Reading Code

Dylan Lederle-Ensign

12/15/11

Introduction

Underlying every video game, work of electronic literature or mundane computer application there is source code that forms the processes we interact with. As players, readers or end users we deal with the output of a multitude of code, often very complex and interesting in its own right. While there is code underlying all digital artifacts, I am most interested in examining the source code of electronic games.

In his article *Combat in Context*, Nick Montfort introduces a five layer conceptual model for thinking about video games. He defines the five layers, in order, as:

- Platform
- Game Code
- Game Form
- Interface
- Reception and Operation

Platform is the underlying software and hardware system that the game runs on, such as a Windows PC, an X-box or an Atari. **Game Code** is "the computer program that realizes a game" (Montfort, 2006). **Game Form** is the game's rules or procedures. **Interface**, "sits between the player and the game form" (ibid.). **Reception and Operation** is the level at which a player experiences the game, and forms their interpretations of the game as a work of literature.

Critical work can take place at each of these levels, with most current work on video games taking place at either the Reception and Operation level, or the Game Form level. The different layers are present simultaneously in every game, and I believe that the most insightful criticism takes them all into account. I am most concerned with the Game Code level, particularly as it informs the processes that take place in the Game Form level. In this paper, I attempt a "full stack" criticism that comments on several levels of Jason Rohrer's Gravitation.

Following Bogost's thinking on the procedurality of games (*Persuasive Games*, 2007), I view the rules or processes that occur in the Game Form layer as the most critically significant for any game. The processes allow the game's author to make arguments and change the way players think and feel about the world. As a programmer, untangling and understanding other people's code is a useful exercise for learning new techniques, but as a literary scholar studying video games, reading code allows for crystal clear explanation of rules which are often difficult to describe in natural language. If the rules are the most important part of a game, and the code forms the rules, then the code is a worthwhile object of study.

Critical Code Studies

While there had been earlier academic work done examining the source code of electronic literature, the term Critical Code Studies was coined in 2006 by Mark Marino in an article for the Electronic Book Review. He describes Critical Code Studies (CCS) as "an approach that applies critical hermeneutics to the interpretation of computer code, program architecture, and documentation within a socio-historical context." (ibid.)

This sort of criticism was being done with respect to electronic literature such as Perl poetry in which the work was essentially code. However, in these works the focus is on the algorithm that generates the work, as the output is frequently nonsensical, and the actual code illegible and obfuscated in order to compress it. This is similar to work from the *Oulipo* which focused attention on the process of making poetry rather than the output.

The innovation Marino offers is broadening the field of inquiry to include all of the code that forms electronic literature. Marino builds on Cayley who, in an article titled [*The Code is not the Text (unless it is the Text)*] (2002), pushes back against much of what he was seeing in electronic literature that involved code. Cayley critiques "codework" that is not executable, rightfully criticizing such work as one dimensional, and not taking full advantage of the programmable nature of computers. Perhaps worse, critical treatments of works that are computable treat the code as a

surface output, as a flat text rather than the rich process oriented work that it is.

In Marino's definition of CCS, he proposes "that code itself is a cultural text worthy of analysis and rich with possibilities for interpretation" (2006). Much of the remaining article is spent exploring and justifying this position, and in doing so he has established the humanistic study of code as a unified practice, rather than several disparate practitioners. A central point that CCS is based upon is the unique property of high level computer code, its doubly encoded nature as a computational object and a natural language text. I explore the implications of this further in the next section; however it is worth noting here how different this is from other forms of media.

One counterargument that Marino addresses is that code is just a byproduct of the process of producing electronic literature. He quotes Jeremy Douglass who, playing devil's advocate, contrasts code with film saying, "Why do I have to study film-development processes, if they do not have a perceivable affect (to the naked, untrained eye) on the image that results." (ibid.) Marino's response is that, unlike the byproducts of film production, code is itself already a text. "This text can be read, understood, and of course, altered." (ibid.)

