Teacher Transformations in Developing Computational Thinking: Gaming and Robotics Use in After-School Settings

Alan Buss and Ruben Gamboa

Abstract The challenges of addressing increasing calls for the inclusion of computational thinking skills in K-12 education in the midst of crowded school curricula can be mitigated, in part, by promoting STEM learning in after-school settings. The *Visualization Basics: Using Gaming to Improve Computational Thinking* project provided opportunities for middle school students to participate in after-school clubs focused on game development and LEGO robotics in an effort to increase computational thinking skills. Club leaders and teachers, however, first needed to develop proficiency with the computational tools and their understanding of computational thinking. To achieve these goals, teachers participated in two online professional development courses. After participating in the courses, teachers' understanding of and attitudes toward computational thinking skills were mostly positive. Observations of club sessions revealed that teachers provided a mix of structured and open-ended instruction. Guided instruction, such as using detailed tutorials for initial exposure to a concept or process, was most commonly observed. One area identified for improvement was the duration of the courses, which provided limited time for teachers to develop deep and robust computational thinking skills. Despite this limitation, the data collected thus far suggest that teachers' understanding of and attitudes toward computational thinking skills improved.

Keywords Game development • Robotics • Teacher professional development • Computational thinking

A. Buss (✉) • R. Gamboa
University of Wyoming, 1000 E. University Ave, Laramie, WY 82071, USA
e-mail: abuss@uwyo.edu; ruben@uwyo.edu

© Springer International Publishing AG 2017

P.J. Rich, C.B. Hodges (eds.), *Emerging Research, Practice, and Policy on Computational Thinking*, Educational Communications and Technology: Issues and Innovations, DOI 10.1007/978-3-319-52691-1_12

Introduction

The history of US public schools is replete with calls for increased skills for dealing with current and future challenges. These calls include improvements in problem-solving and critical thinking skills (Educational Policies Commission, 1961), twenty-first-century skills (Uchida, Cetron, & McKenzie, 1996), and, more recently, computational thinking skills (Barr, Harrison, & Conery, 2011; Wing, 2006). Responding to these calls is a significant challenge on multiple fronts, including curriculum constraints and professional development. Demands on teachers' time to address existing curriculum requirements are high, leaving little or no room for new content, such as with computational thinking (Grover & Pea, 2013; National Research Council, 2011). Out-of-school activities, including after-school programs, however, provide greater opportunities to address computational thinking (CT) due to greater flexibility with curriculum and widely available web-based resources (National Research Council, 2011).

In 2013 the University of Wyoming received NSF Innovative Technology Experiences for Students and Teachers (ITEST) funding to implement a three-year program focused on developing middle school students' computational and spatial visual thinking skills in after-school settings. The resulting program, *Visualization Basics: Using Gaming to Improve Computational Thinking (UGICT)*, helped public school teachers and community members form after-school game development and robotics clubs. As most club teachers did not have experience with programming or robotics, professional development (PD) was provided in the form of two synchronous web-based courses. Data were gathered on teachers' understanding of CT and instructional practices through the use of pre-post surveys and club observations. This chapter focuses on results from years 1 and 2 of the grant project, in which 28 teachers in grades 4–8 from 18 schools in Wyoming participated.

Theoretical Framework

While there is a definite "cool" factor for selecting game development and robotics as tools for improving computational thinking skills, the use of such technology tools for learning is rooted in the ideas of constructionism. The key to learning is activity and experience (Dewey, 1916, 1958), whether through social interaction (Lave & Wenger, 1991; Salomon & Perkins, 1998; Vygotskiĭ & Cole, 1978), play (Honeyford & Boyd, 2015; Piaget & Inhelder, 1969), experimentation, or creation and construction (Burke, O'Byrne, & Kafai, 2016; Kafai, 1995, 2006; Kafai & Burke, 2013; Papert & Harel, 1991). Using programming to create new artifacts such as games and robotic controls is an effective tool for learning computer science concepts, mathematics, and problem-solving (Akcaoglu, 2016; Ardito, Mosley, & Scollins, 2014; Denner, Werner, & Ortiz, 2012; Kafai, 1996; Li, 2010; Papert, 1980). Furthermore, the use of game design and robotics promotes a specific type of

thinking skills known as computational thinking (Atmatzidou & Demetriadis, 2016; Bers, Flannery, Kazakoff, & Sullivan, 2014; Nickerson, Brand, & Repenning, 2015; Repenning et al., 2015).

