Initialization in Scratch: Seeking Knowledge Transfer

Diana Franklin[§], Charlotte Hill^b, Hilary A. Dwyer[§], Alexandria K. Hansen[§], Ashley Iveland[§],
Danielle B. Harlow[§]

[§]Center for Elementary
Math and Science Education
University of Chicago
1100 E. 58th St
Chicago, IL 60637
dmfranklin@uchicago.edu

^bGevirtz School of Education
^bDepartment of Computer Science
UC Santa Barbara
Santa Barbara, CA 93106-9490
{hdwyer, akillian, aockey, dharlow}@education.ucsb.edu
charlottehill@cs.ucsb.edu

ABSTRACT

With the growing movement to use visual block-based languages (VBBLs) in elementary and middle school classrooms, questions arise about the learning outcomes of such activities. While some schools are content to use VBBLs to spark interest and motivation for the future pursuit of computing, others are asking, “Does this early exposure produce knowledge that transfers to traditional text-based languages (TBLs)?” If transfer is a goal, then a corollary is, “How do we design the transition to maximize the transfer?” This paper focuses on initialization of state and variables, exploring the differences between Scratch and two TBLs: C and Java. Based on observations of 9-12 year old students in a VBBL curriculum, we identify four “pieces of knowledge” that are critical for C and Java but are not nearly as obvious in Scratch, including whether, when, and how to perform initialization. We conclude with suggestions for instruction and development environment that may improve transfer.

Keywords

Initialization; Scratch; VBBLs; K-6; knowledge transfer

1. INTRODUCTION

For decades, computer scientists have been developing activities for young students in an effort to engage them in computer science early, leading to many educational programming platforms[7, 14], activities[3, 13] and outreach programs[21, 9]. Two goals of such early exposure to computer science include recruitment[6, 15] and computational thinking skills. It is hoped that early exposure to computer science makes students more likely to either become computer scientists, work with computer scientists, or program as needed. Block-based programming languages were developed to allow students to learn fundamentals of programming without the overhead of typing or memorizing commands. An open question is how much of what they learn is useful only for programming in VBBLs, and how much helps them when moving to a TBL. In this paper, we restrict ourselves to *imperative* and *object-oriented* languages. This question has gained importance as the Hour of Code and other efforts have piqued the interest of elementary schools. Recent research has shown that some knowledge does transfer between

VBBLs and TBLs when “bridging” and “hugging” techniques to support transfer are used[1, 8].

Here, we consider transfer to occur if a student applies principles from one context to another, even when the learning environment and language details differ and consider the conditions of the two types of languages (VBBLs and TBLs) that support and hinder transfer. In this study, we closely examine initialization of variables and objects as accomplished in Scratch, C, and Java, and the relationship between these. The contributions of this paper are:

- Detailed comparison of the data models in Scratch and C/Java.
- Identification of key concepts that are integral to initialization in C/Java but overlooked by students in Scratch.
- Suggestions for the development environment and instruction that may improve transfer.

The rest of the paper is organized as follows. We provide background on visual block-based languages, prior work on transfer, and our theoretical framework in Section 2. Section 3 describes initialization in more detail. Sections 4 and 5 present our curriculum from a high level and the methodologies derived from educational research used in our study. We then present our results in Section 7. Finally, Section 8 contains our conclusions and future work.

2. BACKGROUND

There are several visual block-based development environments, including Alice[7], Scratch[14], and LaPlaya[12] with common attributes like, a drag-and-drop development environment to replace much of the typing, available commands displayed visually, images are favored over numerical and textual output, and “sprites” are created with a friendly interface hiding variables such as position, orientation, and size.

In our work, we use a variant of Scratch, a common programming environment for novices in a variety of contexts[16]. Scratch differs from more traditional, text-based languages in both control and data. First, while it has some similar control structures (loops, conditionals), the program paradigm is entirely event-driven, with no single thread of control in the program. Second, the “data” consists mostly of “sprites” represented by 2D images with attributes such as size, position, orientation, and visibility that can be manipulated.

