Measuring Gameplay Affordances of User-Generated Content in an Educational Game

Drew Hicks
North Carolina State University
911 Oval Drive
Raleigh, NC 27606
aghicks3@ncsu.edu

Zhongxiu Liu
North Carolina State University
911 Oval Drive
Raleigh, NC 27606
zliu24@ncsu.edu

Tiffany Barnes
North Carolina State University
911 Oval Drive
tmbarnes@ncsu.edu

ABSTRACT

Level creation is a creative game-play exercise that resembles problem-posing, and have shown to be engaging and helpful for players to learn about the game's core mechanic. However, in user-authoring environments, users often create levels without considering the game's objective, or with entirely different objectives in mind, resulting in levels which fail to afford the core gameplay mechanic. This poses a bigger threat to educational games, because the core gameplay is aligned with the learning objectives. Therefore, such levels fail to provide any opportunity for players to practice the skills the game is designed to teach. To address this problem, we designed and compared three versions of level creators in a programming game - BOTS. These level creators differ in their unit of construction - terrain-block, program and Pattern-block. We measured levels' gameplay affordances as their relevance to the core game mechanic and learning objective: correct use of programming concepts such as loops, functions and subroutines, and used a zero-inflation model alongside expert tagging of created levels to analyze the differences between the three different editors. The results show that a simple-to-use building-block editor is more likely to guarantee levels that contain some affordances, but a more complex editor designed to use the same core mechanic as gameplay results in the best-quality levels in the best case.

Keywords

User-created Content, Educational Game, Educational Data Mining, Survival Analysis, Complex Problem Solving, Learning Analytics

1. INTRODUCTION

In previous work with our programming game, BOTS, we demonstrated that user-created levels in our game frequently contain appropriate gameplay affordances, which reward specific, desired patterns of gameplay related to the game's learning objectives. Such levels demonstrate the creator's understanding of those learning objectives, and offer other players opportunity to practice using those concepts. However, alongside these high-quality submissions there also exist various negative patterns of user-generated content, specifically sandbox, griefer, Power-Gamer, and trivial levels which ignore or replace the game's core learning objectives and challenges. In order to implement user-created levels into the game itself, an additional filtering and evaluation step is needed to identify and remove these low-quality submission. Our initial attempt at filtering these levels, a "Solve and Submit" procedure, was effective at reducing the number of these types of levels which were published, and additionally was somewhat effective at reducing the number of these levels created to begin with; however, some users created fewer levels under this condition, indicating that the barrier after level creation discouraged further creation. Our next step is to make further improvements to the content authoring tools in order to increase the overall quality of submitted content. In order to do so, we will investigate three versions of the game's level editor. The initial, free-form editor, and two constrained editors employing different types of constraints. Previous work has shown that players are engaged when constraints are posed that are restrictive enough to encourage demonstration of the game's target learning concepts, but not so restrictive as to require them, lest players feel as though they are unable to create what they want to create. We propose to evaluate level editors with two different forms of constraint added. The Programming Editor, where the length (in lines of code) of the solution is constrained, similarly to the Point Value Showcase in Bead Loom Game. Second, where the construction of the level itself is constrained by providing authors with a limited selection of "Building Blocks". For this work, we hope to answer (or gain insight into) the question: Does providing game-like scaffolding in the form of objectives and points related to elements of high-quality content result in users authoring higher quality content?

2. BACKGROUND

User-generated content has been revolutionizing gaming, and the potential applications in educational games are intriguing. Commercial games such as Super Mario Maker[18] and Little Big Planet[17] rely almost entirely on user-submitted levels to provide an extendible gameplay experience, with the creation process itself serving as the meat of the built-in gameplay. Creative gameplay avoids many of the mo-

tivational pitfalls of educational games, such as relying on competitive motivators, that may make the intervention less successful for non-males, who may have a more social orientation towards gameplay, or may have less experience with traditional video games [12, 13, 5].

