Playing by Programming: Making Gameplay a Programming Activity

David Weintrop
Uri Wilensky

Video games are an oft-cited reason for young learners getting interested in programming and computer science. As such, many learning opportunities build on this interest by having kids program their own video games. This approach, while sometimes successful, has its drawbacks stemming from the fact that the challenge of programming and game building are left distinct from the gameplay experience that initially drew learners in. An alternate strategy to engage learners in programming that builds on their interest and enjoyment of video games is to integrate programming into the gameplay experience directly through the design of *program-to-play* games. These games make programming a central part of the activity of playing the game, fully immersing programming within the game context. In this article, the authors develop the genre of program-to-play games, discuss key features of these environments and their pedagogical potential, and highlight some exemplar program-to-play games.

David Weintrop is a doctoral student in the Learning Sciences at Northwestern University. 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 (e-mail: dweintrop@u.northwestern.edu). **Uri Wilensky** is a Professor of Learning Sciences, Computer Science, and Complex Systems at Northwestern University. He is the founder and director of the Center for Connected Learning and Computer-Based Modeling and a founding faculty member of the Northwestern Institute on Complex Systems. His core research interests are in the design of learning technologies and agent-based modeling. He has published more than 200 refereed articles and he is the author of the widely-used *NetLogo* agent-based modeling environment. He has employed *NetLogo* to develop computationally-based curriculum for all levels of education. He received his Ph.D. from MIT in 1993 (e-mail: uri@northwestern.edu).

Rethinking Games and Computer Science Education

Video games have become nearly ubiquitous in the lives of kids growing up today. Be it addictive mobile games, sophisticated blockbuster console games, or casual online games played through social networking sites—video games are everywhere. The growth in popularity of video games has not gone unnoticed by educators and designers of educational technology. As a result, a growing number of educational designers are using video games as a context for learners to engage with new ideas and concepts. In this way, the video game medium is being used to provide meaningful learning opportunities, often with the hopes that positive play experiences will increase interest in the domain for the player. This is especially true for the field of computer science, as the video games being played are themselves products of the concepts and practices that are central to computer science. Thus, interest in and enjoyment of video games is a frequently cited reason for why learners want to study computer science (Carter, 2006). Blending video games and computer science education often has learners create their own games. Numerous game-authoring tools and game-building workshops exist that make it easy for programming novices to create their own unique games in hopes of providing early, positive computer science experiences. This approach, while sometimes successful, has the potential drawback that the challenge of programming and game building are disconnected from the gameplay experience that initially drew in learners.

We propose *program-to-play* games as an alternate strategy to engage learners in computer science. Program-to-play games build on learners' interest and enjoyment of video games while integrating programming directly into the gameplay experience. These games make programming an integral and essential part of the activity of playing the game, fully immersing programming within the game context. In this article we introduce the genre of program-to-play games, discuss key features of these environments, argue for the educational and pedagogical potential of games that employ this gameplay mechanism, and finally, highlight some existing program-to-play games.

Program-to-Play Games

Program-to-play games are defined by the use of programming as a central mechanism of gameplay. Rather than simply "gamifying" introductory programming activities to motivate and engage learners, program-to-play games blend programming and gameplay to provide an authentic and enriching programming experience. Though this blending can be accomplished in a number of ways, all games in this genre challenge players to conceive of

strategies and then figure out how to encode that idea using the game's programming tools. Program-to-play games shift gameplay away from in-the-moment reaction and reflex towards a more deliberate, premeditated, and thoughtful gameplay interaction, with programming spurring and mediating in-game design thinking.

During gameplay, players engage in authentic programming practices, including programmatically expressing ideas, constructing and manipulating computational abstractions, iteratively developing solutions, and debugging constructed artifacts (Weintrop, Holbert, Wilensky, & Horn, in press). This game design approach aligns the concepts to be learned with the way the learner interacts with and plays the game, a strategy found to be effective for learning through gameplay (Clark, Nelson, Chang, Martinez-Garza, Slack, & D'Angelo, 2011; Habgood & Ainsworth, 2011; Holbert & Wilensky, 2014). Program-to-play games are one instantiation of the genre of constructionist video games, which more broadly place construction at the heart of gameplay (Weintrop, Holbert, Wilensky, & Horn, 2012) and brings a "code-first" design (Horn Brady, Hjorth, Wagh, & Wilensky, 2014) to the video game medium.

