Program-to-play video games: Developing computational literacy through gameplay

David Weintrop, Northwestern University
Uri Wilensky, Northwestern University

Introduction

The ability to express ideas in a computationally meaningful way is becoming an increasingly important skill (National Research Council, 2010, 2011; Papert, 1980, 1993; Wilensky, 2001; Wing, 2006). diSessa (2000) argues that being able to express ideas in a computationally meaningful way can serve as the foundation of a powerful new literacy that will have widespread positive effects on society. Central to this new literacy is the ability to read, share, and express ideas in a form that a computational device can interpret and execute. Traditionally, these practices have been confined to the domain of computer science, but this view is being challenged by researchers and educators who argue that computational literacy skills are beneficial in a wide range of disciplines (Guzdial & Soloway, 2003; Wing, 2006). Part of the challenge of introducing learners to the skills foundational for computational literacy is designing learning environments that support the act of computational expression in a way that enables them to have early successes in a meaningful context. This paper presents the program-to-play approach, a design strategy for creating game-based learning environments designed to support novices in expressing ideas in a computationally meaningful way. Using RoboBuilder (Weintrop & Wilensky, 2012), we introduce the program-to-play paradigm and present data showing how this design approach scaffolds learners in developing computational literacy skills.

Prior Work

In response to the growing recognition that students can benefit from learning to express ideas in computationally meaningful ways, educational researchers have been developing "low-threshold" programming languages that are easier to learn but still permit significant expressivity. Beginning with Papert and colleagues' Constructionist Logo project (Feurzeig et al., 1970; Papert, 1980), there have been many efforts to bring learning environments for supporting computational expression to a wide audience of learners. diSessa (2000), in formulating his conception of computational literacy, used Boxer as one example of what form such tools might take. Boxer uses a graphical interface based on the naïve realism theory of mind to create a "glass-box" programming environment where "users should be able to pretend that what they see on the screen is their computational world in its entirety" (diSessa & Abelson, 1986, p. 861). A second notable line of work stems from Wilensky and colleagues who have responded to this design challenge by creating low-threshold, programming environments that focus on students building computational models of emergent phenomena (Wagh & Wilensky, 2012; Wilensky & Reisman, 2006; Wilensky, 1999, 2001; Wilkerson-Jerde & Wilensky, 2010). A third approach to low-threshold programming environments utilizes a graphical, grid-based model in which learners define states and transitions that enable the creation of games and simulations in two and three dimensional worlds (Ioannidou, Repenning, & Webb, 2009; Repenning, Ioannidou, & Zola, 2000). Yet another approach taken by Resnick and colleagues (Resnick et al., 2009) employs a blocks-based programming language, that leverages a blocks-as-puzzle-pieces metaphor, to enable young children to express themselves through creating games and stories.

A second active area of research studying the affordances of technology for creating learning environments is the growing literature on video games as a medium for learning (Barab et al., 2005; Gee, 2003; Holbert & Wilensky, 2014; Shaffer et al., 2005; Squire, 2003). This work looks at the potential use of games in both formal (Clark et al., 2011) and informal settings (Stevens, Satwicz, & McCarthy, 2007), and has created a great deal of excitement due to the increasingly ubiquity of video games in youth culture (Lenhart et al., 2008). While games have been designed to teach a diverse range of content areas, computer science educators have been particularly active in the use of video games as learning contexts as there is a natural match between the computational context of a video game and computer science content (Barnes et al., 2007; Bayliss & Strout, 2006; Li & Watson, 2011).

Our own work has focused on the goal of designing learning environments that make the skills associated with computational literacy more accessible and appealing to a broad range of learners. Towards this end we have developed the program-to-play design strategy that situates the practice of expressing ideas in a computationally meaningful way in a game-based learning environment where

learners compose small programs in order to play a game. While a similar approach has been used in the educational game space, such as the IPRO learning environment (Berland, Martin, & Benton, 2010) and PlayLOGO 3D (Paliokas, Arapidis, & Mpimpitsos, 2011), this paper seeks to formalize the design strategy, provide theoretical justification, and present evidence towards its effectiveness for teaching skills associated with computational literacy.

