Informatics in Education, 2014, Vol. 13, No. 2, 307–321

DOI: http://dx.doi.org/10.15388/infedu.2014.10

Situating Programming Abstractions in a Constructionist Video Game

David WEINTROP, Uri WILENSKY
Center for Connected Learning and Computer-based Modeling
Learning Sciences, Northwestern University
e-mail: dweintrop@u.northwestern.edu, uri@northwestern.edu

Received: January 2014

Abstract. Research on the effectiveness of introductory programming environments often relies on post-test measures and attitudinal surveys to support its claims; but such instruments lack the ability to identify any explanatory mechanisms that can account for the results. This paper reports on a study designed to address this issue. Using Noss and Hoyles' constructs of webbing and situated abstractions, we analyze programming novices playing a program-to-play constructionist video game to identify how features of introductory programming languages, the environments in which they are situated, and the challenges learners work to accomplish, collectively affect novices' emerging understanding of programming concepts. Our analysis shows that novices develop the ability to use programming concepts by building on the suite of resources provided as they interact with the computational context of the learning environment. In taking this approach, we contribute to computer science education design literature by advancing our understanding of the relationship between rich, complex introductory programming environments and the learning experiences they promote.

Keywords: programming, computer science, constructionist video games, webbing, situated abstractions.

1. Introduction

With the ever-growing landscape of introductory programming environments, an important, yet unanswered (or at least not sufficiently answered), question is that of the relationship between a programming tool and the understandings it promotes. Does a student writing a program in Logo develop the same understanding of conditional logic as a student working in Scratch? Does learning to program with Alice, Snap!, or starting off with "Hello World" in Java all result in the same understanding of the programming concepts used? To a veteran programmer, the answer might be yes – a conditional statement is a conditional statement is a conditional statement; syntax might differ, but the underlying concept is constant. Studies investigating the affordances of different intro-

ductory programming tools often rely on direct comparisons between two environments; do students perform better on a post-test after they use environment A or environment B? (For example: Lewis, 2010). While much can be learned with this approach, it does not yield insight into the questions we pose above. By relying on post-test measures, we learn the outcome, but are unable to identify any explanatory mechanisms that can account for the differences. Answering these questions requires a different methodology and a different set of theoretical constructs. In this paper we present a study designed to address this issue, focusing on one specific type of introductory programming environment: program-to-play constructions video games. The goal of this study is to answer two, interrelated research questions:

(1) How are programming concepts encountered and used while playing a program-to-play constructionist video game?
(2) How do features of the game and the gameplay context contribute to a player's developing understanding of these programming concepts?

Using Noss and Hoyles' (1996) constructs of webbing and situated abstraction, we analyze programming novices playing RoboBuilder (Weintrop and Wilensky, 2012), a program-to-play constructionist video game of our own design, to answer these questions.

2. Literature Review

The idea that young learners should be taught to program, and that doing so has far reaching benefits, originated with the Logo Project (Feurzeig *et al.*, 1970; Papert *et al.*, 1979). Papert (1980) found that "when a child learns to program, the process of learning is transformed. It becomes more active and self-directed…The new knowledge is a source of power and is experienced as such from the moment it begins to form in the child's mind" (p. 21). This view was part of a larger vision of the transformative potential of computers to fundamentally changing how learning takes place. diSessa (2000) argues that it is not just the act of programming, but also the medium, that promotes this new form of thinking: "Programs are not just analytic and a basis for reasoning. They are also synthetic. They can be run…Programming turns analysis into experience and allows a connection between analytic forms and their experiential implications" (p. 34).