The idea of using video games to situate and motivate programming is not new. There is a long history of projects and tools that have successfully brought video games and programming together to create meaningful experiences for learners. This is most commonly done through an *authorship model,* where learners engage in programming by writing their own video games. This approach was pioneered by Papert and students as part of the Instructional Software Design Project (ISDP), which had students author games about fractions in the *Logo* language as part of a larger constructionist learning experience (Harel & Papert, 1990; Kafai, 1994). The ISDP project found that students learned more about both programming and math than students who learned the content separately in the traditional curriculum.

Since early successes with *Logo,* a number of other introductory programming tools have been used in game-making educational contexts, including *Scratch* (Resnick *et al.*, 2009), *Alice* (Werner, Campe, & Denner, 2012), *AgentSheets* (Basawapatna, Koh, & Repenning, 2010), *NetLogo* (Holbert & Wilensky, 2011; Wilensky, 1999), and *StarLogo TNG* (Begel & Klopfer, 2007). Additionally, new tools have been specifically designed to make it easy for novice programmers to author fun and engaging games, including *Stagecast Creator* (Smith, Cypher, & Tesler, 2000), *Game Maker* (Overmars, 2004), and *ToonTalk* (Kahn, 1996).

Another variant of the game authorship approach provides players with tools that enable the creation of new content in existing games, be it new items, levels, or opponents (Robertson & Good, 2005).

While these tools have been successful, there are a number of features of the program-to-play approach that build on the strengths of the game authorship model and also address some of its shortcomings.

Benefits of Program-to-Play Games

The strengths of the program-to-play approach that differentiate it from the authorship model, and make it an effective context for learning to program, stem from the parallels between the act of playing video games and the practice of programming. Additionally, there are features of the video game context that are particularly well suited to foster productive programming practices and encourage players to interact with and use concepts central to programming.

The Parallels Between Gaming and Programming

There are many similarities between programming and playing video games. 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 approaches, refining strategies, and learning from prior mistakes as they progress. 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 successive debugging. A willingness to trying different approaches to see what works, and the ability to learn from prior mistakes without getting frustrated at a lack of immediate success, are productive dispositions for both programming and playing video games. 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 we leverage in the program-to-play approach is the iterative, incremental nature of both activities. When playing a game, challenges are laid out sequentially such that later levels depend on skills and abilities mastered in earlier levels. At the outset of a game, players develop foundational capabilities that are then incrementally built upon as the player's skill improves and the in-game challenges get more difficult. Likewise, when composing programs, at the outset, basic functionality is implemented that enables the program to carry out simple and more central aspects of the ultimate goal. As the development of the program proceeds, these basic functions are built upon, revised, and improved to support more sophisticated behaviors that are added later in the development process. In this way, programs are iteratively and incrementally built. By embedding the programming challenge within the game itself, the incremental nature of both activities helps learners develop an iterative perspective on the practice of programming.

A third similarity between programming and playing video games is not a feature of games specifically, but of

the larger set of norms that accompany gameplay. For any given game, players are only a few Web searches away from being able to find walk-through videos, game guides, and other online support materials. Likewise, for any given programming challenge, a carefully crafted search query will quickly lead you towards information that can help you accomplish the task at hand. While this feature can be abused (in both video games and programming), knowing how and when to take advantage of useful resources available beyond the immediate problem context is an important skill. More often than not with both programming and video games, potential solutions found externally cannot (or should not) be directly applied, but instead need to be adapted and configured to address the problem at hand. This process of adaption entails figuring out how a discovered solution works, then appropriating just the necessary pieces of it. Through studying solutions developed by others and deciding if and how the solution can be integrated into their own game, players engage in the authentic programming practice of using online resources, while also further developing the skills of code comprehension and reasoning about the programming constructed used as part of the found solution.

Contextualizing and Scaffolding Programming with Games

Along with these productive parallels between video games and programming, there are other productive reasons to situate a programming activity within a video game. Research from the computer science education community has found that context is an important component of learning to program, especially early on (Cooper & Cunningham, 2010). Video games, and the engaging and motivating experiences they provide, can serve as an effective context within which meaning-making around programming concepts can occur (Weintrop & Wilensky, 2014b). By situating computational abstractions within video game contexts, game resources can help learners make sense of abstract ideas and understand how to incorporate them into programs. Further, games provide a visual environment in which to watch programs run, providing immediate and meaningful feedback about whether a program behaved as intended.

