ABSTRACT

HICKS, ANDREW GREGORY. Design Tools and Data-Driven Methods to Facilitate Player Authoring in a Programming Puzzle Game. (Under the direction of Tiffany Barnes.)

Games-Based Learning systems, particularly those that use advances from Intelligent Tutoring Systems (ITS) to provide adaptive feedback and support, have proven potential as learning tools. Taking their lead from commercial games such as Little Big Planet and Super Mario Maker, these systems are increasingly turning to content creation as a learning activity and to better engage a broader audience. Existing programming puzzle games such as Light-Bot, COPS and Gidget allow users to build their own puzzles within their games. However, open-ended content creation tools like these do not always provide users with appropriate support. Therefore, player-authors may create content that does not embody the core game mechanics or learning objectives of the game. This wastes the time of both the creator and of any future users who engage with their creations. Better content creation tools are needed to enable users to create effective content for educational games.

A significant barrier to using user-authored problems in learning games is the lack of expert knowledge about the created content. Many Games-Based Learning systems take cues from ITS and use expert-developed contextual hints and content for individualized support and feedback. Data-driven methods exist to generate hints and estimate which skills particular problems may involve, but these methods require sufficient data collection and knowledge engineering to turn prior student work into hints. No prior work has shown that data-driven methods can be used within a programming game. In addition, data for user-generated levels may be very sparse, so evaluation is needed to determine if these methods can be used in this domain.

In this work, I present a set of best practices for designing authoring tools that encourage users to build gameplay affordances into their content. First, I show that requiring users to solve their own levels effectively filters some of the least desirable puzzles, including deliberately impossible or unpleasant levels. Next, I show that the quality of user-authored BOTS puzzles can be improved using level editors designed with these practices. Furthermore, I show that hints and feedback for this content can be created using data-driven methods. I also provide estimates of the amount of data needed to provide adequate hint coverage for a new level using this method. Combined, these findings form an effective framework for integrating user-authored levels into the BOTS game and provide an example of how to build a game from the ground up with user-generated content in mind.

These contributions will help future designers create tools that can effectively guide, filter, and leverage user-generated content that will contain gameplay affordances that support the intended game and learning objectives.

Design Tools and Data-Driven Methods to Facilitate Player Authoring in a Programming Puzzle Game

by
Andrew Gregory Hicks

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

James Lester	David Roberts
Roger Azevedo	Tiffany Barnes Chair of Advisory Committee

BIOGRAPHY

Drew Hicks was born in Parkersburg, West Virginia. While earning a Bachelor's Degree in Computer Science from Marietta College in Marietta, Ohio, he attended a Research Experience for Undergraduates under Dr. Tiffany Barnes at the University of North Carolina at Charlotte. Her mentorship and guidance led him to pursue a Master's in Computing and Informatics, during which he began pursuing the research questions that form the basis of this dissertation. During his academic career, Drew earned a Graduate Research Fellowship from the National Science Foundation, taught computer programming to young students at Athens College in Greece, and helped mentor several undergraduates to begin their own research. He collaborated with researchers at TERC to predict player dropout and measure implicit science learning in games, and led a team including graduate students and undergraduates to design and develop BOTS, a programming puzzle game, and to implement the game as an activity in camps and after-school programs for STEM outreach. Drew is a tabletop and digital game designer and member of the Game Designers of North Carolina. Since 2016, he has worked at IBM Watson developing deep learning models to improve the processing of medical text. These experiences have driven him to never stop learning, never stop teaching, and never stop playing.

ACKNOWLEDGEMENTS

I would like to thank my fellow graduate and undergraduate students who have contributed to BOTS development and outreach: Michael Eagle, Michael Kingdon, Alex Milliken, Aaron Quidley, Barry Peddycord III, Aurora Liu, Veronica Catete, Rui Zhi, Yihuan Dong, Trevor Brennan, Irena Rindos, Vincent Bugica, Victoria Cooper, Monique Jones, and Javier Olaya. Thanks to the team who worked on BeadLoom Game, including Acey Boyce, Dustin Culler, Antoine Campbell, and Shaun Pickford, for their advice and assistance with outreach activities. I would also like to thank my committee members Dr. Tiffany Barnes, Dr. James Lester II, Dr. Roger Azevedo, and Dr. David Roberts for their support and guidance. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 0900860 and Grant No. 1252376.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

As they become increasingly culturally relevant, games are being looked to as a way of improving education. Game-based learning systems are said to provide intrinsic engagement that contributes to their efficacy [Pre03; Gee07; Sha06]. Well-designed learning games allow students to work on meaningful, motivating tasks and to practice skills. Garris, Ahlers, and Driskell classified the factors that are important to games' effectiveness for learning, identifying motivation to play and to play again as key features of the best-designed and most effective instructional games [Gar02].

One benefit of educational games is the increased motivation students may have to complete material presented in a game environment [Gee07]. This effect is apparent in voluntary experiments, where subjects are free to fail with no penalty. However, this effect may not translate well when games are used as required or graded learning activities. In order to better harness their motivational advantages, game-like systems must provide a different experience than traditional assignments. Ideally, games should allow players to engage in in-depth learning and continued practice over time.

Another potential benefit of educational games is improved educational outcomes [Pre05; Gee07; Sha06; Squ13]. Though it is often inferred that the increased engagement observed in game-based assignments would necessarily lead to improved educational outcomes, there is a lack of rigorous research to show that is the case [May14; Vog06]. However, recent literature reviews and research performed by Mayer et al. and collected in the book Computer Games for Learning [May14] showed that, when carefully designed, computer games can achieve the same learning gains as traditional assignments while reaping the aforementioned motivational benefits and in some cases can improve educational outcomes beyond traditional assignments. Work by Mayer, Lester, Boyer, Shute, Squire and Steinkuhler (among others) [MJ10; Les14; Boy15; Shu13; Shu08; Squ13; Ste12; SS14] demonstrated this for various domains. Other authors such as Boyce [Boy11] and Millis [Mil16] explored the effects of adding game-like features to existing systems, with some success. However,

as Mayer found, there is also a substantial amount of research where these gains are not shown. One explanation for why a game may fail to achieve the same learning gains as a traditional assignment is the extrinsic cognitive processing hypothesis. The mechanisms, story, and controls of the game, where they do not align with the learning goals, may add too many other elements for students to keep track of. The addition of game elements may thus result in cognitive overload. This was hypothesized to be the case by Mayer in his writing about the games Crystal Island and Cache-17 [Ada12; Spi11; Koe08].

Although they may have significant benefits for students, educational games can be costly to build, particularly in terms of expert time. In the Intelligent Tutoring Systems (ITS) literature, it is estimated to take around 300 hours of expert time to develop an hour of educational content for an ITS [MA02]. This estimate is insufficient for Serious Games, for several reasons. First, additional content must be developed in the form of game assets, including graphics, sounds, and other game elements. Second, additional time must be spent teaching users the mechanics, interface, and controls of the game, as well as explaining the parallels between the game and the target content. Finally, additional expert time is needed to ensure that the underlying game design aligns with the target content and does not distract from it. Many Serious Games projects are developed by small research teams with limited resources, as opposed to the larger teams available to entertainment software developers. Because of this, many Serious Games are relatively short, often comprising a single assignment's worth of educational content.

While few Serious Games transcend the single assignment boundary, there are examples of Intelligent Tutoring Systems (ITS) that do. Systems like ASSISTments are consistently used over a longer duration, allowing for and encouraging practice. The ability to revisit problems and practice skills is a crucial feature of these systems [Raz05]. While practice is important for learning, choosing what and when to practice are also important. This is a potential tension within educational game design; providing a sense of control over the experience must be balanced with learning needs. Work on gamification of the pre-algebra tutoring system Lynette showed that players would over-practice problems they knew the answers to in order to earn the "gamification"-style medals that were offered as rewards [LA14]. This is an example of a common gamification pitfall, where the game's rules come into conflict with the learning activity. In these cases, the players' most effective strategy to practice and master the game's rules does not align well with an effective strategy for practicing and mastering the learning objectives. If players are given increased control to explore a larger body of content, it is therefore vital that the game's rules and the learning objectives be well aligned.

1.1 Research Questions

One strategy that may mitigate the high cost of content creation while delivering learning benefits and the ability to replay for practice is to allow users to create content in ITS and Serious Games. In addition to its usefulness for improving a game's longevity and replayability, content creation has been shown to have significant motivational benefits, appealing to players that current gamification approaches fail to address. Boyce et al. showed that adding a creative gameplay element such as level design within a Serious Game could increase learning gains and improve motivation while appealing to players with diverse motivations to play games [Boy14]. Problem-posing systems such as Animal Watch and MONSAKUN [BB08; Hir07] demonstrated the educational potential of content creation activities. However, in order to include the designed content for reuse with future players, it must be of sufficient quality, both in terms of gameplay and educational value. If user-authored content fails to address the educational target concepts, then any time users spend playing it is effectively "off-task" time. If user-authored content is uninteresting from a gameplay perspective, users will simply not play it.

To evaluate the potential for user-authored content, I developed a game called BOTS that builds upon previous game-like novice programming environments like Logo, Light-Bot, and Scratch. By investigating user-authored content in this game, I gained insight into the broader possibilities for user-generated content in educational games and systems. The research questions guiding this work are as follows:

1.1.1 Research Question Q1

Research question Q1 is "How can non-expert student authors create high-quality levels that afford and reward the use of core mechanics in BOTS?" To answer this question I first collected and analyzed user-created levels in a simple version of the game. Based on insights from this analysis, I made iterative changes to the level editor to encourage the inclusion of "gameplay affordances" and use of advanced gameplay mechanics from later players.

1.1.2 Research Question Q2

Research question Q2 is "How can level authoring tools designed using creative constraints help promote the creation of high-quality levels that afford and reward the use of core mechanics in BOTS?" To answer this question, I evaluated versions of the level editor against each other, based on different methods of gamifying content creation. First, I compared open submission of levels to self-moderation by asking users to propose solutions to their own levels. Second, I compared

a building-block based system that uses pre-built level segments to a "deep gamification" based editor that uses the main game's mechanics. Each approach had different effects on the presence of "gameplay affordances" that afford and reward the use of the game's core mechanics.

1.1.3 Research Question Q3

Research question Q3 is "How can existing data-driven methods for knowledge discovery and feedback generation from Intelligent Tutoring Systems be applied to brand new user-authored levels?" To answer this question, I first adapted the Hint Factory algorithm to open-ended programming puzzles. Then, I evaluated hint coverage for several BOTS puzzles to determine how much data is needed so that hint generation works for about 80 percent of user states.

1.2 Hypotheses

To answer my research questions, I conducted a sequence of experiments evaluating different designs for the level editors available in BOTS and iteratively developed the level editor in BOTS based on the insights from each experiment. My hypotheses for these experiments are as follows:

1.2.1 Hypothesis H1

Hypothesis H1 is "Asking authors to provide solutions to their levels before publication will increase the quality of published levels and reduce the number of submitted problems with trivial, tedious, or non-existent solutions." H1 is supported by work in Chapters 3 and 4. I conducted experiments with two versions of the game, with different level submission mechanisms, and compared the levels created by measuring the presence or absence of qualities of trivial, tedious, or non-existent solutions.

1.2.2 Hypothesis H2

Hypothesis H2 is "Player traces can be used to automatically generate low-level hints for player-authored problems." H2 is supported by work in Chapter 5. I conducted an analysis of the Hint Factory method applied to BOTS data, made modifications to adapt the technique to open-ended puzzle environments such as BOTS, and determined how much data is needed to achieve 80-percent hint coverage for BOTS puzzles.

1.2.3 Hypothesis H3

Hypothesis H3 is "Authoring tools designed with creative constraints will result in authors more frequently creating levels that afford the game's core mechanics, when compared with the original free-form authoring tool." H3 is supported by work in Chapter 6. I developed two new level editors for BOTS based on existing methods of gamifying content creation. I then compared new levels created in these editors to levels created in previous editors, based on the presence of "gameplay affordances" in the levels as measured by the difference between expert and naive solutions.

1.2.4 Hypothesis H4

Hypothesis H4 is "Levels that contain these gameplay affordances will more frequently contain correct use of the game's advanced mechanics in players' solutions." H4 is supported by work in Chapter 7. I ran an experiment where new players solved the levels created in the new level editors. I then measured how well a level's "gameplay affordances" predicted whether observed player solutions would use an advanced game mechanic.

1.3 Contributions

By synthesizing data-driven methods from Intelligent Tutoring Systems and current research into Gamification and Game Design, I designed a system to accomplish my stated goals. With BOTS as a case study, I also developed design principles that will help future developers effectively integrate user content creation into Serious Games and other game-like cyberlearning systems. The contributions of this research are as follows:

1. BOTS, itself; a serious game for novice programmers that supports user-generated levels via a gamified level editor. This will be the first such system, as well as the first such system for novice programmers.

2. A novel application of Intelligent Tutoring Systems methods for knowledge discovery and hint generation to user-authored levels, and evaluation of the amount of data needed to effectively cover a user-authored problem in BOTS.

3. Design principles for level editors to improve the quality and usability of user-generated levels.

1.4 Organization

The remainder of this chapter presents the terminology used throughout the dissertation. Chapter 2 outlines the literature that guides the design choices for BOTS and the related work in intelligent tutoring systems, education, and games. Chapter 3 describes the BOTS game and a pilot study to learn more about what kinds of content players make when given the opportunity. Chapter 4 describes an experiment to investigate the impact of moderation types on the kinds of puzzles that players create, and provides puzzle classifications and feature tags that are used to describe (primarily undesirable) features in user-generated puzzles in the remainder of the work. Results from Chapter 4 led to implementing self-moderation, where players must solve their own puzzles before they are published, for all puzzles in BOTS. Chapter 5 illustrates that data-driven methods can be used to generate hints for user-generated puzzles in BOTS. Chapter 6 introduces two new level editors, the Programming and Building-Blocks editors, and shows that these design changes did encourage players to create more desirable content while they design levels. Chapter 7 shows that levels with higher potential for reduced lines of code actually result in more player solutions using loops and functions. Of particular interest is that the level editor type did not impact whether players solved puzzles using loops or functions. Instead, the combined potential for reducing the lines of code in the solution and the presence of obvious visual and structural cues was more predictive of whether solutions contained loops or functions. Chapter 8 reviews the research questions, confirms support for my hypotheses, and highlights the contributions of this work. Chapter 8 also provides direction for future work in BOTS and design recommendations for promoting effective user-generated content in learning games.

1.5 Terminology

The research terms used in this dissertation are defined here:

- Level
  Puzzle: In this document, *Level* and *Puzzle* refer to individual exercises in the BOTS game.
- Level Editor: In this document, a *Level Editor* is an in-game tool that can be used by players to create levels.
- Author: In this document, an *Author* is a player who creates levels using the level editor. If used in reference to a specific level, the author is the player who created that level.

- User-Generated Content: In this document (and in reference to BOTS), *User-Generated Content* or UGC refers exclusively to *Levels* generated by students.
- Puzzle Game: In this document, *Puzzle Game* refers to a single-player game where that player solves distinct stand-alone problems using discrete steps, with different solutions having varying degrees of success.
- Core Mechanics: In this document, a *Core Mechanic* refers to the central idea of the game, and the mechanism players use to act upon that idea within the game. For BOTS, the Core Mechanic is optimizing programs through correct use of loops, subroutines, conditionals, and variables.
- Affordance: In this document, a level contains an *Affordance* for a core mechanic if performing that core mechanic within that level improves a player's score, and that level contains structural cues to communicate appropriate places for players to perform that core mechanic.
- Creative Constraint: In this document, a *Creative Constraint* refers to a constraint imposed upon the author by the system, with the goal of encouraging or discouraging specific types of creative behavior. For example, a limit on the number of operations is a creative constraint that may encourage authors to build more optimally or to omit unnecessary structures.
- Naive Solution / Expert Solution: In this document, a *Naive Solution* to a given level is a solution that only makes use of the basic movement blocks, without taking advantage of loops or functions. Such a solution may indeed follow the shortest physical path to the goal. Conversely, an *Expert Solution* is a solution taking full advantage of loops and functions. Such a solution may not always follow the shortest physical path to the goal, but will use a smaller number of instructions than the Naive Solution.
- Quality: In this document, the *Quality* of a level is measured according to the dimensions outlined in chapter 2. To briefly summarize here, those dimensions include a level's complexity (measured by the difference in length between Naive and Expert solutions), structural cues (instances where the level geometry coincides with the changes in the robot's path between optimal and sub-optimal solutions), and non-concept difficulty (parts of the level that are difficult for reasons unrelated to the core mechanic of programming). This measurement of quality aligns with Mayer's extrinsic processing hypothesis [May14] that game elements that are not well-aligned with the learning content and that require additional processing can result in cognitive overload for players or students.

Chapter 2

Related Work

In this chapter, I outline my motivations for investigating the potential of content-creation gameplay in Game-Based Learning (GBL) environments, both as an engaging gameplay mechanism and as an effective content-creation tool. To do so, I draw upon previous work from the fields of Game Design and Intelligent Tutoring Systems.

2.1 Game Design

2.1.1 Types of Gameplay and Players

To motivate the use of content creation as a gameplay activity, I look to the field of Ludology, and the game definitions that Ludological researchers have employed.

In his book Homo Ludens, Huizinga defines a game as "A voluntary activity or occupation executed within certain fixed limits of time and place, according to rules freely accepted but absolutely binding, having its aim in itself and accompanied by a feeling of tension, joy, and the consciousness that it is 'different' from 'ordinary life' " [Hui55].

In his work on play, Caillois defines two different types of play: Paedic play and Ludic play. Paedic play can be generally described as 'make-believe' play, loosely structured with freely shifting objectives. Ludic play involves constraining systems of rules and objectives, and can be thought of as "game play." For Caillois, nearly all games fall under the category of ludic play. Caillois further classifies games into four categories.

- agôn, contests and competitive games
- alea, games of gambling and chance

- mimesis or mimicry, detailed simulations
- ilinix (or vertigo), games of escapism or reckless abandon

Of these four categories of games, only one is specifically focused on competition.

In work classifying player types, researchers like Bartle and Yee found that if conflicts are in fact crucial to games, there still exist players who prefer other types of play experiences [Bar96; Yee06]. Bartle defines four types of players, Socializers, Explorers, Achievers, and Killers.

- Socializers, who seek to tell stories of the game and build relationships with its players
- Explorers, who seek to explore the game's systems and accumulate knowledge
- Achievers, who seek to gain status and overcome the game's prescribed challenges
- Killers, who seek to overcome the game's rules, and the other players

Of these, only Killer-type players are primarily motivated by direct conflict. Using Bartle's types as a base, Yee defines ten motivators for players in online games: Advancement, Socializing, Discovery, Mechanics, Relationships, Role-Playing, Competition, Teamwork, Customization, and Escapism [Yee06]. Of these ten motivators, only one explicitly identifies a competition, with the others highlighting other aspects that serve to draw players in.

In their anthology Rules of Play, Salen and Zimmerman provide a somewhat more formal definition of a game: "A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome" [SZ04]. Both Caillois and Huizinga's definitions, and Bartle and Yee's taxonomies of player motivations, reject the core idea that Salen and Zimmerman explicitly include: the idea of conflict as central to 'game-ness'. Game Design scholars have argued, variously, whether a defined conflict is crucial for games, but in popular mainstream digital games like Minecraft and The Sims [Moj11; Max00], conflicts, where present, are often challenges devised and self-imposed by the players, and not constructed by the game. This suggests that approaches to Serious Games / Game-Based Learning (GBL) design that focus on conflict and competition may fail to adequately engage some segment of their audience. Including additional styles of play in GBL, such as creative play through authoring and sharing of content, may help address this weakness.

2.1.2 Gamification

Gamification is a term used to describe many different methods of adding game-like elements to non-game contexts. Use of the term (if not the elements it invokes) is becoming increasingly common in software development. Though the term originated as an industry term [Bog11; Det11a;

Det13], the process of introducing game-like elements into a system for motivational purposes has found hold beyond its original scope, with games or gamified systems being created for fitness, budgeting, time management, and employee training. Many of the motivations behind gamification align with those for Serious Games.

In Deterding's work, Gamification (or Gameful Design) is defined as "the use of Game Design elements within non-Game contexts." [Det11b]. Alternately, Kapp's definition of this term is "using game-based mechanics, aesthetics and game thinking to engage people, motivate action, promote learning, and solve problems." [Kap12]. Developers of gamified systems seek to harness the incentive structures present in games to motivate users to complete otherwise tedious tasks. In Serious Games or GBL, the motivation for using a game or game-like system is typically to better engage users, with the aim of increasing learning gains or improving general attitudes towards the system or learning content.

Kapp goes further in delineating two sub-types of Gamification: Structural Gamification and Content Gamification. Structural Gamification (also referred to by Nicholson as "BLAP" gamification) [Nic12] is a method that leaves the content unchanged but "wraps" it in game elements such as badges, level progressions, achievements, and leaderboards. Gamification of this type, as currently practiced, tends to focus on disconnected reward and competition, aspects of games and play that will frequently fail to appeal to many types of players [Rob10a].

Content Gamification, in comparison, is a process that makes the content more playful via the addition of elements such as fantasy, challenge, or choice. Content-oriented approaches to gamification, such as Boyce's "Deep Gamification," tie motivating game elements directly to the practice of the game's learning objective. Additionally, less mechanically focused ways of "gamifying" systems as outlined by authors like Nicholson and Boyce when discussing "Meaningful Gamification" [Nic12] or "Deep Gamification" [Boy14], respectively, can effectively engage not only competitively-oriented players, but also creatively-oriented players.

In their work, Krause et. al define another subtype, Social Gamification [Kra15]. In addition to the game-like elements above such as badges and leaderboards, Social Gamification includes social network-based gameplay similar to popular social games on mobile devices and services like Facebook. In the system the authors developed, players were able to choose an opponent they knew for a match, rather than playing against a random opponent. This element significantly increased retention, measured by the number of lessons a user completed. Users in the Social Gamification condition also had increased post-test scores, although users did not take a pre-test so this comparison is limited.

Another notable variant of gamification is "Games With A Purpose," a class of games initially developed by Luis von Ahn [VAD08]. Exploring the area of "Human Computation", the researchers

selected tasks that are difficult for computers but relatively easy (if perhaps mundane) for humans. These tasks were then situated in a game-like framework to encourage participation and better performance from users. In "The ESP Game," for example, players collaboratively tagged images based on their contents, with only the tags they agreed upon being accepted. This visual recognition task was considered somewhat difficult to solve from a machine learning point of view, but simple for a human. Each player's performance was evaluated based on their ability to tag the given image with the same term as a remote human partner without communicating directly. The most commonly given terms were listed alongside the image, and could not be repeated by either player. This clever twist made the task more engaging while ensuring that tags suggested by players were more likely to be novel but recognizable elements of the image. Von Ahn's other GWAP projects focused on tasks like language translation and identifying important regions of images. With these systems, gamification was used to provide fun and enjoyment as a "compensation" of sorts for accomplishing an otherwise tedious and unpleasant task.

However, the game mechanics added to such systems do not need to be as clever as in the ESP Game in order to motivate users. Simply adding point values to tasks in such systems has been shown to increase participation within the system [Mek13]. In Mekler et al., users performed an image-tagging task under one of four conditions. The first condition was a simple control task, with no intervention. The second was a task with a meaningful frame: that participants' contributions would advance science by helping develop image-recognition software. The third condition was a task with a simple point reward, awarding 100 points per tag. The fourth condition combined framing and point rewards. The authors found that point rewards increased the number of tags submitted, and that when rewards were combined with framing, users were more deliberate in their tag creation, spending more time on the task. Both points and framing served to increase motivation to a similar degree, with no marked increase when both factors were combined. However, adding point rewards alone had no effect on the actual quality of tags submitted, while adding a framing device did increase the quality.

Gamification often focuses on competitive elements like points, badges, and leaderboards that encourage users to compare achievements in the system. The core idea is that users will want to achieve and retain a higher position than their friends or peers. They are therefore motivated to continue with the gamified activity whenever there is a threat of losing standing. This process is outlined in the "Gamification Loop" defined by Liu et al. [Liu11], which consists of six sequential stages:

- Challenge
- Win Condition

- Rewards
- Badges
- Leaderboards
- Social Status

However, the benefits of this model of gamification are fragile. The motivation can only persist within the gamified system, while the player actually cares about their standing on the leaderboard, and while they feel that their standing is threatened. If that motivation weakens players will no longer exhibit the same degree of motivation to participate.

Somewhat counter-intuitively, rewards (particularly performance- or completion-based rewards) can actually decrease motivation and performance when improperly or carelessly applied [Dec99; Dec01; BK12]. Despite frequently being treated as "intrinsic" rewards that make the gamified activity more engaging, rewards from Structural Gamification are only intrinsic to the game elements of the system, and are extrinsic from the core task. This suggests they may be subject to the negative motivational effects outlined in Deci et al.'s metaanalysis of extrinsic rewards [Dec99]. The authors considered 128 different experiments, concluding that extrinsic rewards had a substantially negative effect on intrinsic motivation for interesting activities. Their analysis indicated that adding extrinsic rewards had a negative motivational impact on participants who were already engaged with the task. One weakness of the meta-analysis is that few of the studies reliably measured the fundamental "interestingness" of their underlying task, for example, via participant persistence in the activity during a free choice period.

Similarly, Kathleen Tuite wrote that in systems where the "fun" of the game is provided as compensation for performing an otherwise distasteful task, learning the game and learning the task are placed at odds with each other. Specifically, users tend not to continue using the system or performing the task when given a choice when extrinsic rewards are provided and then discontinued. This results in games with reduced efficacy at both entertaining and teaching [Tui14].

2.1.3 User-Generated Content in Entertainment Games

Prensky stressed that replayability is a major component of successful games [Pre05], and that games constructed as linear one-shot experiences are not particularly replayable. The ability to tweak or remix parts of a game and share these with friends, and to hold conversations about particular parts of a game with the user community, highly increases a game's replayability. As such, the ability to customize and share game artifacts is now a common feature even in narrative-focused games,

where players can bring customized avatars into shared worlds. More advanced content creation tools can be seen in many games, with tools like Little Big Planet's level creator [Med08], Trackmania's track creator [Nad04], or The Sims' online repository of user-created objects [Max00]. User-generated content (UGC) is a boon for these games, allowing for a much wider array of content than could feasibly have been built by the development team. Additionally, users' creativity may lead them to explore areas outside of the "expert blind spot," creating experiences that subvert the game's mechanisms or combine them in new and unexpected ways. For example, Super Mario Maker levels frequently make use of unusual interactions between objects never placed together in official Mario games. In both Mario Maker and Little Big Planet, players create movie-like narrative experiences despite the limited tools offered by the platformer level creator[Nin08; Med08]. Elaborate mini-games have been built by users in Minecraft through clever use of the game's physics and "redstone" circuitry [Moj11].

One example of a successful commercial game focused on user-generated content is Nintendo's "Super Mario Maker" for the Wii U [Nin08]. In this game, players can create levels for one of four 2-dimensional Super Mario titles using a free-form tile-based editor. When the game launched, only simple editing tools were unlocked from the beginning. As players created levels, new "shipments" of more advanced level elements were added to the interface, gradually increasing the complexity as players became more familiar with the tool itself. This slow unlocking process was derided by critics and expert users [McE15], and Nintendo swiftly released a patch drastically reducing the amount of time needed to unlock each new set of game elements by linking the unlocking process to the total number of objects placed. After the patch, players quickly learned that building a level completely full of coins would accelerate the unlocking rate. Luckily, players did not have to actually upload their coin-hoard levels to the game's database in order to earn the rewards. This replicated a common gamification pitfall in a content creation tool; the best way of earning rewards did not align with the best (or most desirable) way of building levels.

The way user-generated levels are discovered and presented in Mario Maker was also the subject of some criticism. In the game, players can choose to award "stars" to levels they complete, and the levels with the most stars will appear at the top of the page when users are browsing levels. Initially, Nintendo incorporated these levels into campaigns generated based on difficulty, as measured by the proportion of failures to successful completions. However, due to the prevalence of Automatic levels (which, as the name implies, play themselves) and Kaizo levels (named after a legendarily difficult Super Mario World hack requiring exploits and single frame precision) the Easy and Hard difficulties became heavily skewed towards these extremes [Her15]. Later, Nintendo added an additional difficulty level to encompass these Kaizo type levels. Nevertheless, many reviewers singled out the presentation of other players' levels as a fault of the game. One particularly negative

review from the Washington Post called the game "an engine for circulating horrible levels" based on "bad ideas and broken gimmicks" [Tho15].

It is clear that user-generated content systems must provide some form of quality control, or be overrun by hastily created, poor-quality content. Heavily UGC-based games like Little Big Planet and Spore [Med08; Max08] have designed solutions such as limiting the number of "upload slots" an individual user has, or allowing the community to vote on whether a creation is useless or offensive. However, this approach requires that unwanted content be shown to at least some users whose judgment is trusted and whose participation will continue. This may be acceptable for some applications of collaborative filtering, (movie reviews, online marketplaces) but it is undesirable for an educational tool.

Within games research, there is existing work examining how expert and novice level designers create levels, and on designing tools to assist or mimic these creators. Hullett and Whitehead described common features of levels in FPS-style entertainment games using a pattern language, and demonstrated the usefulness of such taxonomies [HW10; Hul]. This approach was then applied to describe design patterns in other genres of games such as RPGs and FarmVille-likes. These patterns were also used as objectives for procedural level generation [Smi11c; Lew12; DT13].

Procedural Content Generation (PCG) has been widely explored in the research as a solution to the same replayability problem, in educational games such as Refraction as well as entertainment games [Smi12; MM10; Tog10]. Tools like Tanagra and LaunchPad integrate PCG with human authoring tools, resulting in levels focused on certain objectives while retaining authorial control over the final product. To achieve this hybrid design, Tanagra constructs parts of levels in between key features specified by the author, while Launchpad procedurally generates platformer levels in direct response to player actions [Smi10; Smi11a; Smi11b].

2.2 Cyberlearning Systems

2.2.1 Intelligent Tutoring Systems

Intelligent Tutoring Systems are cyberlearning systems that include some element of intelligent adaptation to their users. The goal of intelligent tutoring systems is to approach or exceed the impact of one-on-one human tutoring without requiring as much expert time. This goal is outlined in Bloom's article, "The Two-Sigma Problem" [Blo84; Cor01]. Most intelligent tutors are built based on a system of two feedback loops: an outer loop that selects the problem or task the student should work on next, and an inner loop that provides low-level feedback within the problem or exercise [Van06]. The purpose of this adaptation is to provide an environment that is more conducive to the

student's learning, presenting content that is difficult for the student to solve, while also providing the scaffolding necessary for them to solve it. This system of scaffolding that fades as the learner becomes more competent mirrors the use of flow in effective game design, as discussed earlier. An effective system should present content and feedback based on an estimation of the user's needs and abilities, challenging them without being too frustrating.

Existing ITS often use expert-authored rules and evaluations of content and difficulty to help make these adaptations [And95]. However, building these systems takes a great deal of time and expert knowledge needed from content experts, instructional designers, and software engineers [Mur03; Bay09]. Murray estimated a 300-to-1 ratio of expert hours to hours of content for Intelligent Tutoring Systems. For systems implementing user authored or procedurally generated content, this is not feasible, since an expert would need to be continually on hand to annotate new content.