Creating exercises has long been used as an educational activity in the form of problem-posing in many STEM domains. In Mathematics in particular, Problem-posing has been promoted as a classroom activity and as an effective assessment of student knowledge [21, 7] Games and ITSs such as AnimalWatch[4] and MONSAKUN[15] have users creating exercises for from expert-selected "ingredients." Work with systems such as "MONSAKUN", "AnimalWatch" and the Peer-to-peer learning community "Teach Ourselves" has shown that systems which facilitate problem creation by students can provide benefits beyond those of systems without this feature.

MONSAKUN [15] is a system which facilitates problem-posing for elemntary arithmetic problems. The authors wanted to influence students to produce word problems whose structure was different from the structure of the mathematical solution. In order to build the word problem, students are given segments of a word problem such as "Tom has 3 erasers" or "Tom buys several pencils" which they arrange in order to construct their problem.

Animal Watch [1, 4] is a pre-algebra tutor which uses data about exotic animals as the theme for the problems presented. The tutor covers topics such as finding average median and mode, converting to different units, and so on. While the tutor contains around 1000 problems authored by the developers, the authors of this paper noted that even with a large number of problems the system can "run out" of appropriate problems to give a student. The pilot mostly investigated student attitudes towards problem posing, finding that students were excited about sharing content with their peers, and proud that content they had created would be online and accessible to others. At the same time, students reported a low self-assessment of learning, and felt that it was easy once they got started.

Later work by Carole Beal involving a system called "Teach Ourselves" investigated these effects further [3], incorporating aspects of gamification. Players earn rewards for solving and creating that are displayed on a leaderboard, and can get "+1" from peers for creating good content in the form of problems and hints. Problems created by students were of usable quality, with an average quality score of 7.5/12 on a scale developed by the system's designers. Teachers who used the system observed increased motivation in their students, and believed that the system encouraged higher-order thinking. Again there is no deeper evaluation comparing learning between students who made problems and students who did not. Even very simple problem-posing interventions have been shown to be effective. In Chang's work with a problem-posing system to teach mathematics, it was demonstrated that when the posed problems were to be used as content for a simple quiz-show-like game, low performing students experienced significantly greater learning gains from the activity, and students reported being more engaged with the activity [8].

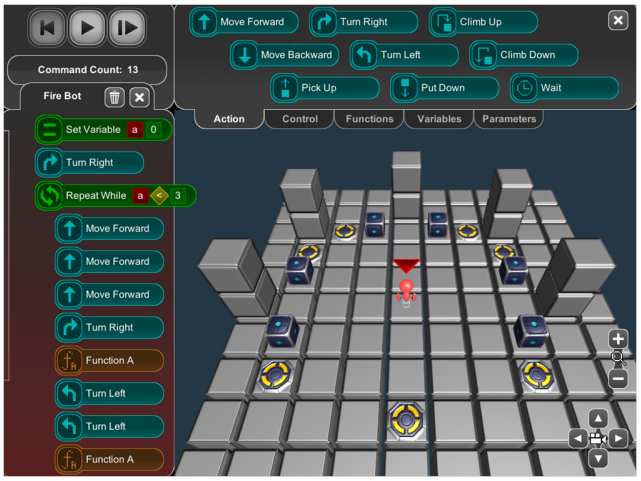

Figure 1: Gameplay screenshot from the BOTS game showing a complex puzzle and partial solution.

3. DESCRIPTION OF BOTS

BOTS (bots.game2learn.com) is a puzzle game designed to teach fundamental ideas of programming and problem-solving to novice computer users. BOTS was inspired by games such as LightBot and RoboRally, as well as the syntax of Scratch and Snap [9, 10, 23]. In BOTS, players take on the role of programmers writing code to navigate a simple robot around a grid-based 3D environment. The goal of each puzzle is to press several switches within the environment, which can be done by placing an object (or the robot itself) on top of them. Within each puzzle, playersâĂŹ scores depend on the number of commands used, with lower scores being preferable. For example, in the first tutorial level, a user could solve the puzzle by using the "Move Forward" instruction 10 times. This is the best score possible without using loops or functions. Therefore, if a player wants to make the robot walk down a long hallway, it will be more efficient to use a loop to repeat a single "Move Forward" instruction, rather than to simply use several "Move Forward" instructions one after the other. These constraints, based on the Deep Gamification framework, are meant to encourage players to optimize their solutions by practicing loops and functions.

