Evaluating a Solver-Aided Puzzle Design Tool

Joseph C. Osborn
Michael Mateas
Expressive Intelligence Studio
University of California at Santa Cruz
Santa Cruz, USA
jcosborn@soe.ucsc.edu
michaelm@soe.ucsc.edu

Proceedings of MICI 2017: CHI Workshop on Mixed-Initiative Creative Interfaces.

Abstract

Puzzle game levels generally each admit a particular set of intended solutions so that the game designer can control difficulty and the introduction of novel concepts. Catching unintended solutions can be difficult for humans, but promising connections with software model checking have been explored by previous work in educational puzzle games.

With this in mind, we prototyped a design support tool for the PuzzleScript game engine based on finding and visualizing shortest solution paths. We evaluated the tool with a larger population of novice designers on a fixed level design task aimed at removing shortcuts. Surprisingly, we found no difference in task performance given an oracle for shortest solutions; this paper explores these results and possible explanations for this phenomenon.

A video recording of the tool in use is available at https://archive.org/details/PuzzlescriptAssistantDemonstration.

Author Keywords

Design support; game design; puzzle games

ACM Classification Keywords

H.5.2. [Information Interfaces and Presentation (e.g. HCI)]: User Interfaces; D.2.5 [Software Engineering]: Testing and Debugging

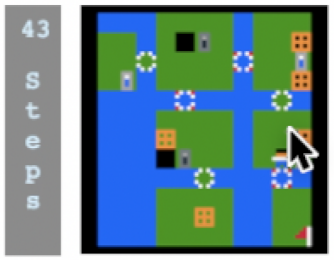

Figure 2: A (shortcut) solution to the first level displayed in the tool. Note the position of the mouse cursor, currently scrubbing through the steps.

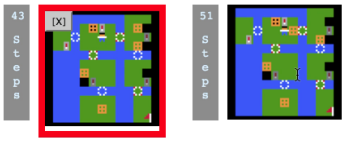

Figure 3: The same solution display updated after a level design repair. The old solution is marked as unavailable with a red border.

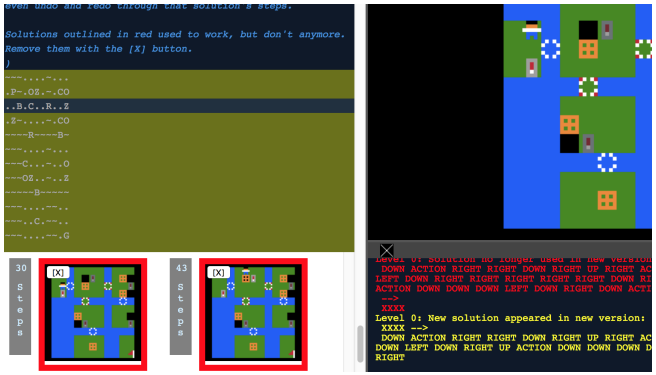

Figure 1: PuzzleScript and the PuzzleScript Analyzer.

Introduction

Designing games is difficult for many of the same reasons that computer programming is difficult: game designers, like programmers, define an initial state and operations on that state given user input, and these operations can interact in unexpected ways or the initial state may be misconfigured. Existing tools for analyzing and testing programs mainly focus on functional requirements, i.e. input/output behavior. In games, however, nonfunctional requirements such as teaching a progression of skills play a comparatively greater role, necessitating new design-focused analysis tools.

Accordingly, several researchers have proposed tools and techniques to assist game designers. Some tools generate elaborated alternatives for the designer's consideration (e.g. candidate game levels [10, 5]), while others show the consequences of design decisions (e.g. solution or reachable space visualizations [8, 1]). We follow the latter class of tools, providing design-time analysis of PuzzleScript levels.

The literature makes the reasonable assumption that such tools will aid the game design process and help designers create higher-quality designs. While many game design tools have had some form of user evaluation, we are aware of no controlled studies that compare design outcomes between two groups of users, one group using a tool and one not (the closest might be [2], which measured no difference in engagement time between fully- and partially-human-authored game level progressions). This means there is little evidence supporting the fundamental assumption behind game design support tools. We present the negative result of our attempt to gather such evidence to prompt discussion on measuring the leverage such tools provide. In the remainder of this paper we present our tool, the experiment, and a discussion of our results.

PuzzleScript Analyzer

The PuzzleScript Analyzer (PSA) can find solutions to levels for any game written in the programming language *PuzzleScript* [4], a domain-specific language for puzzle games. Solution search is done via A*; it is similar to [6], but with a different heuristic and more attention to performance. PSA is integrated into PuzzleScript's web-based editor and is automatically run when the rules or levels change (see Fig. 1). Designers can scrub back and forth through a solution's steps using the mouse (see Fig. 2) and load up the currently displayed state for interactive play by clicking on it. PSA alerts the designer if the shortest solution changes in length or becomes invalid as the game level and rules are changed (see Fig. 3).