Data-driven approaches like the Hint Factory created by Stamper and Barnes [Sta08] use historical data (in particular, traces of solutions to problems) to facilitate the creation of hints for future users. This allows for personalized feedback to be provided to students even on content that experts have not analyzed. In software that uses Hint Factory, as users work on problems their actions are used to build a data structure similar to a Markov Decision Process called an Interaction Network. Work by Eagle et al., defines an Interaction Network as a complex network containing data about student-tutor interactions [Eag12]. Hints can be generated from this data structure by searching the Interaction Network for users with the same or similar solution path. Based on the previous users' actions, a potential next step can be suggested. If no user has succeeded on that path before, the system can suggest the current user try a different approach. Work by Barnes, Stamper, Eagle, and Mostafavi showed this to be a very successful way of automatically generating hints, and a Hint Factory enabled version of the Deep Thought logic tutor has been used in logic classes at UNC Charlotte and NC State University since 2001 [Sta08].

2.2.2 Novice Programming Environments

BOTS is not the first attempt at using a simplified programming language to teach programming. Introductory University CS classes frequently exhibit bimodal grade distributions with an unusually high number of both excellent and poor grades and low number of middling ones. This effect is attributed variously to lack of experience or to fundamental unfamiliarity with required mental models, among other theories [Rob10b; Cas07; BD08]. Systems like Scratch, ALICE, Snap, Greenfoot, Kodu and GameMaker [Res09; Kel07; Har14; Köl10; SF11; Dor12] have been used to address the perceived need for a gentler introduction to programming and computational thinking. These systems use simplified programming languages that help minimize the syntactical hurdles students

must overcome while learning the fundamental concepts of computer programming. Drag-and-drop programming interfaces are increasingly being used to introduce novices to programming, with younger students as well as in early university courses [Har14; RH12; Ker08]. Research with these environments found that while students perceived the block-based programming activity as less "authentic" compared with its text-based counterpart, they also perceived it as easier and as more fun, even when the assignments' content was the same. Additionally, students using a block-based environment were more likely to complete assignments when compared with programming in a text-based but still scaffolded environment [WW15; PB15].

Scratch is a popular novice programming environment developed at the MIT Media Lab [Res09; ML07]. Though initially targeted at a middle-school audience, in particular youths at after-school centers in economically disadvantaged communities, Scratch has since been effectively used by the author of this work (among others) as a language for first-time programmers in introductory courses. Scratch also features a "community" site where users can share programming tips and post their projects for others to "remix." [MHH10; Hil10].

Other environments that focus on a similar audience, and that have similar social computing goals, include the Scratch follow-up Snap [Bar12], the storytelling environment Alice [Coo00; Kel07], and Kodu GameLab by Microsoft Research [SF11], among others. These efforts have a common focus: to teach programming concepts in an environment that minimizes the syntactical errors learners can make, helping instructors focus their instruction on the core concepts rather than on syntax errors and typographical mistakes. Reducing the burden of learning new syntax has been shown to be a useful approach to teaching programming. In work with the Pre-programming Analysis Guided Programming tutor (PAGP), Jin et al. removed syntactical hurdles in favor of a focus on the core programming concepts at work [Jin08]. Students first completed a planning/analysis exercise where they organized the problem steps into sequential chunks and identified relevant variables. Using this tutor helped students avoid several common novice programming errors such as asking for input where it is not needed, or using 'return 0' when a value is needed.

2.2.3 Problem Posing in STEM Education

When designing the level-editor intervention, I looked to K-12 mathematics education, where there is much exploratory research into problem posing [SC96]. There is evidence that having students engage in even very simple generative activity like rewriting story problems can improve students' motivation to learn mathematics, and improve their performance on similar problems [Has87; Lar86]. Problem posing is hypothesized to provide educational benefits above and beyond those gained from typical problem solving, including opportunities to practice explaining problems,

restating problems in new ways, and identifying gaps in the author's knowledge [Cre03]. At the same time, problem posing has been shown to increase motivation when compared to purely solving problems presented by others [DC00; Eng97]. This engagement effect is especially prominent in students who had not previously shown enthusiasm to learn math and science [Hir07].

Problem posing is a useful activity in several ways. First, problem-posing as an activity has roots in models of learning like self-directed learning and inquiry-based learning, and engages students in different ways than traditional problem-solving. Problem posing activities may also help students develop meta-cognitive skills by requiring them to think about how they would solve the problem they have posed, and how changes they make to their problem will affect the solution process. Finally, problem-posing may increase engagement and help alleviate anxiety, especially in cases where they believe their problems will be useful [Eng97].

Problem posing has been acknowledged as important to math education. In their publication of the Principles and Standards for School Mathematics, the National Council of Teachers of Mathematics argued that students should be formulating problems, extending problems, and posing follow-up questions [Mat00]. In his overview of the research into the topic in the field of mathematics education, Silver noted that traditional psychological measurements of creativity often include a problem-posing component [Sil13].

In a study performed by Silver in 1997, 509 middle-school students were tasked with creating problems, followed by a test composed of eight questions on varying math topics, with answers and justifications required. In this study, students were first given a list of information, for example: "Jerome, Elliot, and Arturo took turns driving home from a trip. Arturo drove 80 miles more than Elliot. Elliot drove twice as many miles as Jerome. Jerome drove 50 miles." Then students were prompted to supply three questions that could be answered from the information given. Questions were first categorized as mathematical questions, non-mathematical questions, or statements. Then, the mathematical questions were classified as solvable or not solvable. This final set of problems were then evaluated based on linguistic features to determine their complexity. In this study, nearly 80% of the students generated a mathematical question, and more than half generated only mathematical questions. More than 90% of these were solvable. Although 60% of the questions fell into the simplest linguistic complexity category (only using assignment propositions, questions like "How many?" rather than "How many more?") 60% of the problems were also classified as mathematically complex, requiring the solver to examine two or more relationships between variables. Students tended to create questions whose responses depended on answers to prior questions, with simpler problems leading into harder ones.

Problem posing is also useful for identifying students' misunderstandings. In Alexander's 2010 research published in "Mathematics Teaching In The Middle School" the authors examined the

results of a problem posing experiment [AA10]. Students were asked to create a very specific type of problem, one whose solution involved adding the specific values 1/2 and 2/3. By examining the created problems, the authors were able to identify specific misconceptions common among their students (such as $1/2 + 2/3 = 3/5$).

2.2.4 User-Generated Content in Cyberlearning Systems

Chi posited that one reason human tutors are so effective is that working with a human tutor creates opportunities for external construction [Chi01]. Examples of external construction include forming hypotheses or posing examples or analogies. Problem posing activities may also fall under this category. Some researchers have looked into ways of potentially alleviating the content bottleneck in ITS using user-authored problems, while others seek to implement problem-posing for the educational benefits rather than for the purposes of content generation.

Gehringer investigated student-generated learning exercises and found that student exercises were not conclusively better or worse than expert-generated exercises [GM09]. However, when students generated game-like activities as exercises, the self-reported learning of students decreased while enjoyment rose [GM09].

Work by Aleahmad et al. showed that non-expert users are competent content creators, often able to provide more understandable problem descriptions than self-described experts [Ale08]. The authors also expected that content creators would create higher-quality content when creating content targeted for a fellow user. While this was not shown to be the case, the users who created content for other users were more engaged in the content creation process than users creating content with no specific goal. They spent more time creating material and added more specific details to their problem descriptions. Though the content they created was not judged to be of higher quality from an educator's perspective, it was not evaluated whether that content was of higher quality from a target user's perspective.

Williams et al. created a "learnersourcing" system AXIS [Wil16] that asked learners to generate, revise, and evaluate explanations for future learners. The system then used a machine learning algorithm similar to existing recommendation systems to dynamically determine which explanations to present. Their results showed that this process elicited helpful explanations and enhanced learning when compared to a control condition. Additionally, the quality of crowd-sourced explanations did not differ from explanations generated by an experienced instructor.

There has also been work with peer evaluation and modification of content. Further work by Gehringer with the peer-review system Expertiza adopted design elements from both social games and review-sharing websites like Yelp such as leaderboards, voting, and reputation systems. These

changes led to improved quality of peer assessments [Geh10; Geh16].

In their work, Repenning and Basawapatna defined the "Flow of Inspiration" principles necessary for successful social computing environments [Rep09; BR10; Bas10]. An environment adhering to these principles should:

- Display projects in a public forum
- View and run fellow students' projects
- Provide feedback on fellow students' projects
- Download and view code for any project
- Provide motivation for students to view, download, and give feedback on fellow classmates' projects

Repenning was inspired by students' interest in YouTube and Facebook to create a socially-oriented system called Scalable Game Design Arcade where students could create, maintain, and share living copies of their projects with others, as opposed to working on individual assignments discarded upon completion. In this system, students used a visual programming language called AgentSheets to develop and share simple copies/variations on classic video games. These students could leave feedback and ratings on each others' submissions, and could borrow ideas from other players' projects through a process similar to remixing. The Scalable Game Design Arcade formalized the sharing process to help track the flow of ideas between peers and discourage plagiarism. Students could download the full source of a peer's project, modify it as they like, and re-upload it as their own.

MONSAKUN [HK11] is a system that facilitates problem-posing for elementary arithmetic problems. The system was designed to automatically assess problems posed by students and provide feedback to help correct misconceptions. These tasks are logistically challenging for teachers, particularly when an entire classroom of students is participating. In MONSA-KUN, students are given a solution step (such as "7 - 3 = ?") and tasked with writing a word problem whose solution contains the given step. In order to build the word problem, students must arrange segments like "Tom has 3 erasers" or "Tom buys several pencils". If the resulting problem does not fit the provided solution, the system suggests changes the author can make to fix it (i.e. replacing one of the segments.) The authors evaluated their tool with 39 middle school students who used MONSAKUN over 8 lessons in 13 weeks. Students were given pre- and post-tests with problem-posing situations similar to those presented in the tool. The authors found that students using their system created fewer overly-simple problems, and that the system improved the test scores for low-performing students.

Animal Watch [AW03; BB08] is a pre-algebra tutor with an exotic animals theme. The tutor covers topics such as finding average, median, and mode, and conversion between units. Even though the tutor contains about 1000 developer-authored problems, the system can "run out" of appropriate problems to give a student, especially if those problems are selected problems to match a student's proficiency. The developers studied student content creation in two studies: a pilot in 2003 and a follow-up in 2008. The pilot investigated student attitudes towards problem posing. This initial pilot had only six students. Students were excited about sharing content with their peers, and proud that their content would be online and accessible to others. At the same time, students felt that they didn't learn much from the activity, and felt that it was easy once they got started. Additionally, the students were using an authoring interface meant for teachers, with little in the way of scaffolding or support. The authors provided examples of buggy student-created problems, such as one problem intending to calculate "175 + 175" but that asked "If one eagle can fly 175 miles an hour, how fast can two fly?" These types of problems illustrate the need for support during the authoring process, and filtering before created problems are shown to other users.

To further investigate this, a new interface was designed and a follow-up study was performed in 2008 [BB08]. In the follow up study, 24 students enrolled in a STEM summer camp created problems based on a worksheet. Researchers then asked them to solve and rate the problems created by their peers. Based on the success of the paper intervention, a module was added to the AnimalWatch system to facilitate problem posing within the system. Using this module, students could select data about any species in the system, choose the math skills involved in the problem, and write a problem using the data and skills they selected. Students enjoyed the activity and felt that it helped them learn, and were able to create usable problems with little expert modification. However, most of the student-authored problems were below the level of the concurrent classroom work. For example, some student problems focused on simple whole-number arithmetic rather than the rational numbers and pre-algebra they were learning in class. The authors suggested that additional scaffolding might be needed to help students create problems for topics they are currently studying or struggling with. There was no formal evaluation of learning gains with respect to the authoring activity in this study.

Later work by Carole Beal involving a system called "Teach Ourselves" investigated these effects further [BC12]. Teach Ourselves is a web application targeted at middle-school math and science students, incorporating aspects of gamification and social/"Web 2.0" community features. Players can earn rewards for solving and creating content, the rewards are displayed on a leaderboard, and users can get "+1" ratings from peers for creating problems and hints. In the study, problems created by participants were of usable quality, with an average quality score of 7.5/12 on a scale developed by the system designers. A total of 132 students used the system over a 90-day period,

and created an average of 5.2 problems each. The teachers using the system observed increased student motivation, and believed that the system encouraged higher-order thinking.

Even very simple problem-posing interventions have been shown to be more effective in a social or gamified framing. Chang showed that when students' problems from an earlier problem-posing activity were to be used later in a simple quiz game, low-performing students learned more from the activity, and students in general reported higher engagement [Cha12].

Many successful Serious Games include content creation as a feature, if not a core element of the gameplay. For example, Armor Games' Lightbot [Yar08] contains a level editor that allows players to create and share simple programming problems [Gou13]. In work with Bead Loom Game, Boyce et al. showed that including puzzle creation as another path to the game's core concept enhanced motivation for players less interested in competition [Boy11]. Allowing users to create their own challenges is a very powerful motivator, according to research on design patterns for educational games identified by a team led by Plass at Microsoft Research. She wrote that effective educational games take advantage of the fact that "constructing things is fun and helps learning," and that "a social component (collaboration, competition) makes games fun/engaging [PH09]."

The idea that constructing things is useful for fun and learning did not originate here, nor with Papert, but his definition of constructionism (as opposed to Piaget's constructivism) and the groundwork laid out in LOGO and carried forward through StarLogo and Scratch among others [Res96; Res09] embodies the idea well. In his own words, "Constructionism shares contructivism's view of learning as building knowledge structures through progressive internalization of actions [but] 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" [PH91].

2.3 User-Generated Content in BOTS

My approach with BOTS differs from existing approaches to scaffolded or gamified content-creation in ITS and educational games. MONSAKUN provides expert-authored building blocks for problem posing, but lacks game-like constraints designed to implicitly encourage specific design elements. AnimalWatch and Teach Ourselves provide framing devices in the form of story/theme elements and social elements, respectively, but neither of these is integrated into content creation. Instead, the user interacts with the story and social elements before and after the content creation. In BOTS, game-like elements are integrated with content creation, rather than being wrapped around it.

Additionally, when reviewing research on mathematical problem-posing, I noted a general tendency for users to create content below their skill level, dealing mostly with skills they have

already mastered. Approaches that forced students to focus on the skills of interest were more successful, both at generating appropriate difficulty problems and at helping educators understand students' skill levels and misconceptions. BOTS provides students with gamified constraints *during* content creation that encourage creation of specific patterns of game elements which afford and reward the game's core mechanics. This will result in higher quality content which better addresses the game's learning objectives.

Chapter 3

BOTS: A Serious Game For Novice Programmers

3.1 Context

This chapter describes the game design and specific gameplay mechanics in BOTS. Later chapters all use the same gameplay log format and definitions given in this chapter. Here I present my pre-pilot work that examines the first batch of user-generated levels to find common patterns in their designs. These findings motivated game design changes, as well as my first research question Q1 and hypothesis H1:

- Q1: How can non-expert student authors create high-quality levels that afford and reward the use of core mechanics in BOTS?
- H1: Asking authors to provide solutions to their levels before publication will increase the quality of published levels and reduce the number of submitted problems with trivial, tedious, or non-existent solutions.

3.2 Introduction

BOTS is a programming puzzle game designed to teach fundamental ideas of programming and problem-solving to novice computer users. The goal of the BOTS project is to investigate how to best use community-authored content, also known as user-generated content, within serious games and educational games. BOTS is inspired by games like LightBot [Yar08] and RoboRally [Gar94], as well as the success of Scratch and its online community [Ker08]. In BOTS, players take on the role of

programmers writing code to navigate a simple robot around a grid-based 3D environment, as seen in Figure 3.1. The goal of each puzzle is to press several switches by placing objects or the robot on them. To program the robots, players use simple graphical pseudo-code allowing them to move the robot. Players can also repeat sets of commands using loops, and re-use entire chunks of code using functions. Each puzzle can limit the maximum number of commands, as well as the number of times each command can be used. For example, in the tutorial levels, a user may only use the "Move Forward" instruction 10 times. Within each puzzle, players' scores depend on the number of commands used, with lower scores (and a lower number of commands to solve the same puzzle) being preferable. This encourages players to determine how to program the robot more efficiently. For example, it will be more efficient to make the robot walk down a long hallway by using a loop to repeat a single "Move Forward" instruction rather than to simply use several "Move Forward" instructions. The game's constraints and scoring are meant to encourage players to re-use code and optimize their solutions.

In addition to the tutorial / "Story" mode, BOTS also features a "Free Play" mode, with a selection of puzzles created by other players. The game, in line with the "Flow of Inspiration" principles outlined by Alexander Repenning [Rep09], provides multiple ways for players to share knowledge through authoring and modifying content. Players are able to create their own puzzles to share with their peers, and can play and evaluate friends' puzzles, improving on their own past solutions. The goal of this approach in BOTS is to create an environment where players can continually challenge their peers to find consistently better solutions for increasingly difficult problems.

3.3 Gameplay

In BOTS, players solve puzzles by writing simple drag-and-drop programs to move a robot. The goal of each puzzle, similar to games like "Super Crate Box" and "Sokoban," is to move boxes onto their corresponding switches. The score on each puzzle is based on the number of instructions used to solve it. Using fewer instructions earns a better score and a higher-quality medal. The robot programs are written using basic movement instructions like "Move Forward," "Climb Up," and "Pick Up Box." These commands provide the basic functionality needed to solve levels. However, since players are scored based on the number of instructions they use, players may also use Loops, Variables, and Functions to reuse parts of their code and get a better score. Based on how their solution compares to the current best solution, players earn Platinum, Gold, Silver, or Bronze medals shown on the level select screen, as shown in Figures 3.2 and 3.3. In order to improve on the naive solution (the shortest step-by-step list of basic movements needed to complete the level) players must use more advanced programming concepts like Loops, Variables, and Functions. The implementation of

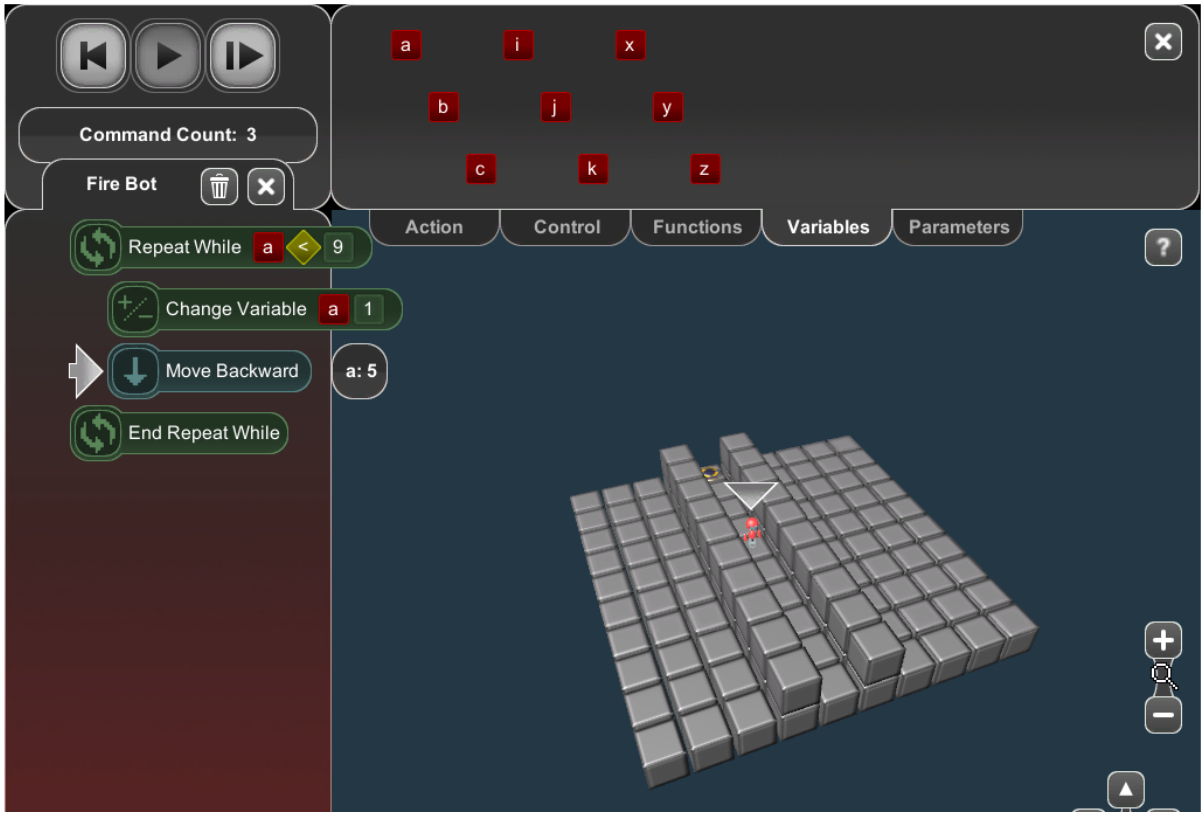

Figure 3.1 An example puzzle in BOTS. On the left is the robot's program. A pointer indicates the current instruction being executed and a window describes the current state of the variables in use. Along the top of the screen are the controls for the game, as well as the toolbox containing the instructions. In this level, the robot must move backwards down a hallway. The most straight-forward solution requires nine instructions, but this user has managed to use only 3.

loops in BOTS is based on the implementation of "while" loops in Scratch or Snap. BOTS uses "while" loops instead of Java or C style "for" loops or Scratch-style "repeat n times" loops because they are simple to construct (needing only a single conditional statement) and require the use of variables (another target concept). Variables in BOTS are scoped to the robot. Since there is only a single robot in each level, variables are effectively global. The primary function of variables in BOTS is for use as counters in loops, but variables can also be used to change the way other code works. For example, a student may create code that walks forward n steps, and may re-use the code several times, changing the value of n each time. Functions in BOTS are snippets of code that can be called from other points in the program. Functions are technically subroutines, as they do not currently support parameters. The primary use of Functions in BOTS is code re-use; if a segment of code is

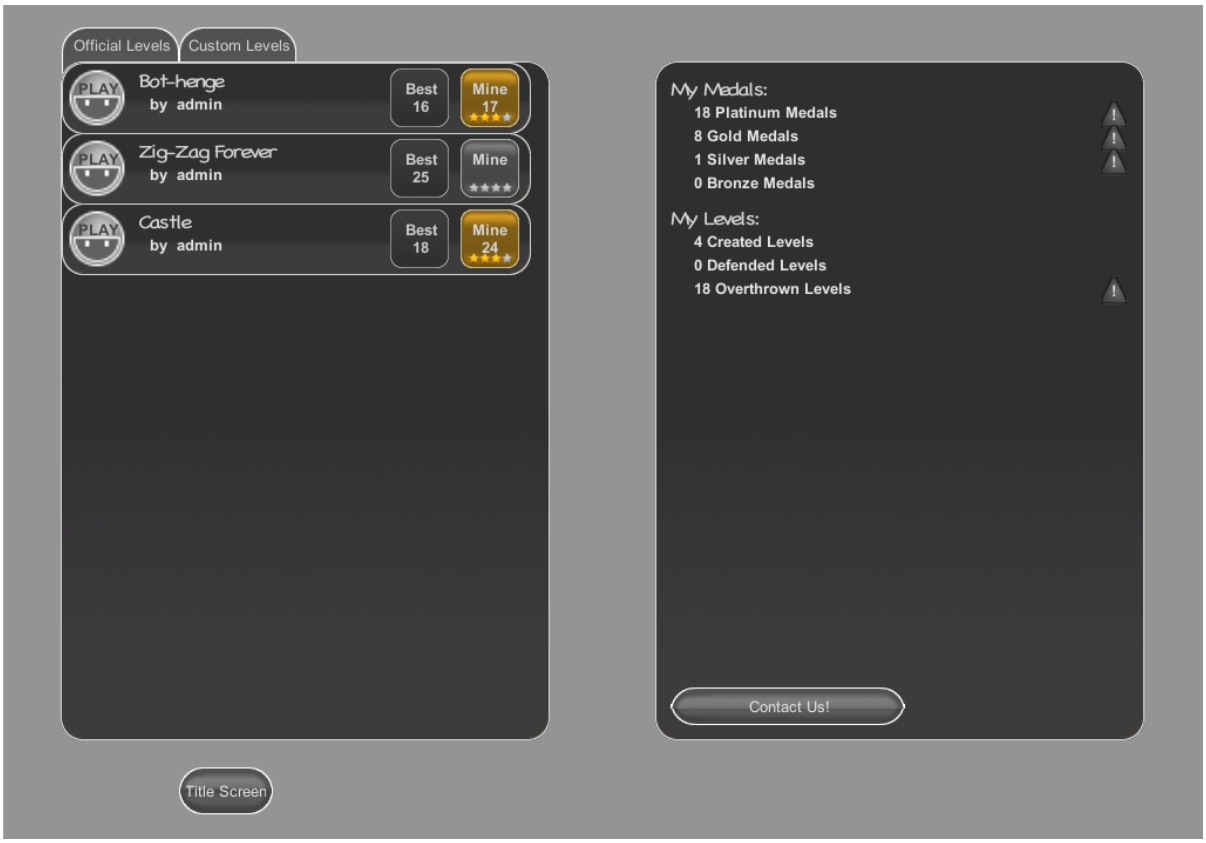

Figure 3.2 Next to each level, the current best score is displayed, as well as the player's best score. Based on how well their score compares to the ideal, players are assigned various medals.

used multiple times within a program, the user may be able to reduce their total lines of code by moving those lines to a function and calling the function instead.

Within BOTS, users are able to create new levels as well as solve levels other users have created. After a user completes a level, the solution is scored with Platinum, Gold, Silver, or Bronze based on their comparison to the current 'best' solution. This means that if a better solution is found, users' medals may decrease in value, encouraging them to revisit the level and discover more optimizations for their programs.

3.4 BOTS - Content

BOTS is designed to teach the following computer science or "computational thinking" skills using a more general problem-solving approach [BR12].

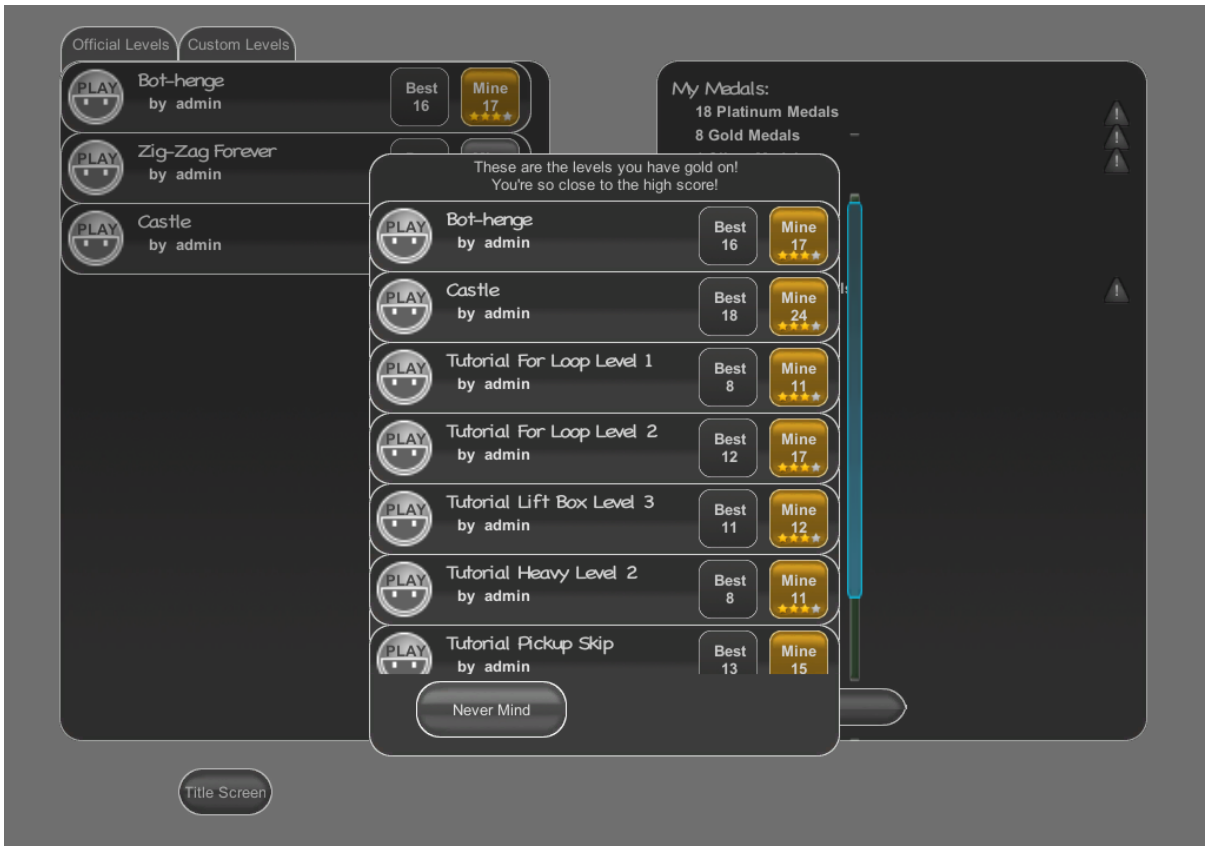

Figure 3.3 Clicking the prompts on the right side of the level editor brings up additional listings of levels. Players looking to improve their overall medal count may find new levels to beat here.

- Loops and Iteration (know what the arguments are to a for loop, know how to use a for loop to repeat a command or set of commands, and know when it is appropriate to use a loop to solve a problem)
- Functions and Code Re-use (know when it is appropriate to create a function, notice opportunities for code re-use)
- Variables (know how variables work with loops, know how to use a variable to produce different results with the same code)
- Optimization (notice opportunities to make a solution shorter, and given a solution or set of instructions, point out ways to make the instructions simpler and cleaner)

BOTS also has the potential to be used to address the following concepts, and they can be practiced within the game, but they have not yet been integrated into the game's content.

Figure 3.4 The basic level editor interface in BOTS, featuring one of the available skins. Players click the edges of blocks to add new ones, using an interface similar to the popular game Minecraft. Players drag objects out of the toolbar to add them to the level.

- Conditionals (levels can have multiple parts that must be completed independently)
- Recursion (functions can be nested recursively but there is little reason to do so. With different types of puzzles this behavior could be made more useful.)
- Parallel Processing (levels can contain multiple robots, with each executing a part of the instruction set)

3.5 Pre-Pilot Methods

To begin my investigation into the use of user-generated content with this game, I deployed BOTS as part of a STEM workshop during multiple sessions over June 2011 at UNC Charlotte. The gameplay data collected was anonymized, so information about specific players or sessions is not available. As part of the workshop activities, participants had already used GameMaker and BeadLoom Game, other tools and activities that teach similar concepts (graphical programming and iteration, respectively).

During this study, each student was instructed to complete the tutorial section of the game, with guidance from the teachers. Afterward, students could complete "Free Play" levels designed by the game's designers, complete "Custom Levels" created and shared by peers, or create new levels of their own. Creating (or playing) custom levels was completely optional during this study. Students were instructed that the best newly-created levels would be considered for inclusion in the game's Free Play mode. After the activity's completion, I analyzed the created levels, made informal notes on them, and selected some levels for inclusion in the game.

During gameplay, BOTS recorded two types of user logs: the User Program Log (containing the instructions the user has selected for the program) and the User Interaction Log (containing all of the user's interactions with the GUI). These logs contained data about student solutions to existing problems. I analyzed this data to determine when and how often students used the more difficult commands such as while loops, function calls, and variable operations.

Overall, 15 students created 37 levels during the course of this workshop. To examine the created levels, I played each level, made informal notes on it, then marked whether or not the level should be kept for the next iteration of the game. After this process, I examined the compiled levels and notes, and identified common features.

3.6 Pre-Pilot Data and Discussion

Of the 37 levels created during this workshop, several exhibit common features, both desirable and undesirable. The most prominent are listed here. The notes on each created level and a recording of whether it contains each selected feature or was selected to include in the game's next iteration, are shown in Appendix B. Only 10 of the 37 levels were selected for inclusion in the next iteration of the game.