Previous work with BOTS focused on how to restrict players from constructing negative design patterns in their levels [14], and how to automatically generate low-level feedback and hints for user-generated levels without human authoring[20]. Our next steps with this game are to further improve the level authoring tools to increase the quality of the levels which don't exhibit these negative design patterns.

3.1 Gameplay Affordances

The term *Affordance* has its origins in psychology, where it is defined by Gibson as "what [something] offers the animal, what it provides and furnishes" [22]. This concept was later introduced to HCI, where Norman defined affordance as "the perceived or actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used" [19]. Norman's definition centers on users' perspectives. If a user does not read an action with an object possible, then the object does not afford that action.

With respect to affordances in games, James Paul Gee wrote that games create a match between affordances and what he calls "effectivities" [11]. In his writing, *effectivities* are defined as the abilities of the player's tools in the game; for example a character in a platforming game may be able to run, climb, and jump. On the other hand, *affordances* describe relationships between the world and actors, or between tools and actors. Other work taxonomizing level design patterns in video games also referred to the desired gameplay produced by these types of structures. For example, in Hullet and Whitehead's work with design patterns in single-player First-person shooter (FPS) levels, the Sniper Location design pattern is a difficult to reach location with a good view of the play area, occupied by an enemy [16]. This pattern is described as forcing the player to take cover. The presence of other gameplay elements such as Vehicles and Turrets herald similar gameplay changes [2].

In BOTS, the primary educational goal is to teach students basic problem solving and programming concepts such as using functions and loops to handle repetitive patterns. Students (who see robot as tool) should look at puzzles as opportunities for optimization with loops and functions. Thus, affordances in bots means objects or patterns of objects should signal the presence of these opportunities to students.

Though the objects in BOTS signal gameplay patterns, players building levels in BOTS frequently place them in misleading or irrelevant ways, where the gameplay decisions informed do not lead to a correct or successful solution. For example, a player can put a crate which communicates the affordance of "pick up" action. However, when the puzzle contains no circle to put the crate on, the affordance of the crate is meaningless. Thus, instead of single affordance on object, our primary focus is on subsets of affordances that improved outcomes in terms of the core mechanism of the game - problem solving and solution optimization. These subsets of affordances is referred as gameplay affordance in remaining sections.

3.2 Level Editors

Specific discussion of the design principles behind the two level editors used for this study can be found in our previous work, (cite GLS paper). For the sake of space, we will only generally discuss those design principles here, instead focusing on the tools available to users in the two designs.

In all versions of the level editor, levels consist of a 10x10x10 grid, where each grid square can be populated by a terrain block or an object. Levels must contain at minimum a start point and goal, and can optionally contain additional goals which must be covered with movable boxes before the level will be completed.

In the Free-Form Drag-and-Drop editor, players will be asked to create a level in a Free-Form editor which uses controls analogous to Minecraft. Players can click anywhere in the world to create terrain blocks, and can select objects from a menu such as boxes, start points, and goals, to populate the level with objectives. At any point during creation, the player can save the level (which must, at minimum, contain a start point and a goal.) The player must then complete the level on their own before the level is published and available to other users.

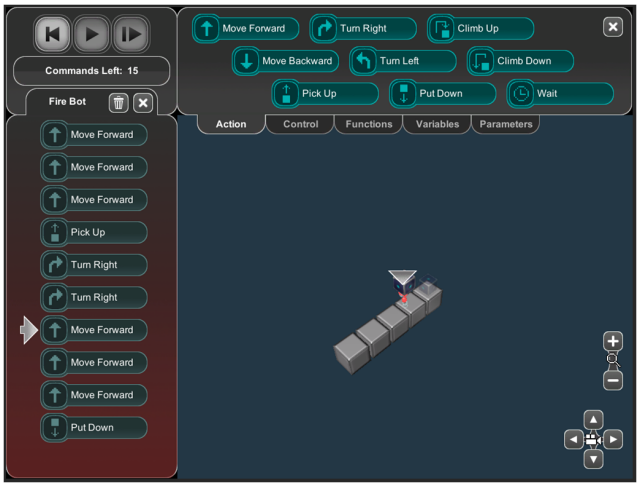

Figure 2: The Programming editor interface.

