Assessing Adaptive Learning Styles in Computer Science Through a Virtual World

by

Nizar Kury

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2017 by the
Graduate Supervisory Committee:

Brian C. Nelson, Chair
Ihan Hsiao
Yoshihiro Kobayashi

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

Programming is quickly becoming as ubiquitous and essential a skill as general mathematics. However, many elementary and high school students are still not aware of what the computer science field entails. To make matters worse, students who are introduced to computer science are frequently being fed only part of what it is about rather than its entire construction. Consequently, they feel out of their depth when they approach college. Research has discovered that by teaching computer science and programming through a problem-driven approach and focusing on a combination of syntax and computational thinking, students can be prepared when entering higher levels of computer science education.

This thesis describes the design, development, and early user testing of a theory-based virtual world for computer science instruction called *System Dot*. *System Dot* was designed to visually manifest programming instructions into interactable objects, giving players a way to see coding as tangible entities rather than text on a white screen. In order for *System Dot* to convey the true nature of computer science, a custom predictive recursive descent parser was embedded in the program to validate any user-generated solutions to pre-defined logical platforming puzzles.

Steps were taken to adapt the virtual world to player behavior by creating a system to detect their learning style playing the game. Through a dynamic Bayesian network, *System Dot* aims to classify a player's learning style based on the Felder-Sylverman Learning Style Model (FSLSM). Testers played through the first half of *System Dot*, which was enough to test out the Bayesian network and initial learning style classification. This classification was then compared to the assessment by Felder’s Index of Learning Styles Questionnaire (ILSQ). Lastly, this thesis will also discuss ways to use the results from the user testing to implement a personalized feedback system for the virtual world in the future and what has been learned through the learning style method.

ACKNOWLEDGMENTS

Thank you, Dr. Brian Nelson, for not only being my thesis committee chair, but also a mentor for the past two years and a guide through the frustrating world of computer science education. Thank you, Dr. Sharon Hsiao, for serving as a member on my committee as well as being patient with me when it came to picking your brain about software adaptiveness. Lastly, thank you, Dr. Yoshihiro Kobayashi, for being a member of my committee on short notice and teaching me these game development principles for the past three years.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

1. COMPUTER SCIENCE EDUCATION TODAY

1.1 Overview

This thesis describes the design, development, and early user testing of a theory-based virtual world for computer science (CS) instruction called *System Dot*. *System Dot* was designed with learning theory in mind. The virtual world visually manifests programming instructions into interactable objects, giving players a way to see coding as tangible entities rather than text on a white screen. In order for *System Dot* to convey the true nature of computer science, a literature review was conducted surveying the perceptions, problems, and solutions of current CS education. The next two sections will explore the design and technical decisions taken to achieve an effective CS learning environment.

Afterward, the thesis will shift gears and discuss the steps taken to adapt the virtual world to player behavior through the implementation of a dynamic learning style detection system. Through a dynamic Bayesian network, *System Dot* aims to classify a player's learning style based on the Felder-Sylverman Learning Style Model (FSLSM). To test out the Bayesian network and initial learning style classification, testers played through the first half of *System Dot*. This classification was then compared to the assessment by Felder's Index of Learning Styles Questionnaire (ILSQ). Finally, this thesis will also discuss ways to use the results from the user testing to implement a personalized feedback system for the virtual world in the future and what has been learned through the learning style method.

1.2 Introduction

According to the US Bureau of Statistics, more than half the jobs today require some form of experience with using computers. By 2023, experts predict that number will increase by twenty-seven percent (Microsoft, n.d.). Unfortunately, most K-12 schools do not currently express the importance of the computer science field. Currently, opportunities to learn about and practice computational thinking and design in schools is set aside for a relatively small sub-population of students who actively pursue it. To make matters worse, few schools in the United States allow CS to contribute towards graduating high school, only five percent of high schools offer AP Computer Science, and only forty percent of schools offer any kind of CS course (Freeman et al., 2014). This lack of exposure to the field at a young age handicaps students by not providing them with a foundation to build on when seeking a logic-intensive degree.

When asked why CS courses are not actively being implemented in their schools, principals and superintendents claim that there is an absence of interest from their students (Gallup, 2016b). However, a poll by Gallup indicates that 90% of parents and students in the US have a favorable view of CS (Gallup, 2016b). Perhaps this perception by principals and superintendents stems from the lack of success students achieve in CS courses. Even if CS courses become the new norm in elementary and high schools, there is still the problem of the field being introduced incorrectly in a way that harms a student’s academic future. Schools are essentially boarding a myriad of students on the “CS Ship” to an uncharted territory, but not preparing them for the dangers and mysteries this land holds. If a solution arises to why students are struggling with CS both in and before college, then programming classes can be introduced in the *right* way.

The following papers mentioned in this section of the thesis discuss ways in which CS can be taught in an effective and resourceful fashion while also avoiding the pitfalls many educational institutions fall in. For instance, problem-solving (a key component of computational thinking and programming) should not be taught in a repetitive, "plug-and-chug" kind of way. Instead, students should approach problems with a computational, creative mindset. Furthermore, CS is being deceptively introduced, malforming students' preliminary perceptions of CS. Afterward, we will explore how *System Dot* implements the solutions presented in the studies.

1.3 Perceptions of Computer Science

One of the primary reasons for the lack of student participation in CS courses may be their preliminary perception about the difficulty, relevance, and 'coolness' of the material. K-12 schools introduce CS haphazardly, often failing to illustrate what the field entails. When students receive an inaccurate depiction of CS when they are first introduced to the subject, it not only reduces the possibility that the student will take any interest in computing, but it also prevents them from giving a career they would have thought interesting a chance (Bell, Alexander, Freeman, & Grimley, 2009). There are three main distinguishing factors that sway a child's mind in an adverse direction when thinking about CS. The first is a skewed understanding of CS itself-- a majority of students mistake CS for mere computer use and Internet browsing (Ben-Ari, 2001;

Carter, 2006; Gallup, 2016a; Gallup, 2016b). The second is a teacher’s unwarranted emphasis on math when introducing CS and when describing the requirements for a CS degree, which deters weaker math students or those with low self-efficacy in math from discovering how logic and not math-intensive the field actually is (Bell et al., 2009; Ben-Ari, 2001; Carter, 2006; Gallup, 2016a; Gallup, 2016b). Lastly, popular culture is painting CS in a harmful, “uncool” light causing not only students but also parents to negatively stereotype and judge the field with extreme prejudice (Carter, 2006; Gallup, 2016a; Gallup, 2016b). Each of these factors will be discussed in further depth below with problems and solutions outlined by relevant studies.

1.3.1 What is Computer Science?

What exactly does computer science entail? According to Gallup, computer science is “... the study of how computers are designed and how to write step-by-step instructions to get them to do what you want them to do” (Gallup, 2016b). Of course, computers are used to exercise these sequential instructions, but that should not be associated with the main purpose of CS. However, when 836 high school pre-Calculus and Calculus students were asked to define CS in one study, four out of five students reduced the field to a process of writing step-by-step instructions, or programming (Carter, 2006). The Association of Computing Machinery or ACM states that CS not only includes programming, but also hardware design, networking, graphic systems, databases, computer security, logic, and artificial intelligence, just to name a few (Barr &

Stephenson, 2011). Surprisingly, only two percent of students in the previous study correctly identified the full extent of CS (Carter, 2006).

Two major problems arise from this misunderstanding of CS, representing opposite sides of a spectrum. On one hand, students may be disinterested at the thought of writing code and getting entangled with its syntax. Consequently, they will immediately dismiss any desire to explore CS and not discover that the subject revolves around much more. On the other hand, students may think that CS *only* involves programming and become addicted to writing programs and executing them. However, when they reach more difficult concepts like memory management and algorithmic complexities, they start to feel like the material is out of their depth. Their early misconceptions start holding them back. With either of these two problems, students give up on CS based on the false perception that CS only encompasses coding.

These problems stem in part from a learning theory called *constructivism*. According to Ben-Ari (1998), constructivism is how "... knowledge is actively constructed by the student, not passively absorbed from textbooks and lectures…" (Ben-Ari, 2001). When a student is told in the beginning of their CS career that CS is only about programming, they will frame subsequent knowledge building about CS based on their initial misconceptions. Consequently, no matter how many times later lectures or textbooks stress material outside of programming, the student will continue to find a way to tie it into their early conception of CS as programming only.

Ben-Ari also points out that, just like math instruction, teachers refuse to expose the true nature of CS in order to keep the student engaged. Over time, as the student learns more about the hardware, artificial intelligence, and so on, they begin forming misconceptions about CS that derail their progress. Conversely, if the student had a relaxed model of CS and approached the subject from a constructivist point of view, then Ben-Ari argues that those misconceptions do not become errors of judgement, but logical conceptions from an unwavering perception of CS. Therefore, the student remains fascinated and persistent with their pursuit of CS rather than deceived and fatigued by its surprises (Ben-Ari, 2001).

1.3.2 The Façade of the Need for Strong Math Skills

Problem-solving has been traditionally taught in math classes at an elementary school level; it is a child's first foray into the unknown world of numbers and how to solve problems with them. Because CS involves solving similar types of conundrums, teachers and principals tend to associate CS to math. A recent study by Gallup (2016a) estimates that half of teachers and principals believe that people who are proficient with math are equally as proficient with CS. As a result, they (1) stress that only math-oriented students should take CS and (2) CS requires an intensive amount of mathematical knowledge to be successful in it. Gallup polled students' thoughts on math and discovered that about forty percent think they are "very skilled" in math or science. Because students believe they need to be "very skilled" in math, they start to view CS as a field that only "smart" people are successful in. So much so that half of students believe that they need to be <u>very</u> smart to learn CS. The more mental barriers are placed on CS, the less students will be enticed with giving it an opportunity.

A strong emphasis on mathematics also harms how CS is being introduced in K-12 classrooms. In these classes, students are being taught methods for solving complex arithmetic and number theory problems. One of these is the infamous “plug-and-chug” method, which simply relies on the student finding a problem of similar stature and applying the old problem’s methodologies of solving it to the new problem. Since teachers and principals associate CS so heavily with mathematics, they begin to teach these classes like a math class. When approaching computational problems, they employ a plug-and-chug method. However, as illustrated by Barr and Stephenson (2011), computer science demands *computational thinking* and not simply a rote systematic approach when tackling its problems.

According to the efforts by the Computer Science Teachers Association (CSTA) and the International Society for Technology in Education (ISTE), computational thinking is “... an approach to solving problems in a way that can be implemented with a computer. Students become not merely tool users but tool builders” (Barr & Stephenson, 2011). Instead of giving students the tools to solve computational problems, teachers should start exposing students to the intricacies of these tools and how to build them from scratch. Traditionally, when a student gets a math question wrong, they tend to give up on finding a solution another way because the systematic way it is being solved does not allow flexibility or hope that a solution is there. It is like being told to “follow the yellow brick road” and then finding a dark pit at the end; the student is instantly lost once they reach the end of the systematic approach.

However, as outlined by the CSTA and ISTE, computational thinking opens a student’s mind to this flexible thinking style. When a student fails, teachers embrace that failure and stress to students that preliminary failure is the path to success just like how a computer continues to find a solution to a problem no matter how many roadblocks are in its way. Additionally, students have more tools in their belt when problem-solving creatively. Instead of exclusively plugging-and-chugging, students will be able to decompose a problem, negotiate between different avenues of attack, and then agree on an overall plan (Barr & Stephenson, 2011). If that plan fails, they have other avenues to try. These learning methods need to be implemented in order for computer science to be introduced and taught effectively and engagingly.

1.3.3 Amusing Ourselves to Disassociation

Popular culture plays an important role in forming stereotypes many students and parents think of when someone asks them who they know in CS. Popular television shows like *Silicon Valley* and *Big Bang Theory* typically depict programmers as skinny white males who spend hours in front of a computer, tirelessly coding or hacking away with little to no social interaction. According to Gallup (2016a), only seven percent of minority students like Blacks or Hispanics report seeing people who look like them doing computer science in movies or TV shows. Furthermore, more than sixty percent of students say boys are more suited for and interested in learning CS than girls. Research by Google shows that high school students who saw reflections of themselves handling computers and programming were more likely to be enticed to learn about CS (Gallup, 2016a). These perceptions stem from what students see every day; CS is not being advertised appetizingly in our culture and that needs to change.

The classrooms cannot fix the problem. When students don’t see other students of similar race or gender pursuing CS, they will be highly unlikely to pursue it. As a result, when those students don’t learn CS, it continues to perpetuate a CS student population made up mainly of white males. The problem becomes a Catch-22 that K-12 schools cannot internally fix unless CS becomes a required course for graduation. However, if this perception of CS in our popular culture through television shows, fiction novels, and films changes, there can be a push towards an acceptance of these diverse CS students into the field.

Nonetheless, Carter (2006) discovered that even if the CS population becomes more diverse, there is still a negative perception of what CS professions actually do. Based on her survey, the strongest influence against CS was the lack of desire to sit in front of a computer all day. Additionally, students, specifically female, perceived that CS had no relation to another college major they were choosing and did not even consider it. Perhaps these perceptions come from programmers depicted in popular culture as sitting in their parents’ basement playing video games and coding. When all females see are people who know CS using it to isolate themselves from society or enjoying computerized forms of entertainment, it is hard for them to visualize how CS can be extended to include their academic passions (Carter, 2006).

CS is incredibly social and applicable to almost any field. Whether it is a medical paper, business plan, or defense case, the logical skills learned from CS can be applied to anything that requires a logical construction of information. For instance, the imperative flow of a program acts like the argumentative flow of a philosophical paper. Furthermore, companies nowadays have found it a necessity for programmers to communicate with each other in order to overcome hurdles more cost-effectively and time efficiently. In fact, Williams, Wiebe, Yang, et al. (2002) conducted a study about how students retain more knowledge when they *pair program*. According to them, pair programming is "... a style of programming in which *two* programmers work side-by-side at *one* computer, continuously collaborating on the same design, algorithm, code, or test" (Williams et al., 2002). When they implemented this practice with a classroom, they saw a 23% increase in passing grades and overall enjoyment over a classroom that remained solo. Socializing and working in teams is a vital component in CS and it is popular culture's turn to catch up to this trend. Overall, until students stop disassociating themselves from CS and creating excuses for why they are not the *right* fit for CS, schools should instead be trying to fit CS into students' lives by destroying these harmful deceptions being advertised.

1.4 Problems with Current Computer Science Instruction

In addition to the lack of CS emphasis in many K-12 schools, there is also a problem in the way CS is being taught in the relatively few schools that are offering a class or club. For example, the College Board's AP Computer Science Exam facilitates a negative viewpoint of CS by stressing programming and syntax over general computer knowledge and infrastructure (Carter, 2006; Gallup, 2016b). Additionally, some past CS curricula attest to this false prioritization of syntax rules by failing to mention how any programming is used in the foundation of a computer. Teachers also tend to use external tools like *Scratch* or *CodeAcademy* to introduce or strengthen concepts, but they do not realize that use of these kinds of tools may be damaging the way students think about CS (Cooper, Dann, & Pausch, 2003; Meerbaum-Salant, Armoni, & Ben-Ari, 2013). Let us discuss more specifically how each of these methods of current instruction are harming the way students are being introduced and taught CS.

1.4.1 Downfalls of the AP CS Curriculum

As discussed previously, the association of CS education with math instruction can lead to CS being taught using a rote, systematic approach rather than computationally and creatively. Unfortunately, this false methodology is also being stressed in the structure of the College Board's AP Computer Science exam, a high school student's first real preface to the CS college major. The main programming language the course revolves around is Java, a high-level object-oriented language that is English-like in its syntax and relatively simple to understand. According to the AP Computer Science A Course Description from Fall 2014, which can be found on the College Board's website, the main topics are divided as follows for the main multiple-choice section of the exam:

- 55-75% Programming Fundamentals
- 20-40% Data Structures
- 5-15% Logic
- 25-45% Algorithms and Problem Solving
- ...

- 2-10% Software Engineering

Over three-quarters of the exam directly revolves around programming while the other sections (logic and software) are derivatives of programming taught around a coding framework. As discussed in ACM’s definition of computer science, the AP Computer Science Exam does not address hardware design, networking, software architectures, graphic systems, databases, computer security, logic, artificial intelligence, and so on. None of this would be a problem if there were additional K-12 CS classes introducing these tenets. However, since AP Computer Science is the *only* college-supported CS class for K-12 schools and is introduced so late in a student’s academic career, it reinforces misconceptions of what CS is and does not give the appropriate support for students continuing to learn CS in college.

1.4.2 Drag Until You Drop

Some methods of initial CS instruction, particularly for younger students, involves taking students from an integrated development environment and placing them inside a virtual world to learn programming and its semantic concepts. According to Carter (2006), the number one reason male students were interested in CS was the prospect of developing computer games. Therefore, a large push by the CS community to have prospective students be more excited about CS is to immediately give them the ability to create interactive games using beginner-friendly tools.

Imagine taking a page of code from a program, translating the code into general action and statement blocks, dragging these blocks into an organizational tower, and executing those blocks of actions one by one from the top down. This is what the programming environment looks like in *Scratch* and it allows users to create their own pieces of software including games. Scratch, designed by MIT primarily for children aged 8 to 16, aims to provide a user-friendly, approachable way to program stories, animations, and games. Scratch users see how different programming concepts are color-coded into puzzle pieces that need to be correctly connected through dragging and dropping instead of typing. Then, when the user hits the green “Play” button, they see what blocks are being executed through highlighting. This workflow promotes imperative programming-- there is literally a step-by-step execution of instructions. However, the problems with CS education described earlier persist with this new form of syntax-less programming, and this is shown through an experiment conducted by Meerbaum-Salant, Armoni, and Ben-Ari.

The goal of Meerbaum-Salant, etc. (2010) experiment was to investigate how *Scratch* can accelerate the learning of CS concepts through its learn-by-making constructivist philosophy. By testing forty-six students in two classrooms, the experiment consisted of eleven conceptual CS chapters ranging from the theory of concurrency to arrays and loops. Each chapter featured a *Scratch* project that students would work on after brief instruction about a needed specific CS concept from the teacher. To evaluate the results of their experiment, the group used three tests (pre, interim, and post). They discovered that not many students could give an adequate explanation of basic CS concepts such as initialization, loops, and variables. Only about half of the students completely understood what concurrency entailed, and most used *Scratch* instead of CS terms to describe programming. Once again, disassociating CS from how it is harms students' construction of the material later in their academic lives and may deter any long-term interest in CS (Meerbaum-Salant et al., 2013).

Cooper et al. (2003) performed an experiment using a virtual world to teach CS. They aimed to teach CS to three introductory classes at Ithaca and John Hopkins University with an objects-first approach rather than an example-based strategy where simple example programs are shown off and then gradually turn into complex ones. Instead of *Scratch*, they used a 3D virtual world called *Alice* where code is also generated through a drag-and-drop editor. Once again, this eliminates the need for users to understand and make use of the syntax-heavy structure of regular programming. After a semester, they discovered that even though students had a strong understanding of objects and their behaviors, they could not wrap their head around Java/C++ syntax. The authors of this study argued that teaching this extra component of syntax would be easy. However, students have already formed a misconception of CS once they constructed the idea of CS around a graphical user interface instead of an integrated development environment. Rather than being a simple transition, teaching these students the syntax component of programming will be challenging because their mental understanding of what CS was in the *Alice* projects would fight them throughout that process of knowledge acquisition (Cooper et al., 2003).

The graphical drag-and-drop interface of introductory programming development environments like *Scratch* and *Alice* are disrupting the way CS should be introduced. Even though these applications are trying to show students that CS can be used to create extravagant worlds and games, they are also concealing the reality of what CS is. Imagine teaching an English class using audio books instead of text. Despite giving those students the ability to construct literary worlds and derive thematic meaning, it would be extremely difficult for them to translate that knowledge into a well-written in-depth literary analysis without deconstructing sentence structure and wordplay. CS needs to be introduced with some level of programming syntax for students to construct a reasonable initial image of CS to branch from.

1.4.3 Unplugging the Physical Barriers

In an effort to reach more students in economically impoverished areas or those attending schools without available computers, an initiative by Canterbury University called "Unplugged" aims to teach CS concepts through kinesthetic, physical activities rather than through computer labs and textbooks (Bell et al., 2009). Because computers are not involved, a majority of these concepts deal with hardware design or logistical conundrums. For instance, to demonstrate deadlocking in an operating system, students sit in a circle and pass fruits to students around until the fruit matches the student's shirt. This form of instruction promotes not only the realistic sense of social interaction that occurs with CS, but also encourages the child to find solutions for themselves in a constructivist way.

Lambert and Guiffre (2008) conducted a study testing the effectiveness of CS Unplugged in elementary schools, more specifically three fourth grade classes. Being able to target fourth graders allowed the pair to conduct their study on a "fresh" slate of students eager to acquire new knowledge. After three activities and weekly pre-tests and post-tests evaluated on a five-point Likert scale, they discovered that students were more interested in CS overall and had a higher cognitive competency level. In these instances, computational thinking is being addressed despite a lack of programming and its syntax (Lambert & Guiffre, 2009).

Even though CS Unplugged fills in the gaps of other CS instructional methods through an overall depiction of the CS field, the lack of interactivity with a computer and programming still hinders a student's progress with CS similar to drag-and-drop interfaces. Just like how a prospective chef needs to be in a kitchen handling cooking utensils, a prospective CS student should be familiar with a lab environment dealing with computers. Otherwise, their construction of CS is flawed. As Ben-Ari pointed out, CS is not "...open to social negotiation" (Ben-Ari, 2001). A student cannot argue their stance on the belief of CS because it has been well-established and proven in the world. Consequently, there is a need to expose these K-12 students to the computer environment; that is what CS thrives in. Otherwise, these K-12 students are not learning CS correctly.

1.5 How Computer Science Should Be Taught

There is not one way CS should be taught. The previous sections have identified the problems with how CS is introduced and how it is currently being learned, but, just like CS, there is not one solution to a problem. There are a multitude of teaching methods to better shape the perceptions of CS with K-12 students and have them be immersed in the learning process; therefore, this section will delve into specific case studies of those teaching styles and identify the key points that should be adopted. From digital gaming to pair programming to adaptive learning environments, each of these studies cover a variety of hands-on and unconventional practices educators have used to get more K-12 and college freshmen students involved with CS.

1.5.1 Implications of Gamification

According to Hamari, Koivisto, & Sarsa (2014), *gamification* is "...the process of enhancing services with (motivational) affordances in order to invoke gameful experiences and further behavioral outcomes" (Hamari, Koivisto, & Sarsa, 2014). In their analysis of this process, they have consolidated evidence that gamifying something influences the participant psychologically in a way that keeps them engaged and unfatigued. Unlike *Scratch* and other drag-and-drop tools that are explicitly demonstrating CS through a user-friendly interface, gamification involves masking the intent of CS through a game-like intermediary. The biggest difference is both forms' processes of progression. Drag-and-drop tools motivate through accessibility and ease-of-use while gamification motivates through mystery and prestige-- *what will the next level be like? Can I get the highest score?* The gamification literature study has concluded that gamifying does provide overall positive effects albeit with some caveats like making sure the context being gamified is strong enough for it (Hamari et al., 2014).

Papastergiou (2007) put this gamification theory to the test by developing a video game about system memory management using a game development tool called

GameMaker. As the students progressed through chapters of a memory management unit, they also progressed through stages of the game with increasing levels of difficulty. She also exposed the same material to another class non-digitally and compared results of pre and post tests. Ultimately, she discovered that digital-based game learning was both more effective in reinforcing students' knowledge with CS concepts and motivational in capturing their interest. She also argues that this type of gamified learning beats traditional forms of instruction because of the continued time of engagement and enjoyment by students (Prensky & Prensky, 2007).

Gamification is a significant way younger children can get excited about CS in a non-intimidating environment. Instead of dealing with the information directly through a teacher, they can interact with an application that feeds this information at a speed that fits with their learning pace. Additionally, subconsciously teaching advanced CS topics through a game-like experience allows students to retain more information and feel prepared when tackling more advanced CS topics (Hamari et al., 2014). However, there are challenges with this type of approach. One of the biggest concerns Papastergiou has with game-like learning is a student's inclination to compare these gamification techniques to existing ones in other mediums. For instance, good modern video games have extraordinary graphics, alluring stories, and riveting gameplay and educational games are not at that point to compete. Consequently, students may be disinterested with playing a relatively sub-par game and be distracted with the game-like elements rather than enhanced by them. Even though gamification of CS education is not completely refined, it is a step in the right direction for introducing CS in an enticing manner.

1.5.2 Active Teaching Methodology

When students sit at a desk and listen to an instructor lecturing, they are passively acquiring information. While this type of teaching method has been employed for generations, the literature presented here exposes the drawbacks with teaching CS in this way. Instead, teachers should put students in front of the steering wheel and driving their own accrual of CS knowledge. By being active, the students oversee their own understanding and cannot blame an external force for restricting their access of information.

Much of this philosophy stems from Ben-Ari's ideas of constructivism within CS education. When teachers give students a conception of CS, they are being placed in a world they constructed. However, Ben-Ari argues that CS cannot otherwise be constructed without some guidance or first steps because a student begins with no effective model for a computer and there is undeniable truth to how a computer is designed and built that cannot be argued with. He suggests that if teachers hand students nuggets of CS information, they will start to form their own edifice of CS (Ben-Ari, 2001). Additionally, misconceptions are exposed almost immediately with CS through the process of debugging. As a result, CS is extremely feasible for this approach of active learning.

Freeman, Eddy, etc. (2014) conducted a study where they meta-analyzed 225 studies that reported on their performance of STEM (science, technology, engineering, and mathematics) examinations based on passive and active learning environments. They discovered that examination performance increased by under half a standard deviation and “teaching-by-telling” increases failure rates by over fifty-five percent. Ultimately, they concluded that a constructivist approach of “ask, don’t tell” leads to stronger student ability in all STEM disciplines like CS and of all class sizes. They encourage more classes revolve their design around students problem-solving in groups, conducting interactive tutorial workshops, and gauging the progress of a class through peer-to-peer guidance and response (Freeman et al., 2014).

One of the greatest ways teachers can teach CS actively is by encouraging more socialization. As discussed previously, pair programming is becoming a rising new collaborative technique in the software development world that has increased knowledge retention by over 25% in CS classrooms (Williams et al., 2002). Using CS Unplugged activities also engages multiple students in a cooperative manner to learn CS concepts. The trouble that comes with this type of approach, however, is helping the teacher enforce this learning methodology in a nonchaotic and practical way. Without guidance, this type of constructivist approach to learning CS will not completely work.

