A Software Studies Approach to Interpreting Passage

Dylan Lederle-Ensign
Center for Games and Playable Media
UC Santa Cruz
dlederle@soe.ucsc.edu

William Robinson
Concordia University
Technoculture, Art and Games (Research Centre
william.robinson@concordia.edu

Johnathan Pagnutti
Center for Games and Playable Media
UC Santa Cruz
jpagnutt@soe.ucsc.edu

Michael Mateas
Center for Games and Playable Media
UC Santa Cruz
michaelm@soe.ucsc.edu

ABSTRACT

The following paper argues for the value of critically analyzing game code, demonstrating two potential methodologies. These draw from literary theory and software studies to reach hermeneutic readings of procedures. We use Jason Rohrer's *Passage* as a case study to demonstrate these methods. *Passage* is one of the most frequently studied, discussed and taught works in game studies. However, many critics take for granted the claims that Rohrer makes in his creator's statement [16] regarding the game's message. This paper seeks to complicate several of these claims, particularly regarding the balance between marriage or exploration, and their rewards.

We explain in detail the steps we undertook to investigate the game's code. First, a technique of closely playing the game, following a theorycrafting approach [15], was deployed to make sense of four particularly opaque lines of code. By combining close play with close readings of the code, not only did we solve a previously unconsidered puzzle, but added additional richness to existing interpretations of Rohrer's work. Second, we produced modified versions of the game, following what McGann and Samuels call the critical practice of deformance [12]. We replayed these distorted version's of the game in order to enlarge our considerations for its design. In doing so, we came to understand the rhetorical emphases placed by Rohrer and the shifts that small changes could have. Both of these techniques were used in our interpretation of *Passage*, and we present them as methods which other scholars can build on and use with other open-source games.

Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015), June 22-25, 2015, Pacific Grove, CA, USA.
ISBN 978-0-9913982-4-9.

Categories and Subject Descriptors

K.8.0 [**Personal Computing**]: General—Games

General Terms

Game studies, Software studies

Keywords

Passage, Jason Rohrer, Critical Code Studies

1. INTRODUCTION

Despite Game Studies' interest in digital games, the field has paid comparatively little attention to the algorithms underlying the medium. The following paper argues for the value of game code analysis and works towards potential methodologies. Platform studies has by necessity done some of this work in its appeals to analyzing game hardware and software along with the affordances and constraints they place on media (see Montfort and Bogost 2009 [13]; Salter and Murray 2014 [17]). Rather than a focus on platforms, however, this paper explores the meaning of code in the context of a game whose processes and mechanics are intended to be read metaphorically (Bogost 2007 [2]; Treanor and Mateas 2010 [22]). Where analyses focusing on operational logics focus on the abstract processes underlying representation, (Mateas and Wardrip-Fruin 2009, [11], Fox Harrell 2013 [7]), in this paper we focus on a detailed code-level analysis. Code-level readings of games do exist (Sample 2013 [18], Marino 2006 [10]), but these have tended to focus on comments and variable names, rather than analyzing the relationship between the code-level expression of process and player interpretation. Drawing from works in software studies (Fuller 2008 [5]) and game studies, we argue that appeals to code may offer insights into the development process, and can clarify ambiguities about a game's procedures. Specifically, we engage in a code-level analysis of Jason Rohrer's art-game *Passage* (2007) to understand the relationship between code-level development decisions and player interpretation. Despite having been theorized by multiple game scholars (for instance: Parker 2012 [14]; Bogost 2008 [3]; Whalen 2012 [24]), the fact that *Passage*'s code is open-source has never been called into question, never

mind examined. While its code is not representational in the same ways as its output, it does hold meaning that is detectable upon a procedurally literate inspection. What is more, as an artwork with its computational underpinnings left intentionally accessible for public consumption, it is not unreasonable to consider it in this way.

Given the scope of this paper and the fact that *Passage* has thousands of lines of code, only two portions of Rohrer's work will be discussed here. The first relates to the hidden contents of in-game chests. We use this portion to illustrate our use of Close Code Playing and the ways in which it relates to Rohrer's use of scoring in *Passage* . In his creator statement, Rohrer discusses the game's metaphoric chests, suggesting that "not every pursuit leads to a reward—most of them are empty." Rohrer continues the metaphor, explaining that these chests are marked with a sequence of gems and that "During your lifetime, you can learn to read these sequences and only spend your precious time opening worthwhile treasure chests." Current writing on *Passage* has not addressed this puzzle, which we solved using a combination of close inspection of the code and iterative playing for data mining similar to theorycrafters (see Paul 2011 [15]).

The second section of code is responsible for the game's procedurally generated labyrinths. We use this portion, combined with knowledge of the codebase gained from our earlier code reading, to create small changes that noticeably alter the game's dynamics. We draw this critical practice from literary criticism, much like the previous use of close reading, but instead focus on what Jerome McGann and Lisa Samuels dub "Reading Backwards" or deformance. We examined the map generation by altering the code to produce new playable versions of the game . We call this process of code manipulation and recompilation for critical purposes "procedural deformation".