Block-based languages have been used at the college and pre-college levels to provide an early introduction to computing with a transition to a traditional language either within the same first course (college)[8] or in a second course (pre-college)[1]. Two types of approaches have been used. Some courses present each concept first in a block-based language, and then in the traditional language. Other courses present a set of concepts in a block-based language, and then transition to a traditional language, revisiting each of those concepts in the new language.

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

SIGCSE '16, March 02-05, 2016, Memphis, TN, USA.

© 2016 ACM ISBN 978-1-4503-3685-7/16/03\$15.00

DOI: <http://dx.doi.org/10.1145/2839509.2844569>

The difference between visual block-based languages and “traditional” languages is debatable. In this work, we assume two major differences between the types of languages. First, the programming environment differs, with traditional languages depending on typing on a blank screen, and much less scaffolding of the available instructions and syntax. Second, the type of data operated on differs. VBBL projects are multimedia projects, operating mostly on graphical data (2D or 3D), and the languages provide primitives to manipulate these attributes (changing size, location, orientation, etc.); Traditional languages typically operate on strings and numbers.

Perkins and Salomon[18] define two techniques, “bridging” and “hugging,” that help students transfer knowledge from one context to another. Bridging refers to how the concept is taught in the first context. These principles can be applied to computer science when teaching general concepts within a single language. When presenting a new topic in the language, a bridging discussion would describe how the general concept is applied to other contexts (e.g. real-world or other languages) to prepare students for the use in other languages. This gives students a broader understanding of the concept, allowing them to more easily apply it to the new language. Hugging refers to how the concept is taught in the second language. Initial examples are chosen to make the second context as similar to the first as possible in order to encourage students to see the relationship between the two languages. More examples follow that show the different ways this concept can be applied to the second concept.

Dann et al.[8] applied bridging and hugging to a college-level blended Alice/Java course. Student achievement increased across the entire course - improving their final scores from 61% to 85% and 82% (with similarly large gains with every subject in the course) compared to a Java-only course. Another pertinent result was that in order to improve hugging, the block-based language sometimes must fundamentally change in order to allow for simpler correlating examples in the second language. In particular, parallelism and inheritance are not subjects that match well between Alice and Java at this time.

Armoni et al.[1] performed a study in two different classrooms. First, they taught students Scratch in a middle-school course. Then, some students enrolled in a high school CS1 course in which they were mixed with students who had not taken the Scratch course. They found that the students who had taken the Scratch course did significantly better (20 percentage points) than the other students in bounded and conditional loop execution, but there was no difference in variables and conditionals. In addition, teachers covered the material more quickly when students had already taken the courses. They found that students’ knowledge was very limited to the contexts in which they had been taught, lending credence to the importance of bridging and hugging.

One line of work seeks to build learning progressions for young VBBL learners. Seiter[19] used existing projects across different grade levels to identify how computational thinking concepts varied by level in their Progression of Early Computational Thinking (PECT) Model. However, existence of a block in a project is not evidence of learning, as Brennan et al.[4] found out when interviewing Scratch contributors.

In order to understand how students learn a topic, learning progressions have been proposed that illustrate a path (or choice of paths) that students go through in understanding a particular topic, with the final goal being some level of understanding appropriate to the student’s age. When thinking of computer science from a learning progressions perspective, it is tempting to consider the learning of VBBL programming at the bottom of the

```

int index = 0;
int index = x+5;
(a)
Sprite(int s, int dir, int x, int y){
    size = s;
    direction = dir;
    xPos = x;
    yPos = y;
}
Sprite mySprite(100, 90, 0, 0);
(b)

void init_sprite(sprite *sp, int s,
    int dir, int x, int y){
    sp->size = s;
    sp->direction = dir;
    sp->xPos = x;
    sp->yPos = y;
}
(c)
sprite s;
init_sprite(&s, 100, 90, 0, 0);