From a requirements standpoint, our work is most similar to two design tools for the game *Refraction*, which teaches fractional arithmetic. The first tool evaluates puzzles to ensure that every solution satisfies designer-provided properties [9], while the second generates progressions of levels to teach designer-provided concept sequences [2].

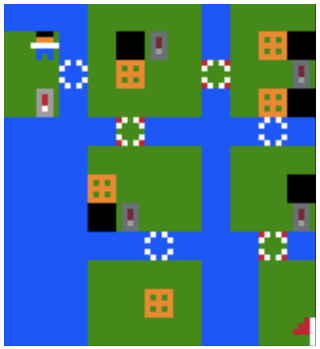

Figure 4: The first level used in the experiment.

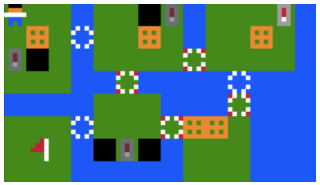

Figure 5: The second level used in the experiment.

Following the Refraction tools and e-mail interviews we conducted with five PuzzleScript users, we believed visualizing the shortest solution would help designers. Three of our interviewees asserted that *shortcuts*—solutions which did not require learning or using desired concepts—were a problem, especially when levels were meant to teach concepts in order. More formally, shortcuts (or *workarounds*) are a class of design bug where the designer's *intended solution* is either not optimal or is not the only solution. PSA helps find *true shortcuts*, i.e. those which both circumvent and are shorter than the intended solution.

We could not simply visualize solutions by superposing the player's path on the level: puzzles are often transformed during play, and players might control multiple characters. Instead, we provide an interface for scrubbing through solution steps to see intermediate puzzle states. Moreover, puzzle game rules and levels both undergo iteration; following *Inform 7's Skein* [7], we inform the user when previously-seen shortest solutions are no longer optimal or valid.

Our experimental approach differs from previous evaluations of mixed-initiative co-creative interfaces in e.g. [11]. First, PSA certainly meets that work's requirements of co-creativity: unanticipated proposed solutions can prompt lateral thinking, and scrubbing through solutions aids diagrammatic reasoning, potentially replacing (some) manual playtesting as part of the iterative design cycle. This indeed was the case: reduced manual playtesting and increased level edit counts were the biggest differences between our experimental and control groups.

Unlike [11], we are more interested in evaluating the final artifacts than the creative process itself. Moreover, our goal is not to support unconstrained creativity, but creative solutions to the challenge of avoiding shortcuts or workarounds in puzzles. This may be more relevant to junior designers or in games where levels have specific learning objectives that must be satisfied, e.g. in educational games or to ensure players learn necessary skills before proceeding.

Experiment

We evaluated PSA on a population of 195 novice puzzle designers (game design undergraduates) on a fixed level design task split into two conditions: one without and one with the tool. The control group used PuzzleScript's standard editor, while the experimental group also had the level highlighting and solution viewer interfaces highlighted in Fig. 1. Novice designers in an artificial environment are not an ideal population, since they might not be comfortable with or interested in the design task. On the other hand, we hoped that our tool could help even novice designers work more effectively. To account for differences in PuzzleScript experience, every user received a 1-hour lecture on PuzzleScript in the week prior to performing the task. The day each group performed the task, we gave an additional 1-hour introductory workshop on PuzzleScript.

For our purposes, the task of puzzle design is the arrangement of elements from a fixed set. We devised a small puzzle with four rules:

1. Push *boxes* by walking into them.
2. Flip *switches* to toggle which of the *red* and *blue* colored *bridges* are up or down.
3. Some switches have adjacent black *targets*, which must be filled with a box in order to use the switch.
4. If a box is on a bridge and the bridge is toggled down, the player may walk over the *lowered box*.

For the experiment, we defined two levels (Figs. 4, 5). The first was intended to teach rules 1-3 and the second rule 4. Two bugs were intentionally inserted into each level: one

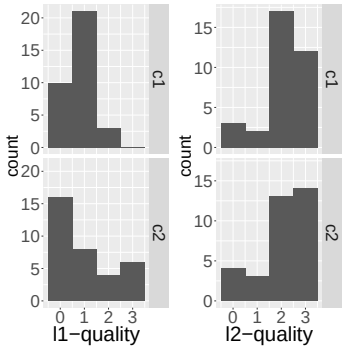

Figure 6: Solution quality by condition and level. The median solution for level 1 was worse than that for level 2, with no substantial differences between the two groups.