Computational Thinking in K-12 Education

Capturing the essence of CT, particularly in the context of K-12 education, in a simple definition is a vexing problem (Atmatzidou & Demetriadis, 2016; Barr & Stephenson, 2011; Grover & Pea, 2013; NRC, 2011). The definitions of CT that have been offered differ in some details, but they are largely consistent with one another. One of the earliest and most widely accepted definitions is from Jeannette Wing (2006), who emphasized that CT is a general attitude and broad skill set, as opposed to an explicit and narrow list of facts.

Wing's seminal ideas on CT had broad influence, and have been largely incorporated into the definition of CT from the *International Society for Technology and Education (ISTE)* and the *Computer Science Teachers Association (CSTA)*. These groups are two of the main voices in the establishment of K-12 computing education, and proposed an authoritative definition of CT comprised of two parts (Barr et al., 2011). The first involves *characteristics* of the CT *process*, which include the ability to:

Formulate problems in a way that enables the use of computers
Logically organize and analyze data
Represent data through abstractions such as models and simulations
Automate solutions through algorithmic thinking
Identify, analyze, and implement different possible solutions with efficiency in mind
Generalize this problem-solving approach to a wide variety of problems

Technical computing skills are not sufficient by themselves to solve problems via the use of computing power. Problem-solving with computers is a difficult and often lengthy process, so success also requires a set of attitudes that allow students to persevere in the face of adversity. These attitudes include:

Confidence in dealing with complexity
Persistence in working with difficult problems
Tolerance for ambiguity
The ability to deal with open-ended problems
The ability to communicate and work with others

There is a rich and growing research base on the use of gaming and robotics to address specific CT skills in students (Atmatzidou & Demetriadis, 2016), providing evidence of increased communication and collaboration skills (Ardito et al., 2014; Khanlari, 2013; Yuen et al., 2014), motivation (Webb, Repenning, & Koh, 2012), complex problem-solving skills (Akcaoglu, 2016; Akcaoglu & Koehler, 2014), abstraction (Nickerson et al., 2015), and transfer (Repenning et al., 2015).

Another important issue is how much CT skills the teachers themselves need (Repenning et al., 2015; Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014). Additionally, simply having the CT skills may not be enough; self-awareness of these skills may be necessary. While it is convenient to believe that teachers with general skills and expertise in non-computing subjects can learn just enough computing through professional development to introduce CT in their classrooms, we believe that it is crucial that teachers have first-hand experience with the affective challenges that face anyone who is learning CT.

Professional Development of CT

Rather than attempting to address all of the elements of CT in the UGICT project, an operational definition was developed based on the following precepts:

- Modeling is at the heart of CT.
- CT is not just about programming skills.
- Solutions can be generalized and transferred to other situations.
- CT is about persistence and dealing with failure.

To help teachers achieve these understandings and skills, the UGICT professional development focused around a set of modeling challenges involving both game programming and robotics, such as writing a simple version of *Pac-Man* and making a robot move in a circle with a 1 m diameter. From the computing perspective, these challenges may only be moderately difficult, requiring only sequential thinking and the basic principles of variables, alternation, and loops. However, for teachers who had little or no prior training in computing, these were daunting challenges. Additionally, teachers of different backgrounds found the challenges to be easier or more difficult, depending on those backgrounds. It was also natural for participants to find themselves working at different rates. This meant that the PD had to be very flexible.

Class Descriptions

In the first 2 years of the UGICT project, 28 participating teachers enrolled in short courses to prepare them for working with the target gaming and robotics technologies. These courses focused on software functionality and an exploration of how gaming and robotics can be used effectively to develop computational thinking skills, both in the participating teachers and in their students.

In the first year of the project, a single 8-week course was delivered, with 4 weeks dedicated to gaming and 4 weeks to robotics. In the second year, additional time allowed for the delivery of two 8-week courses, one for each technology. Due to the low population density of Wyoming, it was infeasible to have classes meet

face-to-face every week. Class sessions were held synchronously online to allow for screen and file sharing, chatting, and display of instructor webcam video. Class sessions were held once a week in 2-hour blocks and were recorded.

During the gaming segment participants learned about programming using AgentSheets, AgentCubes (Repenning, 2012), Scratch, and Bootstrap authoring systems. The second segment introduced participants to building and programming with the LEGO EV3 system. Common threads of the courses included (1) modeling (meaning, data, and knowledge representation) as the heart of programming, (2) the computational thinking skills that are required to build computer games and solutions to robotics challenges, and (3) how these relate to appropriate STEM content standards.

AgentSheets is a visual programming environment that can be used to create 2D games and simulations. The playing field, called a *worksheet*, is comprised of a 2D array of cells, each of the same size, e.g., 32 × 32 pixels. Each cell can house one or more *agents*, which may be stacked on top of each other. Agents make up all of the visual elements in the game, including the background, stationary objects like rocks, an avatar for the player to control, the antagonists, and any other game components such as robots and chairs. Programming in AgentSheets consists of choosing which agents to place in a worksheet and providing behavior via custom rules.

