ComPuzzle *User Manual*

Pennsylvania Western University of California

Senior Project I: Software Engineering

(CMSC-4900-001)

With Dr. Chen

Abigail Dehart

Brianna Dulik

Brandon Mastin

Cameron McGill

**Instructor Comments and Evaluation**

**Table of Contents**

Project Overview and Application\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

Motivation\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

Comparison to Existing Projects\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_7

Community or Social Implications\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_9

Project Implementation Details\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10

Project Engine Introduction\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10

Code for Command Blocks\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_13

Command Block: If\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_14

Command Block: Loop\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_17

Command Block: Function\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_22

Command Block: Error Checking\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_26

Randomized Non-Playable Characters\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_28

Playable Robot/Car\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_31

Map/Character Scaling\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_34

Asset/Map Creation\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_38

Differences from Design Document\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_41

Challenges During Implementation\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_45

Use of Software Engineering Principles\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_48

User Manual/Usage\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_49

The Game\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_49

Map\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_50

Function Selector\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_50

Command List\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_51

Functions\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_52

Playing\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_56

References\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_58

Glossary\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_58

Appendix\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_59

Team Details/Contributions\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_59

Workflow Authentication\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_60

**Project Overview and Application**

Our project is ComPuzzle, a computer puzzle game including simple coding ideas. We wanted to make a fun, code-related game. This game includes basic pseudocode blocks and a problem-solving aspect that is aimed towards teaching the user how to think in ways likened to simple coding. Pennwest California themes can be seen throughout the game. The player will go through a day at Pennwest California, with a pseudocode/maze twist. Our game is a fun dive into learning coding processes.

Incorporated in our game are many early programming concepts, such as loops, conditional statements, and the ability to create your own functions. Having a visual representation of these concepts in a video game form will allow the user to learn while having fun. This hands-on method will play out the user’s efforts to create a working program right before their eyes. This visual representation of the user’s code arrangement will be displayed as a character going through a maze. The maze aspect invokes a sort of challenge, with the attempt to calculate the exact steps needed to reach your goal through the windy paths. The code arrangement submitted needs to be accurate for your character to land perfectly on the goal to complete the level. The need for exactness can teach the user that even the smallest mistakes or errors can keep your program from running as planned. The user can then learn from their mistakes and ‘debug’ their little program until level completion. Watching your code being played out by a character going through a maze is an entertaining way to step through the basic coding process.

To include more than just repetitive maze maps, we decided to go with a Pennwest California-themed playthrough. Not only will the player be manipulating their character through mazes, they will also be working their way through a day at CalU with every level. We created a little playthrough of what your day might look like when going to class. This includes parking, making your way to class, finding your desk, and many more college themed tidbits. This leveling system will get harder as the levels pass, allowing for user technique advancement, and interest will be kept by including different feelings of accomplishment with each unique map.

Motivation

Our main motivation was to create a game for beginner programmers. A game that we, personally, would have enjoyed when first starting out. When you are first introduced to the world of programming, things can seem quite daunting. You do not know where to start, or what exactly to do to improve. You may be unsure of what concepts are most important to practice. Thus, we wanted to develop something that can teach a younger student the basic ideas of code. Our game allows the user to understand different coding ideas based off pseudocode blocks and also see exactly how it will play out in real time. Using pseudocode supports readability and understanding on this basic level. We believe that hands-on activities are a great way to learn. It is not coding itself, but a visual learning game that will allow the user’s brain to think logically. A game that would have sparked interest in our own beginner programmer minds.

Secondly, games are fun! What computer scientist would not jump at the opportunity to create a video game for their senior project? This video game idea kept us on our feet thinking of what concepts we can implement, what Pennwest California elements we can include, how the game should play along, what would be the best methods for aiding the user’s learning experience, and not to mention the game’s logic itself. For a senior project, we got a pretty lucky choice with a lot of room for creativity.