The Program-to-Play Approach

The central activity of program-to-play games is players defining instructions for their in-game characters to follow through a programming, or programming-like, interface. This is in contrast to a conventional video game interaction where players control their on-screen avatars directly as the game unfolds. The challenge that underpins all program-to-play games is for players to conceive of a strategy for their character and then figure out how to encode that idea using the tools provided by the game's programming interface. Players' learning how to express their own ideas and intentions in a way that the computer can interpret and execute is a key component of computational literacy. In looking at the use of video games as a context for computational expression, it is important to distinguish the program-to-play approach from tools designed for game authorship (Jones, 2000; Kafai, 1994). While learning environments that have students design and build games have been found to be a successful and motivating way to introduce learners to programming, the program-to-play model we present herein has additional, desirable features that build on the strengths of the game authorship approach. These strengths stem from the parallels that exist between the act of playing video games and the practice of programming.

For example, when playing a video game, players do not expect to be successful on their initial attempt, instead, game norms dictate that players will need multiple tries to accomplish an in-game challenge; trying different approach, refining strategies, and learning from prior mistakes along the way. In this way, games are low-stakes environments where failure is a part of success (Squire, 2005). Programming shares this feature as programs rarely work correctly on the first try. Instead, writing working programs requires many attempts. Trying different approaches to see what works and learning from prior mistakes without getting frustrated at a lack of immediate success are critical in the practice of programming. By aligning the construction of programs with the act of gameplay, players are situated in a context where early failures are expected and provide valuable learning experiences.

A second productive parallel between gameplay and programming that the program-to-play approach leverages is the iterative, incremental nature of both activities. When playing a game, players often attempt the same challenge a number of times, then, upon completing that task, proceed to the next, having gained experience and knowledge along the way. Programming shares this feature, as the completion of one component of a program leads to immediately working on the next, but with gained experience and new functionality to show for it. Additionally, the iterative, incremental characteristic of program-to-play games provides a natural, unobtrusive way for the game designer to scaffold players in moving from simple to more sophisticated programs that utilize more complex constructs. As players progress in a program-to-play game, the challenges become more difficult, thus, players need to respond by creating more sophisticated programs. This characteristic is unique to this design approach. In a game authoring learning environment there is no motivation provided by the tool itself to use advanced language features or create larger, more sophisticated challenges. Similarly, other open-ended, exploratory programming environments such as Scratch or Alice do not natively have a way to encourage more sophisticated constructions. In program-to-play environments on the other hand, as players progress, the game creator can design challenges that encourage and reward players for using more advanced programming constructs and creating more sophisticated programs. In RoboBuilder, this takes the form of opponents that demonstrate progressively more sophisticated concepts using a strategy we call "Learning from Your Opponent" (Weintrop & Wilensky, 2013a).

Meet RoboBuilder – A Program-to-Play Game

RoboBuilder (Figure 1) is a blocks-based, program-to-play game that challenges players to design and implement strategies to make their on-screen robot defeat a series of progressively more challenging opponents. A player's on-screen robot takes the form of a small tank, which competes in one-on-one battles against opponent robots. The objective of the game is for players to defeat their opponents by giving their robot instructions to locate and fire at their opponent while avoiding incoming fire; the first robot to make its opponent lose all its energy wins.

To facilitate this interaction, RoboBuilder has two distinct components: a programming environment (right pane of Figure 1), where players define and implement their robot's strategy; and an animated robot battleground (left pane in Figure 1), where players watch their robot compete. Players first interact with the programming interface to define their robot's behaviors before hitting the 'Go!' button,