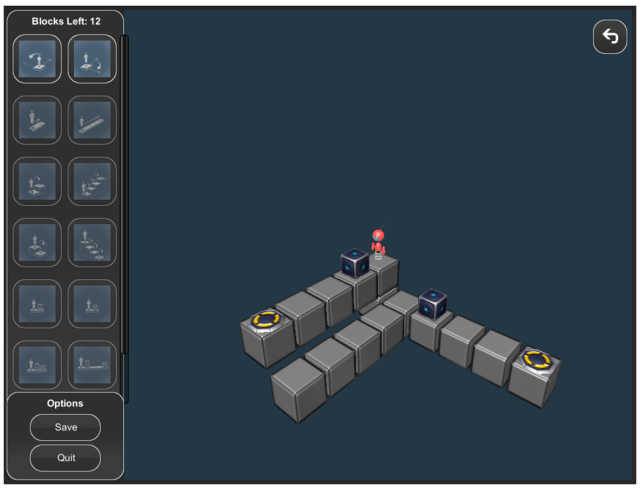

Figure 3: The Building-Block editor interface.

In the Programming Editor (inspired by the Deep Gamification based level editor in BeadLoom Game [6]) players will be asked to create a level by programming the path the robot will take. Players will be constrained to using a limited number of instructions. This is analogous to the level creation tools in BeadLoom Game where players created levels for various "showcases" under similar constraints. This type of constraint has been shown to be effective for encouraging players to perform more complex operations in order to generate larger more interesting levels under the constraints. One challenge with this approach is that since simple solutions are still permitted, and most programs are syntactically correct, users who are experimenting with the level creation interface with no goal in mind may create levels that they themselves do not understand.

In the Building-Block editor, we constrain level creation by providing meaningful chunks to authors in the form of "Building Blocks." This is inspired by problem-posing activities as presented in systems like MONSAKUN [15] and

AnimalWatch [1, 4] in which players are asked to build a problem using data and problem pieces provided by experts. In this version of the level editor, players will be asked to create a level only using our "Building Blocks" which are pre-constructed chunks of levels. These "Building Blocks" will be partial or complete examples of the patterns identified in Chapter 4, specific structures which correspond to opportunities to use loops, functions, or variables. We hypothesize that this may lead to better levels because it explicitly promotes the inclusion of these patterns, which will lead to opportunities for players to use more complex programming constructs like loops and functions. We also believe that this will encourage students to think about optimizing the solution to the level while they are making the level. One potential challenge with this approach is that students may find these constraints too restrictive, which might reduce engagement for creatively-oriented players [6]. By evaluating these two versions of a gamified level editor against each other, we will determine which practices best suit our game. In particular, which version of the activity leads to the production of better content for future users.

4. DATA

This paper reports gameplay data from 175 students (57 in the Free-Form condition, 46 in condition 1, 72 in condition 3) across all classes/workshops that used the BOTS game as part of their activities. In total, 229 levels were created by these players (82 / 54 / 93), 175 (49 / 33 / 93) of which were published and made public.

Data was collected in 90 minute sessions, in which all students followed the same procedure. First, each student created a unique account in the online version of the game. Players then completed the Tutorial up to the final challenge level which functions as sort of a "collector" stage; Players aren't expected to complete this level with optimum score, but exploring this level allows faster students to continue practicing while the rest of the class catches up. During the tutorial segment, instructors were told to prompt players to reread the offered hints for their current level carefully, if they became stuck, and only to offer more guidance after the player had carefully read the instructions. This part of gameplay took 45 minutes.

For the remaining 45 minutes, students were instructed to build at least one level in their version of the level editor interface. After collecting this level, players could continue creating levels, or could play levels created by their peers.

The way the level editor was selected varied per data collection. In the first data collections, all students used the "Free-Form" level editor; this editor was also used in previous work on BOTS. One subsequent data collection used only the "Programming" level editor; this data was used to evaluate the interface design of that editor. In the remaining data collections, students were randomly assigned an interface between the "Programming" editor and the "Building-Block" editor. Stages from all data collections are used in this analysis.

