TechTrends (2016) 60:114–123
DOI 10.1007/s11528-016-0022-y

ORIGINAL PAPER

Design and Implementation of the Game-Design and Learning Program

Mete Akcaoglu[1]

Published online: 26 February 2016
© Association for Educational Communications & Technology 2016

Abstract Design involves solving complex, ill-structured problems. Design tasks are consequently, appropriate contexts for children to exercise higher-order thinking and problem-solving skills. Although creating engaging and authentic design contexts for young children is difficult within the confines of traditional schooling, recently, game-design has emerged as an alternative context to provide young children with opportunities to practice design and thinking skills. Despite the increasing interest from educators and researchers to use game-design as a platform to teach young students higher-order thinking skills, literature documenting design and development of such learning experiences has been scarce. This paper provides a detailed account of how a complex network of pedagogies, theories, and technologies were brought together to design and develop the Game Design and Learning (GDL) program, with the purpose of teaching students basics of computer programming, and to give them hands-on experiences in game-design, and teach them complex problem-solving skills. The GDL program can serve as an example for efforts aiming to create similar technology-rich environments.

Keywords Problem-solving · Game-design · Microsoft Kodu · Constructionism · Design · STEM

Design is an important cognitive task and a quintessential ill-structured problem (Goel and Pirolli 1992; Jonassen 2011). It involves creating new objects, processes, or ideas by synthesizing different variables in innumerable and unique ways (Simon 1995). During this process, designers engage in important cognitive skills such as problem-solving, problem-finding, and inquiry (Smith and Boling 2009).

Design skills, and the underlying problem-solving processes that are activated and practiced through design, are utilized frequently in our daily lives (Smith and Boling 2009). In addition, jobs in science, technology, engineering, and math (STEM) domains inherently involve solving ill-structured design problems (Jonassen 2011). Developing design skills is, therefore, especially essential for young children to be successful in most future careers (Eseryel et al. 2013; Jonassen 2011). Opportunities to practice design and problem-solving skills, however, are difficult to come by in formal schooling contexts, because schools often emphasize solving well-structured problems, as opposed to ill-structured real-world problems that require direct engagement with objects and situation-specific competencies (Resnick 1987).

Since 1970s, researchers and educators have been interested in the potential of computer programming and software design in creating authentic and meaningful contexts to teach young students design and problem-solving skills (e.g., Harel and Papert 1990; Papert 1980). In the last decade, digital game-design tasks have attracted attention as an engaging task to teach children complex thinking skills (e.g., Akcaoglu and Koehler 2014; Denner et al.2012; Hwang et al.2013).

The attractiveness of game-design tasks as educational activities can be explained by several factors. First, games are inherently attractive for young children (Gee 2003; Prensky 2003). The process of game-design, therefore, has a natural appeal, because the outcome of the process (i.e., the games) is meaningful and fun for the creators. This process is, therefore, enjoyable and intrinsically motivating as the students get to work on things they personally value. Second, games are

✉ Mete Akcaoglu
makcaoglu@georgiasouthern.edu

[1] Department of Leadership, Technology, and Human Development, College of Education, Georgia Southern University, Statesboro, GA 30458, USA

complex systems, made up of many interrelated variables (Fullerton 2008; Robertson 2012). Similarly, the game-design process is also ill-structured, requiring designers to satisfy many interrelated variables, troubleshoot emerging problems, and make key decisions for their games to work. In this sense, game-design process, as a design task, requires activation of important metacognitive skills, making it an ideal context to practice problem-solving skills (Ke 2014). Third, during the design process young children get a chance to create external (visual) representations of their otherwise abstract ideas (Baytak and Land 2010). Externalization of mental representations is an important aspect of the design process and problem-solving (Bonnardel and Zenasni 2010), and during the game-design process, children benefit greatly from being able to see the outcomes of their design actions through immediate visual feedback. Finally, during the creation of digital games students have the opportunity to engage in computer programming (Denner et al. 2012). Especially with new programming interfaces, new generation game-design software (e.g., "Microsoft Kodu" 2012; "Scratch" 2012) are intuitive, even to the degree to be used easily by children as young as nine or ten (Fowler and Cusack 2011). For these reasons, game-design tasks are fun, authentic, and appropriate contexts to teach design and problem-solving skills to young children.

Designing technology-rich and engaging learning environments for young students (i.e., digital game-design) while also aiming to teach them important thinking skills such as problem-solving, however, requires going through a rigorous instructional design process. The process of design needs to be based on theory, grounded in data, and focused on problem-solving (Smith and Boling 2009). The creation of the technology-rich environment, in addition to bringing together content and pedagogies to create a harmonious whole (Mishra and Koehler 2006), also requires working with technology. The addition of technology into this already complex instructional design task makes the process of technology integration more complex (Koehler and Mishra 2009).

In this descriptive case, the design process of an instructional intervention called the *Game Design and Learning* (GDL) program is detailed. The GDL program was offered as an after-school program (7, three-hour sessions) or a summer camp (8, five-hour sessions) to middle school students with the purposes of teaching them game-design, programming, and problem-solving. In the remainder of the paper, details regarding the design process of the GDL activities and an overview of the theories, pedagogies and the technologies is provided. By explaining the rationale behind design decisions and giving examples from practical aspects of the GDL program, this paper may serve as an important design document detailing the design process of a technology-rich environment to teach students higher-order thinking skills.