I begin my discussion with the negative features present in these levels, since BOTS should be designed to discourage these features. The most common unwanted feature in these levels, identified in 30 of the levels, is *irrelevant structure*. An irrelevant structure consists of terrain blocks

that the robot will not interact with. Instead, an irrelevant structure forms a shape such as a maze, castle, or skyscraper. In fact, these levels are often named after the structure they contain.

The second most common feature, identified in 29 levels, is *irrelevant objects*. These are extra boxes that are not needed for the solution. Most commonly, this occurs in levels with a single box and a single goal. Since the goal can be activated by the robot itself, the box is unused.

The next most common feature, identified in 14 of the levels, is *vision-blocking terrain* used to hide a goal, box, or robot from vision when the level loads. In this initial version of BOTS, the camera was centered on the robot, so hiding an object from vision in this way is an effective way to prevent completion of the level.

On a related note, 3 levels were identified as featuring *blank terrain* where no change was made to the basic 10x10 floor present in the base level of the editor.

The next three categories are closely related. 11 each of the levels were identified as either *trivial* or *impossible without loops or functions*. The version of BOTS used in this activity imposed limits on the number of lines of code and the number of times a single line of code could be repeated. Therefore, some larger levels were actually impossible to solve. The definition of trivial used here was initially vague, so during the secondary analysis of these levels, I adopted a more rigorous definition. A level is trivial if the level can be solved using fewer than 3 unique lines of code. Three levels were identified as *impossible or self-solving* meaning either the goal is unreachable, no goal is present, or the robot starts on top of the only goal. These levels cannot be played.

Finally, only 13 of the created levels were found to contain *patterns or visual cues* for places where loops or functions could be employed. These levels account for 8 of the 10 levels selected to be included in the next iteration.

The high ratio of low difficulty content is expected. Previous work with problem-posing activities found that a relatively large proportion of generated content will be too easy, spam-like, unwanted, or unusable [Sil13; BB08]. This pattern also holds true for UGC-centric games such as Super Mario Maker [Nin08] and Little Big Planet [Med08]. Many criticisms of these games called out the large number of trivial, malicious, or irrelevant levels [McE15; Tho15; Her15].

As expected, I found similar patterns in BOTS. Under a free-form level editor, many levels exhibited creation of irrelevant patterns, structures, or objects. Only 13 of 37 created levels contained patterns that could be optimized, and 14 of 37 contained no potential for optimization whatsoever. However, some levels showed promise, featuring little extraneous terrain and few irrelevant objects, and highlighting areas for optimization by placing raised catwalks, walls, or repeated patterns of boxes and goals.

3.7 Initial Hypothesis and Pilot Evaluation Design

Based on the observed patterns I decided to implement some changes in the level creation process specifically targeting these negative patterns. At the start of this research, I sought to investigate methods of filtering low-quality content out of the publicly-available user-created content pool for an educational game. Quality within entertainment games is subjective. However, in an educational environment, core learning objective and game mechanics that afford learning are important. As observed with the first iteration of BOTS, players often submit content that fails to create affordances for the game mechanics. These kinds of levels include sandbox attempts to discover what they are allowed to build, attempts to create frustrating levels, or attempts to create multiple very easy levels and solve them quickly to earn points. If this is permitted, the pool of available levels will become flooded with levels that fail to address the learning goals and fail to provide a fun experience for their players. Therefore I have specifically defined the following qualities that a level must have in order to be considered "passable". These conditions are necessary, but not sufficient, for inclusion in the game.

First among these qualities is non-triviality. The best solution should not be an immediately apparent solution – a level's layout should require at least three different kinds of actions, with repetition. An example of a level from the initial corpus that fails this criterion is a long hallway with a goal at the end, since it requires only two actions. Another example is a level with a goal immediately adjacent to the player's start position. These trivial levels are unwanted because they are boring, they fill the list of levels, and they fail to address the learning goals of the game. Included in this category are levels with no goals, or no clear paths to a goal.

A second required quality is non-tediousness. The level must be completable in a reasonably short time, and have a solution that does not require an excessive number of commands. In BOTS, a level with many small irreducible tasks that are hard to simplify using a loop or function would fail this criterion. One example is a level with many straight hallways of different lengths where the lengths are not in an ascending or descending order that would facilitate using a loop. Another example of a tedious level failing this criterion is a level including scattered boxes and goals with no pattern to their placement. Both of these levels will take too long to complete, potentially causing players to lose engagement. In addition, since there are zero or few possible optimizations, these levels also do not address the learning goals of the game.

Finally, levels should be completable. While this seems obvious, the early version of discussed here BOTS features limitations other than just physical limits on where the robot can move. These limitations (e.g. specific numbers of each command, specific numbers of free spaces for selected commands, etc.) make it somewhat complex for a novice user to figure out whether or not a large

level can actually be completed. Presenting users with impossible challenges should be avoided by both game designers and educators.

The reasons players create these types of content are varied. Examples of some potential reasons may include: Players may be “sandboxing,” attempting to learn the level creation interface and posting their creations as an afterthought. Players could be “gaming the system,” creating simple levels and solving them to boost their place on in-game leader boards or for bragging rights. Players might be openly malicious “griefers,” developing levels to make other users’ experiences less pleasant. For the purposes of this study, the motive for creating such content is unimportant. What matters is that this content needs to be prevented from reaching other users, lest they become frustrated or lose interest in the game.

In order to determine a consistent measure of each student’s engagement with the BOTS game, I used the Game Engagement Questionnaire developed in [Bro09]. This survey consists of 19 items evaluating various aspects of engagement with the game, with participants answering “No," “Sort Of," or “Yes" to each item. This questionnaire was given to the students immediately after completing their BOTS workshop. This Game Engagement Questionnaire is included in Appendix A.

I also created a Demographic and Creation Process Questionnaire to be completed at the end of the BOTS workshop. These 9 questions focus on the player’s perception of the quality of the levels created by their peers. Participants are asked to rank on a 4 point scale how many of the other player levels they played were too hard, too easy, too boring, or impossible. Additionally, participants are asked questions like “Have you taken any programming classes, or have you used other games/tools for programming like Scratch, Alice, or GameMaker?" and “How many hours of video games do you play per week?", as well as asked to evaluate the difficulty on a 4 point scale of levels they played, and the difficulty of creating levels as they wished. I used this survey to determine each participant’s previous exposure to programming and video games, to gauge whether they enjoyed making and publishing their own levels, and to collect their perceptions of BOTS levels. The answers were used to identify any preferences the participants seemed to indicate for their submission technique. The full questionnaire is included in Appendix A of this document.

To analyze the levels created, I examined the created levels to identify the most common design elements (positive, negative, and neutral) present. I then used this analysis to identify specific, measurable criteria for judging levels and to inform the further development of the level editor. For each level, an expert involved with the game and pilot development (either myself, Veronica Cateté, or Monique Jones) played the level and graded it according to the rubric. Additionally, I recorded a short informal description for each level outlining the distinctive elements of that level and its solutions. Full descriptions and notes for each level created during this session are collected in Appendix B.

3.8 Pilot Methods

To evaluate the hypothesis that self-moderation does not harm participation when compared with no moderation and admin moderation, I conducted a BOTS workshop with middle school students enrolled in the SPARCS after-school technology program during November of 2012. SPARCS (Students in Programming, Robotics, and Computer Science) is a technology education initiative in five middle school locations across North Carolina to encourage student development and spark interest in fun and engaging computing topics. Workshops are hosted at the school sites on 1-2 times per month to continually introduce students to new technology software and tools. The overall goal of the SPARCS program is to help foster creative problem solving skills, abstract thinking, and an interest in computing especially for females and underrepresented minority students in hopes that they would be encouraged to pursue Computer Science courses in high school and beyond.

The participants were 9 students in grades six through eight who were enrolled in the weekly SPARCS program at Centennial Campus Magnet Middle School, whose parents completed the BOTS parental consent form.

Participants were divided among the three test conditions at account creation based on the modulus of the numeric ID their accounts were assigned. 3 students were assigned to the "no moderation" group, 3 students to the "Self-moderation" group, and 3 students to the "admin approval" group. After creating an account, players were directed to the in-game tutorial that leads them through the game interface, explaining the object of the game and the various commands available within the game using a series of levels. At the end of the tutorial period, players were asked to exit the tutorial and enter free play mode, where they either created a level or played an already created level. Players were then allowed to explore the game in Free Play mode for the remainder of the activity period. Finally, I distributed the demographic questionnaire and the GEQ at the end of the class and collected them before students left for the day.

I compared the average number of levels created and submitted by users across groups, to evaluate whether or not the test condition had an effect on the number of levels that users created or submitted.

3.9 Pilot Results And Discussion

Although the sample size is too small for statistical analysis to detect significant differences, this pilot was used to begin devising simple design changes that could be used to limit undesirable game content. Descriptive statistics for this pilot are presented in Appendix B. Though I am unable to perform statistical analysis on this small sample, there is a difference in mean quality score between

groups (No moderation $M_1 = 2.0$, admin-moderation $M_2 = 5.0$, self-moderation $M_3 = 4.5$.) The very small difference between the self-moderation and admin-moderation conditions leads me to believe self-moderation merits further investigation. This is critical because self-moderation can be implemented without a delay from level submission to publication. Even with such a small number of participants, our research team found that admin-moderation introduced a large source of delay, as levels continued to queue up for moderation while the moderators were busy. This both frustrated users whose levels were in queue, and users who wanted to play their friends' new levels. Therefore, I no longer investigated admin moderation after this pilot.

The data collected from the pilot are insufficient to draw a conclusion regarding my hypothesis that admin approval would discourage level submissions. For this group, the admin approval group had the highest number of total submissions as well as the highest number of published submissions. On average, students in group three created more levels than students in groups one or two, but there were no differences between groups in the number of published levels.

Several negative design patterns were generalized from the user-created levels which I will describe here. The most common negative design pattern (present in seven of the thirteen levels in this pilot) is visual obstruction. Players build high walls or deep pits to hide the goals of the level. In many cases, these obstructions interfere with otherwise well-built levels. The camera in this version of the game is forced to be centered on the player-character. Therefore, these obstructions can completely hide objects from the player. This can make a level appear to be impossible until the player moves the camera to the proper angle.

Another negative feature (recognized in seven levels) is the absence of a trivial solution due to simple length. In this version of the game, players have access to a limited number of instructions and a limited number of each instruction. A level can have no trivial solution, if solving it without loops or functions would exceed these limits. Such levels also tend to be tedious to complete even using an optimized solution. Therefore, this solution length also serves as a benchmark for tolerable length of gameplay.

Another common negative design pattern (identified in five levels) is the construction of a containing structure. The part of the level where gameplay happens is left largely clear. Walls or pillars are constructed surrounding that area, framing it. The structures in these levels typically look like houses or castles, and are often called out as such in level names or descriptions. They can also function as obstructions if they're built too large. In general these structures have no impact on gameplay, but in the worst case can actually distract or obstruct the player.

Seven of the levels were noted to contain visually clear patterns or repetitions. These exclude cases where optimization is possible but not visually called out. Compare a path with walls on either side with a path across an open flat plane. In opposition to this, five levels have objectives placed

"trivially" (very close to the start) or "randomly" so that no common patterns exist between how those objectives should be achieved.

From this design analysis of the created levels I identified several common elements that can be addressed by a straightforward change to the level editor; requiring creators to supply a solution to their own levels. I expected that tedious levels or levels with deliberate obstructions would be less likely to be submitted if their author had to solve them first. To investigate this, I collected further data between the self-moderation and no-moderation conditions for further analysis.

Chapter 4

Filtering Low-Quality Submissions

4.1 Context

This chapter investigates the potential for user-generated content (UGC) within BOTS, as well as the efficacy of having users solve their own created problems as a means of filtering out various types of low-quality submissions. This chapter extends work originally published in the proceedings of the Foundations of Digital Games conference in 2014 [Hic14b]. Since its initial publication, I have performed additional statistical analysis of these results, included here. This work addresses my research questions Q1-Q2 and hypothesis H1:

- Q1: How can non-expert student authors create high-quality levels that afford and reward the use of core mechanics in BOTS?
- Q2: How can level authoring tools designed using creative constraints help promote the creation of high-quality levels that afford and reward the use of core mechanics in BOTS?
- H1: Asking authors to provide solutions to their levels before publication will increase the quality of published levels and reduce the number of submitted problems with trivial, tedious, or non-existent solutions.

4.2 Methods

There are two parts to the methodology for this chapter. First, I examine existing user-created puzzles, in an effort to find patterns of unwanted UGC and hopefully identify common features that would make those patterns easier to identify and remove. Second, I implement a “solve-and-submit” mechanism into the game’s level editor, and conduct an experiment to investigate what effect this

mechanism has on the quality and quantity of created and published levels when compared with open submission.

4.2.1 Elements and Patterns of User Generated Content

While the quality of a game puzzle is subjective, I have developed a set of criteria necessary for a BOTS puzzle to be solvable and relevant to the game's core concepts of using variables, loops, and functions. These criteria are inspired by the use of design patterns in level design analysis in Hullett and Whitehead's work [HW10]. I developed these criteria by examining existing puzzles for BOTS and identifying common structures and patterns that exist across multiple puzzles. These patterns can positively or negatively impact gameplay. In this section I discuss why a puzzle creator could be motivated to create them, and how the game could incentivize or discourage puzzle creators from using them.

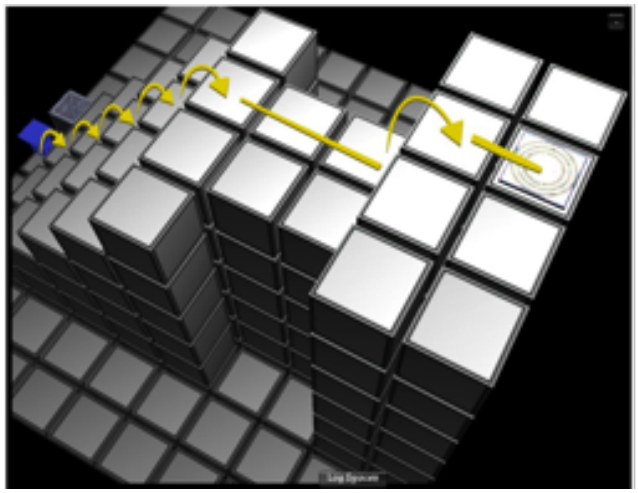

Figure 4.1 This puzzle has a clear trivial solution. The blue robot can climb up the steps and walk towards the target (as shown with the yellow arrows). There are also opportunities for optimization in the repeated "Jump" and "Move Forward" actions needed to move up the stairs.

The most important characteristic for user-created BOTS puzzles is the opportunity for puzzle solvers to use the core mechanics in BOTS. To satisfy this condition, the user-created puzzle must first have a *trivial solution* that requires only simple direct commands such as Move Forward or Turn. This allows users to solve the puzzle in a straightforward way. The puzzle must also have an *advanced solution*, a higher-performance solution that uses the game's advanced concepts of

Figure 4.2 The BOTS interface. The robot's program is along the left side of the screen. The "toolbox" of available commands is along the top of the screen.

iteration and functions.

As an example, 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. However, the straightforward solution still works. These constraints are meant to encourage players to re-use code and optimize their solutions.

This ability for players to approach puzzles simply at first, and then gradually increase proficiency with more advanced concepts is a hallmark of good game play. Good games balance ease with challenge to keep players in a flow state [Csi04] as players start as novices and increasingly become more skilled. If all puzzles have both simple and advanced solutions, then this increases replayability since players can master new skills and later revisit older puzzles to improve their performance.

In addition to possessing both trivial and advanced solution paths, a good user-created puzzle should contain *structural cues* that help guide the player between those solutions. In his 2009 GDC talk, "Puzzles as User Interface," Randy Smith [Smi09] stressed that in order for puzzles to challenge players without making them feel dumb, they should try to avoid "red herring" elements that are not actually relevant, while also re-using familiar structures accompanied by visual patterns to indicate where specific gameplay actions can be performed.

In the case of BOTS, these must be *obvious patterns in the placement of obstacles and objectives* that show the user where loops and functions would be best implemented. Wholly unnecessary elements can have the opposite effect, distracting users from the goal by presenting useless paths or unreachable objectives. A good user-created puzzle should make it apparent what the user is

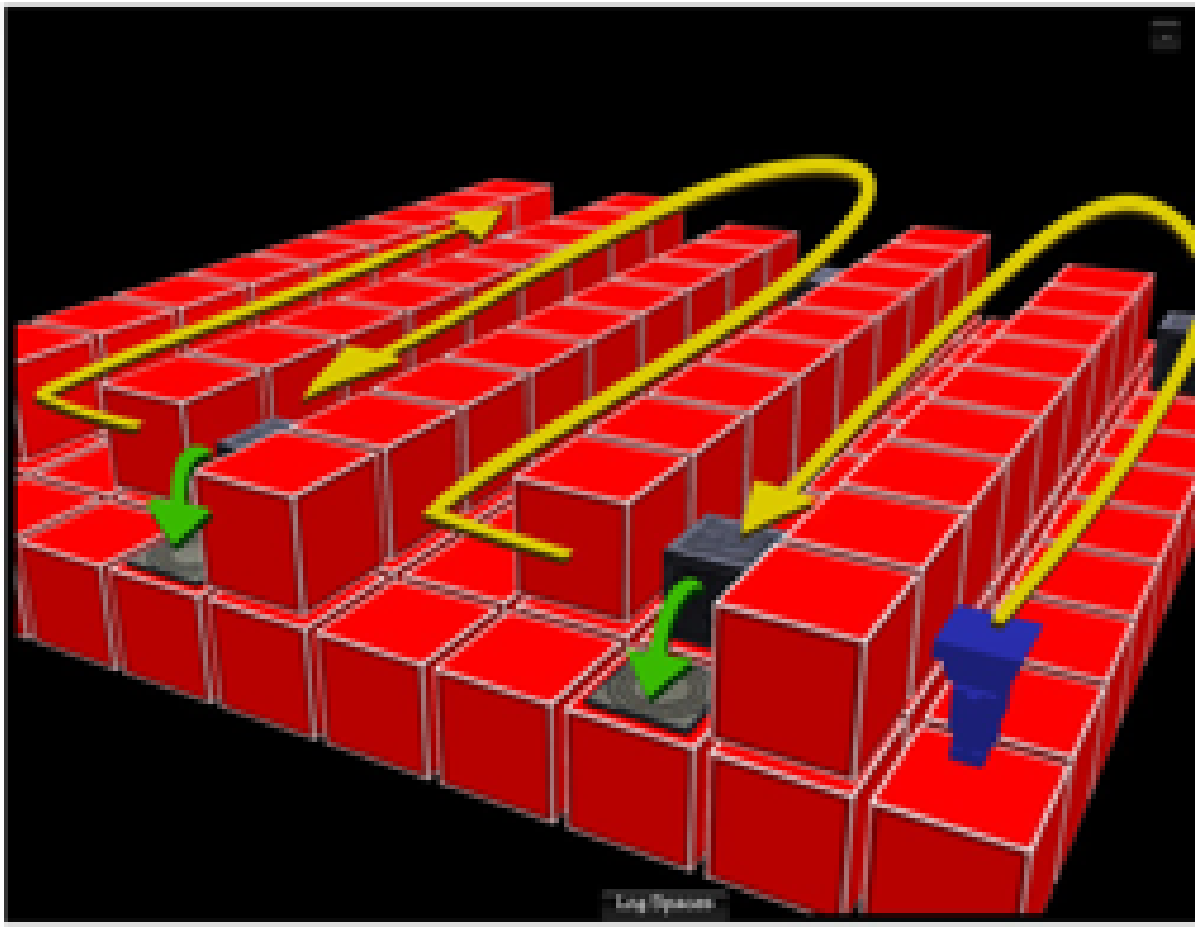

Figure 4.3 Puzzle with obvious structural cues for optimization of the solution. Hallways, stairs, or patterns of objects can be used to show where looping may be valuable, or where code can be reused. In this level, the first "leg" of the puzzle uses the same layout as the second "leg".

supposed to do using structural cues, and provide as *few unnecessary structural elements* as possible to avoid distracting the player from their goal.

Finally, a good user-generated puzzle should derive difficulty from changing the trivial solution into the advanced one, rather than from forcing the player to struggle through many repetitive actions. The advanced solutions should *not be tedious solutions* that try the user's patience rather than test their knowledge [Pla11]. That is, the difficulty should be related to the core mechanics of the game, rather than unrelated UI interactions. Consider a very long, completely featureless level in a 2D platforming game. The task itself is not providing difficulty; rather the interface itself has become an obstacle [JN09; Juu10]. Though this seems obvious it is clear when looking at existing user-authored content (both in BOTS as well as other UGC-centric games) that users often achieve difficulty in their levels by forcing these kinds of repetitive actions.

Using these criteria, I identified several categories of unwanted user-created puzzles to eliminate or discourage. For easier understanding I have also categorized these puzzles by the expected player type of the author using Bartle's player types: *killers, achievers, explorers,* and *socializers* [Bar96; Bar04]. *Killers* may create very difficult puzzles, hoping to frustrate the progress of others in a competitive way. *Achievers* might create simple puzzles, completing them in order to get their name on the high-score list. *Explorers* might play with the interface as a toy, and *Socializers* might use the interface to communicate with others, riffing off of the designs of other players or spelling messages with objects. Bartle also defined a sub-type of the killer group known as the *griefer*. Rather than creating difficult puzzles, a griefer might create intentionally impossible puzzles or flood the game with "spam" puzzles. These behaviors correspond well with the types of content that I have observed.

4.2.2 Puzzle Categorization

I describe the undesirable puzzles I have identified using four main categories.

Sandbox Puzzles

The first category of unwanted user-generated content is the *sandbox puzzle*. An example Sandbox puzzle is shown in Figure 4.6. Sandbox puzzles are characterized by the following traits:

Trivial or Non-existent solution The solution is straightforward once it is found and requires few loops or functions. Difficulty comes more from visually finding the correct solution than from programming the robot to complete it.

Distracting structural elements Elements such as terrain blocks or unnecessary objectives are

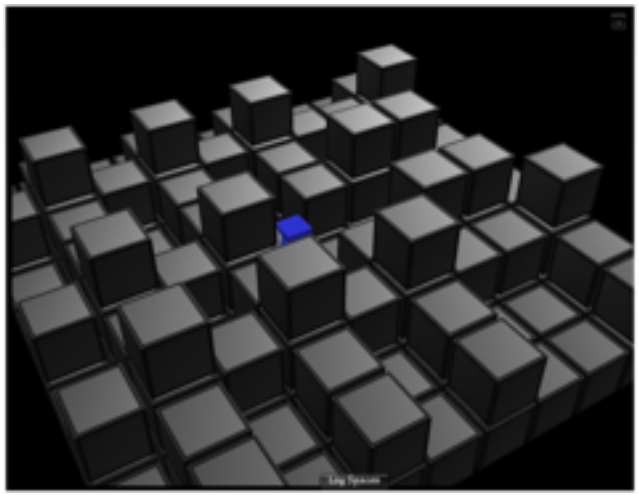

Figure 4.4 Puzzle with misleading or distracting structural cues. This puzzle contains irregularly placed blocks that don't lend themselves to any particular pattern. It is not apparent how loops or functions could intuitively help players with this puzzle.

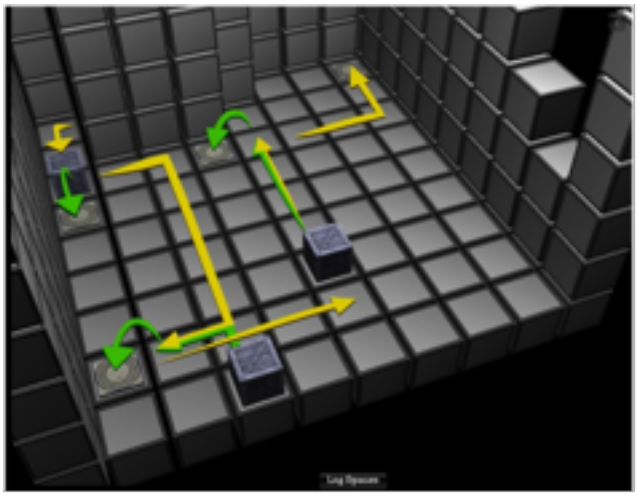

Figure 4.5 Puzzle with an obvious but tedious best solution. The robot (off screen) should move each block to the closest target (as shown with the green arrows) then stand on the target in the top right. However, there are very few iterative or repeated patterns that would allow a user to make their program shorter.

placed off-path where the robot will never interact with them during the course of solving the problem.

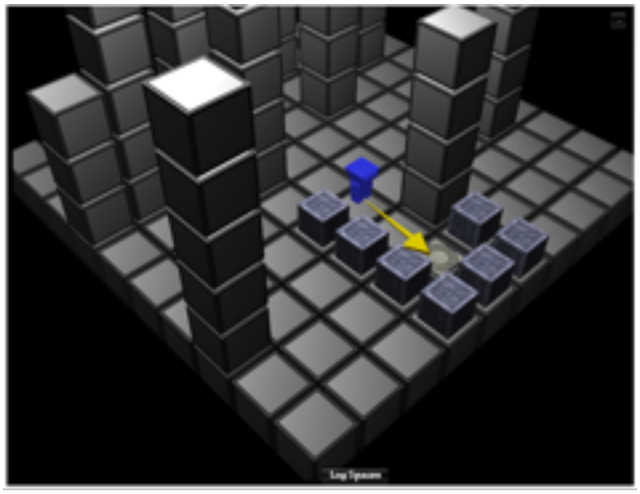

Figure 4.6 An example "sandbox" puzzle. The puzzle is simple to solve by moving the robot onto the visible target. The level contains many distracting crates and towers that aren't relevant to the solution.

Sandbox puzzles may be built by explorer players who are either new to the puzzle-creation interface, or socializer players who simply want to share something as quickly as possible. The solution (if there is one) is typically very straightforward. Structural elements are added for purely aesthetic value, spelling out words or forming shapes in the "unused" part of the puzzle. These puzzles are unwanted because such trivial puzzles do not address the core mechanics of the game. Additionally, these puzzles can be boring for players to solve.

Power-Gamer Puzzles

The second type of unwanted user puzzle is the *power-gamer puzzle*, such as the puzzle shown in Figure 4.7. Power-gamer puzzles are characterized by the following traits:

Tedious solution The straightforward solution requires a very large number of "lines of code" even when written by an expert. The solution may be entirely impossible due to limited space.

Trivial or Non-existent solution as described above.

Few repeated patterns Repeated patterns give players opportunities to optimize their solutions by

re-using code. These puzzles have no repeated patterns so each objective must be handled separately.

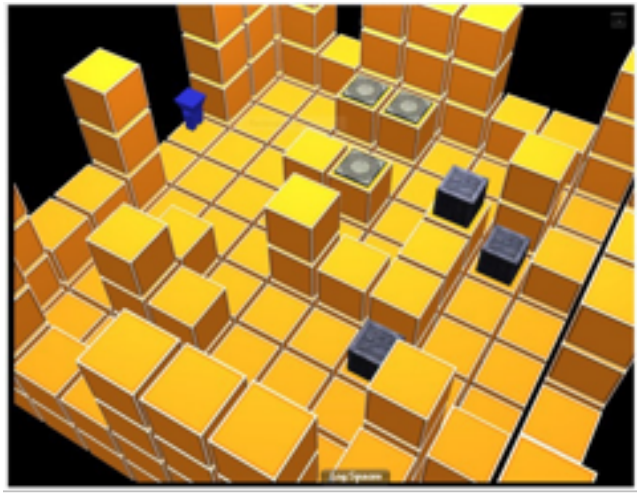

Figure 4.7 An example "power-gamer" puzzle. This puzzle is not impossible but requires over thirty instructions to build the most straightforward solution.

To create a difficult puzzle, griefer or killer players might seek to increase the time it takes to build a solution. This results in puzzles with conceptually simple solutions that take a long time to construct. These puzzles are undesirable because players may simply leave the game rather than solve them.

Griefer Puzzles

The third type of unwanted puzzle is the *griefer puzzle*, as shown in Figure 4.8. Griefer puzzles exhibit the following characteristics:

Distracting structural elements as defined above

Trivial or Non-existent solution as defined above, but usually the puzzle is obviously impossible because the robot is trapped, or the goal is inaccessible

Subversion of game mechanisms players use the puzzle title to communicate, spell words using objects, or block the player's view of the puzzle with objects or terrain

Griefer players create griefer puzzles in an effort to get other users to waste time on them before noticing that they simply cannot be completed. Griefer puzzles are distinguished by the perceived (and often communicated) intent of the designer to make them impossible.

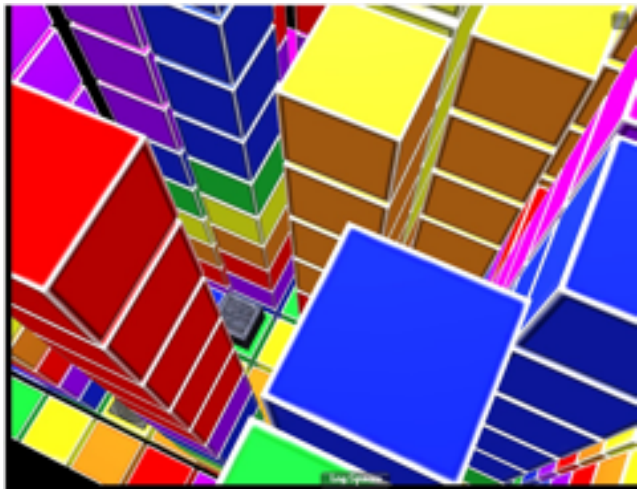

Figure 4.8 An example "griefer" puzzle. The puzzle contains high walls around the robot, the crates, and the targets, making it difficult for another player to maneuver the camera to see what they're doing.

Trivial Puzzles

Finally, a fourth type of unwanted puzzle is the *trivial puzzle*. An example trivial puzzle can be seen in Figure 4.9. This type of puzzle has the following traits:

Trivial Solution as defined above

Simplistic Solution that requires fewer than 5 commands to solve.

These puzzles can usually be solved in one or two moves. Oftentimes the puzzles show a lack of understanding of the game's mechanics. For example, in a puzzle with one switch near the start and a crate placed at the end of a challenging maze, the author might intend for the player to go get the crate. However, the player can solve the puzzle by simply stepping forward onto the switch. This demonstrates that the author did not fully realize how the player could accomplish the level goal.

While the theoretical individual users' motive for creating unwanted pieces of content is useful for categorizing puzzles, my primary focus is on the qualities of the *content* not qualities of the *creators*.

My goal is to prevent puzzles with those qualities from causing frustration and disengagement for other players.

4.2.3 Hypothesis and Evaluation Design

I sought to discourage both the creation and publication of puzzles that were too easy, difficult, obscure, or boring. In order to do this without hand curation, or developing a domain-specific classifier to learn which types of content to filter, I added a constraint to the puzzle submission process.

After examining the previously created puzzles and grouping them into the categories above, I identified the following *desirable qualities* for a puzzle to have:

The puzzle SHOULD be solvable.

The puzzle SHOULD have a straightforward solution. Players should be able to make progress towards completing the puzzle without having to use functions or loops.

The puzzle SHOULD contain opportunities for optimization. The straightforward solution should contain repeated patterns that can be encapsulated by functions or simplified by loops.

The puzzle SHOULD contain structural cues that make it obvious that optimization can be used. Hallways with raised walls highlighting the section to be repeated, or repeated patterns of crates and switches are good examples.

