California Polytechnic State University

Code Wizard

Senior Project

Cameron Casey
CSC 491/492
Dr. Michael Haungs
7 June 2022

Introduction

As every industry continues to incorporate technology, it is becoming more valuable, and arguably necessary, to learn basic programming skills, even if you are not going to become a software developer explicitly. These skills are used in many different fields today in some form or another. However, these concepts are not a part of the curriculum for most students in the country. Schools have slowly begun to adapt to these changes, but many of the ones that have done so do not make programming classes required.

A common conception is that programming is difficult and confusing and that only people who are very intelligent will figure out how to do it. In reality, programming can be learned in some capacity by everyone. There are video games that are used to teach school children various skills. Personally, I was taught how to type by playing a typing game similar to Space Invaders. The same is true for games about programming. While these games are not meant to replace a traditional curriculum, they aid in introducing and reinforcing learning concepts. There is a wide range of options on the market today, and many of the games that are oriented toward school children are turn-based logic puzzles. That is, the player gives a series of commands to complete some objective, like moving between two points for example, and the character on screen executes them.

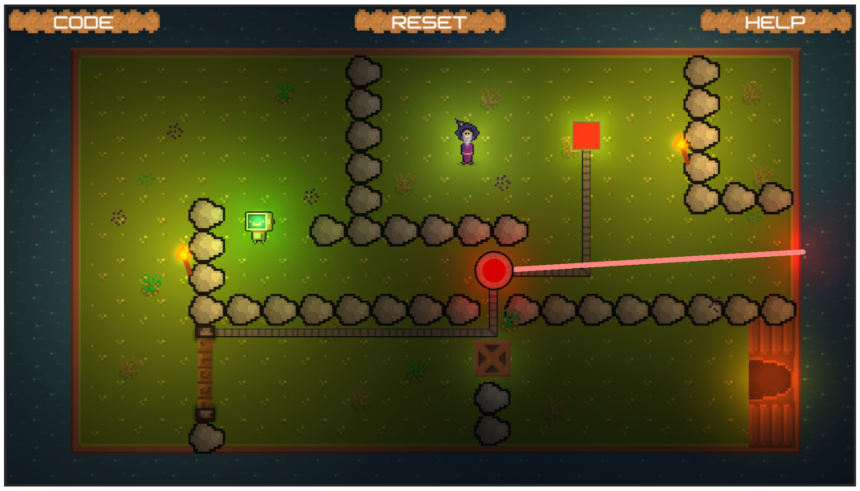
Many of these games may be perceived as slow or boring to a school-age child, especially compared to the AAA titles they can play at home. My game, titled “Code Wizard” aims to keep these players engaged by providing a more fast-paced gameplay loop and by encouraging the player to experiment with the code and engage in the puzzle. This is done by changing the attributes of the objects in the game. It is meant to be a fun introduction to the basic concepts of coding, such as variables and functions. The game will also force the player to think outside the box in a small capacity. When the player learns a concept through their own curiosity and experimentation, the sense of satisfaction and feeling of accomplishment greatly reinforces the concept learned. “Code Wizard” aims to make players feel smart, and by doing so the player is driven to learn more and overcome the next obstacle all on their own.