The video game context also provides a natural, unobtrusive way to scaffold players in moving from simple to more sophisticated programs that utilize more complex constructs (Weintrop & Wilensky, 2014a). Given the iterative and incremental nature of video games, challenges become more difficult as a player advances in program-to-play games. Players thus need to respond by creating more sophisticated programs. This provides a mechanism for the game designer to encourage and reward players for using more advanced programming constructs and creating more sophisticated programs. In this way the game can scaffold players in moving from simple, straightforward programs to more complex programs that include more sophisticated programming ideas.

Potential Limitations

While initial studies of program-to-play games as introductory programming environments have been promising, questions remain about potential drawbacks and limitations of the approach. One central concern is that the use of video games could potentially contribute to the existing gender disparity within the field of computer science. While video game playing is becoming increasingly widespread among girls, it is still perceived as a largely male activity; thus, it is important to consider and incorporate themes and mechanics that appeal to a broad, diverse audience.

A second concern is that games that provide clear outcomes (in the form of success/failure or wins/losses) could promote the sense that unsuccessful programs are 'wrong,' as opposed to 'fixable.' Likewise, players might look for the one 'correct' program, and not learn that in programming, there are a wide variety of ways to accomplish a specific task.

Finally, where video games are often seen in a competitive light, programming is a much more communal, collaborative activity. By introducing programming in a potentially combative context, it is possible that learners will develop an individualist, competitive view of the practice and miss the cooperative, community-oriented aspects of programming.

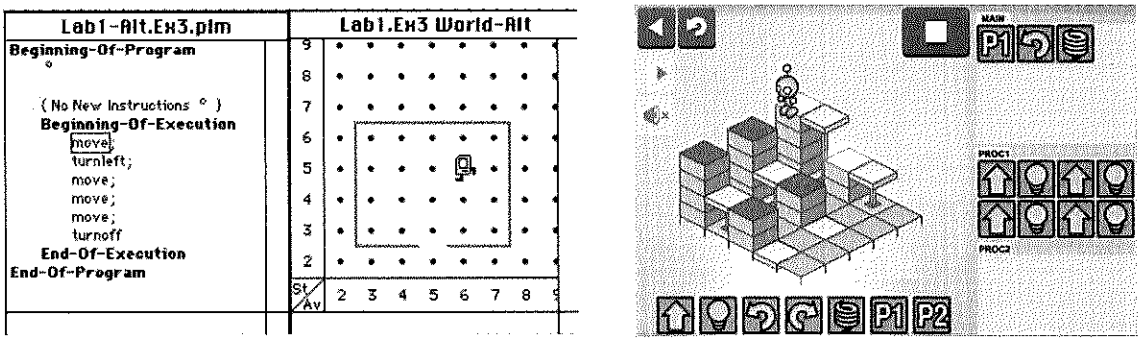
Research is currently underway seeking to understand the extent of these issues on learners and to investigate potential design solutions to mitigate their effects.

Examples of Program-to-Play Games

Having introduced the concept of a program-to-play game, and discussed some of the benefits of the approach, we now present some exemplar program-to-play games to concretize our conceptual introduction of program to play games and emphasize the diversity of the genre. This is by no means an exhaustive list, but instead intended to describe various approaches to integrating programming into gameplay.

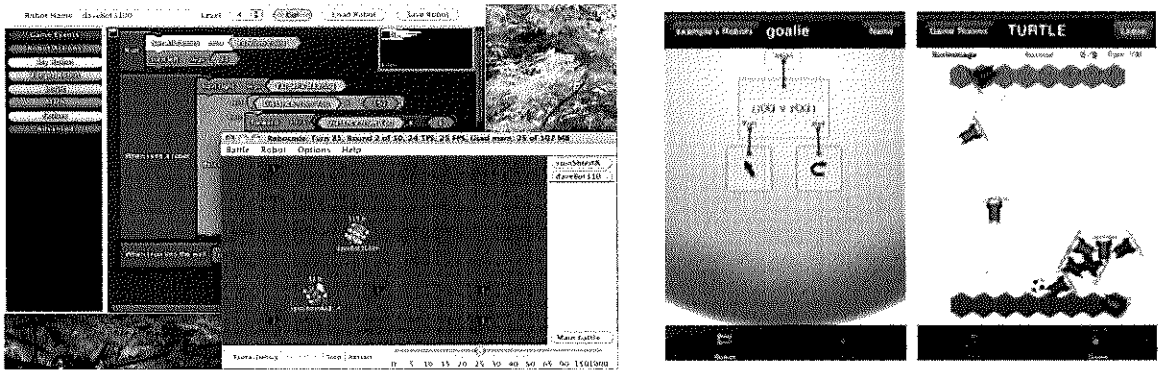
Programming the Character: The Legacy of *Karel*