To analyze the differences between created levels, we played each level to find the shortest-path solution from start to goal, and used a solver to find the shortest program to produce this optimal solution. The algorithm used by the solver is relatively simple: First, a program that recreates the shortest-path using only simple commands is constructed. Then, sets of repeated commands are identified in this program by treating the commands as words and identifying repeated n-grams. Then, recursively, each possible combination of optimization on these n-grams is applied: either replacing the -gram with a subroutine identifier wherever it appears, or replacing adjacent -grams with a single instance of that -gram, wrapped in loop commands. After each step, the program is re-evaluated, until the shortest, most optimal version of the solution is found. The shortest-path solution itself is the *naive solution* which uses only simple commands such as moving and turning. The optimized shortest-path solution is the *expert solution* which uses loops and subroutines to optimize the shortest path solution. The difference between these solutions, in terms of lines of code, is used as a measurement of how well the level affords the use of those game mechanics.

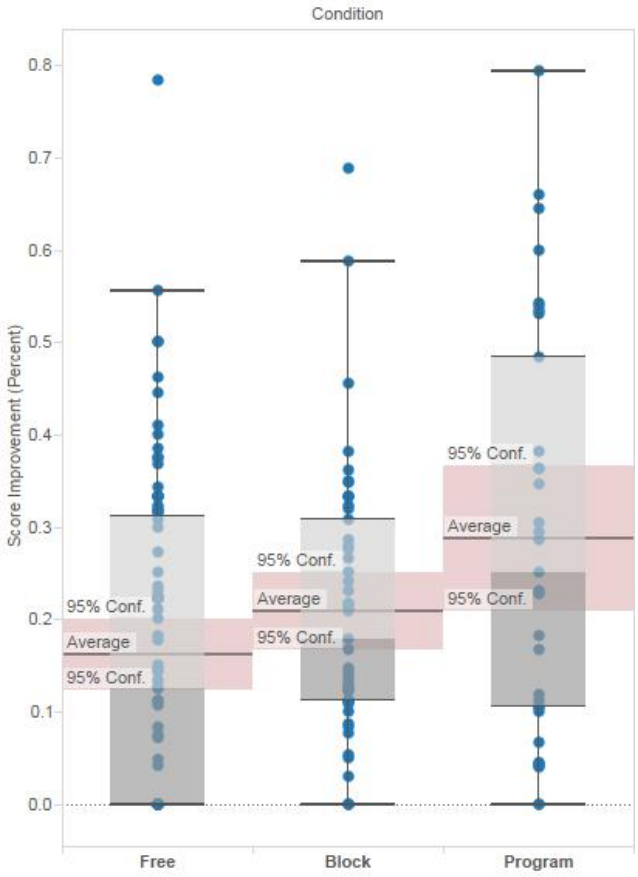

Figure 4: Plot comparing the distribution of levels between the three conditions. Each point in this plot represents the difference in number of commands between a naive and expert solution, as a percentage of the expert solution.

5. METHODS AND RESULTS

In this section, we describe our analyses, both to identify any differences in the presence of gameplay affordances, and to identify differences in how experts tagged the created levels across conditions.

5.1 Direct Measurement of Improvement

To evaluate the differences between levels on a direct measure of improvement (the difference in length between a naive solution and an expert solution) we employed a Zero-Inflation model. This model is a hierarchical model used when a zero result is thought to have different implications than a small, but non-zero result. In this case, a level with zero improvement contains no affordances, while a level with only a small improvement may still contain affordances that, though present, are less rewarding to the player.