I think that the choices made in the production of electronic literature or video games do impact the processes or Game Form, which is felt by the reader or player. Another counterpoint I would add is that the code is, in my mind, inherently part of the work, in a way that film byproducts, notes in the margins of a novelist's notebook or sketches of a work of sculpture are not. With a complete copy of the source code it is usually trivial to compile and reproduce the output, whereas in other media the preproduction versions are entirely different works.

Marino acknowledges the challenges of studying code when much of it is hidden behind proprietary licenses. He also accepts that the vast majority of code is not poetry, it is industrial in nature and terse in diction. Given this, I believe that the greatest potential of Code Studies is investigating the Game Code of digital games and unraveling the ways that this impacts every layer leading up to the surface that the player experiences.

Code Aesthetics

> We want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. (Abelson and Sussman, Preface of *Structure and Interpretation of Computer Programs*).

Abelson and Sussman point to the human legibility of computer code, which makes code studies a viable method of criticism. While many uses of the word 'code' imply obfuscation or hidden meaning, well written computer code should do the opposite. It should clearly communicate its purpose and underlying procedures to any reader with programmatic literacy. In software engineering, there are many practical concerns about readability of a computer program. It allows for easier collaboration with others, it is more maintainable and is more portable. Easily legible code also draws attention to the unique property of computer code, that it exists on two communicative layers. On one it speaks to a computer, by way of a compiler or interpreter, and on the other to a human. [1]

In their paper, *A Box Darkly: Obfuscation, Weird Languages, and Code Aesthetics*, Michael Mateas and Nick Montfort (2005) explore the idea that code exists on both planes, and that clear communication with a human reader is not always the goal.

They describe obfuscation contests, such as the International Obfuscated C Code Contest which aims, among other things, "To show the importance of programming style, in an ironic way." The contest archives contain some truly incomprehensible abuses of the language, which are still compilable and runnable programs. The very existence of a contest dedicated to (bad) programming style indicates that writing a computer program is not solely an engineering task.

Mateas and Montfort also cover parodic languages which comment on certain features of industrial or academic languages. INTERCAL parodies non-intuitive programming language design with key words that have no overlap with other known languages, and a compiler that skips non-compiling code rather than flagging it. Brainfuck is a minimalist language which calls "attention to the very small amount of structure needed to create a universal computational system" (2005). It contains just eight single character commands which manipulate pointers and memory locations.

The final class of parody draws attention to "double coding," code that communicates with computers and humans simultaneously. They highlight Shakespeare, in which all variables are declared in the "dramatis personae" and must be named after characters from William Shakespeare plays. Chef, in which variables are ingredients, and "all operations have a sensible interpretation as a step in a food recipe" (Mateas, 2008). In a later adaptation of this paper Mateas comments that

> it is possible to write programs in Chef that might reasonably be carried out as a recipe. Thus, in some sense, Chef structures play to establish a triple-coding: the executable machine meaning of the code, the human meaning of the code as a literary artifact, and the executable human meaning of the code as steps that can be carried out to produce food (ibid.).

These weird languages and obfuscations are notable for their abberance. By diverging from typical coding practice, they provide commentary on code aesthetics, and highlight aesthetic aspects of code that critical code studies should take into account.

"Extra-Functional Significance"

In computer coding, it is considered best practice to leave comments for other users of the code, most often yourself in the future. Comments surround functional pieces of code, specially designated for the computer to ignore at compile and run time. In good software, comments should concisely explain what is going on in a confusing bit of code. Descriptively naming variables is also an

important way to clarify their purpose. Variables are important for the proper functioning of the code; however the computer does not care what designation is chosen for them, so long as it is distinct from other variables. Choosing variable names that describe what they represent matters only to clarify their purpose for human readers of the code. Sometimes, just a few words is all it takes.

Sample describes the controversy that erupted when a variable was discovered in the commercial game *Dead Island* (*Techland*, 2011) named "FeministWhore" (Sample, 2011). People were, rightfully, offended by this, and the studio took appropriate steps to discipline the coder responsible. Variable names, particularly such provocatively named ones, are within the purview of Code Studies. They contribute to the rich textuality of code and, because they are not bound by programming language specific reserved words, are a way for coders to establish voice and style.