1.5.3 A Guide for Teaching Teachers

Guided or exogenous constructivism can fall apart when it is being helmed by the wrong hands. Ben-Ari points out several key traits a teacher needs to have when teaching CS through a constructivist philosophy (Ben-Ari, 2001). First, he wants teachers to explicitly teach computer architecture like the CPU, HDDR, RAM, etc. Additionally, he warns to not introduce abstraction immediately because students do not have the initial framework for grasping underlying object-oriented principles. Above all, teachers should be receptive to students’ different constructions of CS and not punish conformity. Lastly, they should approach CS instruction minimalistically and inject as many errors as possible so students can start learning from follies rather than successes. As discussed before, students need to build their own tools and not rely on another person’s ways of solving a problem (Ben-Ari, 2001).

Carter (2006) also suggests her own guidelines for instructors when teaching CS. She prioritizes formal CS education training for many K-12 teachers who plan to teach the subject. If the teacher does not know the material as strongly as the students, then the integrity of the knowledge will be compromised by the students. Additionally, she proposes that there needs to be a fun aspect to learning CS. Teachers should relate CS to its applications whether that be biological, architectural, or industrial and employ activities that reinforce these applications. Not everything has to be directly related to a computer (Ben-Ari, 2001).

Barr and Stephenson (2010) also explore how to prepare teachers to change the current way they teach CS. They agree with Carter that teachers need to be professionally taught and competent with CS through learning communities and peer summer summits or workshops. Furthermore, they encourage the school administration to provide resources for teachers like models and simulations, activities, and websites for students to independently work on. Both Barr and Stephenson and O’Hara and Kay (2003) advocate for the acquisition of open-source tools and software like GNU, BSD, and Ubuntu for cheap alternatives to otherwise expensive CS learning tools (Barr & Stephenson, 2011), (O'Hara & Kay, 2003).

CS Unplugged (Bell et al., 2009) is another way teachers can expose students through interactive social activities. The makers of CS Unplugged designed this learning process around the idea of teachers being able to see a student's face instead of the back of a computer. They believe having greater interactivity between students and teachers will encourage a greater sense of constructing a computer model. Some additional recommendations they have for teachers is to focus on demonstrating CS concepts rather than programming, make activities gender-neutral and involve engaging teamwork, and encourage Socratic style questioning and probing. The more prepared a teacher is with how to instruct students with CS, the better CS will stick with not only high school students, but all K-12 schools.

1.5.4 Adaptive Learning Environments

One of the main problems with the current way CS is being taught is how teachers are applying the traditional question-answer classroom format to an inherently problem-driven field. One alternative approach that has been widely implemented in other facets of education for years has recently started being heavily applied to CS called *adaptive learning*. A conglomeration of studies (Kerr, 2016; Seters et al., 2012; Hauger & Köck, 2007) all agree that adaptive learning is not where the student is adapting to the instruction, but where the instruction is changing based on the learning style and habits of the student. Seters et al. (2012) firmly assert that adaptive learning is beneficial because it does not discriminate against a particular set of students. No matter how sub-optimally a student is performing, an adaptive learning environment will conform to the students' abilities and present user-specific challenges at a reasonable and steady pace (Hauger & Köck, 2007).

Not only does an adaptive learning environment personalize a student's learning experience, but it also gives useful feedback to the student about their progress. With this useful feedback, Seters et al. (2012) argue students will reach the ultimate learning goal of *self-regulated learning* where a student is regulating and evaluating their learning process to achieve goals they set out. Klasnja-Milicevic et al. (2011) performed a study where they put students in both an adaptive and non-adaptive virtual learning environment. They found that students in the adaptive learning environment continuously completed more lessons successfully than students in the non-adaptive learning environment. More than two-thirds of the students in the adaptive environment agreed that in the age of huge information clusters, it was helpful to have a system that would *filter* and *sort* through the data for them. Consequently, over 60% of adaptive learning students were satisfied with their increased speed and accuracy.

When it comes to CS, there have been a variety of adaptive e-learning environments set up to take advantage of a "one server to many students" scenario of instruction. One such example is *iWeaver*, a multidisciplinary research project aimed to provide a flexible environment to the learner through "adaptive hypermedia techniques" (Wolf, 2003). *iWeaver* and other implementations of adaptive e-learning environments use the Felder-Silverman Learning Style Model (FSLSM) to concisely categorize the various learning styles of a student (Carmona, Castillo, & Millán, (2007); Wolf, 2003; Klasnja-Milicevic et al., 2011). In summary, they divide the student's (1) processing, (2) perception, (3) input, and (4) understanding of material into a (1) active vs reflective, (2) sensing vs intuitive, (3) visual vs verbal, and (4) sequential vs global relationship, respectively. Additionally, they take the format of the multimedia combined with the student's rating of that media into account as well. In the end, the goal of the adaptive learning environment is to take the learning style and multimedia format as inputs and then output the probability that the learning object is appropriate for the user (Carmona et al., 2007). Essentially, the higher the probability, the more likely that learning object will be given to that student. In the case of *iWeaver*, hyperlinks to resources were disabled to prevent unprepared users from progressing too rapidly through the program (Wolf, 2003).

Let us assess how an adaptive learning environment will react to the following high-level example. For instance, a student selects, twice in a row, learning materials that have an extensive number of images inside. As a result, their probability of selecting an image the next time will be higher than a text-filled document. The learning environment will make a judgement that there is a high chance the student is a visual learner who struggles with reading extensive amounts of text. When they finish looking at the image, the next item that will be suggested to the student will involve an image. However, if the student rates the previous image poorly when asked, the probability of an image appearing slightly decreases because the system weighs the fact that the student chose it beforehand more than his low rating score of the image. The driving factor behind how the learning environment *knows* is through a machine learning algorithm known as a Bayesian network (Carmona, Castillo, & Millán, 2008). A similar approach was taken when designing *System Dot*. The architecture of System Dot's adaptive system will be discussed later.

1.6 Summary

In order to overcome the deficit of programmers in the American workforce, schools need to expose more students to CS in K-12 schools in the *right* way. Prospective college students not only have a negative perception of CS, they do not even know what the field entails. Our current educational system has hidden CS behind a door only a select number of white male students have the privilege of opening for no reason other than a lack of knowing *how* to teach CS. As a result, they tend to associate CS with mathematics and teach it in that way. Unfortunately, mathematical teaching principles like "plug-and-chug" do not nicely transfer to CS. Perhaps the reason why most teachers are CS-deficient is because most CS college graduates tend to pursue more fruitful careers in the software industry than face an uphill battle in a high school classroom.

Nonetheless, most of the problem rests with the lack of proper combination of computer concepts and programming. Whether it becomes the syntax-less drag-and-drop applications like *Scratch* or the syntax-ful curriculums like the College Board's AP Computer Science Exam, there has yet to be a perfect middle ground for teaching CS.

1.7 The Relevance and Rationale Behind *System Dot*

That is where *System Dot* comes in. *System Dot* aims to find that perfect harmony between computer architecture concepts and programming by placing the player literally in a computer. Not only do players explore the basics of computer architecture in a fun and engaging way, but they also observe the underlying code governing the world they inhabit. For example, seeing a *while* loop on a blank white canvas in an integrated development environment conveys nothing about its capabilities. However, running into a factory crushing machine that won't stop smashing victims below it without altering its internal *while* loop provides a visual manifestation of what a loop actually does. *CS Unplugged* conveys the importance of visualizing how a computer fundamentally works, and what better way to do so than through a virtual world.

System Dot does not pioneer the virtual world space when it comes to education in computer science. There has been an influx of software the past five years that harnesses the procedural way of thinking when it comes to programming such as *Scratch*[1] or *The Foos*[2]. However, they use a drag-and-drop interface for user execution of actions and behaviors. As discussed beforehand, these drag-and-drop interfaces do more harm than good to a student's construction of the computer science field. *System Dot* takes that into mind by exposing the player to actual code. A virtual world allows the player to be immersed in an environment brimming with mystery and adventure, but then have

[1] https://scratch.mit.edu/
[2] http://thefoos.com/

computer science principles integrated seamlessly in the surroundings (as illustrated with the factory crusher machine example above).

Virtual worlds not only offer the tools for students to get immersed in a learning environment, but they also let students set their own pace when learning. For instance, a user cannot access the level filled with conditional statements until they beat the zone testing for data types and objects. Like math, future programming principles rely on the comprehension of past programming concepts and a virtual world prohibits users from progressing too rapidly through significant topics. Furthermore, *System Dot* also assesses the player's learning style in an effort to adapt to their play style in the future through personalized feedback. The construction of the virtual world's learning style classification system will be described extensively later, but in short, a machine learning algorithm takes in player heuristics such as number of syntax errors, objects "hacked", and keystrokes and appropriately classifies the player on the Felder-Silverman Learning Style Model through a dynamic Bayesian network.

As many education researchers have stressed, teachers need to give students a perfect construction of what CS entails. If teachers avoid programming and jump right into computer architecture and binary arithmetic, they are not directly exposing the student to its application. However, if teachers only show a student programming, they may get lost and give up when they are introduced with the other facets of CS. *System Dot* sets out to be the perfect middle ground--it exposes students to both programming principles and computer architecture. Once the student reaches the end of *System Dot*, they should have an appropriate grasp of CS and not be surprised when they reach college or the professional industry.

1.7.1 Goals of the Thesis Study

System Dot was started as an undergraduate Honor’s thesis project by Grant West and me involving only the first level and boss as a proof of concept (Kury & West, 2016). We had set out to demonstrate that thrusting a student into a virtual world replicated a problem-driven teaching style that would reap better retention and learning results than a traditional teaching style. User testing outcomes of the project were positive-- most students reported enjoying the game immensely and wanting to play a completed version...even though they had minimal programming experience or interest in video games.

This current thesis greatly expands on the scope and aims of the earlier project by completing the game with four levels and a computer-centric, engaging plot. More importantly this thesis, steps were taken to implement an adaptive component to the game by constructing a way for the virtual world to classify how a player learns throughout their play session. Therefore, another goal of this thesis study is to walk through the implementation and user testing of this classification system and provide an exploratory analysis of the data gathered from users of the updated game focusing on its potential use with an adaptive feedback component in the future, along with initial qualitative feedback from users on the game itself.

2. VIRTUAL WORLD DESIGN AND FLOW

In order to demonstrate how the virtual world implements the learning theories and methodologies discussed in the previous sections, most of this design section replicates and/or updates material from the earlier Honor's thesis by Grant West and me. An extension has been added detailing how later levels and progression continue to conform to this implementation method (Kury & West, 2016).

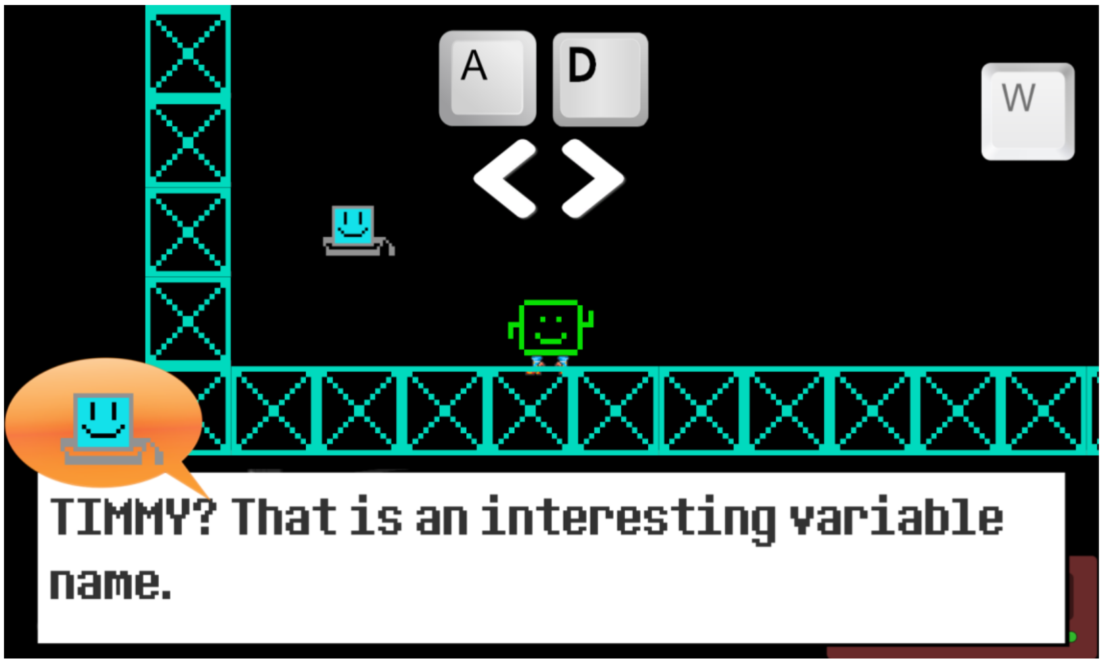
2.1 A Problem-Driven Approach

System Dot deals with four programming areas-- (1) output systems and basic syntax; (2) data types, objects, and boolean logic; (3) conditional statements; (4) and loops. Consequently, the virtual world is divided into four levels with each level dealing with one of the major programming topics. Each of these levels are further divided into four segments that facilitate a Socratic-like learning environment. These segments include (1) a tutorial, (2) a traversal of a fully functioning environment, (3) the introduction of faults, and (4) an assessment. This type of learning environment is problem-driven in nature-- the player is exposed to basic knowledge of a topic and then is thrust into a problem with minimal or no prior instruction or guidance. Depending on how they perform, the virtual world will assist accordingly, but the problem-driven approach still remains the focus. In order to understand this learning flow in the game, let us analyze the first level dealing with output systems and basic syntax.

2.2 Tutorial

The player enters the world with a blue monitor hovering right beside them (see Figure 2.1). The monitor introduces itself as **IntelliSense**, or IS for short. Thrusting the player right into the action of the game allows the game to approach the player with a constructivist frame of mind. As discussed beforehand, *constructivism* is the theory of learning where people form their own understanding of a world based on their experiences and interactions with the objects around them (Hein, 2016). In *System Dot*, each level slowly introduces the player to new concepts and ideas through never-before-seen enemies and programming syntax. Through this guided constructivist approach, the virtual world takes advantage of the theory of cognitive apprenticeship to help the player or learner construct their own understanding of the world around them with the guidance of IntelliSense. The theory of *cognitive apprenticeship* says that people tend to learn from one another through techniques like observation and coaching [Collins et al]. IntelliSense primarily employs the *exploration* approach to apprenticeship, which is the approach that gives players room to problem solve independently while mentioning strategies of how to approach various problems (Collins et al., n.d.). Despite IntelliSense's condescending tone, his overall demeanor can be related to a friend who is annoyed by the player's presence. It is through this annoyance that players will naturally strive to seek his direction throughout the game.

In order to allow the player to feel like they are this green character on the screen, IntelliSense prompts for their name. This name will be saved for future story elements and will help the player feel like their playthrough is a unique experience.

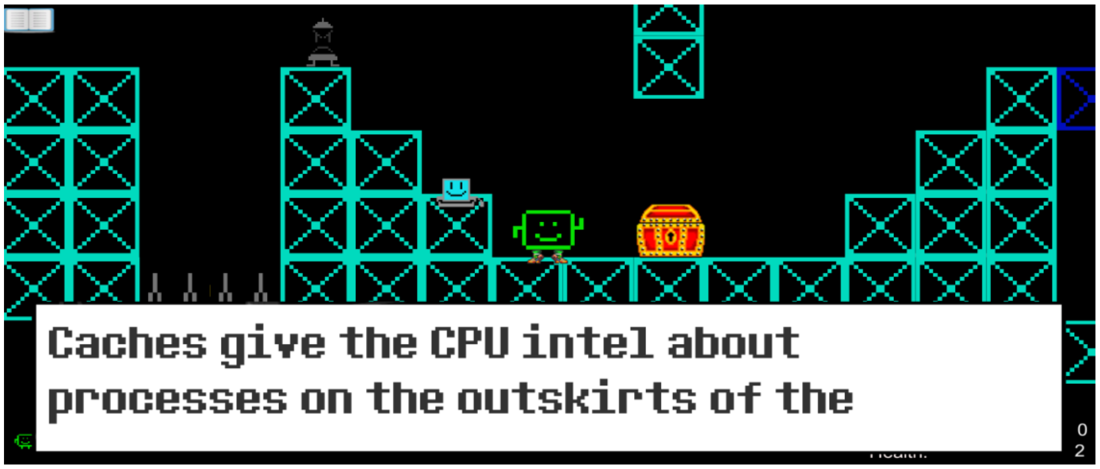


TIMMY? That is an interesting variable name.

Figure 2.1: Interaction between Player and IntelliSense

Every short tutorial section will progress the plot of the game, which is the escape of the player from a computer in which they are trapped. Situating players inside a computer gives ample opportunities to not only briefly explore computer architecture, but also puts the player in the learning context of computer science. For instance, IntelliSense accuses the player of being a virus because it cannot identify them in the system (akin to how a virus acts in a real computer). After the player provides identification and verifies to IntelliSense that they are not a virus, the objective in the first level is to escort IntelliSense to the CPU. Later in the level, players will encounter “caches” that contain bits of information that “give the CPU intel about processes” in the system (see Figure 5). This is exactly how real caches work in computer architecture and having a visual manifestation of this real-world application can help solidify that knowledge. Additionally, the idea that caches resemble chests in-game allows players to associate them with treasure or goods and feel like they are rewarded for finding them. *System Dot* already addresses the problem of current computer science education by exposing the player to both coding and computer architecture. As a result, players will construct a greater understanding of what the computer science field involves.

Figure 2.2: IntelliSense Providing Information about Caches



Caches give the CPU intel about
processes on the outskirts of the

Lastly, the short tutorial segment of a level provides a brief moment of exogenous constructivism with direct instruction by IntelliSense. *Exogenous constructivism* is a subset of general constructivism distinguished by David Moshman (1982) that specifies that players construct the world around them directly from the environment and objects themselves. In other words, players' interactions with objects clarify certain features in the learning environment. In *System Dot*, whenever the player confronts an unknown/new entity, IntelliSense steps in to assess the situation through his condescending humor. In one instance of the first level, the player confronts a “black VBot”, an enemy they have never encountered before. IntelliSense immediately tells the player to “hack it”. Through the player’s interaction with the “black VBot” object, they have further constructed a perception about the virtual world where all abnormal future objects they will see can be hacked.

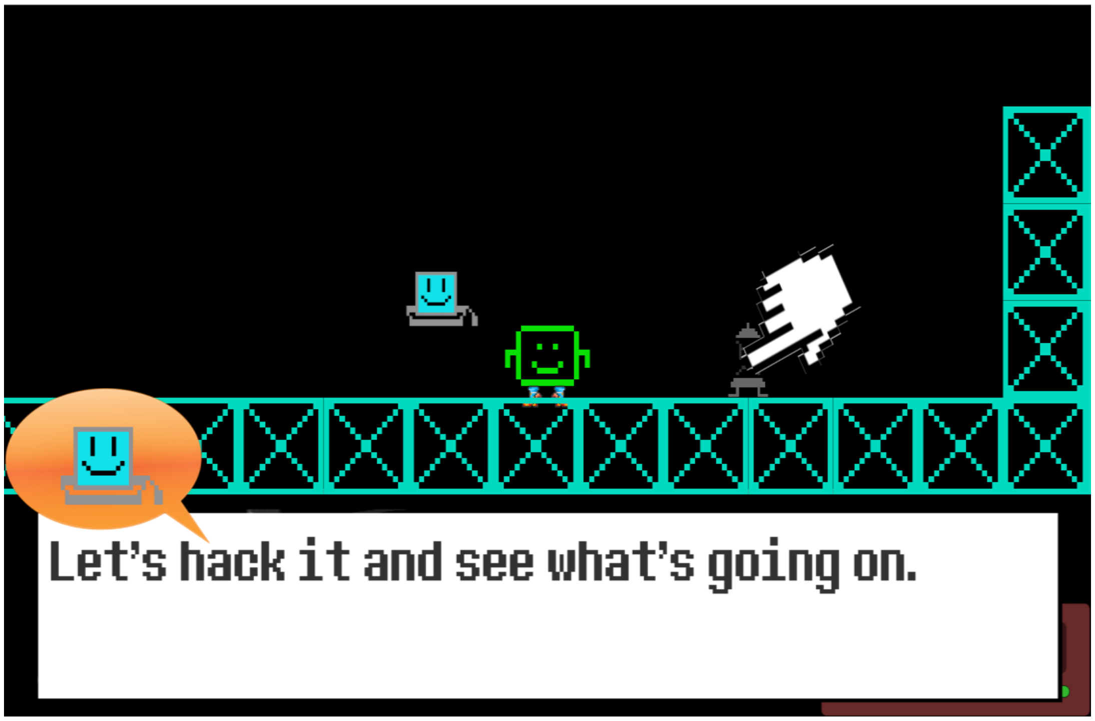
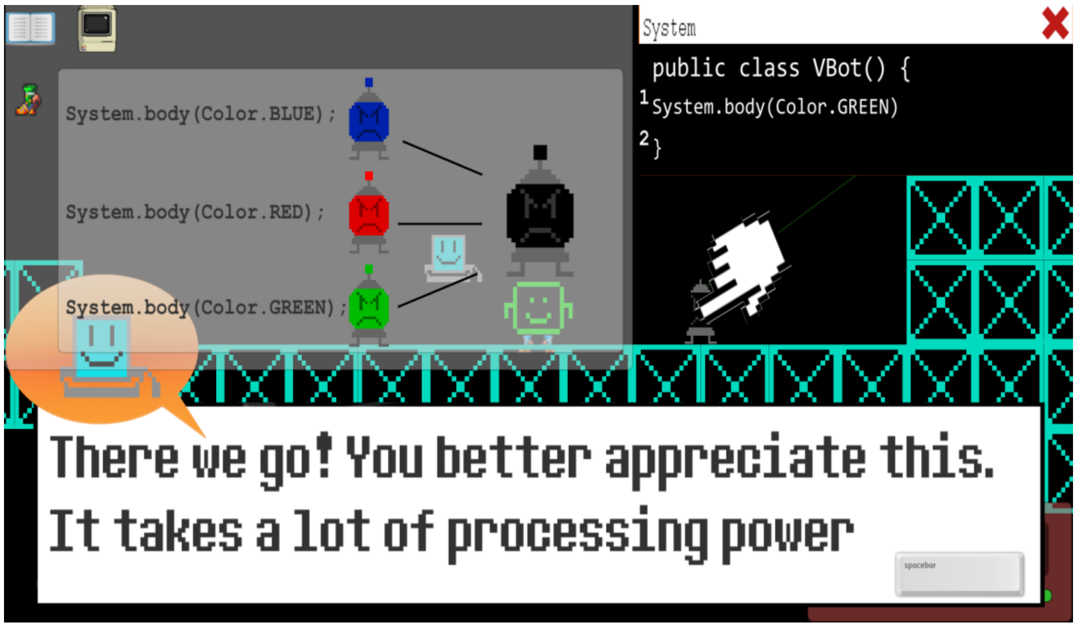

Let’s hack it and see what’s going on.

Figure 2.3: IntelliSense Teaching the Player to Hack

Once the player clicks on the object, IntelliSense introduces the “API”, a glossary of code that will assist players with syntax. Now, it becomes as simple as finding the difference between the black VBot’s code and the correct API code. Once the player solves the problem, IntelliSense will congratulate them and verbally affirm their actions. A series of these types of short tutorials will allow the player to smoothly transition to a fully functioning environment.



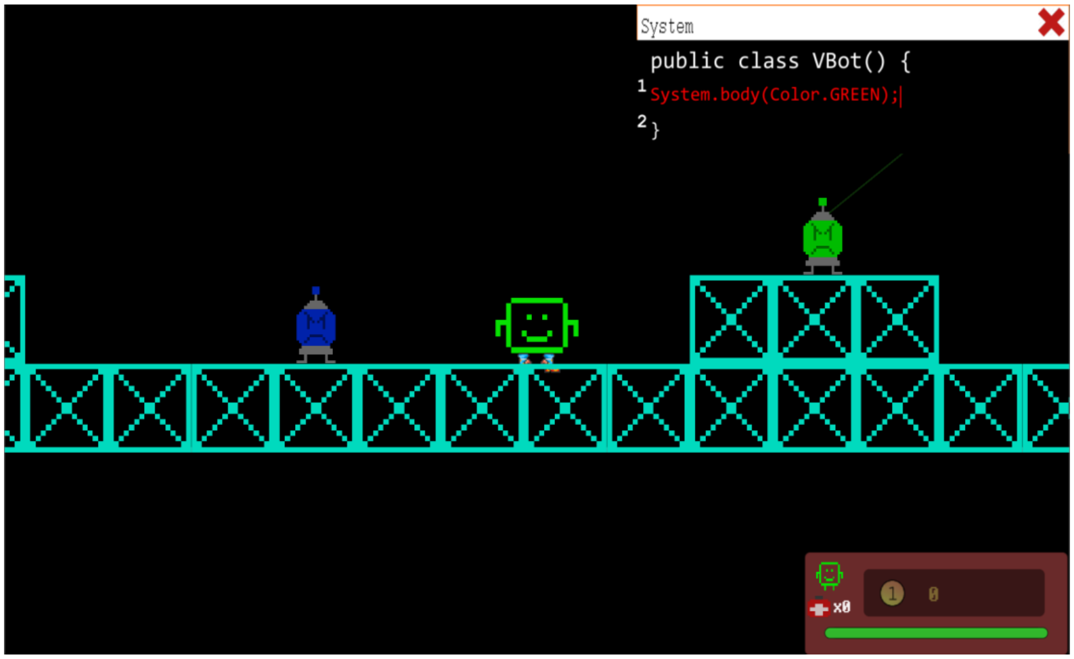
```
System.body(Color.BLUE);
System.body(Color.RED);
System.body(Color.GREEN);
```

```
public class VBot() {
System.body(Color.GREEN)
}
```

There we go! You better appreciate this. It takes a lot of processing power

Figure 2.4: A Look at the In-Game API

2.3 Traversal of Fully Functioning Environment



```
public class VBot() {
System.body(Color.GREEN);
}
```

Figure 2.5: Observing objects with legacy code in a fully functioning environment

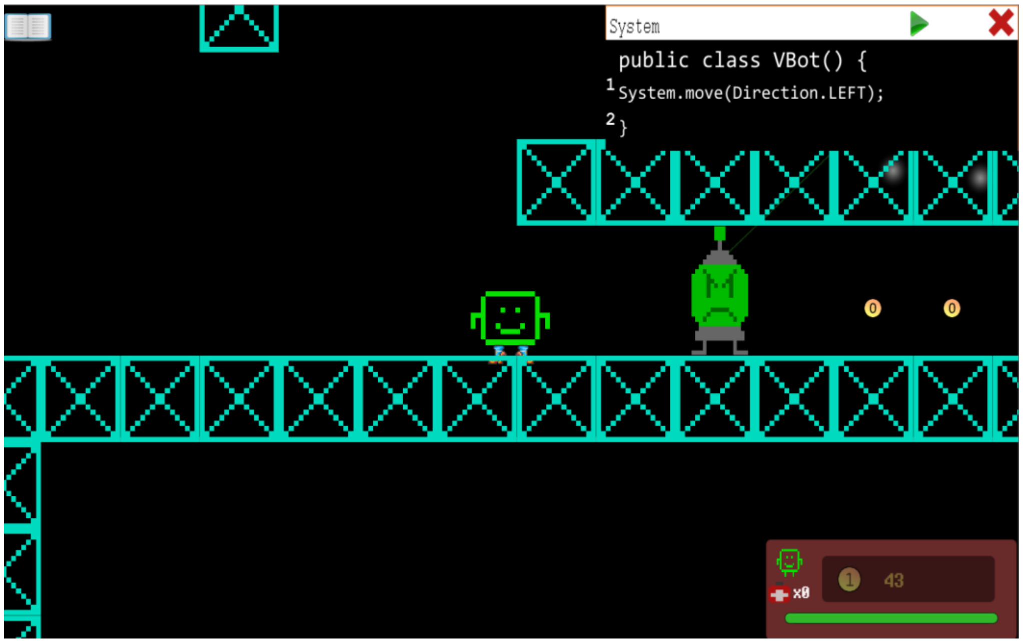
After the tutorial, the player views a fully functioning environment-- objects have correct syntax and appropriate behavior in the world. In these segments, players can either (1) safely explore their surroundings, hacking objects out of curiosity and observing how the code works or (2) battle/exploit objects to solve puzzles and progress through the level without pausing to hack code, but doing it because they want to. Having this chance to learn about correct programming principles by observing them in analogous scenarios allows the player to be situated in a computational thinking environment.