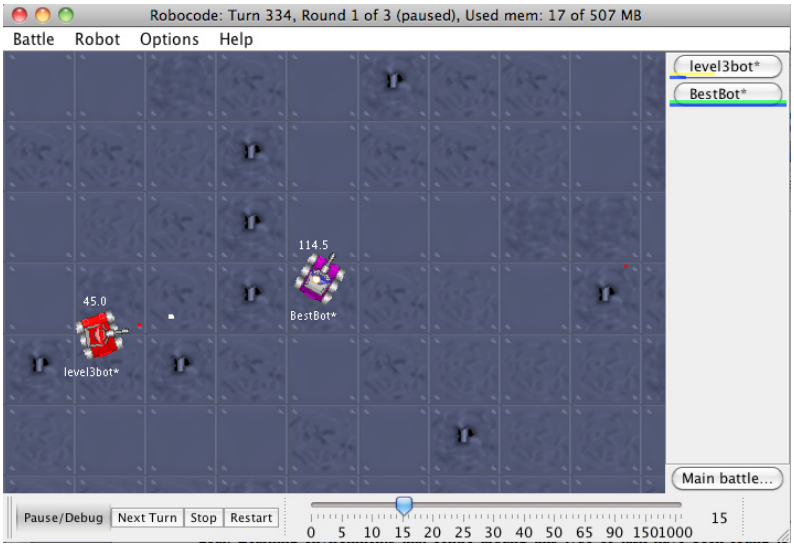
Studying the process of learning to program begins with a programming language. For Papert it was Logo, for diSessa it was Boxer. Logo, Boxer, and other early languages designed for novices, such as Smalltalk (Kay and Goldberg, 1977), seeded an ever-growing tree of low-threshold, high-ceiling languages. New languages and environments brought with them new ideas and innovations for making programming accessible and empowering for younger learners. NetLogo (Wilensky, 1999) introduces learners to emergent phenomena through programming large numbers of agents, making it possible for novice programmers to explore the dynamics of complex systems. Scratch (Resnick *et al.*, 2009), using a graphical programming interface, allows users to construct programs with only a mouse and provides an online forum for learners to explore, share, and remix programs written by others. Programming tools like Tern

(Horn and Jacob, 2007) use physical manipulatives to make writing programs a hands-on activity. The computer science education community has also created and studied many introductory programming tools (for a review, see Kelleher and Pausch, 2005). Across these and other introductory programming environments, a large body of literature has emerged chronicling the thinking and learning that takes place as they are used. Papert's early work with Logo focused on types of mathematical thinking that develop in young learners through working in computational settings, arguing for promoting heuristic concepts like problem simplification and debugging over mathematical formalisms and terminology (Papert, 1972). This work launched multiple avenues of study, including the development of mathematical meaning with Logo and other computational microworlds (Noss *et al.*, 1997; Noss and Hoyles, 1996), as well as the growth of programming practices like debugging and other general problem solving strategies (Clements and Gullo, 1984). Low-threshold programming environments have also been used to study thinking and learning in other disciplines beyond math and computer science (Blikstein and Wilensky, 2009; Goldenberg and Feurzeig, 1987; Wilensky and Reisman, 2006). It is also important to mention the successes constructionist programming environments have had on shifting attitudes and perception of programming among girls and other underrepresented populations (Bruckman *et al.*, 2002; Maloney *et al.*, 2008).

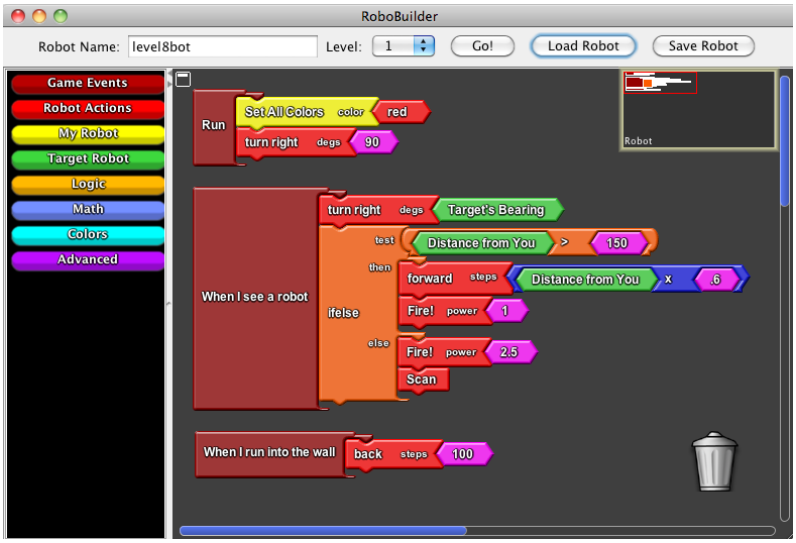
3. Constructionist, Program-to-Play Video Games

To study the relationship between the understanding that develops in learners and the programming language, environment, and activity, we use a program-to-play, constructionist video game (Weintrop *et al.*, 2012). Constructionist video games bring central constructionist ideas (learner-directed exploration, personally meaningful constructions, emphasis on powerful ideas) to the video game design genre. Program-to-play video games are a specific form of constructionist video games that make the writing of programs the central activity of gameplay (Weintrop and Wilensky, 2014). We chose this type of environment as it provides a rich set of features for learners to leverage, including a blocks-based programming interface, a domain-specific programming language, a visual execution environment, and a familiar form of digital interaction. Collectively these features provide a productive context for analyzing how design features of the environment affect novices' emerging understandings of programming concepts.