Application

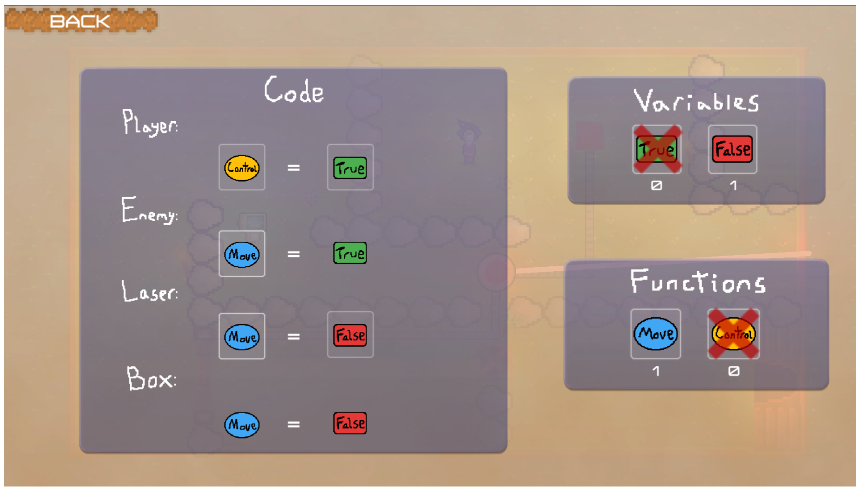
The game can be played on itch.io here: https://ccasey.itch.io/code-wizard

Unity offers various ways to build the game for distribution to different platforms. When the player boots up the game, they are presented with the main menu. From here, the player can press play to start the game at the introduction. There is also an option to play any level using the level selection menu. From here they can also quit the game, closing the program.

The player will first be presented with the game world. The player can see the whole level at once, including the player character, enemy, laser, walls, doors, and the various other objects that constitute a level. At the top left corner of the screen, there is a button labeled "code". This button allows the player to open up the code menu and modify game object attributes. In the center, there is a button labeled "reset" to restore the level to its original state. The "help" button is in the top left corner and it provides an overview of the controls and rules of the game. The player can return to the main menu from here at any time.



In the code menu, there are a few distinct sections. On the right-hand side, there are panels containing extra variables and functions. These are split up by type to make it more clear what tools the player has at their disposal. The rest of the screen contains the code to be manipulated. It is split up by object, with slots for setting functions and their variables. This is done by dragging a block into the code and dropping it in the desired slot. The player is presented with various pop-up windows that explain how the various parts of the game world and code menu work, both here and in other levels where new mechanics are introduced.


BACK

Code

Player:
Control = True

Enemy:
Move = True

Laser:
Move = False

Box:
Move = False

Variables
True 0
False 1

Functions
Move 1
Control 0

Background

A game engine is a software application used for game development. It includes relevant libraries and support programs, as well as providing an environment to streamline development. For the development of this game, I am using the Unity engine. Unity is one of the most popular game engines in the world, especially for smaller indie games and freelance developers. The software itself is free to use, and there are many free assets also available when creating a game. First released in 2005, it was intended to be an editor similar to the movie editing software Final Cut Pro, but for video games.

Some of its most important features include the ability to create and modify scenes, import and add assets to the scenes, and modify the assets in the inspector window. When the user opens a new window, it allows the user to access assets, modify the scene layout, inspect and edit the properties of an asset, and run the game in its current state all in one window. There are also menus for adding UI, shaders, effects, sounds, and a large number of additional features to enhance the game. The user can also manipulate script variables without needing to manually change the code.

Unity is a good choice of engine for many small indie studios and freelance developers. It offers a diverse set of tools for programmers to bring their vision to life. Some of the advantages of unity include multi-platform development and the ability to render in both 2D and 3D, as well as a large community of developers and purchasable assets on the asset store. However, for larger game development studios, Unity is rarely the first choice. A project such as a large AAA game needs a lot of optimization. Licensing assets from the asset store can be expensive for freelance developers. As with any game engine, Unity has its pros and cons. However, many of its cons are not in the scope of this game, so Unity is the best option.

Design

The purpose of this game is to be both a fun puzzle game and a good introduction to the basic concepts of programming. The target audience is high-school-age children and above. The game should be fun and rewarding, with a focus on thinking logically rather than stumping the player.

The modifications the player can make to the code change how each game object functions. The player is able to experiment with the code in a way that does not break the game but produces interesting results that force the player to think outside the box. For example, there is a function labeled "control" which can be applied to any object, allowing you to control it.