For instance, in the first level, immediately after the tutorial, the player can progress without ever hacking code, but seeing the code that they learned in the tutorial being put to action. After IntelliSense has the user hack the first enemy's code and view why it is blue, the player will then encounter other enemies of color (green and red) without going through a similar tutorial. The virtual world applies the knowledge the player has been exposed to from before with IntelliSense and has the player directly interact with that knowledge. On top of that, "termination boots" have also been introduced to the player, which is a combat mechanic where the player can step on objects of equal color to the boots and destroy them. The boots can be toggled between red, blue, and green by pressing the Q key. The color scheme was chosen because it is reflective of the standard color model in most computers (RGB). For instance, when the player encounters the blue VBot, IntelliSense mentions the functionality of the boots and how it can kill enemies of equal color. The player can further put this to action by defeating red and green VBots. The introduction of "legacy code", which are red snippets of code in a terminal window of an enemy that cannot be modified by the player, allows the virtual world to display exemplary pieces of code that the player can refer to in future puzzle scenarios.

2.4 Introduction of Faults

In the case of introducing faults, this segment expands the player’s knowledge about what they learned in the tutorial and fully-functioning traversal segments by applying it to similar but new scenarios. Unlike the tutorial section, this part of the level will have an endogenous constructivist approach, which is the opposite of exogenous constructivism. In *endogenous constructivism*, people construct the world around them from within rather than from external objects and environment (Wiebell, 2011). In these moments of the *System Dot* level, the player will not be guided or handheld by anyone and will have full freedom constructing their own view of the world around them.

In the first level, this can be directly seen with the narrow tunnel proceeding the moving platform tutorial section. In the area before this narrow hallway, the player is introduced to the code that allows an object to move left or right. Once they pass this tutorial by standing on moving platforms to bypass pits of spikes, they will be confronted with an enemy blocking the path. The player cannot skip the enemy because there is no space above or below for the player to slip through. The player also cannot defeat the enemy because the narrowness of the hallway prevents the player from jumping. In all cases, the enemy acts like a boulder blocking the player from continuing their journey through the level. Additionally, IntelliSense does not appear to help the player through this puzzle.



```
public class VBot() {
System.move(Direction.LEFT);
}
```

Figure 2.6: Thinking Creatively about Solutions to Puzzles

There is a fault in the world-- the enemy prevents the player from progressing because the source code of the object does not have the correct computational code to let the player through. However, through the previous tutorial section and viewing fully functional code of moving objects, the player will be able to apply this newfound piece of code to the existing enemy blocking the path (even though it does not look like a platform that can move). Not only will this give a sense of self-confidence and accomplishment within the player, but it will also reveal that *any* object can be affected by any piece of behavioral code (all while not directly telling the player). This sort of revelation will stick with the player longer because of independent discovery rather than a forced mentioning by IntelliSense.

2.5 Assessment

In order to ensure that the player adequately understands all the elements taught in the level they just played, there will be a final assessment or test in the form of a boss battle at the end of each level. The player cannot progress through the game without vanquishing each level's boss, which will be a culmination of everything the player has learned in that level.

Let us explore all the gameplay mechanics the player has learned in the first level and then see how these mechanics are reinforced and assessed in the first level's boss. In order of progressing the first level, the player learns:

A. Movement (including jumping and double-jumping)

B. The purpose of termination boots (terminating objects of equal color)

C. How to hack an object by clicking on it

D. How to type code into the terminal code and execute it by clicking the green debug button

E. How to change non-colored objects into colored objects using *System.body*

F. How to move platforms using *System.move*

The first boss in the first level, "Binapede", is designed around the Atari game *Centipede*. A huge snake-like creature will start at the top of the screen and slowly make its way downward to strike the player. Each component of its body looks like the spherical VBot; it is hackable and has a color (red, blue, or green) associated with it. As each component of its body slowly makes its way downward towards the player, the head of the Binapede shoots spikes at the player that they will need to dodge. Additionally, there are spiky moving platform objects along the sides of the walls for the player to take advantage of to make the fight easier; they can hack them to pierce each segment of the boss. If the player does decide to wait for the body parts to reach them, they can jump on them with a specific boot color to terminate them like they have with VBots in the first level.

As each part of the boss's body gets killed, its overall health goes down (shown in the bottom-left corner of the screen) and more pieces of its body appears at the top of the screen again to slowly make its way towards the player. The closer the Binapede is towards death, the faster it moves towards the player, the more spikes Binapede's head will shoot out, and the more ambiguous its body parts will be. What ambiguity means in this case is that some body parts will be the color gray and it is the responsibility of the player to hack and fix that.

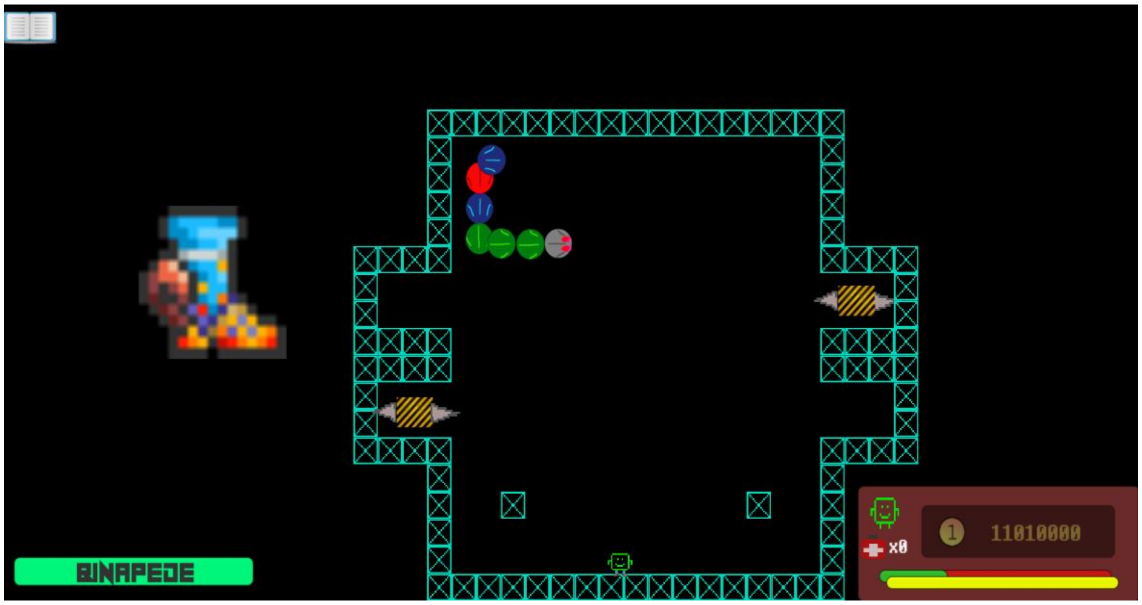


Figure 2.7: Binapede Boss Fight

Let us delve into how each of the topics the players learned in the level beforehand are assessed in this boss battle:

A. Movement is incredibly important when facing Binapede. In order to dodge his spike attacks, the player needs to press “A” or “D” to get under the floating blocks. When the body parts reach the bottom of the screen and start traversing the first floor, the player needs to press “W” once or twice to get up on a floating platform to dodge it. Quite frankly, if the player does not know how to move their character, they should not have reached this point to begin with.

B. Due to how far away the player is from the camera, the color of their boot is represented as a huge boot icon on the left-hand side of the screen. When the player sees the circular body pieces of Binapede, the intention is that the player will translate this entity as a terminatable object with their corresponding boot color. There is no other way for the player to defeat the body parts of Binapede without jumping on them with equal boot color when they reach the bottom floor of the arena.

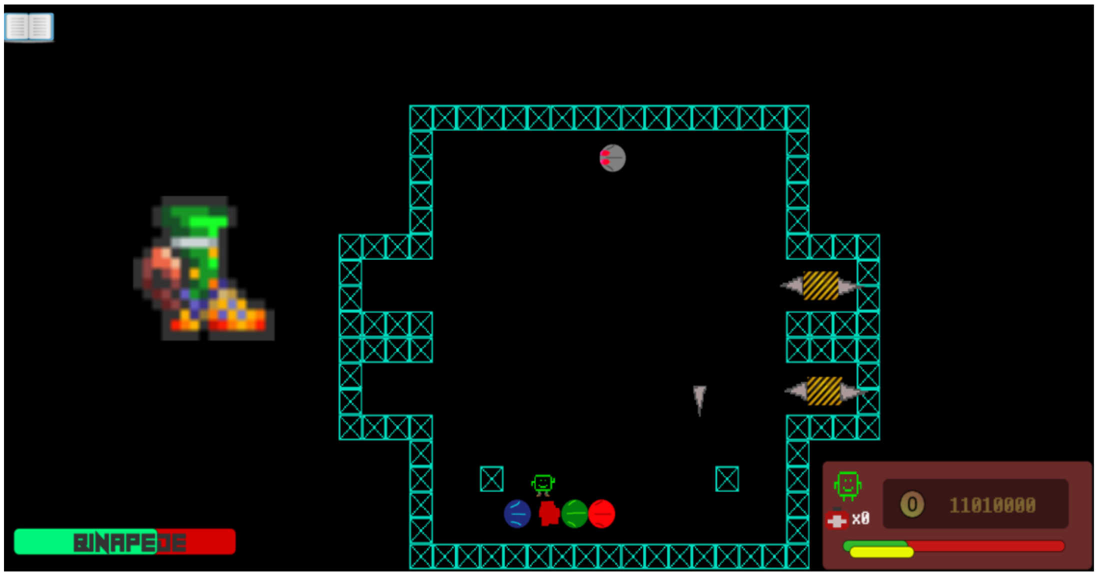
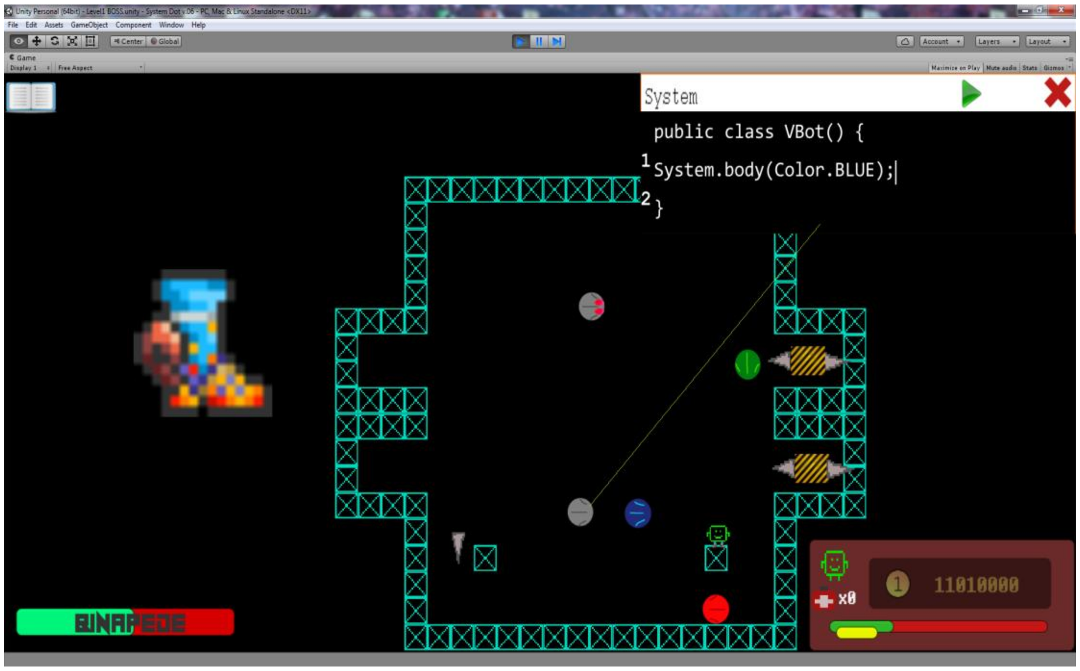


Figure 2.8: Vanquishing Binapede through Jumping

C. Like almost every object in *System Dot*, the player can click on any object with their mouse and see their internal source code. Likewise, when facing Binapede, the player can click on any component of the boss's body and see the code driving it. For instance, clicking on a red body piece will expose *System.body(Color.RED);* to the player in a legacy code format (red text color and uneditable). To prevent the player from hacking Binapede's head, which is the main driving force behind the boss's actions, there is a legacy, red-colored comment saying *//<- KILL KILL KILL ->* to let the player know that it is unhackable and programmed to destroy. Clicking on already colored objects may not be necessary to defeat the boss, but it does give context to the player and continues to construct the computing world around them. Later on in the level though, player will need to use their knowledge of clicking in order to hack gray objects into colored ones.

D. As stated before, there are some objects that are modifiable and can be hacked by the player (i.e. gray body parts or moving platforms). Therefore, the player is expected to take their knowledge of being able to modify the terminal window and apply it to those instances.

E. Gray body parts will begin to appear when Binapede's health reaches closer to zero. Similar to how black VBots told the player that they needed to hack them in order to reveal their true color, the hope is that players will associate a discoloration with the body part as a need to hack it. Once they do, the player can type in any *System.body* command in order to expose its color. Therefore, by the time the body part reaches the player, they will have a boot color that will terminate it.



```
public class VBot() {
System.body(Color.BLUE);
}
```

Figure 2.9: The Necessity to Hack during Binapede's Boss Fight

F. Similar to seeing a discoloration of body parts, the construction zone pattern on a platform will signal the player to investigate it by clicking on it. Once they do, they can move the spiky platform left or right and time them to pierce Binapede's body parts and reduce the amount of body parts the player needs to stomp on. This action is not totally necessary to defeating the boss, but it gives players an extra bonus for remembering all the different System commands.

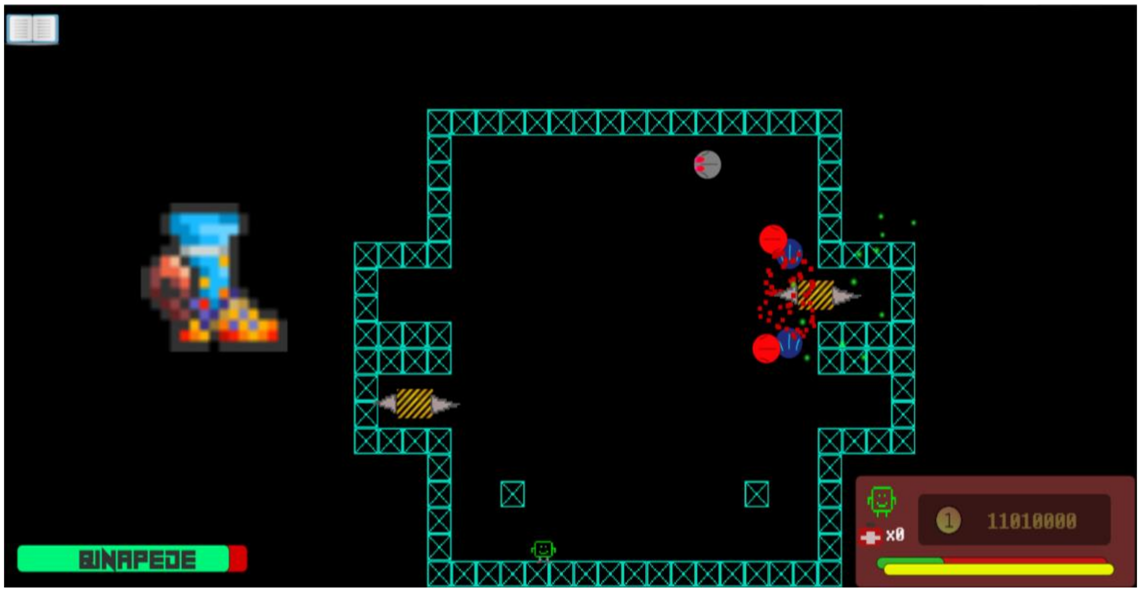


Figure 2.10: Using Movement Commands in Binapede's Boss Fight

In the end, the introduction of boss fights will be a fun way to assess the player's knowledge of programming concepts they have learned in the zone prior. The bosses have been designed in such a way where it becomes vital for the player to combine all the skills gained beforehand in order to vanquish a threatening enemy.

2.6 The CPU (Incentivisation)

Every good game needs to provide the player with some incentive to continue to play. This can be seen in modern games that offer cosmetic unlocks to further customize the player's in-game avatar or with leaderboards that display the player's score, which encourages them to continue playing in order to beat their old record. In *System Dot,* the incentive for the player to keep exploring the virtual world is through purchasing gameplay supplements and cosmetics from the *RAM (*Resource Acquisition Market).

Throughout the game the player will receive the in-game currency, *bits*, by opening caches described before or finding them floating around in the level. For the ease of the player, bits are magnetized to the player when they are nearby.

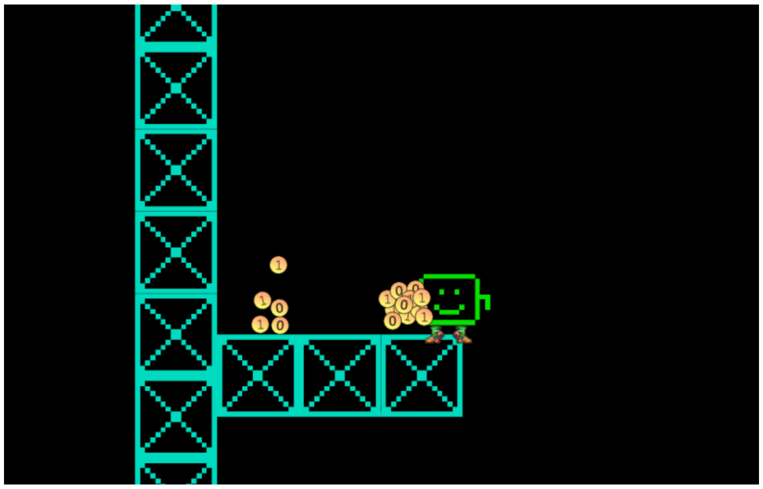


Figure 2.11: Player Absorbing Bits

Therefore, when the they open a cache, all the bits will be absorbed by the player and make a satisfying "ding" sound. This euphoric moment will encourage the player to collect bits anywhere they go, which means they will go above and beyond traversing the world to find them. Additionally, bits can be used to guide players through certain parts of the level. For instance, if there are a conglomeration of bits on a certain platform, then the player will feel obligated to collect them and jump on that platform. If they see a path of bits, they will tend to follow the path to collect them. A combination of these different level design techniques gives an endogenous constructivist approach when the player traverses the world. They are not necessarily explicitly being told where to go, but the world around them slowly nudges them in the right direction through the placement of bits.

The player's current bit wallet is displayed, in decimal, at the bottom right hand corner of the screen along with other player stats such as health and armor. If the player hovers the mouse over the decimal value of their current bits, the value is converted to and displayed in binary. When the mouse is not over the field, then it goes back to decimal. This subtle feature will help the player slowly learn how the world of computing is infused with a binary numerical format and continue to construct this perception around them. This feature will not necessarily teach players *how* to convert between binary and decimal numbers, but will inspire them to pursue avenues where they can.

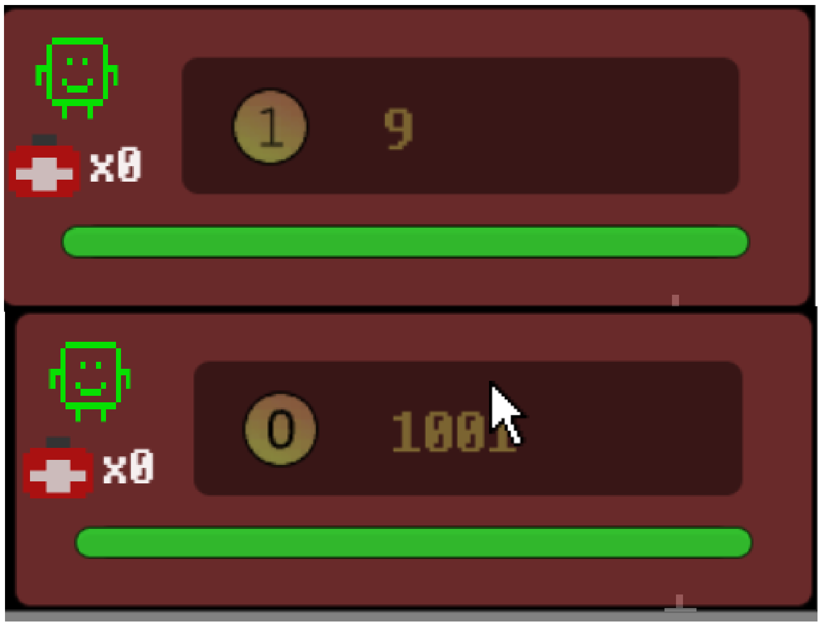
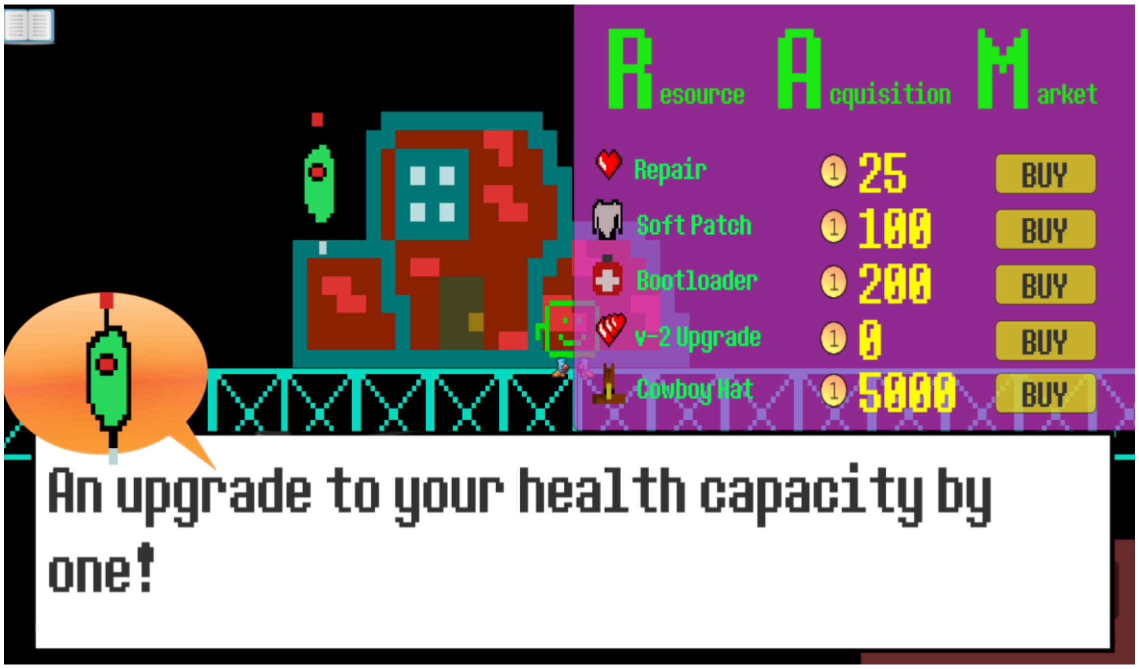


Figure 2.12: Converting from Decimal to Binary through Bits

The Resource Acquisition Market (RAM) is located in the Central Public Union (CPU) of the game. While this is not necessarily indicative of modern computer architecture, it does give some insight to the player about how the CPU and RAM work closely together. Additionally, introducing the player to terms like CPU and RAM while not precisely using them in the context of actual computing continue to construct a computing world around them. For instance, the way the player purchases items from the RAM is to give bits of memory to it (which is the in-game currency the player has collected throughout their journey). This is similar to how actual RAM works-- it relinquishes and acquires memory.



Repair	25	BUY
Soft Patch	100	BUY
Bootloader	200	BUY
v-2 Upgrade	0	BUY
Cowboy Hat	5000	BUY

An upgrade to your health capacity by one!