RoboBuilder (Fig. 1) is a program-to-play constructionist video game that challenges players to design and implement strategies to make their on-screen robot defeat a series of progressively more challenging opponents in one-on-one battle. To be successful, players must define instructions for their robot to locate and fire at their opponent while avoiding incoming fire; the first robot to make its opponent lose all of its energy wins. RoboBuilder's interface has two distinct components: a programming environment (Fig. 1b), where players define and implement their robot's strategy; and an animated robot battleground (Fig. 1a), where players watch their robot compete. Players first interact with the programming interface to define their robot's behaviors



a)



b)

Fig. 1. RoboBuilder's two screens: a) the battle screen, b) the construction space.

before launching the battleground screen. To program their robot, players are provided with a domain-specific, blocks-based programming language that includes basic robot actions, such as '`turn right`' and '`fire!`'. RoboBuilder is a component-oriented microworld that gives players the ability to "build and think in terms of objects that are close to their domain of interest" (Kynigos *et al.*, 1997, p. 231). RoboBuilder builds on two open source projects: Robocode (Nelson, 2001) and OpenBlocks (Roque, 2007).

4. Methods

The data in this paper were collected in hour-long RoboBuilder sessions during which programming novices played the game in the presence of a researcher. Sessions begin with the researcher introducing participants to the game, which includes describing the game objective and features of the language. The gameplay protocol follows a three-phase iterative cycle. First, participants are asked to verbally articulate their gameplay ideas and intentions. Next, participants work in the programming interface, constructing programs to carry out the ideas they just stated. The third phase of the protocol begins with the launching of a battle. As their robots compete, participants are asked to describe what they see their robot doing and whether or not it is behaving as expected. The end of the battle marks the completion of the cycle. The next iteration begins with participants explaining the next round of modifications or additions to their robot strategy they wish to carry out. This protocol is designed to elucidate players' developing understanding of programming concepts over the course of the interview. Twelve subjects were recruited to participate in this study. Older participants were recruited from a university in a large American city. High school-aged participants were recruited through their affiliation with a community center in the same city that serves a predominantly African-American, low SES community. Table 1 provides basic information about each participant's RoboBuilder session(s).

5. Theoretical Framework

The analytic lens we bring to this work is built on a pair of interrelated theoretical constructs. In their analysis of mathematical meaning making in interactive computational

Table 1
Information on the twelve participants included in this study

Name	Grade	Time Played	# Robots Authored	Highest Level
Jeff	9th grade	37:48	14	7
Benjamin	10th grade	31:15	4	2
Daniel	10th grade	44:47	8	7
Allen	11th grade	1:01:31	6	2
John	11th grade	37:27	16	2
Jane	1st year undergrad	39:32	10	1
Ruth	2nd year undergrad	47:15	7	2
Anne	3rd year undergrad	54:19	10	6
Morris	3rd year undergrad	46:03	16	7
Beth	4th year undergrad	3:47:06	46 (across 4 sessions)	7
Joseph	Graduate student	45:59	9	5
Bram	Graduate student	1:00:14	6	6

environments, Noss and Hoyles (1996) developed the construct of *webbing* to capture the rich, diverse and interrelated features of constructionist environments that provide support to the leaner. Webbing describes "a structure that learners can draw upon *and reconstruct* for support – in ways that they choose as appropriate for their struggle to construct meaning" (p. 108). The construct of webbing is intended to capture the full network of supports provided to the learner, not just a single scaffold within the environment, allowing the designer to study the learning process as it emerges through the use of the features of the environment in concert, as opposed to elements used in isolation. Thus, researchers can remain faithful to the recognition that learning is not uniform across pupils, but is unique to the individual and provide a way for researchers to capture the nuance of the learner's activity within a rich computational context.