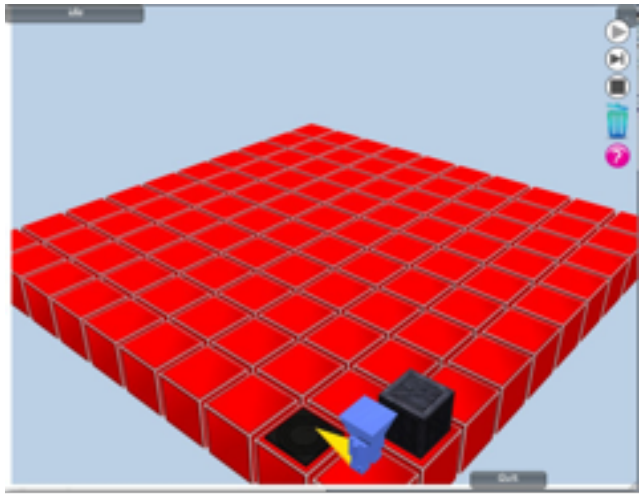

Figure 4.9 An example “trivial” puzzle. Even though the puzzle contains a crate, it is not necessary for the player to pick it up. The player can simply step forward onto the goal.

The puzzle SHOULD NOT contain unnecessary structural elements. Crates or terrain that are not part of the puzzle should be minimal.

The puzzle SHOULD be possible to complete in less than 5 minutes for an expert player. That is, the actual interactions (dragging buttons, creating functions) should take less than 5 minutes without accounting for time spent analyzing the problem. (5 minutes is an arbitrary time limit, but I seek to prevent tedium by limiting this time).

Based on these quality criteria I devised a change to the puzzle submission and publication process to reduce the number of low-quality puzzles submitted and published. To evaluate the change, I examined levels created under two conditions, described below.

Condition 1: Open Puzzle Submission (Control)

Under this condition, there was no filtering process in place and the puzzle was made public and immediately available for play as soon as the participant submitted it. As there was no moderation on this group, I anticipated a pool of user generated puzzles that would not meet the desired design criteria.

Condition 2: "Solve and Submit" (S&S)

With all four types of unwanted content (sandbox, power-gamer, griefer, and trivial puzzles) the primary problem is that a player may be unable or unwilling to complete the puzzle. This could be due to the puzzle being impossible, seeming impossible, being too long, or simply being boring. Thus, I hypothesized that *requiring authors to solve their own puzzles as part of the submission process* would cut down on the number of unwanted puzzles submitted and published. If an author created something that was unpleasant or impossible to solve, I expected it to be unlikely that they would go to the trouble to solve it themselves.

I expected that the total number of published puzzles would be somewhat reduced in the Solve & Submit condition, because submission required additional time and effort from the author. However, I also expected that the overall quality of the published puzzles would be higher for both submitted puzzles and published puzzles because the authors of *griefer* and *power-gamer* puzzles would have less incentive to create multiple unwanted puzzles if other players would never have access to them.

4.2.3.1 Study Design

To evaluate the impact of condition on the number and quality of puzzles submitted in BOTS, I conducted additional outreach activities using the BOTS game with two groups of middle-school age students. The purpose of these additional activities was to collect more user-generated content, in both the original open condition as in the Pilot study, and in the new Solve and Submit condition. As

with the Pilot activity, participants in the additional activities were enrolled in STEM-related weekend day camps and thus were self-selected to participate in computer science activities. However, students in one group were required to maintain a B average or higher in order to participate while the other group had no such requirement. The age ranges for both programs were the same, with students in 6th through 8th grades (ages 11 - 13).

As in the Pilot Study, each game session was 90 minutes long. Players were required to first play through the tutorial up to a cut-off point. Once players reached that point they were free to enter the game's "Free Play" mode where they were able to create and play custom puzzles.

Participants were assigned to one of the two conditions based on the modulus of the numeric ID assigned to their user account in the game upon creation. All players played through the initial guided tutorial portion of the game. Upon tutorial completion and at the start of the free play a pop-up window explained the requirements for the player's level to be published under their assigned condition. Each player could only see puzzles published by other players in their same test condition. Players were shown by the group moderator how to create a puzzle and how to play other users' puzzles. Players could then choose to spend the rest of their time in the game however they wished.

Table 4.1 The number of middle school students in each group according to whether they were required to solve and submit (S&S) before publication or not (Open), and whether participants were limited to those with B-averages and above.

	No Grade Requirement	B Average Requirement
Open	8	5
S&S	9	9

After the session, a grader used the provided guidelines to score each puzzle on its desirable qualities and then classified each puzzle according to the categories defined above. Not all levels were graded by the same research team member. However, all graders used the same guidelines to score puzzles from each session.

4.3 Results and Discussion

To measure the different patterns of level creation present under each condition, I evaluated the levels created under each condition (Open and S&S). Note that the user-created levels contain those in the Pilot study and those in the additional activities described above, for a total of 47 levels for

analysis.

First, since participants were from two different groups I checked to see if there was a difference in quality score of created levels between the participants with no grade requirement ($N = 21, M = 3.33, SD = 2.33$), and participants with a grade requirement ($N = 26, M = 3.38, SD = 2.03$). The groups of levels were not significantly different by a Student's t-Test ($t(45) = 0.16 < 2.01, p = .93$). I next analyzed quality scores between the Open and S&S conditions.

Summarized below are the results of the expert tagging process for all collected levels. The table uses abbreviations as follows: Look Comp. (looks completable), Comp. (completable), Loop/Func (a solution can use loops or functions), Obv. Patt. (the visual representation of the puzzle makes use of loops or functions obvious), Naive Soln. (there is a straightforward naive solution that can be achieved in 5 minutes), Exp. Soln. (there is an expert solution that can be achieved in 5 minutes or less), and $M(q)$ is the mean quality rating across all of these categories.

Table 4.2 Number of levels with a particular tag, and mean quality scores, for each condition

	n	Look Comp.	Comp.	Loop/Func	Obv. Patt.	Naive Soln.	Exp. Soln.	M(q)
S&S	24	22	19	14	10	16	13	3.916
Open	23	15	14	10	7	11	7	2.783

My hypothesis was that levels created under the S&S condition would have higher quality scores than levels created under Open Submission. I expected this to be true for both published and unpublished levels. I used two different methods of evaluation.

For these results, the mean quality score was higher in the Solve-and-Submit condition versus the Free-Form condition. This effect was even more pronounced when only published levels were included. I performed a simple Student's t-Test to evaluate the differences between these conditions, whose results are presented below. Quality scores of *created* puzzles in the S&S condition ($M = 3.92, SD = 1.91$) when compared with those in the Open Submission condition ($M = 2.78, SD = 2.28$) were higher, though not statistically significantly so. ($t(45) = 1.85 < 2.01, p = .07, d = .54$). The value of Cohen's d here shows that scores in the S&S condition were more than half of a standard deviation higher. This is an initial indication that my hypothesis about submission promoting higher quality puzzles merits investigation. However, the quality score is a sum of the six conditions that are not of equal or interchangeable importance. Therefore, a comparison of means is not appropriate for drawing conclusions about the performance of each group. Instead, I analyzed the results using non-parametric statistics. First I used a Chi-Squared test to get an initial estimate of whether there

was a difference in distribution of scores between the groups. Note that the expected value in some cells in this test is lower than 1, and this may harm the strength of the test.

Table 4.3 Number of levels with a particular score for the two conditions. Chi-squared = 8.722, df=6. p=.1898

Score	0	1	2	3	4	5	6
S&S	2	1	1	7	2	4	7
Open	7	2	0	3	5	3	3

I then used Fisher's Exact Test to measure whether the variable of condition affected the distribution of levels (positive and negative) for each of the six criteria. Fisher's exact test is used to compare categorical variables when sample sizes are small, and is more accurate than a chi-squared test [Rou05]. The results of Fisher's exact tests for each classification are presented below:

Table 4.4 Looks Completable: p = 0.0363

	Yes	No	Total
S&S	22	2	24
Open	15	8	23
Total	37	10	47

Table 4.5 Completable via Naive Solution: p = 0.2124

	Yes	No	Total
S&S	19	5	24
Open	14	9	23
Total	33	14	47

These results indicate that levels created under solve-and-submit (S&S) were more likely to look completable. In fact this is the only effect among all of these categories to achieve statistical significance at the alpha <= .05 level. No significant effect was found regarding the length of the levels' actual solutions. For the conditions having to do with the learning objectives and affordances for them in terms of gameplay mechanics, I also found no significant effect. These results indicate

Table 4.6 Improved Via Loops or Functions: p = 0.3868

	Yes	No	Total
S&S	14	10	24
Open	10	13	23
Total	24	23	47

Table 4.7 Contains Obvious Patterns: p = 0.5469

	Yes	No	Total
S&S	10	14	24
Open	7	16	23
Total	17	30	47

that simply asking players to solve their own levels may have an effect on the low-end, preventing the creation of outwardly unplayable levels, but it does not seem to have a pronounced effect on players creating levels that enable, afford, and reward the use of advanced game mechanics such as loops or functions. This makes some sense, as the gameplay elements of BOTS were not yet incorporated into the level editor during creation. This Solve and Submit change to the level editor thus acts more like a filter for bad levels than a guide to creating good ones.

However, as mentioned earlier these results take into account all levels created in either condition, therefore the above results describe the effect of the mechanism on creators before the filter itself is applied. Next, I considered only the levels whose authors successfully completed a solution.

Among the 11 published S&S levels, the mean quality score was much higher ($M = 4.545$) than those simply created in the S&S condition ($M = 3.91$). I then compared the *published* puzzles only, using a t-Test between puzzles in the S&S condition ($M = 4.55, SD = 1.36$) and published puzzles in the Open Submission condition ($M = 2.78, SD = 2.28$). The test showed that the puzzles in the S&S condition scored higher ($t(32) = 2.36, p = .024$) by nearly a full standard deviation ($d = .94$).

However, since the scores were still only ordinal data, I again performed a non-parametric chi-squared test to measure independence between groups, and Fisher's Exact Test for each element of the score.

The patterns from earlier became more apparent when only published levels were considered. As expected, those dimensions related to the simple level completability had significant effects. The Solve and Submit condition had a higher proportion of levels that featured improvements via loops and functions, contained obvious patterns that call out where these improvements are possible, or had a reasonably-designed expert solution. However, these effects were not statistically significant. I did not expect to see strong effects here, since this Solve-and-Submit mechanism only prevented

Table 4.8 Naively Completable in 5 minutes: p = 0.2443

	Yes	No	Total
S&S	16	8	24
Open	11	12	23
Total	24	23	47

Table 4.9 Expertly Completable in 5 minutes: p = 0.1425

	Yes	No	Total
S&S	13	11	24
Open	7	16	23
Total	20	27	47

publication after a level had already been fully designed by its author, and was unlikely to influence the level design before or during its creation.

To ensure that my approach was not inappropriately filtering out acceptable levels I examined the levels completed but unpublished under the S&S approach. One such level is shown in Figure 4.10. Out of 10 acceptable levels created under this condition, 3 were unpublished, and in each of those cases, the level had a quality score of 6, meaning it had all the criteria outlined in Section 2.3. However, all of these levels had *tedious solutions*, taking a large amount of time to solve. Though these levels did not exhibit the other aspects of *power-gamer* levels (as outlined in section 2.2), I suspect that in these cases there was a large disparity between the author's current skill and the skill needed to find the solution. These levels were then too difficult for the author to solve. Were the author to return to the level editor after increasing their skill, these levels could be correctly accepted. However, as a result of the current Solve-and-Submit system the levels remain unpublished.

4.4 Conclusions

These results demonstrate that the "Solve and Submit" mechanism helps to prevent low-quality user-created puzzles from being published when compared to an open submission system. This method prevented some number of unwanted puzzles from being published, but did not improve the quality of created puzzles overall. Additionally, this mechanism also occasionally rejected puzzles that were acceptable but whose authors were currently unable to solve them. In later chapters, I investigate further modifications to the level editor and puzzle submission process that both reduce the number of incorrectly filtered levels and increase the quality of puzzles created and published.

In examining puzzles to create the evaluation criteria, I noted that many lower-quality puzzles

Table 4.10 Tags for levels created under each condition (with published levels separated out)

	n	Look Comp.	Comp.	Loop/Func	Obv. Patt.	Naive Soln.	Exp. Soln.	M(q)
S&S(p)	11	11	11	7	5	10	6	4.545
S&S	24	22	19	14	10	16	13	3.916
Open	23	15	14	10	7	11	7	2.783

Table 4.11 Number of levels with a particular score. Chi-squared performed with S&S(sub): 9.044, df=6, p=.1711

Score	0	1	2	3	4	5	6
S&S (pub)	0	0	0	4	1	2	4
S&S	2	1	1	7	2	4	7
Open	7	2	0	3	5	3	3

"fill the canvas" of the area presented in the level editor, resulting in a high number of structures or extraneous elements included in levels. However, most levels ranked as high-quality left some empty space. Therefore, after this experiment, I changed the starting canvas space from a large empty 10x10 floor to a single empty tile. When players require additional space, they must explicitly add tiles to the floor to build upon. This change discourages the canvas-filling that does not seem to appear in high-quality user-generated puzzles.

Table 4.12 Looks Completable: p = 0.0339

	Yes	No	Total
S&S	11	0	11
Open	15	8	23
Total	26	8	47

Table 4.13 Completable via Naive Solution: p = 0.0172

	Yes	No	Total
S&S	11	0	11
Open	14	9	23
Total	25	9	34

Table 4.14 Improved Via Loops or Functions: p = 0.4646

	Yes	No	Total
S&S	7	4	11
Open	10	13	23
Total	17	17	34

Table 4.15 Contains Obvious Patterns: p = 0.4590

	Yes	No	Total
S&S	5	6	11
Open	7	16	23
Total	12	22	34

Table 4.16 Naively Completable in 5 minutes: p = 0.0238

	Yes	No	Total
S&S	10	1	11
Open	11	12	23
Total	21	13	34

Table 4.17 Expertly Completable in 5 minutes: p = 0.2619

	Yes	No	Total
S&S	6	5	11
Open	7	16	23
Total	13	21	34

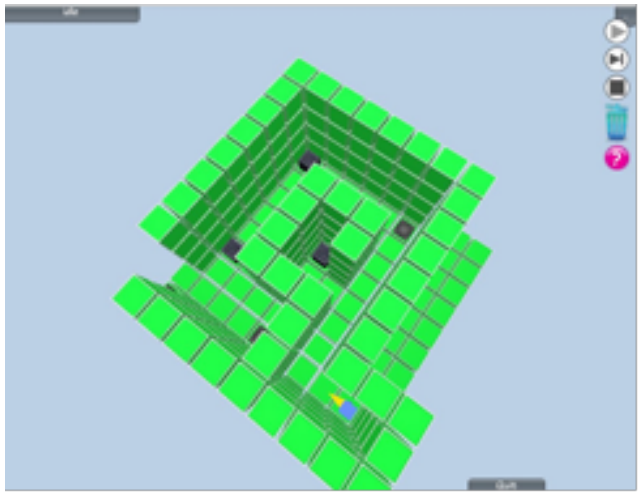

Figure 4.10 One of the acceptable levels that was not published. This level is difficult (and tedious) to solve without using loops.

Chapter 5

Generating Hints For A Serious Game

5.1 Context

This chapter explores data-driven methods for generating hints for new problems in BOTS. This chapter presents work published at the International Conference on Intelligent Tutoring Systems (2014) and Educational Data Mining (2014) [Hic14a; PI]. This work addresses my research question Q3 and hypothesis H2:

- Q3: How can existing data-driven methods for knowledge discovery and feedback generation from Intelligent Tutoring Systems be applied to brand new user-authored levels?
- H2: Hint Factory can be used to generate feedback for user-authored programming problems in BOTS.

With this research, I demonstrate that Hint Factory techniques can be adapted to generate next-step hints for BOTS. The ability to provide step-level feedback on unseen levels means that such feedback can be created for user-authored levels without expert intervention.

5.2 Introduction

The BOTS programming game does not have any mechanisms for personalized feedback or adaptive problem selection or ordering. One method of providing this type of feedback is to have experts create it for each problem. However, BOTS features open-ended problems with many possible solutions, as well as user-generated problems, making such expert annotation difficult. In this chapter, I describe an effort to incorporate ITS-like personalization through data-driven hint generation.

In BOTS, players first play a designer-authored sequence of tutorial levels, but are then able to create their own puzzles for other players to solve. Groups of students often create levels to challenge one another, and some students spend more time creating levels than solving levels in the game. In our experience, user-authored puzzles are most often played by users within the same peer group. Since puzzles are constantly being created, it is not feasible to have experts author detailed hints for these puzzles. To provide hints for new puzzles, BOTS needs data-driven methods, and due to the limited number of times levels will be played, it must be possible to generate hints with very little data.

I would like to find out if data-driven methods are feasible for user-generated levels, and if so, how many records would be needed to provide a high frequency of hints. I am targeting 80% hint coverage, so that each time a student tests their robot program, there should be an 80% chance that the system could provide a hint. Since user-generated levels are played sparsely and within peer groups, I am using the common single-player levels (including tutorial levels) as a test bed for this technique, with the goals of (1) proving the technique can work in this environment and then (2) estimating how many records would be needed to do the same for a user-generated level.

The contributions in this chapter are two-fold. In the first section, I generalize data-driven hint-generation techniques to an educational game, and introduce a novel approach to modeling student states for open-ended programming problems. In the second section, I use the novel world states method to determine the amount of data required to generate hints for new levels, addressing the *cold-start problem*. With these contributions, I demonstrate that the Hint Factory method is applicable to a game with user-generated content. It is my hope that future researchers can generalize these techniques to programming tutors for mainstream languages.

5.3 Hint Factory

In this section I describe the Hint Factory, a data-driven process designed by Barnes and Stamper [Sta08] to use student data to generate problem-solving hints for future students. Hint Factory can be applied in any context where student interactions with a tutor can be defined as a graph of transitions between states. States can be thought of as a snapshot of the current state of the problem solution attempt, while transitions are the actions users take to transform that state to the next observed snapshot. All of the states and transitions are combined into an interaction network. Then, the Hint Factory treats the interaction network as a Markov decision process to compute expected values for each state in the interaction network, with higher values for reaching a puzzle solution. A hint button is constructed, that computes the current state, and selects the edge from the current state that takes them closest to the solution. A hint template is also constructed to select information

from the transition and resulting state to present as a hint.

The success of the Hint Factory relies heavily upon choosing the right state representation (grain size) that breaks problems into small enough steps that can be used to provide hints, but large enough abstractions so that student behaviors have some overlap. The choices of grain size could be anything from detailed click-streams to final solutions; the correct grain size must result in an interaction network that allows for overlap but also provides meaningful hints. In this chapter, I first apply Hint Factory to well-populated levels in BOTS, using the player's code as the state representation. After I examine the resulting interaction networks, I compare them those created using a snapshot of the puzzle after the program completes as an alternate state representation. I compared these networks for overlap and how many of their states lie on solution paths. These are initial indicators that the Hint Factory can be successfully used for hint generation. In the first section of this chapter I show that world states have potential as the correct grain size for hint generation for BOTS. The second section of this chapter presents a formal cold start study comparing hint coverage, the likelihood that a hint can be provided at any given state, between the two methods.

5.4 Applying Hint Factory to BOTS

In this section, I perform a preliminary analysis of using two different types of states to apply Hint Factory to BOTS. These two types of states, discussed in more detail below, are code states and world states. For BOTS, either a student program (code states) or the resulting placement of the robot, boxes, and all targets (world states) could be used to represent states. These states are recorded in the interaction network as nodes. Edges correspond to user actions that change the problem-solving state (either modifying the program for code states, or having a different configuration of the robot and boxes for world states). The remainder of this section describes the application of Hint Factory to BOTS using code states and world states, and examines the resulting interaction networks for their potential for hint generation.

5.4.1 State Representation in BOTS

State representation is a critical part of hint generation. A poor choice of representation can adversely affect both the quantity and quality of hints given to the user. The intelligent tutoring system literature agrees on the definition of *interactions* as the low-level, click-by-click behavior of a student in a tutor [Sta10]. The most straightforward low-level representation of the student code is not ideal for this context, as the state space in BOTS is large and sparsely populated. As a simple example, consider the following programs, 5.1 and 5.2, that accomplish the same goal but have no overlapping

intermediate states.

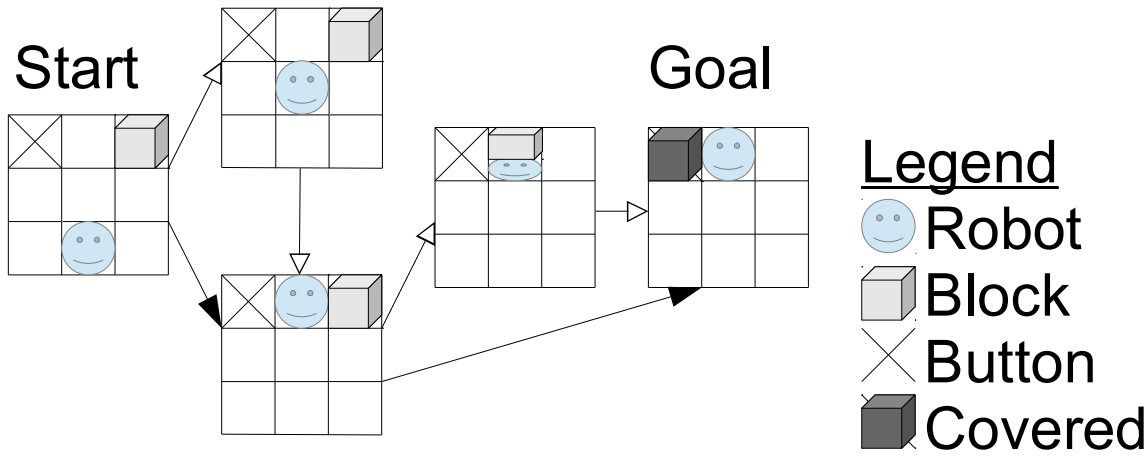

Figure 5.1 An example of the interaction network of a BOTS puzzle solved two ways by two students. One student (empty arrow) wrote and ran four iterations of their program to arrive at the solution, and the other (filled arrow) only wrote and ran two.

5.4.2 Interaction Networks

I used a data structure called an *interaction network* to represent the student interactions with the game [Eag12]. An interaction network is a graph where the nodes represent the states that represent snapshots of student solution attempts, and the edges represent transitions between states. In this research, I compared the effects of using two different types of states in the interaction network: *code states* and *world states*.

An Interaction Network [Eag12] is a graph-based representation of user activity in a puzzle or exercise, and is an extension of the Markov Decision Tree based structure described in the initial Hint Factory work [Sta08]. In an Interaction Network, vertices represent unique configurations of the environment. The edges between vertices represent the next actions taken by users who reached the current vertex, and are weighted with their frequency in the data set. This network is useful not only for providing hints but for analyzing the complexity of the problem and identifying unique strategies within individual problems [Eag14].

In previous work with Hint Factory and data-driven methods [Sta08; RK13; Jin12] the definitions

of states and actions used are narrow. For open-ended BOTS problems, using student code results in very little overlap between student states. Instead, I propose and study intermediate program outputs as another way of representing many similar programs with a single model that explains them all. In a study of 30,000 student source code submissions to an assignment in the Stanford Massive Open Online Course on Machine Learning, it was found that there were only 200 unique outputs to the instructor test cases [Hua13]. In the Machine Learning class, there was only one correct output, but despite there being an infinite number of ways to get the problem wrong, Huang et al. observed a long tail effect where most students who got the exercises wrong had the same kinds of errors. In this chapter, I compare the effects of using two different types of states, *code states* and *world states*, in the interaction network for hint generation for BOTS.

Code States are snapshots of the source code of a student's program, ignoring user generated elements like function and variable names. In Figure 5.3, the screenshot depicts the initial configuration on the left, and three distinct programs that all result in the same output on the right. *World States* instead use the configuration of the entities in the puzzle as the definition of the state. In other words, code states represent student source code while world states represent the output of student source code. In this representation, one "World State" would encompass all three programs. In both

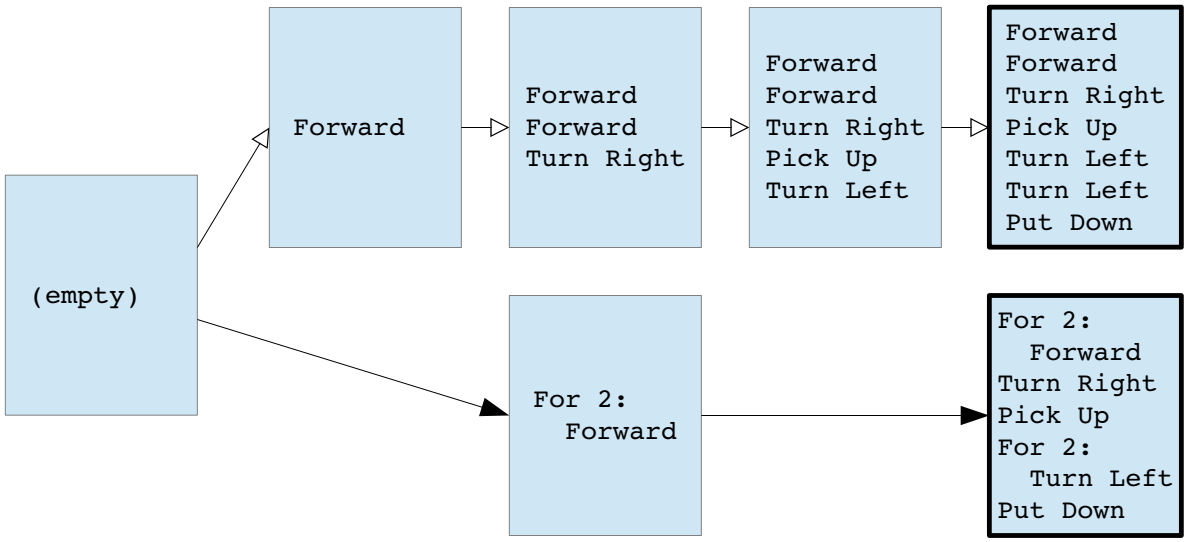

Figure 5.2 This figure shows the code from the same two students with the same solutions. Notice how in this example, these two paths that get to the same goal have no overlapping states - including the goal state.

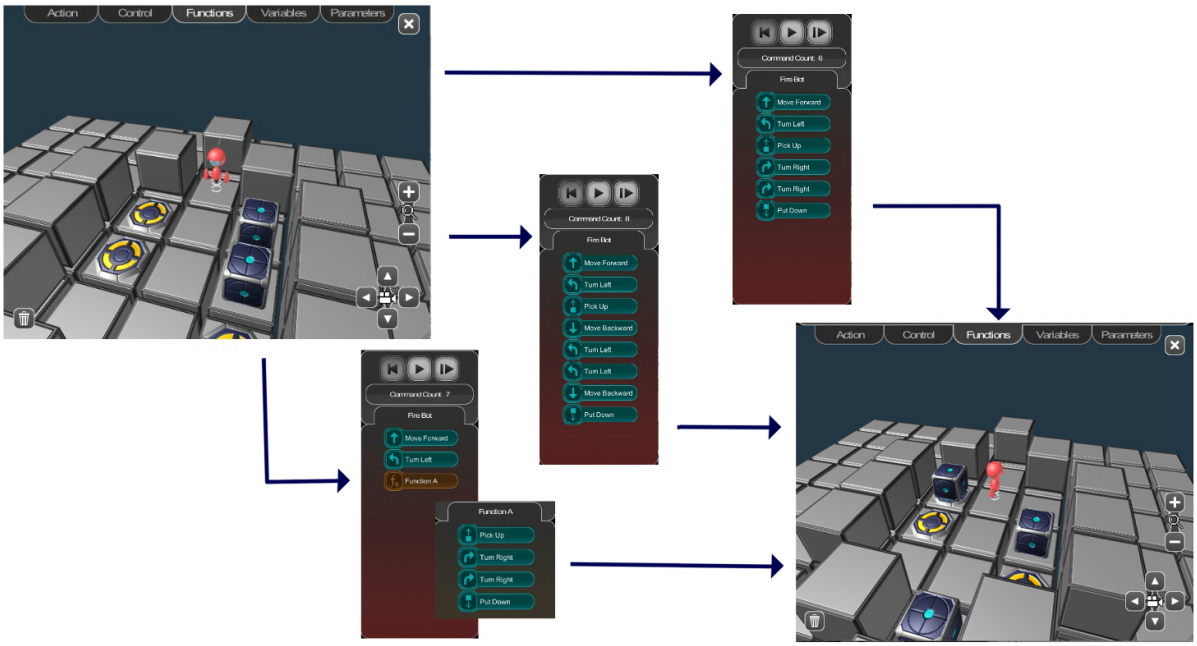

Figure 5.3 In the game, players direct a robot to solve puzzles using a simple drag-and-drop programming language. Here there are three different programs that all result in the same final state: The robot moves a block from one side of the room to the other. The programs correspond to code states while the layout of the puzzle, robots, boxes, and targets comprises a world state.

cases, student interactions can be encoded as an interaction network, enabling the application of the Hint Factory. Figures 5.1 and 5.2 demonstrate how the interaction networks differ between code states and world states. Using world states substantially reduces the number of states, increasing overlap, and reducing the number of programs needed for hint generation. Additionally, a state representation that meaningfully groups users by their behavior can help developers analyze player actions, similar to research on BeadLoom Game [Eag13].

The World State approach is somewhat similar to the approach taken by the developers of iList [Fos09], who compared the current state of a list being manipulated by a student to states created by past students. I represent the student state as the configuration of the stage, robot and blocks after the student's program has run to completion. For this work, I represent the output of a student's program as a grid representing the the stage, with unique markers for boxes, switches, and robots, as well as a height map of the stage. An example can be seen in Figure 5.4. This way, regardless of their programs, students who are performing the same actions (such as putting a particular block on a particular switch) are grouped into the same world state.

5.4.3 Hint Selection Policy

To select a hint, I first use the Hint Factory algorithm to assign scores to the states in the graph. First, each goal state is assigned a score of 100. Then, scores are iteratively back-propagated to their ancestors, with each step incurring a constant penalty. Choosing a hint for a student is a matter of matching the student's last observed state with a state in the graph, and selecting the child node with the highest score. This policy favors shorter paths to any goal state.

Hints are returned as simple state descriptions, indicating the objects located at each grid coordinate in the current puzzle. These hints are presented as potential next steps for the robot, indicating places to move the robot or places to place a block. The hints, as generated, do not contain information about how to program the robot; however, by looking at the program of an individual student within that state, it is possible to suggest interactions by comparing them with the user's current program.

It is important to note that my definition of states and transitions differ slightly for those in the original Hint Factory research [Sta08]. In logic, each transition is a single action, and thus has constant cost. Because this is true in tutors like Deep Thought[Eag14], it is reasonable to choose the adjacent state with the highest utility when offering a hint. However, this is not true in BOTS. Since it is possible for a student to solve a problem in one attempt, the hint given to a student who has just started work could essentially be, "Solve the problem." While there are some merits to showing a completed state as a hint for certain puzzles, it is likely that this hint is too general and too

large-grained to be useful. Despite these differences, I am still able to use the Hint Factory method to generate meaningful hints in most cases.

5.4.4 Data

I applied Hint Factory to the 24 puzzles with the highest number of player solutions. Player solutions consisted of completely anonymized gameplay data from all previous activities conducted using the current version of the game. These activities, listed in Appendix C, included the studies reported in Chapters 3 and 4, additional informal middle-school outreach activities, and one activity with students in a introductory college computer science course for non-majors. For formal settings, the procedures used were the same as those in Chapters 3 and 4. During informal activities, students may have played for more or less time if the BOTS activity was part of a set of activities. Appendix C contains the age ranges for the populations using BOTS in this data set.