```

Figure 1: (a) Primitive type declaration / initialization, C/Java (b) Class constructor definition and use, Java (c) Struct function definition and use, C

progression leading to skills and understanding for traditional languages. However, programming in VBBLs may require *different* ideas and skills than TBLs, making the learning process more complex than the linear learning progression model depicts and more akin to transferring knowledge between related fields of knowledge than directly building knowledge. In fact, other scholars have made the argument that learning progression models, in general, are too simple to capture the learning process. Hammer and Sikorski[11] argue that a particular level of understanding requires different pieces of knowledge. These pieces of knowledge may not have a particular order to them and may be activated differently depending on the learning context.

This perspective learning is related to earlier work by Hammer[10] in which he proposed the idea that, rather than large concepts, learners had smaller ideas that are neither correct nor incorrect on their own, but may be activated appropriately or inappropriately by a context. An important perspective in this work is to break concepts down into smaller chunks. It is not sufficient to study whether students “learned loops” or could “transfer their knowledge of loops” or any other single concept. The idea of transfer (or learning) from this perspective becomes activating appropriate combinations of smaller ideas in a new context.

The goal of this study is to better understand initialization of variables and objects in Scratch and TBLs. More specifically, we seek to identify pieces of knowledge (from student work and observation) and hypothesize what aspects make transfer challenging without careful bridging and hugging.

3. VARIABLE INITIALIZATION

In this section, we describe initialization in Scratch compared to C and Java. We focus on two aspects: the variable types and mechanisms for initializing values, and the lifetime of variables. In the interest of space, we simplify the concepts. In addition, we do not evaluate whether it is *easier* or *harder*; we merely discuss the *differences*.

In C and Java, there are two kinds of variables. Primitive types use simple assignment operators to initialize (Figure 1a). Variables may also be grouped. In Java, a class provides the ability to define functions that operate on variables in an object of that class. In addition, a constructor, contains all initialization of variables in the class (Figure 1b). In C, the user may define an initialization function, but the system does not explicitly support it differently than any other function (Figure 1c).

Variables are created and destroyed throughout the program. The longest lived variables are global variables, which are created when a particular execution begins and destroyed when it ends. All variables should be initialized (written to) before they are used

Figure 2: Blocks that modify named variables

Figure 3: Blocks that initialize sprite attributes

(read from). If the programmer forgets to initialize before use in C, then the variable could have any value in it. For most variables, the value will begin with whatever value was already in that memory location – the ending value of another variable that is no longer present in the program. This leads to unpredictable behavior, and it is difficult to track down such bugs. In Java, however, uninitialized variables are detected by the compiler.

In Scratch, programming occurs in a combined development / runtime environment. There are also two types of variables. Variables can be declared explicitly by the programmer, and standard operators are provided for those (Figure 2). More common, however, are groups of variables in sprites. These are created with a “new sprite” button and given a first costume (image). When a sprite is created, it starts with a default set of values. Although the variable values may not be displayed directly, the visual effect is clear. Figure 3 shows blocks that can be used to initialize such attributes. There are two major differences between these blocks and traditional variable manipulation. First, the blocks are specific to the purpose (size, direction, pen down or up). Second, while some of them do have explicit variable names on which one can operate (e.g., x, y, size), many of them have no way to read the values, only set them (pen up or down, hide/show).

Scratch variables are created during program development rather than program execution. When the program is run twice in a row, the starting values are the same as the ending values from the previous execution. When the program is stored, the current values of all variables are also stored.

Finally, a complicating factor in initialization in Scratch is that there exists no definite “beginning” of a Scratch program. As stated previously, a Scratch program opens in its previously saved state. Scratch is an event-based program, so a program could consist entirely of events such as mouse clicks or key presses. The convention is to begin the program on the green flag, so this is where initialization would occur. However, because this is merely a convention, with no system enforcement, it may not be evident to novice Scratch programmers. In addition, there is no explicit support for initialization like a constructor, and programmers may not create functions.