The puzzles were designed by reverse-engineering the level from the solution. Each puzzle will have an interesting interaction that is both required to solve the puzzle and teaches the players something new about the game logic. The rest of the level is built backward from there, adding obstacles along the way. Concepts can be introduced incrementally in this way, preventing the player from feeling overwhelmed and guiding them to the solution.

Players will solve puzzles by manipulating the code presented and interacting with the environment. While solving a puzzle, they must swap between code editing and playing in the environment. This requires the player to become familiar with how the code works. The smaller problems in the player's way force them to think of a different solution and learn about the fundamental rules of the game, therefore learning to think logically and become familiar with a few basic coding concepts.

Implementation

Game World:

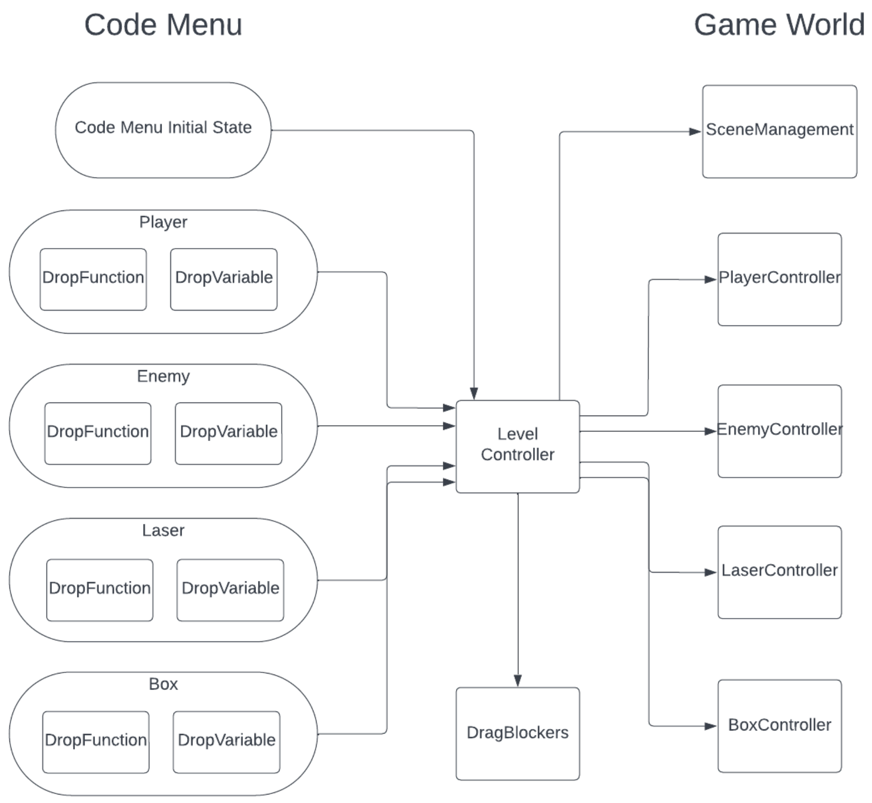
The wizard is the player character, and it is controlled by the player controller script. The enemy, laser, and box have corresponding controller scripts. In each script, there are two booleans that dictate whether the object can move or can be controlled. These flags enable different behaviors in the script. While this leads to a less systemic architecture, this approach allows me to curate the use of each function for each object. For example, the “move” function allows the box to be moved, but when applied to the enemy, it will move from side to side. Both a function and the “true” variable must be present in a given object’s code section for the said function to work.

In the code section of the game, there are two types of variables and two types of functions. These are true and false, and move and control respectively. For one of either the move or control flags to be set in an object script, the function must be set to true in the code section. The communication between the game objects and the code user interface is facilitated by the level controller script. In addition, there are other user interface elements such as buttons for menu swapping and tip panels that teach the player about the game’s mechanics.