Design and Development of the GDL Program

Building on early work by Papert (1980) where children engaged in programming with Logo software, the GDL Program aimed to get students to engage in the construction of meaningful technology-based artifacts. The main objective of the GDL program was to serve as a context to teach young students (9–15 years old) the basics of game-design, programming, and problem-solving, while also providing them with meaningful, situated, and hands-on experiences in design. This objective required the design of the GDL program to: (a) take full advantage of available game-design software (Microsoft Kodu), (b) be based on pedagogies that favor exploration of ideas and construction of artifacts and treat learners as designers, and (c) effectively incorporate methods of teaching problem-solving.

Technology: Microsoft Kodu

Microsoft Kodu was selected as the game-design software for the GDL program. As a free download, one of the main affordances of Kodu is that it allows young learners to get hands-on practice with basic computer programming concepts, such as "if-then" statements, variables, and conditionals (MacLaurin 2011; Stolee and Fristoe 2011). While the primary goal when using the software is to create games, the process of creating digital games naturally requires students to understand and work with computer science concepts.

Among other available options (e.g., Scratch, Alice, GameMaker), during the GDL program Kodu was used for several reasons. First, Kodu environment is three-dimensional (3D), as opposed to other popular game-design software (e.g., Scratch) (Fig. 1). Compared to 2D environments, the ability to create 3D games in Kodu makes it visually more appealing for young students. Thanks to its built in real-time 3D game engine, starting from their first encounter with Kodu, even young learners can create games akin to the games they play regularly (MacLaurin 2011). The real-time 3D game engine helps students create games that can compete with modern console games, while fulfilling common gaming assumptions (e.g., physics) without any effort on the part of the learner.

Kodu also allows for creation of computer-run simulations. Simulations differ from games in that when users run simulations they can function as microworlds to observe and count user-defined events. Such an affordance can be used to reverse-engineer problem scenarios to observe and visualize causes, and devise a solution. An example of such a task can be the creation of a simulation to replicate an environmental issue (e.g., pollution problem), where learners design a system that allows them to count and keep track of sources of pollution, letting them visualize ways of possible solutions. For the reasons listed, Kodu was used during the GDL program.

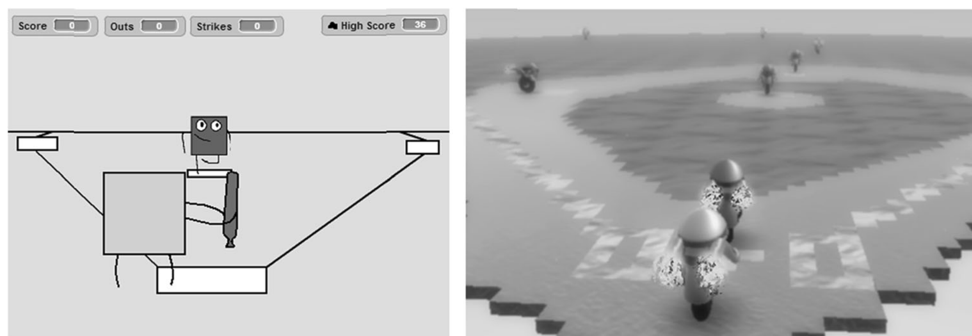


Fig. 1 Screenshots of a baseball game in Scratch (*left*) and Kodu (*right*)

Finally, even with its simplified user interface, Kodu introduces learners to basics of computing concepts (Stolee and Fristoe 2011) in a game-design context. Based on an event-driven visual programming language, Kodu relieves younger learners from the level of abstraction required in other game-design or programming tools (MacLaurin 2011). This helps overcome some of the difficulties learners face while learning programming (Fowler and Cusack 2011). Despite this simplicity, however, Kodu contains important computer programming concepts: global variables, local variables, non-determinism (via random), boolean logic (negation, conjunction, disjunction), objects, control flow (i.e., non-linear program flow), looping, conditionals, states, and parallelism (Stolee and Fristoe 2011; Touretzky et al. 2013).

Pedagogical Approaches

During the GDL program, the students were encouraged to overcome problems, explore and discover concepts independently, while guidance and support also were provided at times to help students establish connections among different concepts (i.e., guided discovery learning) (Mayer and Wittrock 1996). Due to the emphasis on giving students hands-on experiences in game-design, programming, and problem-solving; pedagogical approaches that allow students to actively design and construct knowledge, skills, and artifacts needed to be selected.

Learners as Designers Learning by design is "a constructivist approach that sees knowing as being situated in action and co-determined by individual-environment interactions" (Brown, Collins, & Duguid, 1989; Gibson, 1986; Roschelle & Clancey, 1992; Young, 1993, as cited by Koehler and Mishra 2005, p. 135). During the GDL program, the students found opportunities to act and think like designers by engaging in various game-design tasks. Conceiving of *learners as designers* is important for several reasons. First, in today's world, people are increasingly faced with complex problems that require novel solutions through design (Eseryel et al. 2013). When extant solutions do not apply to the new problems, the solution is usually found through creative design processes that demand higher-order thinking skills and creativity (Goel and Pirolli 1992; Jonassen 2000). Therefore, when learners get a chance to be designers, they get hands-on experiences in practicing an important skill that will be useful in their future careers and lives. Second, the process of design is personal, and therefore engaging. During design tasks, children get opportunities to play and tinker with objects, and construct personally meaningful and useful artifacts (Ackermann 2001). Finally, design tasks are also effective methods of teaching. Learners retain the most when they are engaged in design of their own knowledge (Mayer 1998), where it becomes more connected and meaningful.

