France: Academic Press.

Brennan, K., & Resnick, M. (2013, April). *New frameworks for studying and assessing the development of computational thinking*. Paper presented at the American Educational Research Association (AERA), Vancouver, Canada.

Collins, E., & Green, J. L. (1992). Learning in classroom settings: Making or breaking a culture. *Redefining student learning: Roots of educational change*, 85-98.

Computing Research Association. (2013). *Computing research association Taulbee survey*. Retrieved April 17, 2015, from http://cra.org/resources/taulbee/

Duncan, C., Bell, T., & Tanimoto, S. (2014). Should your 8-year-old learn coding? In *Proceedings of the Ninth Workshop in Primary and Secondary Computing Education* (*WiPSCE '14*) (pp. 60-69). New York, NY: ACM. doi:10.1145/2670757.2670774

Dwyer, H., Hill, C., Carpenter, S., Harlow, D., & Franklin, D. (2014). Identifying elementary students' pre-instructional ability to develop algorithms and step-by-step instructions. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (pp. 511–516). New York, NY: ACM. http://doi.org/ doi:10.1145/2538862.2538905

Dwyer, H. A., Iveland, A., Killian, A., Hill, C., Franklin, D., & Harlow, D. B. (2015, April). *Programming languages and discourse: Investigating the linguistic context in learning computer science during elementary school*. Paper presented at the American Education Research Association (AERA), Chicago, IL.

Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., & Resnick, M. (2013). Designing Scratchjr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 1-10). New York, NY: ACM. http://doi.org/ doi:10.1145/2485760.2485785

Franklin, D., Conrad, P., Aldana, G., & Hough, S. (2011). Animal Tlatoque: Attracting middle school students to computing through culturally-relevant themes. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 453–458). New York, NY: ACM. http://doi.org/ doi:10.1145/1953163.1953295

Franklin, D., Conrad, P., Boe, B., Nilsen, K., Hill, C., Len, M., … Waite, R. (2013). Assessment of computer science learning in a scratch-based outreach program. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (pp. 371–376). New York, NY: ACM. http://doi.org/doi:<ALIGNMENT.qj></ALIGNMENT>10.1145/2445196.2445304

Franklin, D., Hill, C., Dwyer, H., Iveland, A., Killian, A., & Harlow, D. (2015). Getting started in teaching and researching computer science in the elementary classroom. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 552–557). New York, NY: ACM. http://doi.org/ doi:10.1145/2676723.2677288

Grover, S., & Pea, R. (2013). Computational thinking in k–12: A review of the state of the field. *Educational Researcher*, *42*(1), 38–43. doi:10.3102/0013189X12463051

Hansen, A., Iveland, A., Dwyer, H., Hill, C., Franklin, D., & Harlow, D. (2015). Programming digital science stories: Computer science and engineering design in the science classroom. *Science and Children*, *53*(4), 60-64.

Harlow, D. B., & Leak, A. E. (2014). Mapping students' ideas to understand learning in a collaborative programming environment. *Computer Science Education*, *24*(2-3), 229–247. doi:10.1080/08993408.2014.963360

Hawthorne, E. K. (2011). Collaborating with CSTA for computer science education week. *ACM Inroads*, *2*(4), 25–26. doi:10.1145/2038876.2038886

Hill, C., Dwyer, H. A., Martinez, T., Harlow, D., & Franklin, D. (2015). Floors and flexibility: Designing a programming environment for 4th-6th grade classrooms. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 546–551). New York, NY: ACM. http://doi.org/ doi:10.1145/2676723.2677275

International Society for Technology in Education. (2007). *ISTE standards: Students*. Retrieved February 10, 2015 from http://www.iste.org/standards/standards-for-students

Kafai, Y. B. (2006). Playing and making games for learning instructionist and constructionist perspectives for game studies. *Games and Culture*, *1*(1), 36–40. doi:10.1177/1555412005281767

Kafai, Y. B., & Burke, Q. (2014). *Connected code: Why children need to learn programming*. Cambridge, MA: The MIT Press.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, *37*(2), 83–137. doi:10.1145/1089733.1089734

Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. In *Proceedings of the 38th Technical Symposium on Computer Science Education (SIGCSE '07)*. New York, NY: ACM. http://doi.org/ doi:10.1145/1227310.1227388

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with scratch. *ACM SIGCSE Bulletin*, *40*(1), 367–371. doi:10.1145/1352322.1352260

Margolis, J., & Fisher, A. (2003). *Unlocking the clubhouse: Women in computing*. Cambridge, MA: The MIT Press.