However, as Sample points out, this code was found in a "leftover debug file" (ibid. quoting a statement from Techland). He labels it "Zombie Code," code which was supposed to be deleted, does not form any processes in the work, and yet has been given new life. This is what Wm. Ruffin Bailey has called "fossil" code (Bailey, 2008), deleted or uncompiled portions of the software which nevertheless have been discovered.

Bailey covers the "Hot Coffee" controversy that occurred when popular video game *Grand Theft Auto: San Andreas* (*Rockstar*, 2004) was discovered to have a sex minigame contained on the original game disc. Despite the fact that this was inaccessible without a downloaded mod which flipped a bit in one of the game's configuration files, there was considerable controversy in mainstream media outlets, and the game was temporarily pulled from store shelves (Bailey, 2008).

In his eponymous essay on Critical Code Studies, Marino declares that "scholars of CCS will analyze the extra-functional significance of the code" (Marino, 2006). In that respect, comments and variable names are up for consideration. Near the end of the essay, Marino cautions against drawing conclusions that do not properly address the code as a functional object, and which do not shed light on the code itself (ibid.). I would also caution the opposite, to avoid drawing conclusions based

solely on the code, ignoring the electronic literature that it produces.

The fringe cases noted above offer an interesting challenge for critical code studies. They were not intended to be viewed, and so could be considered outside the text of the given game. However, given that most games do not intend their source code to be viewed, all of code studies operates within this fringe. Despite their similarities, these examples are actually quite different in their critical significance.

In the case of *Dead Island*, the Zombie Code had no effect on any other layer of the game stack, and was offensive only at the level of reading the code as a text. This should effect the interpretation of the code as a text, but should not influence our perception of the game itself. In the case of "Hot Coffee," *San Andreas* with the mod enabled is essentially a different game. It would be comparable to considering two different editions of a published book, in which the author has added a new chapter. "Hot Coffee" introduced new processes to the game, changing every layer of the game's stack except platform. Due to the drastic changes it introduces, "Hot Coffee" is a more significant object for a code studies approach.

It is debatable whether "extra-functional" aspects of code such as comments and variable names are as significant as Marino believes them to be. However, the next section provides an example of comments which drastically change the interpretation of a work.

Full stack Criticism

JFK: Reloaded (*Traffic*, 2004) is a simulative game that purports to accurately model the John F. Kennedy Assassination. Players experience the game from the assassin's perspective and are tasked with replicating exactly the findings of the Warren Commission. The game was extremely controversial in mainstream media and was "vocally condemned by a number of prominent people, including Senator Edward Kennedy" (Fullerton, 2008). Despite the initial gut repulsion many feel towards the game, it has attracted serious scholarly attention. By combining their analyses we can

reach a 'full-stack' criticism that takes each layer of Montfort's stack into account.

Tracy Fullerton examined *JFK: Reloaded* as an example of a documentary game. Fullerton's analysis addresses the Perception and Operation level, as well as the cultural context surrounding the game's reception and the scenario it replicates. Ian Bogost (2007) built on an earlier version of Fullerton's ideas, but goes further down the stack, addressing the Game Form, or rules. Bogost is particularly interested in what he calls the procedural rhetoric of the game, which he defines as "the practice of persuading through processes." In this case the processes are the rules of the game.

Fullerton quotes the game's developers, *Traffic*, as saying "we've created the game with the belief that Oswald was the only person that fired shots on that day, although this recreation proves how immensely difficult his task was." The rules of the game seem to confirm this, with a perfect score of 1000 rewarded only to those who exactly replicate the findings of the Warren Commission. *Traffic* offered a cash prize to the player who could produce the closest replication of the real assassination. This proved extremely difficult, as the highest score was 782 out of 1000. Bogost reads the seeming impossibility of the game as its procedural rhetoric, arguing that the game rules contradict its creator's statements, by implying that the Warren Commission's findings were flawed. When in conflict, arguments from within the text are more convincing than arguments from the creator's PR.

Mark Sample delved into the source of *JFK: Reloaded*, specifically the data files that control the many animations in the game (2011). Sample finds comments from the game developers that invalidate *Traffic's* supposed goals with their inaccuracies and sexual crudeness.