While many participants created levels, these levels did not have many unique plays, due to being played mostly by users in the contemporaneous peer group, and very rarely by users outside that group. Since the common levels created by the game developers, including tutorial and challenge levels, have a far higher player count, this study used data from the common levels.

5.4.5 Analysis

As an initial indicator of the ability of the Hint Factory to compress student behaviors into an interaction network with overlap among students, I compared the number of unique programs written to the number of unique world states. I then considered the number of those states for which a hint was available, as shown in Figure 5.1. For the problems analyzed, the world state approach consistently reduced the size of the state space. For puzzle 10, a puzzle with a rich data set of solutions, I was able to reduce the state space from 325 unique programs (code states) to 130 unique world states. However, this reduction is meaningless unless useful hints are provided from the created states. Out of 130 unique observed states, 79 states had potential to generate hints; that is, a student visited that state and then correctly solved the puzzle. Thirty-three of these hints led to Error nodes (where the student gave commands that moved the robot out of bounds or into an obstacle), in cases where the Error is the only observed next-step. Of the remaining 45 hints, 42 were meaningful. It is important to note that while this problem contained more records and students than other problems in the data set, the number of records was still quite small. Despite the lack of data, the world state method was able to provide hints more than half of the time, and could do so for every state reached by multiple users. Therefore, I conclude that world states are more appropriate than code states are for hint generation. Additionally, using world states resulted in

Table 5.1 Results of the hint generation method for 24 single-player mode puzzles from the game. Rows indicate the Puzzle ID, number of students who attempted the puzzle, number of individual attempts, number of unique programs or code states, number of "hintable" world states, and number of unique world states.

Puzzle	Students	Attempts	Unique Programs	Hint-Generating States	Unique States
1	60	95	9	3	5
2	57	284	234	41	65
3	50	189	121	15	21
4	43	77	39	9	9
5	42	181	193	22	24
6	42	84	26	5	7
7	40	127	182	31	41
8	35	50	16	8	10
9	35	89	81	25	29
10	**33**	**227**	**325**	**79**	**130**
11	31	53	77	20	25
12	28	79	41	3	4
13	27	145	187	50	75
14	22	40	57	19	23
15	21	76	119	16	18
16	19	40	96	33	39
17	18	76	103	26	32
18	15	44	59	4	35
19	15	34	64	16	38
20	14	56	43	5	25
21	13	33	34	15	20
22	10	67	71	18	23
23	8	30	25	13	16
24	8	13	32	0	22

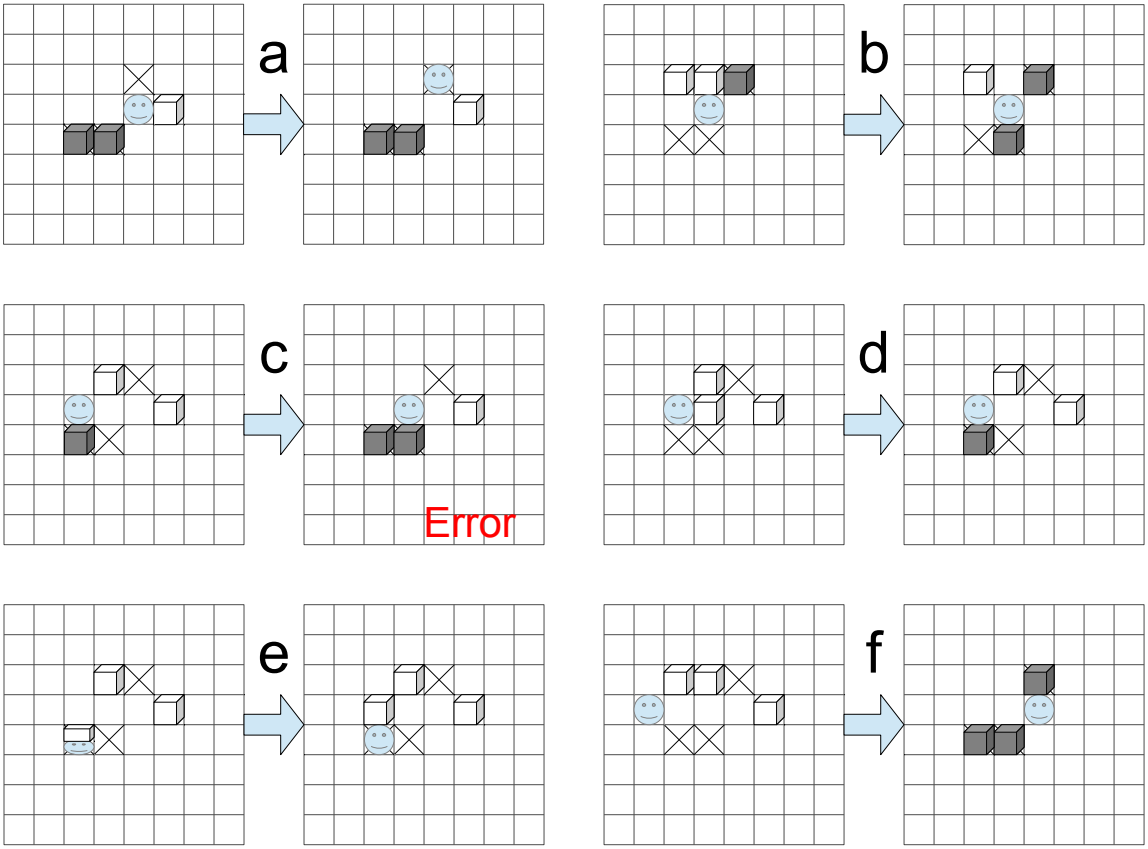

Figure 5.4 Six of the generated hints for a simple puzzle. The blue icon represents the robot. The 'X' icon represents a target where a box or the robot should be places. Shaded boxes are boxes placed on goals, while un-shaded boxes are not on goals. Hint F in this figure depicts the start and most common goal state of the puzzle. Each hint a-f shows a starting world state on the left and a suggested target world state on the right, and an arrow with letters a-f denote suggested transitions, of types a) rethinking, b) subgoal, c) error, d-e) correction, and f) simple solution as described in the text.

hints comprised of much more than a single changed instruction. This impacts the quality of the hints and how the system can present the generated hints.

5.4.6 World States and Transitions

When using world states, the transitions between states comprise more than a single changed instruction. In previous Hint-Factory work, the hint could be described as a recommendation for the student to take a certain action next. However, with world states, this interpretation is not applicable. To understand how world state transitions differ from code state transitions, and how such broader information could best be presented as hints, I analyzed the types of transitions that occurred in student code and classified them into groups based on the properties of the start and end states.

Based on the interaction networks for world states as described above, I divided the set of observed world states into classes. The first class is the *start state* where the problem begins. By definition, every player's path must begin at this state. Next is the set of *goal states* where all the level buttons are pressed. These are reported by the game as correct solutions. Any complete solution, by definition, ends at one such state. Among states that were neither start nor goal states, there were three important classifications: *intermediate states* that a robot moves through during a correct solution, *mistake states* that a robot does not move through during a correct solution, and *error states* that result from illegal action, such as attempting to move the robot out-of-bounds. Based on these types of states, I then classified the hints based on the transitions they represented. These transitions, labeled subgoal, correction, simple solution, rethinking, and error transitions, are described in detail below:

5.4.6.1 Subgoal Transition

(start/intermediate) → (intermediate/goal) This transition occurs when a student moves the robot to an intermediate state rather than directly to the goal. Since players run their programs to produce output, I speculate that these may represent subgoals such as moving a box onto a specific switch. After accomplishing this task, the user then appends actions to their program, moving towards a new objective, until they reach a goal state. Hint B in Figure 5.4 shows a hint generated from such a subgoal transition.

5.4.6.2 Correction Transition

(error/mistake) → (intermediate/goal) This transition occurs when a student makes and then corrects a mistake. These are especially useful because hints can then offered based on the type of

mistake. Hints D and E in Figure 5.4 show hints built from this type of correction transition; however, hint E shows one where a student resolves the mistake in a sub-optimal way.

5.4.6.3 Simple Solution Transition

(start) → *(goal)* This occurs when a student enters an entire correct program, and solves the puzzle in one attempt. This makes such transitions not particularly useful for generating hints, other than showing a potential solution state of the puzzle. Hint F in Figure 5.4 shows an example simple solution transition.

5.4.6.4 Rethinking Transition

(intermediate) → *(intermediate/goal)* This transition occurs when the user deletes part or all of their program and then moves toward a new goal. As a result, the first state is unrelated to the next state the player reaches. Offering this state as a hint may not help guide a different user. Hint A in Figure 5.4 shows an example of this type of rethinking transition. Finding and recognizing these transitions is an important direction for future work.

5.4.6.5 Error Transition

(start/intermediate) → *(mistake/error)* This corresponds to a program that walks the robot out of bounds, into an object, or other similar errors. While I disregard these as hints, this type of transition may still be useful. In such a case, the last *legal* output before the error can be a valuable state. Hint C in Figure 5.4 is one such case of an error transition.

In general, the transitions are at a higher granularity than expected from hints at the code state level (i.e. add or remove a specific instruction) and two important effects occur as a result of the states being collected only when a user executes their program. First, pottering behavior and mistakes such as rearranging misplaced instructions or playing with the interface are not recorded in the player trace. Second, states tend to cluster around discrete subgoals in the puzzle space. Players appear to execute their program once they believe it accomplishes some part of the objective (for example, placing the first box on its target). This means that states are centered around these goals and erroneous attempts to achieve them. This means the hints provided using this method will direct a player toward the next subgoal that a similar player accomplished, rather than the next interface interaction that player made. This is a desirable trait for a hint system.

Another novel feature of the world state representation is that its hints provide more complete direction on how to recover from errors than the hints resulting from the code state representation. From an error state, the code state hint would be to remove a wrong line of code. The world state hint,

as shown above, would comprise the entire correction. In this implementation, the error-recovery hints are currently discarded as hints since they are incident to errors. However, future studies and systems could make use of this potentially useful error correction feedback.

5.5 Cold-Start Study

In this section I present a formal cold start study to determine whether hint generation is a feasible way to support BOTS puzzles. For this study, the criteria for success is 80 percent hint coverage for new players once the baseline amount of data is collected. I first conduct a "cold start" analysis to get a general idea of how quickly hint coverage for BOTS puzzles increases as new user records are added, and how quickly Hint Factory can reach this threshold for a given puzzle. Based on this result, I select a threshold and evaluate its effectiveness for achieving 80 percent hint coverage for challenge levels that do not provide in-game guidance.

5.5.1 Data

The data for this analysis comes from the 16-level tutorial sequence built into BOTS. These levels were divided into three categories: demos, tutorials, and challenges. Demos are puzzles where the solution is already written for the student. Tutorials are puzzles where instructions are provided to solve the puzzle, but the students build the program themselves. Finally, challenges require the student to solve a puzzle without any assistance. Challenge levels are interspersed throughout the tutorial sequence, and students must complete the tutorial and challenge puzzles in order - they can not skip puzzles.

For the purposes of this cold start study, I excluded demos from the results, and ran the analysis on the remaining 8 tutorials and 5 challenges. The data for this analysis came from a total of 125 students in technology-related camps for middle school students as well as an introductory college CS course for non-majors. Not all students completed the entire tutorial sequence, as only 34 of the students attempted to solve the final challenge in the sequence. A total of 2917 unique code submissions were collected over the 13 puzzles, though it is important to note that the number of unique submissions varied greatly across puzzles. This information is summarized for each puzzle in Table 5.2.

5.5.2 Analysis

To demonstrate the effectiveness of using world states to generate hints, I applied a technique similar to the one used to evaluate the "cold start" problem used in Barnes and Stamper's work with

Deep Thought [BS10]. The cold start problem models the situation when a new problem is used in a tutor with no historical data. The evaluation method described in Barnes and Stamper's previous work uses existing data to simulate the process of students using the tutor, and provides an estimate of the amount of data needed for reliable hint generation. As such, it is an appropriate method for determining how much earlier – if at all – world states generate hints as opposed to code states.

The process was performed over 1000 rounds. In each round, the student data for each puzzle was split into a training set and a validation set. Hint Factory was then trained by iteratively adding one student from the training set to the interaction network at a time. I plotted how the number of hints available to the students in the validation set grows as a function of the number of solutions used to generate the interaction network and averaged the results over all 1000 rounds to avoid ordering effects. Figure 5.5 shows the results of this analysis. The specific algorithm is given below.

Step 1 Let the validation set = 10 random students, and the training set = the $n-10$ remaining students

Step 2 Randomly select a single student attempt from the training set

Step 3 Add states from the student to the interaction network and recalculate the Hint Factory MDP

Step 4 Determine the number of hints that can be generated for the validation set

Table 5.2 A breakdown of the tutorial puzzles in BOTS, listed in the order that students complete them. Hint states are states that are an ancestor of a goal state in the interaction network, meaning that they can be used for hint generation.

		Code states		World states	
Name	#Students	Hint	All	Hint	All
Tutorial 1	125	89	162	22	25
Tutorial 2	118	36	50	12	14
Tutorial 3	117	130	210	22	24
Tutorial 4	114	137	225	33	41
Tutorial 5	109	75	106	25	29
Challenge 1	107	348	560	143	191
Challenge 2	98	201	431	86	133
Tutorial 6	90	107	143	33	36
Challenge 3	89	192	278	28	30
Challenge 4	86	137	208	40	45
Tutorial 7	76	206	383	43	57
Tutorial 8	68	112	134	29	30
Challenge 5	34	17	27	13	17

Step 5 While the training set is not empty, repeat from step 2

Step 6 Repeat from step 1 for 1000 rounds and average the results

This approach simulates a cohort of students asking for hints as a function of the amount of data already in the system, and provides a rough estimate as to how many students need to solve a puzzle before hints can be reliably generated. Since this approach is highly vulnerable to ordering effects, I repeated this process over 1000 rounds and summarized the results.

5.6 Results

5.6.1 State-Space Reduction

Table 5.2 shows how using world states as opposed to code states reduces the state space in the interaction network. For example, Challenge 1 had 560 unique code submissions across the 107 students who attempted the puzzle. These 560 code submissions generalized to 191 unique world states. This is important because the Hint Factory requires overlap between solutions to allow for hint generation. If there are fewer states with more overlap, this will result in a higher probability of being able to provide a hint for each observed state.

Table 5.3 This table highlights the number of code and world states that are only ever visited one time over the course of a problem solution. These states would not have hints available when initially visited.

		Code states		World states	
Name	#Students	All	Freq1	All	Freq1
Challenge 1	107	560	146	191	112
Challenge 2	98	431	127	133	84
Challenge 3	89	278	91	30	22
Challenge 4	86	208	65	45	36

After collapsing code states to world states, there was a significant reduction in the number of states containing only a single student. Students who got correct answers overlapped more in terms of their solutions, while the open-ended possibility of incorrect answers resulted in several code submissions that only a single student ever encountered. Table 5.3 shows the number of frequency-one states that students encountered for challenge puzzles 1 through 4.

In all three cases using world states, more than half of the states in the interaction network were observed only once. Challenge 3 is particularly interesting, considering that out of 278 unique code submissions from 89 students, only 8 of the 30 world states were observed more than once. Since the

number of students did not change, this indicates that students' solutions were effectively condensed into fewer states. The degree of overlap demonstrates that world states meet the assumptions necessary to apply Hint Factory.

5.6.2 Cold Start

The graphs in Figure 5.5 show the result of the cold start evaluation for all 13 of the non-demo puzzles. Looking at the two graphs side-by-side, the world states have more area under the curves, demonstrating the ability to generate hints earlier than code states. The effect is particularly pronounced when looking at the challenge puzzles, represented with the solid blue line. In tutorials, code snippets were given to the students, so there was more uniformity in the written code. In levels without guidance, the code became more variable, and so for these levels world states show a more demonstrable improvement.

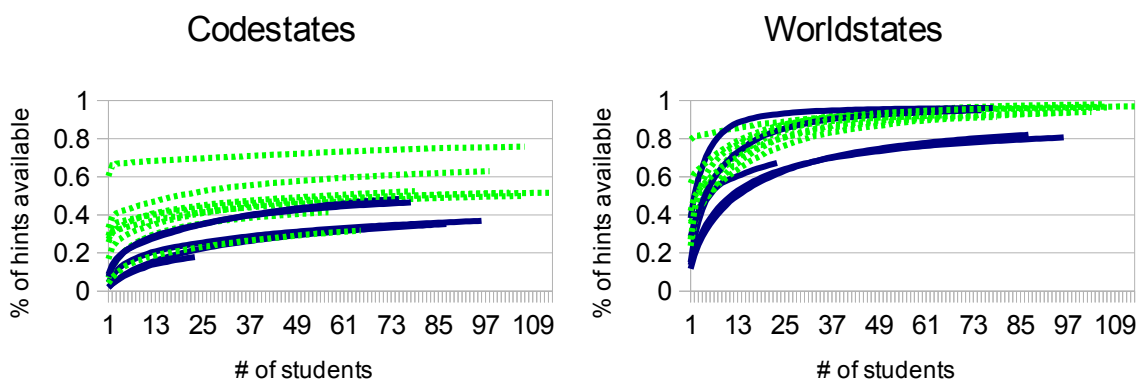

Figure 5.5 These graphs show the overall performance of hint generation for code states and world states. The solid blue lines are the challenge puzzles, and the dotted light-green lines are the tutorial puzzles. The Tutorial puzzles (which tend to be simple, and which have text-based hints) perform better than the more complex and hint-less Challenge puzzles.

It is important to note that since the world states approach will only collapse code states, and never create additional states, the ability to generate at least as many hints as the code states approach is trivially guaranteed. The contribution of creating world states is that it is easier to *scale* hint generation. The Hint Factory applied using world states from a source data set can generate hints for 2-4 times as many states as Hint Factory using code states derived from the same source data set.

Figure 5.6 summarizes these results by averaging the hints generated for the first four chal-

lenge puzzles. Challenge puzzles were chosen as they represent a puzzle being solved without any prompts on its solution, and would be the environment where these automatically generated hints are deployed in practice. Challenge 5 was only attempted by 34 students, so it was left out of the analysis.

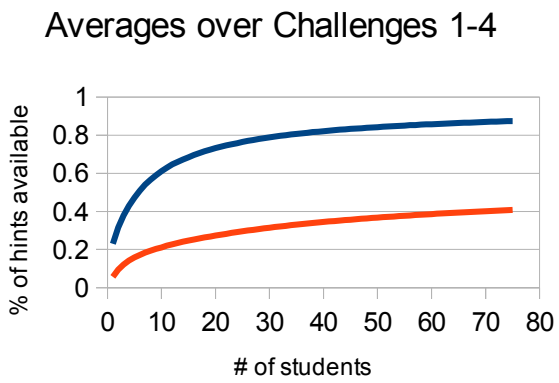

Figure 5.6 This graph summarizes Figure 5.5 by averaging the results of the cold-start analysis over the first four challenge puzzles. The red line is hint coverage (% hints available) for code states, and the blue line is for world states. For these challenge levels, where students do not have guidance from the system, using code states requires much more data than world states do for hint generation.

Table 5.4 This table shows the hint coverage using world states when data from 30 students are in the interaction network.

Name	Average	Median	Min	Max
Challenge 1	0.67	0.66	0.48	0.81
Challenge 2	0.67	0.66	0.44	0.84
Challenge 3	0.94	0.93	0.83	0.99
Challenge 4	0.88	0.87	0.69	0.96

Examining the result in Figure 5.6 suggests that a data set from 30 students should be able to generate hints using world states about 80% of the time for these puzzles. To evaluate this hypothesis, I generated new Interaction Networks with random training sets of 30 students and measured hint coverage on the remaining students for challenge puzzles one through four to measure how often a state from a remaining student would have hint coverage, that is, how often that state would have been visited by one of the students whose solution attempt was in the training set and who later reached the goal. The average, median, maximum, and minimum of the results over another 1000

trials is summarized in Table 5.4. For challenges 1 and 2, hint generation is below the 80% mark, but for challenges 3 and 4, it is well over. This discrepancy is interesting and is discussed in the next section.

5.7 Discussion

The results indicate that less data are needed for hint generation when using world states as opposed to using code states. The reduction in the state space and the number of hints available after only a few dozen students solve the puzzle is highly encouraging. The results are tested on instructor-authored puzzles for this study, but these results potentially make hint generation for user-generated levels feasible as well.

While on average the number of hints available using world states is very high, it is interesting to look at numbers on a per-level basis. For example, in the summary in Table 5.4, there are fewer hints available after 30 students have attempted Challenge Puzzles 1-2 than for Challenge Puzzles 3-4, that are presumably harder. This could be an averaging effect due to the more advanced students remaining in the more advanced levels, but there are some structural differences to the levels that may also have an effect. Figures 5.7 and 5.8 show the puzzles for Challenges 1 and 4.

Challenge 1 is a much smaller puzzle, but it can be solved in many different ways. Any of the blocks can be put on any of the buttons, and the robot can also hold down a button, meaning that there are several goal states, especially considering that the superfluous block can be located anywhere. Challenge 4 is substantially more difficult, but has much less variation in how it can be completed. In this puzzle, the student has to place blocks so that the robot can scale the tower, and because of the way the tower is shaped, there are far fewer ways to approach the problem as there are in Challenge 1.

The results for the earlier stages are more interesting for interpreting these results, because they demonstrate how well world states manage the variability in open-ended problems. While I emphasize the utility of world states for hint generation, it is important to note that code states still have utility. A hint can be generated from any of the data in an interaction network, and using a world-based state representation does not preclude comparison with other collected data. When enough data is collected for the code states to match, more specific, low-level hints can be generated as well, meaning that more data always means more hints, and also more detailed hints that can be applied at the source code level.

5.8 Conclusions

I have adapted an existing approach to modeling student interaction with a serious game. This approach can be used to automatically generate hints with the Hint Factory algorithm. Rather than attempting to encode the programs or step-by-step interactions of the user, I use the configuration of the world after each compilation of the student's code. This represents all of the unique code submissions much more compactly. Note that the analysis of hints was a static analysis on historical data, and that I did not implement hints into any levels in BOTS. While I use a naive implementation of Hint Factory to test the feasibility of hint generation, there are some interesting features of BOTS hints. In prior Hint Factory studies, error states were not considered for hint generation since the system detects and deletes errors. In BOTS, error states were included in the interaction network, so some BOTS hints represent corrections. I felt it was important to include deletions and errors in BOTS since errors in programming are more complex than those for logic, and repairing them could be an important use for hints.

This work demonstrates that even with a small number of records, useful hints could be generated by grouping user actions according to their outcomes. A similar system can be used in real-time

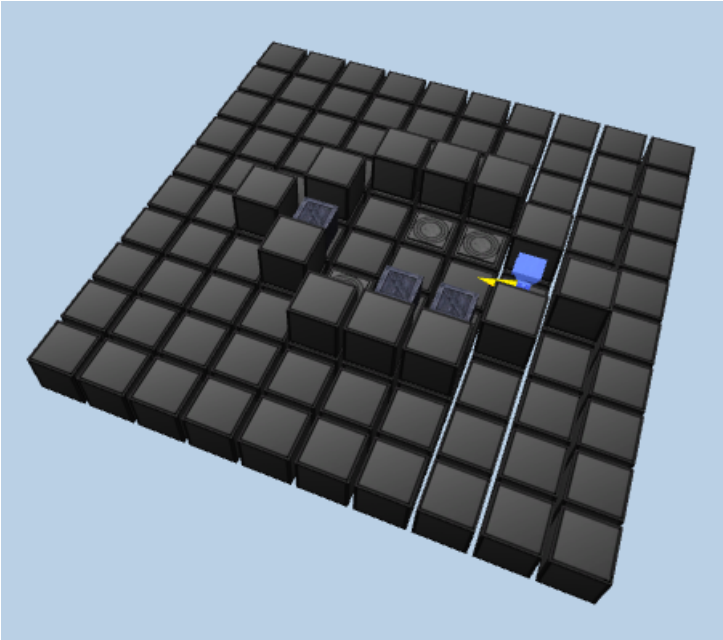

Figure 5.7 A screenshot of challenge 1. There are more blocks than necessary to press all the buttons, since the robot can press a button too.

games, generating hints based on important results or milestones, where using the lowest-level interaction data would be impractical. However, in this work I use well-explored levels used in the game's tutorials. It is possible to use a system like this for user-generated levels, but this would require a similarly large amount of data for hint generation, necessitating some mechanism to direct players to levels until enough data has been collected.

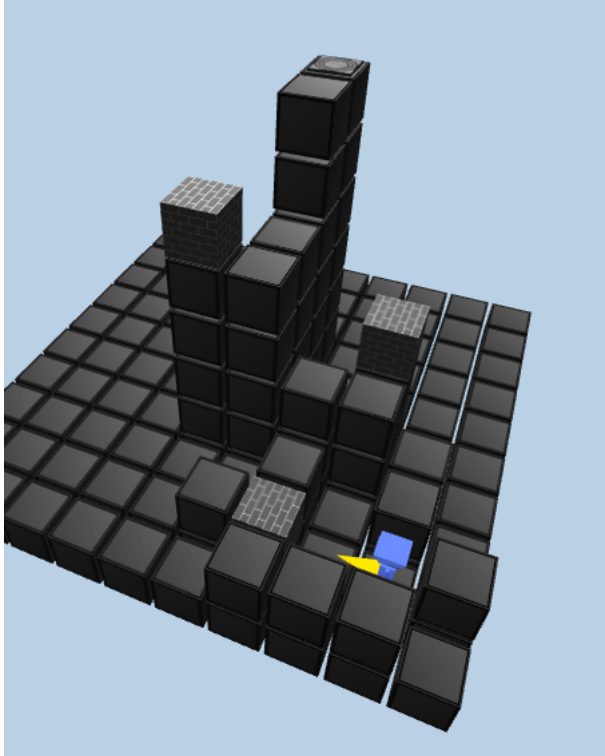

Figure 5.8 A screenshot of challenge 4. The puzzle is more complex, yet has a more linear solution.

Chapter 6

Gameplay Affordances

Since the Solve-and-Submit mechanism presented in Chapter 4 did not directly influence players to create levels that actively encourage loops and functions for their solutions, I designed two new level editors: the Programming editor and the Building-Blocks editor. The goal of these editors was to implicitly constrain certain level creation behaviors (namely, the creation of large empty levels or extraneous structures) while encouraging others (the creation of obvious structural cues that encourage loops and functions).

6.1 Context

In this chapter, I investigate whether the changes to the level editor influenced how authors build levels, in terms of affording the game's core mechanic. This chapter reports and extends work originally published at Educational Data Mining (2016) [Hic16a]. I present additional analysis of user-generated levels filtered out by the game's minimum threshold, and investigate into the impact of tutorial completion on the use of loops and functions to create puzzles. This chapter addresses my research question Q2 and hypothesis H3:

- Q2: How can level authoring tools designed using creative constraints help promote the creation of high-quality levels that afford and reward the use of core mechanics in BOTS?
- H3: Authoring tools designed with creative constraints will result in authors more frequently creating levels that afford the game's core mechanics, when compared with the original Free-Form authoring tool.

6.2 Introduction

I have demonstrated that user-created levels in BOTS can contain appropriate gameplay affordances, that reward specific, desired patterns of gameplay related to the BOTS learning objectives. Such levels demonstrate the creator's understanding of those learning objectives, and offer other players opportunities 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) that ignore or replace the game's core learning objectives and challenges. Before including user-created levels in the game, an additional filtering and evaluation step is needed to identify and remove these low-quality submissions. My initial attempt at filtering these levels was effective at reducing the number of published lower-quality levels. However, some users created fewer levels under this condition, indicating that implementing the barrier after level creation discouraged further creation. The Solve-and-Submit feature was not as effective at reducing the overall creation of undesirable levels; it merely limited their publication. The next step is to make further improvements to the content authoring tools to increase the overall quality of the submitted content. In this chapter, levels created in three versions of the level editor are compared, including the original Drag-and-Drop editor, and the new constrained Programming and Building-Blocks editors. I test how various types of constraints affect the quality of created levels.

Research has shown that players are engaged when constraints encourage the demonstration of the game's target learning concepts, but do not prevent players from creating what they want to make [Boy11]. Therefore, I have designed two variant level editors with two different forms of constraints, inspired by previous approaches to user-generated problems and problem-posing in educational games and intelligent tutoring systems.

The first of my variant editors, referred to hereafter as the "Programming Editor", requires authors to program the robot exactly as they would in the main game, while a level matching their program is generated by the robot's movement. To constrain canvas-filling behavior, the length of the solution, in lines of code, is constrained. I intend this constraint to function similarly to the Point Value Showcase thresholds present in BeadLoom Game, as in Boyce's work [Boy11], where players created puzzles using a limited number of tools. In the BeadLoom Game, these constraints promoted creative players to use more advanced tools and learn more computational thinking while playing.

The second editor, referred to as the "Building-Blocks Editor", constrains authors to building levels with a limited selection of "building-blocks". Except for the smallest "Move 1" and "Rotate 90*" blocks, all building blocks function as gameplay affordances, since they contain repeated patterns. Players are also limited to the use of a certain number of building-blocks regardless of

those building-blocks' sizes. Coupled with the canvas-filling impulse we observed earlier, I expect this to discourage the use of small building-blocks, and encourage use of the larger blocks more frequently. This will result in more gameplay affordances embedded into the created levels.

6.3 Background

In Chapter 2, I discussed how user-generated content has been revolutionizing gaming. Commercial games such as Super Mario Maker [Nin08] and Little Big Planet [Med08] rely almost entirely on user-submitted levels to provide a long-term gameplay experience. Such creative gameplay motivates different demographics and players than competitive gameplay [Gre10; HK06; Bou10]. However, criticism of these systems has frequently pointed out the difficulty in finding good levels that reflect the core mechanics of the game [McE15; Her15; Tho15].

In Chapter 2, I also discussed how educators and researchers have shown that creating exercises (i.e. problem-posing) can be used as an effective assessment of student math knowledge. These researchers also noted the logistical challenges of problem posing and sharing in the classroom, and students' tendency to create simple problems well below the difficulty level of concurrent classroom material [Sil13; Cai13]. Problem-Posing has also been explored as a content-creation method with games and ITS such as AnimalWatch [BB08] and MONSAKUN [HK11], having users create exercises from expert-selected "ingredients." Researchers found that when the posed problems were used in a game, low performing students experienced significantly greater learning gains and engagement [Cha12]. These systems for Problem-Posing inspired my design of the "Building-Blocks" editor.

6.3.1 Gameplay Affordances

In this section, I discuss the idea of "Gameplay Affordances," an idea I will use to evaluate the quality of created levels.

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" [Gib66; Gib14]. 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" [Nor88]. This definition focuses on the user's perspective; if a user cannot identify that an action is possible with an object, then the object does not afford it, even if the action is possible.

With respect to affordances in games, James Paul Gee wrote that games create a match between affordances and what he calls "effectivities" [Gee09]. 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. In platformer games, these are the aspects that make it clear when an object can be jumped upon, climbed, or walked on. For example, an enemy with a perfectly flat surface on top of his head affords being walked upon. Other work creating taxonomies for level design patterns in video games also referred to the desired gameplay produced by game structures. For example, in Hullett and Whitehead's work with design patterns in single-player first-person shooter levels, the "Sniper Location" design pattern is a difficult to reach location with a good view of the play area, occupied by an enemy [HW10]. This pattern is described as one that forces the player to take cover. The presence of other gameplay elements such as vehicles and turrets herald similar gameplay changes [Bac08].