Constructionism Coined by Seymour Papert, constructionism sees learning as an active process of building socially meaningful artifacts and knowledge (Ackermann 2001; Papert and Harel 1991). Although at its core constructionism is very similar to constructivism, Papert's constructionism slightly diverges from the "v" version when it adds the idea that "this happens especially felicitously in a context where the learner is consciously engaged in constructing a public entity whether it's a sand castle on the beach or a theory of the universe" (Papert and Harel 1991, p. 1). During the active construction process, learning becomes more personal and engaging, moves beyond "rote" learning and becomes more meaningful, connected and effective (Kafai 1995). During the GDL program, students actively engaged in creation of personally meaningful artifacts: digital games, making the already engaging process of game-design also meaningful.

Guided Discovery Learning Although in both guided discovery and pure discovery methods learners are expected to construct their own learning, the main difference between the two methods is that in guided discovery learners are supported through this process by getting "hints, direction, coaching, feedback, and/or modeling to keep the student on track" (Mayer 2004, p. 15). Timely guidance can be in the form of supporting learners in abstracting rules, especially when connections are not immediately apparent (e.g., Kurland et al.1986). Results of research comparing *guided* versus *pure* discovery learning methods favor guided discovery learning

when teaching thinking skills (Kirschner et al. 2006; Mayer 2004; Salomon and Perkins 1987). During the GDL program, students were encouraged to explore game-design and problem-solving concepts, but the instructional team provided structure to their experiences by presenting initial directions and timely support when they faced problems during the design and exploration process.

Instructional Methods

The instructional activities during the GDL courses were designed to be aligned with theories of problem-solving, and were based on well-established methods of teaching problem-solving. During the GDL program, problem-solving was operationalized as the cognitive process of overcoming barriers to reach a desired goal state (Funke 2010; Mayer 1977). Problem-solving process requires execution of component cognitive processes: understanding, representing, planning/monitoring, and executing (Jonassen 2011; Mayer and Wittrock 2006; Mayer 1977; Polya 1957). *Understanding* involves using the background knowledge and making sense of a given problem. *Representing* refers transforming external representations of a problem into internal mental representations (Jonassen 2000; Mayer and Wittrock 2006). It also involves generating hypotheses based on interrelationships among variables for a given problem situation. The final steps are to *plan* a solution for the problem by breaking it down its parts, and then *executing* solution (Jonassen 2000; Mayer and Wittrock 2006; Polya 1957). Activities during the GDL program, therefore, targeted showing students the flow of the problem-solving process.

As for instructional approaches to teaching problem-solving, four empirically-supported methods of teaching problem-solving as listed in the two extensive reviews by Mayer and Wittrock (1996, 2006) were used during the GDL program: (a) teaching basic skills, (b) teaching for understanding, (c) teaching by analogy, and (d) teaching thinking skills directly. *Teaching of basic skills* (Mayer and Wittrock 1996) encompasses teaching of specific low-level component cognitive skills pertaining to a task, so that in the ensuing tasks more effort can be placed upon executing higher-level cognitive skills (i.e., automaticity and cognitive load-reduction). In *load-reduction methods* the constraints in the environment that make the process of selecting, organizing and integrating new knowledge difficult are removed to ease the process of solving problems. *Teaching for understanding* (Mayer and Wittrock 2006) involves teaching of new cognitive skills in such a way that the freshly learned skills can be applied to novel situations. There are three widely used methods that can be used to teach for understanding. In *structure-based methods* learners are given concrete objects that can be manipulated. In *generative methods* teaching for understanding is promoted by helping students "generate relations between their existing knowledge and information to be learned" (Mayer and Wittrock 2006, p. 294). Finally, during the *discovery learning* methods, it is believed that learners actively construct connections between existing and new information, which eventually leads to meaningful learning. *Using analogies* is also another approach implemented in teaching problem solving. In this approach, "learners solve a new problem by using what they know about a related problem that they can [already] solve" (Mayer and Wittrock 1996, p. 55). In teaching by analogy, the premise is that by analyzing two problems that have surface similarity, the learners can abstract the underlying rules. Finally, *teaching thinking skills directly* is also a method utilized in teaching problem-solving. This method involves teaching metacognitive skills (e.g., problem decomposition) necessary to solve problems. Early research showed teaching thinking skills directly through after-school programs or courses is an effective way of teaching problem-solving (Mayer and Wittrock 1996, 2006). As it will be detailed later, these methods were incorporated into the GDL activities in different ways.

From Design to Implementation: GDL Activities

Based on the established instructional methods for teaching problem-solving that were applied within the context of guided learning theory, with key concepts of constructionism and learning-by-design, and the affordances of Kodu integrated, four types of activities were created to be offered during the GDL program: (a) game-design (first three sessions), (b) problem-solving (following two sessions), (b) troubleshooting (one session), and (d) free design (one session). The activities were sequenced to be offered at specific points during the GDL program (Fig. 2).