Computational thinking is explicit in the AgentSheets and AgentCubes programming environments through the idea of *Scalable Game Design (SGD)* (Repenning et al, 2015). An important aspect of SGD is the psychological principle of *flow*, which seeks to strike a middle ground between boredom and anxiety for students at different stages in computational thinking. This is accomplished, in part, via a sequenced curriculum with a progression of games that are increasingly difficult to build and with different computational thinking patterns (see Fig. 1). Consequently, as students progress in their technical skills, they are exposed to more challenging

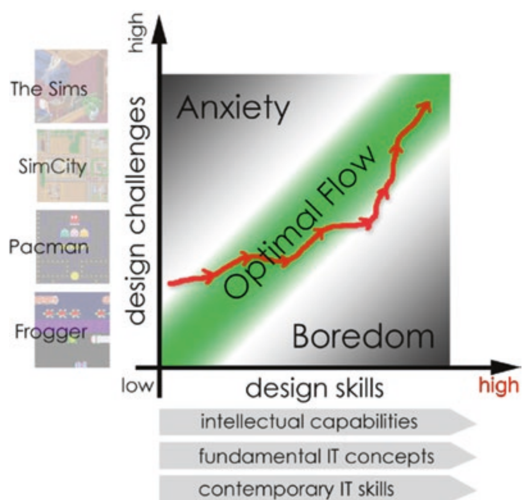

Fig. 1 Interrelationship of design challenges and anxiety in determining optimal flow (*Source*: http://www.agentsheets.com/education/scalable-game-design/index.html)

problems. In turn, as they work on more challenging problems, they learn more computational thinking patterns, helping their skills develop further. In the end, students are working on simulations, as opposed to games, but they learn that concepts such as diffusion or hill climbing that they learned in the context of computer games transfer very naturally to the context of simulations in science, public policy, or any number of different fields.

The notion of computational thinking patterns is also pedagogically important in SGD. The idea is that games and simulations are constructed using a relatively small set of patterns, such as diffusion and hill climbing which are central to the game of *Pac-Man*. In fact, programs in AgentSheets and AgentCubes can be inspected mechanically for evidence that they use these patterns, which provides an easy, automated way of measuring growth in computational thinking patterns, if not the totality of computational thinking as defined by ISTE and CSTA.

Our PD program was designed to help teachers understand how to use AgentSheets and AgentCubes and how these programs foster computational thinking. We proceeded by leading teachers through a sequence of activities that they could use directly in their after-school program, and as we did so, we discussed the CT skills and attitudes that were involved.

The very first task was to create one or more agents. In AgentSheets, the agents are 2D image files, and as mentioned previously, are of a fixed size, e.g., 32 × 32 pixels. Similarly, agents in AgentCubes are 3D models that live inside a volume of fixed size, e.g., 32 × 32 × 32 voxels. This activity was open-ended, and both instructors and participants had the opportunity to be as creative as they wished. Some chose to use minimal artwork, creating nothing more than stick figures, or to find suitable images on the Internet. Others, however, seized the opportunity to exercise their creative talents and produce, for example, magnificent 3D plants and animals. We encouraged this artistic exploration, because it gave teachers and students a chance to make their creations uniquely theirs. An important aspect of this exploration is that it gave participants the opportunity to bring in their sense of culture into their project. There is great value in having each participant produce a different game, one that is uniquely meaningful to him or her, as opposed to having all students produce an almost identical version of *Pac-Man*.

The 2D image or 3D model is only a portion of the agent. Agents can have more than one image, or *depiction*, which they can change programmatically. The other portion of the agent is the programmatic part, which is encoded as a list of behaviors. An agent's behavior is grouped into methods, which are activated upon a trigger. For example, a method may be active when the agent is asked to "move left," or it may simply be active whenever the agent "is running." A method consists of one or more rules of the form *IF some-condition-is-true THEN do-some-action*. The conditions can check the value of program variables and agent variables, check which agents are in the agent's cell or neighboring cells, and also check for user actions, such as pressing the space bar or an arrow key. The second task, then, was to add behavior to the agents, so that they would respond to arrow keys. For instance, when the user pressed the left arrow key, the agent moved left.