While both sections of code were selected because they are illustrative of Rohrer's style of coding, each also illustrates a proposed methodology for game code studies: Close Code Playing and Procedural Deformation.

This paper is divided into three sections. First, a brief review of software studies literature as it relates to game studies. Second, a close reading and explanation of salient points in *Passage*'s code will be provided to give concrete examples of the object in question. Third, the code will be modified to create alternative instances of *Passage*, shedding light on the space of possible interpretations of the game. As it stands, game studies has largely ignored close analysis of code, which is not surprising. After all, players rarely if ever have access to it, and even if they did, it would be largely meaningless. Beginning with the assumption that investigating code may be fruitful, several avenues open up, none of which this paper can map in any complete way. Instead, we hope to demonstrate the game studies possibilities that open up when engaging in a code-level analysis and offer methodological directions for future work in this area.

1.1 Why Passage

In his creator's statement, Jason Rohrer explains that one's life in *Passage* involves exploration and accumulation [16]. For scoring purposes, these goals are competing, because attempting to excel at one will often be to the detriment of the other. Several critics have lauded Rohrer for his elegant ludic metaphor, particularly its balance and "openness to interpretation". Felan Parker has explored this particular reception in his essay "An Art World for Art Games," [14] in which *Passage* is offered as a case study for the institutionalization of games as art. Parker's work painstakingly explores the various actors such as the Kokoromi game collective; the Gamma 256 event; the retro aesthetics of *Passage*; the auteurist stance taken by Rohrer in programming everything himself; Rohrer's artist statement; and the sets of relevant critics. Despite this, Parker elides one of *Passage*'s most salient properties. It is an open source game. In fact, it was the only open source game from the Gamma 256 gallery showing where it was launched. Not only does its ontological status, as a freely available and remixable object, change its social positioning and value, but it allows critics and fans to inspect its code. It is an associated set of signifiers which not only complicate receptions of *Passage*, but offer interesting readings in their own right. While *Passage* is one of any number of games worth exploring, we selected it for particular pragmatic reasons. Firstly, the game is open source. This means that we have access for both reading and re-compiling its code, without purchasing the rights to what would otherwise be proprietary or simply unavailable. Secondly, *Passage* is for better or worse a piece of the game studies canon. Scholars such as Ian Bogost (2011) [4], Fox Harrell (2013) [7], Felan parker (2012) [14], Mike Treanor (2011) [21], John Sharp (2012) [19], and Alison Gazzard (2013) [6] have all written about the game. Finally, *Passage*is relatively small, as far as game software goes. It is possible to read all relevant lines of code, re-write some of them several times and play the 5-minute game to completion in each new instance. Being able to map out the game's possibility space through algorithmic modification has been fruitful and would otherwise be impossible with longer titles requires hours if not days to play through.

2. SOFTWARE STUDIES IN GAME STUDIES

Software Studies is an emerging field in the humanities, which focuses on software, its relation to culture, and how it is situated in the world [5]. The Software Studies book series from MIT Press declares that "Software Studies uses and develops cultural, theoretical, and practice-oriented approaches to make critical, historical, and experimental accounts of (and invterventions via) the objects and processes of software" [23]. Given that video games are software objects, this perspective is likely to reveal new opportunities for critical interpretation. There have been software studies of video games (Harrell 2013 [7], Sample 2013 [18], Kazemi 2014 [8]) but given the importance that software's materiality has for the play experience of video games, the intersection between software and game studies has been underexplored and underdeveloped. Fully examining all of the ways that a software studies approach can be applied to games is a large project, beyond the scope of this paper. Instead, we will examine one strand of previous work on which we build, critical code studies (Marino 2006 [10]), and unpack its promises and limitations.

Video games are made with code, but can reading this code inform our critique? A critical code studies perspective takes the code as its subject, as an aesthetic object in its own right. While this can be a productive approach, code does not on its own stand in for a game. A code studies perspective is only relevant to game studies when it is com-

bined with a detailed understanding of the play which that particular code enables. In the same way that a game is not solely its rules, the rules are not solely their code. The code is, in some ways, far removed from any play experience we might care about when the software is operated by a player. However, particularly in proceduralist games, in which the rules or procedures are a primary form of expression (Bogost 2011 [4]), examining the code that defines these rules can inform our critique.

There are many questions in game studies which cannot be answered by code studies alone. Code level analysis is useful for answering highly specific questions, particularly about the construction of rules. This is understandably a limited approach, from both directions. A well constructed rule set should be understandable by an informed critic simply by playing, limiting the effectiveness of code reading. Similarly, extracting the gameworld, its rules and dynamics, from code is near impossible However, when there is ambiguity about the exact construction of a particular rule, code level investigation can help resolve it. If the code of a game is available, which is its own challenge, we encourage more scholars to open their text editors and look at it.