4. CURRICULUM

Research on how children learn often depends on the curriculum that supports that learning. This is true of our work, as well. Our curriculum is intended to help students develop ideas about computational thinking with two types of activities: activities done off the computer (modeled on CS Unplugged activities[4]) and programming activities done on the computer. The computing concepts covered are the minimum set of concepts necessary to successfully complete a digital storytelling project in Scratch and included two lessons on initialization. The context of the projects were varied and selected to be interesting and accessible to our population of 4th graders (8-9 years old). We also included bonus projects for students who finish early and would like to explore more creatively.

Figure 4: The Animal Sprint stage after the race has occurred.

We included an explicit lesson on initialization (see Figure 4) called Animal Sprint. This lesson animates a race between a horse, rooster, and cat. The animals begin in the correct positions so that students can see where each animal should begin. Then each animal starts running when clicked. When the green flag is clicked to reset the race, only the horse moves to back its initial position at the starting line. The rooster and cat, in contrast, are not properly initialized. For the rooster, both the location and orientation need to be initialized because of its non-linear path to the finish line. For the cat, both the location and size need to be initialized because it stopped and ate several items, growing bigger, on its way to the finish line. In this study, we focus only on the initialization of the cat. Because of the difficulty students had completing this project, we added a second project (Piñata) that required only initializing position.

5. METHODS

We used design-based research, using both qualitative and quantitative data to simultaneously inform curriculum development, research, and practice[2, 4, 23], improving our curriculum as we learned about student learning. Design-based research is “a systematic but flexible methodology aimed to improve educational practices through iterative analysis, design, development, and implementation, based on collaboration among researchers and practitioners in real-world settings, and leading to contextually-sensitive design principles and theories”[23].

This study involved ten 4th-6th grade classrooms (aged 8-12) at five schools across California. In two of these schools (schools B and E), we collected only student project snapshots. In the other three schools (A, C, and D), we observed instruction and interviewed students. Schools varied in numbers of classrooms, grades participating, start dates, and order of projects.

Being present in three classrooms as participant observers[22] allowed us to discover concepts the students were struggling to learn. Researchers wrote field notes following each school visit. In addition, in both distant and local classrooms, for every project, we stored not only the final projects, but also snapshots during the development process. A snapshot was taken and stored into a zip file each time a project was run (*when green flag clicked*) if at least 4 changes had been made.

Computer science researchers read the field notes and inspected a subset of student projects to explore how the code deviated from what was intended. They then identified the specific features of projects that indicated a different view of initialization than we anticipated. After identifying these features, the entire set of student projects was automatically assessed to count how often those features manifested themselves in projects.

Figure 5: Resetting just before the sprint.

Figure 6: Resetting just after the sprint.

The computer science and education researchers then met together to discuss any conceptual issues, sources of those conceptual issues, and changes to make to the lessons, projects, and/or programming environment. These conversations focused on points where students made what appeared to be “errors” from the perspective of experts in computer science.

These conversations made visible understandings about initialization held by the computer scientists that derived from traditional languages and may not have been applicable to Scratch. The disciplinary knowledge of experts differs considerably from that of novices, making it difficult for experts to anticipate where novices will struggle and, in some cases, making them assume an unnecessary ordering of how information is learned, a phenomenon termed “expert blind spot” [17].

While the students programmed in a VBBL, the experts’ knowledge was developed and reified largely through TBLs. As such, our analysis of projects, fieldnotes, and challenges, brought to light differences between the VBBL and TBLs. In some cases, these differences made it difficult for the students to develop the knowledge useful for TBLs through the VBBL curriculum.

This process was repeated with each subsequent wave of classrooms. Finally, at the end of the process, the computer scientists and education researchers identified key pieces of knowledge that were necessary for initialization in TBLs, how those were expressed in Scratch, and the challenges in matching the two languages.