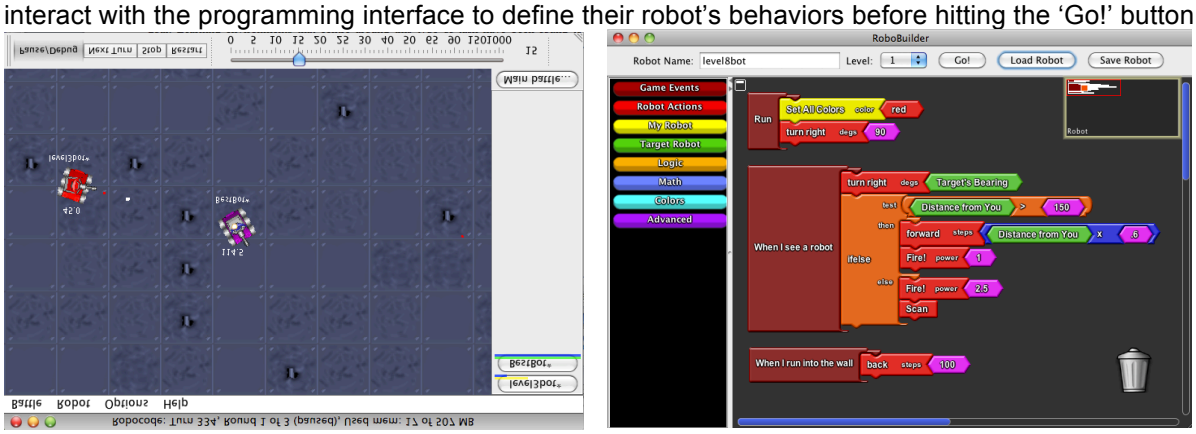

Figure 1: RoboBuilder's two screens: the battle screen (left) and the construction space (right).

which launches the battleground screen. To program their robot, players are provided with a custom deigned graphical programming language in which color-coded blocks encapsulate basic robot actions, such as `turn right` and `fire!`, that snap together to form robot strategies. Players are made aware that all of the opponents in the game were created using the same set of blocks that they are given; thus, it is always possible to recreate the strategy of an enemy robot. Once the battle starts, the player cannot interact with or alter their robot. RoboBuilder was designed for learners with little or no prior programming experience and has been played by a wide range of users including university graduate students and by students as young as ten as part of an afterschool program.

RoboBuilder is a combination of two open source projects: Robocode and OpenBlocks. Robocode (Nelson, 2001) is a problem-based learning environment initially designed to teach students how to program in Java. It has been used in introductory programming classes, where it has been found to be effective and motivating for students (O'Kelly & Gibson, 2006). RoboBuilder's programming interface is a modified version of the OpenBlocks framework (Roque, 2007), an open source Java library used to create graphical, blocks-based programming environments.

Methods

The data we present is from a study designed to explore how players interact with program-to-play games and to identify which features of the environment where utilized to succeed in the game.

Procedure

Our primary data collection activity was an hour-long, one-on-one interview during which a researcher sat alongside the participant as he or she played the game. At the outset of the interview participants were told that they would be playing a video game. They were then shown a pre-recorded robot battle to introduce them to the central challenge of the game. The interviewer then further described the objective of the game and explained the program-to-play method of gameplay. They were then introduced to the construction space including a high-level description of RoboBuilder's programming language, and shown how the blocks could be assembled to create a robot strategy. This first portion of the interview usually took around ten minutes, leaving roughly 50 minutes for gameplay.

Gameplay during the RoboBuilder interview followed a three-phase iterative protocol. In the first phase, participants were asked to verbally explain their intentions; either how they intended on defeating their opponent, or what changes they planned on making to their current strategy. Next, players were given the opportunity to implement their strategy using the provided programming language primitives. Once they were satisfied with the program they had created (or with the set of changes they had made), they launch the battle screen, which begins the third phase of the iterative protocol. During this final phase, players would watch the battle and explain what they observed, paying particular attention to whether or not their robot was behaving as expected. Players had the ability to end the battle at any point. After the battle screen was closed, the next iteration would begin with players again verbally explaining their goals for their robot strategy. Each RoboBuilder session

was recorded using both screen-capture and video-capture software. We also stored a digital copy of each robot strategy constructed during the RoboBuilder interview for further analysis.

Participants

Our main criterion for recruiting participants was that they be comfortable using computers but have little or no prior programming experience. Seven university-aged participants (3 female, 4 male) were recruited from a Midwestern university. Eight high school aged participants (1 female, 7 male) were recruited through relationships with members of the university community or through their affiliation with a community center in a Midwestern city that serves a predominantly African-American, low SES community. Participants played for an average of 48 minutes and 43 seconds (SD 8 minutes 39 seconds) and constructed an average of 11.5 unique robot strategies (SD 4.9). Each participant took part in one RoboBuilder session with the exception of one participant who agreed to four RoboBuilder sessions, each held a week apart. This resulted in a total of over 200 robot strategies being constructed and roughly 19 hours of RoboBuilder footage.