Comparison to Existing Projects

The number of projects that attempt to teach people how to code are vast. Of these, three stuck out to me. These three are successful in what they do, while all having unique differences from one another and our own project. Seeing these projects and how they all utilized similar concepts implemented with different technicalities was compelling.

Though not a video game as our project is, Scratch is a good example of a very successful endeavor made to help beginners learn code visually. Scratch is a visual programming language developed by MIT Media Lab. It is a much larger and more in-depth project than our own, made most obvious with the fact that it is an entire programming language itself, though there are similarities that are worth writing about. To start, Scratch includes block-based coding. Rather than typing their code, the user must arrange blocks to create their programs. These blocks fit together like puzzle pieces. This is much like our own project, encompassing pseudocode blocks. Our project’s blocks do not fit together as Scratch’s do but are selected and run in a top-down manner. The most major difference is that with Scratch, you can create whole programs and mini-video games. With our game, you may only create programs to complete our maze levels. There is a whole community in Scratch where user’s creations are shared with one another. To begin creating in Scratch, all you must do is open a browser and make an account. You learn the basic ideas of how Scratch’s coding system works first by playing around with the movement of a little cat sprite. There are endless options of what you can make your sprite do, all with real code functions displayed in a simpler manner. Scratch is an impressively in-depth and fun way to begin coding. Despite the clear difference in our project’s size/serviceability compared to Scratch, we follow in their visual learning ideology. We can agree that simple blocks are a great way to represent and teach code to a beginner user.

Lightbot is more similar to ComPuzzle than Scratch, in that it is a 2D, top-down video game controlling a character with blocks. This game is shockingly similar to ComPuzzle, unbeknownst to us when we came up with our idea. Lightbot attempts to teach basic logical thinking with the manipulation of a character. It does this, as ours does, with the arrangement of blocks. While our pseudocode blocks have words, their blocks only include pictures depicting each movement. These blocks can be placed in main to be run. You also have the option of making two separate functions. There is a stop and a reset button, as well as a Total Commands teller. This game is void of if statements and loops, which ours include. Other main differences between our games are the general layout and level designs. If you are interested in playing this game, you can play it easily on a browser. Since it is simply in a browser, and you have no account, no progress can be saved. This cute little game teaches beginners that same sort of problem-solving way of thinking that our game presents.

Cargo-Bot is the last similar project I plan to make comparisons to. While reading up on these various similar projects, I played the tutorial of Cargo-Bot, and knew I had to include it. You can play this game for free on any browser, without the creation of an account. This 2D game encompasses a robotic arm used to move around colored boxes. The control for this arm is through the use of control boxes. Again, similar to our own pseudocode blocks, but lacking words. The controls include moving left, right, or down. There are four different programs, similar to functions, that you can add commands to. They are run in a top-down manner. You can implement a sort of looping system by calling on your four different programs. I find it interesting because not only can this implement looping, but it also dips into recursion by allowing each program the ability to call on itself. There are also condition blocks, similar to if statements. These conditions will check for certain colored boxes and will play out the following commands if true. There are many different levels, each of which teach you different coding concepts. The way this game is played and teaches concepts is quite original and fun.

I enjoyed diving into these three projects. It is interesting and inspiring to see all the many ways our similar ideas can be implemented. These projects and our own want to create an enjoyable environment to learn how to logically think. All three are done using visual and hands-on learning. All of these games succeeded in finding a fun and original way to explain complicated coding concepts to beginner users, something that we hope to achieve, as well.

**Community or Social Implications**

Though we wanted our game to be targeted towards beginners interested in Computer Science, anyone can enjoy it. The coding aspects will attract programmers, but for those playing our game who also happen to go to the Pennwest California campus, you will notice CalU themed pieces throughout our game. Our character is a robot mixed with the Vulcan mascot. A little playthrough including parking for school, walking outside, getting to your desk, and a general day at CalU are seen throughout the game. With CalU colors and pride mixed throughout, your day at Cal will be a-maze-ing. We believe that even non-programmers will be able to pick up our game quite easily allowing for a wide and open community.