6. RESULTS

Our final results are the pieces of knowledge[10] derived from the lens of traditional languages but expressed in a way that is applicable to Scratch. We also present the different expressions of that knowledge observed in student projects and discuss the implications on transfer between Scratch and other languages. Finally, for some results (due to space limitations), we also present data as to how often students demonstrated use of concepts in their projects. We present this data to give a sense of the variance and challenge involved in teaching initialization rather than to make precise conclusions.

In this study, we focus on initialization of the cat in the activity called Animal Race. The correct script initializes two attributes: size and location. That is, the cat’s size is reset to 100%, and the cat is placed in the proper starting location.

Figure 7: Existence and timing of initialization for cat’s position in final snapshot of a given student’s project.

Figure 8: Existence and timing of initialization for cat’s position in any snapshot of a given student’s project.

For each project, we looked at two types of data - snapshots and final projects. We looked at final projects to gauge how students ended their programs, and we looked at all snapshots to identify what techniques students used along the way. Many students (47% at school B, 77% at school C, and 56% at school E) successfully initialized the cat to the starting line by their final submission, as seen in Figure 7. Variances in completion rates were due to the number of sessions schools spent on initialization, when they encountered the lesson (later schools’ curriculum had been improved), and the ages of the students.

6.1. Initialization Before Use

In traditional languages, it is imperative that the variable be initialized before it is used. In Scratch, however, this is not as obvious, since all variable states are stored between executions, and the expression of the variables (sprite attributes) are already visible before program execution. Therefore, we found three additional ideas that students needed to understand in order to initialize in Scratch. Although perhaps not obvious, it is actually critical in Scratch to initialize at the beginning rather than the end – execution can be stopped at any time, so only initialization at the beginning is guaranteed to occur.

1a: Initialization cannot occur at the end of a program.

Many students initialized at the very end of the race. Figure 5 is a script that initializes at the beginning of the script, and Figure 6 is an example of initialization at the end of the script.

Figure 8 shows all the places initialization occurred during students’ development. We can see that, at one time or another, 26% of the students from school C placed initialization after the sprint. In addition, almost 10% introduced a race condition

Figure 9: Expected script.

Figure 10: Initializing before race, assuming end position of race.

because they used *when sprite clicked* rather than *when green flag clicked* as the trigger event (creating a second *when sprite clicked* script that ran in parallel with the given sprint script). Finally, several students in classes B and E altered their sprint code enough during their development that the cat no longer reached the finish line (if sprinting from the starting line).

Figure 7 shows that only school C, which spent 3 sessions on Animal Sprint and received very targeted help, had a majority of students complete initialization before the sprint. About half of the projects in the other classrooms had no initialization at all.

The frequency of students initializing after execution shows the effect that carrying the state through alters students' ideas of initialization. It makes sense to students to initialize, or "reset" the state at the end because that will be held. Therefore, unless students are taught explicitly to initialize at the beginning, students may attempt to initialize at the end in traditional languages, as well.

1b: Initialization should occur on the green flag

In TBLs, the only requirement is initialization before use. In Scratch, however, all attributes of sprites are visible, even before the program is running. When, then, should initialization occur? By convention, execution starts when the green flag is pressed, so initialization should occur as the first actions in the green flag script (Figures 9 & 10); all other events are part of the running program. Many students, instead, initialized with the *on cat clicked* script, which made initialization occur immediately before or after the race.

1c: Initialization should be hidden from the user.

In Java and C, not only does initialization occur first, but it is a separate, unique action. Constructors are special pieces of code that are not explicitly called by the user or programmer: the system calls them automatically when a new variable is created. This challenged us to find a way to express the specialness and separateness of initialization in an environment in which the expression of the values are already visible before execution.

We identified this problem through iterative observation and discussion. Initially, several students used blocks that were unexpected to the computer scientists on the team. For students who placed initialization code before the action (e.g., prior to the sprint in Animal Sprint or at the start of the Piñata Project), they chose one of two block types: *go to* (instantaneous) or *glide* (a visible, slow movement).