The second theoretical construct we use in our work provides a way move from the webbing of a specific interaction to the general concepts and practices of the domain of interest. The construct of a *situated abstraction* was developed in order to "afford a means to describe and validate an activity from a mathematical vantage point, without *necessarily* mapping it onto standard mathematical discourse" (Hoyles and Noss, 2004, p. 2). In interacting with computational learning environments "learners web their own knowledge and understandings by actions within the microworld, and simultaneously articulate and mesh fragments of that knowledge – abstracting within, not away from, the situation" (Noss *et al.*, 1997, p. 228). Situated abstractions give us a way to both attend to the situated nature of the activity within the webbing of the environment, while also recognizing the ability of concepts to transcend contexts, and thus providing a way to link in situ activity with more general, abstract forms of conceptual knowledge. In bringing this lens to the analysis of a RoboBuilder, we can see how learners forge connections with the features of the tool and the computational meaning they carry, and interpret and ascribe meaning to this process.

6. Programming Abstractions in RoboBuilder

In this section, we present our analysis of the RoboBuilder sessions. We coded the sessions to see if and how participants encountered and used four specific programming concepts: object state, conditional logic, iterative logic, and flow of control. For each concept, we provide a brief example of a learner encountering it during gameplay, then report on its frequency across the full set of participants. Building on the constructs of webbing and situated abstraction, we link the use of these concepts back to RoboBuilder's interface and the gameplay activity as part of the discussion for each concept. We coded for the use of the concepts throughout the RoboBuilder session, including the planning, construction and evaluation phases of the interview protocol. We begin this section with a vignette to provide a sense of how components of RoboBuilder's webbing were appropriated to situate one learner's emerging understanding, then continue with our analysis of the four programming concepts.

The following interaction occurred early in Beth's interview, during her second battle against the level one opponent whose strategy is to remain motionless until its energy

drops below 50, at which point it begins to move randomly. After seeing her opponent come to life during the first battle, Beth asks:

> *Beth: Do you know when this mysterious other thing is going to happen?*
>
> *Interviewer: It happens at 50.*
>
> *Beth: It happens when it reaches 50? OK, so that robot must have something built into it when it reaches 50. Oh! There we go, so that's what the, that's what the other boxes are for, so like if you reach a certain health level you can change the actions, oh, ok. I didn't understand that.*

In this exchange, we can see the invocation of two programming concepts to explain in-game behavior, and start to get a sense for how the webbing of the game helped situate their use. The two programming concepts Beth employs in this example are object state and conditional logic. Through her statement: "*If you reach a certain health level"* we can see both of the two concepts invoked. First is the recognition that robots have a health level, which is a value that serves to describe a characteristic of the robot (i.e. defines its state). Second, in starting her statement with *"if"* and then describing the consequences for a given condition being reached (attaining a certain health level), she uses conditional logic to explain how to create the observed behavior. Interestingly, Beth's explanation of how to use the programming blocks to create this behavior matches the actual program controlling her opponent (Fig. 2).

A number of features of RoboBuilder contributed to Beth, a programming novice, being able to correctly employ these two programming concepts. First, the displaying of the available blocks in the programming interface provided a key resource in her using these programming constructs; she even refers to the blocks (what she refers to as "*boxes*") in her explanation. Second, Beth is describing the behavior of her opponent, not her own robot. Her being able to draw on opponent behaviors as a way to bootstrap her own understanding is a design strategy we have analyzed elsewhere (Weintrop and Wilensky, 2013) and constitutes another component of the webbing in which the concepts of conditional logic and object state were situated. A third critical aspect of the webbing Beth uses in this episode is the visual enactment of the battle. While Beth did have all of the blocks explained to her during her introduction to the game, from this quote, it is clear that the first, out-of-context, explanation of the blocks was not suffi-

Fig. 2. The blocks that control the level-one opponent that Beth correctly articulated.

cient for her to understand their meaning. Her saying "*so that's what the other boxes are for"* followed by "*I didn't understand that"* highlights the difference between her being told what blocks do versus seeing their behaviors enacted during gameplay. This quote suggests that while she initially did not understand the utility of some of the blocks, through seeing them situated within the webbing of the game, their meaning emerged. Put another way, through the network of resources provided by RoboBuilder (its webbing), Beth was able to use the concepts of state and conditional logic to interpret game outcomes (i.e. situate these two programming abstractions).

6.1. *Programming Concept: Object State*

Object state is the knowledge that computational entities contain data in the form of property-value pairs that define the object at any given moment in time. In RoboBuilder, there are two types of object state we coded for: internal robot state, which pertains to information about the properties of the robot, such as its energy, heading, or speed, and battle-state, which captures the state of a robot during battle, such as being hit or seeing an opponent. The event blocks of RoboBuilder's language reflect the battle-states the robot can be in. The programming blocks, visual battleground, and larger game objectives and interview activity all contribute to the webbing in which the participants situated their developing understanding of object state in their programs.