Playing a Program-to-Play Game

We begin this section by presenting a vignette of gameplay to provide a sense of the dynamics of playing a program-to-play game, before providing data showing how the full set of participants progressed over the course of their gameplay experiences. We start by looking at the first iterations of one interview to show what it looks like to conceive, then computationally express, a RoboBuilder strategy. After being introduced to RoboBuilder, this participant was asked how he was going to defeat his first opponent. He talked through a few ideas, then finally summed up his strategy this way:

> So my master plan is to, like, be continuously moving, so it's harder to hit. If I get hit, kind of change the path so it's different than what you might be expecting however the sequence is running, and then, during that path, adjust to what the opponent is doing to hit them.

He then brings up the composition screen (Figure 1, right side) and starts implementing his idea. Over the course of six minutes, he builds up his strategy, shown in Figure 2, beginning with the `Run` action then adding four more actions, defining and implementing the behavior for each as he goes.

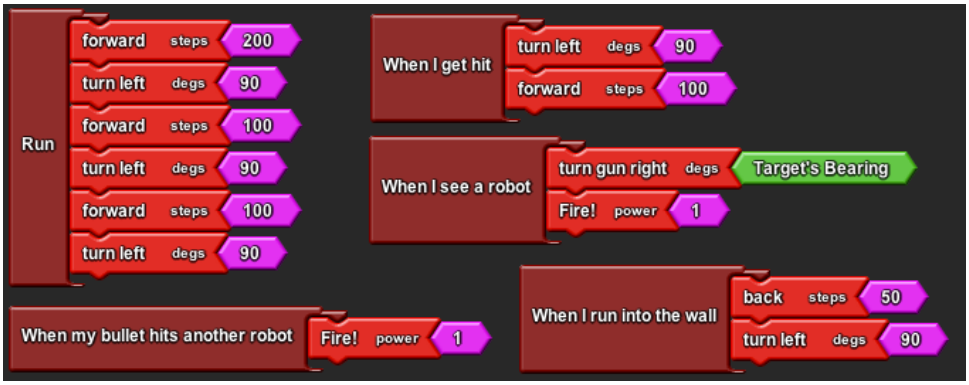

Figure 2. A participant's first robot strategy.

In this composition we can see aspects of his "*master plan*" reified, as well as additional components he added as he implemented his strategy. In the quote above, the participant articulated three distinct ideas, each of which are included in his program. The first verbalized strategy: "*be continuously moving, so it's harder to hit*" was implemented in the `Run` method of his program (left side of Figure 2). This series of instructions will result in his robot remaining in constant motion. His second tactic: "*if I get hit, kind of change the path so it's different*", can be found encoded in his `When I get Hit` event block. The two commands that will execute when his robot gets hit will cause the robot to change its heading and move forward out of the current line of fire. His final idea: "*adjust to what the opponent is doing to hit them*" is captured by his implementation of `When I See a robot` (bottom right of Figure 2), which will result in his robot's gun adjusting to the location of his opponent and firing at it whenever his robot spots its opponent. The participant also added two additional behaviors to his strategy: to backup and turn when he hits a wall, and to first after his robot successfully hits the opponent. These two strategy improvisations emerged based on the suggestive, idea-generating

capacity of the language that was frequently employed by players as they developed their strategies (Weintrop & Wilensky, 2013b). In subsequent iterations, this participant incrementally added new behaviors to his robot strategy. The practice this vignette highlights, that of computationally realizing an idea, is central to computational literacy and a key dynamic in program-to-play games.

Of the 15 programming novices who played RoboBuilder, 14 were able to successfully compose a strategy to defeat the first opponent with 9 participants advancing past level 3. While the size and complexity of players' constructions varied, programs generally got larger and more sophisticated as players progressed. Because of the iterative nature of program-to-play games, players' progressions are visible in the sequence of programs they construct. Beth, a vocal performance major with no prior programming experience, was the participant who agreed to four RoboBuilder sessions, resulting in a total of 46 distinct programs that we can use to map out her trajectory. As Beth progressed through the game, the size and complexity of her programs grew (as can be seen by the number and variety of commands). Figure 3 shows her winning robot strategies for levels 1, 2, and 4 (moving left to right).