Figure 11 shows what blocks students used in their final submissions of the Animal Sprint and Piñata projects. In schools C and D, all students used the *go to* block, whereas at schools B and E, many students used the *glide* block. In School D, for the Piñata projects, the presenter explicitly pointed out that it looked "silly" for the candies to slowly glide up to the piñata. Therefore, students had explicit instruction to use the *go to* block. At school E, no directions were given. In discussions about this concept, we discovered that the computer scientists' expectation of

Figure 11: Instantaneous vs gradual initialization in last submission only for projects that included initialization before the action

instantaneous blocks was a manifestation of the belief that initialization is special and should not be confused with normal execution. Those who were encountering initialization only through Scratch (education researchers and students) did not see this distinction.

The important question is whether teaching this distinction would help students transfer their knowledge to TBLs. More specifically, would teaching that initialization is a special, distinct action that occurs at the beginning of the program and should not be viewed as part of the program, help transfer their knowledge? While teaching that seeing the initialization "looks silly" may have resulted in the students using the instantaneous blocks, it is unlikely that students learned any generalizable concept.

6.2. Initialization does not use prior state

In traditional languages, nothing can be assumed about state prior to initialization. In Scratch, the state is dependent entirely on that variable's end state. This influences the instructions / blocks used for initialization.

2a: Absolute blocks need to be used rather than relative blocks.

In traditional languages, absolute instructions (e.g., `int x = 5`) must be used for initialization (as opposed to `int x += 5`). Likewise, in Scratch, there are absolute blocks (set size to X%) and relative blocks (change size by X%).

In Scratch, because initial state is the same as end state, the case for absolute blocks is not obvious. Figure 8 shows initialization based on an absolute "go to" block, whereas Figure 9 shows a script that moves the cat back to the starting point assuming it had been left at the end point. This works only in the most expected use case: the program is running from beginning to end each time. Unfortunately, this does not work properly if someone accidentally presses the green flag twice without waiting for it to complete, or if someone stops execution in the middle.

Table 1: The four pieces of knowledge identified that results in robust implementations in both Scratch and traditional langs.

Initialization should occur at the beginning of the program
Initialization should occur in the green flag script
Initialization should be hidden from the user
Absolute blocks must be used to initialize

7. DISCUSSION

These findings can be analyzed from several levels. First, we identified several "pieces of knowledge", summarized in Table 1, that signify principles about initialization derived from domain

experts' expectations of Scratch projects. We found that because Scratch saves state across executions and the state is visible as sprite attributes, even the most basic pieces of knowledge of initialization in traditional languages, "Initialize before Use" and "Assume no Prior State", are not obvious to a novice Scratch programmer. Other attributes of Scratch, such as the lack of a definite starting point and the variety of blocks used to alter state, makes it much more difficult to identify exactly where to put the code (the event) and which blocks to use (absolute vs relative).

One might be tempted to conclude that these are irrelevant, especially when using Scratch with no desire for the knowledge to transfer to other languages. Three out of the four principles, however, apply to Scratch projects themselves – only the question of instantaneous vs. gradual initialization has no clear answer in Scratch. Because students appear to design for the simple case (executing the project to the end each time) they often initialize in locations that do not work in other conditions. Once those conditions are encountered, it is difficult to fix their program, especially if they are not present to move everything back.

What about if transfer is one of the goals? We found that, despite explicitly teaching initialization, these principles were not obvious to the students. Therefore, the development environment, language, and instruction will need to be revised to better align these concepts. Within Scratch, students could be shown what happens when a project is interrupted part way through in order to develop a practice of initializing first and with absolute blocks. In addition, bridging should occur to present more general, high-level principles including the separate, special nature of initialization. It should occur in the green flag script (or a special script created just for initialization), and perhaps it should be instantaneous. In addition, a Scratch variant could provide a separate event for initialization that must be executed before the green flag. Finally, hugging could occur when students encounter traditional languages to relate what they learned in Scratch to what they see in the new context. Students need to be explicitly taught that variables are destroyed every time the program ends, so they need to be initialized anew each execution.