In BOTS, like most other grid-based LOGO-Like puzzle games [Vah14] the primary educational goal is to teach students basic problem solving and programming concepts such as using functions and loops to handle repetitive patterns. Players, who see the robot as their tool, must look within puzzles for opportunities to use loops and functions and earn a better score. Thus, affordances in BOTS are objects or patterns of objects that signal the presence of these optimization opportunities to players.

Though the objects in BOTS can signal gameplay patterns, players building levels in BOTS frequently place objects in misleading or irrelevant ways, where the seeming gameplay affordances are not useful for a correct or successful solution. For example, boxes afford being "picked up" and "put down" on targets to achieve game goals. However, when the puzzle contains no goal, or when the best solution does not require that a box be placed on a goal at all, the affordance of a box is meaningless in terms of gameplay reward. Similarly, consider a level with a large repeated structure affording the use of loops, but whose best solution contains a shortcut somewhere along the path. The level affords the use of loops, but does not reward it as much as it rewards the skill of locating the shortcut. Similarly, levels may contain opportunities for higher rewards through the use of loops and functions but may not contain any structure indicating this. For example, a large flat plane with goals and boxes scattered around it is likely to contain an opportunity to use loops or functions to save lines of code, but its visual layout may not help players perceive this potential. In such cases, it is not the structure of the level or presence of objects in the level that communicates this, but patterns within the program the user writes. In this case, we would not call the level's opportunity to use loops or functions a gameplay affordance. Instead, I define gameplay affordances as those that combine visual structural cues with a potential reward, as measured by the difference in gameplay performance between the naive and expert solutions. Therefore, an affordance is only considered a "Gameplay Affordance" if it occurs alongside improved outcomes in terms of the core mechanism of the game. These types of affordances are referred as "Gameplay Affordances" in remaining sections.

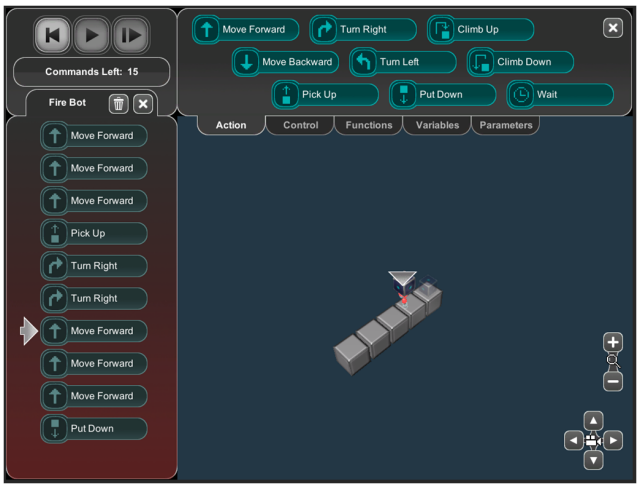

Figure 6.1 The Programming editor interface.

6.3.2 Level Editors

For clarity, a brief review of the design of the editors analyzed in this chapter is provided. Specific discussion of the design principles behind the two level editors used for this study can be found in my previous work, [Hic15] and the previous chapters.

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 that must be covered with movable boxes to complete the level.

In the Free-Form Drag-and-Drop editor, players create a level using 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. Once they have placed a starting point and a goal, the player can save the level. The player must then complete the level on their own before it can be published and available to other users.

In the Programming Editor, inspired by the Deep Gamification-based level editor in BeadLoom Game [Boy14], players create a level by programming the path the robot will take, using a limited number of instructions. This is analogous to the level creation tools in BeadLoom Game where players create 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 creation does not mean comprehension. Since simple solutions are still permitted, and

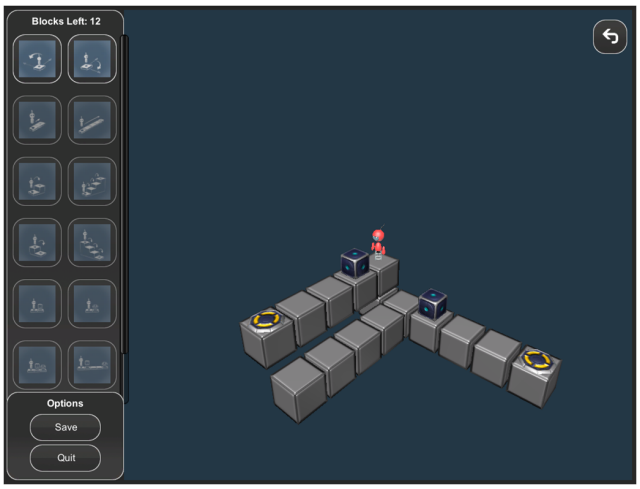

Figure 6.2 The Building-Blocks editor interface.

since most programs are syntactically correct, users who are experimenting with the level creation interface may make levels with commands such as loops or functions, that they themselves may not fully understand. This is important if we were to use the simple presence of loops or functions to determine whether students have fully learned how to use them.

In the Building-Blocks editor, level creation is constrained 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 [HK11] and Animal Watch [AW03; BB08] where players build a problem using data and expert-created problem pieces that are known to contain meaningful, relevant information. In this version of the level editor, players create a level only using "building-blocks" that are pre-constructed chunks of levels. These "building-blocks" are partial or complete examples of the patterns identified in Chapter 4. Building-blocks have specific structures that correspond to opportunities to use loops, functions, or variables. I hypothesize that levels built using these blocks may be better because the blocks explicitly promote opportunities for players to use more complex programming constructs like loops and functions. I also believe that the building-blocks will encourage players to think about optimizing the solution to the level while they are making the level. One potential challenge with this approach is that players may find these constraints too restrictive, and this might reduce engagement for creatively-oriented players [Boy14].

By comparing these level editors, I hope to determine which approach best promotes the creation of levels with gameplay affordances, and high gameplay rewards for these affordances.

6.4 Data

The data used in this analysis are puzzles created by participants in SPARCS and STEM outreach activities conducted with middle school students. For the Free-Form editor, the data also includes anonymized levels from previous experiments and other outreach use of the tool where the same 90-minute session structure described below was followed. This includes gameplay data from 181 unique user IDs ($n = 48$ in the Programming condition, $n = 61$ using Block Editor, $n = 72$ using Free-Form Editor) across all classes/workshops that use the BOTS game as part of their activities (as shown in the Appendix C). In total, 243 levels are created by these players (91 Block / 59 Programming / 93 Free-Form). Of these levels, 10 Block, 7 Programming, and one Free-Form levels are excluded due to bugs, reducing the total number of levels in the sample to 225 levels (81 Block / 52 Programming / 92 Free-Form). We then removed all levels that were unplayable since their expert solutions took fewer than 5 commands. Our zero-inflation model is then applied to the resulting 197 puzzles, published and unpublished, (73 Block / 44 Programming / 80 Free-Form), created by 54, 42, and 64 authors respectively. We note that a total of 175 levels (49 Block / 33 Programming / 92 Free-Form) were published, but we did not distinguish published or unpublished levels in this study.

Data is collected in 90-minute sessions, and 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 that functions as sort of a “collector” stage. Players are not expected to complete this level with the 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 are told to prompt players to re-read the offered hints for their current level carefully if a player becomes stuck, and only to offer more guidance after the player has carefully read the instructions. This part of gameplay lasted 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 creating 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. Data for the “Free-Form” editor was entirely from past data collections, since it was the original, and only, level editor. The first of the three additional data collections analyzed here used only the “Programming” level editor. In the remaining two data collections, students were evenly assigned between the “Programming” editor and the “Building-Blocks” editor based on the modulus of their unique User IDs.

6.5 Methods and Results

In this section, I describe the Expert and Naive solution generation and their score difference as a direct measure of puzzle quality. I present a zero-inflation model to explain the differences in the quality of levels created in each editor. To confirm this numerical analysis of quality, I also compare expert tagging of levels created across editors.

6.5.1 Expert and Naive Solution Generation

To analyze the differences between created levels, each level was played to find the shortest-path solution from start to goal, and a solver was used to find the shortest program to produce this optimal expert solution. As the actual process of solving a BOTS puzzle is as complicated as that of a Light-Bot puzzle [Smi13], I instead used an simple solver to find the best optimization of the shortest discovered path in the level based on available solutions. The algorithm used by the solver was relatively simple: First, the shortest length path among known solutions is recreated using only simple commands. Then, sets of repeated commands are identified in this program by treating the commands as words and identifying repeated n-grams. Then, each possible combination of optimization on these n-grams is recursively applied: either replacing the n-gram with a subroutine identifier wherever it appears, or replacing adjacent repeated n-grams with a single instance of that n-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* that uses only simple commands such as moving and turning. The optimized shortest-path solution is the *expert solution* that uses loops and subroutines to optimize the shortest path solution. The difference in scores (the number of commands/lines of code used) between these solutions is used as a measurement of how well the level affords the use of those game mechanics.

6.5.2 Overview of Level Score Improvement

Figure 6.3 presents the box-plot for score improvement, or Difference, between expert and naive solutions for published puzzles. The pink area shows the mean value and the 95% confidence interval around it. A visual inspection shows that the confidence intervals for the Programming Editor and Free-Form editor do not overlap, implying that the Programming Editor achieved better results on this measurement. This is confirmed in a later statistical analysis.

In Figure 6.3, the light and dark-grey sections show the median value and demarcate the quartiles of the data. The zero-value levels are over-distributed, meaning there are a large number of zero values. This is similar in effect to when a distribution is truncated at zero and all the values that

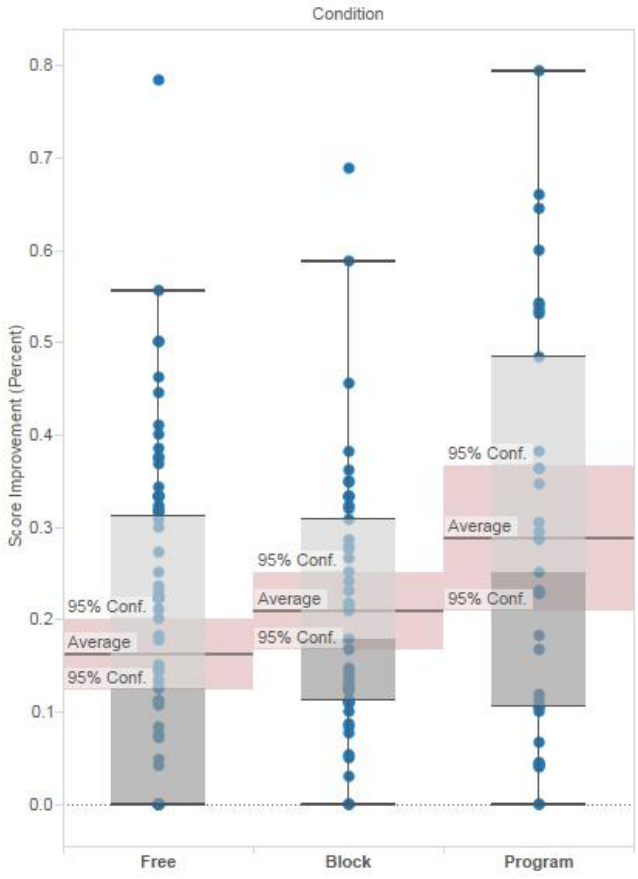

Figure 6.3 Plot comparing the distribution of levels between the three conditions. Each point in this plot represents the difference in number of commands between a naive and expert solution, as a percentage of the expert solution.

would lie below zero are treated as zero values. This can be observed especially in the Free-Form condition. This is shown by the second quartile touching the X axis in one condition and the first quartile touching the X axis in all three conditions. This property of the score distribution impacts which statistical methods can be used to evaluate the measurement of saved commands/lines of code.

6.5.3 Direct Measurement of Improvement

I examined the differences between levels created under all three conditions using a zero-inflated Poisson distribution model. This zero-inflation model is used because the model looks for two

separate effects: first, the effect that causes the dependent variable to be zero or non-zero, and second, the effect that causes the value of the dependent variable to change in the non-zero cases. This is important because the structural elements for levels with zero affordances for advanced game mechanics are very different from those with only a small affordance. In other words, it is expected for the baseline Free-Form editor to have more zero-improvement values for the difference between the naive and expert solutions, and for the other two editors to have more non-zero values for this difference. 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 that limit how advanced game mechanics may be used to optimize the solutions.

I used the zero-inflation model to evaluate the differences between levels on a direct measure of improvement: the score (and length) difference between a naive solution and an expert solution, called "Difference" hereafter. This type of model is used for modeling variables with excessive zeros and over-dispersed count outcome variables. Furthermore, it is used when theory suggests that the excess zeros are generated by a different process from the other values, and can therefore be modeled independently. The data in this experiment has an excess of zeroes since many puzzles had no difference between naive and expert solution scores. The difference between zero improvement and non-zero improvement is important since a level with zero improvement contains no affordances. However, a level with even a small non-zero improvement may still contain gameplay affordances, making it significantly better in promoting the game's objectives than a zero-improvement level.

Table 6.1 Count model coefficients (Poisson with log link) comparing the two editors to the baseline, Drag-and-Drop editor, with auto-filtered levels removed

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	1.90469	0.05422	35.132	< 2e-16 ***
Programming Editor	0.35644	0.07588	4.697	2.64e-06 ***
Block Editor	-0.03060	0.07400	-0.413	0.679

Table 6.2 Zero-inflation model coefficients (binomial with logit link): comparing the two editors to the baseline, Drag-and-Drop editor, with auto-filtered levels removed

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-0.5679	0.2331	-2.436	0.01483 *
Programming Editor	-1.0976	0.4736	-2.317	0.02049 *
Block Editor	-1.0671	0.3944	-2.705	0.00682 **

The results of fitting a Zero-Inflated Poisson model on our data are presented here. The two tests are examined separately: the binomial model relates to whether a level has zero or non-zero results, and the Poisson model relates to the size of the non-zero results. The binomial model shows that the Building-Blocks editor and Programming editor were more likely than the baseline condition (Free-Form Editor) to produce a non-zero result for Difference. This makes sense because only the very simplest building-blocks available to students offer no affordances, and all but the simplest building-blocks are built specifically to contain affordances. In order to construct a zero-valued level, a Building-Blocks student would need to use only the simplest blocks. Some students did use simple blocks exclusively, but as these results show, it was much less likely for students to create zero-affordance levels in the Building-Blocks editor. In the Programming editor, the number of commands available were limited, so building larger levels required using functions or loops, and thus the expert solution to the level would include those same improvements.

The Poisson model shows that, among the non-zero results, the Programming editor was 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 were created using only 3 to 4 commands. If a Building-Block is repeated, a later user can use a loop or function to save many more lines of code. However, if a Building-Block is not used multiple times, these small-valued gameplay affordances are the only ones that are present. On the other hand, in the Programming editor, players were observed to explore 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 have much larger differences between the naive solutions and expert solutions than levels generated from those using multiple unique building-blocks.

Since removing levels with a best solution less than 5 by definition eliminated some very low affordance levels across all conditions, I next examined the zero-inflation model created with those levels included.

Table 6.3 Count model coefficients (Poisson with log link) comparing the two editors to the baseline, Drag-and-Drop editor, with auto-filtered levels included

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	1.88012	0.05288	35.555	< 2e-16 ***
Programming Editor	0.38101	0.07493	5.085	3.68e-07 ***
Block Editor	-0.03356	0.07297	-0.460	0.646

Using this model, the same effects are largely present, with the Programming editor no longer

Table 6.4 Zero-inflation model coefficients (binomial with logit link) comparing the two editors to the baseline, Drag-and-Drop editor, with auto-filtered levels included

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-0.3732	0.2114	-1.766	0.0774 .
Programming Editor	-0.4047	0.3613	-1.120	0.2627
Block Editor	-0.8331	0.3376	-2.467	0.0136 *

having a substantially different effect from the Free-Form editor on the excess zero-generation portion of the model, but retaining the effect on the count portion of the model. Applying an additional filtering step afterward accounts for many trivial zero-affordance levels, but retains the same drawback as the Solve-and-Submit system, that player effort is still spent creating these levels.

In summary, interpreting this model leads to the following conclusions. First, the expected result is verified, that both the Programming editor and the Building-Blocks editor were more likely to produce a non-zero result, statistically significantly more likely than the baseline (Free-Form) condition. The Building-Blocks editor retains this effect even when the automatically-pruned levels are considered. Second, the Programming editor was most likely to result in a higher-value improvement difference between and naive and expert solutions, indicating that it promotes puzzles that allow for more optimization.

6.5.4 Expert Tagging

Next I compared puzzles across all three versions of level editors, with the hypothesis that the more meaningful the level editor's construction unit, the higher quality the puzzles would be. Here, I assumed that "building-blocks" and programs were more meaningful than the terrain blocks used in the Free-Form editor.

One expert, blind to the editor used, identified the presence or absence of negative design patterns in each puzzle. The defined puzzle design patterns are: "Normal" levels with few (or no) negative design patterns, and four categories of levels characterized by specific negative design patterns: *Griefer, Power-Gamer, Sandbox, and Trivial* levels.

I measured each puzzle's quality based on patterns of negative content. These patterns were identified in previous chapters and are used as tags for this study. The following criteria are also 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

I decided on these criteria because the 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 that are impossible, or simply far too tedious, are among the most common negative traits identified in previous designs. The Building-Blocks and Programming level editors specifically address these two criteria via hard constraints on the placements of goals and the size of levels.

Table 6.5 Expert-tagged Categories for Levels Created in the Three Level Editors

	DaD	Program	Block
Normal	66	43	66
Power-Gamer	9	1	13
Griefer	2	0	0
Sandbox	10	0	0
Trivial	5	8	2
TOTAL	92	52	81

Table 4 reports the number of puzzles in each category, created in the three level editors. I use Fisher's exact test for the editors since it is used for comparing categorical variables when sample sizes are small, and it is more accurate than a chi-squared test. Fisher's Exact Test showed a significant difference in the category distributions between each pair of the three level editors (Building-Blocks vs. Programming: $p<0.001$; Building-Blocks vs. Drag-and-Drop: $p=0.0032$; Programming vs. Drag-and-Drop: $p=0.0035$).

The Programming editor had the highest proportion of Normal puzzles (82.7%). Moreover, the Building-Blocks and Free-Form editors created a higher proportion of Power-Gamer levels compared with the Programming editor. These Power-Gamer levels are characterized by extreme length and a high number of objectives. The Free-Form Drag-and-Drop editor is the only level editor where users created Sandbox puzzles. This is not surprising, since our criteria for Sandbox levels include placing off-path objectives and structures, but doing this is quite difficult in the newer editors. Finally, the

Programming editor had the highest proportion of Trivial puzzles, as defined in Chapter 4 as those with obvious easy and short solutions. This may be because the players are novice programmers who may not be comfortable with the more complex commands, and the complex commands are also located on separate tabs.

Since players in the two new editors used a shorter tutorial than players in the Free-Form condition, I investigated whether student performance in this tutorial had an impact on which level editor was more effective. I considered whether the authoring player had completed the new tutorial levels during the allotted time. This analysis is again performed on the reduced data set, where I removed created levels with solutions less than 5 steps long.

Table 6.6 Count model coefficients (Poisson with log link) including tutorial completion

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	1.90469	0.05422	35.132 <	2e-16 ***
Programming	0.32000	0.08126	3.938	8.22e-05 ***
Building-Blocks	-0.05968	0.07760	-0.769	0.442
Finished tutorial	0.10040	0.07763	1.293	0.196

Table 6.7 Zero-inflation model coefficients (binomial with logit link) including tutorial completion

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-0.56787	0.23307	-2.436	0.0148 *
Programming	-1.11728	0.51517	-2.169	0.0301 *
Building-Blocks	-1.08260	0.42350	-2.556	0.0106 *
Finished tutorial	0.05354	0.54440	0.098	0.9217

With this more complex model similar results are shown. Finishing the new tutorial does not have a statistically significant effect. However, the coefficient for the magnitude portion of the model (the z value) is still relatively large, indicating a positive but not statistically significant effect. Finishing the tutorial seems to have no compelling impact on the zero portion of the model. Overall, the “finished tutorial” feature in our model has a positive but not significant effect on the scores of non-zero levels, and no interesting effect on the frequency of zero-score levels, irrespective of condition.

6.6 Discussion

The results indicate that the baseline Free-Form Drag-and-Drop editor was the least likely to result in levels with gameplay affordances for using loops and functions. The baseline editor resulted in the lowest proportion of Normal puzzles, but higher proportions of undesirable Sandbox puzzles and Power-Gamer puzzles. Additionally, baseline editor players created fewer puzzles that could be improved by loops or functions, or that had obvious patterns for using loops or functions. Players using this editor were less likely to consider the gameplay affordances of their levels, adding elements regardless of their effect on gameplay. Additionally, the baseline Drag-and-Drop editor was the only one resulting in Sandbox puzzles. This may be because Sandbox levels are characterized by the presence of extraneous objects, and the other editors operate by creating the robot's path, so designers would have to deliberately stray from (and return to) their intended path to place extraneous objects.

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 typically takes a short time to create but a long time to complete. Since this editor uses the exact same mechanic for creation as completion, the only way to speed up level creation is to also speed up the solution. On the other hand, Programming editor users also built the highest proportion of Trivial puzzles whose solutions were too short to afford the use of loops or functions. This Programming 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, while in the other editors undirected behavior is more likely to result in different level types: Power-Gamer levels in the Building-Blocks editor, and Sandbox levels in the baseline editor.

Finally, the Building-Blocks Editor had a high proportion of Normal puzzles, and the highest proportion of puzzles that could be improved with loops or functions and had 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 Building-Blocks editor also 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. Undirected behavior such as adding blocks randomly likely results in this type of puzzle. I had previously observed that players tended to fill the space available to them in the baseline Drag-and-Drop editor, so Building-Blocks puzzle creators may also have been trying to fill the available space.

6.7 Conclusions

In conclusion, both the Programming and the Building-Blocks editors are more effective than the baseline Free-Form Drag-and-Drop editor at encouraging the creation of levels with gameplay affordances. The Building-Blocks editor is most effective at ensuring a positive non-zero improvement between naive and expert solutions, but perhaps trivially, as the building-blocks themselves are selected to contain small improvements. Levels created in the Programming Editor have larger improvements, and these may be more obvious or more rewarding to players than numerous small improvements.

There are limitations to this analysis and the new editors. Although these analyses show that the levels created in the new Building-Blocks and Programming editors contain opportunities for users to practice, users may not necessarily recognize or take advantage of the opportunities. There were several patterns of negative design that are unique to these new editors, shifting what might have been "Sandbox" designs in the baseline editor into Power-Gamer or Trivial levels. For the Building-Blocks editor, this seems to be largely negative, resulting in over-long, unrewarding levels. In the Programming editor, random exploratory sandbox behavior sometimes results in interesting levels created when the author experiments with loops and nested functions rather than creating levels with an end-goal in mind. Sandbox exploratory creation in the baseline Drag-and-Drop editor is treated as negative, with the output levels being low-quality. In the Programming editor, this sometimes results in Trivial levels but sometimes results in more interesting levels that allow for higher score improvements. In the next chapter, I investigate how players react to and behave within levels created under these conditions.

Chapter 7

Effectiveness of Gameplay Affordances

In the previous chapter, I demonstrated that the Building Blocks and Programming level editors result in levels with more opportunities for players to use more advanced concepts like loops and functions. In this chapter, I explore whether these levels actually encourage player solutions that take advantage of these opportunities more often, addressing my research question Q2 and hypothesis H4:

- Q2: How can level authoring tools designed using creative constraints help promote the creation of high-quality levels that afford and reward the use of core mechanics (loops and functions) in BOTS?
- H4: Levels that contain these gameplay affordances will more frequently contain correct use of the game's advanced mechanics (loops and functions) in players' solutions.

To test this, I ran three additional sessions where players had access to a randomized sequence of 12 levels selected from among those created under the three conditions (Drag and Drop, Building-Blocks, and Programming editors). During each session, players were encouraged to complete as many of the levels as possible.

7.1 Data

Data in this chapter consists of 205 completed solutions by 37 students to the problems created and coded in the previous chapter. These solutions were collected during three outreach sessions where the game was used as a teaching activity.

Each of the three outreach sessions were conducted using the same procedure described here. For the first five minutes, students were instructed to open Firefox and navigate to the website

bots.game2learn.com and create an account, generating a unique, anonymous user ID. Two students had played the game during a previous outreach and accessed a previous account. Students were then instructed to complete the Tutorial at least up to the boss level, called "Bot-Henge", before proceeding on. During this segment, if students get stuck, instructors prompted them to carefully read the hint for their current levels. Players could click the "?" button at the top right to do this. If students were still stuck, instructors could offer a bit more guidance. This is because the purpose of this study was not to evaluate the efficacy of tutorial levels, but to investigate how players solved the user-created levels. The students were playing the tutorial precisely to learn how loops, functions, and variables work in this game, so we can see if later levels offer them affordances to use that knowledge. Therefore, instructors did not withhold advice that would help them learn about how BOTS works.

After completing the tutorial or after 45 minutes (whichever came first), students were instructed to click "Custom Puzzles" to access a 12-Level campaign. This campaign was automatically generated per user. The campaign contained 4 random levels from each of the previous study's conditions, in a random order. Students were instructed to get as far as they could through the Campaign in a 45-minute period. Only 4 of these students completed all 12 levels in the campaign. Most did not complete all 12 levels; the average number of campaign levels reached was 6.075 levels per player.

After this part of the activity, students were free to make levels by clicking the editor button, or to play any other levels.

7.2 Methods

To analyze this data, I gathered the completed correct solutions and coded whether each solution is advanced or naive – that is, whether each solution contains at least one correct application of loops, functions, or variables. This captures two factors: the solution containing an instance of loops or functions, and the solution correctly solving the level. I then used the level tags as well as the expert and naive solutions derived in the previous chapter to build a binomial regression model. This is to determine how well the level-creation condition, the level tags, or the score difference between naive and expert solutions, i.e. the amount of gameplay affordance, allow us to predict whether a solution to that level will be advanced or not.

I performed logistic regression to evaluate the results, since the dependent variable is a binary classification. I then used R's analysis of variance package to evaluate each iteration of the model. This package in R is called Anova. It is important to note the R Anova package does not perform the ANOVA statistical test, which would be inappropriate for binary dependent variables, but instead performs the analysis of variance for the specific type of fitted model object in R. First, I wanted

to determine whether the Condition or Difference (gameplay affordance) Values for a solution's underlying level had an impact on the binary factor of the solution containing correctly-used loops or functions. Because the number of levels played in each condition are unbalanced due to different play patterns among students, I used Type III (weighted) sum of squares so that this imbalance would not impact the results. For this analysis I considered models with and without the interaction term, and found that the model without the interaction term fits the data better when comparing the models based on Akaike information criterion (AIC). AIC is a common measure of the relative predictive power of statistical models in terms of information lost when adopting the model, and takes into account both the goodness-of-fit of the model, and the model complexity.

To better investigate which aspects of the created levels are most likely to lead to advanced solutions from later players, I created a binomial regression model on the data set, predicting whether a solution to a given level is advanced based on whether the level contains a correct application of loops or functions. To construct this model, I used the the add1 and drop1 functions from the *glm* package in the statistical software R. For a given linear model and data set, the *add1* function determines which independent variable from the provided set will most improve the model in terms of AIC. Similarly, the *drop1* function determines which of the terms in a given model can be dropped to improve the fit of the model in terms of AIC. To determine which model best describes the data, I started by considering a basic model including only Condition and Difference variables. I then considered a maximally-complex model including terms for every individual variable in the data set, and all two-term interactions. These terms are the six tags described in earlier chapters, as well as the value of Difference between a naive and an expert solution, and the Condition under which the level is created. I then applied step-wise regression using the *step* function in R [R C]. The step function iteratively calls two functions, add1 and drop1, until the model stabilizes. The add1 function will iteratively add interaction terms that most improve the model in terms of AIC, by increasing goodness of fit more than the model's complexity. The drop1 function will iteratively remove terms that improve the AIC by reducing complexity while not reducing goodness of fit, again improving the model. I then repeated this process, using add1 and drop1 functions to refine the model until it converged on a stable state.

7.3 Results

In the following section, each iteration of the model describing the data is presented. Parameters are presented with the same precision as the R output. Terms that are significant for a given model are denoted with a single asterisk (< 0.05), double asterisk (< 0.01), or triple asterisk (< 0.001). Terms that are "marginally" significant (< 0.1) are denoted with a single period, but this level of significance

is not considered in our results.

I began my analysis by building a simple model only including Condition and Difference factors. In this simple model, with $AIC = 283.8$, Difference accounts for most of the variance among the predictors. The analysis of variance for this basic model (shown in 7.1) shows that this model is insufficient to explain the data.

Table 7.1 Simple regression model for Condition vs. Difference in student solutions

	LR Chisq	Df	Pr(>Chisq)
Condition	0.0428	2	0.9788
Difference	3.2947	1	0.0695 .

After the first step, and adding all relevant two-term interactions, the model summary was as follows:

Table 7.2 Regression model after adding all relevant terms

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-0.95477	0.71283	-1.339	0.1804
Condition 0	0.55116	0.69059	0.798	0.4248
Condition 1	1.22127	0.73661	1.658	0.0973 .
Difference	0.14997	0.07440	2.016	0.0438 *
Obvious Pattern	1.38827	0.52592	2.640	0.0083 **
Improve With Loop/Function	-0.55124	0.76012	-0.725	0.4683
Loop/Function Solution In 5 Min	1.55592	0.78519	1.982	0.0475 *
Trivial Solution In 5 Min	-0.88334	0.50606	-1.746	0.0809 .
Difference * Obvious Pattern	-0.15700	0.08062	-1.947	0.0515 .
Condition 0 * Loop/Function Solution In 5Min	-1.00842	0.82565	-1.221	0.2219
Condition 1 * Loop/Function Solution In 5Min	-1.65776	0.84387	-1.964	0.0495 *

During this phase, terms for the interaction between Difference and Obvious Pattern were added, as well as for the interaction between the levels of Condition and the “Expert Solution Possible in 5 Minutes” tag. Note that Condition is treated by R as a “factor” so Condition 0 and Condition 1, the new Building-Block and Programming editors, are compared to the factor’s baseline, the Drag-and-Drop editor. The model’s AIC improved to 279.81 from the initial model’s 283.8. Next, I removed terms iteratively. The resulting model summary is presented below:

Table 7.3 Regression model after adding all relevant terms

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept) -	1.06399	0.29657	-3.588	0.000334 ***
Difference	0.13778	0.06642	2.074	0.038046 *
Obvious Pattern	1.35750	0.42029	3.230	0.001238 **
Difference * Obvious Pattern	-0.14137	0.07156	-1.975	0.048218 *

I then repeated this procedure until the model converged, but there were no changes to the model after this point. The final model presented above describes the data collected best of all the models I constructed, ($AIC = 272.88$). Note that the Condition factor is no longer present in the final model. Instead, the value of Difference and the presence of Obvious Patterns are predictive of increased likelihood of an advanced solution. The interaction term between the Obvious Pattern tag and the value of Difference is also predictive. I interpret this as meaning that so long as a level has Obvious Patterns and a high value of Difference, it does not particularly matter which Condition it was created under.

This observation is in line with the idea of "gameplay affordances" as presented earlier, as it indicates that a level that both *rewards* a gameplay mechanic – measured here through Difference between an unskilled and skilled solution – and *affords* that solution through the use of obvious structural patterns in the level – measured here by the expert tagging Obvious Pattern tag – will be more likely to elicit targeted solutions from players.

Though the Obvious Pattern tag was found to be a very strong predictor, I must consider that it is very rare for a BOTS level to have obvious patterns but not afford the use of the core mechanics of loops and functions. In the augmented editors (Building-Blocks and Programming), this was even more rare. Counter-examples include "Spirograph" levels that feature elaborate looping patterns but end where they begin, and levels with unintended shortcuts.