As it can be seen in Fig. 2, game-design activities were the through line of the GDL program, while other activities targeting specific skills (i.e., problem-solving) were introduced later, as students gained more experience and confidence in game-design and programming. Sequencing GDL activities in this manner (i.e., offering problem-solving activities only after students learned basics of digital game-design and programming) was based on load-reduction methods (e.g., automaticity and constraint-removal). According to the theories of automaticity and constraint-removal, mastery of low-level cognitive skills makes it easier for people to allocate

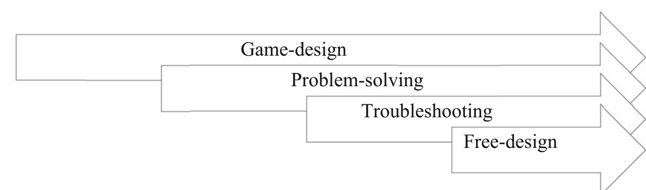


Fig. 2 Progression of different activities during GDL courses

more time, energy and cognitive facilities in solving problems requiring high-level cognitive skills (Mayer and Wittrock 1996). Based on load-reduction principle, the first set of activities at the GDL program aimed to teach students basic skills (e.g., how to use the software, basics of game-design, programming), eventually making it more probable for them to tackle more complex problem-solving tasks.

Overall Activity Structure

In terms of their basic instructional structure, each GDL activity, regardless of its specific type, started with an instructor introducing the activity to the students, and then guiding the students through the initial steps of the design. During the introduction stage, students were also introduced to new concepts in design and programming as well as skills in game-design and problem-solving (e.g., creating flowcharts of games). The sessions continued with students' iterating on their designs with assistance from instructors as needed.

Game-design Activities

Purpose and Structure Game-design activities were offered during the first sessions of the GDL program. The main objective of game-design activities was to teach students basics of game-design and programming. To this end, game-design activities were composed of smaller tasks that targeted: (a) identifying elements of games, (b) creating flowcharts of games to understand their systemic complexity, and (c) creating games in Kodu (from simple to complex).

Example An example game-design activity was creation of a game called "Apple Hunter." This activity was offered on the first session of the GDL programs to give students an initial sense of game-design, without overwhelming them. Apple Hunter (will be described later) was a very simple game where the goal was to eat five green apples and earn five points.

To introduce students to the game-design process and raise their awareness regarding the basic elements that make up games, the activity started with students *identifying basic elements* common to most games: Goals, Rules, Assets, Spaces, Play mechanics, and Scoring (GRASPS). By identifying the GRASPS of popular games as a class, students were guided in thinking about the games that they play at home. It also allowed students to think from a designer's perspective, perhaps for the first time.

Next, students were provided with GRASPS of their first game: Apple Hunter (Table 1). At this stage, the instructor quickly went over the GRASPS of the game with the students to check for understanding. During the initial sessions, since students did not have prior experience with Kodu or game-design process, the creation of Apple Hunter was done as a whole class, step by step, instructor leading each step and helping learners as they moved along. During this stage, students explored the basics of Kodu as well as programming and game-design.

Table 1 GRASPS for Apple Hunter game

Apple Hunter	
Goals	Eat 5 green apples
	Do not eat red apples
Rules	*(optional: character slows down after eating red apples)*
	(optional: in 20 seconds)
Assets	Kodu
	Apples
	Tree
Scoring	Green apple = +1 green point
	Red apple = −1 green point
Play mechanics	Wander around, look for/avoid apples, bump to eat
Spaces	Walled open world

The process of designing Apple Hunter started with creating a playable character and programming it to move with the help of the keyboard. After this first step, students created and programmed a tree that generates apples (green and red) at random intervals. Finally, students programmed their character to eat green apples upon touching them, and gain one point. After creating the basic game, students were encouraged to modify and improve their games, or recreate it from scratch. During this process, they had the option to choose to work independently or in small groups. Instructors provided support on an *as-needed* basis.

Game-design activities also involved creating *flowcharts* of games. Flowcharts are visual representations showing the interrelationship among various components of games as systems. During the game-design activities, students were guided in creating flowchart of the games that they create. This happened in a scaffolded manner: during the initial sessions students received complete flowcharts of the games that they needed to create (e.g., Apple Hunter -– see Fig. 3), then, gradually they received less complete ones, during the final sessions they got a blank sheet of paper and were asked to create the flowchart for a game that they imagined.

To help students understand how games work as systems and see the importance of flowcharts in the design process, the instructors provided initial guidance as to how to utilize them. For example, to give students chances to understand how flowcharts work, the instructor asked them how they would change the flowchart if, for example, the red apples ended the game immediately. Flowcharting was highly guided at the initial stages.