3. INVESTIGATION OF PASSAGE

This section details extensively the methods we used to study *Passage*. While the resulting readings of *Passage* contribute to the critical discourse around the game, our focus on method is intended to help researchers understand ways they can incorporate software studies into their game studies work.

3.1 Chest Puzzle

Close reading is a core methodology in literary criticism, developed to analyze short passages of text. Particular emphasis is placed on word choice, localized meaning, syntax or other structures relating to the immediate and the minute detail. This method has been explicitly deployed in games studies in order to unpack similarly small portions in game play [1]. This was our first approach to *Passage*, particularly because it appeared at first glance applicable to an opaque element in the game. Rohrer, in his creator's statement [16], describes the chests that litter the playing field and how the player comes to learn about their contents:

> Over time, though, you can can learn which pursuits are likely to be rewarding. Each treasure chest is marked with a sequence of gems on its front, and this sequence indicates whether the chest contains a reward. During your lifetime, you can learn to read these sequences and only spend your precious time opening worthwhile treasure chests.

Critics of *Passage* take Rohrer's statements for granted, quoting them liberally without unpacking how his metaphors are constructed. While accepting Rohrer's own explanations may not be problematic for the majority of the metaphorical elements used in *Passage*, these chest puzzles are at no point explained in Rohrer's writing or discussed in the critical literature. Indeed, in our own playing, despite our best intentions, we were never able to crack the chest codes and save any reasonable amount of time. They appeared to be random. We set out to find their solution. Our searching led us to the following lines in game.cpp, the core game file:

```
514    // the gem that marks chests containing points
515    int specialGem = time( NULL ) % 4;
```

Setting aside for the moment an explanation of what this code means, let us continue following the places the variable *specialGem* is used. Further down the same file, we find the code that executes when the player touches a closed chest. This is the only other place that *specialGem* appears in the code:

```
1183           if( getChestCode( (int)playerX,
    (int)playerY ) &
1184               0x01 << specialGem ) {
```

Do you see the solution to the chest puzzle now? Of course not. We will return to this point below, but these two snippets of code begin to give a sense of the futility of isolated code reading as a method for understanding the game elements they implement. These two lines leave us with many more questions than answers. The answers to these questions are all in other parts of the code, and we can search deeper to answer them. At this level of detail we still have no clear solution to the chest puzzle.

The simplest elements in these two lines are the playerX and playerY parameters to the function getChestCode. These indicate the position of the avatar within the grid of the world, with (0,0) being the top left of the world. Next, we need to understand what this function does. A search within this file reveals nothing, but within the folder we can find another file, map.cpp, which contains the function definition:

```
191 unsigned char getChestCode( int inX, int inY )
```

This is the beginning of the function definition. There are several clues here as to how it works, as well as several glaring questions. First, following C language syntax, line 191 declares a function. It takes two integer parameters, and whatever data it returns must have the type unsigned char, or unsigned character. For our purposes, this means it will return a single byte, which in this case is not actually being used as a character. Recall that the output from this function is being used to determine if there is treasure in the chest at a particular (x,y) coordinate. Without any attempt to understand the body of this function or how it calculates this byte, we can return to game.cpp and try to decipher some of the chest code.

We now understand line 1183 to mean the following: a byte, presumably representing the gems at the front of a chest, is calculated using the location of that chest. This byte is then bitwise AND-ed (not logical AND) with another byte (0x01) that has been left-shifted by *specialGem*. This code is making use of a binary bit vector representation of the chest puzzle, rather than higher-level representations that might more typically be used, such as an array.

By making use of such low-level programming, Rohrer makes his work leaner at the cost of increasing its opacity. While this might seem reasonable, the speed gained is meaningless in the face of technologies available to Rohrer even ten years prior. Programming this way demonstrates a preference for "coding to the metal". Rohrer, in addition to gaining the functional results he is after, is making a statement akin to "waste not, want not". As mentioned earlier,

given that this code is open source, positioned as additional available signifier for readings of *Passage*, it is not unreasonable to associate this reading of Rohrer's code with his game's larger message of living efficiently to maximize gains. Setting this additional hermeneutic work aside, we can continue with the code to demonstrate how this interpretation plays out.

In map.h, a header file that is used to declare functions or constants used in map.cpp, we find the following constant declarations:

```
// 8-bit binary indicating which of six external chest
    gems are present
#define CHEST_RED_GEM    0x01
#define CHEST_GREEN_GEM  0x02
#define CHEST_ORANGE_GEM 0x04
#define CHEST_BLUE_GEM   0x08
#define CHEST_YELLOW_GEM 0x10
#define CHEST_MAGENTA_GEM 0x20
```

These constants are not used anywhere in the codebase. They were likely used for debugging. CHEST_RED_GEM is the byte 0000 0001, CHEST_GREEN_GEM is 0000 0010, and so on. Each one moves the 1 further down the byte. While initially opaque, it is a reasonable and elegant solution upon closer inspection. To Rohrer, or others with experience writing byte-manipulating C code, this is perhaps just as natural an implementation as an array would be to someone reared on higher level scripting languages.