**Project Implementation Details**

Project Engine Introduction

We programmed our game in Godot, a gaming engine great for making 2D games. We decided to use Godot’s built in script, GDscript. All drawings were made by our team using Pixelorama. The script, mixed with these imported assets, brought our game into fruition. Following will be pictures of various scripts/scenes and a description on its purpose/how it fits into the whole, including the logic behind each piece. To summarize, the following will include every little bit of how we completed our project.

Starting with the file system, the res:// directory is the working directory for our project. It holds our project in its entirety. Branching from the res:// directory, you will see the folders ComPuzzle Assets, Miscellaneous, Scenes, and Scripts. The ComPuzzle Assets folder contains– you guessed it–all the sprites that make up our game’s graphics. Every button, object, and frame of animation is stored in the assets folder. The script folder comprises of code files that define the behavior/logic of our nodes/scenes. The scene folder holds all of our scenes. Scenes are a collection of nodes organized together to form things such as different maps, characters, menus, etc. and their attributes. Nodes comprise everything.

Nodes are the building blocks of this game engine. Everything in a scene is a node. These nodes are organized in a hierarchal structure. This hierarchy allows for inheritance and GUI layering—such as layering a button on a layer above the map. Scripts can be attached to nodes using signals to define and customize their behavior and allow different nodes and scenes to communicate with each other. These nodes are grouped together to form scenes and provide reusability. There are many different types of nodes of which we utilized: Node2D, Camera2D, CharacterBody2D, AnimatedSprite2D, GridContainer, ScrollContainer, and many more. Here is A screenshot of a computer

AI-generated content may be incorrect.a picture of our game node; the node comprising the setup of our GUI.

A screenshot of a computer

AI-generated content may be incorrect. The game node is a 2D node, used as a base for positioning our sprites, buttons, and maps. All of the nodes branching from the game node are a part of our map/button setup. Camera2D allows these other nodes to be visible on the game screen. The Buttons node is a GridContainer used to arrange our game buttons in rows/columns. The buttons branching from this node are called TextureButtons and display whatever texture we import in. This type of button allows calls to be made when the button is pressed, not pressed, and when disabled. This lets us attach scripts to customize each button. The ScrollContainer node allows for a scrollable area in our UI. In our case, this includes the command list that can be found on the right side of the screen upon a block being pressed. This gives the user the ability to scroll through their instructions. OtherButtons is a 2D node used to organize all the buttons other than the pseudocode blocks. Branching from this organizer are Button nodes. This type allows us to change their text or image, toggle the button to be on or off, check if pressed, or to disable the button under certain circumstances. The ErrorWindow is an AcceptDialog node that creates a popup with title/text of our choosing dueled with the Label. You can click ‘Okay,’ or a customized message, to exit this screen. The FunctionMakerWindow is a ConfirmationDialog node. This node allowed us to create another instance of the previously made buttons to form and save a function that the player can create. The script that brought together our Function maker will be explained in paragraphs following. The TextureRect node was used to make the Victory screen when a level is completed. This node lets you change the texture of the popup to the portrait of your choosing. Buttons can be attached to this node to allow for next level/quitting/replaying. The PeopleHolder node is another 2D node used to hold the character sprites on screen. This includes only the accessory characters, not the playable character.

A screenshot of a computer

AI-generated content may be incorrect. Now that the nodes have had an introduction, I will delve into the script.

These are all of our script files. Our scripts work together with each other and scenes to create a working game with moveable blocks and characters.