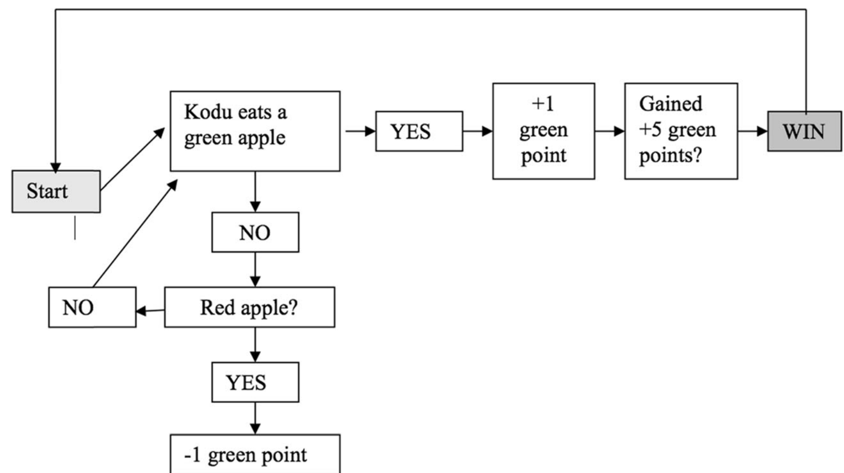


Fig. 3 Flowchart of the Apple Hunter game provided by the instructors

Rationale Game-design activities (and the smaller tasks offered with these activities) were the through-line of GDL program. During the game-design activities the students gained the basic knowledge required (i.e., constraint-removal) so that they could tackle the complexities of later tasks. In each game-design activity, the students were incrementally given more complex games to develop a further understanding of game-design and programming (for example, in the second session students created Pac-Man in Kodu). In addition, they were introduced to the idea of "games as complex systems," and were provided with tools to navigate the complexity. After multiple game-design sessions that solely targeted teaching students game-design and programming, the students were introduced to *problem-solving* activities.

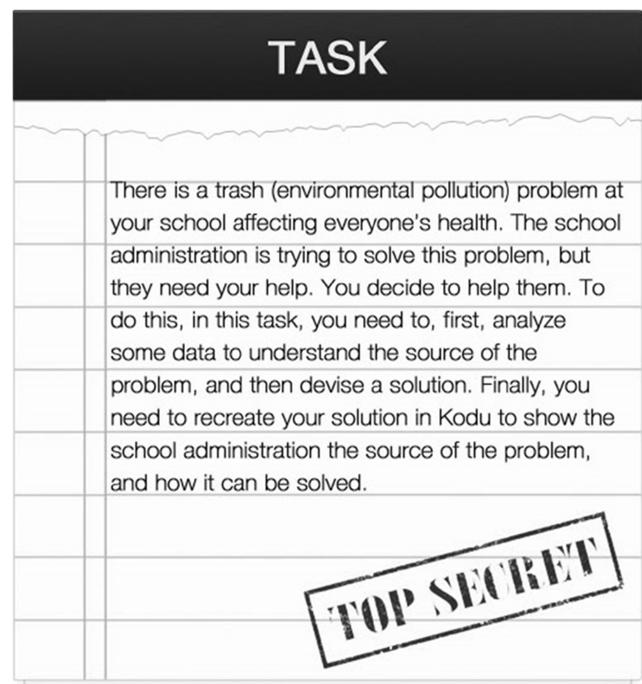
Problem-Solving Activities

Purpose and Structure After learning basic game-design and programming skills, the focus at GDL shifted to instructional activities that targeted teaching students how to solve complex problems. The purpose here was to introduce the students to important cognitive skills required to solve complex problems through meaningful and engaging game-design tasks. Situating problem-solving within the game-design process helped make an unappealing task more appealing. Through problem-solving activities, students not only had chances to practice their general and specific problem-solving skills, but also became familiar with important (basic and metacognitive) thinking skills necessary to solve complex problems.

Similar to game-design activities, problem-solving activities were composed of multiple smaller tasks. The general structure of the problem-solving activities followed the ensuing sequence: students (a) faced complex problem scenarios, (b) solved these problems with instructor/peer guidance, and (c) recreated (reverse-engineered) the problem scenarios in Kodu as simulations.

Example An example problem-solving activity was "SimSchool." The first step of this activity involved introducing students to a problem with a simple scenario (Fig. 4):

After reading the note, the students received data and graphs regarding the issue. By analyzing the data, students had a chance to practice data literacy skills. As it can be seen in Fig. 5, the basic relationship among the variables in the scenario was similar to a predator–prey relationship: student and service staff numbers had an inverse relationship with the amount of trash.


TASK

There is a trash (environmental pollution) problem at your school affecting everyone's health. The school administration is trying to solve this problem, but they need your help. You decide to help them. To do this, in this task, you need to, first, analyze some data to understand the source of the problem, and then devise a solution. Finally, you need to recreate your solution in Kodu to show the school administration the source of the problem, and how it can be solved.

TOP SECRET

Fig. 4 An example problem-solving scenario students received during the GDL program

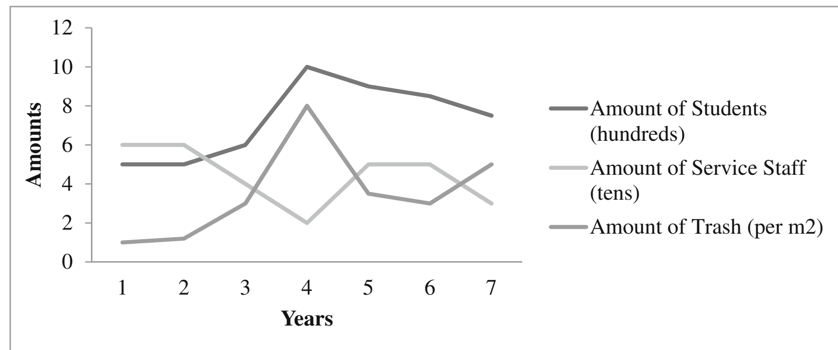


Fig. 5 SimSchool problem scenario – visualized data source

It was at this first step, through instructor guidance, that the students were also introduced to basics skills underlying problem-solving. Based on the method of *teaching basic thinking skills directly*, during this step the students received instruction on how-to solve problems following four basic steps (Polya 1957): (a) understand the problem, (b) devise a plan, (c) carry out the plan, and (d) look back. In their first problem-solving task (SimSchool), for example, the students were especially encouraged to take their time, look at the patterns in the data to *understand* the source of the problem. During this step, instructors provided support by walking around and making sure that everyone could follow necessary steps to understand the problem.