In the vignette above, we saw Beth encounter object state as she thought through how her opponent used its energy level, a characteristic of its internal state. A second example of a novice attending to state can be seen early in Ruth's RoboBuilder session. In explaining a strategy for defeating her level one opponent she says: "*In level one, the robot doesn't move much, so if you're already facing the robot and hitting it, then there is not point in moving more.*" By saying: "*you're already facing the robot*", she invokes one form of object state, describing her robot in terms of what direction it is facing. She continues by saying "*and hitting it*", describing the second type of object state present in RoboBuilder, that of her robot having successfully hit its opponent. Here, Ruth, like Beth, draws on the visual enactment of the battle, RoboBuilder's language, and the overarching game objective as part of the webbing in which to situate the concept of object state.

6.2. *Programming Concept: Conditional Logic*

Conditional logic builds on the previous concept to allow a player to introduce differential behaviors based on the state of the robot or the state of the game. Above, we saw Beth encounter conditional logic while trying to make sense of the level one opponent's behavior, which moved only if its energy was below 50. In coding for conditional logic, we looked for players proposing different outcomes based on state or explicitly referencing or using RoboBuilder's conditional logic blocks. Conditional logic was also used by players when thinking through strategies, as we can see in this quote from Allen:

> *While I'm shooting at the other robot, if he misses, I'm pretty sure he'll have to still shoot because I'm pretty sure the point of the game is to hit the other robot… If [my robot] does get hit, I guess he's probably too close to the other robot, so I might have to tell him move back… If he has higher energy than the other robot, let's say, he's probably 50 higher, I'd probably just tell him to get closer to the robot and just start shooting 'cause he's got energy to spare.*

Here we see Allen laying out his robot's strategy as a series of conditional statements. In some cases, he depends on the battle-state to dictate his robot's behavior ("*if he misses…*"); in other cases Allen uses robot-state to inform his robot's behavior ("*if he has higher energy…*"). For Allen, the video game context served as an important resource for him to situate his use of conditional logic – by drawing on the webbing of the game (the game objects, the nature of gameplay, the in-game objects themselves) – he was able to articulate a potential robot strategy built around the concept of conditional logic.

6.3. *Programming Concept: Iterative Logic*

Iterative logic can be used to repeat commands either a fixed number of times or until a specified condition is met. In our analysis, this code was applied when participants referred to an iterative aspect of the game (such as an in-game event repeating) or when either of RoboBuilder's two iterative blocks (`while` and `repeat`) were used or discussed. We can see how the webbing of RoboBuilder supported the use of iterative logic and the situated nature of the in players in-the-moment understanding of it in how Joseph resolved the problem of his robot getting stuck against the wall of the battleground. Instead of constructing a sequence of instructions that would run once to solve the problem, Joseph composed a strategy that relied on iteratively running as many times as necessary. When his robot hit a wall, he gave it the instructions: `back 10` then `turn right 30` then `forward 50`. This caused his robot to hit the wall repeatedly, turning a little bit each time. Upon completion of the implementation of his solution, Joseph explained: "*this way [my robot will] back up and kind of parallel park away…till he's not at the wall anymore.*" Here, Joseph devised a strategy that relied on iteratively running as many times as necessary to move his robot away from the wall. In comparing the maneuvering of his robot to the act of parking, he includes parallels between in-game and out-of-game events as part of the webbing he uses to build his understanding of the concept.

6.4. *Programming Concept: Flow of Control*

Flow of control captures the knowledge that programs are executed sequentially and serially. In analyzing the interviews, the flow of control code was applied when players discussed the execution order of blocks either within an event or across game events. In some cases, the concept of flow of control was confronted directly, including instances of participants systematically testing out the order of execution of the program by chang-

ing the order of blocks within their program then running it to see how the behavior changed. In other cases, players encountered flow of control by trying to understand how and when different events ran. For example, when Bram was trying to reason through his implementation of `When I see a robot`, he slowed down the execution of his program, focusing specifically on how his robot behaved after first spotting its opponent. "*So when* [the `When I see a Robot` event] *finishes, it moves up and does* [the `Run` event]." Here we see Bram make a statement about how flow of control moves from one event to another, relying on the ability to slow down the speed at which his program executes to help him figure it out. Like with the other programming concepts, the visual execution, iterative nature of gameplay, and the ability to control the speed of program execution all contributed to the webbing in which he developed his understanding of the concept of flow of control.