Returning back to map.cpp and getChestCode, we find a fifty line comment among the code, explaining in extensive detail a latent bug, present all along, that was only discovered when *Passage* was ported to iPhone. The function is supposed to return a 6-bit string representing the gem puzzle (which gems are present). Rohrer explains in the comment that, while all six gems appeared, "the 6-bit gem strings were always of the form 00XXXX or 11XXXX where XXXX is a random 4-bit sting (sic)...So, 2 of the gems were tied together in their on/off status, essentially turning 6 gems into 5."

This is a very subtle bug, and only became evident when switching to a different platform. The exact reason for the bug has to do with the way platforms handle casting from unsigned to signed integers, and is not particularly important for our purposes here. After porting to the iPhone, it resulted in a major bug in which "half the treasure chests, on average, had no gems at all." The distribution of chest rewards is a key element in *Passage*'s procedural meaning, as we will discuss further in the procedural deformation section. Investigating this major bug seems to have led Rohrer to understand the original bug with the gem strings. Rohrer's response to this discovery is fascinating:

> Of course, a proper fix, as described above, would go beyond just making it work on these platforms, but would also change the behavior of the game (those last two gems would no longer be tied together in their on-off status).
>
> After much deliberation, I came to the following solution:
>
> The following code emulates 2's compliment casting, whether the platform does it or not, making behavior on all platforms identical.
>
> (And preserving the original bug!!)

In all distributed versions of *Passage*, this original bug is preserved, tying the yellow and magenta gems together. This example demonstrates several things which are critically relevant. First, its an example of the unexpected complexity that underlies computational media. Glitches are everywhere and they can be very difficult to notice, especially in single developer teams without extensive quality assurance departments. These glitches may affect gameplay, which can change player interpretations (Lederle-Ensign and Wardrip-Fruin 2014 [9]). Second, Rohrer recognizes this and demonstrates a concern with preserving gameplay elements between *Passage* versions. While there is no evidence than any player would notice this, as there is no record of players discussing the chests, Rohrer considers them important. As a "proceduralist game" (see Bogost 2011 [4]; Treanor and Mateas 2011 [21]) *Passage*'s processes must be consistent across platforms to avoid a schism in interpretation. This is a fact that Rohrer is not only aware of, but considers important enough to code a workaround that preserves the "bug" across platforms.

Intriguingly, our solution to the chest puzzle was not found in the code. Rather, our code level investigation pointed us in the direction of an instrumental way of playing that led to solving the puzzle. Even with a relatively clear understanding of the way the puzzle was implemented, we continued to miss the key. Our first attempt at solving the chest puzzle through play led us to record each gem string and whether or not the associated chest contained a star (and thus an ingame of value 100 points). A half-dozen games were played with the intent of getting the most possible chests to produce enough data for pattern recognition. The resulting spreadsheet indicated that chest codes did not reliably indicate the contents of a given chest because the same chests would appear with and without stars inside. Upon inspecting the game code, it became clear that there were in fact five distinct sets of chest codes, one which would be randomly chosen for a given playthrough. Recall:

```
515    int specialGem = time( NULL ) % 4;
```

The "% 4" creates five possible permutations, something we had initially missed. The time function returns the current time (in milliseconds) and is executed when the game starts, making any prediction of which gem will be "special" impossible. By playing several games and attempting to create enough data to perform pattern recognition, we inadvertently produced contradictory data. By this point, we began to doubt the efficacy of Rohrer's design, as we asked ourselves how anyone could possibly find enough patterning in the modicum of chests available in a given play through to act any differently. We restarted the empirical data collection, this time clearly denoting which playthrough a given code was found in. After eight playthroughs the spreadsheet remained opaque. The sheet was then color coded for clarity, at which point the answer to our question became intelligible. Not only was this practice aligned with close reading, but also theorycrafting. Chris Paul has written on the topic in his essay "Optimizing Play," [15] whereby players in a given game community explore the hypothetical code of games through empirical experimentation. Not only are these players interested in how a game works, but do so in order to play the game more effectively.

In a given playthrough, only a single gem matters. For ex-

ample, in one game, all chests with a green gem will contain stars, and any without the gem will not. This is what the code refers to as a "special" gem; it denotes the chests that contain rewards. After understanding the solution and reviewing the code which implemented it, we were astonished we had missed it the first time through. Once again, this points to the difficulty of understanding code in isolation from the associated game processes.