None of the other tags are retained by the final model; however, an earlier model created through a different process (with higher $AIC = 273.78$) retained the "Trivial Solution in Under 5 Minutes" tag as a predictor.

To determine the relative impact of the player's in-game performance when compared with the level editor used, I next explored the impact of player experience on the quality of levels created in each editor. I used both the authors' and players' tutorial completion in the model. 22 of these students completed the entire tutorial up to the Bot-Henge boss level. I added these terms to the model manually and performed an analysis of variance on the resulting model, shown in Table 7.4.

This model retains the same important predictors as the previous model but is comparatively

Table 7.4 Improved model for Condition vs. Difference in student solutions with author tutorial completion

	LR Chisq	Df	Pr(>Chisq)
Condition	0.0686	2	0.9663029
Difference	5.0823	1	0.0241713 *
Obvious Pattern	10.9079	1	0.0009576 ***
Finished Tutorial	0.2622	1	0.6086111
Author Finished Tutorial	0.2622	1	0.6086188
Difference * Obvious Pattern	4.1791	1	0.0409249 *

weaker ($AIC = 280.2$). This model may have been weakened due to the low campaign completion among the players in this study, as only 4 students successfully completed all 12 available levels. Additionally, authors who created levels under the Drag and Drop condition played an extended tutorial sequence that included Challenge levels, and none of the authors completed the sequence up to Bot-Henge in that condition. Therefore the predictive value of this model is diminished based on these factors. Further data collection to allow players to complete more levels may improve this model. However, using the data collected, I cannot conclude that either of these terms was a significant predictor of advanced solutions.

7.4 Conclusions

These models of player behavior in user-authored puzzles provide evidence to support that "Gameplay Affordances" are valuable in promoting puzzle solvers to use target game mechanics. The interaction between visual representation and actual reward (Obvious Patterns and Difference in number of lines of code) was a predictor of student behavior within the puzzle. This supports the idea that guiding or incentivizing authors to include gameplay affordances in their created levels is an effective way to encourage later players to make use of them. This means that the Building-Blocks and Programming editors are influencing the levels created by users in a positive way. Since the level editors improve the potential rewards and promote the creation of more levels with obvious patterns, the resulting created levels afford users more opportunities to practice their skills.

Chapter 8

Conclusions and Future Work

8.1 Review

In this chapter, I summarize how the work in the previous chapters contributed to validating my hypotheses and answering the research questions outlined in the introduction. To recap, my hypotheses and research questions and contributions for this work are as follows:

8.2 Research Questions

1. Q1: How can non-expert student authors create high-quality levels that afford and reward the use of core mechanics in BOTS?
2. Q2: How can level authoring tools designed using creative constraints help promote the creation of high-quality levels that afford and reward the use of core mechanics in BOTS?
3. Q3: How can existing data-driven methods for knowledge discovery and feedback generation from Intelligent Tutoring Systems be applied to brand new user-authored levels?

8.3 Hypotheses

1. H1: Asking authors to provide solutions to their levels before publication will increase the quality of published levels and reduce the number of submitted problems with trivial, tedious, or non-existent solutions.
2. H2: Player traces can be used to automatically generate low-level hints for player-authored problems.

3. H3: Authoring tools designed with creative constraints will result in authors more frequently creating levels that afford the game's core mechanics, when compared with the original free-form authoring tool.

4. H4: Levels that contain these gameplay affordances will more frequently contain correct use of the game's advanced mechanics in players' solutions.

8.4 Contributions

The contributions of this research are as follows:

1. BOTS, itself; a serious game for novice programmers that supports user-generated levels via a gamified level editor.

2. A novel application of Intelligent Tutoring Systems methods for knowledge discovery and hint generation to user-authored levels, and evaluation of the amount of data needed to effectively cover a user-authored problem in BOTS.

3. Design principles for level editors to improve the quality and usability of user-generated levels.

8.5 Research Question Q1

Research question Q1 is "How can non-expert student authors create high-quality levels that afford and reward the use of core mechanics in BOTS?" From my pilot work, I learned that while non-expert student authors can create high-quality levels that afford BOTS' core mechanics, the vast majority of these levels do not contain any gameplay affordances. Later modifications to the level editor tools (Building-Blocks and Programming) better allow users to create high-quality levels.

Using the two new level editors, the number of high-quality levels created by students in the same age group and with similar background with computational thinking was shown to increase. Quality in terms of affording and rewarding gameplay mechanics was measured by generating comparable solutions with and without use of those gameplay mechanics, and calculating the differences in score (number of lines of code) used in these solutions. Additionally, new patterns of level design were enabled by the new versions of the level editors (Programming and Building-Blocks) that were more likely to afford the use of the game's core mechanics than canvas-filling or experimental behaviors undertaken in the baseline, Free-form Drag-and-Drop editor.

8.6 Research Question Q2

Research question Q2 is "How can level authoring tools designed using creative constraints help promote the creation of high-quality levels that afford and reward the use of core mechanics in BOTS?" Throughout my work, I have implemented three main changes to the level editor in BOTS. First, I implemented a "Solve and Submit" system for the baseline Free-form Drag-and-Drop level editor. This Solve and Submit system required players to provide solutions to their own levels before they could be published for other players. This type of solution is present in commercial entertainment games based on UGC such as Super Mario Maker, King of Thieves, and Little Big Planet. This mechanism proved somewhat effective at improving the quality of published levels. However, it did not have a strong effect on discouraging these design patterns during level creation. This meant that this design change did not prevent players from wasting their time creating levels that should never be published, or levels that don't address the targeted learning objectives of loops and functions.

Next, I designed two alternative level editors, based on two different sets of design principles: one taken from the Deep Gamification framework, and one derived from existing tutoring systems where the results of problem-posing activities are intended to be incorporated into the system. For the first Deep Gamification approach, I designed a Programming Editor where students create levels by programming, using precisely the same interface as they would when playing a BOTS level. For the second Problem-posing approach, I designed a Building-Blocks level editor, where users construct levels from sets of pre-built "building-blocks" that are either known to contain gameplay affordances, or that by their repetition would create a gameplay affordance. I evaluated these editors against the existing Drag-and-Drop editor using a zero-inflation model. I used this model because my pilot work uncovered negative design patterns that necessarily result in specific types of zero-affordance levels, which are different from other low-affordance levels. Using a zero-inflation model allowed me to determine if any editor was more likely to produce a zero result, and if any editor was more likely to produce a high-value result in terms of score improvement between naive and expert solutions. The results showed that the Building-Blocks editor was the least likely to produce excess-zero results that correspond to zero-affordance negative design patterns. Puzzles created in either of the new editors were more likely than those created in the Drag-and-Drop editor to produce a large difference between a naive and an expert solution, with the Programming Editor producing the largest differences. My interpretation is that the "ceiling" of expression for skilled users is higher in the Programming Editor. However, since it is much more difficult to create undesirable zero-affordance levels using the Building-Blocks editor, it may be best to use the Building-Blocks first with novice users.

8.7 Research Question Q3

Research question Q3 is "How can existing data-driven methods for knowledge discovery and feedback generation from Intelligent Tutoring Systems be applied to brand new user-authored levels?" For this question, I wanted to test the feasibility of using the Hint-Factory method for generating low-level next step hints on user-generated content. Specifically, I wanted to find out how many user records are needed before this method can be used to reliably provide hints for a new user-generated level. To determine this, I used the tutorial and campaign levels in the corpus with the highest number of plays across all players, to estimate how many plays of a new level are needed. I also tested a generalized version of the algorithm that compared states based on their output (world-states) rather than their program contents (code-states). I discovered that using Hint Factory on world-states requires approximately 30 records for puzzles of similar complexity to the levels examined from the campaign. Based on the sparsity of player exploration of existing levels, I believe the number of records needed is still too high in a case where players freely select levels to play. However, if students are guided towards similar levels via a "campaign" ordering used in Chapter 8, this number of records can be collected quickly.

8.8 Hypothesis H1

Hypothesis H1 is "Asking authors to provide solutions to their levels before publication will increase the quality of published levels and reduce the number of submitted problems with trivial, tedious, or non-existent solutions." H1 is supported by work in Chapters 3 and 4. This work showed that the solve-and-submit condition did increase the quality of published levels. However, the effect on all submitted levels was unclear. This result, coupled with the burden of moderating even a small sample of puzzles, was sufficient to exclude human moderation from future experiments.

8.9 Hypothesis H2

Hypothesis H2 is "Player traces can be used to automatically generate low-level hints for player-authored problems." H2 is supported by work in Chapter 5. This work showed that, with some modifications to find equivalent states by output rather than code, the Hint Factory method will work for this domain, and that a general guideline of 30 students is needed to be able to meet the advised 80% threshold in cases where levels are of a similar complexity to those in the tutorial and campaign. Tools designed to use this method will need to take this requirement into account when selecting how to present new or underplayed user-authored levels to users.

8.10 Hypothesis H3

Hypothesis H3 is "Authoring tools designed with creative constraints will result in authors more frequently creating levels that afford the game's core mechanics, when compared with the original free-form authoring tool." H3 is supported by work in Chapter 6. Both new Building-Blocks and Programming editors performed better than the free-form Drag-and-drop editor at producing levels with a greater improvement between the naive and expert solutions. The Programming editor produced the greatest value of improvement, and the Building-Blocks editor was most likely to produce a non-zero result. This suggests that the Building-Blocks editor is the safer choice for new users, while the Programming Editor may allow experienced or adventurous users more room to improve their creations.

8.11 Hypothesis H4

Hypothesis H4 is "Levels that contain these gameplay affordances will more frequently contain correct use of the game's advanced mechanics in players' solutions." H4 is supported by work in Chapter 7. Specifically, while the in-game reward (in terms of saved lines of code) was an important predictor of an advanced solution, expert-identified obvious patterns were a stronger predictor when added to the model, and the combination of these patterns with a high reward was also a valuable predictor. This means that levels constructed with gameplay affordances (gameplay reward alongside obvious visual signifiers or clues) are most important.

8.12 Conclusions

Based on my initial analysis of user-authored puzzles, I identified several characteristics common to these puzzles that I wished to minimize or eliminate. I used these characteristics to group levels into common categories: "Sandbox" puzzles exhibiting canvas-filling behavior or random placement of objects, "Power-gamer" puzzles whose goals are difficult in terms other than the game's core mechanic, "Trivial" puzzles that do not afford the core mechanic of the game, and "Griefer" puzzles that are designed to be impossible or frustrating. I characterized created levels using these characteristics, and devised potential changes to the level editor in order to help users avoid creating levels with these characteristics.

My first experiment was a pilot study comparing an entirely free-form Drag-and-drop version of the editor to versions with human moderation and with self-moderation in the form of a "Solve-and-Submit" system. My theory was that integrating gameplay in the creation process by requiring a user

to complete their own levels would inhibit the creation of all negative design patterns described previously, especially the “Griefer” and “Power-Gamer” levels. This has been shown to be true. However, the “Solve-and-Submit” system did not strongly inhibit submission of “Trivial” levels. Additionally, some quality levels were also never submitted under this condition. I believe this occurred because of the large amount of additional effort required to submit a level after it had been created. To address this, I began examining ways that gameplay objectives could be better integrated throughout a player’s design process.

I designed two new versions of the level editor to achieve this goal. First was a “Building-Blocks” editor, where players construct levels from “building-blocks” representing segments of previously-designed levels where loops or functions were useful in improving a player’s score. This design was based on problem-posing tutors, where students built problems for other students to solve from expert-curated pieces, but did not themselves solve these problems [HK11][BC12]. Second was a “Programming” editor where users write code, using the main game’s programming interface, that becomes the solution to the level. As the robot moves, the level is generated beneath it to match its motion. This level editor adhered to the principles of the Deep Gamification framework that promotes high integration between a game’s objectives and all the mechanics that players apply during gameplay [Boy14].

Expert analysis of the levels generated by users under these level-editor conditions showed that the Building-Blocks editor was the most effective at avoiding zero-improvement results (usually “Trivial" levels) among the three conditions. However, the Programming editor encouraged a high value for “reward”, that is, the number of lines of code saved by using loops or functions. When coupled with the presence of obvious visual patterns, the high value of “reward” indicates the presence of gameplay affordances in the level. Follow-up analysis confirmed that these rewards and obvious patterns are important predictors of whether or not a puzzle’s players will use advanced gameplay concepts in their solutions. The obvious patterns were found to be more influential than the reward value, and both were influential in tandem.

One concern about using user-generated puzzles in the game was the lack of expert in-game feedback. The expert-authored levels in the game have some specific built-in player feedback. In outreach sessions, experts were present and familiar enough with the tutorial levels to be able to offer players personalized assistance. This was not, and will not be, the case with user-authored levels. While expert knowledge can be embedded in the tutorial levels by providing hints, it is too costly to do this for even the relatively small volume of user-authored puzzles, let alone for all of those created across multiple classrooms or schools. To address this, I investigated the Hint Factory method of data-driven feedback generation. I wanted to see if this method could be adapted to the BOTS environment, since BOTS is significantly different than the Hint Factory’s original domain

of propositional logic. I also wanted to determine the amount of data needed to provide a high probability of hints being available on any single player step. I found that 30 students is a good estimate for the number of students needed to achieve 80% hint coverage in BOTS. This is not a large number, but I observed that students without direction tended to play only levels from their own peer group. This means that older levels tend to be ignored. Therefore, some directed presentation of levels would be needed to ensure that each level will have sufficient data for hint generation if this technique is to be used with user-authored puzzles.

From this work, I have derived the following design guidelines for building user-authoring tools in a puzzle game:

- *Account for griefing behavior.* Some users are likely to want to create impossible or frustrating levels. Griefing behavior is documented in commercial games like Super Mario Maker [Nin08], Little Big Planet [Med08] and Light-Bot [Yar08], and has been identified as one of the major negative patterns to reduce or eliminate. I have addressed this behavior by requiring a solution before level publication, and by strictly limiting the types of patterns that can be placed in the Building-Blocks editor.

- *Account for canvas-filling behavior.* Do not allow users access to more space or more game elements than they need, in creating the types of levels you expect. Canvas-filling behavior is documented in commercial games like Super Mario Maker [Nin08], Little Big Planet [Med08] and Light-Bot [Yar08], and has been identified as one of the major negative patterns to reduce or eliminate.

- *Build core mechanics / game objectives into authoring tools.* Many negative patterns of level design result from the discrepancy between the work required to make a puzzle, and work required to complete it. This leads to patterns that are easy to build but difficult or tedious to solve.

- *Integrate core mechanics throughout the design process.* My first attempt to 'integrate' game objectives into the design process was to implement a "Solve & Submit" system similar to the system present in Super Mario Maker [Nin08]. This has been effective at reducing some types of submitted levels, particularly exceptionally difficult or impossible levels. However, the submitted levels are similarly likely to be trivial or canvas-filling, because providing a solution is not a significant barrier by definition. To address these types of submissions, an editor that requires gameplay throughout the design process is more effective, as creating these types of structures becomes MORE difficult than solving a level that contains them.

- *Intelligently present levels.* I have found that 30 solutions are required, as a baseline estimate, to provide data-driven next-step hints 80% of the time. While this is a small number of users, actual patterns of gameplay indicate that users are more likely to play recent levels, and levels within their classroom peer-groups. This implies that without some direction, levels are unlikely to reach this threshold. In a system designed to use data-driven feedback for user-authored levels, additional work must be done to ensure that a sufficient number of players attempt each level in order for hint generation to achieve sufficient coverage.

8.13 Future Work

The first step in expanding this research is to continue the experiments in a classroom environment. Additional classroom gameplay data will help measure gameplay patterns to determine how quickly the 30-student threshold for Hint Factory can be reached in practice. Additionally, this will allow us to better identify specific patterns in levels that affect player behavior. Of particular interest are patterns that increase the likelihood that a student will write an advanced solution, or that a student will fail to complete the task entirely. Our ongoing work with Interaction Networks in games like Quantum Spectre has led to techniques for uncovering problem regions in the solution state space, determining which ones are especially treacherous, and determining which ones are the most straightforward for users to navigate [Hic16b; Eag15]. I have already completed some preliminary work with intelligent problem ordering, using the q-matrix method to derive a content model for levels in BOTS. However, creating a content model also requires some existing data, so some bootstrapping is needed.

Other UGC-based systems have found success at improving the quality of submitted content with recommender-system-like social mechanics like voting and ranking. These mechanics may also be useful for directing players to play similar levels. Another possibility is to use this user feedback to evaluate levels. Users can be offered in-game incentives to play newly-created levels, and answer short yes/no prompts to help evaluate the levels after completion. It is also possible to use gamified rewards to encourage collaboration or peer instruction, with players offering advice to level creators on how to improve their levels to make them more fun, difficult, or relevant. This approach was implemented by Morschheuser et al. in their gamified collaboration tool designed for an innovation community [Mor17]. In this system, players are rewarded for providing comments on others' ideas. Rewards consist of graphical flourishes added to a virtual communal space. In existing UGC-driven games like Mario Maker and Little Big Planet, a competitive gamification method has often resulted in novel content like automatic levels or exceptionally difficult levels, rather than content that rewards the use of existing mechanics. The application of a collaborative or cooperative

gamification method may help avoid these pitfalls, particularly when compared with “BLAP” or competitive gamification techniques.

I am also interested in intelligently directing content generation using in-game incentives, not only to fill gaps in the difficulty curve and content spectrum, but also to provide more personalized experiences. As cited earlier, work done by Aleahmad [Ale08] shows that content creators spend more time creating content when they are creating it for a specific individual’s use, even if that individual was unknown or fictional. There was no change in general quality as evaluated by experts, and since the target users were entirely fictional their opinions of the tailored content were not available. One extension of BOTS could allow content creators to create hints for existing levels in the form of partial solutions, or entire, smaller “hint levels”. These hint levels can be structured to provide guidance to players who are having trouble in a specific circumstance. For example, a player who is having trouble completing a complicated level involving nested loops can request help, and an expert player can respond by creating a simpler level with scaffolding code to help the struggling player grasp the concept. Such hint levels would help students develop the target skills needed to complete the harder levels [DB17].

Finally, I have observed that players transform the BOTS game into a social experience even where no in-game social mechanics exist. Players communicate with each other via level names either directly (like “Joel Play This" or “I dare u") or indirectly by naming their levels in response to each other (for example “Impossible" and “REALLY Impossible"), challenging each other back and forth to create bigger, better things. Work by Ogan and Finkelstein with partners and strangers in an intelligent tutoring system found that this type of “face threat” had positive effects on learning between friends, but negative effects between strangers [Oga12]. Adding more tools for friendly players to communicate during gameplay and level creation might have a positive impact on their play experience. Such tools can even allow for information to flow more freely between users, enhancing learning gains. It is possible to test two versions of the game; an “isolated” version with all usernames removed, where content is still created and shared but not socially, and a “connected” version where players can view not only who made a level, but who played it, and how they liked it via tags or comments. I believe that adding this level of connectivity between users will increase users’ engagement and their sense of ownership of their content, but it may promote off-task socializing behavior. Social aspects also present several additional design challenges, as outlined by the developers of Dragon Architect when considering pair-programming-like activities within their game [Bau17].

BIBLIOGRAPHY

[Ada12] Adams, D. M. et al. "Narrative games for learning: Testing the discovery and narrative hypotheses." *Journal of Educational Psychology* **104**.1 (2012), p. 235.

[Ale08] Aleahmad, T. et al. "Open community authoring of targeted worked example problems". *Proceedings of the ninth International Conference on Intelligent Tutoring Systems (ITS 2008)*. Springer. 2008, pp. 216–227.

[AA10] Alexander, C. M. & Ambrose, R. C. "Digesting Student-Authored Story Problems." *Mathematics Teaching in the Middle School* **16**.1 (2010), pp. 27–33.

[And95] Anderson, J. R. et al. "Cognitive Tutors: Lessons Learned". *The Journal of the Learning Sciences* **4**.2 (1995), pp. 167–207.

[AW03] Arroyo, I. & Woolf, B. P. "Students in AWE: changing their role from consumers to producers of ITS content". *Proceedings of the 11th International Conference on Artificial Intelligence and Education*. Citeseer. 2003.

[Bac08] Bacher, D. "Design patterns in level design: common practices in simulated environment construction" (2008).

[Bar12] Barnes, T. "AP CS Principles Pilot at University of North Carolina at Charlotte". *ACM Inroads* **3**.2 (2012), pp. 64–66.

[BS10] Barnes, T. & Stamper, J. C. "Automatic Hint Generation for Logic Proof Tutoring Using Historical Data." *Journal of Educational Technology & Society* **13**.1 (2010), pp. 3–12.

[Bar96] Bartle, R. "Hearts, clubs, diamonds, spades: Players who suit MUDs". *Journal of MUD research* **1**.1 (1996), p. 19.

[Bar04] Bartle, R. A. *Designing virtual worlds*. New Riders, 2004.

[BR10] Basawapatna, A. R. & Repenning, A. "Cyberspace meets brick and mortar: an investigation into how students engage in peer to peer feedback using both cyberlearning and physical infrastructures". *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. ACM. 2010, pp. 184–188.

[Bas10] Basawapatna, A. R. et al. "Using scalable game design to teach computer science from middle school to graduate school". *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. ACM. 2010, pp. 224–228.

[Bau17] Bauer, A. et al. "Dragon architect: open design problems for guided learning in a creative computational thinking sandbox game". *Proceedings of the twelfth International Conference on the Foundations of Digital Games (FDG 2017)*. ACM. 2017, p. 26.

[Bay09] Bayliss, J. D. "Using games in introductory courses: tips from the trenches". *ACM SIGCSE Bulletin*. Vol. 41. 1. ACM. 2009, pp. 337–341.

[BC12] Beal, C. R., Cohen, P. R., et al. "Teach Ourselves: Technology to Support Problem Posing in the STEM Classroom". *Creative Education* **3**.04 (2012), p. 513.

[BB08] Birch, M. & Beal, C. R. "Problem Posing in AnimalWatch: An Interactive System for Student-Authored Content." *FLAIRS Conference*. 2008, pp. 397–402.

[Blo84] Bloom, B. S. "The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring". *Educational Researcher* **13**.6 (1984), pp. 4–16.

[Bog11] Bogost, I. *Gamification Is Bullshit*. 2011. URL: `http://bogost.com/writing/blog/gamification_is_bullshit/`.

[BD08] Bornat, R., Dehnadi, S., et al. "Mental models, consistency and programming aptitude". *Proceedings of the tenth conference on Australasian computing education-Volume 78*. Australian Computer Society, Inc. 2008, pp. 53–61.

[Bou10] Bourgonjon, J. et al. "StudentsâĂŹ perceptions about the use of video games in the classroom". *Computers & Education* **54**.4 (2010), pp. 1145–1156.

[Boy11] Boyce, A. et al. "BeadLoom Game: adding competitive, user generated, and social features to increase motivation". *Proceedings of the sixth International Conference on Foundations of Digital Games (FDG 2011)*. ACM. 2011, pp. 139–146.

[Boy14] Boyce, A. K. "Deep Gamification: Combining Game-based and Play-based Methods." (2014).

[Boy15] Boyer, K. et al. "ENGAGE: A Game-based Learning Environment for Middle School Computational Thinking". *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM. 2015, pp. 440–440.

[BR12] Brennan, K. & Resnick, M. "New frameworks for studying and assessing the development of computational thinking". *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*. 2012, pp. 1–25.

[Bro09] Brockmyer, J. H. et al. "The development of the Game Engagement Questionnaire: A measure of engagement in video game-playing". *Journal of Experimental Social Psychology* **45**.4 (2009), pp. 624–634.

[BK12] Byron, K. & Khazanchi, S. *Rewards and creative performance: a meta-analytic test of theoretically derived hypotheses.* 2012.

[Cai13] Cai, J. et al. "Mathematical problem posing as a measure of curricular effect on students' learning". *Educational Studies in Mathematics* **83**.1 (2013), pp. 57–69.

[Cas07] Caspersen, M. E. et al. "Mental models and programming aptitude". *ACM SIGCSE Bulletin*. Vol. 39. 3. ACM. 2007, pp. 206–210.

[Cha12] Chang, K.-E. et al. "Embedding game-based problem-solving phase into problem-posing system for mathematics learning". *Computers & Education* **58**.2 (2012), pp. 775–786.

[Chi01] Chi, M. T. et al. "Learning from human tutoring". *Cognitive Science* **25**.4 (2001), pp. 471–533.

[Coo00] Cooper, S. et al. "Alice: a 3-D tool for introductory programming concepts". *Journal of Computing Sciences in Colleges*. Vol. 15. 5. Consortium for Computing Sciences in Colleges. 2000, pp. 107–116.

[Cor01] Corbett, A. "Cognitive computer tutors: Solving the two-sigma problem". *User Modeling 2001*. Springer, 2001, pp. 137–147.

[Cre03] Crespo, S. "Learning to pose mathematical problems: Exploring changes in preservice teachers' practices". *Educational Studies in Mathematics* **52**.3 (2003), pp. 243–270.

[Csi04] Csiksczentmihalyi, M. et al. "Flow: The psychology of optimal experience". *Australian Occupational Therapy Journal* **51**.1 (2004), pp. 3–12.

[DT13] Dahlskog, S. & Togelius, J. "Patterns as Objectives for Level Generation". *Proceedings of the Second Workshop on Design Patterns in Games;* ACM. 2013.

[DC00] De Corte, E. et al. "Computer-supported collaborative learning of mathematical problem solving and problem posing". *Proceedings of Conference on Educational Uses of Information and Communication Technologies. 16th World Computer Congress 2000*. 2000.

[Dec99] Deci, E. L. et al. "A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation." *Psychological bulletin* **125**.6 (1999), p. 627.

[Dec01] Deci, E. L. et al. "Extrinsic rewards and intrinsic motivation in education: Reconsidered once again". *Review of educational research* **71**.1 (2001), pp. 1–27.

[Det11a] Deterding, S. "Situated motivational affordances of game elements: A conceptual model". *Gamification: Using Game Design Elements in Non-Gaming Contexts, a workshop at CHI*. 2011.

[Det11b] Deterding, S. et al. "From game design elements to gamefulness: defining gamification". *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*. ACM. 2011, pp. 9–15.

[Det13] Deterding, S. et al. "Designing gamification: creating gameful and playful experiences". *CHI'13 Extended Abstracts on Human Factors in Computing Systems*. ACM. 2013, pp. 3263–3266.

[DB17] Dong, Y. & Barnes, T. "Evaluation of a template-based puzzle generator for an educational programming game". *Proceedings of the twelfth International Conference on the Foundations of Digital Games (FDG 2017)*. ACM. 2017, p. 40.

[Dor12] Doran, K. et al. "Outreach for improved student performance: a game design and development curriculum". *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM. 2012, pp. 209–214.

[Eag12] Eagle, M. et al. "Interaction Networks: Generating High Level Hints Based on Network Community Clusterings." *Proceedings of the fifth International Conference on Educational Data Mining (EDM 2012)*. 2012, pp. 164–167.

[Eag13] Eagle, M. et al. "Exploring player behavior with visual analytics." *Proceedings of the eighth International Conference on the Foundations of Digital Games (FDG 2013)*. 2013, pp. 380–383.

[Eag14] Eagle, M. et al. "Exploration of Student's Use of Rule Application References in a Propositional Logic Tutor". *Educational Data Mining 2014*. 2014.

[Eag15] Eagle, M. et al. "Exploring Problem-Solving Behavior in an Optics Game." *Proceedings of the eighth International Conference on Educational Data Mining (EDM 2015)*. 2015, pp. 584–585.

[Eng97] English, L. D. "Promoting a Problem-Posing Classroom." *Teaching Children Mathematics* **4**.3 (1997), pp. 172–79.

[Fos09] Fossati, D. et al. "I learn from you, you learn from me: How to make iList learn from students." *AIED*. 2009, pp. 491–498.

[Gar94] Garfield, R. *RoboRally*. [Board Game]. 1994.

[Gar02] Garris, R. et al. "Games, motivation, and learning: A research and practice model". *Simulation & gaming* **33**.4 (2002), pp. 441–467.

[Gee07] Gee, J. P. *What video games have to teach us about learning and literacy.: Revised and Updated Edition*. Macmillan, 2007.

[Gee09] Gee, J. P. "Deep learning properties of good digital games: How far can they go". *Serious games: Mechanisms and effects* (2009), pp. 67–82.

[Geh10] Gehringer, E. et al. "Motivating effective peer review with extra credit and leaderboards". *American Society for Engineering Education*. American Society for Engineering Education. 2010.

[GM09] Gehringer, E. F. & Miller, C. S. "Student-generated active-learning exercises". *ACM SIGCSE Bulletin*. Vol. 41. 1. ACM. 2009, pp. 81–85.

[Geh16] Gehringer, E. F. et al. "What Peer-Review Systems Can Learn from Online Rating Sites". *State-of-the-Art and Future Directions of Smart Learning*. Springer, 2016, pp. 341–350.

[Gib14] Gibson, J. J. "The theory of affordances". *GIESEKING, Jen Jack; MANGOLD, William; KATZ, Cindi; LOW, Setha* (2014), pp. 56–60.

[Gib66] Gibson, J. J. "The senses considered as perceptual systems." (1966).

[Gou13] Gouws, L. A. et al. "Computational thinking in educational activities: an evaluation of the educational game light-bot". *Proceedings of the 18th ACM conference on Innovation and technology in computer science education (ITICSE 2018)*. ACM. 2013, pp. 10–15.

[Gre10] Greenberg, B. S. et al. "Orientations to video games among gender and age groups". *Simulation & Gaming* **41**.2 (2010), pp. 238–259.

[HK06] Hartmann, T. & Klimmt, C. "Gender and computer games: Exploring femalesâĂŹ dislikes". *Journal of Computer-Mediated Communication* **11**.4 (2006), pp. 910–931.

[Har14] Harvey, B. et al. "Snap!(build your own blocks)". *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM. 2014, pp. 749–749.

[Has87] Hashimoto, Y. "Classroom practice of problem solving in Japanese elementary schools". *Proceedings of the US-Japan seminar on mathematical problem solving. Columbus, OH: ERIC/SMEAC Clearinghouse (ED 304 315)*. ERIC. 1987, pp. 94–119.

[Her15] Hernandez, P. *Mario Maker players are in arms race to make the hardest level ever*. 2015. URL: `http://kotaku.com/1739431432` (visited on 11/13/2017).

[Hic14a] Hicks, A. et al. "Building Games to Learn from Their Players: Generating Hints in a Serious Game". *Proceedings of the twelfth International Conference on Intelligent Tutoring Systems (ITS 2014)*. Springer. 2014, pp. 312–317.

[Hic14b] Hicks, A. et al. "Part of the Game: Changing Level Creation to Identify and Filter Low Quality User-Generated Levels" (2014).

[Hic16a] Hicks, A. et al. "Measuring Gameplay Affordances of User-Generated Content in an Educational Game". *International Conference on Educational Data Mining*. 2016.

[Hic15] Hicks, D. et al. "Applying Deep Gamification Principles to Improve Quality of User-Designed Levels". *Proceedings of the 11th international conference on Games + Learning + Society (GLS11)*. 2015.

[Hic16b] Hicks, D. et al. "Using Game Analytics to Evaluate Puzzle Design and Level Progression in a Serious Game". *Proceedings of the Sixth International Conference on Learning Analytics & Knowledge (LAK 2016)*. Edinburgh, United Kingdom: ACM, 2016, pp. 440–448.

[Hil10] Hill, B. M. et al. "Responses to Remixing on a Social Media Sharing Website." *ICWSM*. 2010.

[HK11] Hirashima, T. & Kurayama, M. "Learning by problem-posing for reverse-thinking problems". *Artificial Intelligence in Education*. Springer. 2011, pp. 123–130.