Table 1: Pearson Residuals

Min	1Q	Median	3Q	Max
-1.7873	-1.1181	-0.4330	0.7073	8.0089

Table 2: Poisson model coefficients, comparing the two editors to the baseline, Drag-and-Drop editor

	Estimate	Std. Error	z	p
(Intercept)	1.87410	0.05037	37.207	$< 2e\text{-}16$
Program	0.38704	0.07318	5.289	1.23e-07
Block	0.03060	0.07400	0.413	0.679

Table 3: Binomial model coefficients, comparing the two editors to the baseline, Drag-and-Drop editor

	Estimate	Std. Error	z	p
(Intercept)	-1.63502	0.31821	-5.138	2.77e-07
Program	-0.03042	0.52083	-0.058	0.95343
Block	1.06715	0.39444	2.705	0.00682

Presented here are the results of fitting a Zero-Inflated Poisson model on our data. We can look at the two tests separately: the binomial model relates to whether a level will have zero or non-zero results, and the Poisson model relates to the size of the non-zero results. From the binomial model, we can see that the Building-Block editor is more likely than the other conditions to produce a non-zero result for Difference. This makes sense because, of the Building Blocks available to students, only the very simplest ones offer no affordances, and in fact, the blocks are built out of instances where previous levels contained affordances. So in order to construct a zero-valued level, a Building-Blocks student would need to use only the simplest blocks, though indeed this appears to have been the case in several of the constructed stages.

Looking at the Poisson model, we see that considering the non-zero results, the Programming editor is likely to have a higher value of Difference than either other condition. In the Building-Blocks editor, each block contains only a small affordance since the blocks themselves are only 3 to 4 commands long. If blocks are not repeated, this pattern will persist in the repeated level. However, in the Programming editor, we observed players exploring more, wrapping code in functions and loops to see what would happen, and changing their code until the level looked how they wanted it to look. Levels generated in this manner will have much larger differences between the naive solutions and expert solutions, than levels generated from multiple unique Building Blocks.

Using a zero-inflated Poisson distribution model, we were able to examine the differences between levels created under our various conditions. We used this zero-inflation model because the model looks for two separate effects: first, the effect that causes the DV to be zero or non-zero, and second, the effect that causes the value of the DV to change in the non-zero cases. This is important because the structural elements for levels with zero affordance for advanced game mechanics are very different from those with only a small affordance. Zero-affordance levels tend to be trivially short or entirely devoid of patterns, while small-affordance levels may contain patterns but with small changes between them which limit how advanced game mechanics may be used to optimize the solutions.

To summarize these results, by using this model, we were able to observe the following effects. First, the Building Block editor is most likely to produce a non-zero result, statistically significantly more likely than either other condition. Second, the Programming editor is likely to have a higher-value of difference for the non-zero results that are created.

5.2 Expert Tagging

We compared puzzles across three versions of level editors, with the hypothesis that the more meaningful the level editor's construction unit, the higher quality the puzzles. Here, we assume that "Building Blocks" from version 3 and programs from version 2 are more meaningful than terrain blocks in version 0. We also hypothesize that the Programming editor will result in more reusable puzzles from a player perspective, and that the Building Block editor is more likely to encourage loops and functions.

We used an expert, blind to which editor was used to create the puzzle, to identify the presence or absence of negative design patterns. We used the defined puzzle design patterns as identified in our previous work: "Normal" levels which contained few (or no) negative design patterns, and four categories of levels characterized by specific negative design patterns: *Griefer, Power-Gamer, Sandbox, and Trivial* levels.

We measured a puzzle's quality based on previously identified patterns of negative content, which were used as tags for this study. The following criteria were used to assign tags:

- a) it is readily apparent that a solution is possible
- b) a solution actually is possible
- c) the solution can be improved with loops or functions
- d) patterns in the level design call out where loops or functions can be used
- e) the expert solution can be entered in reasonable time

- f) the naive solution can be entered in reasonable time

We decided on these criteria because the pedagogical goal of BOTS is to teach students basic problem solving and programming skills. Thus, a good quality puzzle should help players focus on the problem, and should encourage the use of programming concepts such as flow control and function. Levels which are impossible, or simply far too tedious, are among the most common negative traits identified in previous designs, so updated versions of the level editor specifically addressed these two criteria via hard constraints on the placements of goals and size of levels.

Table 4: Categories of Puzzles Created by Three Versions of Level Editors

	D&D	Program	Block
normal	66	43	67
Power-Gamer	9	1	13
griefer	2	0	0
sandbox	10	0	0
trivial	5	8	2
TOTAL	92	52	82

Table 4 reports the number of puzzles in each category, created by the three level editors. Fisher's Exact Test showed a significant difference in the category distributions between each pair of the three level editors (Building-Block vs. Programming: $0<0.001$; Building-Block vs. drag-and-drop: $p=0.0032$; Programming vs. Drag-and-Drop: $p=0.0035$).