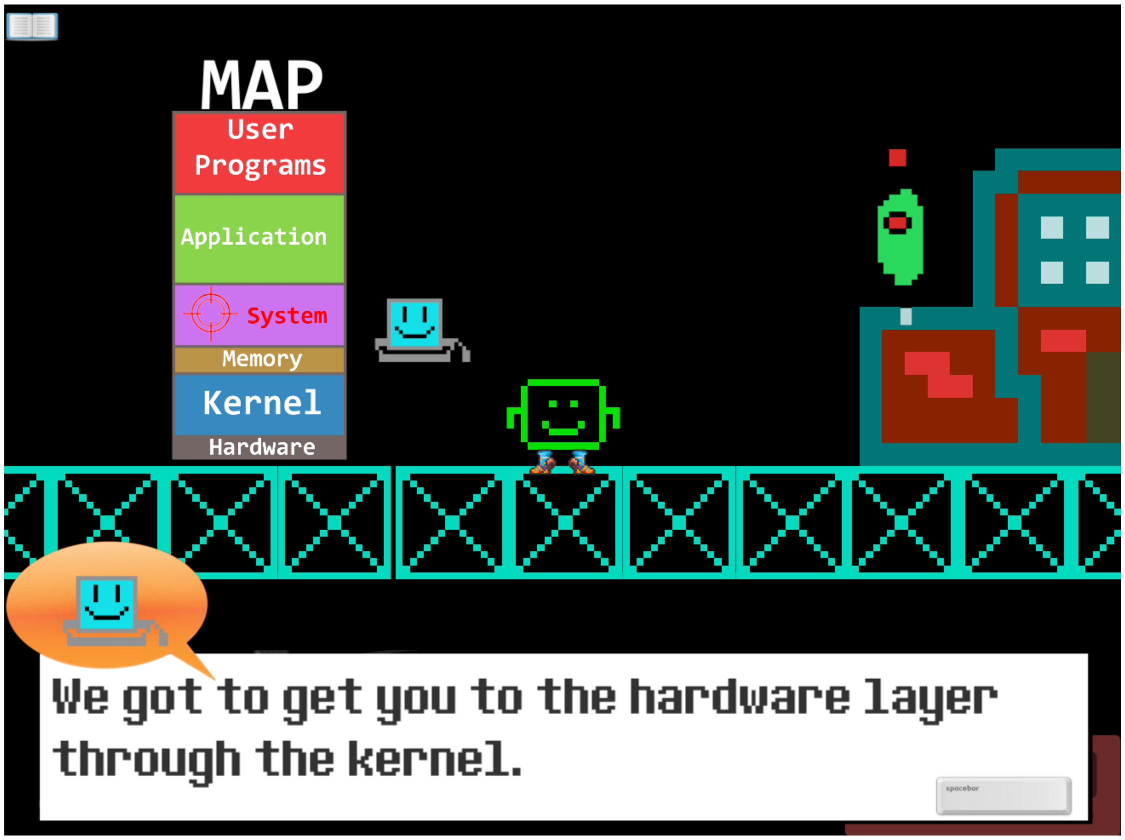
Figure 2.13: A Look at the RAM Shop

When the player visits the market, they have access to several different items in the game, including health, revival kits, armor, and cosmetics. The health and armor serve to make the player's progression through the levels easier while the cosmetic items are the key to incentivizing the player to continue playing the game. Through this mechanism we hope to keep players returning to the game to learn and have fun.

2.7 The Instructional Flow Beyond the First Level

As expressed throughout the dissection of the first level, the goal of *System Dot* is to properly convey the field of computer science by combining visual manifestations of programming concepts and the context of computer architecture and its basic functionality. The dissection of the first level and CPU from the Honor’s thesis briefly convey the design decisions taken to accomplish that goal, but let us quickly highlight aspects of other levels that continue to demonstrate this constructivist approach.

The whole objective of the player is to figure out why they are stuck in a computer and how to escape. In the first level, IntelliSense says it can help but that players need to reach the CPU first. When the player does reach the CPU, they are shown the basic hierarchy of an operating system. IntelliSense explains that they are in the system sector right now and they need to reach the kernel by taking a “Data Link” from the “Transport Layer”. In the game, the transport layer looks like a train station and the data link is a train. To leave the CPU, the player will need to board the train. Once they do, a cinematic ensues. In the cinematic, the player rides the train until a hiccup occurs and the whole “data transfer” explodes, similar to how an interruption works in an actual computer. Getting to the kernel and out of the computer could not be that easy…

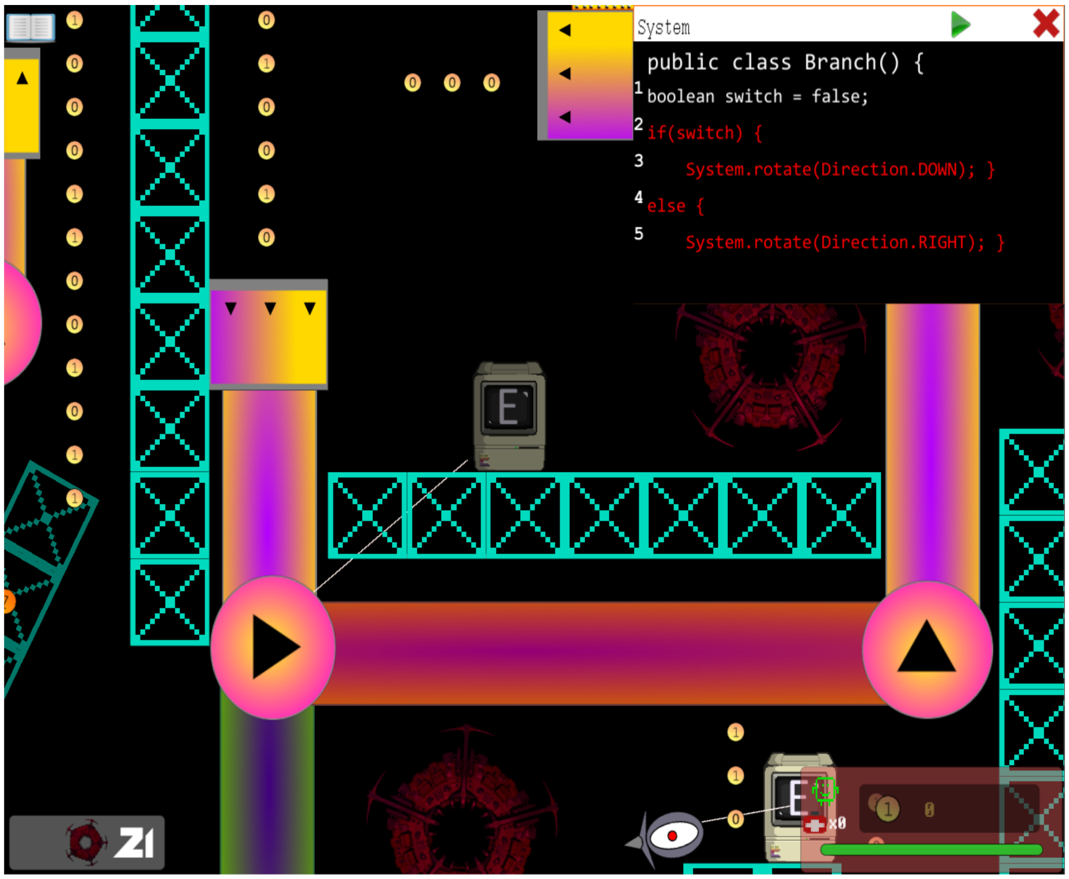


We got to get you to the hardware layer through the kernel.

Figure 2.14: The System Map

The player and IntelliSense awaken to find Auddreyss, an engineering process that appears to know what is going on. In this interaction, the player discovers that they are the owner of the computer they are trapped in and because the owner/player is not in the real world to maintain the computer, the system has been breached by a virus. Now, the objective of leaving the computer becomes as prominent as ever. Before the player can leave this level, though, Auddreyss reveals that it is also in charge of an address table (a gated enclosure) that needs its “variabulls” back (a pun on “variables”). The player’s mission is to find the variabulls scattered throughout the level and deliver them back to the address table. This is exactly how data assignment and types work in a computer-- variables are allocated onto stack memory via an address table. Once the player finds a variabull, they will introduce themselves in a quirky fashion indicative of the data type they represent (i.e. the sesquipedalian nature of the string variable or the statistics-driven double variable). They will also introduce their name and the value their holding and then display that to the user in the terminal window. Once again, visualizing the computational thinking behind the purpose for these data structures in computer science allows the player to better understand its function. For instance, integers are depicted by powering up activation units that require a certain finite amount of power; doubles are dealt with rotating actual objects-- .5 would be half a rotation while 1 would be full; strings are used to manipulate "word blocks" that obstruct the player's path; and booleans power up "gates" that open doors like an on-off switch. After completing the second level, the player is taken to the second boss named "Virus" who combines all the knowledge learned previously.

After defeating the "Virus", something strange happens. A black hole suddenly erupts from the middle of the arena and starts absorbing game objects, including the player. Unfortunately, IntelliSense gets sucked in almost immediately. This is intended to not only shock the player, but prepare them for future levels without focused guidance or help. The system has crashed. When the player gets absorbed by the black hole, the third level begins.



```
public class Branch() {
boolean switch = false;
if(switch) {
    System.rotate(Direction.DOWN); }
else {
    System.rotate(Direction.RIGHT); }
```

Figure 2.15: Conditional Statements Mapped to Pipes

The player wakes up again to find a strange “?” object hovering over them. Apparently, the system crash has thrown the player down into the recesses of the system memory known as the “nullsphere”. In order to demonstrate the computational meaning behind conditionals, the virtual world introduces the “pipe” mechanic, a method of transportation that locks the player into a certain movement pattern based on the directional arrows. The multiple paths of the pipes mimic a conditional statement; one path represents the “true branch” while another may represent the “false branch” of a conditional statement. The player is also introduced to the system distance command

(*System.distance(obj)*) that returns a numerical value denoting how far apart that object is from the one in the command's parameter. A pink line connects the two to avoid confusion with the player. None of this is explicitly defined by the floating "?"; all it wants to do is leave this area. In order to escape the "nullsphere", all the viruses in the level need to be cleansed by external methods only. These external methods involve a review of past subject matter like moving objects with *System.move( )* or rotating objects with *System.rotate( )*. At certain intervals (i.e. when the player has killed eleven viruses or if there is only one remaining), the "?" character will encourage the player to keep going, but not give them any hints or solutions to solving the level.

After defeating all the viruses, the player can ascend to the boss of the level-- "Malrus". There is no way for the player to use their termination boots to damage the boss. The only way to do so is to use the boss's attacks against it through pipes. The gimmick behind the boss's attack pattern is that it will rain down balls ridden with spikes that hold a numeric value. There are also platforms scattered throughout the arena that hold conditional statements similar to the ones encountered by the players in the pipe section. Once the number ball touches one of the platforms, its value is evaluated by the condition on the platform and will tilt either left if the condition is false or right if it is true. The ball then drops into a pipe that leads to an area right above the pipe entrance. The goal is for the player to drop this ball right on the boss, damaging it. This boss fight not only strengthens the purpose of conditional statements, but the ability to write them since some platforms will need to be filled in.

After defeating the third boss, the player is led to the final level revolving around loops. After exploring a bit, the player finds a corrupted IntelliSense incapable of speaking or moving. The goal in this level is to escort IntelliSense to charging pads that reinvigorate the process and reboot the IntelliSense program. In order to do so, the player will need to press the “forward” or “pause” button to cause the defunct guide to move to the right or stop abruptly. Meanwhile, the player will encounter enemies that have unique attributes. For instance, as illustrated before in the **Relevance and Rationale Behind *System Dot*** section, there is an enemy that continuously smashes the ground, preventing the player from progressing. After hacking the object, they notice an infinite while loop with a condition of “true”. Simply changing this condition to false (the only other option the player can make) will cause the loop to stop executing and the enemy from smashing. Existing enemies now have new attributes; they vibrantly change color every few seconds. When the player hacks the code, they will discover that this is caused by an infinite loop transitioning between colors using the new *System.wait( )* command. To demonstrate a for-loop, a new “Spawner” enemy can emit objects based on the parameter in the for loop. Its source code involves a *System.output( )* command that is repeatedly being called by the bounds of the for-loop. When the player sees the number of objects spawning based on the number in the bounds for the for-loop, they should be able to associate it to the number of times the loop executes. Once again, these are some of the ways programming concepts are infused with the gameplay style of *System Dot*.

2.8 Summary

A virtual world allows for a natural progression of programming concepts through a problem-driven learning environment. By exposing the problem first and then guiding them implicitly to the solution, the player starts to become a tool-builder instead of a tool-user. To summarize, here is a breakdown of the instructional goals and main tasks for each level:

Table 2.1: Virtual World Design Breakdown

Level Name	Instructional Goals	Main Tasks
1: Compiler Shire	- Introduce controls and combat of the game - Expose simple custom language syntax - Reveal use of the API	- Hack objects through terminal window - Use *System.body* commands to defeat enemies - Use *System.move* commands to manipulate obstacles
2: Enumeration Station	- Run through instantiation of variables - Use variables corresponding to the following data types: o Integers o Doubles o Strings o Booleans	- Rescue the four “variabulls” and return them to their address table (each modelled after a different data type) - Use *System.activate* to manipulate power lines with integer values - Use *System.rotate* to rotate objects with double values - Use *System.body* and *System.delete* to fill in and cross word blocks with strings - Use *substring* method to dissect strings into chunks to bypass obstacles - Use *System.output* to play with power gates and activate certain doors with booleans

3: Nullsphere	- Understand boolean logic and its use in conditional statements - Recognize the syntax of an if-else statement, if-elseif-else statement, and nested conditional statements - Follow the flow of a conditional statement and its purpose in computing	- Eliminate 21 viruses scattered throughout the level using other objects (level can be roamed freely) - Use *System.distance* command to evaluate conditional through in-game metrics like distance from player to an object - Ride pipes to progress through the level; branches in the pipe will follow the flow of the conditional statement
4: Precipice of Memory	- Understand the syntax of a while and for loop - Comprehend the computational logic behind the flow of a loop	- Escort a corrupt IntelliSense to charging stations littered throughout the level - Bypass objects with infinite loops (including nonstop crushing machines, enemies changing colors periodically, and objects that follow the player within a certain proximity) - Destroy “Spawners” that spawn enemies at a rate corresponding to the bounds of their for loop

3. TECHNICAL DESCRIPTION

The technical foundation of *System Dot* fosters the learning theories and design of the virtual world described beforehand. The virtual world was designed with the Unity 5.5 game engine (www.unity3d.com) and the C# programming language. Unity3D is used both commercially and in the industry as an intuitive way to make 2D and 3D virtual simulations and games. *System Dot* is a 2D game because the genre can be used to emphasize gameplay mechanics rather than visuals. Additionally, having one less plane to manipulate eases new gamers and reduces the complexity of the problems introduced. Since this thesis is an extension of my Honor's thesis, more emphasis will be placed in the integration of the learning theories and learning style classification system over specific code and game systems. For an in-depth look at how the game functions systematically, please refer to my Honor's thesis (Kury & West, 2016)

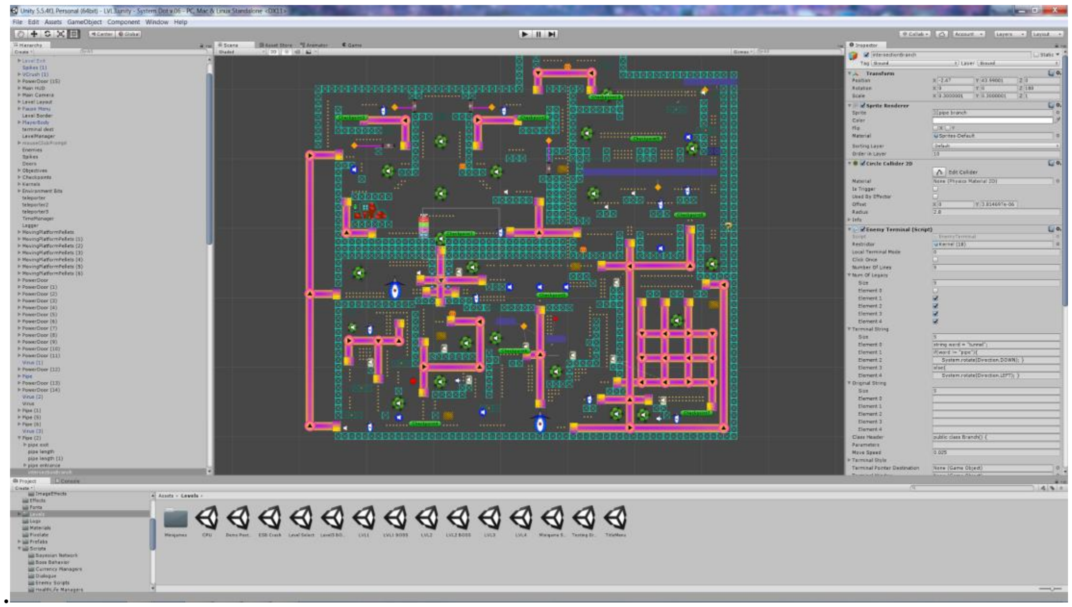


Figure 3.1: A Look at the Unity Environment

3.1 Custom Parser

The custom parser is what distinguishes *System Dot* from other virtual worlds/video games that teach programming. For instance, unlike *CodeCombat* or *CodeAcademy*, which asks the user for a specific segment of code from a list and will not execute unless that code is entered, *System Dot* gives players the freedom to write whatever they want and see their creations come to life. Instead of writing the same lines of code over and over again to reinforce programming principles in other instructional programming software, the player in *System Dot* can write the same code in a variety of ways and see how those different ways affect the behavior of the object they are modifying. Once again, this amount of liberty is what will cause players to continue playing; they have no bounds on their creativity other than the locations of the game objects in the game world.

The custom parser can handle the following programming principles:

- **Built-in commands** like *System.body(Color.BLUE);*
- **Commenting** such as *// This is a comment*
- **Data Types** like integers, doubles, booleans, and strings (i.e. *int x = 4;*)
- **Object methods** like substring and length of a string
- **Basic arithmetic**
- **Type casting** like *int y = 2; float x = y;* (x = 2.0)
- **Type checking** like *boolean x = 2;* would return an error
- **Boolean logic** like *true && false* || *false* would return *false*
- **Full conditionals** with *if(...) { … } else if (...) { … } else { … }*

- **Nested conditionals**
- **While loops**
- **For loops**
- **Error handling** like *int x = 2* would return “missing semicolon on line 1”

Consequently, the custom language governing the parser can be expressed by the following context-free grammar:

```
[1]    program          ->    stmt_list | ϵ
[2]    stmt_list        ->    stmt stmt_list | stmt
[3]    stmt             ->    assign_stmt | while_stmt | if_stmt
       stmt             ->    for_stmt | sys_stmt | comment_stmt
[4]    type_name        ->    INT | DOUBLE | BOOLEAN | STRING
[5]    assign_stmt      ->    type_name ID EQUAL numExpr SEMICOLON
       assign_stmt      ->    type_name ID EQUAL strExpr SEMICOLON
       assign_stmt      ->    type_name ID EQUAL boolExpr SEMICOLON
[6]    while_stmt       ->    WHILE condition body
[7]    if_stmt          ->    IF condition body elseif_stmt else_stmt | ϵ
[8]    elseif_stmt      ->    ELSEIF condition body elseif_stmt | ϵ
[9]    else_stmt        ->    ELSE body | ϵ
[10]   for_stmt         ->    FOR for_cond_stmt body
[11]   numExpr          ->    numTerm (PLUS | MINUS) numExpr
       numExpr          ->    numTerm
[12]   numTerm          ->    numFactor (MULT | DIV | MOD) numTerm
       numTerm          ->    numFactor
[13]   numFactor        ->    LPAREN numExpr RPAREN | NUM | REALNUM | ID
[14]   strExpr          ->    (QUOTE ID QUOTE | DOUBLEQUOTE)
       strExpr          ->    strExpr PLUS strExpr
[15]   boolExpr         ->    TRUE | FALSE | LPAREN boolExpr RPAREN
       boolExpr         ->    boolExpr (AND | OR) boolExpr
[16]   condition        ->    LPAREN (ID | condExpr) RPAREN
[17]   condExpr         ->    primary relop primary
[18]   primary          ->    ID | NUM | REALNUM | TRUE | FALSE
```

```
[19]  relop            ->    GREATER | GTEQ | LESS | LTEQ | NOTEQUAL
      relop            ->    EQUALEQUAL
[20]  body             ->    LBRACE stmt_list RBRACE
[21]  for_cond_stmt    ->    LPAREN assign_stmt SEMICOLON condExpr
[22]  sys_stmt         ->    SYSTEM DOT jump_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT body_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT open_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT close_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT move_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT check_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT output_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT wait_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT smash_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT gravity_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT activate_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT rotate_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT delete_stmt SEMICOLON
      sys_stmt         ->    SYSTEM DOT distance_stmt SEMICOLON
[23]  jump_stmt        ->    JUMP LPAREN RPAREN
[24]  body_stmt        ->    BODY LPAREN COLOR DOT BLUE RPAREN
      body_stmt        ->    BODY LPAREN COLOR DOT GREEN RPAREN
      body_stmt        ->    BODY LPAREN COLOR DOT RED RPAREN
      body_stmt        ->    BODY LPAREN COLOR DOT BLACK RPAREN
[25]  open_stmt        ->    OPEN LPAREN RPAREN
[26]  close_stmt       ->    CLOSE LPAREN RPAREN
[27]  move_stmt        ->    MOVE LPAREN DIRECTION DOT RIGHT RPAREN
      move_stmt        ->    MOVE LPAREN DIRECTION DOT LEFT RPAREN
[28]  check_stmt       ->    CHECK LPAREN ID RPAREN
[29]  output_stmt      ->    OUTPUT LPAREN ID RPAREN
[30]  wait_stmt        ->    WAIT LPAREN (NUM | REALNUM) RPAREN
[31]  comment_stmt     ->    DOUBLESLASH <characters> NEWLINE (\n)
[32]  smash_stmt       ->    SMASH LPAREN RPAREN
[33] gravity_stmt      ->    GRAVITY LPAREN (TRUE | FALSE | ID) RPAREN
[34] activate_stmt     ->    ACTIVATE LPAREN (NUM | REALNUM | ID) RPAREN
```

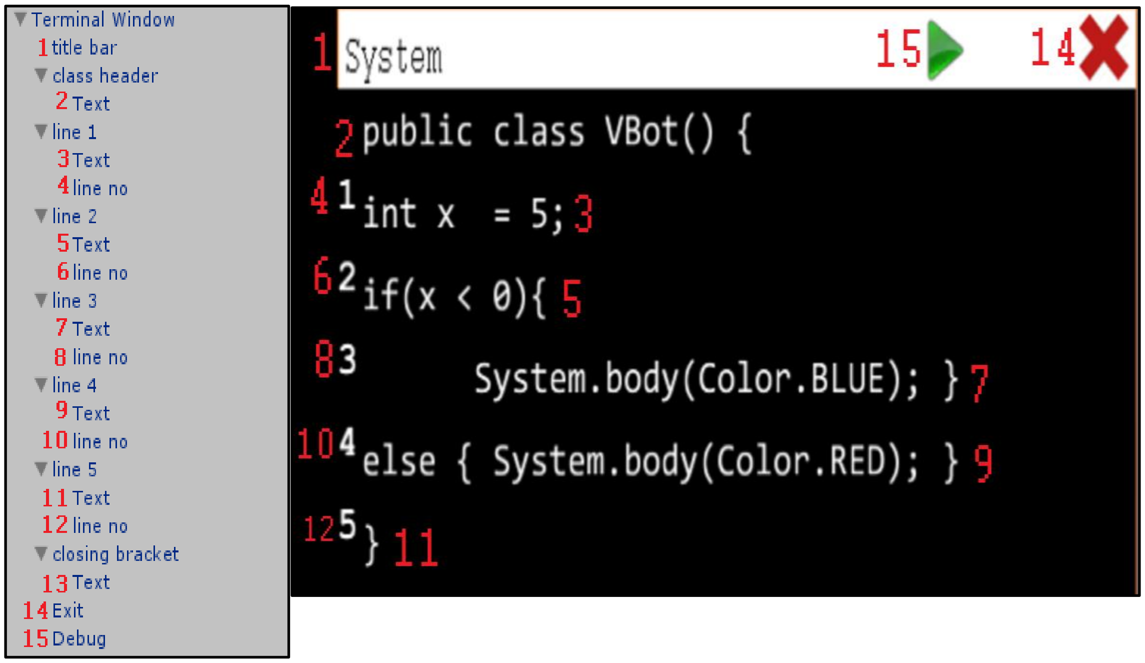
```
[35] rotate_stmt          ->    ROTATE LPAREN (NUM | REALNUM | ID) RPAREN
[36] delete_stmt          ->    DELETE LPAREN (ID | strExpr) RPAREN
[37] dist_stmt            ->   DISTANCE LPAREN ID RPAREN SEMICOLON*
[38] ncn                  ->    NUM | NUM COMMA NUM
[39] substr_method        ->   ID DOT SUBSTRING LPAREN ncn RPAREN SEMICOLON
[40]  indexOf_method      ->    ID DOT INDEXOF LPAREN ID RPAREN SEMICOLON
[41]  length_method       ->    ID DOT LENGTH LPAREN RPAREN SEMICOLON
```

In the end, the goal with the custom parser is to provide a canvas for students to flex their computational thinking skills through diverse code problems. It also allows the system to keep track of any syntactical mistakes and the errors associated with those mistakes. This data can then be leveraged for the adaptability component by giving players appropriate feedback based on the severity of their syntax follies. This will be described further in the next section.

3.2 Object Terminal Window

Depicting an accurate coding environment is essential to the learning process of CS. As discussed previously, presenting a misconception about the field does more harm than good in the long run. Consequently, the coding environment in *System Dot* looks like a *vim* or console window in order to convey the very foundation coding was originally written on. The mundanity of the coding environment stems from the belief that integrated development environments tend to handhold the programmer through the IntelliSense feature and colored code. Then, when a student in CS starts learning about lower-level environments with none of those helpful features, they take longer to adjust to the point of possible surrender. To prevent that from happening, *System Dot* holds a coding environment with minimal handholding and an emphasis on text. In the world, it is coined the “terminal window”.

From a technical perspective, the terminal window is a conglomeration of input fields, text labels, and buttons. In order to simulate a text area, custom code was written to make the text caret jump between consecutive input fields when pressing the ENTER key or shift between them when pressing the UP and DOWN arrow keys. The following image shows the Unity hierarchy of game objects responsible for building the terminal window mapped with what it actually looks like in-game:



```
System
public class VBot() {
1 int x  = 5;
2 if(x < 0){
3         System.body(Color.BLUE); }
4 else { System.body(Color.RED); }
5 }
```

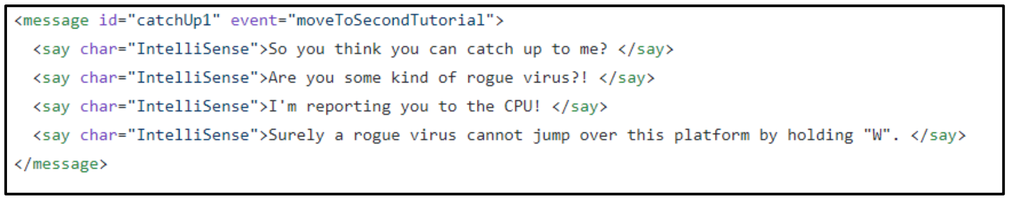
Figure 3.2: A Mapping of Unity’s Inspector with In-Game View

Essentially, the main way the player interacts with the virtual world is through this terminal window; it acts as a middleman between the code the player has written and the custom parser illustrated before. Once the player has written or observed the code, they can either execute the code through the green play “debug” button or exit the window without executing. When the player exits the terminal window through the “red X” button, a notification will appear notifying them that none of their code changes have been executed. Players can either click the debug button or use the “F5” keyboard shortcut to execute the code. This is extremely similar to how an integrated development environment functions. Even though this functionality is not faithful to how a *vim* or console window works, leeway was still given ease the player into the environment. The way the user interacts with the terminal window influences how the adaptability component of the game classifies their learning style. If the player uses the “F5” keyboard shortcut frequently, it shows that they are sensible and willing to perform from memory. More details will be discussed later in the next section.