Having understood the chest codes, we began playing once more, attempting to reach higher scores. The results were remarkable, with large increases in final scoring. The semantic content of the game changed at this point. It was no longer possible to reach comparable scores with a spouse than without. In *Passage* there is a suggested goal made available by the presence of the high score. We have already discussed how Rohrer emphasizes efficiency in this respect. Given that the game is about living life, the suggestion read here is that not all lives are equally successful, particularly because Rohrer has taken the time to abstractly represent things numerically, where this number refers us to the high scores of game culture (i.e. that which is desirable). When one plays *Passage* and attempts to reach a high score, it appears at first that both paths, choosing a wife or going after chests alone, lead to similar outcomes (about 700-1000 points). Harrell has argued that this form of balancing leads to a very similar claim to Robert Frost's metaphoric The Road Not Taken, where the paths "both equally lay" [7]. Rohrer's simulation of life becomes a game by directing us to achieve a high score, leading us to realize that there are equally good lives to lead and that neither is preferable.

While this reading of *Passage* is somewhat unromantic, as we have Rohrer contemplating life having not married his wife, and realizing it would be about equally valuable, it is not complete. Instead he hides this fact in plain sight, appearing content to have selected company for his life's journey. He suppresses his anxious feelings of thwarted potential, allowing only those who, like him perhaps, obsessed with code, design and puzzles to see how limited he feels. His memento mori is cryptically tragic.

3.2 Map Generation

> *Passage* represents life's challenges with a maze...The world in *Passage* is infinite.
>
> -Jason Rohrer's creator's statement [16]

Our next question was about *Passage*'s map. Following Rohrer's claims of infinity, it appears to be procedurally generated. What procedure produced this map? What is the maze like? How are the chests placed? These questions lead to the deeper question of how we can understand and critique a generative process when only experiencing a small slice of its output. In this section we will build towards that question by exploring *Passage*'s map generation code.

The first thing we noticed, before even looking at the code, is the image file "tileSet.png" (Figure 1) within the gamma256/gameSource directory. This is where the textures for the world are stored, and a quick play through the game confirms that the top row of tiles are the floors for each section of the passage world, and each column corresponds to a column of game world. These textures are not procedurally generated. Instead, they were pre-authored, presumably by Rohrer, and are applied to level geometry after it is generated by some process.

Figure 1: tileSet.png

The next step in determining how the level geometry is generated leads us to map.cpp. Its header file shows two functions that may be the answer to our questions: isBlocked and isChest. Both take an (X,Y) pair and return a character. To figure out why they return characters and what they mean, we must look at the context in which these functions are used. A clearer understanding of the purpose of the function can help answer questions about why it was implemented in a particular way.

In this case, isBlocked is primarily used in another file, World.cpp. This file contains extensive graphics operations which build the screen in memory before sending it to a library that displays it on hardware. As you might expect from the name, this function is being used to determine if a particular tile should render as an empty space, or as a wall. Returning to its implementation, we can begin to understand the way *Passage* generates maps. The private function isBlockedGrid, where the work is done, begins with a couple of basic conditional checks.

```
93 char isBlockedGrid( int inGridX, int inGridY, char
    inDoNotRecurse ) {
94
95    // wall along far left and top
96    if( inGridX <=0 || inGridY <= 0 ) {
97       return true;
98       }
99
100   // empty passage at top
101   if( inGridY <= 1 ) {
102      return false;
103      }
```

A return value of true means there is a wall, and line 96 draws one along the top and side border. Line 101 ensures that the first row of the *Passage* grid is left blank, which the comment implies is another meaning of the title. These two lines ensure the world conforms to Rohrer's level design, regardless of any randomness. The core of the map algorithm can be found a little further down.