> In just three lines of code commentary, the developers at Traffic absolutely undermine the entire stated pedagogical project of their doc-game. Their outwardly respectful "interactive reconstruction of John F. Kennedy's assassination" is undone by inaccuracies and misspellings (Nelly for Nellie, "desparate") but even more so by the explicitly sexual re-framing of this traumatic event. It's difficult to take JFK Reloaded as a serious exploration of history when under the hood it resembles an adolescent

> joke, preoccupied with sex and making light of death ("before he croaks") (Sample, 2011).

These comments appear to invalidate the supposed "seven months to research" (Fullerton, 2008) that *Traffic* put into the game.

The next step would be to determine the validity of Bogost's intuition, that the game was not intended to model the Warren Commission's findings. With access to the source code of the game, not just the data files, it would be possible to dissect the scoring algorithm. It could be impossible to achieve a perfect score within in the game. How would that change our interpretation of the game? If the game is merely difficult, then the rhetoric highlights the difficulty the real assassin experienced. If it is impossible, it casts doubt on the physical possibility of such an assassination even occurring the way the Warren Commission claims it did. Sadly, it is unlikely that the original source files will ever be made available to scholars.

These scholars also make passing references to the other two layers, Interface and Platform. Fullerton comments on its release on PC instead of consoles. This choice of Platform informs the controversial content *Traffic* is able to produce. Consoles are typically locked down by the company that produces them, while the PC has no governing body. It is unlikely that any of the console manufacturers would approve a game like this.

Bogost comments on the Interface, saying it is one of the most engaging simulations of sniping that he has experienced. Sniping is included in most first person shooters, one of the most prolific video game genres. However, Bogost states that

> few are as physically demanding as *JFK Reloaded*. The precision and accuracy required to pull off the three shots of the Warren Commission Report not only struck me as nearly impossible (again casting doubt rather than clarity on the historical record) but also gave me the chilling feeling of the assassin's psychopathy. The

> precision of the game's stated goal helps the player depersonalize its consequences, further emphasizing the simulation of the psychopath-assassin (Bogost, 2007).

The choice of a first person interface intended to place the player closer to the assassin is critically significant. Rather than viewing this historical moment in the large, for its impact on US and world history (Sample, 2011) this game is tightly focused on a five-minute slice of time. The first person perspective serves to make the assassin anonymously insignificant, foregrounding the familiar scene occurring in the plaza below. However, as Bogost shows, players stepping back and reflecting on the experience will be more preoccupied with the peculiar nature of the assassin and his task.

JFK: Reloaded is a historical game, but the criticism presented here is, for the most part, tightly focused within the text. In the next section, the larger social and historical context is critical to interpreting the code.

Reading Industrial Code

Reading code is not just a useful technique for analyzing works of electronic literature. In 2010, the stuxnet virus was discovered infecting computers in Iran, and the anti-virus giant Symantec put a small team on the task of sifting through the source code. While not a widespread threat, the sophistication and novelty made it clear that this was one of the most important computer attacks ever. Wired Magazine documents how the team gradually became convinced that this was the work of a state agent trying to disable Iran's uranium enrichment facilities (Wired, 2011). From a geopolitical perspective, Israel has obvious motivation, and is a logical suspect. Interestingly, the malware researchers discovered several clues in the code that they thought pointed to Israel.

> When Stuxnet first infected a system, before installing its malicious files, it checked the Windows registry for the number 19790509. If the number was there, Stuxnet passed over the system and didn't infect it - like lamb's blood marking the door frames of Jewish homes in ancient Egypt to ward off the Death of the Firstborn plague.

While poetic, this is not unusual, as many malware authors leave such marks to prevent self infection. More unusually this particular registry number can be read as the date May 9th, 1979. Wired explains that this was "the day an Iranian Jewish businessman named Habib Elghanian was executed by firing squad in Tehran."

This combination of reading the source code for both its complex technical capabilities and its historical allusions shows an interesting way forward for reading the source code of electronic literature. The next case study extensively studies both the code and its inspiration.