6.5. *Programming Concepts – Frequency Across Participants*

Table 2 shows the results of our coding the full set of interviews. Three things about this table are noteworthy. First, every participant encountered at least one programming concept during their session, with most participants encountering a majority of the concepts. Object state was observed in all 12 of the participants' interviews, while flow of control and conditional logic were both used at least once by a majority of participants. Iterative logic was the least frequently encountered concept of the four we coded for, but was still seen in half of the sessions. Participants encountered the coded-for programming concepts an average of 19 times during their hour-long RoboBuilder sessions.

Table 2

Frequency of each programming concept across the full set of participants

Participant	State	Conditional logic	Iterative logic	Flow of control	Total
Jeff	10	0	0	0	**10**
Benjamin	9	0	0	0	**9**
Daniel	10	0	1	4	**15**
John	6	0	0	1	**7**
Allen	12	7	0	7	**26**
Jane	17	2	0	1	**20**
Ruth	10	7	1	2	**20**
Anne	15	3	1	3	**22**
Morris	17	4	1	3	**25**
Beth	12, 15, 14, 23	3, 1, 6, 2	4, 2, 1, 1	2, 4, 2, 1	**21, 22, 23, 27**
Bram	10	0	0	5	**15**
Joseph	10	3	3	7	**23**
Mean	12.7	2.5	1	2.8	**19**
Total	**190**	**38**	**15**	**42**	**285**

A second item of interest is the relatively high frequency of players employing the concept of object state. This can be attributed to the nature of the in-game objective and the accessibility of the concept. The central activity of controlling a robot, paired with the visual execution of the battle and the design of the protocol, facilitated players attending to the current state of the robot in terms of its position, energy, and the events that did (or could) occur during battle. Reference to any of these constituted an invocation of the concept of state. Additionally, when players were asked to describe what they were observing while watching their robot compete, comments frequently referred to different components of the game-state or robot-state. We interpret this to mean that object state was more accessible and central to the gameplay activity than the other concepts and see it as evidence for how the design of a the environment and the webbing it provides can foreground certain concepts over others.

Finally, these data show that as participants advanced in their schooling, the frequency of use of the concepts increased. Pre-university aged participants (rows 1–5) used an average of 13.4 concepts during their RoboBuilder session, while university aged participants (rows 6–12) used an average of just over 21 concepts. This seemed especially true for the iterative logic concept as only one of the pre-university participants used it, while all but two of the older participants did. While these data do suggest a developmental trend for the use of programming concepts in a program-to-play game, as this was not the focus of the study, we hesitate to make any strong claims of a developmental nature and instead note it as a possible avenue for future research.

7. Discussion

Programming concepts are often taught removed from a meaningful, authentic contexts, resulting in the observation that students know "the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs" (Winslow, 1996, p. 17). Research in the learning sciences shows that creating a meaningful context is important for learning (National Research Council., 2000), a finding replicated in the CS education literature (Cooper and Cunningham, 2010; Guzdial, 2010). In our constructionist, program-to-play game, learners encounter programming concepts situated within a context that provides a rich array of resources upon which learners can interpret and employ them. The language primitives developed meaning for the players through the iterative construction process central to gameplay. Providing such webbed contexts aid in the meaning-making process, as "these meanings become reshaped as learners exploit the available tools to move the focus of their attention onto new objects and relationships" (Noss and Hoyles, 1996, p. 122). By having players express their ideas in the computational medium and then witness their expressed ideas enacted, the game context provides an opportunity for learners to interact with the programming concepts and develop an understanding of the computational behavior they embody. This promotes understandings of programming concepts that are built upon the webbing of the learning context, situated within the game play experience, and consistent with the abstract, transferable version that educators seek to teach.

In this study we used a video game context to introduce learners to programming concepts and study how the webbing it provided shaped the experience the learners had. There are a number of features of the game context that make it an effective medium for such a task including the expectation of challenges and early failures, a progression from easy to more difficult objectives, and the cultural syntonicity between programming and the computational nature of the video game medium. The rich, dynamic interactions of video games provide an array of potential scaffolds that collectively can serve as an effective webbing for situating programming abstractions that can be leveraged by a diverse range of learners based on their specific disposition, prior knowledge, and general approach to gameplay. It is important to note that using video games as a medium for introducing programming concepts to learners does have its drawbacks. For example, as most learners are familiar with video games, they come with expectations about video games that may not be desirable. As one participant put it: "[In RoboBuilder]*, you actually have to think, but like with other games you just sit there with the remote control and just play or whatever.*" The statement that most games do not require thought is not an ideal mindset for learners to have going into an educational experience. A second potential drawback for the game context is from potential repercussions if the game does not conform to players' expectations. One high school aged participant in our study decided to end his RoboBuilder session early, explaining that he was not a "*computer gamer*" and that he was more of a "*systems person*" (i.e. Play Station or Xbox). While this only happened once, it is important to be aware of the various ways learners might interpret and respond to the designed environment.