All in all, the terminal window is a pivotal component in *System Dot* as it interfaces to player with computational coding puzzles and exposes them to how coding looks like in the real world. The main goal for its place in the game is to allow the user to construct their own perception of programming from their experiences with the terminal window and hopefully spark a flame within them to continue pursuing the CS discipline.

3.3 Dialogue XML Trees

System Dot uses dialogue to not only propel the player to the end of the level, but to subtly teach them important programming concepts during crucial moments. As described in the **Design of the Virtual World** section, the virtual world uses the learning theory of *cognitive apprenticeship* through IntelliSense. In order for IntelliSense to be a mentor for the learner, it will need to establish a rapport based on the actions of the player. That is why, technically, the dialogue system was designed with an event-driven approach in mind. In order to easily modify what is being said by IntelliSense, local XML files were used to modularize when and where dialogue would be spoken. Here is an example of what a section of dialogue would look like from the XML side.



```
<message id="catchUp1" event="moveToSecondTutorial">
  <say char="IntelliSense">So you think you can catch up to me? </say>
  <say char="IntelliSense">Are you some kind of rogue virus?! </say>
  <say char="IntelliSense">I'm reporting you to the CPU! </say>
  <say char="IntelliSense">Surely a rogue virus cannot jump over this platform by holding "W". </say>
</message>
```

Figure 3.3: Example of Dialogue XML

An object-oriented approach was taken to design how the dialogue system works for each level in the virtual world. The parent class, *Dialogue*, is in charge of parsing through the XML file and splitting up the speech into events and *sayLists* that can be performed by its child classes. A *sayList* has both a who and what attribute detailing the non-playable character who is speaking the dialogue and the actual text of the dialogue, respectively. One of these child classes is *IntelliSense*, who has a specific way of starting and finishing dialogue by zooming in and out of the player. Since IntelliSense is in more than one level with different actions, a specific instance of IntelliSense is also created. In this example, *IntelliSenseLevel2* holds the code responsible for deciding what IntelliSense says when the player performs certain actions in the second level.

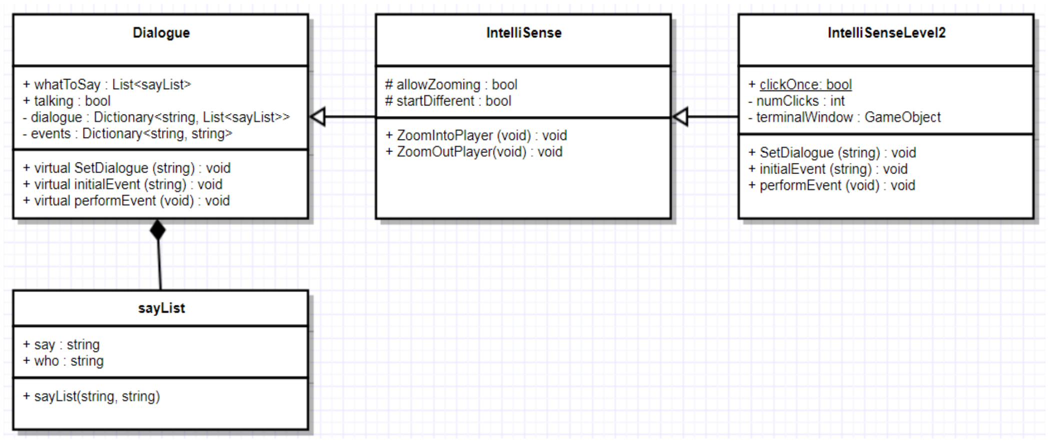

Dialogue
+ whatToSay : List<sayList>
+ talking : bool
- dialogue : Dictionary<string, List<sayList>>
- events : Dictionary<string, string>
+ virtual SetDialogue (string) : void
+ virtual initialEvent (string) : void
+ virtual performEvent (void) : void
IntelliSense
# allowZooming : bool
# startDifferent : bool
+ ZoomIntoPlayer (void) : void
+ ZoomOutPlayer(void) : void
IntelliSenseLevel2
+ clickOnce: bool
- numClicks : int
- terminalWindow : GameObject
+ SetDialogue (string) : void
+ initialEvent (string) : void
+ performEvent (void) : void
sayList
+ say : string
+ who : string
+ sayList(string, string)

Figure 3.4: Class Diagram of Dialogue System

When IntelliSense is speaking, the player freezes and all actions in the game cease to matter. A white box appears at the bottom of the screen with revealing text (similar to most role-playing video games). The text appears at a moderate and reasonable pace. If it is too slow for the player, they can progress through the dialogue at their own pace by simply hitting the spacebar before all the text appears. The speed at which the player advances through the dialogue will be recorded in the log files, which will be discussed further below.

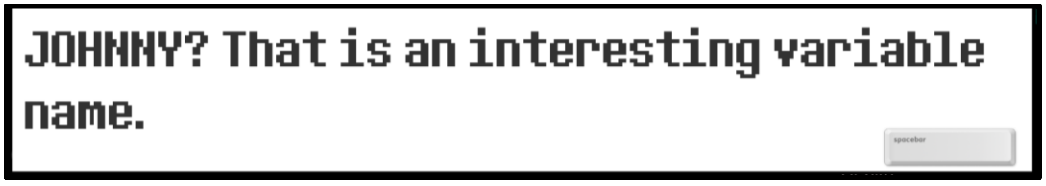
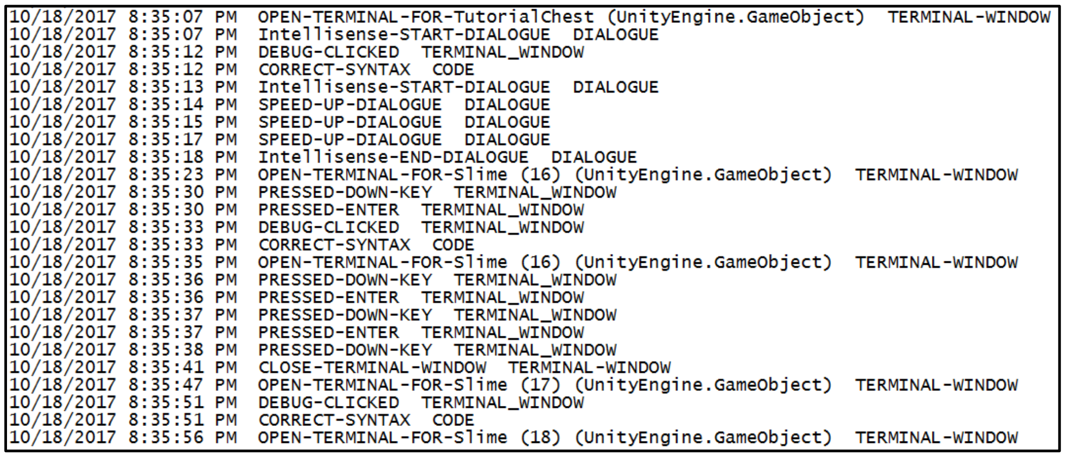

JOHNNY? That is an interesting variable name.

Figure 3.5: In-Game View of Dialogue Window

The dialogue spoken by IntelliSense is responsible for teaching the basics of control input and coding syntax in a friendly and inviting fashion. Even though the tone is inherently condescending, there is a playful and naive nature to IntelliSense's direction and assistance. By the end of the player's interaction with IntelliSense, they should be able to form a bond with this floating blue image and earnestly listen to everything it has to say. None of this would be possible if it were not for the XML parsing system producing easily modifiable dialogue statements. If a statement or line does not work, it is as simple as opening up the XML file and changing that line; no need to modify any existing logic in any of the child *Dialogue* scripts.

3.4 Data Logging

An adaptive learning environment needs to take in data, evaluate its significance, and then output a personalized experience to the user. Since *System Dot* aims to take a player's behavior as input to the adaptive system, there needed to be a way to keep track of every action the player performs. Logger code has been injected into gameplay scripts involving moments where the player directly interacts with the game. For instance, a log is kept for all the times a player opens the terminal window for an enemy and executes code. All these details are being written into a serialized file in the *LogToFile* class. Each line of the log file corresponds to the timestamp, action, and action type of a player's action in that order. Here is a sample of what one player's log file looked like:

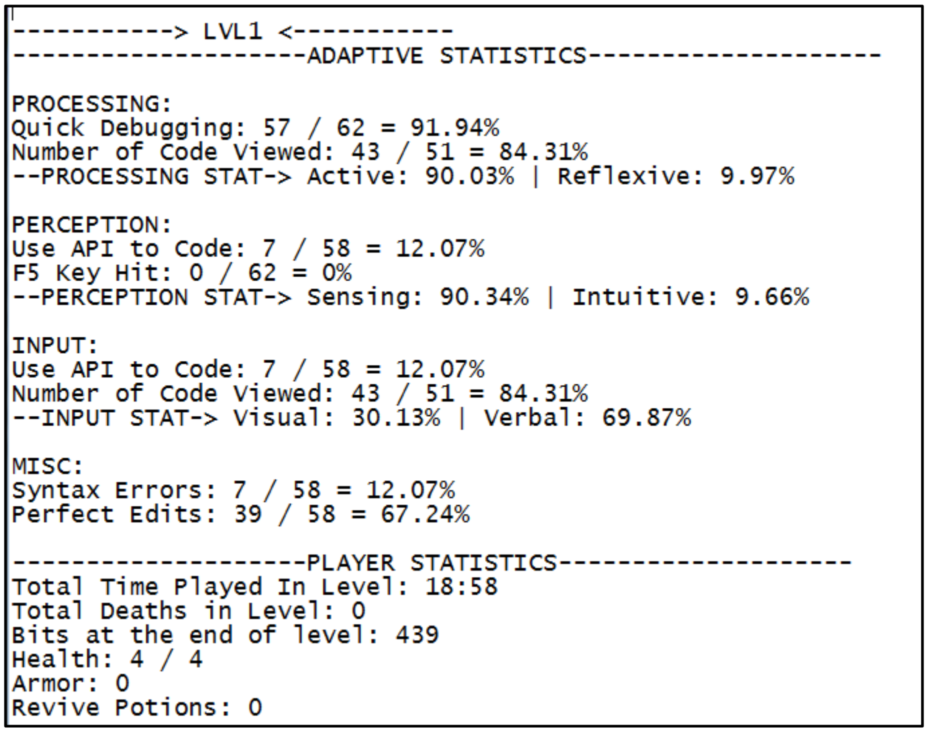


```
10/18/2017 8:35:07 PM  OPEN-TERMINAL-FOR-TutorialChest (UnityEngine.GameObject)  TERMINAL-WINDOW
10/18/2017 8:35:07 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/18/2017 8:35:12 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/18/2017 8:35:12 PM  CORRECT-SYNTAX  CODE
10/18/2017 8:35:13 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/18/2017 8:35:14 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/18/2017 8:35:15 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/18/2017 8:35:17 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/18/2017 8:35:18 PM  Intellisense-END-DIALOGUE  DIALOGUE
10/18/2017 8:35:23 PM  OPEN-TERMINAL-FOR-Slime (16) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/18/2017 8:35:30 PM  PRESSED-DOWN-KEY  TERMINAL_WINDOW
10/18/2017 8:35:30 PM  PRESSED-ENTER  TERMINAL_WINDOW
10/18/2017 8:35:33 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/18/2017 8:35:33 PM  CORRECT-SYNTAX  CODE
10/18/2017 8:35:35 PM  OPEN-TERMINAL-FOR-Slime (16) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/18/2017 8:35:36 PM  PRESSED-DOWN-KEY  TERMINAL_WINDOW
10/18/2017 8:35:36 PM  PRESSED-ENTER  TERMINAL_WINDOW
10/18/2017 8:35:37 PM  PRESSED-DOWN-KEY  TERMINAL_WINDOW
10/18/2017 8:35:37 PM  PRESSED-ENTER  TERMINAL_WINDOW
10/18/2017 8:35:38 PM  PRESSED-DOWN-KEY  TERMINAL_WINDOW
10/18/2017 8:35:41 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/18/2017 8:35:47 PM  OPEN-TERMINAL-FOR-Slime (17) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/18/2017 8:35:51 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/18/2017 8:35:51 PM  CORRECT-SYNTAX  CODE
10/18/2017 8:35:56 PM  OPEN-TERMINAL-FOR-Slime (18) (UnityEngine.GameObject)  TERMINAL-WINDOW
```

Figure 3.6: Example of a Log File

The timestamps indicate the order of which the player performed the following actions. Then, the action gives a brief description of how the player interacted with the virtual world. The action types are similar to the main points discussed previously in the **Technical Description of the Virtual World** section. The *dialogue* action type deals with any form of interaction with IntelliSense such as skipping through its dialogue. The *terminal-window* action type deals with how the user plays around with the built-in coding environment including the keystrokes taken like the ENTER or ARROW keys or the objects clicked like the green debug button. Lastly, the *code* action type revolves around the custom parser built. Does the code conform to the context-free grammar defined beforehand? How often does the player make syntax mistakes? These action types provide an easy method to parse and data mine the log file for relevant information to answer these types of questions.

Players' actions are not the only items tracked by *System Dot*. Specific in-game statistics like the time taken to beat a level and the number of deaths track the player's performance. These attributes are also printed to a serialized file in a class called *StatsLog*. On top of in-game statistics, adaptability behavior was also tracked to build the user's model for adaptation including the percentage of times the player used the API to aid in their journey or how many objects they viewed in a certain level. This will be fleshed out extensively in the next section, but here is a snippet of a real user's statistics through the first level.



```
-----------> LVL1 <-----------
--------------------ADAPTIVE STATISTICS--------------------

PROCESSING:
Quick Debugging: 57 / 62 = 91.94%
Number of Code Viewed: 43 / 51 = 84.31%
--PROCESSING STAT-> Active: 90.03% | Reflexive: 9.97%

PERCEPTION:
Use API to Code: 7 / 58 = 12.07%
F5 Key Hit: 0 / 62 = 0%
--PERCEPTION STAT-> Sensing: 90.34% | Intuitive: 9.66%

INPUT:
Use API to Code: 7 / 58 = 12.07%
Number of Code Viewed: 43 / 51 = 84.31%
--INPUT STAT-> Visual: 30.13% | Verbal: 69.87%

MISC:
Syntax Errors: 7 / 58 = 12.07%
Perfect Edits: 39 / 58 = 67.24%

--------------------PLAYER STATISTICS--------------------
Total Time Played In Level: 18:58
Total Deaths in Level: 0
Bits at the end of level: 439
Health: 4 / 4
Armor: 0
Revive Potions: 0
```

Figure 3.7: Example of Adaptive Statistics in Log File

Most of this data would become obsolete or inaccurate if there was not a way for it to persist through multiple play sessions. Having an auto-save feature allows the player to take a break, close the game, and then continue their play session at a later time while retaining all of their progress from their previous play, including adaptive statistics and logged player actions. *System Dot* creates a template of a game state, populates the template with information from the current game state, and stores it on the local system whenever the player hits a designated checkpoint in the level. When the game is loaded again at a later time, the values saved into the game state template are loaded in by the system and then presented to the user. The class responsible for storing this game state template is *PlayerStats*.

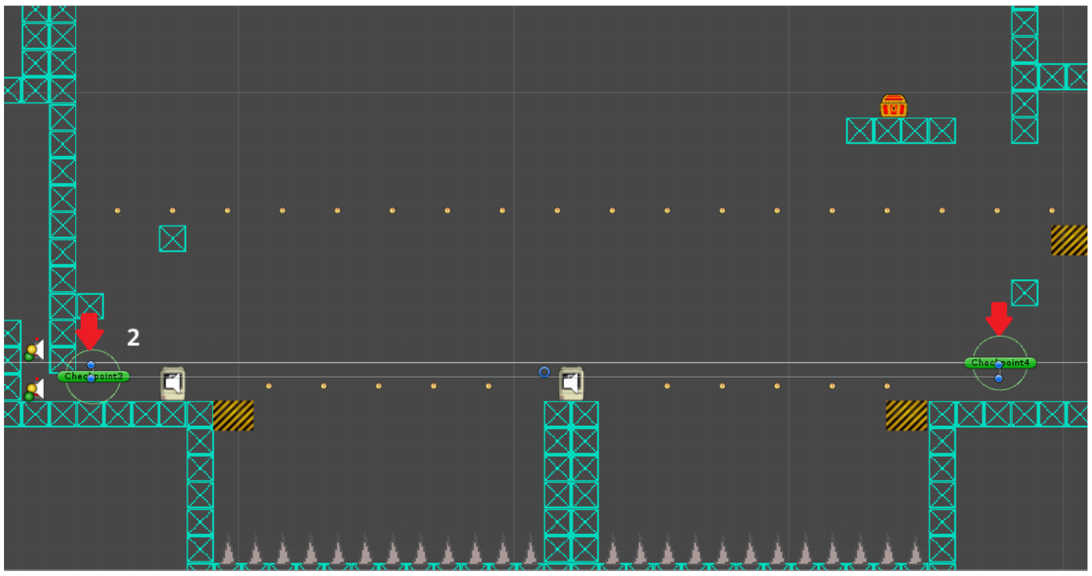


Figure 3.8: Example of Checkpoints in the First Level

The entire playthrough of *System Dot* will take more than five hours to complete. It is not expected of players to be able to finish the whole experience in one play session. Therefore, it is essential that there is a way for a player to resume the game in the most recent state they left it in similar to how a student comes into class the next day continuing a prior topic. Having data persist between scenes and play sessions allows *System Dot* to eliminate mental fatigue and have the player turn on their brains when they decide to continue playing. Having the player oversee their learning is the best way to get them engaged and one of the biggest advantages a virtual world has over a traditional classroom learning environment.

4. THE ADAPTABILITY COMPONENT

4.1 Introduction

The ultimate goal of this thesis is to develop a system that can adapt its messages and content to a player's learning styles. However, this current thesis and game does not yet include this piece. To work toward that goal, it is important to understand approaches to creating an adaptive system, and to describe the approach used to develop the underlying system for *System Dot* once the learning style model is tested for its ability to detect a player's learning style.

There have not been an extensive number of adaptive systems developed for educational virtual worlds. Among those that have been developed, the virtual world tends to be an extension of an Adaptive Educational Hypermedia System (AEHS). For instance, Chittaro and Ranon (2007) integrated a virtual environment within a web browser through a 3D plug-in to place the user into a 3D virtual environment representing their hypermedia content. They fed user behavior into the *AHA!* Engine, which uses the adaption/domain model to framework a user's interaction with the system. The engine outputs personalized data that will then be translated into the virtual environment for the player to observe. For example, if the knowledge level of the user is subpar, then the virtual 3D world hides certain objects that hold more advanced topics. This approach is incredibly similar to the hiding of hyperlinks in an AEHS.

Scott, Soria, and Campo (2017) found that a majority of educational virtual worlds use a rule-based approach, which involves a static method of determining whether the system needs to adapt to the student through the adaptation/domain learning model. They agree that the main advantage of using a virtual world as a learning environment is to not only reach a learning objective but to also author these adaptation rules through a storyline (Scott, Soria, and Campo). In the case of *System Dot*, the storyline stealthily interweaves the foundation of computer hardware knowledge in a logical and engaging way.

Most researchers agree that there are various ways adaptive systems can alter the way it displays itself to the learner (Hauger & Köck, 2007; Carmona et al., 2007; Garcia, Amandi, Shiaffino, & Campo, 2005; Scott, Soria, & Campo, 2017). The content being displayed could add prerequisite information or additional explanations to a topic, variants of information by hiding confusing details that the learner is incapable of grasping, and sorting information according to their relevance (Hauger & Köck, 2007). In the case of *System Dot*, the virtual world will hide or show information corresponding to the learning style of the player. For the virtual world to assess what to display to the player, it must first model the player and grasp how they acquire and learn information (Carmona et al., 2008).

Brusilovsky and Millán (2007) claim that a user model is necessary to make adaptability work. A user model is a “representation of information about an individual user” such as how the user interacts with the system’s interface or the types of inputs the user feeds the application (Brusilovsky & Millán, 2007). There are five types of information that can be modeled—knowledge, interests, goals, background, and individual traits.

Since *System Dot* aims to be an educational learning environment, knowledge is an essential piece that needs to be tracked and assigned to the user's model. In the case of this virtual world, the only indication that a user has or has not successfully attained knowledge is through the amount of times they compile code with and without errors. Because there is not a clear indication that a player's completion of a level guarantees expert mastery of the level's central programming concept, this quantitative tracking of syntax errors provided the only reasonable measurement. Interests, goals, and background were not taken into heavy consideration when modeling the player in *System Dot* because the game-like virtual environment did not set out to filter information, distinguish between multiple learning goals, and explicitly query the user for personal information. However, the feature heavily tracked and discussed extensively in the next sections is the user modeling of individual traits.

Brusilovsky and Millán (2007) categorize individual traits in two ways: cognitive and learning styles. Since *System Dot* is an educational learning environment, learning styles were chosen. Individual traits allow for content-level adaptation—what the virtual world presents to the user will depend on how likely they will perceive it based on the way they typically adopt new information. Unlike the user's background, Brusilovsky and Millán (2007) suggest that individual traits need to be extracted from a psychological test. What better way to mentally test the user than through a game-like virtual world. The following two sections will detail what type of learning styles were selected and how they pertain to *System Dot*.

4.2 The Learning Style Model

Carmona et al. (2008) define a learning style as the way a person “collects, processes and organizes information”. Amidst the various learning style models discussed before, *System Dot* implements the Felder-Silverman Learning Style Model (FSLSM) due to its success rate and popularity amongst e-learning environments (Felder, 2003). The FSLSM classifies a learner in four ways:

- **Processing**- Active people tend to be impulsive learners by jumping into the material immediately and trying out different methodologies; they often say “let’s try it out and see how it works”. Meanwhile, reflective people contemplate about what they have learned before jumping into the material; they often say “let’s think it through first”.
- **Perception-** Sensing people typically learn best through concrete facts, details, and data; they focus on memorization and regurgitation of knowledge and do not expect anomalies. On the other hand, intuitive people love to deal with theory and abstractions; they look at the big picture and recognize patterns.
- **Input**- Visual people find images and visual information more appealing than straight text while verbal learners prefer explanations with both written and spoken words.
- **Understanding-** Sequential people adopt information more quickly when it appears to them in an ordered, step-by-step manner. Contrarily, global people organize their information in a holistic fashion in no particular order or reason. Since the design of the virtual world revolves around a sequential progression of

programming concepts, there is not an opportunity for the player to jump around from one concept to another. Therefore, the understanding attribute of the learning theory model has been neglected for this thesis.

With these learning styles in mind, the main goal of *System Dot* is to classify the player as a certain type of learner. Then, the virtual world can adapt the feedback given to the player based on the type of learner he or she is, and on the kind of errors the player makes. For instance, if the player is classified as a visual learner, then more emphasis will be put on the "API" section of the game, which holds a visual glossary of what most of the programming concepts mean. In another example, if the player is classified as an active learner but suffers through a lot of syntactical errors, then *System Dot* will suggest that the player slows down and takes some time to reflect on their code alterations. Ultimately, the virtual world generates a perception of the player that it then uses to maximize its teaching potential. Until the desired precision of the learning style classification system is achieved, this feedback system was only planned and not implemented in this thesis.

4.3 Dynamic Bayesian Network Implementation

There is not one way to implement an adaptive learning system. Scott et al. (2017) cite more than five methods including clustering and machine learning, decision trees, and rule-based adaptation rules. Due to the uncertainty surrounding how particular actions affect the learning style classification in *System Dot*, a probabilistic approach was taken in the form of a Bayesian network.

Bayesian networks are used to adapt many systems. They can be used to represent skills and knowledge in a domain with probabilistic certainty (Le, 2016). Before a Bayesian network can be used, the problem domain must be clearly defined such as the student model and learning objects being assessed (Carmona et al., 2008; Garcia et al., 2005; Culbertson, 2016; Le & Pinkwart, 2015).

Garcia et al. (2005) used a basic Bayesian network to detect the learning style of a student in a Web-based educational system. They used a knowledge-based approach to construct their network. For example, when classifying the student as an active or reflective learner, they evaluated the student's presence in chat rooms and forums. To determine the impact of a certain parameter on the learning style, they performed prior small experiments that cross-examined a student's behavior with the learning style derived by the Index of Learning Styles Questionnaire (Felder & Soloman, 2017). With the appropriate conditional probability tables, they could classify a student's learning style with a high degree of precision.

Carmona et al. (2008) approached the classification of a student's learning style through a dynamic approach. Unlike the previous static model, Carmona sought to use a temporal network to monitor continuous interactions from the user with the system. When the student selected learning objects in the future, a new time slice of the network was generated and the learning style would be updated. Ultimately, each succeeding time slice classified learners based on the current learning objects selected and past classification by the network. Likewise, *System Dot* is a progressive game that tracks the temporal state of the student model based on levels completed. Consequently, a similar

Dynamic Bayesian Network was used to model the student's learning style in *System Dot*.

A DBN or Dynamic Bayesian Network (Dean & Kanazawa, 1989) is a representation of a problem domain through a series of random variables and their dependencies with each other. The network has both a quantitative (represented by the conditional probability distribution of these random variables) and qualitative part (shown through its visual structure). Each dependency is visualized as an arc from one node to another, and the strength of that dependency relies on the conditional probability. *System Dot* uses a DBN over a standard Bayesian Network (BN) because the virtual world is changing over each level. Consequently, the results improve over time based on the prior classification in a previous time period. In the case of *System Dot*, the time granularity between each network is the time when the player finishes a level.

Let us run through the player statistics tracked by *System Dot* and how they pertain to the three learning styles assessed:

Processing (Active vs. Reflective)

- *Time To Click Debug After Change (TTD):* represents the time taken to hit the "debug" button or F5 key to execute the code after the player modified it. A threshold of five seconds has been set to determine whether the player was "quick" or not. The probability taken is the number of times the player quickly executed the modified code under five seconds over the total number of times the player executed modified code. The higher the probability, the less contemplative the player is in assessing their modifications to a piece of code and the more active they are in their learning style.

- *View Object Code (VOC)*: Scattered throughout the level are enemies whose code does not necessarily need to be seen. The player can go about their play session without ever hacking these objects. This field represents the number of enemies viewed by the player. When the player actively clicks around to observe their environment, they are taking control of their learning by jumping right into the material and observing their surroundings. The metric represents the number enemies viewed over the total amount of enemies in the level.

Perception (Sensing vs. Intuitive)

- *Use API*: The API is a cheat sheet at the voluntary use of the player. When they open the API, they are exposed to a visual library of programming knowledge that they can use to their advantage. This metric is tracked when the API was opened within the time frame that the player made an edit to the code. As a result, the probability of the API being used is the total number of times it was opened during a code edit over the total number of code edits. The higher the number, the less sensing the player is with memorizing important syntax and the more intuitive they are with understanding abstractions.