After understanding the problem, in the next step, the students were asked to *plan* a simulation of the scenario to replicate the problem. During this planning stage, the students were encouraged to create flowcharts of their simulations, to help them visualize the system behind their solutions. Having planned a solution, students worked on recreating a simulation of the scenario in Kodu. This design step was important because it allowed the students to *execute* their plans, and see if their solution worked. During the problem-solving activities, the students practiced important metacognitive skills (e.g., Polya's steps of problem-solving) and also engaged in hands-on complex problem solving.

Rationale During problem-solving activities, in addition to teaching basic skills, other effective methods of teaching problem-solving were also used. One such method was *using analogies*. In analogies, solutions for existing problems can be employed to solve new problems with surface differences, but structural similarities (Gick and Holyoak 1980; Mayer and Wittrock 2006). During the GDL program, analogies were provided through introducing scenarios that were different on the surface, but essentially built on same underlying principles. For example, once students solved the SimSchool scenario, in the next session they were given another scenario called Kodu "EcoSystem." In the Ecosystem scenario, the relationships between the problem components were similar to those in SimSchool: three living organisms have a predator–prey relationship where their population numbers depend on each other, and the goal is to maintain order. Using analogous problems, and pointing the similarities between the problems openly to the students, students participating in the GDL program had experience in identifying patterns in problems.

In addition to analogies, from a larger perspective, by using Kodu to recreate problem scenarios, the problem-solving activities in the GDL program can also be considered as digital versions of *structure-based methods*. In structure-based methods, teachers use concrete objects (e.g., beads and sticks) to teach abstract rules (e.g., simple computation problems) (Mayer and Wittrock 1996). During the GDL program, simulations that students created serve as external representations of problems, and helped them understand the abstract rules behind complex problems.

During the process of recreating problem scenarios in Kodu, *generative methods* of teaching problem-solving were also implemented. Generative methods are designed to get students to generate relationships between their own experiences and the target information during learning activities (Mayer and Wittrock 1996). During the GDL program, by selecting scenarios from students' daily life experiences (e.g., SimSchool), and by letting students recreate these scenarios in worlds that they imagined, generative methods were implemented.

Troubleshooting Activities

Purpose and Structure As a stand-alone activity, *troubleshooting* was only offered in the later phases of the GDL courses. Depending on availability of time, troubleshooting activities sometimes were not offered at all. The goal of these activities was to give students structured practice opportunities in troubleshooting and to develop their confidence in troubleshooting. Through troubleshooting activities the students also got a chance to see the importance of troubleshooting during the game-design process, and design tasks in general.

Example An example troubleshooting activity was to fix a game that was intentionally broken by the instructors. The games were picked from games the students were familiar

with, such as, Frogger (created in Kodu). Upon receiving the game, the students were asked to play the game and analyze it to find the flawed or missing codes, structures, or game elements. The students were reminded to explore the game to find the source of each problem, one by one, and fix them one by one. In Frogger, students were encouraged to initially focus on fixing the code that controlled the main character. To fix this issue, the students needed to add code to their character so that can be controlled using the keyboard. Once the character is controllable, students realized the cars in the game move too fast, making it impossible to win the game. Upon identifying the problem, their task was to find the line of code controlling the cars and to adjust speed parameters so that cars move at an optimum level: not too fast (i.e., impossible to win) or not too slow (i.e., too easy game).

Rationale During troubleshooting activities, due to the immediate feedback provided by the game-design software (i.e., changes to codes can easily be tested by running the game), students had opportunities to practice their troubleshooting skills in real time. Such troubleshooting activities were important in helping students build both the skill set necessary for troubleshooting, and confidence in their ability thanks to being in a safe, sandbox-like environment (Papert 1980). As noted previously, dedicated troubleshooting activities were sometimes not offered during the GDL program, because troubleshooting is a natural part of game-design and occurs naturally during other activities. When time allowed, however, instructional time was devoted to these separate troubleshooting activities, because during such dedicated opportunities students was able to focus their attention to the troubleshooting process, and instructors were able to make sure the students understood the procedures and skills underlying the task.

Free Design Activities

Purpose and Structure In the final phases of the GDL courses, even if it is for just one session, the students were given chances to work on creating their own games. Having learned how to use the game-design software, basics of game-design, programming, and problem-solving during the earlier phases of the course, at this final stage students got a final chance to create personally and socially meaningful artifacts, in the spirit of constructionism (Papert 1980).

Example Each student, based on their personal preferences and the amount of technical skills they gained during the GDL program, chose to work on creating a game of their choice. For example, a student who was an avid player of World of Warcraft (WoW) and was comfortable with game-design and programming concepts decided to build one of the WoW storylines within Kodu. Similar to the actual game, the game included three types of playable characters: a magician, a healer, and a warrior. Moving along a path, the team of three players supported each other toward reaching the final boss and beating it. Similarly, another student who played Grand Theft Auto (GTA) opted to create a game similar to the GTA where a character roamed the streets of a city and engaged with multiple independent tasks. Finally, there were students who chose to develop less complex games. In such cases, students picked a game that they had been working on during the earlier stages of GDL and added more rules and challenges to the game. For example, one student developed a two-level game out of the broken game given during the troubleshooting activities. After fixing Frogger, he continued to add another mini-game that players would access only after successfully completing Frogger.