Code Menu:

The level controller allows communication between the user interface and game object scripts. In the user interface, there are two types of blocks that the player can interact with: drag blocks and drop blocks, each of which has distinctions between functions and variables. These scripts are modified versions of Unity’s example UI code. The “variables” and “functions” sections of the code menu contain the drag blocks and the “code” section contains the drop blocks. This setup prevents players from dropping variables in the place of functions or vice versa. Limits can be set on the number of variables or functions of each type available through the “drag blockers”.

The following diagram demonstrates the flow of data from the code menu drop blocks, through the level controller, and to the various scripts that affect the game world.



Drag blockers prevent the player from using functions or variables if there are none available. When a drag block is released on a drop block, the type of variable or function is identified by the level controller through a public output variable called “status”. In the level controller, a function drop block and a variable drop block are associated with each type of game object. The status of these two blocks is run through a few simple if-else statements to determine whether the object can move, can be controlled, or neither.

The following is pseudo-code for updating the control flag of a given object. The code for the move function is nearly identical.

```
IF object is not null and a change occurred in the code
    IF function is set to control and variable is set to true
        THEN set object control flag to true,
        set object move flag to false,
        adjust the number of functions/variables available

    ELSE IF function is set to control and variable is set to false
        THEN set both object control and move flags to false,
        adjust the number of functions/variables available
```

Almost all of the variables mentioned above are given to the level controller through private serialized fields. Using one script for communication between objects allows me to keep the data private while setting the corresponding variables in the editor. This approach was convenient for level development but did not scale particularly well. It allowed me to make changes to the UI and level logic on a case-by-case basis for each level, but maintaining consistency between levels became a small hurdle.

Other:

While the user interface and its communication between the game objects comprise the bulk of the implementation, there are many smaller aspects that are necessary for a complete game. The game uses simple 2D point lights to light the majority of the game. In addition, many of the sprites in the game were drawn by myself, except for the wizard, enemy, and environment sprites. The assets that were not made by me were downloaded from the Unity Asset store. This includes music, sounds, and a few other sprites.

Analysis / Verification

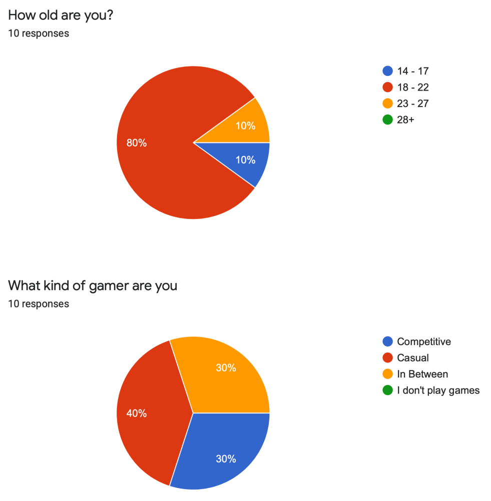
For the process of playtesting, most sessions were conducted over Zoom. Ten people were tested for each iteration. Other than basic information such as controls and the fact that they are playing a puzzle game, play testers were given no prior knowledge of the game before testing. While playing, the playtester was encouraged to think aloud as I observed their behavior in the game. After the player completed the levels, they were asked to fill out a short survey about their thoughts on the experience.

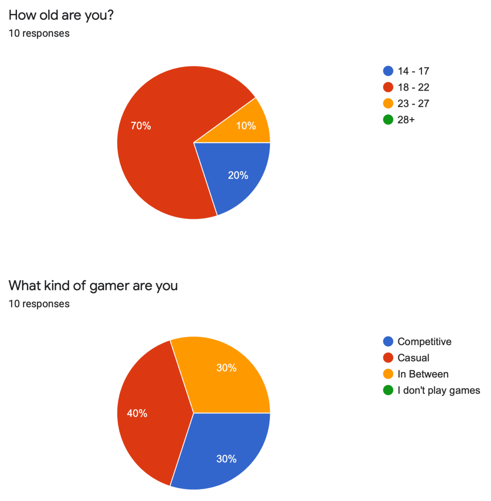
Follow-up survey Google Form 1: https://forms.gle/68bo1iDgr2azcL868

Follow-up survey Google Form 2: https://forms.gle/qrMgrS9vo7SJkbSEA

Testing was conducted in two iterations where the main difference between iterations was the new levels to be tested. During the first iteration, players completed the game in 15 minutes on average. During the second iteration, the completion time rose to 35 minutes on average, which is to be expected. The playtesters were first asked simple demographic questions and a question about their background in coding.