- *F5 Hit:* The player has the option to hit F5 as a shortcut to execute the code. It is completely optional, but a "sensing" learner would primarily take advantage of this shortcut as a memorization tool.

Input (Visual vs Verbal)

- *Use API:* As described before, if the player uses the API as a crutch, then they tend to represent important programming concepts through their visual counterparts.
- *View Object Code*: If the player tends to explore why an object visually looks like it does through the code, then we can say they are less prone to understand information through just text and words.

Feedback Indicators

- *Number of Syntax Errors*: The number of times the player made a mistake with the coding syntax over the total times the player modified the code. While this metric is not used in conjunction with the previous metrics or in the DBN in general, it is helpful both in modeling the knowledge level of the player and determining the type of feedback returned in future development of the feedback system. Ideally, if the player suffers more syntactical errors than flawless code executions, then the virtual world will suggest that the player change their learning style since it is not working.
- *Number of Perfect Edits:* Perfect edits constitute the situations where the player modified the code flawlessly the first time. The higher the number, the better the system rates their performance. Unlike the number of syntax

errors, *System Dot* will deem the player knowledgeable and capable of learning through the environment. As a result, it will interfere less when the feedback system is implemented later.

To concisely summarize, here is a table that sorts the above variables with their corresponding learning style:

Table 4.1: Summation of DBN Parameters

Learning Style	**Player Behavior**	**Assumption**	**Algorithm**
Processing	Time to Click Debug After Change (TTD)	Threshold to click debug is 5 seconds	p(f) = $\frac{\# \; of \; debugs}{total \; \# \; of \; debugs} > .5$ ~ active
	View Object Code (VOC)	An object is seen if it has been clicked by the player	p(f) = $\frac{\# \; of \; objects \; seen}{total \; \# \; of \; objects} > .5$ ~ active
Perception	Use API	API is considered used if it is open while code is shown	p(f) = $\frac{\# \; of \; times \; API \; open}{total \; \# \; of \; edits \; to \; objects} > .5$ _~ intuitive
	F5 Key Hit	Tracked when the F5 key is hit while code is shown	p(f) = $\frac{\# \; F5 \; key \; hit \; on \; debug}{total \; \# \; of \; debugs} > .5$ ~ sensing
Input	Use API	API is considered used if it is open while code is shown	p(f) = $\frac{\# \; of \; times \; API \; open}{total \; \# \; of \; edits \; to \; objects} > .5$ ~visual
	View Object Code (VOC)	An object is seen if it has been clicked by the player	p(f) = $\frac{\# \; of \; objects \; seen}{total \; \# \; of \; objects} > .5$ ~visual

There were multiple ways the DBN could have been drawn. One way is for all the nodes to be drawn into one huge DBN [A].

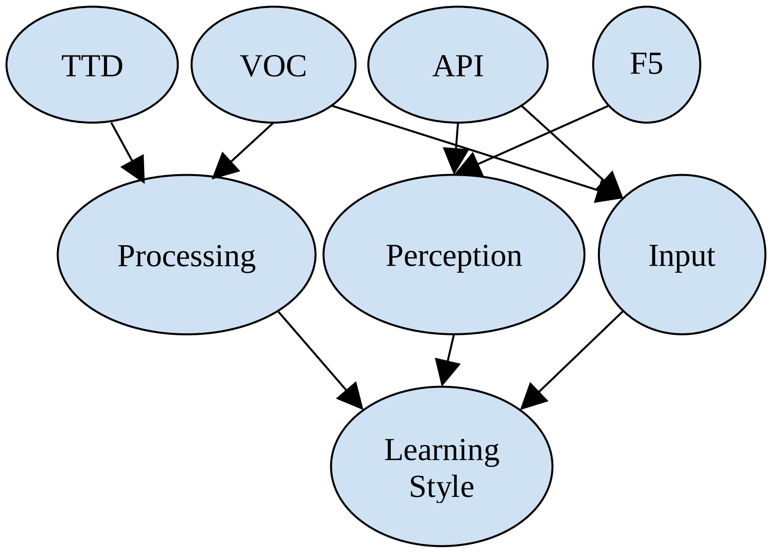


Figure 4.1: Conglomerated DBN [A]

The other way is for each learning style to be divided into separate sub-networks [B].

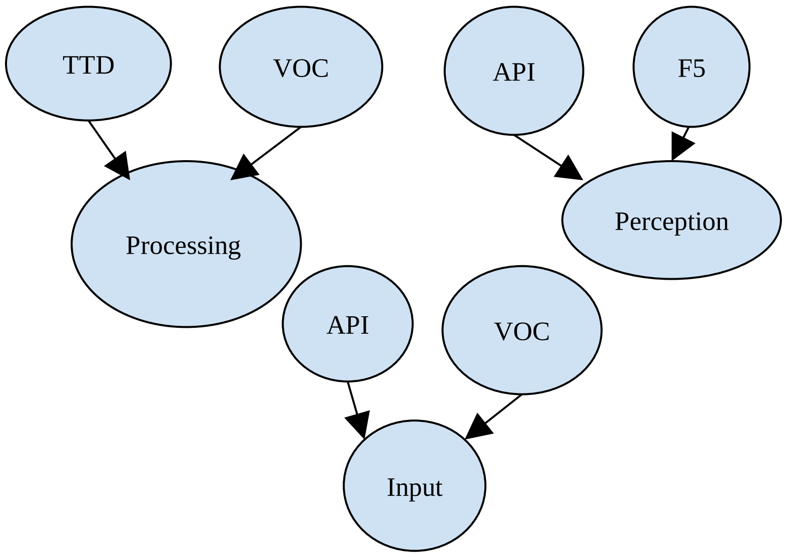


Figure 4.2: Separated DBN [B]

DBN B was chosen because of the reduced computational complexity. Network A has 128 parameters while Network B has a maximum of 8 parameters.

The above networks suggest the following:

- Determining if the player is active or reflective depends on the percentage of times the player quickly executes the code after changing it and the percentage of viewed legacy code
- Determining if the player is sensing or intuitive depends on the percentage of coding that the player used to reference the API and the number of times the player hits the F5 key when executing code.
- Determining if the player is a visual or verbal learner depends on how frequently they open the API and view enemy's legacy code.

Right now, these relationships are snapshots of a player's behavior through a particular level. However, if we would like to capture the progression of levels the player goes through, then we would need to make this network dynamic. This means that once the player finishes a level, a time slice of that network is generated based on the above player behaviors and the previous level's results. To define the DBN's parameters, *System Dot* uses a combination of

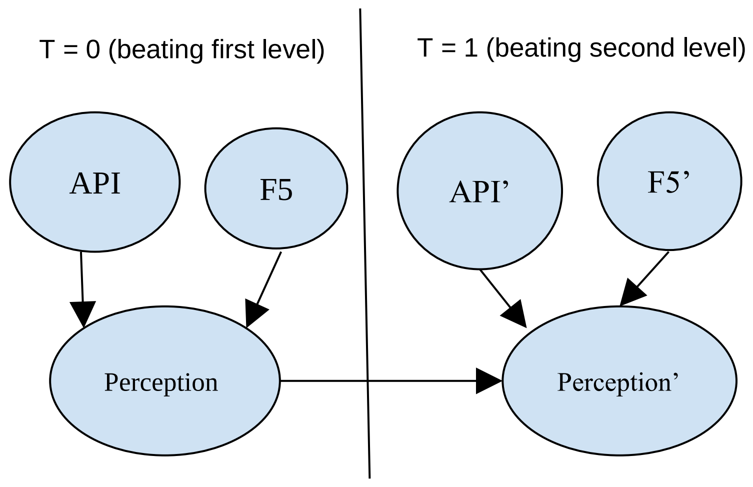


Figure 4.3: DBN Adjusts Over Time

personal knowledge and player data when determining how a certain player behavior affects a learning style. The initial probabilities of the parameters are taken from the results of the player completing the first level. The method for estimating the conditional probabilities involve assigning an assumptive weight to all these parameters and then calculating the effect it had on determining the learning style of the player. It is impossible to determine accurate conditional probabilities without actual data so preliminary ones were set according to personal assessment. Let us assess each learning styles' conditional probability table and see how these probabilities were determined.

Table 4.2: Preliminary Processing Learning Style Parameter Weights

TTD	VOC	Processing- Active	Processing- Reflective
T	T	1	0
T	F	.75	.25
F	T	.25	.75
F	F	0	1

The swiftness of a player executing code they modified is weighed more than the opportunity they had actively viewing object code examples. The chosen weight of TTD was three times more indicative of an active or reflective learner than VOC. For instance, the first row depicts a situation where the player has viewed a majority of the enemies and typically executes code immediately. With both traits suggesting that the player is an active learner and they are both being shown, there is 100% certainty that the player is an active learner (of course, this will change based on the results of the actual data).

Likewise, the second row depicts the impact of the TTD's weight by assigning the certainty of the player being an active learner to three times as likely as being a reflective learner.

Table 4.3: Preliminary Perception Learning Style Parameter Weights

Use of API	F5 Key Hit	Perception - sensing	Perception - intuitive
T	T	0	1.00
T	F	.2	.8
F	T	.8	.2
F	F	1.00	0

Using the API frequently when modifying and viewing code is weighed more than the number of times the player hit the F5 key to debug their code. The chosen preliminary weight of using the API was four times more indicative of sensing or intuitive learner than the number of times they hit the F5 key. Just like the previous conditional probability table, the first row shows the favorable extremes of both player metrics. In a situation where the player uses the API and also hits the F5 key to execute their modified code, the network assigns a 100% certainty that the player is an sensing learner (once again, this will change once actual data is analyzed). Similarly, the second row depicts the impact of using the API by assigning the certainty of the player being an intuitive learner to four times as likely as being a sensing learner.

Table 4.4: Preliminary Input Learning Style Parameter Weights

Use of API	VOC	Input - Visual	Input - Verbal
T	T	1.00	0
T	F	.75	.25
F	T	.25	.75
F	F	0	1.00

Similar to the processing conditional probability table, the impact of VOC is three times less than the use of the API to maintain consistency with previous models. Once again, these values are based on personal intuition and assumptions and only serve as preliminary data to kickstart the DBN.

4.4 Sample Assessment

The first time *System Dot* will classify the player as a certain type of learner is when they successfully complete the first level. Let us see how the "Perception" learning style is classified if the player opens the API menu 21 out of the 50 times they could have when they changed the code and hit the F5 key 58 out of 80 times to debug. The corresponding probabilities that the player would use the API and hit the F5 key is 43% and 72%, respectively, simply by taking the number of occurrences and dividing it by the total number of opportunities for the metric to occur. Based on this knowledge, to determine if the player is sensing, we must calculate the following probability *P(Perc = sensing, API, F5)*, which is the probability that the player is a sensing learner and uses the

API and hits the F5 key. Through the chain probability rule, this joint probability can be reduced to the following marginal probability:

P(Perc = sensing, API, F5) = P(Perc = sensing | API, F5) * P(API) * P(F5)

Since we are marginally calculating the probability, the above equation can be expanded to:

P(Perc = s, API, F5) = P(Perc = s | API = T, F5 = T) * P(API = T) * P(F5 = T) +

P(Perc = s | API = T, F5 = F) * P(API = T) * P(F5 = F) +

P(Perc = s | API = F, F5 = T) * P(API = F) * P(F5 = T) +

P(Perc = s | API = F, F5 = F) * P(API = F) * P(F5 = F)

Substituting what we know, we can calculate P(Perc = s, API, F5) like so:

P(Perc = sensing, API, F5) =

0 * .43 * .72 + .2 * .43 * .28 + .8 * .57 * .72 + 1 * .57 * .28

= 0 + .024 + .328 + .1596 = .5116 = **~51% sensing**

After the first level, the player is classified as either a “sensing” or “intuitive” learner based on the higher posterior probability. In this case, the player is nearly classified as a “sensing” (51%) rather than an “intuitive” (49%) learner.

When the player completes the second level, a new set of probabilities will be generated for opening the API and hitting the F5 debug button. These metrics will be updated from the previous level’s probability. *System Dot* uses the previous perception probability of the first level to determine the growth or decline of the player’s current perception learning style. Essentially, past data will update the beliefs of the system’s current data. This will allow the DBN to precisely refine its learning style classification over time.

For instance, let us assume that in the second level, the player viewed the API 10 out of 47 times and hit the F5 key 61 out of 98 times. The corresponding probabilities of API’ and F5’ will be (19 + 10) / (50 + 47) =29.89% and (58 + 61) / (80 + 98) = 66.85%, respectively. The calculation of the second level’s perception learning style is as follows:

P(Perc’ = s, API’, F5’) =

P(Perc’ = s | API’ = T, F5’ = T) * P(API’ = T) * P(F5’ = T) +

P(Perc’ = s | API’ = T, F5’ = F) * P(API’ = T) * P(F5’ = F) +

P(Perc’ = s | API’ = F, F5’ = T) * P(API’ = F) * P(F5’ = T) +

P(Perc’ = s | API’ = F, F5’ = F) * P(API’ = F) * P(F5’ = F) =

0 * .2989 * .6685 + .2 * .2989 * .3315 + .8 * .7011 * .6685 + 1 * .7011 * .3315 =

0+ .0198 + .3749 + .2741 = .62718 = **~63% sensing**.

It makes sense that the classification of the learner as a “sensor” is higher than before. The player continues to scarcely use the API, which increases the “sensing” metric. As the player continues to play, their decisions and behavior inside the virtual world will continue to precisely classify them as a certain type of learner. These calculations occur for each of the three sub-networks illustrated before-- “processing”, “perception”, and “input”. With these learning styles classified, the system can better adapt by giving insightful and personalized feedback rather than generic and repetitive ones.

5. USABILITY TESTING CASE STUDY AND RESULTS

5.1 Introduction

The main goal of this thesis is to discuss the method taken to develop an adaptive feedback system in the future for the virtual world *System Dot*. The approach decided upon was a detection system capable of classifying a player's learning style according to the Felder-Silverman Learning Style Model. This system can also adapt its style classification to a player's response to changes in the game as shown through a case study. User testing was ultimately conducted to test the validity of this detection system and evaluate the effectiveness of the learning parameters presumed in the previous section. Once satisfied with the accuracy and precision of this detection system, steps can then be later taken to implement a personalized content and messaging management system for the virtual world. Lastly, a qualitative feedback survey modeled after the System Usability Survey (SUS) format was also distributed to assess the accessibility of the virtual world. The culmination of both results will reveal new ways to improve the design and technicality of the virtual world in the future.

5.2 Data Collected

Felder and Soloman (2008) introduced a psychometric Index of Learning Style Questionnaire (ILSQ) to formally classify a student as a strong, moderate, or mild type of learner for each one of their learning styles. The basis of the adaptability component in *System Dot* is inherently assumptious. A combination of player behavior data and personal judgement was used to preliminarily determine the impact player behavior had on a certain learning style.

Due to the length of development and time constraints, an IRB-approved research study was not held for the intended target audience of eighth graders to high school students. Instead, the study was approved for adult participants aged 18 and up and focused on validating these assumptions and gathering data rather than determining if the virtual world was truly effective at teaching programming. The user testing was conducted with a convenience sample consisting of friends and family who ranged from 18 to 26 years old with varying degrees of programming experience. Because the full playthrough of *System Dot* is estimated to take more than five to six hours, a “demo” version of the game was developed that took the player through less than half of the game in about one to two hours. The first level dealt with output systems and basic syntax while the second level ramped up in difficulty with data types and object methods.

Dividing the game in half still lent itself as a suitable test environment for the DBN since the two levels with two boss fights provided four time stamps to capture and refine the player’s learning style classification. Once the tester played through the game, they were then asked to send over the log files the game tracked throughout their play session. Afterward, they would fill out the ILSQ and a short usability feedback survey. Ultimately, the results of the ISLQ would either validate or invalidate the learning style classification generated by the virtual world through the log files. With this data, the system could be refined and harnessed to classify a player’s learning style more precisely in the future.

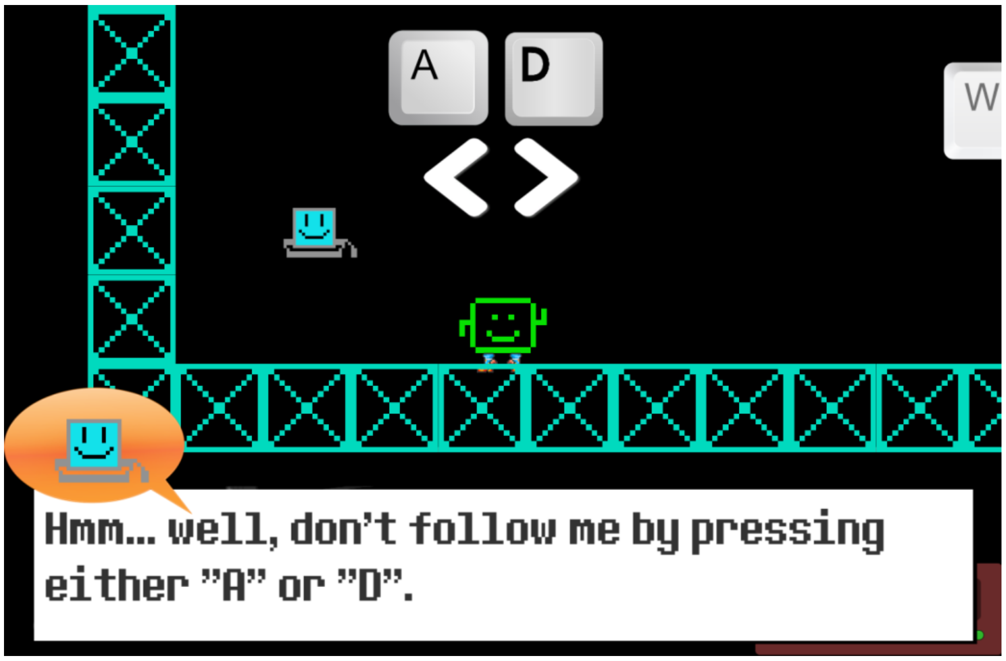
5.3 A Case Study

Due to the limited amount of user tests, it would be more worthwhile to observe one of the tester's actual learning style derived from the ILSQ, heavily evaluate their logged behavior as they progressed through the demo of the game, and assess how accurately the network classified the player in accordance to the ILSQ rating. The ILSQ classifies a learner on a scale from 1 to 10, with 1 being a mild, neutral form of adopting the learning style while 10 being the strongest. The player we will be studying was classified by the ILSQ as follows:

Table 5.1: Player's ILSQ Rating

Processing (Active vs. Reflective)	Active - 3
Perception (Sensing vs. Intuitive)	Intuitive - 7
Input (Verbal vs. Visual)	Visual - 3

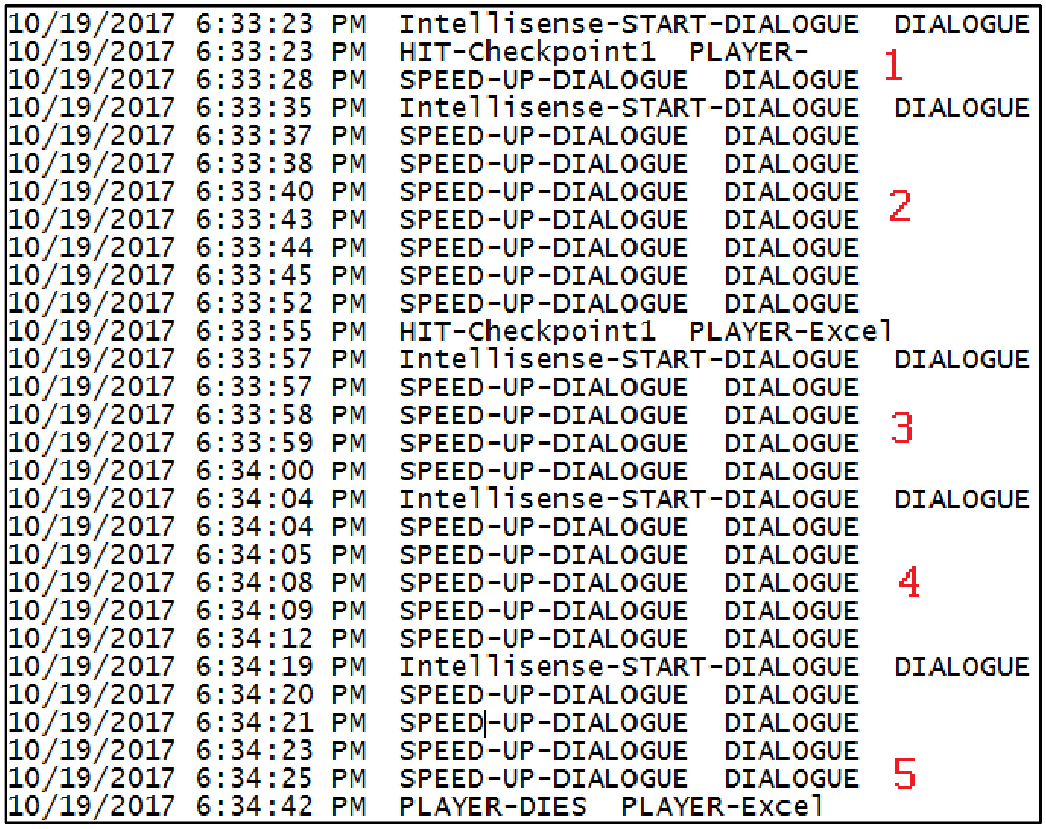
The player is a strongly intuitive and mildly active and visual learner. We can imagine that, based on the definitions of these learning styles, that the player typically thinks with the big picture in mind, thinking abstractly and visually when approached with new information, impulsively implementing different methodologies to better grasp the material. Now, let us run through how the player behaved throughout the virtual world and tie these behaviors to the classification generated by the ILSQ.



Hmm... well, don't follow me by pressing either "A" or "D".

Figure 5.1: IntelliSense's Movement Tutorial

When the player first enters the virtual world, they are greeted by IntelliSense, who asks for the player's name (1) and then undergoes a long, drawn-out tutorial explaining the basic controls of the virtual world (2). To reinforce basic movement, those same controls are depicted visually above the player as a continuous reference. As the tutorial progresses, more controls are being introduced including jumping (3), double-jumping (4), and changing the color of the player's boots (5). The time a dialogue segment starts and whether or not the player pressed the SPACE key to skip it is logged in the level's corresponding log file with a timestamp. We can see from the log file below that the player sped through all five dialogue sequences, hitting the SPACE key to quickly progress through IntelliSense's speech. Since the player was originally classified as a visual learner, we can assume that they must have understood the visual cues the game was giving them and did not bother to *read* the same instructions. While that tutorial would have taken at least two minutes to completely digest and progress through for a player who did not skip through it, this player took half the time with only a minute. However, after the fifth dialogue sequence, the player is confronted with a small assessment, checking whether or not they had paid attention to the tutorial.

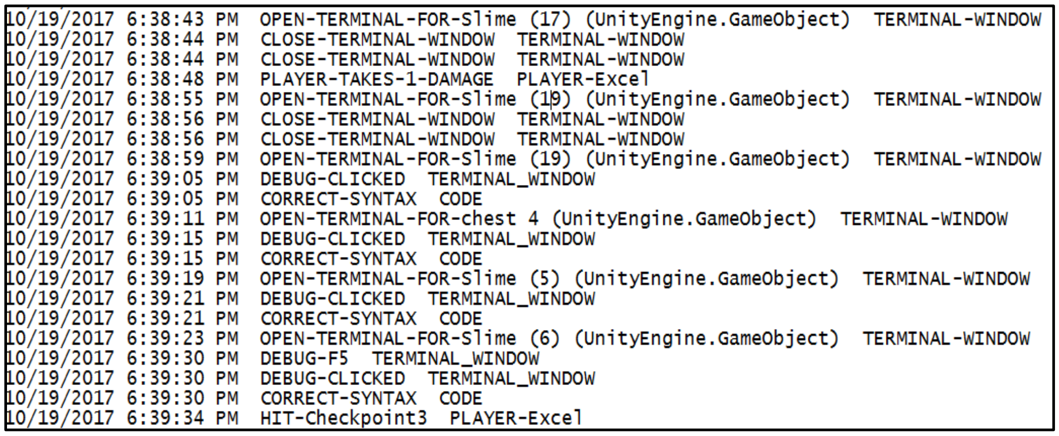


```
10/19/2017 6:33:23 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/19/2017 6:33:23 PM  HIT-Checkpoint1  PLAYER-
10/19/2017 6:33:28 PM  SPEED-UP-DIALOGUE  DIALOGUE   1
10/19/2017 6:33:35 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/19/2017 6:33:37 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:33:38 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:33:40 PM  SPEED-UP-DIALOGUE  DIALOGUE   2
10/19/2017 6:33:43 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:33:44 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:33:45 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:33:52 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:33:55 PM  HIT-Checkpoint1  PLAYER-Excel
10/19/2017 6:33:57 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/19/2017 6:33:57 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:33:58 PM  SPEED-UP-DIALOGUE  DIALOGUE   3
10/19/2017 6:33:59 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:00 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:04 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/19/2017 6:34:04 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:05 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:08 PM  SPEED-UP-DIALOGUE  DIALOGUE    4
10/19/2017 6:34:09 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:12 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:19 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/19/2017 6:34:20 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:21 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:23 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 6:34:25 PM  SPEED-UP-DIALOGUE  DIALOGUE   5
10/19/2017 6:34:42 PM  PLAYER-DIES  PLAYER-Excel
```

Figure 5.2: Player’s Log File during IntelliSense’s Movement Tutorial

After the player movement tutorial, the player is confronted with a pit of spikes that can only be passed if they switch their boots' colors from blue to green using the 'Q' key. According to the very end of the log file, the player failed this test, perishing to the pit of spikes. We can assign this impulsiveness to the active learning style of the player as classified by the ILSQ. Within the first minute of the game, we have already observed player behavior that supports the ILSQ's classification of the learner as an active and visual learner.