These first activities addressed many of the aspects of CT. In particular, participants could readily appreciate the power of abstraction. For example, agents appear to follow a corridor, but the program is simply checking that the image in the cell next to the agent is a depiction of a floor which can be transversed. Additionally, participants were able to automate solutions through algorithmic thinking, such as creating rules for controlling the movement of agents. In this case, the basics may seem obvious: If the user presses the left arrow key, then the agent moves to the adjacent left cell. However, even this simple rule is riddled with complexities, such as "What if the agent is in the leftmost cell?" or "What if the cell to the left is already occupied?" As this illustrates, before attempting the task participants needed to clearly organize their thoughts, an act which is the essence of computational thinking. Through these activities, participants also learned to appreciate the attitudes necessary for success in these activities, such as the ability to work on open-ended problems and persistence.

Persistence is probably the most important quality one needs to have when dealing with computers. Computing professionals spend more time correcting their programs than writing them. Some errors are caused by nothing more than carelessness. For example, once the rule for moving left is complete, it is easy to modify it to create a rule for moving right. In doing so, however, it is possible, and even likely, that the new rule is slightly wrong, perhaps by still moving the agent left instead of right. These errors can be painful, and almost all participants experienced the frustration of not being able to spot these trivialities immediately.

More subtle problems arose because of misunderstandings. The simplicity of AgentSheets belies a very complex execution model. For example, consider two agents close to each other. The one to the left moves right whenever the cell to the right is unoccupied, and the one to the right does the same, but moving left. Is it possible for both agents to move to an unoccupied cell at the same time? This depends, of course, on the order in which the tests and movement of the agents take place. In other words, this depends on the way that AgentSheets implements the agent behavior, and these details are deliberately hidden from the programmer. It is, after all, what makes AgentSheets simple.

Normally, this does not present a problem, because however AgentSheets chooses to implement the agents' behavior will not materially affect the outcome of the game. In those cases where it does make a difference however, it is important to determine exactly what will happen, and the only way to know is through experimentation. The designing of good experiments, which is to say small programs, requires more aspects of computational thinking. In particular, it requires participants to formulate problems in ways that enable the use of computers, and logically organize and analyze data.

Once the participants understood the basics of AgentSheets, they could begin to create playable games. So for the next activity, we asked the participants to consider what makes an arcade-style game. The main components were quickly identified, such as an avatar, one or more dangers, one or more goals, and one or more antagonists. The first project was the game of *Frogger*, with the frog as the protagonist, trucks and water as antagonists, and the grotto across the river as the goal.

To ease into this complex game, the participants first developed a simple game in which a protagonist moved according to user inputs, and one or more antagonists moved at random. Participants were given no further instructions, so had to be creative in choosing the game's setting, characters and rules.

There was no single right solution to this activity, and many participants found this freedom of choice unsettling. They received more detailed instructions for the next activity, however, which was to recreate a small version of *Frogger*. Participants were surprised to discover that no new skills were required to build *Frogger*. The only differences were in scale and complexity, in that there were many more agents in the game of *Frogger*.

The final project that participants were asked to build was a small version of *Pac-Man*. The primary difference between *Pac-Man* and the first activity they engaged in is the behavior of the ghosts. Whereas in the first activity the antagonists moved at random, in *Pac-Man* the ghosts *appear* to follow the avatar. We emphasize the word "appear," because the ghosts are actually following a much simpler rule. Again, this was used to build another connection to computational thinking, namely, formulating problems in a way that enables the use of computers and representing data through abstractions.

The way in which the ghosts appear to chase *Pac-Man* is quite clever, and we openly shared this solution with the participants. The protagonist, *Pac-Man*, is a source of "heat," so the cell in which *Pac-Man* resides is very hot. Heat flows from hotter cells to the neighboring cells in a process called *diffusion*, which SGD counts as one of the basic computational thinking patterns. The ghosts can sense the temperature of their cell and the surrounding cells, and they move toward the hottest neighboring cell, breaking ties at random. This process is called *hill climbing*, and it is another of the basic computational thinking patterns.

The combination of diffusion and hill climbing creates the illusion that the ghosts are chasing *Pac-Man*, but in reality each process is a simple mathematical rule that looks only at the value of "temperature" in neighboring cells. This last example emphasizes the importance of abstraction in computational thinking. The entire concept of "temperature" flowing from *Pac-Man* to its surroundings is a fable born of abstraction. The more mundane reality is strictly about cell values and averages. However, it is the essence of computation that seemingly complex behaviors – such as ghosts chasing *Pac-Man* – are the product of simple rules. This is the last lesson that participants gained from game programming, and it is an important one.