The majority of those surveyed were other college students, and I believe it is important to acknowledge how this may skew my data. Specifically, my peers most likely have more coding knowledge than an average high-schooler, so the puzzles were designed to be easier than the testing data suggests. In addition, most of them play games in their free time, so there is not much data to indicate how successful an inexperienced player will be.



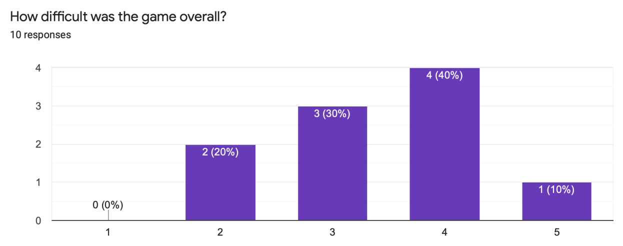


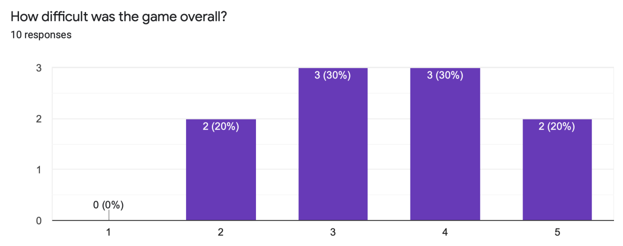
Iteration #1

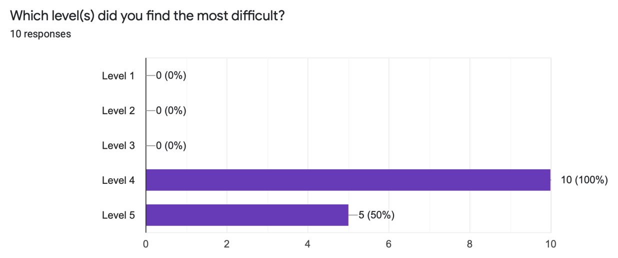
Iteration #2

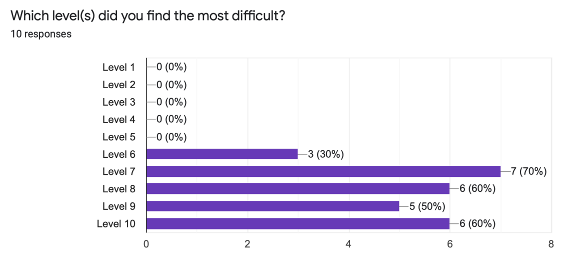
Demographic Data

Playtesters were then asked about their thoughts on the overall difficulty of the game, as well as which specific levels they found the most difficult. Surprisingly, one level stood out as being exceptionally challenging for people. During the first iteration, the most difficult level was level 4. During the second iteration, this level was moved to level 9 because of its difficulty. The second iteration was especially valuable for tuning difficulty, as the first four levels are meant to serve as an introduction to the concepts in the game and are not representative of the overall difficulty.