Rationale Free game-design activities served as an important method to check student understanding, because at this final stage the students had a chance test the boundaries of their skills as well as the game-design software. During activity students utilized important design skills such as problem finding and troubleshooting. For example, it was very common to find a student trying to create something that was not readily available in Kodu. In such cases, for example creating teleportation gates out of hockey pucks, students started thinking of alternative ways to overcome problems that lacked an obvious solution. Such opportunities were important because they helped the students build confidence in their game-design, programming, and problem-solving skills. What was especially unique about the GDL program was that the important design and problem-solving skills were utilized through engaging game-design tasks and in the low-stakes and safe atmosphere of the GDL program.

Free design activities were also important because they were highly personalized, as students chose and worked on design ideas that they personally valued. Being invested in a personally meaningful problem helped them persist in the task even in the face of difficulties. The GDL program culminated with student presentations of the games they have developed.

Discussion and Conclusions

In this paper, the design, development, and implementation of a technology-rich learning environment designed to teach young students game-design, programming, and problem-solving skills through engaging game-design activities was detailed. As emphasized, creating such instructional activities requires finding a balance among technology, pedagogy, and content. Having multiple goals during the GDL program required design of a multi-faceted intervention, incorporating multiple instructional techniques, and activities with different objectives.

Although the design process detailed here is specific to the GDL program where the goal was to teach students problem-solving through game-design tasks, the design process detailed here can serve as an example for other contexts where different types of tasks (e.g., app-making) are used to teach different types of thinking skills (e.g., computational thinking). In this regard, the GDL context can be considered as an example case of effective technology integration, where technology was used to solve an educational problem (i.e., teaching problem-solving in an engaging context). Due to the special attention given to natural interplay among technology, pedagogy, and content, the approach to technology integration at GDL was in line with the Technological Pedagogical Content Knowledge (TPACK) framework (Mishra and Koehler 2006). According to TPACK, effective technology integration is possible by first taking into account the complexity of teaching, and then understanding the unique and permutable nature of the interplay between technology, pedagogy, and content. Accordingly, GDL courses were designed to utilize the affordances of available technologies (i.e., game-design software) while satisfying pedagogical (i.e., constructionism), and the content needs (e.g., game-design and programming concepts).

Reporting on detailed reports on cognitive outcomes of the GDL program was beyond the scope of the current paper and such accounts can be found elsewhere (Akcaoglu 2014; Akcaoglu and Koehler 2014). It should be noted, however, that the students who attended the GDL program showed significant improvement in their problem-solving skills (Akcaoglu 2014; Akcaoglu and Koehler 2014). Findings from the research on the effectiveness of GDL speak to the success of the program in reaching its intended goals, as well as the effectiveness of the instructional design process employed.

The success of the GDL implementations also paves the way for future innovations. For example, just as effectively as embedding of thinking skills, content knowledge can also be embedded in GDL program. For example, the scenarios given to students can be changed to address important environmental problems, shifting the focus to teaching environmental literacy or ecological awareness. Future research can investigate this connection and evaluate the potential of such intervention in teaching content skills, raise awareness, and teach thinking skills simultaneously. Such research can possibly also look at the changes in students' interest and utility value of careers in science (or STEM) fields.

Recently, game-design has been championed as an alternative context to teach programming and thinking skills (Denner et al. 2012; Ke 2014; Li 2010; Weintrop and Wilensky 2012) and as a method of encouraging STEM careers. With the increasing interest toward using game-design for instructional purposes, there is an increasing need to develop curricula to harness the full potential of these activities, as well as a need to engage in research to understand the outcomes from such work. The teaching context detailed here is an attempt to pave the way for such future efforts.

References

Ackermann, E. (2001). Piaget's constructivism, Papert's constructionism: What's the difference? *Future of Learning Group Publication, 5*(3), 1–11.

Akcaoglu M. (2014) Learning problem-solving through making games at the game design and learning summer program. *Educational Technology Research and Development, 62*(5), 583-600. doi:10.1007/s11423-014-9347-4.

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.

Baytak, A., & Land, S. M. (2010). A case study of educational game design by kids and for kids. *Procedia - Social and Behavioral Sciences, 2*(2), 5242–5246. doi:10.1016/j.sbspro.2010.03.853.

Bonnardel, N., & Zenasni, F. (2010). The impact of technology on creativity in design: an enhancement? *Creativity and Innovation Management, 19*(2), 180–191. doi:10.1111/j.1467-8691.2010.00560.x.

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.

Eseryel, D., Ifenthaler, D., & Ge, X. (2013). Towards innovation in complex problem solving research: an introduction to the special issue. *Educational Technology Research and Development, 61*(3), 359–363. doi:10.1007/s11423-013-9299-0.

Fowler, A., & Cusack, B. (2011). Enhancing introductory programming with Kodu Game Lab: An exploratory study. In M. Lopez & M. Verhaart (Eds.), *Proceedings from 2nd Annual Conference of Computing and Information Technology Research and Education New Zealand (CITRENZ2011)* (pp. 69–79). New Zealand: Christchurch.