After learning how to build games with AgentSheets and AgentCubes, participants switched to robotics with the LEGO EV3 system. Programming the EV3 is quite different than game programming with the SGD platform. EV3 programs are constructed by dragging and connecting LEGO-style bricks on the screen. Each brick corresponds to a programming concept, such as an IF-statement or a loop, and the connections between the blocks specify the order in which blocks are executed. Blocks can have different parameters, such as the amount of power for a specific motor, and parameters may be filled in directly (e.g., 50%) or taken from another block by connecting the two with a wire. Despite the LEGO-style interface, programming the EV3 is a lot closer to traditional programming than the SGD platform, because the blocks and

wires correspond very naturally to programming language constructs, such as control statements and variables. Moreover, the execution of an EV3 program is mostly sequential, so that students can think of "which block is currently executing," much as programmers in Python or another traditional language think of "which line is currently executing." This stands in contrast to AgentSheets and AgentCubes, where each agent is executing its own program at the same time as all other agents, and as mentioned previously, this can lead to subtle timing interactions between agents.

There is, of course, another fundamental difference between game programming with SGD and robotics programming with the EV3. Robotics programming includes a physical component, which is the EV3 robot, its sensors, and the motors that communicate with the outside world. This creates an opportunity to emphasize a computational thinking principle, namely, that programs are *models* of certain aspects of a world. The world could be completely virtual, as in a game, where the laws of physics may be substantially different than in our own. Or it may be our world, in which case the model needs to capture enough of the real world to be useful. For instance, in robotics, the model may need to take into account the friction between the robot's wheels and the ground.

The first robotics activity was intended to familiarize the participants with the EV3 "brick" robot, its motors, and sensors. Participants started with the most common sensors, including the color and ultrasound sensors, which can be used to follow a road and stop when approaching an obstacle. They also learned about the touch sensor, which is commonly used as a button or to confirm contact with a fixed object. Participants also learned about the buttons on the EV3 brick and how it can be connected with a computer running the EV3 software, so they could download simple programs to the robot. They were then given a simple task to build a robot with a single motor and a rotation indicator. Building the robot was the focus of this task, which was intended to help participants become comfortable with the physical materials.

Once this first task was complete, participants were asked to become familiar with the EV3 programming environment. In particular, they learned about the different (virtual) blocks in the environment and how they interact with the input (sensors) and output (motors) of the EV3 brick. They also learned the more abstract blocks that correspond to programming concepts, such as wait, loops, conditionals, and variables. The activities then turned to debugging programs. This process is complicated in robotics, because the programming takes place on the EV3 environment, but the program is run in the actual EV3 brick. So when participants wrote a program (by placing and connecting virtual LEGO blocks on the screen), they had to imagine what the robot would do. Later, when they ran the program, they observed what the robot actually did, and from those observations had to infer what went wrong and make adjustments to the program.

Next, we gave the participants a program that drives the robot around a square. However, the program intentionally contained four bugs, which the participants were asked to find. Some of these bugs were subtle, and participants were unlikely to find them without actually running the program and observing what the robot did. For example, one of the bugs was that the robot turned right but said "left" as it did so. The purpose of this exercise was twofold. First, it exposed participants to the

unique challenges of debugging programs that run on a real-world robot. Second, it reinforced the idea that debugging is a natural part of the programming process and one that should not make them feel embarrassed or inadequate.

When participants engaged in a brute-force approach to the program/observe/modify cycle, they learned very little computational thinking. For example, suppose there is a goal to move the robot by 20 cm, and the programmer commands the motors to turn for 10 seconds. When the program is run the robot moves 21 cm, so the programmer changes the time to 9 s, which is not quite enough. By repeating this process, the programmer can eventually find the time required to move 20 cm, but with no full understanding of how the motor run time is related to the distance the robot travels.

A more nuanced approach is steeped in computational thinking. Instead of running the program once and seeing how far the robot moves, participants were encouraged to run the program multiple times with the same settings and record their observations. Surprisingly, the robot did not move the exact same distance each time. What participants then recognized is that the real world includes some variability; for example, as the robot moved along a carpet, it experienced different drag due to loose strands in the material. By taking multiple observations, they could find the average distance traveled for a given time, and from this table of facts, they could infer the exact time required to move exactly 20 cm. All of this reinforced the idea that the program is really modeling an aspect of the real world. Moreover, the simplistic model that is suggested by measuring the circumference of the wheels ignores the interaction between the wheels and the ground, so only works in ideal circumstances – what physicists refer to as “rolling without slipping.”

Assessment of Teacher CT Attitudes and Practices

A pre-post survey of attitudes toward CT, modified from Yadav, Zhou, Mayfield, Hambrusch, and Korb (2011), was administered to each cohort of participating teachers. Twenty-one items presented statements about CT and CS in five key areas, to which participants responded on a four-point strongly agree/disagree Likert scale with no neutral option (see Fig. 2). These areas include understanding CT, self-efficacy, intrinsic motivation, integration of CT in classroom practice, and career relevance of CT.