8. Conclusion

In this paper we showed how features of an introductory programming environment's language, design, and activity could inform and support how novices encounter and use programming concepts. Using a program-to-play constructionist video game, we analyzed when and how 12 programming novices encountered and used object state, conditional logic, iterative logic, and flow of control in order to accomplish in-game goals. Using Noss and Hoyles' (1996) constructs of webbing and situated abstraction, we identified how specific components of the environment supported the use of programming concepts and tied the meaning making and abstraction processes to specific instances of gameplay. In taking this approach, we argue for the importance of evaluating an environment's full set of design features, along with the programing activity learners engage in, and the prior knowledge they bring to the experience, when studying novices learning to program, as they collectively contribute to the webbing within which learning occurs. This stands in contrast to approaches that rely solely on post-test comparisons as the primary analytic tool for such research. In bringing this lens to introductory programming experiences, we showed in more detail how learners experience programming concepts in a way that post-test analyses cannot. As the importance of programming grows, the question of how best to introduce novices to these concepts becomes more important. Building on work from the constructionist tradition, we showed how a detailed investi-

gation of a single environment can advance our understanding of the way learners draw on the environment to make sense of these ideas in a way that complements the post test comparison approach. Combined, these methodologies paint a clearer picture and provide insights into designing new introductory programming tools to teach the next generation of programmers.

References

Blikstein, P., Wilensky, U. (2009). An atom is known by the company it keeps: a constructionist learning environment for materials science using agent-based modeling. *International Journal of Computers for Mathematical Learning*, 14(2), 81–119.

Bruckman, A., Jensen, C., DeBonte, A. (2002). Gender and programming achievement in a CSCL environment. In: *Proceedings of the Conference on Computer Support for Collaborative Learning: Foundations for a CSCL Community*. International Society of the Learning Sciences, Boulder, Colorado, 119–127.

Clements, D.H., Gullo, D.F. (1984). Effects of computer programming on young children's cognition. *Journal of Educational Psychology*, 76(6), 1051.

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

diSessa, A. A. (2000). *Changing Minds: Computers, Learning, and Literacy*. MIT Press, Cambridge.

Feurzeig, W., Papert, S., Bloom, M., Grant, R., Solomon, C. (1970). Programming-languages as a conceptual framework for teaching mathematics. *SIGCUE Outlook*, 4(2), 13–17.

Goldenberg, E.P., Feurzeig, W. (1987). *Exploring Language with Logo*. MIT Press, Cambridge.

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

Horn, M.S., Jacob, R.J.K. (2007). Tangible programming in the classroom with tern. In: *CHI '07 Extended Abstracts on Human Factors in Computing Systems*. ACM, New York, 1965–1970

Hoyles, C., Noss, R. (2004). Situated abstraction: mathematical understandings at the boundary. In: *Proceedings of Study Group 22 of ICME-10*. 7, 212–224.

Kay, A., Goldberg, A. (1977). Personal dynamic media. *Computer*, 10(3), 31–41.

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.

Kynigos, C., Koutlis, M., Hadzilacos, T. (1997). Mathematics with component-oriented exploratory software. *International Journal of Computers for Mathematical Learning*, 2(3), 229–250.

Lewis, C.M. (2010). How programming environment shapes perception, learning and goals: Logo vs. Scratch. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. New York, NY, 346–350.

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.

National Research Council. (2000). *How People Learn: Brain, Mind, Experience, and School*. The National Academies Press, Washington, D.C.

Nelson, M. (2001). Robocode. *IBM Advanced Technologies*.

Noss, R., Healy, L., Hoyles, C. (1997). The construction of mathematical meanings: connecting the visual with the symbolic. *Educational Studies in Mathematics*, 33(2), 203–233.

Noss, R., Hoyles, C. (1996). *Windows on Mathematical Meanings: Learning Cultures and Computers*. Kluwer, Dordrecht.