Scoring:
Level 1: 3 points if all boxes on targets; 2 if one box on a blue bridge; 1 if one box on a red bridge; otherwise, 0.
Level 2: 3 points if four boxes on targets and one on a blue bridge; 2 if two boxes on targets and one on a red bridge; 1 if at least one box on a red bridge; 0 otherwise.

which allowed skipping a large part of the level, and another which violated the provided description of the intended solution. Users were given the game rules, levels, and intended solution for each level; the experimental group was also given PSA and instructions for its use. Halfway through the task (at 20 out of 40 minutes), a hint was given describing the bugs in each level so that we could discover the effectiveness of the tool in solving design problems even for users who didn't find the bugs themselves. For each level, the user was asked to describe their belief that they fixed the design problems on a 3-point scale.

Both groups were using a version of PuzzleScript's editor instrumented to anonymously record level changes, game play, and other actions. Of our 195 users, 107 made a good-faith effort to complete the task (i.e., they edited the level text at all and marked the task as completed). Solutions were scored on a 0-3 scale based on how many design flaws were fixed and how completely they were fixed (participants were unaware of the criteria). We asked users to make minimal repairs and not to add or remove crates, targets, or bridges so as not to end up with levels that were unrecognizable to our solution metric.

Unfortunately, from our original 195 participants, we had to throw out data for 127 of them for the following reasons: a bug in our telemetry code that failed to collect telemetry for some participants; cases where we could not reconstruct the final game definition from the recorded sequence of edit operations; and cases where the levels were left in unsolvable states (we had no sound way of evaluating the solution quality of unsolvable levels). Ultimately, we were able to analyze 68 of the resulting activity records, of which 34 came from each group. All 68 completed both tasks, and our figures and tables refer to those 68 users. We also asked survey questions assessing proficiency at solving and designing puzzles, using PuzzleScript, and computer programming; responses were similar across the groups.

Evaluation

Our central hypothesis was that *the use of PSA would lead to better solutions faster*. Surprisingly, we found *no significant effect* on solution quality across the two conditions (see Fig. 6). The mean quality scores were basically the same, and a Mann-Whitney U-test across the groups verified that even the small observed differences were insignificant ($p = 1.0$ for level 1 and $p = 0.92$ for level 2).

We also derived an error measure for each user by normalizing the reported confidence value for each level to the 0-3 range used for level score, then taking the difference between that confidence value and the actual quality score. Our second hypothesis was that the experimental group would show *more consistent self-assessment of solution quality*. This measure also showed *essentially no difference* between the two groups either in confidence or in error.

Did the PSA group use the tool at all? The telemetry shows that they did. The experimental group spent less time manually playtesting the level and more time modifying it, performing on average 500 fewer game moves than the control group (about a 30% reduction), with a two sample t-test yielding a strongly significant p-value < 0.005. Everyone in the experimental group scrubbed through solutions, though very few clicked to load a solution step in the editor (those who did load steps did so very frequently, suggesting that the feature is simply non-obvious).

We also saw a near doubling in level edit operations among the experimental group, and a two sample t-test gave a significant p-value < 0.05. This suggests that the experimental group performed much more iteration on their puzzle levels, which conventional wisdom suggests would yield better

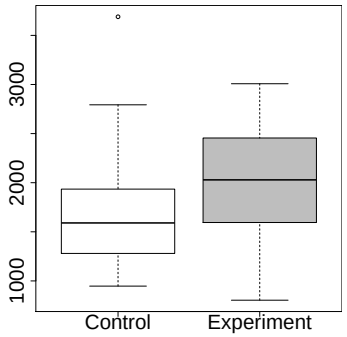

Figure 7: Time taken to complete the task in seconds for each condition. The experimental group took about 8 minutes longer on average to complete the tasks.

solutions. We then looked for a significant correlation between level edit counts and level quality. There was a significant moderate increase in the first level's solution quality with increased edit counts in the experimental group (increased scores by about 0.5, $p < 0.005$), but nothing that held across both levels or across the two groups (all small and insignificant effects with $p > 0.3$).

Discussion

It was clear that the PSA group used the tool, yet we saw no difference in performance. PSA encouraged increased iteration on levels, but this did not lead to superior solutions! This invalidates our hypotheses and some assumptions behind previous work, so we looked for other explanations.

Was the population ill-suited to the task? We only evaluated PSA on novice designers. We did ask for self-assessments of puzzle solving and design proficiency, but found no correlation between these measures and solution quality. This suggests either that the task does not measure these skills or that the students' self-assessments were inaccurate. The self-assessments were not informative, since puzzle familiarity did not correlate strongly with either confidence or solution quality. If our population truly consists of only novices, then PSA does not help novice designers on the level repair task; if the users were a mix of novices and experts, PSA does not help *anyone* on the level repair task and the task is equally hard for novices and experts. The former possibility seems more likely, though it remains surprising. Future work with expert users could help answer this question.