The help scripts create pop ups for the help menus of each command. Main is the script for our main menu screen, also holding the saved data for locked/unlocked levels in the user’s directory. Button functionality handles the visuals of our pseudocode button blocks. SpawnButtons is used to spawn the buttons into the command list on the right when the player selects them. Victory is the script for our level completion screen, and the saving of the user’s current and best score for that level. ParentMap is the setup for all maps following. Every map is a branch off of this map’s properties. Function\_maker is the handling of the user’s creation of functions and their saving. Main includes the initializations of the game and everything in it, with a few other inclusions such as retrieving unlocked maps, saving a map when beaten, robot collision, and altogether useful things in the clearing/creation of maps. MapList is simply a list of each map’s individual information; this includes variable names and corresponding scene paths, along with deciphering if each map is a bot or car map. Game includes a lot of functions encompassing which button is pressed, what action that leads to, getting the next tile, clearing list, checking for the goal, and many more. Command\_List is the logic behind the pseudocode blocks. Robot is primarily the logic for the robot’s movement, setup, tile position, and facing.

Now that you have been introduced to our script files and the general use of each, we will jump into various script files to inspect their functions’ logic and functionality. We will look into how our script works together to create a video game.

A screenshot of a computer program

AI-generated content may be incorrect.Code for Command Blocks

This is the processNode function, the core reader of our normal command list blocks. It identifies the command block at the current list index and triggers the appropriate function call to handle it. The function checks whether the current command is an If or a A screen shot of a computer program

AI-generated content may be incorrect.Loop and processes them by calling their respective functions- commadIf and commandLoop. Custom Function’s command reading is dealt with in readFunctionList. For any other command block, doFunc handles the corresponding node. Once all commands in the list have been processed, the function returns the index.

Other than the processing of commands, this function is what changes the color of the command blocks while they are running. During processing, the current node’s texture is changed to a white version of the image. Then, a filter is applied over top of this white image. ‘Color(0, .36, .85)’ is the color filter we chose, with RGB values, respectively. Once the node is finished being processed, the image is returned to its original-colored version and the filter removed. For Loops and Ifs, the white image and filter stay on until all node’s present within the start and end node are processed. This function acts as the hub for all node processing, besides the blocks found in a custom function call, whose processing will be seen in the function section.

Command Block: If

This is the function that handles the visual logic of our in-game If statement.

A screen shot of a computer program

AI-generated content may be incorrect.The\_name is a variable that holds the type of the current node. The variable i holds the next block after the current. Child stores the current node that is being processed. elseTime checks if you have hit an else yet, and retSpot keeps track of the original position of the If block. Variable condition is the condition that the player chose, and variable isTrue holds if that condition is true.

Now, the while loop will process the If’s operation blocks while the endIf is not reached. If the process is cleared midway, the process will not be followed out and the list will be cleared. The current pseudocode block’s name is noted as the list is progressed through. The position in the list is returned so that block processing can resume after this spot. It is checked if each child is an Else or an endIf. If endIf is found, the If is completed and the loop exited. If Else is found, the blocks within the Else and endIf are processed. If the If is found, the blocks within the If are processed. The index is incremented, and the list is moved through. The next spot is returned. This while loop is run until all nodes within the If/Else and endIf are processed. Every node is checked to discern what it is and what action needs to be called until the endIf is found.

A screen shot of a computer program

AI-generated content may be incorrect.This function here is the more backend version of the If block. This section makes the correct function calls depending on the conditions it takes in. It is checked if the condition is true and then decides whether to run the If-then or Else section of code blocks. The index is set to the block following the If, so it can begin checking through each node in the If’s list.

A screen shot of a computer program

AI-generated content may be incorrect. This is the section of code that tests the chosen condition. There are three options for the if statement conditions: if there is a wall, if empty, and if there is a person/car. Whatever condition you chose for the If statement will be checked to see if the following tile in front of your character’s facing holds that data.

This allows the If or Else to then be run depending on if the tile in front of your character holds the condition’s data or not.

Command Block: Loop

This is the function that holds our visual Loop logic. It visually loops the commands that are between Loop and endLoop the chosen amount of times.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

The variable retSpot serves as an index for the Loop node, while totals represents the total number of Loop iterations the player has chosen. loopsLeft tracks how many iterations remain, and innerLoop is used to determine if there are any nested Loops. The if statement checks whether the player has input a valid number in the Loop counter. If the input is not a valid number, an error message is displayed. Body\_start and i identify the node immediately following the Loop block, allowing the Loop's inner actions to be processed.