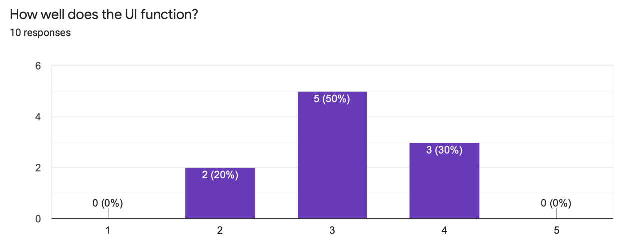


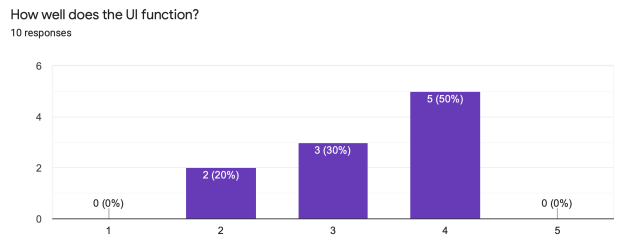


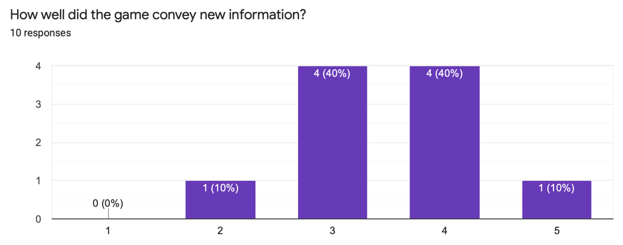
Iteration #1 *Iteration #2*

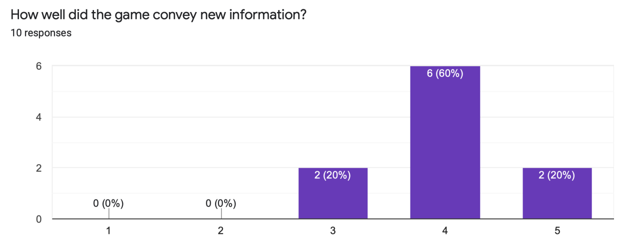
Difficulty

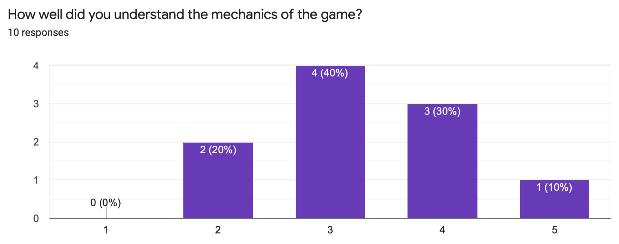
The code section is one of the most integral parts of the game, and it is important that the user interface is clear and easy to use. Playtesters were asked about how well the game's user interface functioned, how well it conveyed new information, and how well the player understood the mechanics of the game. The most suggested improvement to the user interface was the ability to drag and drop blocks within the code section, without having to drag a block from the right side of the screen every time.

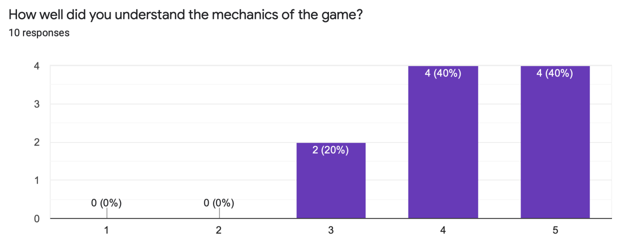












Iteration #1 *Iteration #2*

User Interface

The survey includes a few free response questions in order to collect more detailed feedback from the playtesters. These are meant to highlight any bugs, difficulty spikes, and other issues that are not conveyed in previous questions. This also includes room for suggestions and a section for playtesters to describe their overall thoughts on the game.

The majority of playtesters enjoyed their time playing the game and found it challenging yet fun. The most positive feedback was directed to the concept of incorporating coding into a video game, which encouraged the playtesters to experiment with the mechanics. Unfortunately, swapping blocks within the code section was postponed indefinitely, as I discuss in the future work section.