The first game we want to highlight is a classic that was initially designed as a way to introduce basic programming concepts, and developed a formula that has often been replicated since its release. *Karel the Robot* (Pattis, 1981) is a game in which players write simple programs to control an on-screen robot as it moves around a two-dimensional world (***Figure 1a***). At each level, players are asked to write a program using a simplified programming language to get *Karel* to pick-up a set of "beepers" that have been scattered throughout the world. As players progress, the levels require players to integrate condition-



(a) (b)

Figure 1. *Karel the Robot* (a) and *LightBot* (b) — two player-versus-environment program-to-play games.



(a) (b)

Figure 2. Two examples of player-versus-player program-to-play games: *RoboBuilder* (a) and *IPRO* (b).

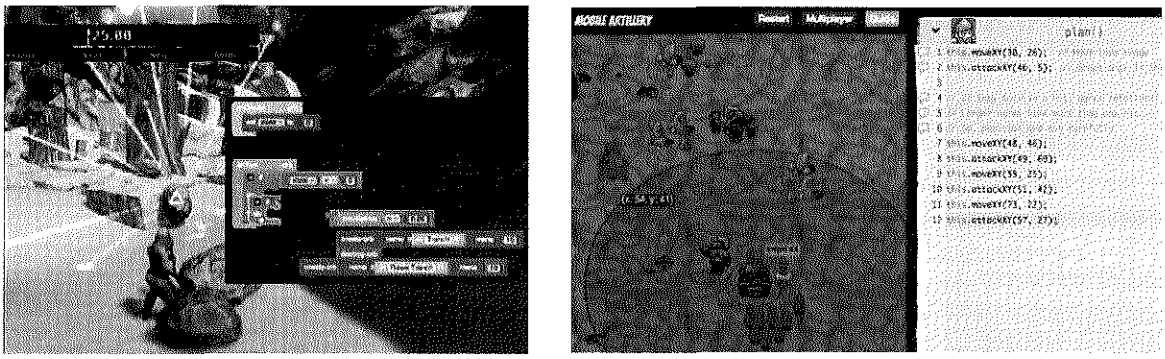
al and looping logic into their programs in order to collect all the beepers. *Karel the Robot* initially used its own *Logo*-like language but has since been modified to use a number of other languages including C++ (*Karel++*), *Python (Guido von Robot* and *RUR-PLE*), *Java (Karel J. Robot)*, and *JavaScript (CodeHS)*. More recent variants of this game design, such as *LightBot* (***Figure 1b***), *Robozzle,* and *Cato's Hike,* replace the text-based programming language with a graphical language to prevent syntactic errors but still have players engage with the same concepts as they play.

The approach of having players write programs to control characters directly is a common form of program-to-play game; but where *Karel the Robot* focused on motion and a simple pick-up/drop mechanism, other games provide a much larger set of capabilities to the player. For example, *Gidget* (Lee *et al.*, 2014) provides players with a *Python*-like language and has early levels similar to *Karel* before moving on to levels that have player use variables, lists, conditional, objects, and looping structures. Other games are designed to teach players specific programming concepts. For example, *Wu's Castle* (Eagle & Barnes, 2008) is a role-playing game in which players instruct characters to create armies of snowmen by issuing *Java* commands that depend on looping structures and array manipulation to succeed at in-game challenges.