The outer while loop runs for the total number of iterations specified by the player, as long as the command list hasn’t been cleared midway. The inner while loop processes the blocks between the Loop and endLoop nodes. After each iteration, the Loop counter is updated, and the next node after the Loop is checked for the endLoop label. The loop continues to process each node with the processNode function call until the endLoop is reached. The outer while loop repeats until the total number of iterations is completed, updating the visual counter and index. If the Loop's total reaches zero, the process jumps to after the endLoop. Finally, retSpot returns the new position in the node list, allowing the realLoop() function to continue processing. A screen shot of a computer program

AI-generated content may be incorrect.

CommandLoops is like our Loop interpreter that runs the specified commands found between Loop and endLoop the chosen number of times. The parameters include the command names (I.E. Loop, endLoop, walk forward, turn, etc.), index, the index of Loop in the array, and count (how many times chosen to loop). ‘Retspot’ stores the position of endLoop so as to know where to continue the block’s processing after the Loop is exited. The outer while starting at line 631 will run while the index is less than count(the user’s input of how many Loop run times) and while the list is not cleared. This brings us to the inner while loop, which will process the command blocks while the list is not endLoop and not cleared. Line 641 moves through the command list one by one, the if statement below again checks if the command is equal to endLoop. If it is, the Loop will be broken out of and the position after the endLoop block will be stored. If it is not endLoop, it will be processed by the processCommands A screen shot of a computer program

AI-generated content may be incorrect.function call.

This is the endLoop function, which works to find the correct matching endLoop for each Loop. This is especially helpful in the case of nested Loops. It was important we made sure that this piece was functional. Consider if there were two Loops present, with one Loop and endLoop present in between another Loop and endLoop’s blocks. It would be a very possible outcome that the first Loop block would attach itself to the second Loop’s endLoop, stealing it from the inner Loop. This would completely destroy the user’s ability to have nested Loops. It would entirely break the correct looping structure, skipping over command blocks.

The parameter num represents the index of the starting Loop node, while the\_name stores the label of the current node. The totLoop variable tracks the total number of Loops present. The while loop, starting at line 283, continues to iterate until it encounters an endLoop or until the total number of Loops reaches zero and the list has not been cleared.

Next, the child variable holds the node immediately following the first Loop block, and the label of this block is assigned to the\_name. The index counter i is then incremented to move on to the next block for processing in the subsequent round. If the current node is another Loop block, the totLoop counter is increased by one. If an endLoop is encountered and the totLoop counter is zero, this indicates that no open Loops remain, so the position immediately after the current node is saved in retSpot. If an endLoop is found and the totLoop counter is not zero, the counter is decremented, and the function continues scanning through the remaining nodes.

Consider the case where there are two Loops, one nested within the other. We begin by entering the while loop upon encountering the first Loop node, with the totLoop counter initialized to 0. This is valid because we are not yet on an endLoop, so the original while loop condition is still satisfied. As we proceed, we examine the next node, which is the nested Loop. Since we encounter another Loop, the totLoop counter is incremented to 1, indicating one level of nesting. The Loop continues, and we examine the next node, which is an endLoop. At this point, we check the if condition: since totLoop is greater than 0, we know this endLoop corresponds to the nested loop. We then decrement the totLoop counter. The while loop continues, and the next node is found. Suppose this node is another endLoop. Now, since totLoop is 0, we know this endLoop marks the conclusion of the original outer loop. The position immediately after this endLoop is noted. At this point, the while loop condition is no longer satisfied—there is no endLoop left, and totLoop is no longer greater than 0. This means that all nested loops and endLoops have been processed in the correct order, and normal node processing can resume.