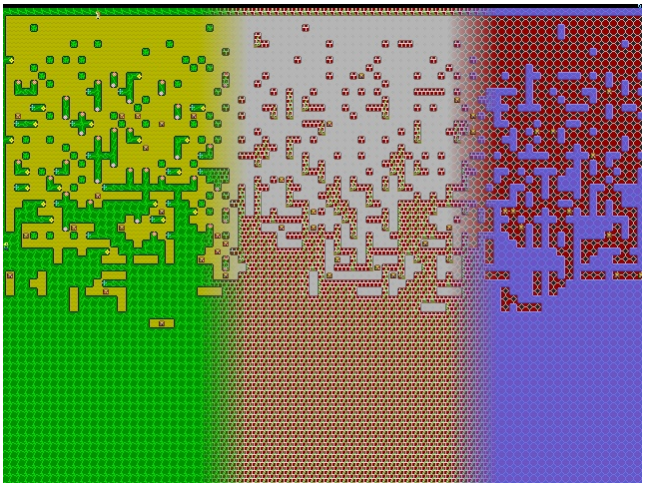

Figure 2: Zoomed out picture of one map

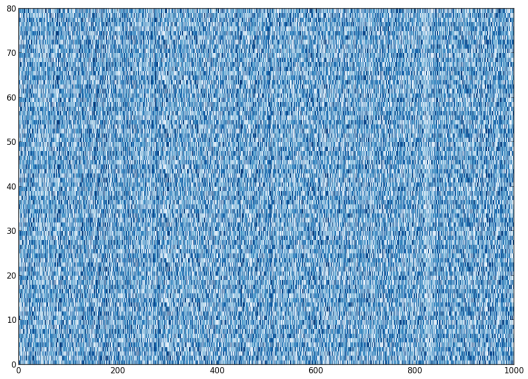

Figure 3: Sample output of *Passage*'s noise function, using X, Y, and a random seed, shaded according to value (0.0 - 1.0)

```
122   // blocks get denser as y increases
123   double threshold = 1 - inGridY / 20.0;
124
125   double randValue = noise3d( inGridX, inGridY,
    seed );
126   char returnValue = randValue > threshold;
```

A threshold is defined, scaling in direct relation to the depth in the world. Then a random value is calculated, using noise3d. Finally, if this random value is larger than the threshold, then this grid square has a block in it. This is almost the entirety of the maze generation, apart from another function that fills in squares which are surrounded on all sides. This method of maze generation does no precalculation. It is simply a way of filtering and constraining random numbers. The final piece of the map generation is the chest placement. It is very similar to wall placement, as we see in the function isChest:

```
165    // chests get denser as y increases
166    // no chests where gridY < 5
167    // even less dense than blocks
168    double threshold = 1 - ( gridY - 5 ) / 200.0;
169
170   // use different seed than for blocks
171   double randValue = noise3d( 73642 * gridX, 283277
    * gridY, seed * 987423 );
172   char returnValue = randValue > threshold;
```

The first three lines of comments explain the differences in threshold calculation, but the core method is the same. Now we understand the way *Passage* maps are generated. Everything becomes increasingly dense as you descend deeper into the world. As Rohrer explains in his statement, "As you go deeper into the maze to the south, the path becomes more convoluted, though an obstacle-free route is always available to the north. However, treasure chests are more and more common as you go deeper into the maze." Figure 2, from a modified version of *Passage* that displays a larger view of the world, is another example of what one fully rendered world looks like.

With this understanding of the basic map algorithm, the only remaining question is how Rohrer generates the random numbers, or noise, that form the basis of the map. Noise functions are a way of calculating pseudo random numbers. See Figure 3. The main source of this noise is the function noise3d, defined within landscape.cpp. At the top of this file, we find the following block comment:

```
1 /*
2  * Modification History
3  *
4  * 2006-September-26 Jason Rohrer
5  * Switched to cosine interpolation.
6  * Added optimizations discovered with profiler.
     Reduced running time by 18%.
7  */
```

As indicated, a profiler is a tool that helps identify performance bottlenecks in running code, and is used as a way of identifying places to optimize. This does not at all fit with the style of the rest of the *Passage* codebase, which has no indications that it has been highly optimized. Tellingly, the date listed here is nearly a year before the announcement of the Kokoromi event Gamma256, where *Passage* debuted in November of 2007. While it's possible that *Passage* was in development before the announcement (Tigsource Forums 2007), it seems unlikely for several reasons. First, this file is called landscape.cpp, and has a main function of the same name, however this function is never called anywhere in the codebase. There are actually five functions defined in the file, but only one is used. While unused code is not unusual, and could represent an alternate approach to map generation, an identical file appears in his 2007 game Cultivation. Indeed, this is an example of code reuse between projects. While we feel an intertextual analysis of Rohrer's code would be worthwhile in exploring the circulation of code, it is beyond the scope of this work.

There are three places where noise3d is used in *Passage*, all within the same map.cpp file we have been looking at. The first use is within isBlockedGrid, as noted above, which determines if an (x, y) location has a wall on it. The second use is within isChest, while the third is within getChestCode. All three uses pass in the (x,y) pair and a seed based on current system time. While isBlockedGrid uses these val-

ues directly, the other two scale them based on apparently arbitrarily chosen numbers. Noise functions are not truly random, and typically have patterns. Figure 3 below shows the output of noise3d across a space larger than the *Passage* map, when plotted without any scalars. There are clear patterns. When plotted with the scalar numbers used in the chest generation functions there are still patterns, but they differ. In a sense, tuning these numbers and adjusting the pattern of random noise is a level design task. However, we currently have no information as to how these numbers were chosen.

Rohrer claims in his artist statement that, "*Passage* represents life's challenges with a maze". However, we have shown that there is no sophisticated maze generation in *Passage*. The map only appears like a maze due to your limited viewpoint. When the player viewpoint is widened (See Figure 4), the lack of coherence in the map reveals it to be a product of noise. With a familiar, optimized noise library at hand, it was easier for Rohrer to generate the topography using noise than to design a new PCG system to create the maze or to hand author many mazes. Maze creation is done as the player encounters it, with the underlying process being semi-random walls that get denser as you go down. Procedurally generating the maze also supports Rohrer's rhetorical purpose of representing infinite possibilities, or as Rohrer claims in his statement, "even if you spent your entire lifetime exploring, you'd never have a chance to see everything that there is to see." This concept of unlimited possibility is common in many games that use PCG.