While *Karel* and its descendants had players author programs to control their robots in a player-versus-environment challenge, another set of program-to-play games uses a similar mechanism, but in a player-versus-player model. *Robocode* (Nelson, 2001) provides players with a *Java* API that allows them to control on-screen robots as they compete in battles against other robots. *RoboBuilder* (***Figure 2a***) uses the *Robocode* library and adds a blocks-based programming language, making it easier for learners to play without prior programming experience (Weintrop & Wilensky, 2012). A second game that uses a similar gameplay mechanism in a very different context is *IPRO* (Martin, Berland, Benton, & Smith, 2013), which has players give directions to soccer-playing robots using a visual programming language running on a mobile device (***Figure 2b***).

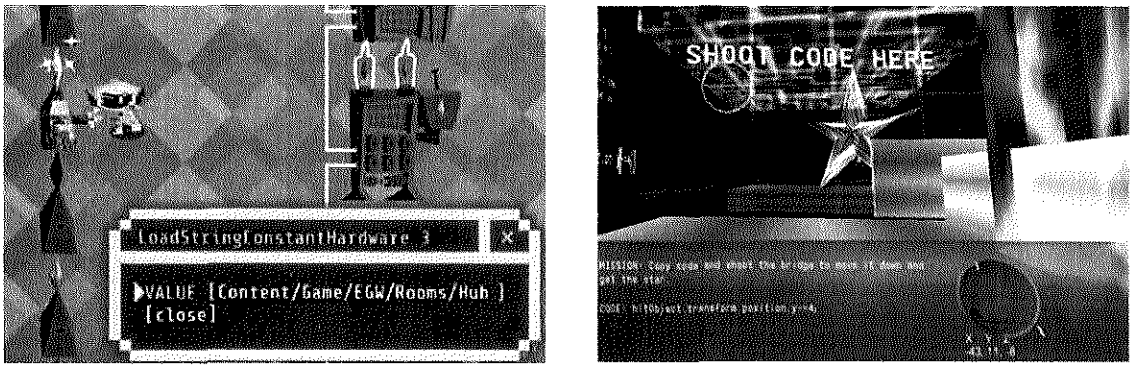
Augmenting Characters Through Programming

The above games have players write programs to control their in-game characters. Another way to integrate programming into gameplay is by having players write programs to add capabilities to their in-game characters. For example, in *CodeSpells* (Esper, Foster, & Griswold, 2013), players create spells for their characters by writing programs that define what the spells will do (***Figure 3a***). In this



(a) (b)

Figure 3. Two program-to-play games that use programming as a way to add new capabilities to in-game characters: (a) *CodeSpells* and (b) *CodeCombat*.



(a) (b)

Figure 4. Two program-to-play games that allow players to write code to manipulate the game environments: (a) *Hack n Slash* and (b) *Code Hero.*

way, programming remains an integral part of the game, but it is a distinct activity independent of controlling the character in real time, leaving in place more conventional gameplay mechanics. A second, similar approach is used in *CodeCombat* (***Figure 3b***), where players define sets of instructions for their in-game characters to carry out as they navigate the world, defeating enemies and collecting items. The items that players collect grant new capabilities to the characters in the form of new instructions that can be included in programs, thus linking in-game accomplishments with new and more sophisticated programs.

(Re)Programming Worlds and Environments

A third way to integrate programming into gameplay is by allowing players to programmatically interact with the environment in which the game is played. The first example of this type of program-to-play game is called *Hack n Slash* (***Figure 4a***). In this game, players have a magic sword that allows them to view and edit the properties of in-game objects. This game mechanic allows players to rewrite the code that is controlling the game and defining the objects in it as they play. For example, if there is an immovable obstacle in the player's way, the player, using their magical sword, can rewrite the properties of that obstacle, making it moveable.

A second example of this type of game is *Code Hero* (***Figure 4b***), a first-person shooter that allows the player to "shoot" small programs at elements in the world. These code snippets execute when they reach their target. So, for example, if a platform is out of reach, but the player wants to jump to it, the player could author a short program that would modify the *y*-position of the object that it hits, and then shoot that code at the platform, lowering it so the player can jump on it.

These two games, along with others presented in this article, illustrate some of the ways that programming has been integrated into gameplay and demonstrate the program-to-play approach to video game design.

Conclusion

Video games often serve as a gateway to introduce learners to the field of computer science, with game authorship activities serving as the primary way to bring these two interrelated domains together. In this article, we have presented an alternative approach to blending video games and programming through the development of

program-to-play games. By integrating the writing of programs into the gameplay mechanics of the game, players can be introduced to fundamental computer science concepts and develop productive programming practices in fun, motivating, and meaningful ways.