Spelunking *Colossal Cave*

In the Digital Humanities Quarterly, Dennis Jerz of Seton Hill University wrote a groundbreaking essay (2007), in which he explored both the source code and source cave of "the classic text computer game *Colossal Cave Adventure*" (Crowther and Woods, 1976). Adventure was an early video game which circulated on the pre-web Internet and was hugely influential for later interactive fiction. The game was originally made by Will Crowther and later modified by Don Woods. One of Jerz' goals in the piece is to settle cases of authorship, which proved difficult when dealing with decades old memories, but was clarified by examining the source code. Jerz also explores the actual cave that Crowther based his original game map on, finding that many in game locations exist exactly as described.

In his study, Jerz explores the source code of *Adventure* before and after it was modified by Woods. These files were found on a backup of Woods' old Stanford student computer account, an interesting find for digital archeology. Jerz determines that, despite common claims to the contrary, Crowther's original game was not solely a cave simulator. It contained fantasy elements, puzzles, and some of the magic words that entered broader computer culture, such as "XYZZY." It had been previously reported that Woods alone added these game-like elements.

Jerz also found an instance of "fossilized" code in the game's data files. A list of possible ingame items included "FUCK", with a line number that is uncalled by any portion of the code. This could be an example of programmer frustration, or some portion of the game that Crowther thought about but did not implement. Jerz notes that it also casts some doubt on Crowther's claims that he wrote the game for his children, although that could be an explanation for its fossilization.

Jerz spends much less time on the source code than he does on the source cave. Crowther was an avid caver, basing the geography and some in game items on the Bedquilt entrance of Colossal Cave in Kentucky. For the essay, Jerz was led through the cave by several old caving companions of Crowther, and he includes numerous photographs of locations in the cave that correspond to the game. By combining the source code, cave, and interviews with Crowther, Woods and their family and friends, Jerz is able to construct a fascinating study of the authorial process of a historic early video game.

While very fruitful for the study of *Adventure*, Jerz' approach is not a replicable way forward for reading code. It closely resembles the process Symantec took in evaluating Stuxnet. However, I would hazard a guess that most games owe more to untraceable imagination, or inspiration from other media, rather than a specific physical location. Indeed, Woods had never been caving, and this shows in the puzzles he introduced which broke from the experience of an actual Kentucky Cave.

Jerz did not intend his study to be "Critical Code Studies." Indeed no such label existed when he began the article. Still, his technique of explaining the workings of the code with relation to the surface level of the game is one I hope to emulate.

Adventure is actually a game whose source code has experienced a relatively large amount of critical study, when compared with most games. It was the "sole example" provided by Donald Knuth (ibid.) of his CWEB system for literate coding. After Jerz' article in DHQ, the Critical Code Studies working group explored the code further.

The discussion ranges across the board, from cultural studies of the 1970's programming subculture to personal recollections of scholars' childhood experiences with the game. Many people commented on the sheer difficulty of reading this code, which is written in a computer language and paradigm that is incomprehensible to those of us who have learned to code post-"goto considered harmful." [2] This again points towards the aesthetics of code. While Knuth considered it beautifully expressive, it is unreadable to most modern programmers.

Gravitation

Jason Rohrer describes his *Gravitation* as "a video game about mania, melancholia, and the creative process". In *Newsgames* it is provided as an example of a "human interest" game, commenting on games' "ability to reconstruct personal emotional experiences rather than just describing them" (Bogost et al, 2010). The game explores Rohrer's cycles of creative mania, and places the player in the position of balancing "abstract, difficult work, and its interplay with the inspiration that comes from family interaction" (ibid.).

The game opens with your character in a small box of visibility tinged with gray, the rest of the screen obscured by blackness. To the right of the screen is a furnace, and moving to the left reveals a small child avatar who begins throwing a red ball to you. If you bat the ball back, a heart appears over the child's head, but if you don't and the ball hits the ground, tears sprout from the child. As you play ball, the visible box grows larger, and the area within becomes more colorful and brightly lit. This is a simple, powerful ludic metaphor for depression, its antidote being human interaction with loved ones.