The Programming editor has the highest proportion of Normal puzzles (82.7%). Moreover, the Pattern-block and Terrain-block editors created a higher proportion of Power-Gamer levels compared with the Programming editor. These levels are characterized by extreme length and a high number of objectives. The Drag-and-Drop editor is the only level editor in which users created Sandbox puzzles. Finally, the Programming editor has the highest proportion of Trivial puzzles.

We then compared the quality criteria among puzzles between the three level editors. We again found that the Pattern-block editor has significantly higher proportion of puzzles that can be improved with loops and functions, compared with the Programming editor ($p=0.0079$) and the Terrain-block editor ($p=0.0038$), confirming our results from above.

6. DISCUSSION

The results seem to confirm that the Drag-and-Drop editor is the least likely to result in levels with gameplay affordances for using loops and functions. The Drag-and-Drop editor resulted in the lowest proportion of Normal puzzles, but high proportions of Sandbox puzzles and Power-Gamer puzzles. Additionally, they created fewer puzzles that can be improved by loops or functions, or which have obvious patterns for using loops or functions. Players using this editor are less likely to consider the gameplay affordances of their levels, adding elements regardless of their effect on gameplay. Additionally the Drag-and-Drop editor is the only level editor in which users created Sandbox puzzles. This may be because Sandbox levels are characterized by the presence of extraneous objects, and both new level editors operate by creating the robot's path, so designers would have to deliberately stray from their intended path to place extraneous objects.

On the other hand, the Programming Editor resulted in a high proportion of Normal puzzles and the lowest proportion of Power-Gamer puzzles. This makes sense because a Power-Gamer puzzle is typically a puzzle which takes a short time to create but a long time to complete. Since this editor uses the exact same mechanic for creation as completion, this is quite difficult to do. However, these users also built a lower proportion of puzzles that can be improved with loops and functions than the users of the Building Block editor, and the highest proportion of Trivial puzzles whose solutions are too short to afford the use of loops or functions. This editor is the most complex to use, so players with little patience for learning the interface may create Trivial puzzles. Additionally, trying options at random to see what they do in the programming editor is likely to result in the creation of a Trivial level. We hypothesize that in the other editors, random behavior results in different level types: Power-Gamer levels in the Building Block editor, and Sandbox levels in the Programming editor.

Lastly, the Building-Block Editor has a high proportion of normal puzzles, and the highest proportion of puzzles that can be improved with loops or functions and have obvious patterns for using loops and functions. The building blocks used to create levels are subsections of previously created levels selected specifically because they afford the use of loops or functions. The Pattern-block editor created the highest proportion of Power-Gamer puzzles. This may be because of the ease of use; adding a block takes one click but may require 5-10 commands from the player who later solves the puzzle. We previously observed that players tended to fill the space available to them in the Drag-and-Drop editor, so Pattern-block puzzle creators may also be trying to fill the available space. In both other editors, it takes longer to solve the puzzle than to create it, but the programming editor minimizes this difference, thereby making the creation of Power-gamer levels less likely.

7. CONCLUSIONS AND FUTURE WORK

In conclusion, including Deep Gamification elements in Level Editors (in the form of creative constraints, building blocks, or integration with gameplay mechanic) did result in an overall improvement in level quality. In both the Programming editor and Building-Block editor were more effective than a Drag-and-Drop editor at encouraging the creation of levels which contain gameplay affordances. The Programming editor was most effective at ensuring a non-zero improvement between expert and naive solutions, but perhaps trivially so, as the building blocks themselves were selected as to contain small improvements. The Programming editor is less likely to ensure a non-zero improvements, but levels created under this condition contain larger improvements, which may be more obvious or more rewarding to players than numerous small improvements.