Right after IntelliSense talks about how a certain command changes the colors of objects, the player is confronted with two colored enemies. Instead of investigating how these objects exhibit different colors than the one they previously saw by clicking on them and observing their code like they did the first enemy, the player continues to play the game by swiftly killing them. This can be seen as another impulsive act, increasing the chance that the player is an active learner.



```
10/19/2017 6:38:43 PM  OPEN-TERMINAL-FOR-Slime (17) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:38:44 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/19/2017 6:38:44 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/19/2017 6:38:48 PM  PLAYER-TAKES-1-DAMAGE  PLAYER-Excel
10/19/2017 6:38:55 PM  OPEN-TERMINAL-FOR-Slime (19) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:38:56 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/19/2017 6:38:56 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/19/2017 6:38:59 PM  OPEN-TERMINAL-FOR-Slime (19) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:39:05 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 6:39:05 PM  CORRECT-SYNTAX  CODE
10/19/2017 6:39:11 PM  OPEN-TERMINAL-FOR-chest 4 (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:39:15 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 6:39:15 PM  CORRECT-SYNTAX  CODE
10/19/2017 6:39:19 PM  OPEN-TERMINAL-FOR-Slime (5) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:39:21 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 6:39:21 PM  CORRECT-SYNTAX  CODE
10/19/2017 6:39:23 PM  OPEN-TERMINAL-FOR-Slime (6) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:39:30 PM  DEBUG-F5  TERMINAL_WINDOW
10/19/2017 6:39:30 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 6:39:30 PM  CORRECT-SYNTAX  CODE
10/19/2017 6:39:34 PM  HIT-Checkpoint3  PLAYER-Excel
```

Figure 5.3: Player's Log File for Faulty Area

Let us observe the player's behavior when they traverse an environment filled with faults. All of the "Slime" objects have broken code inside of them. It is the player's choice to open these objects' code, fix the syntactical errors, turn them into a certain color, and defeat them. None of this is required to advance to the next checkpoint. However, the player voluntarily explores the virtual world by opening the terminal windows of both objects that block the path to the cache ("chest 4"). Opening the terminal window for "Slime (17)" but not correcting its code shows that the player is observant and tends to understand the computational significance through the written code. When the player corrects the syntax error of "Slime (19)" the first time within 5 seconds of opening the terminal window, it shows they spend little time reflecting about the syntactical changes and would rather jump into finding out if their changes were correct. Both the exploratory nature of the player as well as the quick time it takes for them to debug code depicts an intuitive and active player. On the other hand, when debugging both "Slime (5)" and "Slime (6)", the player spends little time in the terminal window and used the 'F5' key as a shortcut to debug. Using shortcuts tends to depict a sensing learner due to the learner's tendency to memorize techniques.

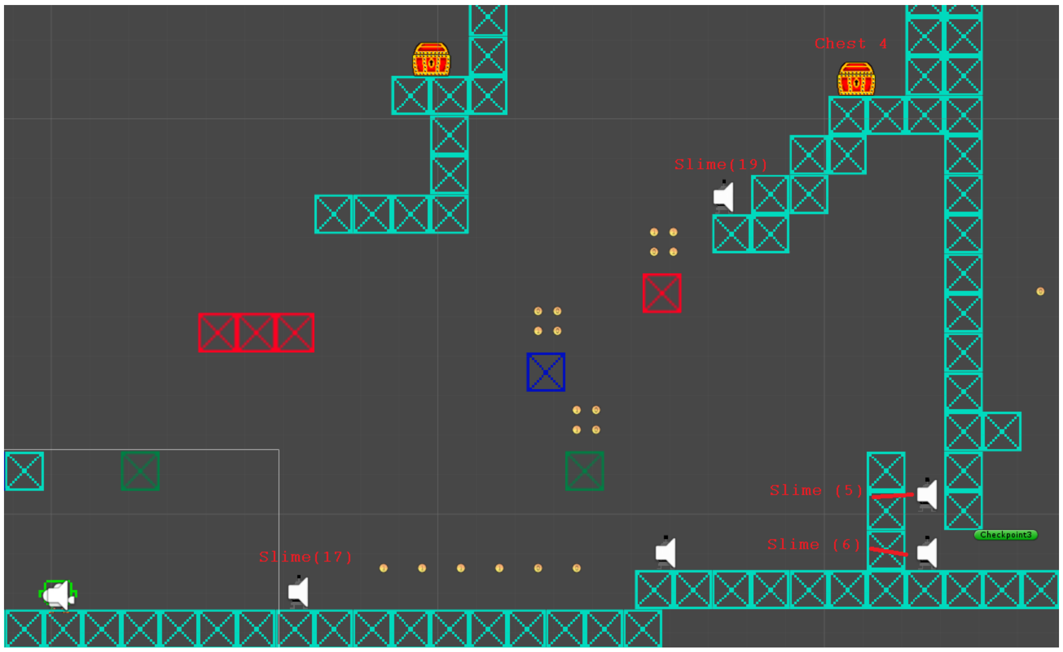
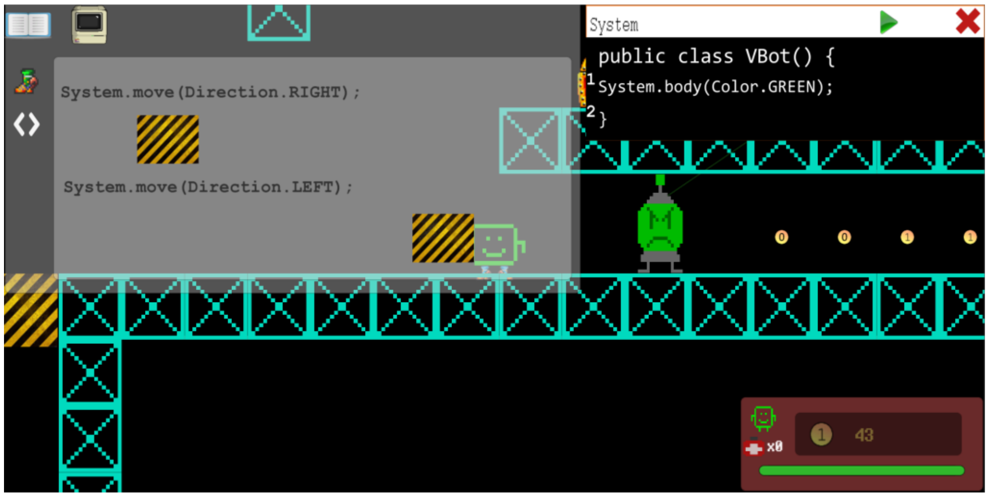


Figure 5.4: Game View of Faulty Area

Most of the challenges in the first level revolve around slight modifications of already existing problems that have solutions. It is as if IntelliSense has taught the player addition and the objects all contain addition problems; it is a way to master already existing material, but not a method to see how the player would react to something *new*. That is where the enemy blocking the path after the fourth checkpoint comes in. The player cannot progress above or below the huge enemy nor can they defeat it by jumping on top of it. Observing how the player reacts to a roadblock they never experienced before will give us insight on how they learn to overcome unexpected scenarios.


```
System.move(Direction.RIGHT);

System.move(Direction.LEFT);
```

```
System
public class VBot() {
System.body(Color.GREEN);
}
```

Figure 5.5: API Showing Solution to Enemy Blocking the Path

The log file below shows the player's interaction with the game to overcome the above obstacle ("Slime (15)"). The player's approach did not immediately reap a valid solution. We can see that within the first 8 seconds of discovering the enemy, they decide to open the object's code, see that it is syntactically correct, and then try to jump over it--to which they take damage and do not pass.

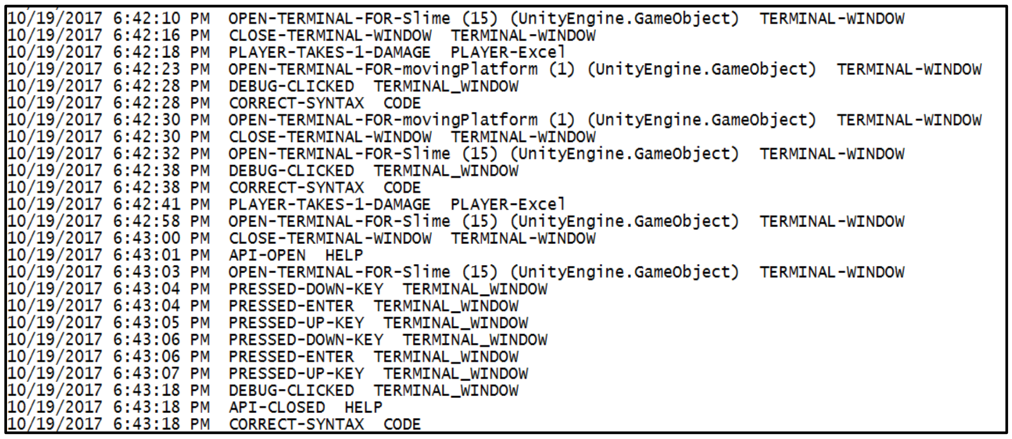
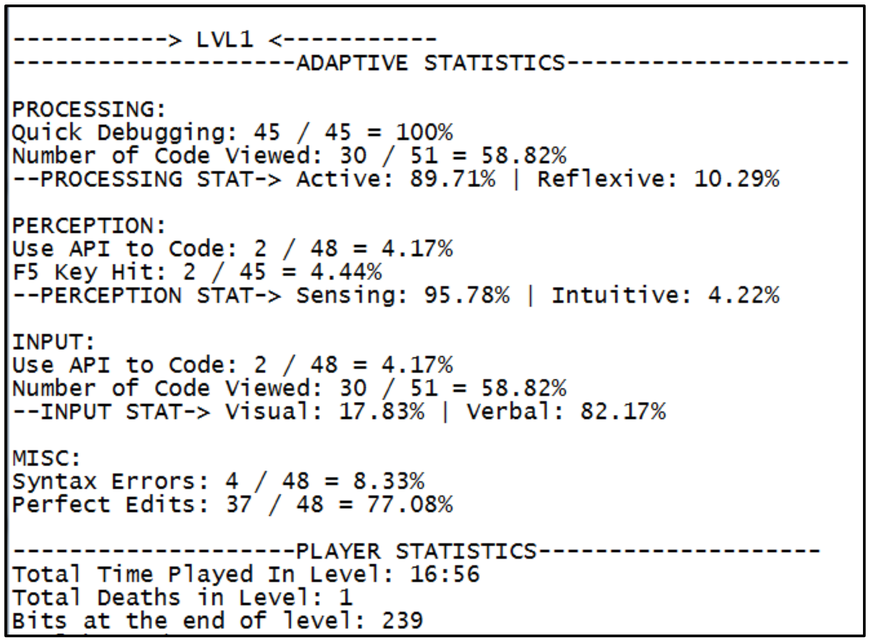

```
10/19/2017 6:42:10 PM  OPEN-TERMINAL-FOR-Slime (15) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:42:16 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/19/2017 6:42:18 PM  PLAYER-TAKES-1-DAMAGE  PLAYER-Excel
10/19/2017 6:42:23 PM  OPEN-TERMINAL-FOR-movingPlatform (1) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:42:28 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 6:42:28 PM  CORRECT-SYNTAX  CODE
10/19/2017 6:42:30 PM  OPEN-TERMINAL-FOR-movingPlatform (1) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:42:30 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/19/2017 6:42:32 PM  OPEN-TERMINAL-FOR-Slime (15) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:42:38 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 6:42:38 PM  CORRECT-SYNTAX  CODE
10/19/2017 6:42:41 PM  PLAYER-TAKES-1-DAMAGE  PLAYER-Excel
10/19/2017 6:42:58 PM  OPEN-TERMINAL-FOR-Slime (15) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:43:00 PM  CLOSE-TERMINAL-WINDOW  TERMINAL-WINDOW
10/19/2017 6:43:01 PM  API-OPEN  HELP
10/19/2017 6:43:03 PM  OPEN-TERMINAL-FOR-Slime (15) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 6:43:04 PM  PRESSED-DOWN-KEY  TERMINAL_WINDOW
10/19/2017 6:43:04 PM  PRESSED-ENTER  TERMINAL_WINDOW
10/19/2017 6:43:05 PM  PRESSED-UP-KEY  TERMINAL_WINDOW
10/19/2017 6:43:06 PM  PRESSED-DOWN-KEY  TERMINAL_WINDOW
10/19/2017 6:43:06 PM  PRESSED-ENTER  TERMINAL_WINDOW
10/19/2017 6:43:07 PM  PRESSED-UP-KEY  TERMINAL_WINDOW
10/19/2017 6:43:18 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 6:43:18 PM  API-CLOSED  HELP
10/19/2017 6:43:18 PM  CORRECT-SYNTAX  CODE
```

Figure 5.6: Player's Log File for Passing Enemy Blocking the Path

In this part of the level, the player must semantically decide what code would be the best input to pass the object. Upon failing the first time around, the player decides to explore their surroundings, trying to find an alternative path (hence the interaction with another moving platform). Ultimately, the player realizes there is no other way and tries to hack the object again. The player does so multiple times until they decide to open the API menu for help. This is where the player's intuition and visual learning style become reinforced. It was not until seeing the movement commands physically moving objects in the API that they realized they could do the same trick with the enemy object. Even though the log files indicate that the player's interaction with the object was syntactically correct, further investigation would come to the conclusion that the player had trouble figuring out how to surpass this puzzle. The player's spontaneous behavior by modifying the code more than once and receiving damage by trying to jump over the object also suggests an active learning style with a "try it" attitude rather than sitting and reflecting.



```
-----------> LVL1 <-----------
--------------------ADAPTIVE STATISTICS--------------------

PROCESSING:
Quick Debugging: 45 / 45 = 100%
Number of Code Viewed: 30 / 51 = 58.82%
--PROCESSING STAT-> Active: 89.71% | Reflexive: 10.29%

PERCEPTION:
Use API to Code: 2 / 48 = 4.17%
F5 Key Hit: 2 / 45 = 4.44%
--PERCEPTION STAT-> Sensing: 95.78% | Intuitive: 4.22%

INPUT:
Use API to Code: 2 / 48 = 4.17%
Number of Code Viewed: 30 / 51 = 58.82%
--INPUT STAT-> Visual: 17.83% | Verbal: 82.17%

MISC:
Syntax Errors: 4 / 48 = 8.33%
Perfect Edits: 37 / 48 = 77.08%

--------------------PLAYER STATISTICS--------------------
Total Time Played In Level: 16:56
Total Deaths in Level: 1
Bits at the end of level: 239
```

Figure 5.7: Player's Adaptive Stats after the First Level

In the end, the player received the above statistics about their learning style after the first level. We can observe throughout the log file of the first level that the player was incredibly active-- every time they opened an object's terminal window, they were able to modify the code and hit debug less than 5 seconds. We can also see throughout the log file that not every object was clicked on. For instance, "Slime (18)", "Slime (16)", and so on were neglected, suggesting an inactive learning style. Considering the ILSQ rated the player as an active learner, the preliminary assessment of the player's processing learning style by the virtual world was fairly accurate.

However, both the perception and input learning styles were not assessed identically to the ISLQ. The only two times the player opened the API was when they were forced to in the opening tutorial of the game or stuck for an extended period of time on a puzzle. Otherwise, the API remained untouched. Because the lack of opening the API data skewed the end result adversely, it might be wise to both revise how the API is integrated within the virtual world and lower its parameter weight in the DBN. Moreover, through our observations of the log file, we also noticed other indicators influencing how the player learned throughout the virtual world that we could leverage for the DBN. Both these thoughts will be elaborated more in the **Discussion** section below.

After the first level, the player is seen as an active, sensing, and verbal learner by *System Dot*. The first level's boss Let us now observe how these learning styles are adjusted after the player progresses through the second level, which has fewer tutorials and more involved programming concepts like data types and object methods.

Once again, at the start of the level and its log file, the player speeds past all forms of verbal dialogue giving context to what the player has experienced so far. This can be shown through almost all instances of dialogue within the level. Instead of delving extensively into each player choice in the second level, let us evaluate how the player approached a problem similar to the huge enemy blocking the path in the first level.

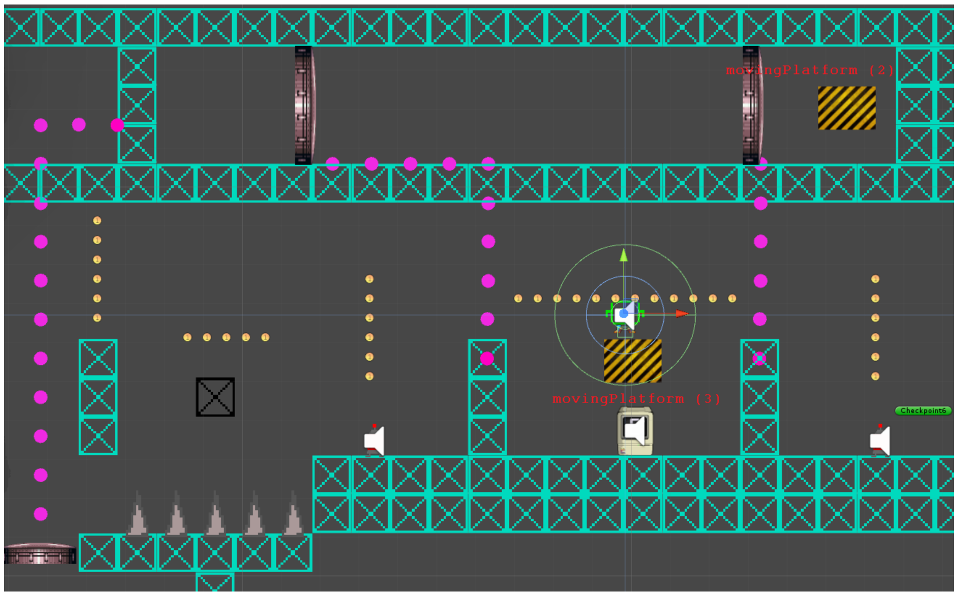
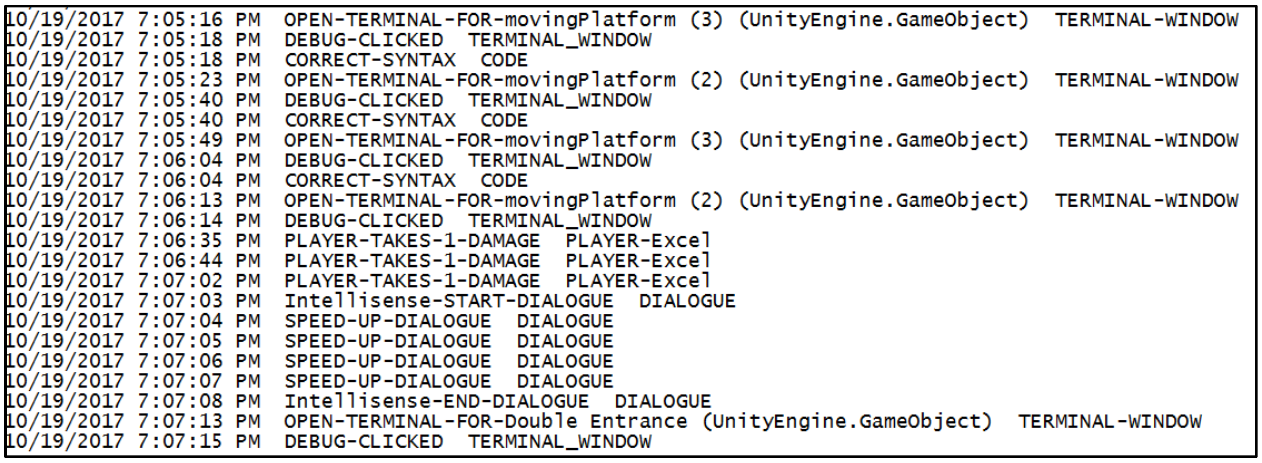


Figure 5.8: Second Level's Activation Platform Puzzle

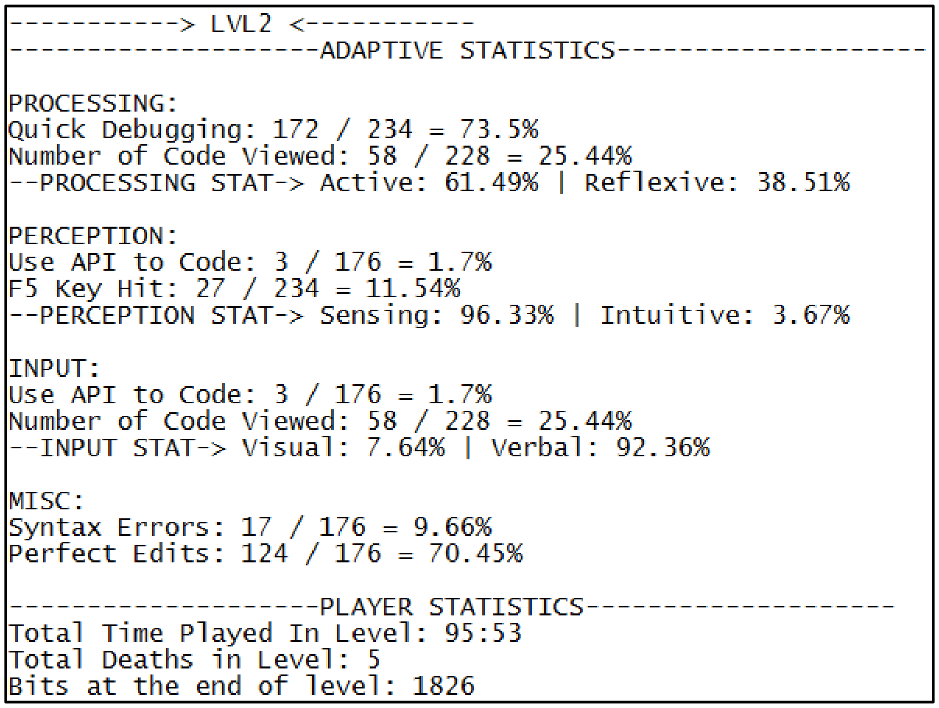
Near a quarter through the level, a moving platform puzzle prevents the player from leaving. In order to surpass it, they must combine what they learned in the first level (the ability to move platforms) and the *System.activate*( ) command of the second level to open the two doors preventing the moving platform from moving to the left and powering the energy line. The ability to solve this problem involves intuition-- can the player think about the bigger picture and move the platform they are standing on to the right first rather than the left? Will the player conquer the problem actively by testing out different paths and trying new code or will they spend time planning ahead? The way the player tackles this conundrum says a lot about their processing and perception learning styles.



```
10/19/2017 7:05:16 PM  OPEN-TERMINAL-FOR-movingPlatform (3) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 7:05:18 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 7:05:18 PM  CORRECT-SYNTAX  CODE
10/19/2017 7:05:23 PM  OPEN-TERMINAL-FOR-movingPlatform (2) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 7:05:40 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 7:05:40 PM  CORRECT-SYNTAX  CODE
10/19/2017 7:05:49 PM  OPEN-TERMINAL-FOR-movingPlatform (3) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 7:06:04 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 7:06:04 PM  CORRECT-SYNTAX  CODE
10/19/2017 7:06:13 PM  OPEN-TERMINAL-FOR-movingPlatform (2) (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 7:06:14 PM  DEBUG-CLICKED  TERMINAL_WINDOW
10/19/2017 7:06:35 PM  PLAYER-TAKES-1-DAMAGE  PLAYER-Excel
10/19/2017 7:06:44 PM  PLAYER-TAKES-1-DAMAGE  PLAYER-Excel
10/19/2017 7:07:02 PM  PLAYER-TAKES-1-DAMAGE  PLAYER-Excel
10/19/2017 7:07:03 PM  Intellisense-START-DIALOGUE  DIALOGUE
10/19/2017 7:07:04 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 7:07:05 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 7:07:06 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 7:07:07 PM  SPEED-UP-DIALOGUE  DIALOGUE
10/19/2017 7:07:08 PM  Intellisense-END-DIALOGUE  DIALOGUE
10/19/2017 7:07:13 PM  OPEN-TERMINAL-FOR-Double Entrance (UnityEngine.GameObject)  TERMINAL-WINDOW
10/19/2017 7:07:15 PM  DEBUG-CLICKED  TERMINAL_WINDOW
```

Figure 5.9: Player's Log File for Passing Second Level's Activation Platform Puzzle

The ideal way to solve this problem would be for the player to move the "movingPlatform (3)" to the right, activate the door, move "movingPlatform (2)" to the left, move "movingPlatform (3)" to the left with the correct power activation, and then finally continue moving "movingPlatform (2)" to the left to power up the door. The optimal solution requires four total debugs. After examining the above portion of the second level's log file, we can see that the player opened the terminal window for "movingPlatform (2)" and "movingPlatform (3)" twice each, totaling four. We can conclude that the player intuitively thought through the solution to the puzzle. Furthermore, we can also see a disparity of time it took the player to properly code the solution. It took the player a mere two seconds to move the first platform. Then seventeen seconds to move the next platform. Then fifteen and finally one. The elongated period of time it took in between could attribute to the player's reflective mindset. Scenarios like these are scattered throughout the second level and the player approached them in a similar fashion. At the end of the level, the player's adaptive statistics looked like the image below:



```
-----------> LVL2 <-----------
--------------------ADAPTIVE STATISTICS--------------------

PROCESSING:
Quick Debugging: 172 / 234 = 73.5%
Number of Code Viewed: 58 / 228 = 25.44%
--PROCESSING STAT-> Active: 61.49% | Reflexive: 38.51%

PERCEPTION:
Use API to Code: 3 / 176 = 1.7%
F5 Key Hit: 27 / 234 = 11.54%
--PERCEPTION STAT-> Sensing: 96.33% | Intuitive: 3.67%

INPUT:
Use API to Code: 3 / 176 = 1.7%
Number of Code Viewed: 58 / 228 = 25.44%
--INPUT STAT-> Visual: 7.64% | Verbal: 92.36%

MISC:
Syntax Errors: 17 / 176 = 9.66%
Perfect Edits: 124 / 176 = 70.45%

--------------------PLAYER STATISTICS--------------------
Total Time Played In Level: 95:53
Total Deaths in Level: 5
Bits at the end of level: 1826
```

Figure 5.10: Player's Adaptive Stats After Second Level

Overall, in the second level, the player did not quickly debug all the objects, taking a longer time on sixty-two of them and reducing the probability for "time to debug" by 26.5%. Additionally, the player did not observe a majority of the objects within the level, reducing the "code viewed" probability by 33.38%. Consequently, the previously strong active statistic dropped by 28.22%, which still classifies the learner as active, but not as strong. The input learning style also changed due to the continued lack of opening the API in the level. However, the perception learning style remained relatively the same.