Program-to-play games draw on parallels between the activity of playing video games and productive practices in programming, as well as features unique to the video game medium, to create a new way to introduce and engage younger learners with ideas from the field of computer science. Program-to-play video games contribute a new approach to the growing number of ways that educational designers can bring programming and computer science into the lives of young learners. In doing so, we hope that these games will help educate and inspire the next generation of great programmers. □

References

Basawapatna, A. R., Koh, K. H., & Repenning, A. (2010). Using scalable game design to teach computer science from middle school to graduate school. In *Proc. of the 15th Annual Conference on Innovation and Technology in Computer Science Education* (pp. 224–228). New York: ACM.

Begel, A., & Klopfer, E. (2007). *Starlogo TNG:* An introduction to game development. *Journal of E-Learning.*

Carter, L. (2006). Why students with an apparent aptitude for computer science don't choose to major in computer science. In *Proc. of the 37th Technical Symposium on Computer Science Education* (pp. 27–31). New York: ACM.

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.

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

Eagle, M., & Barnes, T. (2008). *Wu's Castle:* Teaching arrays and loops in a game. *ACM SIGCSE Bulletin, 40*(3), 245–249.

Esper, S., Foster, S. R., & Griswold, W. G. (2013). *CodeSpells:* Embodying the metaphor of wizardry for programming. In *Proc. of the 18th Annual Conference on Innovation and Technology in Computer Science Education* (pp. 249–254). New York: ACM.

Habgood, M. P. J., & Ainsworth, S. E. (2011). Motivating children to learn effectively: Exploring the value of intrinsic integration in educational games. *Journal of the Learning Sciences, 20*(2), 169–206.

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

Holbert, N., & Wilensky, U. (2011). FormulaT Racing: Designing a game for kinematic exploration and computational thinking. In *Proc. of the 7th Games, Learning, & Society Conference,* Madison, WI.

Holbert, N., & 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, 19*(1–2), 53–79.

Horn, M. S., Brady, C., Hjorth, A., Wagh, A., & Wilensky, U. (2014). *Frog Pond:* A codefirst learning environment on evolution and natural selection. In *Proc. of the 2014 Conference on Interaction Design and Children* (pp. 357–360). New York: ACM.

Kafai, Y. B. (1994). *Minds in play: Computer game design as a context for children's learning.* New York: Routledge.

Kahn, K. (1996). ToonTalk: An animated programming environment for children. *Journal of Visual Languages & Computing, 7*(2), 197–217.

Lee, M. J., Bahmani, F., Kwan, I., LaFerte, J., Charters, P., Horvath, A., ... Ko, A. (2014). Principles of a debugging-first puzzle game for computing education. In *Proc. of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 57–64). New York: IEEE.

Martin, T., Berland, M., Benton, T., & Smith, C. M. (2013). Learning programming with *IPRO:* The effects of a mobile, social programming environment. *Journal of Interactive Learning Research, 24*(3), 301–328.

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

Overmars, M. (2004). Teaching computer science through game design. *Computer, 37*(4), 81–83.

Pattis, R. E. (1981). *Karel the Robot: A gentle introduction to the art of programming.* New York: John Wiley & Sons.

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

Robertson, J., & Good, J. (2005). Story creation in virtual game worlds. *Comm. of the ACM, 48*(1), 61–65.

Smith, D. C., Cypher, A., & Tesler, L. (2000). Programming by example: Novice programming comes of age. *Comm. of the ACM, 43*(3), 75–81.

Squire, K. (2005). Changing the game: What happens when video games enter the classroom? *Innovate: Journal of Online Education, 1*(6).

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

Weintrop, D., Holbert, N., Wilensky, U., & Horn, M. S. (in press). Computational thinking in constructionist video games. *International Journal of Game-Based Learning.*

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

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

Weintrop, D., & Wilensky, U. (2014b). Situating programming abstractions in a constructionist video game. *Informatics in Education, 13*(2), 307–321.

Werner, L., Campe, S., & Denner, J. (2012). Children learning computer science concepts via *Alice* game-programming. In *Proc. of the 43rd technical symposium on Computer Science Education* (pp. 427–432). New York: ACM.

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