Statement	SA	Agree	Disagree	SD
Computational thinking involves using computers to solve problems.				
Computational thinking can be incorporated in the classroom by allowing students to problem solve.				

Fig. 2 Sample CT/CS survey items

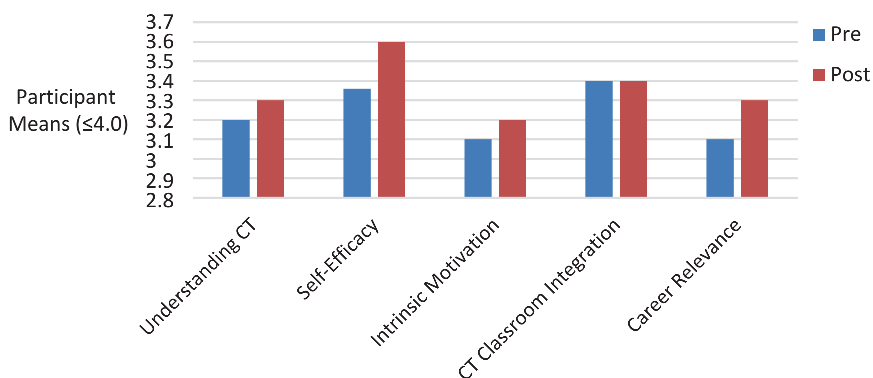

Fig. 3 Changes in teacher CT understanding and attitudes

The first year cohort consisted of twelve teachers, so analyses of survey results were not conducted for statistical significance. Descriptive data from the first cohort of twelve teacher participants revealed that attitudes and dispositions toward CT were positive and remained relatively stable in all five areas (see Fig. 3).

While participating in the PD classes, teachers demonstrated evidence of their own computational thinking. For instance, one challenge the participants faced was that of programming a robot to drive in the shape of an equilateral triangle, stopping as close as possible to the start point. The robot construction guide was simplified for quick assembly, using as few pieces as possible, including the use of a non-turning rear wheel. The participants soon discovered that the robot design was not adequate for what they needed to accomplish, both in terms of robot stability and maneuverability. One teacher noted, "The drag on the back of the robot would cause the robot to go off track and course. The attachments to the wheels are not tight so that impacted it as well." To solve this, some of the participants replaced the wheel with a ball bearing. Another suggested, "You could also make a swivel wheel with the NXT kit that doesn't have the ball."

Site visits were also made to conduct observations of teacher practices during club sessions. From these observations, patterns of teacher behaviors emerged that appeared to either facilitate or inhibit student success and CT development. Some teachers, out of a desire to let students have maximum freedom to create and explore, provided little direct instruction and left learning activities unstructured. These teachers were mostly confronted with frustrated and unsuccessful students. One middle school teacher, for instance, allowed students to work individually or in groups on their unique robotics projects. No instruction on the use of programming solutions or strategies was provided. As a result, students primarily used trial and error to address problems. A student working by himself had built a robot that was meant to drive forward and knock objects out of the way with a rotating arm. The student repeatedly set the robot on the floor aimed at a specific object and activated the program. Most of the tests resulted in the robot missing the object, as the rotating arm would randomly change the robot's path. The student's solution was to

move the object closer to maximize the likelihood of contact. The student persevered for the entire session, but was frustrated with his lack of success. It was later learned that he had not considered the use of sensors to detect the object and had limited programming expertise. Thus, his robot was programmed to simply drive forward in a straight line.

The most successful teachers provided a mix of direct instruction and open-ended exploration. These successful teachers were observed scaffolding student knowledge of computational thinking through the use of specific tutorial lessons. This often took the form of teaching a specific skill or concept at the beginning of a session. All of the teams would then be asked to create a simple program that would then incorporate the skill or concept and then create a larger project. Students in one club learned how to use a sound block to create a single tone on the music scale and then string together four or five tones to play the beginning of a familiar tune. Teams were then challenged to program their robots to move rhythmically or "dance" while playing a full tune of their choice. One team successfully tackled the challenge of programming the entire melody of "The Star-Spangled Banner."

For gaming, many successful teachers used *Frogger* tutorials as a starting point for their students. This allowed students to learn the functions of the software and game design processes, including debugging, in a structured setting, with increasing level of difficulty. Some teachers then asked students to use the *Pac-Man* tutorials, while others asked students to create original maze games based on the same premise.

Regardless of the teaching approach, most of the teachers were observed providing encouragement and problem-solving hints and tips, while asking probing questions to develop and extend computational thinking skills. These typically took the form of "what if you were to," "how would you," and "have you considered" probes. To develop problem-solving skills, teachers stated that they also promote the use of other strategies, including drawing solutions, discussing alternative solutions as teams, and relating challenges to more familiar circumstances. One teacher said that she tells her students, "Failure is a learning opportunity, not an end." Another told her students to "work backward when you encounter a roadblock – see where the problem is." Through this, she was trying to teach her students the concept of "resilience."