Command Block: Function

A screen shot of a computer program

AI-generated content may be incorrect. The doFunc function in our game is for the Function creator block. When the user clicks this block, they can choose from their previously made Functions to run in the command list, or make a new one. When making a new Function, a temporary version of our game environment is created. This allows the player to try out any blocks and sort of preview what they will do.

Match acts like a switch-case statement in Godot. Every possible character movement block is checked for and the character is moved/turned accordingly, just as the normal character walking mechanics work. This is accompanied by the command blocks’ color change when active. Additionally, it also includes the random student’s/car’s randomized movement call. In the usual game logic and the Function’s separated version, we perform these specific actions to keep them separated. This is necessary in the case that a Loop or endLoop command will be wrongly sent to this function. In this scenario, we do not want to execute any actions for these blocks. By isolating these actions in distinct cases, we ensure that only actions relevant to each case are taken. Additionally, we check for ‘if child’ after any ‘await’ because the list can be cleared/erased at any moment. Without these checks the game would crash.

A screen shot of a computer program

AI-generated content may be incorrect.

This is the processCommands function, used to process the Ifs and Loops present specifically in a Function. It reads the name of the command block that the index is currently pointing at and performs the proper function call to handle the node. Checks are made to decipher if the current command is an If or a Loop. They are processed accordingly by calling their separate functions. If the command is any other command block, the commandDo function is called and the corresponding node processing is performed. The index i is returned after all the commands present in the list are processed. This function acts as the hub for all node processing present in a custom Function.

A screen shot of a computer program

AI-generated content may be incorrect.

This function is responsible for saving the custom Function that the user created. This encompasses the name and included commands chosen by the user. It first opens the file that will be written to. It then reads each child node of the Function created and gets their names. These nodes will then be written to the file. If there are extra parameters, such as Loop’s count or an If condition, it will save the number of Loops/condition type to the file. This is all saved to the user’s directory in the LIST.txt file.

A screen shot of a computer program

AI-generated content may be incorrect.

This function, readFunctionList, reads from the LIST.txt file that the user’s Functions are saved to. It will read the file of the chosen Function and sets the nickname. It will then create an array of the included commands present in the Function. This array of commands will be called to run upon being chosen and the run button being pressed. \_on\_function\_maker\_nameconfirmed and readFunctionList are both one piece of our saving system. As long as the player does not start a new game or delete their Function, these saved Functions will be present. Every Function made can be loaded and played on any level, not exclusively the level you made it in. If the user quits off the game and decides to continue later, the previously made Functions will still be ready to load.

Command Block: Error Checking

A screen shot of a computer program

AI-generated content may be incorrect. This function, validate, is what checks for errors in the user’s list of commands. This is a very important piece of knowledge for the user’s learning. It helps them to learn how important syntax is in programming, and how it is necessary to follow the structure and rules of a language. This function will read through the user’s entire list of commands. There are Boolean flags to track these errors. While the commands are not running, the syntax analysis will commence. The command blocks are looped through, and the label of each node is fetched. If it is a Loop, the Loop counter is incremented. If the node is an endLoop but the Loop counter is 0, there is an endLoop before a Loop and that error flag is set. If it is an If, the If counter is incremented. If it is an Else, and there is not an If previous to the Else, the outElse flag is set true. If it is an additional Else without its own matching If, the extraElse flag is set. If the command is an endIf, it checks if an If is currently open using the If counter. If no If is open, endBeforeif is set to true. If everything is good for the Loop/If and they have their corresponding End pairs, their counters are decremented.

A screen shot of a computer program

AI-generated content may be incorrect. Next, once all command blocks have been checked, the error flags are analyzed. Each true flag emits its own unique signal to match its error type. These errors signals are emitted to the showError function, with the corresponding text to signify what type of error it is. If there are no errors, realReadList is called and the commands can be processed. Command blocks cannot be processed until no errors are present.