Papert, S. (1972). Teaching children to be mathematicians versus teaching about mathematics. *International Journal of Mathematical Education in Science and Technology*, 3(3), 249–262.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic books, New York.

Papert, S., Watt, D., diSessa, A., Weir, S. (1979). *Final Report of the Brookline Logo Project: Project Summary and Data Analysis (Logo Memo 53)*. MIT Logo Group, Cambridge, MA.

Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., … Silver, J. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11), 60.

Roque, R.V. (2007). *OpenBlocks: An Extendable Framework for Graphical Block Programming Systems* (Master's Thesis). Massachusetts Institute of Technology.

Weintrop, D., Holbert, N., Wilensky, U., Horn, M.S. (2012). Redefining constructionist video games: marrying constructionism and video game design. In: Kynigos, C., Clayson, J., Yiannoutsou, N. (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.

Weintrop, D., Wilensky, U. (2012). RoboBuilder: a program-to-play constructionist video game. In: Kynigos, C., Clayson, J., Yiannoutsou, N. (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.

Weintrop, D., Wilensky, U. (2013). Know your enemy: learning from in-game opponents. In: *Proceedings of the 12th International Conference on Interaction Design and Children* . ACM, New York, 408–411.

Weintrop, D., Wilensky, U. (2014). Program-to-play videogames: developing computational literacy through gameplay. In: *Proceedings of the 10th Games, Learning, & Society Conference*. Madison, WI.

Wilensky, U. (1999). *NetLogo*. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University. `http://ccl.northwestern.edu/netlogo`

Wilensky, U., Reisman, K. (2006). Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories – an embodied modeling approach. *Cognition and Instruction*, 24(2), 171–209.

Winslow, L.E. (1996). Programming pedagogy – a psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17–22.

D. Weintrop is a graduate student at Northwestern University pursuing a PhD in the Learning Sciences. He has a B.S. in Computer Science from the University of Michigan and spent five years working at a pair of software startups as a developer before returning to graduate school. His research focuses on the design and implementation of accessible and engaging programming environments that support learners in successfully encoding their own ideas in computationally meaningful ways. This includes questions of interface design, language features, and ways of leveraging the prior knowledge and experiences learners bring to an activity. He is also interested in the use of technological tools in supporting the exploration of non-computer science subjects, particularly within the STEM disciplines. His work lies at the intersection of computer science, cognitive science, and the learning sciences.

U. Wilensky is a mathematician, educator, learning technologist and computer scientist. While in Boston, he founded and directed the Center for Connected Learning and Computer-Based Modeling, now relocated to Northwestern University. He is involved in designing, deploying and researching learning technologies – especially for mathematics and science education. Much of his work of late has focused on the design of computer-based modeling and simulation languages, including networked collaborative simulations. He is very interested in the changing content of curriculum in the context of ubiquitous computation. A particular interest is in complexity and systems thinking. He has received numerous grants from NSF, NIH and the Department of Education. In 1996 he received a Career Award from the National Science Foundation and in 1999, a Spencer/NAE fellowship. He is a founder and an executive editor of the *International Journal of Computers for Mathematical Learning.*

Programavimo abstrakcijų išdėstymas kontrukcionistiniame vaizdo žaidime

David WEINTROP, Uri WILENSKY

Mokymosi aplinkų, skirtų programavimo pradmenims mokytis, efektyvumo tyrimai dažnai priklauso nuo galutinio testo vertinimo rezultatų ir požiūrių apklausų. Tačiau pastarieji būdai netinkami gaunamiems rezultatams paaiškinti. Straipsnyje pristatomas šioje srityje atliktas tyrimas. Naudojant Noss ir Hoyles' susietumo konstruktus ir abstrakcijų išdėstymą buvo analizuotas pradedančiųjų mokytis programavimo konstrukcionistinis vaizdo žaidimas, kuriuo buvo siekiama nustatyti, kaip įvadinių programavimo kalbų aspektai, aplinkų, kuriose žaidžiama, iššūkiai, su kuriais susiduria mokiniai, daro įtaką programavimo sąvokų supratimui. Analizė parodė, kad pradedantieji geba naudoti programavimo sąvokas, veikdami skaičiuojamajame kontekste su pateiktais ištekliais. Atlikti tyrimai papildo informatikos mokslo projektavimo literatūrą, atskleisdami ryšį tarp praturtintų, kompleksinių programavimo aplinkų ir mokymosi patirčių, kurias jos suteikia.