Conclusion

We learned much from our observations. Probably the most important and hopeful realization we made is that promoting computational thinking requires many skills and that teachers already have most of them. Dealing with complexity, having perseverance, and accepting open-ended problems are important skills in the computational thinking context, but this is not the only context in which these attitudes are useful. Teachers are already consciously helping these students to develop these skills, and where they need help is in placing these skills in the context of computational thinking.

Another observation we made regards the difference between computational and technology skills. As a general rule, both students and teachers tended to be well versed in technology. At the risk of overgeneralizing, we can also add that most students tend to be more comfortable with technology than most teachers. This can create an obstacle, as some teachers question whether they can teach their students about computing at all. This fear, however, may stem from confusion between computational thinking and knowing about technology. As we have seen, computational thinking is a rich mixture of cognitive skills and attitudes, whereas knowing about technology simply entails extensive time with the latest devices. What many students and teachers fail to realize is that becoming an expert in playing video games does not translate into expertise in programming, whether game programming, robotics programming, or any other form. Familiarity with technology is helpful, for example, in understanding about files, printers, or creating images, but it does not lead directly into computational thinking.

We also found that classrooms that were focused on questions, as opposed to answers, were more effective in fostering computational thinking. For instance, when a student is failing at solving a problem, such as having a robot move in a straight line for a specific distance, the teacher can respond either by suggesting a solution or by asking an appropriate question. In this particular case, a teacher may respond by showing the student how to change the block that controls the motors, or she may ask the student how far he thinks the robot will go if the wheels turn ten times. This type of inquiry leads to deeper insights and to the discipline at the heart of computational thinking. Providing teachers with good questions to ask will better prepare them to help their students to learn computational thinking, not just to solve the computing problem at hand.

We also identified some deficiencies of the program, which should lead to changes in future iterations. The PD class we offered teachers was only 8 weeks. This was barely enough time to familiarize the teachers with the projects and activities that they could share with their students in their after-school programs. Teachers were asked to perform significant computing tasks, and not all could afford the commitment of time required to finish these tasks. Consequently, many teachers were still uncertain about their own abilities in computational thinking, and that led to significant stress as they engaged with their own students. Moreover, the short time did not allow the teachers to delve deeply into the question of methods for imparting computational thinking to their own students. Both of these issues can be addressed by lengthening the PD.

Despite these limitations, the data already collected suggests that these after-school programs do work in enhancing students' computational thinking skills. Moreover, teachers who are sufficiently confident in their own skills to let students work independently – as opposed to blindly following instructions – are the most effective. Further support to increase teachers' comfort with computing and the pedagogy of computational thinking will lead to improved success.

Acknowledgments This material is based upon work supported by the National Science Foundation (DRL #1311810). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Akcaoglu, M. (2016). Design and implementation of the Game-Design and Learning program. *TechTrends, 60*(2), 114–123. doi:10.1007/s11528-016-0022-y.

Akcaoglu, M., & Koehler, M. J. (2014). Cognitive outcomes from the Game-Design and Learning (GDL) after-school program. *Computers & Education, 75*, 72–81. doi:10.1016/j.compedu.2014.02.003.

Ardito, G., Mosley, P., & Scollins, L. (2014). WE, ROBOT: Using robotics to promote collaborative and mathematics learning in a middle school classroom. *Middle Grades Research Journal, 9*(3), 73–88.

Atmatzidou, S., & Demetriadis, S. (2016). Advancing students' computational thinking skills through educational robotics: A study on age and gender relevant differences. *Robotics and Autonomous Systems, 75*(Part B), 661–670. doi:10.1016/j.robot.2015.10.008.

Barr, D., Harrison, J., & Conery, L. (2011). Computational thinking: a digital age skill for everyone: the National Science Foundation has assembled a group of thought leaders to bring the concepts of computational thinking to the K-12 classroom. *Learning & Leading with Technology, 38*(6), 20.

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

Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education, 72*, 145–157. doi:10.1016/j.compedu.2013.10.020.

Burke, Q., O'Byrne, W. I., & Kafai, Y. B. (2016). Computational Participation. *Journal of Adolescent & Adult Literacy, 59*(4), 371–375. doi:10.1002/jaal.496.

Denner, J., Werner, L., & Ortiz, E. (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education, 58*(1), 240–249. doi:10.1016/j.compedu.2011.08.006.

Dewey, J. (1916). *Democracy and education: an introduction to the philosophy of education*. New York: Macmillan.