[Hir07] Hirashima, T. et al. "Learning by problem-posing as sentence-integration and experimental use". *AIED*. Vol. 2007. 2007, pp. 254–261.

[Hua13] Huang, J. et al. "Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC". *AIED 2013 Workshops Proceedings Volume*. 2013, p. 25.

[Hui55] Huizinga, J. *Homo Ludens: A Study of the Play-elememt in Culture*. Beacon Press, 1955.

[Hul] Hullett, K. "Cause-effect relationships between design patterns and designer intent in fps levels". *Proceedings of the Second Workshop on Design Patterns in Games (DPG 2013), ser. DPG*. Vol. 13.

[HW10] Hullett, K. & Whitehead, J. "Design patterns in FPS levels". *proceedings of the fifth International Conference on the Foundations of Digital Games (FDG 2010)*. ACM. 2010, pp. 78–85.

[Jin08] Jin, W. "Pre-programming analysis tutors help students learn basic programming concepts". *ACM SIGCSE Bulletin*. Vol. 40. 1. ACM. 2008, pp. 276–280.

[Jin12] Jin, W. et al. "Program representation for automatic hint generation for a data-driven novice programming tutor". *Proceedings of the eleventh International Conference on Intelligent Tutoring Systems (ITS 2012)*. Springer. 2012, pp. 304–309.

[Juu10] Juul, J. "In search of lost time: on game goals and failure costs". *Proceedings of the fifth International Conference on the Foundations of Digital Games (FDG 2010)*. ACM. 2010, pp. 86–91.

[JN09] Juul, J. & Norton, M. "Easy to use and incredibly difficult: on the mythical border between interface and gameplay". *Proceedings of the fourth International Conference on the Foundations of Digital Games (FDG 2009)*. ACM. 2009, pp. 107–112.

[Kap12] Kapp, K. M. *The gamification of learning and instruction: game-based methods and strategies for training and education*. John Wiley & Sons, 2012.

[Kel07] Kelleher, C. et al. "Storytelling alice motivates middle school girls to learn computer programming". *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 2007, pp. 1455–1464.

[Ker08] Kereki, I. F. de. "Scratch: Applications in computer science 1". *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*. IEEE. 2008, T3B–7.

[Koe08] Koenig, A. D. *Exploring effective educational video game design: The interplay between narrative and game-schema construction*. ProQuest, 2008.

[Köl10] Kölling, M. "The Greenfoot programming environment". *ACM Transactions on Computing Education (TOCE)* **10**.4 (2010), p. 14.

[Kra15] Krause, M. et al. "A playful game changer: Fostering student retention in online education with social gamification". *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. ACM. 2015, pp. 95–102.

[Lar86] Larkin, S. "Word problems for kids by kids". *Teaching mathematics: strategies that work* (1986).

[Les14] Lester, J. C. et al. "Designing game-based learning environments for elementary science education: A narrative-centered learning perspective". *Information Sciences* **264** (2014), pp. 4–18.

[Lew12] Lewis, C. et al. "Motivational game design patterns of'ville games". *Proceedings of the seventh International Conference on the Foundations of Digital Games (FDG 2012)*. ACM. 2012, pp. 172–179.

[Liu11] Liu, Y. et al. "Gamifying Intelligent Environments". *Proceedings of the 2011 International ACM Workshop on Ubiquitous Meta User Interfaces*. Ubi-MUI '11. New York, NY, USA: ACM, 2011, pp. 7–12.

[LA14] Long, Y. & Aleven, V. "Gamification of Joint Student/System Control over Problem Selection in a Linear Equation Tutor". *Proceedings of the twelfth International Conference on Intelligent Tutoring Systems (ITS 2014)*. Springer. 2014, pp. 378–387.

[ML07] Malan, D. J. & Leitner, H. H. "Scratch for budding computer scientists". *ACM SIGCSE Bulletin* **39**.1 (2007), pp. 223–227.

[Mat00] Mathematics, N. C. of Teachers of. *Principles and standards for school mathematics*. Vol. 1. Natl Council of Teachers of, 2000.

[MM10] Mawhorter, P. & Mateas, M. "Procedural level generation using occupancy-regulated extension". *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE. 2010, pp. 351–358.

[Max00] Maxis. *The Sims*. [Video Game]. Electronic Arts, 2000.

[Max08] Maxis. *Spore*. [Video Game]. Electronic Arts, 2008.

[May14] Mayer, R. E. *Computer games for learning: An evidence-based approach*. MIT Press, 2014.

[MJ10] Mayer, R. E. & Johnson, C. I. "Adding instructional features that promote learning in a game-like environment". *Journal of Educational Computing Research* **42**.3 (2010), pp. 241–265.

[McE15] McElroy, G. *Super Mario Maker Review: The Blueprint*. 2015. URL: `https://www.polygon.com/2015/9/2/9190435/` (visited on 11/13/2017).

[Med08] Media Molecule. *LittleBigPlanet*. [Video Game]. Sony Computer Entertainment, 2008.

[Mek13] Mekler, E. D. et al. "Disassembling gamification: the effects of points and meaning on user motivation and performance". *CHI'13 Extended Abstracts on Human Factors in Computing Systems*. ACM. 2013, pp. 1137–1142.

[Mil16] Millis, K. et al. "The Impact of Game-Like Features on Learning from an Intelligent Tutoring System". *Technology, Knowledge and Learning* (2016), pp. 1–22.

[Moj11] Mojang. *Minecraft*. [Video Game]. Microsoft, 2011.

[MHH10] Monroy-Hernández, A. & Hill, B. M. "Cooperation and attribution in an online community of young creators". *Computer* (2010), pp. 469–470.

[Mor17] Morschheuser, B. et al. "Designing Cooperative Gamification: Conceptualization and Prototypical Implementation." *Proceedings of the twentieth ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2017)*. 2017, pp. 2410–2421.

[Mur03] Murray, T. "An Overview of Intelligent Tutoring System Authoring Tools: Updated analysis of the state of the art". *Authoring tools for advanced technology learning environments*. Springer, 2003, pp. 491–544.

[MA02] Murray, T. & Arroyo, I. "Toward measuring and maintaining the zone of proximal development in adaptive instructional systems". *Proceedings of the sixth International Conference on Intelligent Tutoring Systems (ITS 2002)*. Springer. 2002, pp. 749–758.

[Nad04] Nadeo. *TrackMania*. [Video Game]. Enlight, 2004.

[Nic12] Nicholson, S. "A user-centered theoretical framework for meaningful gamification". *Proceedings of the 8th international conference on Games + Learning + Society (GLS8)* **8** (2012).

[Nin08] Nintendo. *Super Mario Maker*. [Video Game]. Nintendo, 2008.

[Nor88] Norman, D. A. *The psychology of everyday things*. Basic books, 1988.

[Oga12] Ogan, A. et al. "Rudeness and Rapport: Insults and Learning Gains in Peer Tutoring." *Proceedings of the eleventh International Conference on Intelligent Tutoring Systems (ITS 2012)*. Springer. 2012, pp. 11–21.

[PH91] Papert, S. & Harel, I. "Situating constructionism". *Constructionism* **36**.2 (1991), pp. 1–11.

[PI] Peddycord III, B. et al. "Generating Hints for Programming Problems Using Intermediate Output" ().

[PH09] Plass, J. L. & Homer, B. D. "Educational game design pattern candidates". *Journal of Research in Science Teaching* **44**.1 (2009), pp. 133–153.

[Pla11] Plass, J. L. et al. "Learning mechanics and assessment mechanics for games for learning". *G4LI White Paper* **1** (2011), p. 2011.

[Pre03] Prensky, M. "Digital game-based learning". *Computers in Entertainment (CIE)* **1**.1 (2003), pp. 21–21.

[Pre05] Prensky, M. "Computer games and learning: Digital game-based learning". *Handbook of computer game studies* **18** (2005), pp. 97–122.

[PB15] Price, T. W. & Barnes, T. "Comparing textual and block interfaces in a novice programming environment". *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM. 2015, pp. 91–99.

[R C] R Core Team. *R Documentation Add or Drop All Possible Single Terms to a Model*. URL: `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/add1.html` (visited on 08/05/2017).

[Raz05] Razzaq, L et al. "The Assistment project: Blending assessment and assisting". *Proceedings of the 12th Annual Conference on Artificial Intelligence in Education* (2005), pp. 555–562.

[Rep09] Repenning, A. et al. "Making university education more like middle school computer club: facilitating the flow of inspiration". *Proceedings of the 14th Western Canadian Conference on Computing Education*. ACM. 2009, pp. 9–16.

[Res96] Resnick, M. "StarLogo: An environment for decentralized modeling and decentralized thinking". *Conference companion on Human factors in computing systems*. ACM. 1996, pp. 11–12.

[Res09] Resnick, M. et al. "Scratch: programming for all". *Communications of the ACM* **52**.11 (2009), pp. 60–67.

[RK13] Rivers, K. & Koedinger, K. R. "Automatic Generation of Programming Feedback: A Data-Driven Approach". *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*. 2013, p. 50.

[RH12] Rizvi, M. & Humphries, T. "A Scratch-based CS0 course for at-risk computer science majors". *2012 Frontiers in Education Conference Proceedings*. IEEE. 2012, pp. 1–5.

[Rob10a] Robertson, M. "Can't play, won't play". *Hide&Seek: Inventing new kinds of play* **2013** (2010).

[Rob10b] Robins, A. "Learning edge momentum: A new account of outcomes in CS1". *Computer Science Education* **20**.1 (2010), pp. 37–71.

[Rou05] Routledge, R. "Fisher's exact test". *Encyclopedia of biostatistics* (2005).

[SZ04] Salen, K. & Zimmerman, E. *Rules of play*. MIT press, 2004.

[Sha06] Shaffer, D. W. *How computer games help children learn*. Macmillan, 2006.

[Shu13] Shute, V. et al. "–Modeling Student Competencies in Video Games Using Stealth Assessment". *Design recommendations for intelligent tutoring systems* **1** (2013), pp. 141–152.

[Shu08] Shute, V. J. "Focus on formative feedback". *Review of educational research* **78**.1 (2008), pp. 153–189.

[Sil13] Silver, E. A. "Problem-posing research in mathematics education: looking back, looking around, and looking ahead". *Educational Studies in Mathematics* **83**.1 (2013), pp. 157–162.

[SC96] Silver, E. A. & Cai, J. "An analysis of arithmetic problem posing by middle school students". *Journal for Research in Mathematics Education* (1996), pp. 521–539.

[Smi12] Smith, A. M. et al. "A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game". *Proceedings of the seventh International Conference on the Foundations of Digital Games (FDG 2012)*. New York, NY, USA: ACM, 2012, pp. 156–163.

[Smi13] Smith, A. M. et al. "Quantifying over play: Constraining undesirable solutions in puzzle design." *Proceedings of the eighth International Conference on the Foundations of Digital Games (FDG 2013)*. 2013, pp. 221–228.

[Smi10] Smith, G. et al. "Tanagra: A mixed-initiative level design tool". *Proceedings of the fifth International Conference on the Foundations of Digital Games (FDG 2010)*. ACM. 2010, pp. 209–216.

[Smi11a] Smith, G. et al. "Launchpad: A rhythm-based level generator for 2-d platformers". *IEEE Transactions on computational intelligence and AI in games* **3**.1 (2011), pp. 1–16.

[Smi11b] Smith, G. et al. "PCG-based game design: enabling new play experiences through procedural content generation". *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM. 2011, p. 7.

[Smi11c] Smith, G. et al. "Situating quests: Design patterns for quest and level design in role-playing games". *International Conference on Interactive Digital Storytelling*. Springer. 2011, pp. 326–329.

[Smi09] Smith, R. *Helping Your Players Feel Smart: Puzzles as User Interface*. Game Design Track at the Game Developers Conference. 2009.

[Spi11] Spires, H. A. et al. "Problem solving and game-based learning: Effects of middle grade students' hypothesis testing strategies on learning outcomes". *Journal of Educational Computing Research* **44**.4 (2011), pp. 453–472.

[Squ13] Squire, K. D. "Video game–based learning: An emerging paradigm for instruction". *Performance Improvement Quarterly* **26**.1 (2013), pp. 101–130.

[Sta08] Stamper, J. et al. "The hint factory: Automatic generation of contextualized help for existing computer aided instruction". *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track (ITS 2008)*. 2008, pp. 71–78.

[Sta10] Stamper, J. et al. "PSLC dataShop: a data analysis service for the learning science community". *Proceedings of the tenth International Conference on Intelligent Tutoring Systems (ITS 2010)*. Springer. 2010, pp. 455–455.

[SS14] Steinkuehler, C. & Squire, K. "Videogames and learning". *Cambridge handbook of the learning sciences* (2014).

[Ste12] Steinkuehler, C. et al. *Games, learning, and society: Learning and meaning in the digital age*. Cambridge University Press, 2012.

[SF11] Stolee, K. T. & Fristoe, T. "Expressing computer science concepts through Kodu game lab". *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM. 2011, pp. 99–104.

[Tho15] Thomsen, M. *Super Mario Maker is an engine for circulating horrible new Mario levels*. 2015. URL: `https://www.washingtonpost.com/news/comic-riffs/wp/2015/09/15/` (visited on 11/13/2017).

[Tog10] Togelius, J. et al. "Multiobjective exploration of the starcraft map space". *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE. 2010, pp. 265–272.

[Tui14] Tuite, K. "GWAPs: Games with a Problem". *Proceedings of the ninth International Conference on the Foundations of Digital Games (FDG 2014)*. 2014.

[Vah14] Vahldick, A. et al. "A review of games designed to improve introductory computer programming competencies". *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 2014, pp. 1–7.

[Van06] Vanlehn, K. "The behavior of tutoring systems". *International journal of artificial intelligence in education* **16**.3 (2006), pp. 227–265.

[Vog06] Vogel, J. J. et al. "Computer gaming and interactive simulations for learning: A meta-analysis". *Journal of Educational Computing Research* **34**.3 (2006), pp. 229–243.

[VAD08] Von Ahn, L. & Dabbish, L. "Designing games with a purpose". *Communications of the ACM* **51**.8 (2008), pp. 58–67.

[WW15] Weintrop, D. & Wilensky, U. "To block or not to block, that is the question: students' perceptions of blocks-based programming". *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM. 2015, pp. 199–208.

[Wil16] Williams, J. J. et al. "AXIS: Generating Explanations at Scale with Learnersourcing and Machine Learning". *Proceedings of the Third (2016) ACM Conference on Learning@Scale*. ACM. 2016, pp. 379–388.

[Yar08] Yaroslavski, D. *LightBot*. [Video Game]. Armor Games, 2008.

[Yee06] Yee, N. "Motivations for play in online games". *CyberPsychology & behavior* **9**.6 (2006), pp. 772–775.

APPENDICES

Appendix A

A.1 Tests and Surveys

The following appendix contains copies of all additional surveys used in this work.

A.1.1 Game Engagement Questionnaire

For each of the following items, please answer “No”, “Sort Of”, or “Yes” When I am making levels in BOTS:
1 I lose track of time
2 Things seem to happen automatically
3 I feel different
4 I feel scared
5 The game feels real
6 If someone talks to me, I don’t hear them
7 I get wound up
8 Time seems to kind of stand still or stop
9 I feel spaced out
10 I don’t answer when someone talks to me
11 I can’t tell that I’m getting tired
12 Playing seems automatic
13 My thoughts go fast
14 I lose track of where I am
15 I play without thinking about how to play
16 Playing makes me feel calm
17 I play longer than I meant to
18 I really get into the game

19 I feel like I just can't stop playing

A.1.2 Demographic and Player Preference Questionnaire

Have you taken any programming classes, or have you used other games/tools for programming like Scratch, Alice, or GameMaker?
None at all - Not much - A little - A lot

How many hours of video games do you play per week?
0-5 5-10 10-20 More than that

When you were playing other players levels:
How many of them were too hard?
None - A few - A lot - Almost All

How many of them were too easy?
None - A few - A lot - Almost All

How many of them were impossible?
None - A few - A lot - Almost All

How many of them were boring?
None - A few - A lot - Almost All

How many of them were interesting?
None - A few - A lot - Almost All

When you were creating levels:

Were you able to make the levels that you wanted to make
Never - Sometimes - Most of the Time - Always

How many of your levels were actually published?
None - A few - A lot - Almost All

How many of your levels did other people play?
None - A few - A lot - Almost All

Did you like making levels?
Not at all - Not much - A little - A lot

Appendix B

B.1 Pre-pilot and Pilot Notes And Data

The following appendix contains notes and summary statistics for the pre-pilot and pilot studies described in this work.

Table B.1 Pre-Pilot Level Notes

Author	Name	Description
P001	Level Of Awesomeness	auto-completes
P002	a house	Well, it's a house. Have to go into one "room" to get the box and another to place it. Not trivial, but no patterns.
P002	the big city	Lots of skyscraper shapes. Carry the box down an "alley" and a street? Patterns exist (simple hallways) to optimize on. Hard to see.
P002	r.a.w.	it's a wrestling ring? simple, one box, one goal. no real patterns. hard to see b/c of walls/towers
P002	the amazing maze	another city-looking level. one box, one goal, lots of blocked areas and irrelevant areas. some patterns.
P003	5panels	it's huge, fills up the entire canvas. There is a tower in the middle with a useless box on it for decoration. Walking around the outside lets me repeat code using a function. Too large to solve without functions.

Table B.1 Pre-Pilot Level Notes Continued

P003	3	again very large. must go to one side of the level and pick up a box, then the other to drop it off, but the paths are asymmetrical so only part can be repeated (and there's not a real cue as to which part)
P004	twisty	a maze-like structure which is unfortunately entirely irrelevant
P004	hallway	a halway with tall walls on either side. the hallway doesn't indicate where I should walk... the middle of the stage just has a repeated block puzzle, so at least I can optimize there.
P004	high	almost completely trivial? looks like I'm walking a plank.
P005	robot cant do it	hides a goal in far corner behind tower, places boxes very far from goals. impossible without loops/functions. still an extra box though.
P006	confusing kingdom	large castle like structure hides the robot and holds an irrelevant box. i just walk around it. need to use loops since the walk is long, but there are no patterns really
P007	New Level	Jump off a building, run to one corner to move a block, then back to the other corner. too large w/o loops. seems like there are a couple places designed to catch the robot (the first jump can get him stuck and force a restart)
P008	a..mazing	Bunch of random irrelevant terrain, since I just walk to the corner and move the box one space... actually i can not use the box. Only one row of this level matters.
P009	city	skyscrapers hiding both the goal and the box. i can ignore the box. hallways (catwalks?) are the same length so I can use functions. Annoying, but ok.

Table B.1 Pre-Pilot Level Notes Continued

P009	maze	sort of interesting structure in the middle, unfortunately the level forces me to jump off a building and afterward I cannot climb back up, so it is impossible.
P010	New Level	box isnt needed, just walk over to goal, totally ignoring structures to the side... short, trivial, can use loops to save 2 instruction
P011	New Level	blank level with one box and one platform. just move in a straight line, so basically trivial, and box isnt really needed.
P011	quadpod	I have to wlak across the full level for no real reason. Once i get there, there is a repeated block puzzle. This part is pretty nice.
P011	the hallway two	created as a response to other level? anyway it's a hallway filled with useless bozes... basically I walk forward ten steps and I am done. everything else in the level can be ignored, and then it's basically identical to the first tutorial level
P011	the most awesome puzzle	A troll level? there's a structure in the middle with a spiral shape, and a box. You can move the box to there, but there is another goal placed on top of a tower to hide it, so the level is impossible.
P011	unbeatable	Not unbeatable. The level has rotational symmetry, making optimization very useful. Still an extra box, though. Too large to do without function/loop
P012	New Level	blank level, trivial solution, box not needed.
P012	shantelle	5 castle towers, no goals. impossible (rather, autocompletes), purely a structure.
P012	showie land	"roller-coaster" like structure around the outside, but the player can ignore it and just walk to the goal, albeit not trivially.

Table B.1 Pre-Pilot Level Notes Continued

P012	supfa	trivial level, walk to goal, ignore box, big tower in the middle
P012	the castle	castle-like crenellations around the outside but again I can just walk straight forward, ignore the box, and solve the level. Can save 1 instruction with a loop, if the box wasn't there.
P013	the level	elevated catwalks with a couple repeated patterns. I can ignore this box. a maze-like level, way too long to do without loop/functions.
P014	Death Drop 2	elevated catwalks, boxes in the center and goals around the corner. I can ignore one box. the catwalks aren't similar so i can only use functions for the long walks. too long w/o loops and functions
P014	Death Drop 2	level is the full height of the screen! elevated catwalks, goals in corners, boxes in middle over zigzag paths. Can ignore one box. No patterns in center of level, and one of the sides has a turn in it, making using functions/loops only minimally helpful.
P014	Javi	seems like I'm supposed to be tricked into jumping down, but the entire platform is 2 units high. I need to cross the little bridge to the other platform and walk to the goal. The box can be ignored. Lots of long walks make it impossible to complete without loops but no real patterns for where to use them
P015	Gummy Bears	raised up platform useless box... I can walk to the goal to solve this. Long walks for loops, no repetition for functions.
P015	Mission Impossible	3 goals, 4 boxes, tons of towers blocking the view, and no real pattern to any of it. Very long, even with functions, certainly impossible without them. City structure?

Table B.1 Pre-Pilot Level Notes Continued

P013	maze	raised catwalks, but they're only 1 unit high so I can just jump down and ignore them. long walks for loops. no real patterns otherwise. box is irrelevant.
P013	my lvl	a 1 unit high maze (meaning bot can just climb over walls) and the goal is on the other end of the level... there is a climbup/climbdown pattern but im not sure that was intended since the rest of the level seems designed to make me walk to the other corner to grab the (irrelevant) box first
P013	pyramid of awesome	wow, very tedious! but there is a pattern... pick up a box, spin around, put it down, over and over. Certainly cannot win without loops/functions. Lots of irrelevant stuff that you can't possibly interact with. Still, it does have patterns and use loops/functions.
P013	skyscraper	walk around this structure to goal, ignore box... small walls seem to be in place to force me to walk around, but they are 1 unit high so i can jump over and get a shorter solution (albeit one that uses loops/functions less effectively)

Table B.2 Pre-Pilot Tag Legend

IRR	(Contains Irrelevant Objects)
IMP	(Impossible or Auto-Completes)
LRG	(Too large for naive solution)
BLC	(Blocks vision of goals or boxes)
BNK	(Blank terrain)
REP	(Contains repeated patterns)
TRV	(Contains trivial solution)
STR	(Contains terrain structure)
Keep	(Keep in next iteration)

Table B.3 Pre-Pilot Level Tags

Author	Name	IRR	IMP	LRG	BLC	BNK	REP	TRV	STR	Keep
P001	Level Of Awesomeness	0	0	0	0	0	0	1	0	0
P002	a house	0	0	0	1	0	0	0	1	0
P002	the big city	0	0	0	1	0	1	0	1	0
P002	r.a.w.	0	0	0	1	0	0	0	1	0
P002	the amazing maze	0	0	0	1	0	1	0	1	0
P003	5panels	1	0	1	0	0	1	0	0	1
P003	3	1	0	1	0	1	0	0	0	1
P004	twisty	1	0	0	1	0	0	1	1	0
P004	hallway	1	0	0	0	0	1	0	1	1
P004	high	1	0	0	0	0	0	1	1	0
P005	robot cant do it	1	0	1	1	0	0	0	1	0
P006	confusing kingdom	1	0	0	1	0	0	0	1	0
P007	New Level	0	0	1	1	0	0	0	1	0
P008	a..mazing	1	0	0	0	0	0	0	1	0
P009	city	1	0	0	1	0	1	0	1	1
P009	maze	0	1	0	1	0	0	0	1	0
P010	New Level	1	0	0	0	0	0	1	1	0
P011	New Level	1	0	0	0	1	0	1	0	0
P011	quadpod	0	0	0	0	1	1	0	0	1
P011	the hallway two	1	0	0	0	0	1	1	1	0
P011	the most awesome puzzle	1	1	0	1	0	1	0	1	0
P011	unbeatable	1	0	1	0	1	1	0	0	1
P012	New Level	1	0	0	0	1	0	1	0	0
P012	shantelle	1	1	0	0	1	0	1	1	0
P012	showie land	1	0	0	0	0	0	0	1	0
P012	supfa	1	0	0	0	0	0	1	1	0
P012	the castle	1	0	0	0	0	0	1	1	0
P013	the level	1	0	1	0	0	1	0	1	1
P014	Death Drop 2	1	0	1	0	0	1	0	1	1
P014	Death Drop 2	1	0	1	1	0	0	0	1	1
P014	Javi	1	0	1	0	0	0	0	1	0
P015	Gummy Bears	1	0	0	0	0	0	1	1	0
P015	Mission Impossible	1	0	1	1	0	0	0	1	0
P013	maze	1	0	0	0	0	0	0	1	0
P013	my lvl	1	0	0	0	0	1	0	1	0
P013	pyramid of awesome	1	0	1	0	0	1	0	1	1
P013	skyscraper	1	0	0	1	0	0	0	1	0

Table B.4 Pilot Level Notes

Name	Type	Notes
walls	Griefer	No visual patterns, many obstructions and extraneous blocks
trippy stuff man	Griefer	Many obstructions. Some opportunity for loops once goal is seen recognized.
ANGER management	Normal	Relatively simple repetition, but it's clear what to do. Lots of improvement. Very long.
centennail	Sandbox	Extraneous structure. Very "sandboxy" with things placed randomly. Can use loops, but no callouts for where to do so. Very long (40+ commands, trivially)
Evil Black Walls of Evil	Power-Gamer?	Lots of obstructions. Relatively long, trivial solution is very long. Once player sees the goal, patterns for loops are obvious. Quite a good level apart from the obstructions.
cursed land	Sandbox / Power-gamer	A "castle" structure is built. Trivial solution quite long. Patterns may exist (walking along wall) but aren't very clear, and getting the box breaks pattern.
a game	Sandbox	Objects placed randomly, no patterns. Solution also quite long.
Evil Trippy Walls of Evil	Power-Gamer?	Essentially identical to previous level "Evil Black Walls," many obstructions but good opportunity for repetition. primary change is distracting color pattern.
the maze	Power-Gamer	Another castle structure. Very long solution. Spme patterns for first box, not for second box. Extra boxes placed. Obstructions.
My New Level	Normal	Patterns, climbing out of hole structure, but relies on student recognizing jump up and jump down are the same. Obstructions.

Table B.4 Pilot Level Notes Continued

hardest level	Griefer / Power-Gamer	Lots of obstructions. Very long. Random placement of goals and boxes. Very frustrating type of level to play. Trivial solution certainly impossible.
My New Level	Trivial	Completely trivial.
MasterChef	Sandbox?	Trivial. Structure which must be climbed over, trivially. No clear visual opportunity for optimization. Solvable and optimizable, though.

Table B.5 Pilot Tag Legend

LCP	(Looks completable)
ACP	(Actually completable)
ADV	(Improved with loops or functions)
OBV	(Obvious Patterns)
NAI	(Naive Solution in 5 minutes)
EXP	(Expert solution in 5 minutes)

Table B.6 Pilot Level Evaluations

Level Name	LCP	ACP	ADV	OBV	NAI	EXP
walls	No	Yes	Yes	Yes	No	Yes
trippy stuff man	No	Yes	Yes	Yes	No	Yes
ANGER management	No	Yes	Yes	Yes	No	Yes
centennail	No	Yes	Yes	Yes	No	Yes
Evil Black Walls of Evil	No	Yes	Yes	Yes	No	Yes
cursed land	No	Yes	Yes	Yes	No	Yes
a game	No	Yes	Yes	Yes	No	Yes
Evil Trippy Walls of Evil	No	Yes	Yes	Yes	No	Yes
the maze	No	Yes	Yes	Yes	No	Yes
My New Level	Yes	Yes	Yes	Yes	Yes	Yes
hardest level	No	Yes	Yes	Yes	No	Yes
My New Level	Yes	Yes	No	No	Yes	No
MasterChef	Yes	Yes	Yes	Yes	Yes	Yes

Table B.7 Pilot Student Responses

x	GEQ																Demographic Survey								
Username	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8	9
P101 X	3	2	1	1	3	2	1	1	1	3	3	3	2	1	3	3	4	4	3	2	1	1	5	3	5
P102 X	3	1	2	1	1	3	3	2	2	2	2	1	3	1	2	2	2	3	3	2	2	3	2	2	5
P103 X	2	2	1	1	1	1	1	2	1	1	1	1	1	1	1	2	5	3	4	2	4	5	3	1	4
P104 X	2	1	1	1	2		1	3	1	1	3	2	1	1	1	3	1	2	1	5	2	2	5	5	5
P105 X	2	1	1	1	2	1	1	2	1	1	1	3	3	1	2	3	2	4					2		3
P106 X	1	1	1	1	2	2	1	1	1	1	3	3	3	2	1	3	2	1	2	1	1	3	1	1	5
P107 X	2	2	1	1	2	2	2	2	2	3	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1
P108 X	3	1	1	1	1	2	1	3	2	2	1	1	3	1	2	3	5	3	3	4	2	2	5	3	5
P109 X	1	2	1	1	2	1	2	1	1	1	1	3	1	1	2	2	3	2	2	3	2	3	2	3	3

Table B.8 Pilot Gameplay Data

Username	Condition	Levels Created	Levels Published	Programs Played	Levels Completed	User Levels Played	User Levels Completed
P101 X	3	3	3	31	6	6	
P102 X	2	1	1	34	12	10	1
P103 X	2	2	0	33	4	16	
P104 X	3	3	3	64	7	40	2
P105 X	2	1	1	46	7	7	1
P106 X	1	2	2	60	7	3	
P107 X	1	0	0	23	1	9	
P108 X	3	3	3	33	5	10	
P109 X	1	1	1	27	2	2	

Appendix C

Table C.1 Organized Data Collections used for this work. All tutorial gameplay data collected prior to 4/11/15 was anonymized and used for the hint-generation work in Chapter 5. All collected puzzle data was anonymized and used for puzzle analysis in Chapter 7, except for data collected prior to 11/16/2012.

Date	Program	Age Range	Users	Puzzles	Conditions	Surveys	Chapters
2/6/16	MSEN @ NCSU		26	0	Debug		Ch8
1/9/16	MSEN @ NCSU		19	0	Campaign		Ch7 only
12/5/15	MSEN @ NCSU		30	0	Campaign		Ch7 only
7/28/2015	Girls Summer Camp		28	32	BB / Prog		Ch6
7/06/2015	NCSU Summer Camp		25	49	BB / Prog		Ch6
6/22/2015	NCSU Summer Camp		26	59	BB / Prog		Ch6
6/15/2015	NCSU Summer Camp		29	76	BB / Prog		Ch6
4/25/2015	SPARCS @ NCSU (DNS)		13	21	BB / Prog		Ch6
4/11/15	SPARCS @ NCSU (CCMS)		23	58	Prog (Pilot)		Ch5, Ch6
12/14/2013	MSEN @ NCSU	6th	15	21	Free/ S&S	GEQ	Ch4
			17	21	Free/ S&S		
11/16/2013	SPARCS @ CCMS	6th-8th	15	18	Free/ S&S / Admin	GEQ	Ch4
			29	25	Free/ S&S		
12/18/2012	GameCATS @MMS		0	0	Free/ S&S	(none)	
			12	11	Free / S&S		
12/13/2012	BJC	University	28	18	Free / S&S		
11/16/2012	SPARCS @ CCMS	6th-8th	9	13	Free / S&S / Admin	GEQ	Ch3, Ch4
			16	24	Free / S&S / Admin		
7/11/2012	UNC Charlotte Summer Camp		19	35	Free		Ch3