Table 5.2: Summation of Player's Learning Style Classification Compared to ILSQ

	After Level 1	After Level 1 Boss	After Level 2	After Level 2 Boss	**ILSQ**
Processing (active)	89.71%	89.22%	61.49%	61.06%	**Active- 3**
Perception (sensing)	95.78%	95.95%	96.33%	96.44%	**Intuitive- 7**
Input (visual)	17.83%	17.22%	7.64%	7.62%	**Verbal- 3**

The rate of change for each of these learning styles have been plotted below:

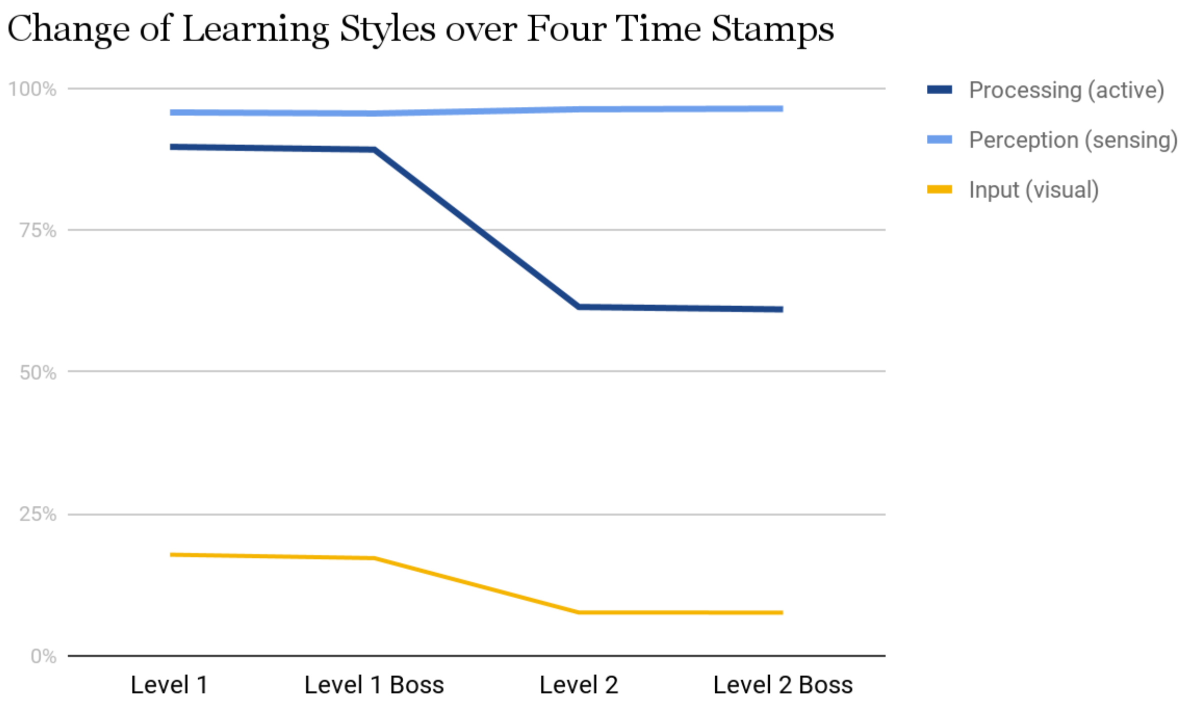


Figure 5.11: Change of Learning Styles over Four Time Stamps in Case Study

In the end, *System Dot* appropriately classified the player as an active learner according to the ILSQ, but failed to match the perception and input learning styles. Since the purpose of this initial playtesting session was to refine the preliminary DBN by adjusting weights for each of the learning metrics (“time to debug”, “code viewed”, etc.) and gauging the relevance and significance of these learning metrics, the misclassification by *System Dot* is normal and necessary to bootstrapping the virtual world. Details on how these weights and learning parameters will be adjusted can be found in the **Discussion** section below.

5.4 Usability Test Results

On top of gathering player behavior data through log files, the eleven users who tested the game were also asked to fill out a brief usability feedback form. A variation of the System Usability Scale (SUS, https://measuringu.com/sus/) survey was used because of its high accuracy rate and popularity amongst reputable software companies. On a scale from 1 (strongly disagree) to 5 (strongly agree), users of *System Dot* rated the following ten questions:

1. I think that I would play this game frequently.
2. I found the game unnecessarily complex.
3. I thought the game was easy to play.
4. I think that I would need the support of another person to be able to play the game.
5. I found the various functions in the game were well integrated.

6. I thought there was too much inconsistency with the game.
7. I would imagine that most people would learn to play this game very quickly.
8. I found the game very cumbersome to play.
9. I felt very confident playing the game.
10. I needed to learn a lot of things before I could get going with the game.

In order to score the SUS assessment, the value of odd-numbered questions were subtracted by 1 while the value of even-numbered questions were subtracted from 5. These new values would then be added together and multiplied by 2.5. The average score of all eleven users were taken into consideration when evaluating the usability of the game.

The average SUS score of over 500 studies was 68. Therefore, a "good" SUS score would be a number above 68. The table on the next page shows the average scores from all eleven users for each question after processing it through the above procedure. The higher the score, the better the outcome of a certain question. For instance, since the first question ("I think I would play this game frequently") is the lowest score out of all ten questions, it represents that most players would not want to play *System Dot* often. On the contrary, since the last question ("I needed to learn a lot of things before I could get going with the game") had the highest score, we could conclude that, relative to most players, there was not a barrier to entry when delving into the game. Overall, though, the average SUS score of *System Dot* was below 68 meaning that *System Dot*'s usability was below average. Insight taken from the usability test will be discussed in the next section.

Table 5.3: SUS Scores

Question	**Score**
1. I think I would play this game frequently.	2.73 / 10
2. I found the game unnecessarily complex.	4.32 / 10
3. I thought the game was easy to play.	4.32 / 10
4. I think I would need the support of another person to be able to play the game.	6.14 / 10
5. I found the various functions in the game were well integrated.	6.59 / 10
6. I thought there was too much inconsistency with the game.	5.23 / 10
7. I would imagine that most people would learn to play this game very quickly.	3.41 / 10
8. I found the game very cumbersome to play.	5 / 10
9. I felt very confident playing this game.	7.05 / 10
10. I needed to learn a lot of things before I could get going with the game.	7.95 / 10
Total SUS Score:	**~53 / 100**

6. DISCUSSION

6.1 Adaptability Component of the Virtual World

Even though the DBN worked functionally, the preliminary assumptions of the dynamic Bayesian network did not accurately classify the player's learning style as expected. With insight from user testing, there are several factors that may have caused this misclassification and homogeneity of data that could be leveraged to improve the DBN and future classifications. After looking at the case study, not all the learning style metrics in the virtual world were integrated well enough to warrant its effective use as a parameter in the DBN. Moreover, not enough emphasis was being placed on data logging the semantic comprehension of programming.

6.1.1 Readjustment of Learning Parameters

A careful amount of consideration was given to the amount of handholding a player would experience by IntelliSense throughout their progression of the virtual world. In the first level, IntelliSense would heavily guide the player through every interaction with a foreign object. As shown through the previous case study, the player hardly used the API to assess the correct syntax of a coding statement because IntelliSense had already nudged them in the right direction. Consequently, it would be advantageous to reduce the weight of the API parameter in the DBN for preliminary assessment of a player's learning style in the first level. Moreover, since most of the level revolves around custom commands like *System.body(Color.BLUE);* or

System.move(Direction.LEFT); with diction that is English-like, it appeared easier for the player in the case study to make that logical memory jump to the right command. Furthermore, almost all the objects in the level required just a line of code to write and comprehend, which could contribute to the high number of quick debugs by the player.

However, the transition to data types in the second level provided the real challenge and introduction to core programming concepts. There was a noticeable decrease in handholding by IntelliSense and increase of multi-line coding problems, which led to an increase of testers needing to take time to problem-solve like the player in the case study. However, the API on average was used minimally, possibly because there were no indicators or IntelliSense notifying the player of updates to the API. As a result, time taken to debug was not as quick as the first level but the API remained closed just as often. The average decrease of the “active” learning style due to the longer times to debug code was about 18.25%. There were no trends observed between the decrease of this learning style and the actual learning style classified by the ILSQ. As a result, the common decline of the processing learning style attribute must be attributed to the vast contrast between the design of the two levels.

The way the virtual world was designed makes it difficult to assess whether certain player behavior influences one of the learning styles. For instance, while the API does provide a visual glossary of key programming syntax and uses, it does not need to be opened too frequently when approaching the “traversing a fully functional world” section of the level. This was shown through the lack of API references during the first level of the player in the case study. Most of the objects in these areas are non-modifiable and serve to demonstrate the computational significance of programming concepts. These parts of the levels are inherently visual and require an intuitive mindset to grasp, but none of the metrics used for the DBN such as number of times API was open or F5 key was hit represent this intuitive acquisition of knowledge.

Several weights need to be adjusted based on the case study observed beforehand. The noticeable difference of the probability that the learner has an active learning style between the first and second level (from 89.71% to 61.49%) indicate that the “time to debug” and “number of objects’ code viewed” parameters do provide a reasonable assessment of a player’s processing learning style. If this trend continues for future levels, we can predict seeing further refinement of this learning style as the player plays the game.

The consistency between both the perception and input learning styles could mainly be attributed to the API component of the game rarely being used. Perhaps if more attention was given to the feature within the game, this metric could be a better predictor of both learning styles. However, because the combination of the limited awareness of the API and the large weight given to the API parameter (over four times heavier than its counterpart metric), it ultimately skewed the data in a homogenous direction. For the input and perception learning styles, the weight of the API parameter should be lowered in relation with the learning style’s other parameters. Additionally, instead of determining the likelihood of a player opening the API through sampling over the total number of code edits, it would be wiser to sample over the total number of modifiable enemies since each modifiable enemy can be considered its own problem.

Readjusting and honing the API's weight could lead to an increased variance in the perception and input learning style. Finally, upon further evaluation of the "objects seen" parameter, the impact should be readjusted to the verbal rather than visual input learning style because a visual learner tends not to open an object's code and view its verbal context as seen through the case study. This amalgamation of changes should lead to more precise and different classifications of multiple users.

6.1.2 Introduction of New Learning Parameters

Furthermore, to allow for a smooth progression of concepts, the "introduction of faults" part of the level introduces a single line or two of modifiable code, but not a completely empty terminal window to work with. For instance, in the second level, once the player figures out the use of *System.activate(#);*, they encounter challenges revolving around activating doors using integer variables. However, instead of writing variables from scratch, the challenges involve manipulating integer variables to satisfy the *System.activate( );* parameter. The player can simply change the integer value of *key* to the number that will satisfy the power line to a specific door through trial and error. The only indicator that they are performing poorly is if there is an overestimation and/or lack of progress through the puzzle. However, the player behaviors tracked primarily observe the syntactic rather than semantic performance of the player.

Unlike a web environment, *System Dot* does not have a discrete way to verify if a player inputted the right code in a certain scenario other than determining its syntactical accuracy. In an adaptive web learning environment aimed at teaching computer science, the system can validate the output generated from the user with a set of test cases. If the virtual world was purely a learning environment, then we could simply validate an individual's code with the correct solution. Due to the infusion of a platforming game on top of the learning environment, however, *System Dot* relies on both how well the player can play the game as well as how their code can get them to certain areas in a reasonable amount of time with limited attempts. As seen in the case study, the outcome is dynamic and cannot be generalized for all the computational puzzles presented in the virtual world. For instance, in the first level, there is a huge enemy blocking the path without the ability for the player to defeat it by jumping on its head. Some players, like in the case study, moved the object to the left, sending the enemy to its doom in a spiky pit and removing the obstacle forever. On the other hand, alternative players decided to move the object to the right, allowing them to progress to the next zone, but still blocking the path to the next door. They then needed to input another command to change the enemy's color and finally pass. In both situations, the outcome was the same-- the players conquered the challenge by thwarting the enemy. However, one situation was handled more efficiently than the other by permitting some of the players to quickly pass the object with one line of code rather than two. Which method of solving the problem was better? According to an adaptive web environment, both cases would have been deemed acceptable and changed the impact of a corresponding learning style similarly. However, in a virtual world, players who moved the object to the left were more intuitive by looking at the bigger picture of the problem at hand.

Based on the oversight of the semantic nature of the gamified learning environment and after a heavy analysis of the case study before, there were several player behavior styles that could provide beneficial parameters to the DBN in future playtesting sessions. For instance, the speed at which the player progresses through the verbal tutorials can indicate whether they are visual (if fast) or verbal (if slow). Additionally, the time it takes for the player to start modifying the code could be compared to a threshold similar to the "time to debug" parameter. If the player opens a terminal window and spends an extended period of time reading and understanding the code before exiting, debugging, or modifying the code, then we can say they are being more reflective; otherwise, active. In puzzling scenarios like the enemy blocking the path in the first level or the order of which to activate power lines in the second level, a "par" attribute like golf can be assigned to these situations. If the player exceeds the optimal amount of times they should be modifying the code, they will be penalized as an intuitive learner. The farther they are from the problem's par, the less they are considered an intuitive learner. Finally, gameplay metrics like the amount of player deaths can act like a semantic error— if the player dies at a key moment, it can negatively affect a certain learning style. For instance, a player death after a verbal tutorial from IntelliSense showing the player how to avoid said death can negatively affect how likely the player is at being a verbal learner. These suggested learning parameters to the DBN would not have been realized without observing the log files of a player's progression throughout the virtual world.

6.2 Usability Testing

The intention of *System Dot* was to introduce programming in a fun and intuitive way while accounting for the shortcomings of current CS education today. The usability testing did not set out to determine if *System Dot* succeeded in teaching programming. Instead, its goal was to evaluate *System Dot*'s effectiveness at integrating a computer science learning environment in a game-like virtual world. According to the below average SUS score of 53, there needs to be improvement with the current design of *System Dot*.

After observing and viewing feedback from players on the usability survey, there are positive takeaways from a player's interaction with the game. For instance, there is a consensus that the way programming concepts were integrated within the virtual world was seamless and creative. When testers were introduced to new concepts, almost all of them believed that it tied together with ideas seen before in the level. In terms of the interface of the virtual world, most testers believed the terminal window was a proper coding environment, but those who had the most programming experience complained about the lack of proper text caret maneuverability (i.e. holding the CONTROL key and highlighting an entire word). When asked how often they used the visual glossary API at the top-left corner of the screen, most of them agreed that they would have used it more if there was additional clarification for the later challenges (hence why it was rarely considered for adaptability). Finally, most testers enjoyed the atmosphere and setting of the game, highlighting the sound design and setting as "really fitting". As discussed before, a virtual world allows the player to physically put themselves into the player's shoes and experience new challenges along with their avatar. Wrapping the learning environment around a computing setting with CS lingo further reinforces that desire to learn and remain engaged.

Nevertheless, let us address the design decisions that may have resulted in the below-average SUS score. The first level of *System Dot* was constructed as an introduction to the game component of the virtual world-- the essentials of player movement, the basics of interacting with a terminal window, and a sneak peek at the system commands. When informally observing the players after the first level, most of them were progressing at a smooth pace and not getting lost. This was as expected because similar results were found in the Honor's thesis user testing of the first level. In that user testing session, most players did not find the layout or progression of the level confusing, but complained about the extraneous guidance of IntelliSense. Therefore, when approaching the design of the second level, there was a conscious decision to drastically reduce the amount of handholding by IntelliSense and employ a more endogenous constructivist philosophy where the learning happens from within the player rather than from the external environment.

Unfortunately, this new design decision made players feel like they were "left in the dark" when approaching these new problems. Without any guidance, they questioned their choices and were not fully confident that their actions would reap the best results. This may be indicative of a problem-driven learning environment, but the lack of any guidance from IntelliSense made the players feel less like they were in an environment to learn and more like they were in a playground with unintentional design choices. For instance, several times throughout one tester’s play session, they kept wondering if the actions they were performing in the “string section” of the game were acceptable. This type of wavering thought process in the game leads to the lack of confidence and need for guidance reflected in the SUS results. Trying to find that balance of handholding and independent discovery will require a continued process of trial and error with players.

7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

Computer science is essential to the prosperity of our computer-centric economy, but not enough attention has been placed on its significance in current K-12 education. Research has shown that shifting our focus from a traditional teaching style to a more problem-driven approach with an emphasis on computational thinking can not only get more K-12 students interested but also prepared for college and advanced topics. Moreover, the recent introduction of adaptive learning environments in the computer science field have also shown an increase in the rate of retention and engagement by elementary and high school students. Therefore, this thesis revolved around the construction and implementation of these solutions into a problem-driven, adaptive virtual world called *System Dot*.

The Felder-Silverman Learning Style Model was implemented due to popularity and success rate with other adaptive systems. Using a dynamic Bayesian network to classify a player's learning style based on this model provided an imperative first step to building an adaptive virtual world. Testing *System Dot* with more than ten users allowed us to see the faults with this approach by delivering insight into how to properly readjust certain learning parameters and introduce new ones. Furthermore, the poor SUS score from the usability surveys continue to emphasize a reevaluation of the virtual world's design from a hands-off to a more guided problem-driven teaching style. In the end, continued effort will be made with *System Dot* to harness its teaching potential in computer science and provide a personalized experience for anyone planning to go into the computer science field. With relentless refinement of the adaptability component of the virtual world and placing the game in front of as many users as possible, *System Dot* can be in the hands of millions of students with an interest in computers.

7.2 Contributions to the Field

The findings of this thesis will redound to the field of computer science and development of educational virtual worlds revolving around computer science instruction. First, this thesis involves a careful application of learning theory-based design elements to a virtual world for teaching computer science to young people. Secondly, it also revolved around the building and testing of a model for detecting learner styles through actions in a virtual world-based educational game. Lastly, this thesis built a foundation for an adaptive learning game in which on-the-fly adaptive feedback and gameplay can be achieved by detecting a user’s learning style, even when they adapt to new situations in the virtual world-based game.

7.3 Future Work

While there are some sections of the virtual world that accomplished the original intention of this thesis, there are several areas that could be improved and measured in the future to make the game more effective at introducing programming.

7.3.1 Finding the Balance for External Guidance

The usability tests have shown that it is difficult to balance how much guidance is given to the player without it being overbearing. With too many tutorials, the player feels underestimated as a learner and becomes disinterested in the learning environment. On the other hand, having little to no guidance throughout the virtual world diminishes a player's confidence and they start questioning the integrity of their experience. The poor SUS score is indicative of that contrast from an environment that holds the learner's hand to one that casts them out into the wild. As a result, more trial and error should be done in the future when designing the presence of external influencers in the virtual world like IntelliSense.

7.3.2 Fixing Bugs

It is impossible to account for all the bugs that will appear in a game due to the infinitesimal ways it can be played without the developer's intention. Hundreds of bugs have been squashed since the game's inception during the Honor's thesis, but many more have been discovered since the usability tests. Bugs ranging from causing blocks to reform on top of the player and shifting their avatar outside the world to defects where players would cease taking damage because they purchased a health upgrade right before they entered a boss fight. Similar to finding the perfect balance between extraneous and absent guidance, iterative testing must be performed on a regular basis to identify and eliminate these bugs in a timely manner.

7.3.3 Introducing Physical Learning Parameters

More attention can be given to the physical nature of the player as they are progressing through the virtual world. For instance, the patterns of their mouse movements or keystrokes can be leveraged to assess a certain learning style. To a greater extreme, eye-tracking software can also be used to assess the gaze of the player as they squirm in difficult puzzles or discover a solution to a perplexing problem. Log files can only get so far when they primarily deal with a rudimentary way the player interacts with the system. Being able to extend to this new dimension allows for a greater refinement of the adaptability component of the virtual world and a more reliable dynamic Bayesian network outcome.

7.3.4 A Neural Network Approach

There was a lot of discussion about how the virtual world's design did not allow for an easy way to track whether a certain player's behavior impacted a learning style. Because of this indiscrete way to measure data, it would be advantageous to take this logged player behavior as input and then the corresponding learning style output from the ILSQ and feed this data into a neural network. A neural network approach was going to be implemented for this thesis, but due to time constraints and a lack of a large amount of data to properly train the network, a Bayesian network was used instead. However, if data can be gathered by more than a hundred participants, a neural network can be integrated within *System Dot* in the future to truly create a machine learning algorithm capable of self-adjusting and self-improving as the player continues to play through the game.

REFERENCES

Alexandra Gasparinatou, & Maria Grigoriadou. (2015). Supporting Student Learning in Computer Science Education via the Adaptive Learning Environment ALMA. *Systems*, *3*(4), 237-263.

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12. *ACM Inroads*,*2*(1), 48. doi:10.1145/1929887.1929905

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology*, *13*(1), 20-29.

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, *20*(1), 45-74.

Brusilovsky, P., & Millán, E. (2007). User models for adaptive hypermedia and adaptive educational systems. In *The adaptive web* (pp. 3-53). *Springer Berlin Heidelberg*.

Carmona, C., Castillo, G., & Millán, E. (2007). Discovering student preferences in e-learning. In *Proceedings of the international workshop on applying data mining in e-learning* (pp. 33-42).

Carmona, C., Castillo, G., & Millán, E. (2008, July). Designing a dynamic bayesian network for modeling students' learning styles. In *Advanced Learning Technologies, 2008. ICALT'08. Eighth IEEE International Conference on* (pp. 346-350). IEEE.

Carter, L. (2006). Why students with an apparent aptitude for computer science don't choose to major in computer science. *ACM SIGCSE Bulletin*,*38*(1), 27. doi:10.1145/1124706.1121352
Chittaro, L., & Ranon, R. (2007). Adaptive hypermedia techniques for 3D educational virtual environments. *IEEE Intelligent Systems*, *22*(4).

Cognitive Apprenticeship (Collins et al.) - Learning Theories. (n.d.). Retrieved November 17, 2016, from https://www.learning-theories.com/cognitive-apprenticeship-collins-et-al.html

Cooper, S., Dann, W., & Pausch, R. (2003, February). Teaching objects-first in introductory computer science. In *ACM SIGCSE Bulletin* (Vol. 35, No. 1, pp. 191-195). ACM.

Culbertson, M. J. (2016). Bayesian networks in educational assessment: The state of the field. *Applied Psychological Measurement*, *40*(1), 3-21.

Dean, T., Kanazawa, K. (1989). "A model for reasoning about persistence and causation". *Computational Intelligence*, 5, (1989) pp.142-150

DesJardins, M. (2015, October 22). The real reason U.S. students lag behind in computer science. Retrieved from http://fortune.com/2015/10/22/u-s-students-computer-science/

Felder, R.M., Soloman, B.A. "Index of Learning Style Questionnaire (ILSQ)". URL last accessed on 10-2017. http://www.engr.ncsu.edu/learningstyles/ilsweb.html

Felder, R.M., Soloman, B.A. "Learning styles and strategies", (2003). URL last accessed on 10-2017. http://www.ncsu.edu/felder-public/ILSdir/styles.htm
Freeman, S., Eddy, S. L., McDonough, M., Smith, M. K., Okoroafor, N., Jordt, H., & Wenderoth, M. P. (2014). Active learning increases student performance in science, engineering, and mathematics. *PNAS,111*, 23rd ser., 8410-8415. doi:10.1073/pnas.1319030111

Gallup (2016). Images of Computer Science: Perceptions Among Students, Parents, and Educators in the U.S. Retrieved from https://services.google.com/fh/files/misc/images-of-computer-science-report.pdf

Gallup (2016). Trends in State of Computer Science in U.S. K-12 Schools. Retrieved from http://services.google.com/fh/files/misc/trends-in-the-state-of-computer-science-report.pdf

García, P., Amandi, A., Schiaffino, S., & Campo, M. (2005). Using Bayesian networks to detect students' learning styles in a web-based education system. *Proc of ASAI, Rosario*, 115-126.

Graesser, A. C., Chipman, P., Haynes, B. C., & Olney, A. (2005). AutoTutor: An intelligent tutoring system with mixed-initiative dialogue. *IEEE Transactions on Education*, *48*(4), 612-618.

Hamari, J., Koivisto, J., & Sarsa, H. (2014, January). Does gamification work?--a literature review of empirical studies on gamification. In *System Sciences (HICSS), 2014 47th Hawaii International Conference on* (pp. 3025-3034). IEEE.

Hauger, D., & Köck, M. (2007, September). State of the Art of Adaptivity in E-Learning Platforms. In *LWA* (pp. 355-360).

Hein, G. E. (2016). Constructivist Learning Theory. Retrieved November 16, 2016, from https://www.exploratorium.edu/education/ifi/constructivist-learning

Keith J. O'Hara , Jennifer S. Kay, Open source software and computer science education, Journal of Computing Sciences in Colleges, v.18 n.3, p.1-7, February 2003

Klašnja-Milićević, A., Vesin, B., Ivanović, M., & Budimac, Z. (2011). E-Learning personalization based on hybrid recommendation strategy and learning style identification. *Computers & Education*, *56*(3), 885-899.

Koulouri, T., Lauria, S., & Macredie, R. (2015). Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches. *ACM Transactions on Computing Education (TOCE)*, *14*(4), 1-28.

Kury, N., & West, G. (2016). *System Dot: Shifting the Programming Paradigm* (Honor’s thesis). Arizona State University, Tempe, Arizona.

Lambert, L., & Guiffre, H. (2009). Computer science outreach in an elementary school. *Journal of Computing Sciences in colleges*, *24*(3), 118-124.

Le, N. T. (2016). A classification of adaptive feedback in educational systems for programming. *Systems*, *4*(2), 22.

LE, N. T., & PINKWART, N. (2015). Bayesian Networks For Competence-based Student Modeling. In *Proceedings of the 11th International Conference on Knowledge Management*.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning computer science concepts with scratch. *Computer Science Education*, *23*(3), 239-264.

Microsoft (n.d). *Building the workforce of tomorrow, today.* Retrieved from https://www.legis.iowa.gov/docs/publications/SD/21103.pdf

Moshman, D. (1982). Exogenous, endogenous, and dialectical constructivism. *Developmental review*, *2*(4), 371-384.

Philip Kerr; Adaptive learning, *ELT Journal*, Volume 70, Issue 1, 1 January 2016, Pages 88–93, https://doi-org.ezproxy1.lib.asu.edu/10.1093/elt/ccv055

Prensky, M., & Prensky, M. (2007). *Digital game-based learning* (Vol. 1). St. Paul, MN: Paragon house.

Scott, E., Soria, A., & Campo, M. (2017). Adaptive 3D Virtual Learning Environments—A Review of the Literature. *IEEE Transactions on Learning Technologies*, *10*(3), 262-276.

Van Seters, Ossevoort, Tramper, & Goedhart. (2012). The influence of student characteristics on the use of adaptive e-learning material. *Computers & Education*, *58*(3), 942-952.

Weibell, C. J. (2011). *Principles of learning: 7 principles to guide personalized, student-centered learning in the technology-enhanced, blended learning environment.* Retrieved July 4, 2011 from https://principlesoflearning.wordpress.com

Williams, L., Wiebe, E., Yang, K., Ferzli, M., & Miller, C. (2002). In Support of Pair Programming in the Introductory Computer Science Course. *Computer Science Education,12*(3), 197-212. doi:10.1076/csed.12.3.197.8618

Wilson, B. C., & Shrock, S. (2001, February). Contributing to success in an introductory computer science course: a study of twelve factors. In *ACM SIGCSE Bulletin* (Vol. 33, No. 1, pp. 184-188). ACM.

Wolf, C. (2003). iWeaver: towards ‘learning style’ - based e-learning in computer science education. In: Australasian Computing Education Conference, Adelaide, Australia. vol. 20

Zaïane, O.R. (2002). Building a recommender agent for e-learning systems. In: The International Conference on Computers in Education, ICCE’02. 55–59.

Zywno, M. S. (2003, June). A contribution to validation of score meaning for Felder-Soloman’s index of learning styles. In *Proceedings of the 2003 American Society for Engineering Education annual conference & exposition* (Vol. 119, pp. 1-5). Washington, DC: American Society for Engineering Education.