Dewey, J. (1958). *Experience and Nature* (Vol. 1st ser, 2nd ed.). La Salle, IL: Open Court Publishing Company.

Educational Policies Commission. (1961). *The Central Purpose of American Education* (p. 27). Washington, D.C: National Education Association. Retrieved from http://eric.ed.gov/?id=ED029836.

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

Honeyford, M. A., & Boyd, K. (2015). Learning through play: portraits, photoshop, and visual literacy practices. *Journal of Adolescent & Adult Literacy, 59*(1), 63–73. doi:10.1002/jaal.428.

Kafai, Y. (1995). *Minds in Play: Computer Game Design As a Context for Children's Learning*. Hillsdale, NJ: L. Erlbaum Associates Inc..

Kafai, Y. (1996). Software by kids for kids. *Communications of the ACM, 39*(4), 38.

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

Kafai, Y., & Burke, Q. (2013). Computer programming goes back to school: Learning programming introduces students to solving problems, designing applications, and making connections online. *Phi Delta Kappan, 95*(1), 61–65.

Khanlari, A. (2013). Effects of robotics on 21st Century Skills. *European Scientific Journal, 9*(27.) Retrieved from http://search.proquest.com.libproxy.uwyo.edu/docview/1524821792/abstract/E91502C4C03E4CCEPQ/1.

Lave, J., & Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation (Learning in Doing: Social, Cognitive and Computational Perspectives)*. Cambridge, UK: Cambridge University Press.

Li, Q. (2010). Digital game building: learning in a participatory culture. *Educational Research, 52*(4), 427–443. doi:10.1080/00131881.2010.524752.

National Research Council. (2011). *Committee for the Workshops on Computational Thinking: Report of a Workshop on the Pedagogical Aspects of Computational Thinking*. Washington, DC: National Academies Press.

Nickerson, H., Brand, C., & Repenning, A. (2015). *Grounding Computational Thinking Skill Acquisition Through Contextualized Instruction* (pp. 207–216). Proceedings of the eleventh annual *International Conference on International Computing Education Research*. Omaha, NE: ACM Press. doi:10.1145/2787622.2787720.

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

Papert, S., & Harel, I. (1991). Situating Constructionism. In *Constructionism* (pp. 1–11). Norwood, NJ: Ablex Publishing Corporation. Retrieved from http://www.papert.org/articles/SituatingConstructionism.html.

Piaget, J., & Inhelder, B. (1969). *The Psychology of the Child*. New York: Basic Books.

Repenning, A. (2012). Programming goes back to school. *Communications of the ACM, 55*(5), 38. doi:10.1145/2160718.2160729.

Repenning, A., Webb, D. C., Koh, K. H., Nickerson, H., Miller, S. B., Brand, C., et al. (2015). Scalable game design: a strategy to bring systemic computer science education to schools through game design and simulation creation. *ACM Transactions on Computing Education (TOCE), 15*(2), 1–31. doi:10.1145/2700517.

Salomon, G., & Perkins, D. N. (1998). Individual and social aspects of learning. *Review of Research in Education, 23*, 1–24. doi:10.2307/1167286.

Uchida, D., Cetron, M., & McKenzie, F. (1996). *Preparing-Students-for-the-21st-Century*. Lanham, MD: Rowman & Littlefield Education. Retrieved from https://rowman.com/ISBN/9781578860470/Preparing-Students-for-the-21st-Century.

Vygotskiĭ, L. S., & Cole, M. (1978). *Mind in Society: the Development of Higher Psychological Processes*. Cambridge: Harvard University Press.

Webb, D. C., Repenning, A., & Koh, K. H. (2012). Toward an emergent theory of broadening participation in computer science education. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 173–178). ACM. Retrieved from http://dl.acm.org.libproxy.uwyo.edu/citation.cfm?id=2157191.

Wing, J. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35.

Yadav, A., Mayfield, C., Zhou, N., Hambrusch, S., & Korb, J. T. (2014). Computational thinking in elementary and secondary teacher education. *Transactions on Computing Education, 14*(1), 5:1–5:16. doi:10.1145/2576872.

Yadav, A., Zhou, N., Mayfield, C., Hambrusch, S., & Korb, J. T. (2011). Introducing computational thinking in education courses. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education* (pp. 465–470). New York, NY, USA: ACM. doi:10.1145/1953163.1953297.

Yuen, T. T., Boecking, M., Stone, J., Tiger, E. P., Gomez, A., Guillen, A., & Arreguin, A. (2014). Group tasks, activities, dynamics, and interactions in collaborative robotics projects with elementary and middle school children. *Journal of STEM Education: Innovations and Research, 15*(1), 39.