At a certain point, when your "mood" raises high enough more of the world is revealed, your head appears to catch fire and you can jump incredible distances. Jumping above your starting level reveals stars, which fall downwards when touched. Your "mania" only lasts a short time, after which your view closes up and the colors darken again. Unable to jump higher, you navigate your way down to the original level, where the stars have become blocks of ice with point values. The points

start at 9 and count slowly downwards. Pushing the blocks of ice into the furnace on the right of the screen adds their current point value to your total. The child continues to throw the ball when you return, though the piles of ice can cut you off from him until they are cleared off. Initially, straining against your arrow keys can feel difficult as the ice moves slowly towards the fire. Playing ball with the child increases your mood, and if you re-enter mania ice piles can be cleared more quickly.

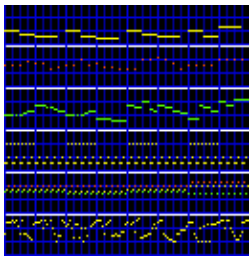
There are several additional aspects that make *Gravitation* an interesting work of literature. Firstly, while playing with the child can increase your mood, it does not increase your point total. It is possible to play through the entire game just batting the ball around, but your score will remain 000. You also do not receive points for harvesting the stars, only for pushing the blocks of ice into the furnace. While you have spurts of depression when you are exploring the space above the ground level, they can be waited out and your mania will return. You can continue to explore the heights of pure creativity above for the entire game, but without the difficult work of pushing those ideas into the furnace, your score remains 000.

This constant returning to the home level makes *Gravitation* feel like a balancing act, enforcing the work-life balance theme that is at its core. *Gravitation*'s most moving portion comes towards the end of the game. At a certain point, while you are collecting stars above, the child leaves. You return home to find the ball sitting alone on the ground. Mechanically, the game continues unchanged, but the emotional tenor shifts drastically. Collecting your stars begins to feel empty.

One of the key components of the game's atmosphere is the music, which keeps pace with the shifting mood. In the beginning of the game, the music is slow and mournful. As your mania increases, the initial music becomes overlaid with other tracks, which fade out again as your mood darkens. The volume and tempo increases and the music becomes discordant as your mood intensifies.

Rather than a typical audio type like .ogg or .mp3, the game's music is encoded in a .tga file, an extension most typically used for image files. In fact, if you open the file with an image viewer, it

reveals waves that appear to represent six different musical tracks.



The thicker white horizontal lines delimit the different tracks. The first track has long yellow lines that correspond to the slow, low brass sounding initial track. The second track has red dots, that sound like a xylophone. The third green track is the initial melody. The next three tracks are activated during mania, and its build up. The two tracks with many small dots spaced at regular close intervals to each other appear to be the game's drum tracks. The last track has yellow dots which only play during the height of mania, and provide a rapid, high pitched counterpoint to the initial melody.

Investigating the musicPlayer.cpp file (lines 17-20) reveals that the music changes based on the game state:

```
// smoothly fade in particular tracks based on player emotion
 // low emotion plays only first track... high emotion plays all tracks
 extern double playerEmotion;
```

Rohrer's comment (delimited by the double slash marks) precedes his loading of the floating point variable "playerEmotion", which represents the avatar's emotional state. It operates on a scale with 0 being total melancholia and 1 being total mania. It is used later (lines 233-236) as a multiplier for fading tracks in and out.

```
// factor in player emotion

 // level from 0..(numTimbres-1)
 double trackFadeInLevel = playerEmotion * (numTimbres-1);
```

Music is not the only place the playerEmotion variable comes into play. It is initially set in game.cpp (lines 536-538) with comments to explain its usage:

```
// 1 = manic
 // 0 = depressed
 double playerEmotion = 0.4;
```

Several lines later Rohrer defines another variable:

```
double defaultDeltaPlayerEmotion = -0.0010;
 double deltaPlayerEmotion = defaultDeltaPlayerEmotion;
```

The second variable, deltaPlayerEmotion, is the rate at which player emotion changes each frame of the game, and is initialized to -0.0010, meaning that you begin the game on a downswing. In the next few lines Rohrer defines several upswing variables that provide a "natural depression recovery" (line 548). These variables provide a snapshot of the fairly simple system Rohrer uses to model human manic depressive cycles. It affects everything, from the music to the size and color of the visible world.