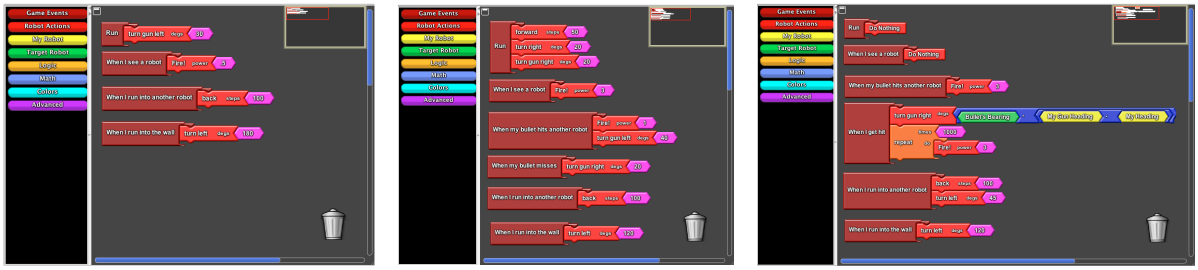

Figure 3. Three of Beth's Robots, progressing from earliest (left) to latest (right).

As she progressed, her robots grew larger (used more blocks), more complex (implemented more events), and more sophisticated (used a larger variety of blocks). Figure 4 depicts the trajectory of her programs over the course of her entire four hours of gameplay. Each line depicts the frequency of different types of blocks being included in her program, with the top line being the total.

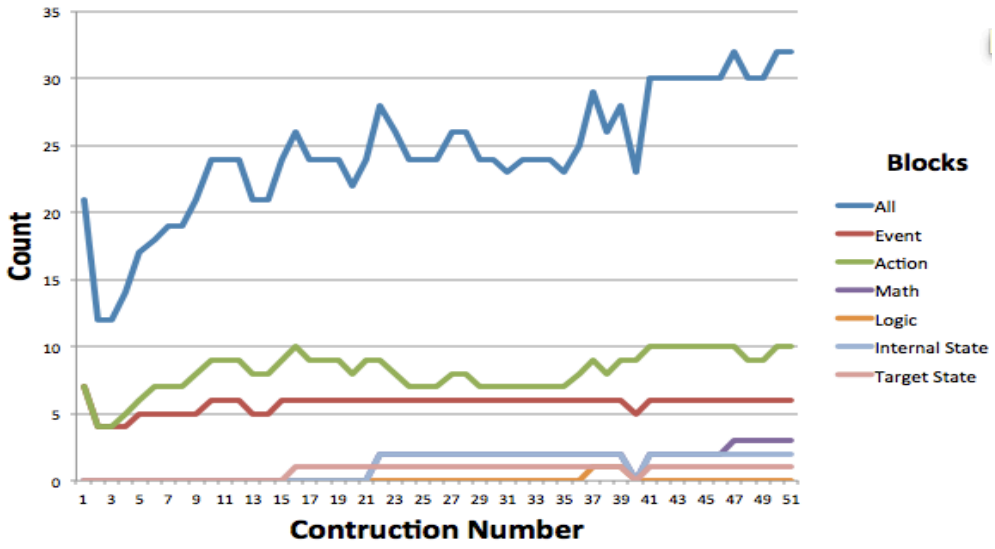

Figure 4. The blocks in Beth's robot constructions over the course of her gameplay.

Looking across the full set of participant, we can see how Beth's trajectory was typical as most participants progressed from small, simple programs to larger, more complex robot strategies. On average, each participant added 1.6 blocks to their strategy for each level they advanced. If we only consider the cases where participants revised their robot strategies between successful battles, the number of blocks added per level increases to 3.3 blocks for each new successful robot construction. Only two players' robot constructions got smaller as they progressed through the game; a third player's strategy remained at a fixed size; the remaining players' constructions grew as they progressed in the game. Figure 5 shows the trend lines for the size of each participant's robot construction over the course of their RoboBuilder sessions. By evaluating the novices' success in

expressing ideas within the medium provided, we can see how the program-to-play design approach resulted in players not only creating successful programs, but over the course of a single hour of gameplay, moving from small, simple constructions to larger more sophisticated programs.