Fullerton, T. (2008). *Game design workshop*. Boston, MA: Elsevier.

Funke, J. (2010). Complex problem solving: a case for complex cognition? *Cognitive Processing, 11*(2), 133–142.

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

Gick, M. L., & Holyoak, K. J. (1980). Analogical problem solving. *Cognitive Psychology, 12*(3), 306–355.

Goel, V., & Pirolli, P. (1992). The structure of design problem spaces. *Cognitive Science, 16*(3), 395–429.

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

Hwang, G.-J., Hung, C.-M., & Chen, N.-S. (2013). Improving learning achievements, motivations and problem-solving skills through a peer assessment-based game development approach. *Educational Technology Research and Development*. doi:10.1007/s11423-013-9320-7.

Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development, 48*(4), 63–85.

Jonassen, D. H. (2011). *Learning to solve problems: A handbook for designing problem-solving learning environments*. New York: Routledge.

Kafai, Y. B. (1995). *Minds in play: Computer game design as a context for children's learning*. Hillsdale: Lawrence Erlbaum.

Ke, F. (2014). An implementation of design-based learning through creating educational computer games: a case study on mathematics

learning during design and computing. *Computers & Education, 73*(1), 26–39. doi:10.1016/j.compedu.2013.12.010.

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist, 41*(2), 75–86.

Koehler, M. J., & Mishra, P. (2005). What happens when teachers design educational technology? The development of technological pedagogical content knowledge. *Journal of Educational Computing Research, 32*(2), 131–152. doi:10.2190/0EW7-01WB-BKHL-QDYV.

Koehler, M. J., & Mishra, P. (2009). What is technological pedagogical content knowledge (TPACK)? *Contemporary Issues in Technology and Teacher Education, 9*(1), 60–70.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research, 2*(4), 429–458.

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

MacLaurin, M. B. (2011). The design of Kodu: a tiny visual programming language for children on the Xbox 360. *ACM SIGPLAN Notices, 46*(1), 241–246.

Mayer, R. E. (1977). *Thinking and problem solving: An introduction to human cognition and learning*. Glenview, Illinois: Scott, Foresman and Company.

Mayer, R. E. (1998). Cognitive, metacognitive, and motivational aspects of problem solving. *Instructional Science, 26*(1), 49–63.

Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? The case for guided methods of instruction. *The American Psychologist, 59*(1), 14–19. doi:10.1037/0003-066X.59.1.14.

Mayer, R. E., & Wittrock, M. C. (1996). Problem-solving transfer. In D. C. Berliner & R. C. Calfee (Eds.), *Handbook of educational psychology* (pp. 47–62). New York, NY: Macmillan Library Reference.

Mayer, R. E., & Wittrock, M. C. (2006). Problem solving. In P. A. Alexander & P. H. Winne (Eds.), *Handbook of educational psychology* (pp. 287–303). Mahwah, NJ: Lawrence Erlbaum Associates.

Microsoft Kodu. (2012). Microsoft Research. Retrieved from http://research.microsoft.com/en-us/projects/kodu/.

Mishra, P., & Koehler, M. J. (2006). Technological pedagogical content knowledge: a framework for teacher knowledge. *Teachers College Record, 108*(6), 1017–1054. doi:10.1111/j.1467-9620.2006.00684.x.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic Books, Inc.

Papert, S., & Harel, I. (1991). Situating constructionism. In S. Papert & I. Harel (Eds.), *Constructionism* (Vol. 36, pp. 1–11). Norwood, NJ: Ablex Publishing Corporation.

Polya, G. (1957). *How to solve it*. Garden City, NY: Doubleday/Anchor.

Prensky, M. (2003). Digital game-based learning. *Computers in Entertainment (CIE)*. Retrieved from http://dl.acm.org/citation.cfm?id=950596.

Resnick, L. (1987). The 1987 presidential address: learning in school and out. *Educational Researcher, 16*(9), 13–20.

Robertson, J. (2012). Making games in the classroom: benefits and gender concerns. *Computers & Education, 59*(2), 385–398. doi:10.1016/j.compedu.2011.12.020.

Salomon, G., & Perkins, D. N. (1987). Transfer of cognitive skills from programming: when and how? *Journal of Educational Computing Research, 3*(2), 149–169.

Scratch. (2012). Retrieved from http://scratch.mit.edu/.

Simon, H. A. (1995). Problem forming, problem finding, and problem solving in design. In A. Collen & W. W. Gasparski (Eds.), *Design and systems: General applications of methodology (Vol. 3)ems* (Vol. 3, pp. 245–257). Brunswick, NJ: Transaction Publishers.

Smith, K., & Boling, E. (2009). What do we make of design? Design as a concept in educational technology. *Educational Technology, 49*(4), 3–17.

Stolee, K. T., & Fristoe, T. (2011). Expressing computer science concepts through Kodu Game Lab. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 99–104).

Touretzky, D. S., Marghitu, D., Ludi, S., Bernstein, D., & Ni, L. (2013). Accelerating K-12 computational thinking using scaffolding, staging, and abstraction. In *Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 609–614). ACM.

Weintrop, D., & Wilensky, U. (2012). RoboBuilder: Video game program-to-play constructionist. In *Constructionism 2012* (pp. 1–5). Athens, Greece.