Related Work

The game can be categorized as a logic game. A logic game is one that gives the player a set of rules and the player has to apply those rules to solve a puzzle. Some examples include Poly Bridge, World of Goo, and Contraption Maker. In a bridge-building game such as Poly Bridge, each available section of the bridge has a cost, strength, and maximum length. The player must balance these aspects in a way that creates a functioning bridge without going over the maximum cost. However, as opposed to the games previously listed, my game will allow more open-ended manipulation of the rules. There are not many games that allow the player to change the logic of the game.

A widely known example of an open-ended logic game is Scribblenauts. In Scribblenauts, the player can type almost any noun they can think of and it will appear in the game world. This is used to solve the puzzle presented by each level but allows the player freedom and creativity in achieving the solution. The open-ended design of Scribblenauts and similar games creates an interesting problem in preventing the player from breaking the experience. In Scribblenauts, many of the puzzles can be solved too easily if the player thinks of an item circumventing them. In my implementation, this pitfall will be avoided by placing some restrictions on the player.

While this means the game is not truly open-ended, it will provide more challenges and prevent unintended solutions.

One of the best examples and main inspirations for my game is the title Baba is You. Baba is you is a 2D puzzle game where the player pushes around blocks with words in them to create sentences and solve puzzles. For example, there may be three blocks in a row, labeled “door”, “is”, and “closed”. For the player to open the door. They must push a block labeled “open” in the place of the closed block, completing the phrase “door is open” and opening the door. My game takes this concept and centers it around programming concepts. To change the logic of the game, the player will edit sections of code to manipulate the objects in the level.

Future Work

There are a few features I had originally planned to include but was unable to implement. As the user testing made clear, the ability to drag a function or variable to another within the code section would be a nice quality of life improvement. However, because the code section can function without it and the gameplay emphasis on logical thinking rather than speed, it was continually pushed back by more important features.

In its current state, the game does not provide much incentive to push forward through the levels. On top of this, the player has the ability to switch between any level instantly, essentially eliminating a sense of progression outside of completing the levels. This would be partially solved with a save system where each level is only unlocked once previous ones are completed. In addition, one person tested said they would like to be able to customize their wizard. Adding progression-based customization options may further incentivize players to continue playing.

With such an emphasis on puzzles and logic, the art, animations, and other effects were similarly pushed back. Overall, there is very little animation in the game and the animation that is present is very basic. Articulated character movements would bring a lot of life to the otherwise static world. In addition to animation, improving other elements such as particle effects, screen effects, lighting, and audio would contribute to making the game feel more responsive and exciting.

Conclusion

The puzzles turned out to be challenging without being too frustrating, which is a difficult balance to strike. The game has a clean and simple art style that looks good and clearly conveys the state of the world to the player. Originally, this game was intended to be aimed toward middle-school-age children. I believed that I could provide a more challenging experience, so I pivoted on this idea after designing the first set of puzzles. I am glad that I made this decision, as I believe it led to a more interesting game.

The game is a little more static than I had hoped. “Juice” is a term used to describe the feedback a game provides. A great example of juice can be seen in any video game gun. The muzzle flash, sound effects, particles, and screen shake make the gun feel like it packs a punch. Due to the amount of attention I focused on the fundamental aspects of the game, I did not add a lot of juice to the game. Improvement in aspects such as animation, particle effects, and lighting would help the game feel more dynamic. I think these effects would have made my game feel a lot better to play.

Despite the work I didn’t get to, I am very pleased with how this project turned out as a whole. I thought the concept itself was really interesting and I am glad other people have given me similar feedback. However, this meant I had to tweak or completely redesign both game fundamentals and puzzles constantly as I tried to mesh coding with a puzzle game. I learned a lot about puzzle design throughout the process, especially how valuable play testing is for a puzzle game. As the designer, there we so many small things that I had overlooked in how people approach each level. People often found creative solutions that I hadn’t thought of but were within the rules of the game. I’m terms of creating a thought-provoking and challenging puzzle game, I believe I succeeded.