Gillian Smith has developed a critical framework for discussing game design aspects of PCG in games (Smith 2014 [20]), Using this language *Passage*'s map generation is a constraint based system, with fuzzy constraints (as noise follows trends, not hard constraints). The map generator provides no way for the player to interact with or control the generator and indirectly changes player experience. The map generator works on the subcomponent level, coloring tiles with a sprite map. The map is generated online, as more of the "maze" is generated as the player explores it.

The qualities of the maze in *Passage* appear to lend themselves to the Searching a Vast World dynamic that Smith identifies, however, that is only true if a player does not care about score. Rohrer claims that score does not matter to *Passage* in his artist statement, however, the sheer fact that score exists rewards particular kind of play. The aesthetics of *Passage* change depending on how the players feel about score, with the non-scorers focusing on discovery, whereas the scorers focus on challenge.

This dichotomy is also reflected if players replay *Passage*. There is some validity to claiming that *Passage* was meant to be played only once, particularly given its original context within an art gallery. There are no affordances for replaying *Passage*– when the game ends, there is no replay button. To play *Passage* again, you need to exit and restart the game. When you just play once, score clearly doesn't matter– score isn't saved in *Passage*, and there is no leaderboard to compare your last run against. In this case, again, the map generation is all about showing a vast world that you can never explore all of. But, it is possible to play *Passage* more than once– the game doesn't destroy itself after all– and score is the one thing you can use to mark your progress and "skill" at playing *Passage* against previous runs. In this case, the fact that *Passage* never shows you the same map twice and the chest puzzle solution changes with each new playthrough, leans towards *Passage* being a game about reaction. The ability to quickly solve the chest puzzle and track successful paths through the maze lead to higher and higher scores.

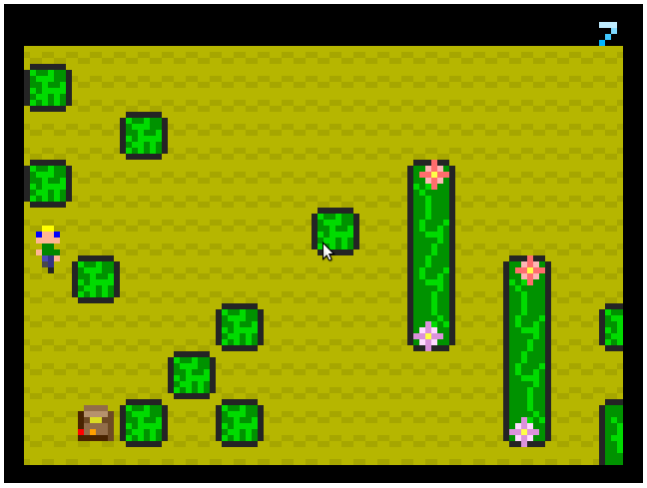

Figure 4: Clear screen version of *Passage* with distortion effect removed, and taller field of view

3.3 Procedural Deformation

We made several alternate versions of *Passage*, with changes ranging from small changes to the weights of random numbers, to larger changes that expand the field of view and expose the entire map clearly. While Rohrer characterizes the map as a maze, it only functions as such when the player has visual distortion and a restricted field of view. Revealing the full field of view removes the maze-ness of the game, and exposes more of the underlying wall placement algorithm. Similarly, modifying the weights with which walls or chests are placed in the world can significantly alter the game's rhetoric. In an alternate version with denser chests, exploring to the right becomes significantly less appealing, as chests are much easier to discover. These deformations allow us to interrogate the values and parameters that Rohrer chose, and lead to a deeper understanding of their rhetorical purpose. This section describes several alternate versions that we made and briefly discusses other potential uses for procedural deformation.

The first change we made is the full view version (Figure 4). It reveals that the "maze" is not actually a maze. The walls do not consistently form connected paths and simply become denser as we descend. More subtly, with the increased field of view it is much easier to make "correct" decisions about chest pursuits, particularly when combined with the solution to the chest puzzle. This allows an informed player to reach much higher scores than previously possible, and significantly changes the balance of the game.

A simpler change was to increase the speed of player movement. The game still takes five minutes, but it is possible to explore much more of the world in a given playthrough. It is also much less costly to explore downwards, as recovering from paths without chests is fast.

A rather extreme change we made was to decrease the threshold with which walls are generated. At a low enough

point, this leads to a single clear path along the top of the world. It eliminates all player choice, where your only options are to walk forward and die, or stay in one place and die. Similarly, it is possible to modify the other thresholds to create a map with no walls, a map filled with chests, or a map without chests. These versions each have fairly predictable rhetorical effects, and essentially break any coherent metaphor. Overall, these tweaks change the weighting on the tension between getting married or not. These tweaks also reveal extra rhetorical devices behind exploration– if you remove the ability to explore, the game is suddenly about being stuck on a single path and doing that same thing until you die.