A screen shot of a computer program

AI-generated content may be incorrect. This is the show\_error function. It handles the popups for each error message. Not all error types were processed in the last function, some were processed in their own command block’s function. Each respective if checks for the various error messages to see if these signals have been sent. If any error is found, the errorLabel text is received and a popup errorWindow will present upon the user trying to run their invalid list of commands.

A computer screen with text and images

AI-generated content may be incorrect.Randomized Non-Playable Characters

Above is the function responsible for spawning in our randomized students/cars into each map. There are arrays holding each student/car sprite. The car/student sprites are randomly chosen from this list depending on whether it is a car map or a student map. These sprites spawn wherever we place their corresponding custom tile spawn point. Once created, the characters will be added into the PeopleHolder 2D node, allowing them to be visible and interact with the game. The character’s scale and movements are determined by the call to the SetMapStuff function, allowing these characters to scale correctly to each different map size.

There are five randomly selected options for these character’s movements found in the movePeople function. These movements are stored as numbers in an array and are randomly chosen and subsequently matched for the character to take its action. The options for character movements include turning right and taking a step, turning left and taking a step, turning left twice and taking a step, taking a step forward, or doing nothing.

A screen shot of a computer program

AI-generated content may be incorrect.

The peopleWalker function below handles how the characters other than the robot move on the tile map. The direction the character is facing is retrieved, and the next tile in front of them is checked. It is determined if this tile is a wall, object, or a goal, in which case it is not walkable for these NPCs. If the characters could walk on the goal, they could potentially block the goal from the user, delaying their winning indefinitely. It is next checked if there are other NPCs on that tile or if the main character is, in which case it is not walkable. If it is walkable, the characters will take their random action, if it is not, the character will wait and check the next time around. Each randomized movement is matched with one movement per the main robot character. This lets there be some uncertainty if a character will end up in front of you during a turn, allowing more usefulness for an if statement to check for people.

A screen shot of a computer program

AI-generated content may be incorrect.

Playable Robot/Car

The code that goes into our robot/car character is focused on its movements throughout our grid/tile layout, the speed in which it should move, and the collision between the robot and the sprites throughout the map. Important code snippets from robot.gd are found below, with the rest summarized here. The \_ready function initializes the scale of the robot’s animated sprite and captures the values that depend on the scale per map. This ensures that the robot is sized to fit each map, in other words the sprite will not look out of place and calculations will be performed correctly. The \_turnLeft and \_turnRight functions handle the robot’s rotation. A match statement is used to change the robot’s compass direction (north, south, east, west) based on its current facing. These functions update the robot’s animation based on this facing. There is only one horizontal animation model which is flipped depending on whether facing east or west.

A screen shot of a computer program

AI-generated content may be incorrect.

The walk function, as seen above, moves the robot based on its current compass facing. It calculates the target position for the next step and creates a tween to smoothly move the robot to that position. The tween is adjusted based on whether the robot is moving vertically or horizontally. A tween is a built-in function used to create smooth transitions/animations easily, taking care of animation timing for us. Without the tween, our game would not appear as polished and movement animations may appear unsmooth, or to be flickering between changes.

A screen shot of a computer program

AI-generated content may be incorrect. The little chunk of code below was taken from robot.gd. It extends the CharacterBody2D, connecting movements and speed declarations to the robot animation node. The initializations include the speed the robot will move through the x and y axis. This speed is important for ensuring the robot moves smoothly along. The TileSize is set to 32. The VERTICAL\_TILE\_DISTANCE and HORIZONTAL\_TILE\_DISTANCE are also set to 32, this is the distance that the robot will take per one walk command. This sets the robot to take one tile per walk, allowing for the user to be able to count tiles to configure their pseudocode accordingly. The robot’s spawning position is facing south, and it is default to be stationary. ‘@onready var anim: AnimatedSprite2D = $AnimatedSprite2D’ initializes the ‘anim’ variable which stores the robot’s animated sprite pertaining to movement. The ‘$AnimatedSprite2D’ syntax gets the child node, AnimatedSprite2D, from the scene, linking this variable to movement animations.