National Governors Association Center for Best Practices & Council of Chief State School Officers. (2010). *Common core state standards*. Washington, DC: National Governors Association Center for Best Practices, Council of Chief State School Officers.

NGSS Lead States. (2013). *Next generation science standards: For states, by states*. Washington, DC: The National Academies Press.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books, Inc.

Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning*, *1*(1), 95–123. doi:10.1007/BF00191473

Papert, S., & Harel, I. (1991). Situating constructionsim. *Constructionism*, *36*, 1–11.

Partnership for 21st Century Skills. (2011). *Framework for 21st century learning*. Retrieved February 10, 2015 from http://www.p21.org/our-work/p21-framework

Pilco, S. (1990). *The XO laptop in the classroom*. Puno, Peru: Sdenka Zobeida Salas Pilco.

Resnick, M. (2006). Computer as paintbrush: Technology, play, and the creative society. In D. Singer, R. Golikoff, & K. Hirsh-Pasek (Eds.), *Play=Learning: How play motivates and enhances, children's cognitive and social-emotional growth* (pp. 192–208). Oxford, UK: Oxford University Press. doi:10.1093/acprof:oso/9780195304381.003.0010

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., & Kafai, Y. et al. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67. doi:10.1145/1592761.1592779

Ryan, C. (2013). *Language use in the United States: 2011*. U.S. Census Bureau. Retrieved from http://www.census.gov/prod/2013pubs/acs-22.pdf

ScratchEd. (2014). *Scratch curriculum guide*. Retrieved February 10, 2015 from http://scratched.gse.harvard.edu/resources

Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cunniff, D., & Verno, A. et al. (2011). *CSTA K-12 Computer science standards: Revised 2011*. New York, NY: Computer Science Task Force.

Seiter, L., & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the 9th Annual International ACM Conference on International Computing Education Research (ICER '13)*. New York, NY: ACM. http://doi.org/ doi:10.1145/2493394.2493403

Tai, R. H., Liu, C. Q., Maltese, A. V., & Fan, X. (2006). Planning early for careers in science. *Science*, *312*(5777), 1143–1144. doi:10.1126/science.1128690 PMID:16728620

Wing, J. M. (2008a). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 366*(1881), 3717–3725. http://doi.org/<ALIGNMENT.qj></ALIGNMENT>10.1098/rsta.2008.0118

Wing, J. M. (2008b). Five deep questions in computing. *Communications of the ACM*, *51*(1), 58–60. doi:10.1145/1327452.1327479

Wolz, U., Stone, M., Pulimood, S. M., & Pearson, K. (2010, March). Computational thinking via interactive journalism in middle school. In In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 239-243). New York, NY: ACM. http://doi.org/ doi:10.1145/1734263.1734345

Wright, T., & Cockburn, A. (2005). Evaluation of two textual programming notations for children. In Proceedings of the Sixth Australasian Conference on User Interface (Vol. 40, pp. 55–62). Darlinghurst, Australia: Australian Computer Society, Inc.. Retrieved from http://dl.acm.org/citation.cfm?id=1082243.1082251

KEY TERMS AND DEFINITIONS

Coding: The process of creating scripts.

Command: An instruction given to a computer. In block-based programming environments, commands are represented by a block with a word or a phrase on it.

Computational Thinking: A problem solving method that can be used to represent problems in a way that computers can solve them. This method includes abstractions, algorithmic thinking, organizing data, and other skills.

Computer Programming: The process of planning, debugging, and creating scripts to accomplish a goal or task.

Digital Story: Using digital tools to tell a story. This can include computer animation, programming, video, photos, or other digital tools.

Development Environment: An environment in which one creates and runs programs. Most provide tools for text-based languages such as autocompleting and coloring keywords. For block-based languages, they contain the blocks that make up the language.

Programming Language: A set of syntax and key words used to create programs.

Script: In a block-based environment such as Scratch, a series of blocks which control the actions of a sprite.

Sprite: A programmable object visually represented by an image.