Was the study designed poorly? While we tried to pick a natural task—removing unwanted solutions from a puzzle level—some artificiality was unavoidable if we wanted to compare results across two groups. This may point to an innate challenge in evaluating creativity support tools, but our study did have some specific avoidable issues. For example, we provided a hint that described the level design bugs; this meant that if PSA played a role in *finding* (as opposed to *resolving*) the bugs, that effect may have been reduced. On the other hand, most users finished in under 30 minutes, so this is possible but seems unlikely (see Fig. 7). In fact, the experimental group took on average 17% longer to complete the task (a two-sample t-test yielded $p < 0.05$).

Is the tool UI unhelpful? It could be that having solutions provided automatically leads users, perhaps especially novices, to pay less attention and think less about their decisions. This could account for the increased edit count among the experimental group to no apparent benefit, as initial wrong guesses regarding the true design issue required corrections. Given that PSA's interface does show specific steps to produce problematic solutions, this seems unlikely to us. Another issue with PSA is that it does not foreground the specific instant of departure from the intended solution, or indeed validate the intended solution at all. While specifying intended solutions formally is a lot of work, it stands to reason that a tool which accepted and enforced those specifications would be more useful than a tool which merely shows solutions and asks users to check that they are acceptable. Still, one might expect that *something* is better than *nothing*, which is not borne out by our study. This leaves us with two questions:

- Does increased iteration on puzzle levels make them *safer* with respect to intended solutions?
- Does puzzle design benefit with respect to solution-safety from the use of automatic solution finding?

If the answers to these questions are negative, we must ask about the role of design support tools, especially design *validation* tools, in the game design process. The second question in particular seems to be a central assumption

of many computational game design aides, and either it is false in general—a claim for which this paper is weak evidence, but evidence nonetheless—or it does not hold in our evaluation scheme. This may be because the task is too easy or because solvers do not adequately support the specific subtasks explored in this evaluation.

We intend to perform more targeted evaluations of the PSA to explore these possibilities. For immediate future work, this evaluation should be conducted with populations of expert and novice users, perhaps using other puzzle games, other level design bugs, or all of the above. After all, it has been shown for some creative tasks that novices benefit from tool support [3] — perhaps the support that PSA provides does not help novices effectively. It may also be the case that PSA best supports puzzle *rule* design iteration as opposed to puzzle *level* design iteration, or that it helps prevent bugs from being added rather than helping to remove bugs. PSA's utility as an automated regression testing tool was not evaluated in this study. Integrating specific support for enforcing intended solutions into PSA could help answer some of the study design questions above, as could improving the user interface based on small-scale user studies.

References

[1] Aaron William Bauer and Zoran Popović. 2012. RRT-Based Game Level Analysis, Visualization, and Visual Refinement. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[2] Eric Butler, Adam M Smith, Yun-En Liu, and Zoran Popovic. 2013. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 377–386.

[3] Nicholas Davis, Alexander Zook, Brian O'Neill, Brandon Headrick, Mark Riedl, Ashton Grosz, and Michael Nitsche. 2013. Creativity support for novice digital filmmaking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 651–660.

[4] Stephen Lavelle. 2013. PuzzleScript. http://puzzlescript.net. (2013).

[5] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. 2013. Sentient Sketchbook: Computer-aided game level authoring. In *Proceedings of the Eighth International Conference on the Foundations of Digital Games*. 213–220.

[6] Chong-U Lim and D Fox Harrell. 2014. An approach to general videogame evaluation and automatic generation using a description language. In *2014 IEEE Conference on Computational Intelligence and Games*. IEEE, 1–8.

[7] Aaron Reed. 2010. *Creating interactive fiction with Inform 7*. Cengage Learning.

[8] Mohammad Shaker, Noor Shaker, and Julian Togelius. 2013. Ropossum: An Authoring Tool for Designing, Optimizing and Solving Cut the Rope Levels. In *Proceedings of the Ninth Aaai Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press.

[9] Adam M Smith, Eric Butler, and Zoran Popovic. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design.. In *FDG*. 221–228.

[10] Gillian Smith, Jim Whitehead, and Michael Mateas. 2010. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*. ACM, 209–216.

[11] Georgios N Yannakakis, Antonios Liapis, and Constantine Alexopoulos. 2014. Mixed-initiative co-creativity.. In *Proceedings of the Ninth International Conference on the Foundations of Digital Games*.