8. CONCLUSIONS

We have discussed the details of initialization of state as they are expressed in Scratch projects, presented several pieces of knowledge necessary to understand initialization, and suggested instructional strategies and Scratch modifications that could increase the chances of successful knowledge transfer.

Initialization is just one of several important concepts for creating digital storytelling projects. Similar analysis needs to be undertaken for sequential execution, events, message-passing, costumes, and scene changes, identifying how 4th graders learn these concepts in VBBLs, and how that might affect their understanding of traditional languages.

9. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation CE21 Award CNS-1240985. We would also like to thank all of the teachers, students, and schools involved in this project.

10. REFERENCES

- Armoni, M., Meerbaum-Salant, O., and Ben-Ari, M. 2015. From Scratch to "Real" Programming. *Transactions on Computing Education*. 14, 4, Article 25 (February 2015)
- Barab, S. and Squire, K. Design-based research: Putting a stake in the ground. *The Journal of the Learning Sciences*, 13(1):1–14, 2004.
- Bell, T., Witten, I. H. and Fellows, M. *Computer Science Unplugged*. 2006.
- Brennan, K. and Resnick, M.. New frameworks for studying and assessing the development of computational thinking. In *AERA*, 2012.
- Brown, A. L. Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. *The Journal of the Learning Sciences*, 2(2):141–178, 1992.
- Lori Carter. 2006. Why students with an apparent aptitude for computer science don't choose to major in computer science. *SIGCSE Bulletin* 38, 1, 27–31.
- Dann, W., Cooper, S., and Pausch, R. Making the connection: programming with animated small world. ITiCSE, 2000.
- Dann, W., Cosgrove, D., Slater, D., Culyba, D., and Cooper, S. Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd SIGCSE Symposium*. ACM, New York, NY, 141–146, 2012.
- Franklin, D., et. al. Animal Tlatoque: attracting middle school students to computing through culturally-relevant themes. In *Proceedings of the SIGCSE Symposium*. 2011.
- Hammer, D., Elby, A., Scherr, R. E., & Redish, E. F.. Resources, framing, and transfer. *Transfer of learning from a modern multidisciplinary perspective*, 89-120, 2005.
- Hammer, D., and Sikorski, T. Implications of complexity for research on learning progressions. *Science Education*, 99(3) 424–431, 2015.
- Hill, C., et al. Floors and Flexibility: Designing a programming environment for 4th-6th grade classrooms" In *SIGCSE '15*. ACM Press, 2015.
- Hood, C. S., and Hood, D. J. Teaching programming and language concepts using LEGOs. In the *Conference on Innovation and Technology in Computer Science Education* (ITiCSE), June 2005.
- Maloney, J., et al. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):1–15, Nov. 2010.
- Margolis, J., and Fisher, A. *Unlocking the Clubhouse: Women in Computing*. MIT Press, Cambridge, MA, 2003.
- Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. Learning Computer Science Concepts with Scratch. In the *International Workshop on Computing Education Research (ICER10)*, Aarhus, Denmark, pp. 69-76, 2010.
- Nathan, M.J., and Petrosino, A. Expert blind spot among preservice teachers. *American educational research journal*, 40(4), 905-928, 2003.
- Salomon, G., and Perkins, D. Teaching for transfer. *Educational Leadership*, 22-32, September, 1988.
- Seiter, L. and Forman, B. Modeling the learning progressions of computational thinking of primary grade students. In the *International Workshop on Computing Education Research (ICER13)*. ACM, 2013.
- Spradley, J. P. Participant observation. Holt, Rinehart and Winston, 1980.
- The stars alliance: A southeastern partnership for diverse participation in computing. NSF STARS Alliance Proposal. <http://www.itstars.org/>.
- Vygotsky, L. *Mind in society: The development of higher psychological processes*. Harvard University Press, 1978.
- Wang F., and Hannafin, M. Design-based research and technology-enhanced learning environments. *Educational Technology Research and Development*, 53(4):5–23, 2005.