In this code Rohrer is treating the "playerEmotion" to be state of the game player's avatar, not the player themselves, which would be much more difficult to assess. However, the discordant music will affect the human player's mood. By tying the visual and auditory representation of the world to the avatar's emotional state, Rohrer can change the human player's mood. Even the mechanical changes, such as increased strength and jumping ability serve to make the player feel powerful during mania and weak during depression.

Playing ball with Mez increases the playerEmotion variable, by 0.15:

```
if( mezCaughtBall ) {
      playerEmotionSmoothTransitionTarget = playerEmotion + 0.15;
```

playerEmotionSmoothTransitionTarget gives the point that playerEmotion is aiming for, but Rohrer slows the transition for a less abrupt jump upwards. In a separate World.cpp file I found the portion of the code that explains the way touching stars, or prizes as Rohrer calls them, changes your emotional state. From within a function named touchPrize (line 1374-1378):

```
// renew mania
 playerEmotionSmoothTransitionTarget = 1.0;
 // accellerated descent toward depression
 deltaPlayerEmotion *= 2;
```

The first non-comment line resets your current emotional target back to full mania, while the second line of code doubles your rate of descent. It results in the sometimes wild swings of emotion that can be experienced while searching for stars. This is one of the most fascinating portions of the code, as it reveals something new that I was unaware of from playing the game alone. It was apparent that touching prizes had some effect on your emotion, but I was unaware of the doubled descent into depression. From a rhetorical perspective, this rule has some fascinating implications. Collecting the prizes is actually a faster way to gain mania, but note that the accelerated descent is a multiplier not a fixed value. Collecting five prizes in a row would result in a tenfold increase in your downswing. This is in contrast to the slower mania building of playing ball which has no such emotionally turbulent side effects.

game.cpp also contains the code controlling the child's disappearance (lines 1406-1414):

```
// last 3/8 of game
 if( timeLeft <= 0.375 * totalTime ) {
      // mez "sneaks" away if he's off screen near the end of the game
      if( ! isMezOnScreen() ) {
           hideMez();
      }
 }
```

Several things are happening in this code. The first line is a comment explaining what the second does, which is to determine how much time is left in the game as a ratio to the total time of the game. As the next comment explains, after that point, any time the child is off screen he will leave. The isMezOnScreen() function returns Mez' location on screen. A null value, meaning Mez is offscreen, will be interpreted as false by the if statement. The exclamation point can be read as "not", so the entire line checks if Mez is being displayed, and if not, moves to the body of the if statement, hiding Mez.

In some ways seeing it in code is anticlimactic. It clarifies some things about the game. The child is a boy named Mez, which turns out to be Rohrer's son's name. His disappearance is nearly inevitable, unless you spend the entire last 3/8 of the game with him. Furthermore, there is no chance of Mez remaining if you leave.

When explained in clear code, this portion of Gravitation can be read as an exploration of the parental impulse to "helicopter parent". On the one hand, doing so prevents the child from moving on to a fruitful adulthood, but in *Gravitation* you cannot watch Mez grow up. A boolean true/false value gives you no middle ground.

There is much more to explore in the source code of this game. It is very clearly written, profusely commented, and from both a programmer's and literary critic's perspective it contains some clever tricks. I am particularly interested in locating where the score is calculated. While it seems to be a clear linear reduction of points after the ice blocks hit the ground, investigating the code would verify the algorithm.

The most interesting aspect of *Gravitation's* platform is that it is open source, enabling this paper. The openness of the code allows people to swap in new music files, changing the tenor of the entire experience. It also enables anyone to modify the game the way they believe it should be. In a different version of the game it could be possible to score just as many points by remaining with Mez and playing ball. Such a game would be procedurally portraying different values, and would be a

significantly altered game, without the balancing act that makes *Gravitation* such a challenge.