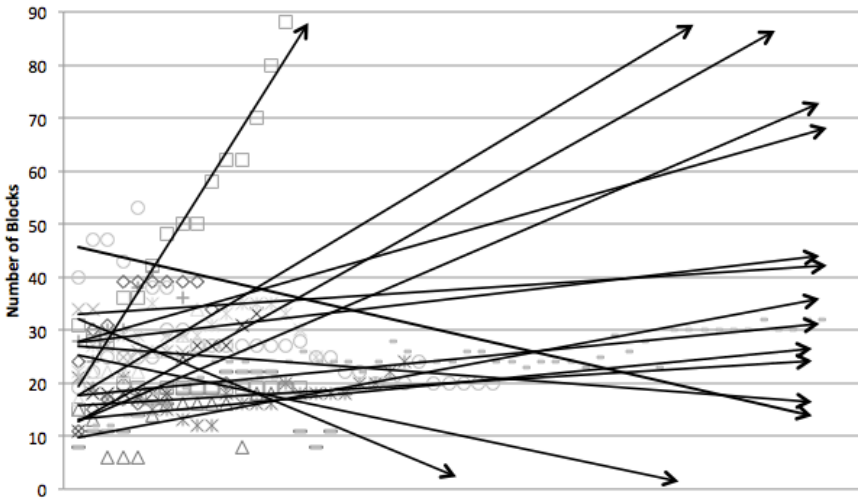

Figure 5. The trend lines of the changing size of players' constructions (projected forward).

Conclusion

The ability to express ideas in a computationally meaningful way is quickly becoming a core literacy one needs to succeed in our increasingly computational society. In this paper we introduced the program-to-play design strategy; an approach to creating game-based learning environments to teach fundamental computational literacy skills. This approach draws on video game norms that parallel programming practices and are productive when trying to express ideas in a computationally meaningful way. Using RoboBuilder, a program-to-play environment of our own design, we provided evidence for the potential of this approach. Novice programmers using RoboBuilder were able to create successful, sophisticated working programs in a short amount of time with minimal instruction. Our hope is that program-to-play games can serve as effective tools that fit within youth culture and can be meaningfully used to give young learners the experience of expressing their ideas in computational meaningful ways, helping us move towards a computationally literate society.

References

Barab, S., Thomas, M., Dodge, T., Carteaux, R., & Tuzun, H. (2005). Making learning fun: Quest Atlantis, a game without guns. *Ed. Technology Research and Development*, *53*(1), 86–107.

Barnes, T., Richter, H., Powell, E., Chaffin, A., & Godwin, A. (2007). Game2Learn: Building CS1 learning games for retention. In *ACM SIGCSE Bulletin* (Vol. 39, pp. 121–125).

Bayliss, J. D., & Strout, S. (2006). Games as a "flavor" of CS1. *SIGCSE Bull.*, *38*(1), 500–504.

Berland, M., Martin, T., & Benton, T. (2010). Programming standing up: Embodied computing with constructionist robotics. In *Proc. of Constructionism 2010 Conference*. Paris, France.

Clark, D. B., Nelson, B. C., Chang, H. Y., Martinez-Garza, M., Slack, K., & D'Angelo, C. M. (2011). Exploring Newtonian mechanics in a conceptually-integrated digital game: Comparison of learning and affective outcomes for students in Taiwan and the United States. *Computers & Education*, *57*(3), 2178–2195.

diSessa, A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: MIT Press.

diSessa, A., & Abelson, H. (1986). Boxer: a reconstructible computational medium. *Comm. of the ACM*, *29*(9), 859–868.

Gee, J. P. (2003). *What video games have to teach us about learning and literacy*. New York: Palgrave Macmillan.

Guzdial, M., & Soloway, E. (2003). Computer science is more important than calculus: The challenge of living up to our potential. *SIGCSE BULLETIN*, *35*(2), 5–8.

Harel, I., & Papert, S. (Eds.). (1991). *Constructionism*. Norwood N.J.: Ablex Publishing.