Our next steps are to investigate how players react to levels created under these conditions. We know that these levels contain opportunities for users to practice, but if the users

don't recognize or simply don't take advantage of the opportunities, the improvement is lost. Additionally, we noticed several patterns of negative design that are unique to these new editors, shifting '"Sandbox" design into Power-Gamer or Trivial levels. For the Building Block editor, this seems to be exclusively negative, resulting in overlong, unrewarding levels. In the Programming editor, this behavior sometimes resulted in interesting levels created when the author was experimenting with loops and nested functions rather than creating with an end-goal in mind. This experimental usage of the previous level editor was treated as negative, with the output levels being low-quality. In the Programming editor, that is not always the case, so re-evaluation of how these levels are identified is needed.

8. ACKNOWLEDGMENTS

Thanks to Michael Kingdon, Aaron Quidley, Veronica Catete, Rui Zhi, Yihuan Dong, and all developers who have worked on this project or helped with our outreach activities so far. This project is partially funded under NSF Grant Nos. 00000000

9. REFERENCES

[1] I. Arroyo and B. P. Woolf. Students in awe: changing their role from consumers to producers of its content. In *Proceedings of the 11th International Conference on Artificial Intelligence and Education*. Citeseer, 2003.

[2] D. Bacher. Design patterns in level design: common practices in simulated environment construction. 2008.

[3] C. R. Beal, P. R. Cohen, et al. Teach ourselves: Technology to support problem posing in the stem classroom. *Creative Education*, 3(04):513, 2012.

[4] M. Birch and C. R. Beal. Problem posing in animalwatch: An interactive system for student-authored content. In *FLAIRS Conference*, pages 397–402, 2008.

[5] J. Bourgonjon, M. Valcke, R. Soetaert, and T. Schellens. StudentsâĂŹ perceptions about the use of video games in the classroom. *Computers & Education*, 54(4):1145–1156, 2010.

[6] A. K. Boyce. Deep gamification: Combining game-based and play-based methods. 2014.

[7] J. Cai, J. C. Moyer, N. Wang, S. Hwang, B. Nie, and T. Garber. Mathematical problem posing as a measure of curricular effect on students' learning. *Educational Studies in Mathematics*, 83(1):57–69, 2013.

[8] K.-E. Chang, L.-J. Wu, S.-E. Weng, and Y.-T. Sung. Embedding game-based problem-solving phase into problem-posing system for mathematics learning. *Computers & Education*, 58(2):775–786, 2012.

[9] I. F. de Kereki. Scratch: Applications in computer science 1. In *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*, pages T3B–7. IEEE, 2008.

[10] R. Garfield. Roborally. [Board Game], 1994.

[11] J. P. Gee. Deep learning properties of good digital games: How far can they go. *Serious games: Mechanisms and effects*, pages 67–82, 2009.

[12] B. S. Greenberg, J. Sherry, K. Lachlan, K. Lucas, and A. Holmstrom. Orientations to video games among gender and age groups. *Simulation & Gaming*, 41(2):238–259, 2010.

[13] T. Hartmann and C. Klimmt. Gender and computer games: Exploring femalesâĂŹ dislikes. *Journal of Computer-Mediated Communication*, 11(4):910–931, 2006.

[14] A. Hicks, B. Peddycord III, and T. Barnes. Building games to learn from their players: Generating hints in a serious game. In *Intelligent Tutoring Systems*, pages 312–317. Springer, 2014.

[15] T. Hirashima and M. Kurayama. Learning by problem-posing for reverse-thinking problems. In *Artificial Intelligence in Education*, pages 123–130. Springer, 2011.

[16] K. Hullett and J. Whitehead. Design patterns in fps levels. In *proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 78–85. ACM, 2010.

[17] Media Molecule. LittleBigPlanet. [Video Game], 2008.

[18] Nintendo. Super Mario Maker. [Video Game], 2008.

[19] D. A. Norman. *The psychology of everyday things*. Basic books, 1988.

[20] B. Peddycord III, A. Hicks, and T. Barnes. Generating hints for programming problems using intermediate output.

[21] E. A. Silver. Problem-posing research in mathematics education: Looking back, looking around, and looking ahead. *Educational Studies in Mathematics*, 83(1):157–162, 2013.

[22] D. Vyas, C. M. Chisalita, and A. Dix. Dynamics of affordances and implications for design. 2008.

[23] D. Yaroslavski. LightBot. [Video Game], 2008.