These simple modifications also give some insight into the tuning process behind Rohrer's map algorithm. With a few simple parameters, vastly different play experiences can be created. Using these as knobs, the designer can rapidly iterate until a suitable balance is found.

Procedural deformation offers a small counter to the "black box" of computational media (Sample 2013 [18]). Rather than written critiques of a simulation, we can produce alternative versions and then play them. Game objects and players together are what matter, not just rule sets. By implementing an actually playable model rather than an unplayable thought experiment, we can engage in deeper critique of the original games' models and assumptions. While games do not typically expose these features to the player, imagining alternatives to the designer's status quo can be an important critical practice informing interpretation.

4. CONCLUSION

This paper contributes to the critical discourse surrounding the art game *Passage*. Our extensive investigation of its codebase added new subtleties to our understanding of the game, and complicated some of the creator's stated claims. Our combination of closely playing the game while closely reading the source proved highly useful, and can be replicated in other games. Our practice of procedural deformation, or modifying the game to induce rhetorical changes, is likewise a useful tool for others seeking to undertake software studies of games. Our hope is that the extensive detail we have related is useful for others who wish to follow along, and continue studying the software of games.

There are numerous further directions this study could go. One could pursue further modifications of the game, perhaps swapping around graphics as Zach Whalen has done to unpack the cultural privilege that students bring to the game [24]. Another could go beyond critiquing the map generation and produce an alternative approach to map generation, implementing, for example, true maze generation. Yet another could undertake porting the game to another language, as an exercise in exploring Rohrer's distinctive style of coding. Beyond *Passage*, all of Rohrer's games are open source. A comparative study could be done, tracing shared code and uncovering further details about his development practice.

5. REFERENCES

[1] J. Bizzocchi and J. Tanenbaum. Well Read. *Well Played*, 3, 2011.

[2] I. Bogost. *Persuasive Games: The Expressive Power of Videogames*. MIT Press, 2007.

[3] I. Bogost. Fine processing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5033 LNCS, pages 13–22, 2008.

[4] I. Bogost. How to Do Things with Videogames. University of Minnesota Press, 2011.

[5] M. Fuller, editor. *Software Studies: A Lexicon*. 2008.

[6] A. Gazzard. *Mazes in Videogames: meaning, metaphor and design*. McFarland, 2013.

[7] D. F. Harrell. *Phantasmal Media: An Approach to Imagination, Computation, and Expression*. MIT Press, 2013.

[8] D. Kazemi. *Jagged Alliance 2*. Boss Fight Books, 2014.

[9] D. Lederle-Ensign and N. Wardrip-Fruin. What is Strafe Jumping ? idTech3 and the Game Engine as Software Platform. In *Proceedings of DiGRA*, 2014.

[10] M. Marino. Critical Code Studies. `http://www.electronicbookreview.com/thread/electropoetics/codology`.

[11] M. Mateas and N. Wardrip-Fruin. Defining Operational Logics. In *Proceedings of DiGRA*, 2009.

[12] J. McGann and L. Samuels. Deformance and interpretations. *New Literary History*, 30(1):25–56, 1999.

[13] N. Montfort. Portal & Passage. *Well Played*, 1, 2009.

[14] F. Parker. An Art World for Artgames. *Loading...*, 7(11):41–60, 2012.

[15] C. A. Paul. Optimizing play: How theorycraft changes gameplay and design. *Game Studies*, 11(2), 2011.

[16] J. Rohrer. What I was trying to do with Passage. `http://hcsoftware.sourceforge.net/passage/statement.html`.

[17] A. Salter and J. Murray. *Flash: Building the Interactive Web*. 2014.

[18] M. L. Sample. Criminal Code: Procedural Logic and Rhetorical Excess in Videogames. *Digital Humanities Quarterly*, 007(1), 2013.

[19] J. Sharp. A curiously short history of game art. In *Proceedings of the International Conference on the Foundations of Digital Games - FDG '12*, page 26, 2012.

[20] G. Smith. Understanding procedural content generation. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*, pages 917–926, New York, New York, USA, Apr. 2014. ACM Press.

[21] M. Treanor and M. Mateas. An Account of Proceduralist Meaning. In *Proceedings of DiGRA*, 2011.

[22] M. Treanor, M. Mateas, and N. Wardrip-fruin. Kaboom! is a Many-Splendored Thing : An interpretation and design methodology for message-driven games using graphical logics. In *Foundations of Digital Games*, 2010.

[23] N. Wardrip-Fruin. *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*. MIT Press, 2009.

[24] Z. Whalen. Using "Passage" to Think about Cultural Privilege. `http://zachwhalen.net/posts/using-passage-to-think-about-cultural-privilege`.