Conclusion

There are still many unanswered questions for Critical Code Studies. Should the code be treated as a separate text apart from its output? What significance, if any, should be placed on "extra-functional" aspects of the code, such as comments and variable names? These, and many others, are questions that are at the forefront of the emerging code studies field.

Code studies is a useful technique for exploring certain works of electronic literature. It will certainly not be applicable everywhere, but when it does succeed the results are intriguing. I learned different code studies techniques from other authors' studies, and combined them to form my own full-stack criticism of a literary video game. Combining close readings of the several layers occurring simultaneously in a game produces insights that could not be gained by reading only one layer at a time.

Endnotes

[1] One amusing example of this in my own life comes from a friend of mine in my computer science classes who consistently tries to name his variables something more interesting than 'i' for an iterator or 'n' for a number. For example, he names his characters 'char mander', 'char meleon', and 'char zard'. Only programs which require three character variables get to fully evolve. These conventions don't alter the runtime characteristics of his program, but they endear him to his classmates and exasperate some of his professors. When struggling through a late night debugging session, seeing 'bool tentabool' is much funnier than it probably should be, and eases some stress.
Return

[2] As one clever slashdot commenter noted "Maybe the inspiration for the 'twisty little passages, all alike' *wasn't* Mammoth Cave, it was the code itself".
Return

Works Cited

Bogost, Ian. *Persuasive Games: The Expressive Power of Videogames*. The MIT Press, 2010. Print.

Cayley, John. "The Code Is Not the Text (unless It Is the Text)." *Electronic Book Review* 26 Jul. 2005. 26 Oct. 2011. <http://www.electronicbookreview.com/thread/electropoetics/literal>.

"Critical Code Studies Conference - Week Three Discussion - Dennis Jerz." 26 Oct. 2011. <http://www.electronicbookreview.com/thread/firstperson/colossal>.

Jerz, Dennis. "Somewhere Nearby Is Colossal Cave: Examining Will Crowther's Original Adventure in Code and in Kentucky." *DHQ: Digital Humanities Quarterly* Summer 2007. 31 Oct. 2011. <http://digitalhumanities.org/dhq/vol/1/2/000009/000009.html>.

Marino, Mark. "Critical Code Studies." *Electronic Book Review* 12 Apr. 2006. 26 Oct. 2011. <http://www.electronicbookreview.com/thread/electropoetics/codology>.

Mateas, Michael. "Weird Languages." *Software Studies, a Lexicon*. Ed. Matthew Fuller. Boston, MA: MIT Press. <http://users.soe.ucsc.edu/~michaelm/publications/mateas-software-studies-2008.pdf>.

Mateas, Michael, and Nick Montfort. "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics." Copenhagen, Denmark, 2005. <http://users.soe.ucsc.edu/~michaelm/publications/mateas2-dac2005.pdf>.

Montfort, Nick. "Combat in Context." *Game Studies* Dec. 2006. 31 Oct. 2011. <http://gamestudies.org/0601/articles/montfort>.

Rohrer, Jason. "Gravitation." 31 Oct. 2011. <http://hcsoftware.sourceforge.net/gravitation/>.

Sample, Mark. "A Revisionist History of JFK: Reloaded (Decoded)." *Play the Past*. 31 Oct. 2011. <http://www.playthepast.org/?p=1519>.

---. "Criminal Code: The Procedural Logic of Crime in Videogames." *Sample Reality* 14 Jan. 2011. 31 Oct. 2011. <http://www.samplereality.com/2011/01/14/criminal-code-the-procedural-logic-of-crime-in-videogames/>.

---. "Rebooting Counterfactual History with JFK Reloaded." *Play the Past* 19 May 2011. . <http://www.playthepast.org/?p=1392>.

---. "Zombie Code and Extra-Functional Significance." *Play the Past* 13 Sep. 2011. 31 Oct. 2011. <http://www.playthepast.org/?p=1989>.

Whalen, Zach, and Laurie N. Taylor. *Playing the Past: History and Nostalgia in Video Games*. Vanderbilt University Press, 2008. Print.