Holbert, N. R., & Wilensky, U. (2014). Constructible authentic representations: Designing video games that enable players to utilize knowledge developed in-game to reason about science. Technology, Knowledge and Learning, 1–27

Ioannidou, A., Repenning, A., & Webb, D. C. (2009). AgentCubes: Incremental 3D end-user development. *Journal of Visual Languages & Computing*, *20*(4), 236–251.
Jones, R. M. (2000). Design and implementation of computer games: a capstone course for undergraduate computer science education. *ACM SIGCSE Bulletin*, *32*(1), 260–264.
Kafai, Y.(1994). *Minds in play: Computer game design as a context for children's learning*. Routledge.
Lenhart, A., Kahne, J., Middaugh, E., Macgill, A. R., Evans, C., & Vitak, J. (2008). Teens, video games and civics. *PEW Internet & American Life Project*.
Li, F. W., & Watson, C. (2011). Game-based concept visualization for learning programming. In *Proc. of the 3rd Int. ACM workshop on Multimedia technologies for distance learning* (pp. 37–42).
National Research Council. (2010). *Report of a workshop on the scope and nature of computational thinking*. Washington, D.C.: The National Academies Press.
National Research Council. (2011). *Report of a workshop of pedagogical aspects of computational thinking*. Washington, D.C.: The National Academies Press.
Nelson, M. (2001). Robocode. *IBM Advanced Technologies*.
O'Kelly, J., & Gibson, J. P. (2006). RoboCode & problem-based learning: A non-prescriptive approach to teaching programming. In *Proc. of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 217–221).
Paliokas, I., Arapidis, C., & Mpimpitsos, M. (2011). PlayLOGO 3D: A 3D interactive video game for early programming education: Let LOGO be a game. In *Games and Virtual Worlds for Serious Applications* (pp. 24–31). IEEE.
Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic books.
Papert, S. (1993). *The children's machine: Rethinking school in the age of the computer*. New York: Basic Books.
Repenning, A., & Ioannidou, A. (2004). Agent-based end-user development. *Comm. of the ACM*, *47*(9), 43–46.
Repenning, A., Ioannidou, A., & Zola, J. (2000). AgentSheets: End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, *3*(3).
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.
Shaffer, D., Squire, K., Halverson, R., & Gee, J. (2005). Video games and the future of learning. *Phi Delta Kappan*, *87*(2), 104–111.
Squire, K. (2003). Video games in education. *Int. J. Intell. Games & Simulation*, *2*(1), 49–62.
Squire, K. (2005). Changing the game: What happens when video games enter the classroom. *Innovate: Journal of Online Education*, *1*(6).
Stevens, R., Satwicz, T., & McCarthy, L. (2007). In-game, in-room, in-world: Reconnecting video game play to the rest of kids' lives. *The John D. and Catherine T. MacArthur Foundation Series on Digital Media and Learning*, 41–66.
Wagh, A., & Wilensky, U. (2012). Evolution in blocks: Building models of evolution using blocks. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proc. of Constructionism 2012*. Athens, Gr.
Weintrop, D., & Wilensky, U. (2012). RoboBuilder: A program-to-play constructionist video game. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proc. of Constructionism 2012*. Athens, Gr.
Weintrop, D., & Wilensky, U. (2013a). Know your enemy: Learning from in-game opponents. In *Proc. of the 12th International Conference on Interaction Design and Children* (pp. 408–411). New York, NY, USA: ACM.
Weintrop, D., & Wilensky, U. (2013b). Supporting computational expression: How novices use programming primitives in achieving a computational goal. Paper presented at the American Education Researchers Association, San Francisco, CA, USA.
Wilensky, U. (1999). *NetLogo*. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University. http://ccl.northwestern.edu/netlogo.
Wilensky, U. (2001). Modeling nature's emergent patterns with multi-agent languages. In *Proc. of EuroLogo* (pp. 1–6). Linz, Austria.
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.
Wilkerson-Jerde, M. H., & Wilensky, U. (2010). Restructuring Change, Interpreting Changes: The DeltaTick Modeling and Analysis Toolkit. In J. Clayson & I. Kalas (Eds.), *Proc. of Constructionism 2010*. Paris, Fr.
Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.