A screen shot of a computer program

AI-generated content may be incorrect.

This is botcontrol, it is responsible for the bot’s interaction with objects and the footprints/tire marks following movements. First, it gets the next tile in front of the character from parentmap.gd. It compares the name of what is on the tile with wall, object, or player array to check if the tile is empty. If there is a wall/object/person in the tile, this tile is not walkable. If it is none of these things, the tile is empty, and the bot can walk forward. The robot’s walk is called from robot.gd to allow a step forward. For feet prints/tire marks, the robot’s facing is gotten to display the correct direction. While the character moves, the correct facing marks are laid on the footprint layer, a layer separate from the background/collision layer to allow for easy attachment.

Every time a level is started, a next level loaded, a reset is performed, or the robot spawns/respawns in general, a new instance of the robot is created and the old one removed. The ‘deleteRobot()’ function frees the robot from memory when called, deleting the robot from the scene.

Map/Character Scaling

A screen shot of a computer program

AI-generated content may be incorrect. This is our Parentmap.gd script. It manages and interacts with grid-based maps, it is the parent to all other maps we have made. It sets the data for all maps to come. Extends TileMap indicates that the script will work with grid-based maps where each cell in the grid is one tile. Start\_tile and end\_tile are the tiles for the robot’s spawn point and goal. The \_ready function is for when the scene starts, the find\_start\_tile() function is called to spawn the robot. \_process is a built-in function that runs every frame in this script. getTileData takes a Vector2i and retrieves and returns the data at that cell position. getTileType takes the data from getTileData and returns the tile type at that cell position. Custom data Booleans are created for each tile type (Walls, Objects, Wanderspawners, and Goals). We can set these customizations for each imported asset. This allows us to easily choose which objects have collision and which do not. This function helps return the specified tile’s type, the objectType, which is then returned.

A screenshot of a computer program

AI-generated content may be incorrect.This is the rest of the script found in ParentMap. The find\_start\_tile function iterates through all of the used cells of the tilemap using get\_used\_cells, which returns all cells used in a map creation. For each and every cell, it retrieves the custom data associated with the tile using get\_cell\_tile\_data. It then checks if the tile’s data includes the ‘Start’ customization. If so, this cell is returned as the starting point. If there is no starting tile, the game will crash. Find\_end\_tile works similarly to find\_start\_tile, but with searching for the goal tile instead. Again, if there is no end tile on a map, the game will crash when you try to run it. The last two getter functions, get\_start\_tile and get\_end\_tile, call the previous two functions to get the starting and finishing tile’s positions and returns them.

This is our basic map setup. DefaultX and defaultY are the starting points/offset of our map. DefaultTileX and defaultTileY comprise the space each tile will take up, 32x32. These tiles may be stretched depending on how we size the width and height of each map. This means that we A screenshot of a computer program

AI-generated content may be incorrect.import our 32x32 images for map creation, but these images are stretched a little when in the map. Our car maps are scaled to fit images 96x96.

A screen shot of a computer program

AI-generated content may be incorrect.  
 This is the function that scales our map and map items. ScaleX and scaleY determine the scale of each individual tile’s size compared to their original size (which is 32x32 by default). They are our scalers. These scalers are variables that are set depending on if the map is a car map or bot map, as our bot maps encompass tile sizes of 32x32, and car map tiles are 96x96 in size. Anim.scale handles the animation sprite’s sizing so that it fits into the scaled map properly. Since your character needs to take steps per tile, it is important that it spawns in each map correctly so as to not mess up the maze path. These VERTICAL\_TILE\_DISTANCE and HORIZONTAL\_TILE\_DISTANCE variables ensure that each one step corresponds to exactly one tile in size. These calculations ensure consistency across differently scaled maps. SpeedX and SpeedY determine the speed of animation movements. The values calculated are scaled to fit each map size to allow for smooth visual movements.

A screen shot of a computer code

AI-generated content may be incorrect.

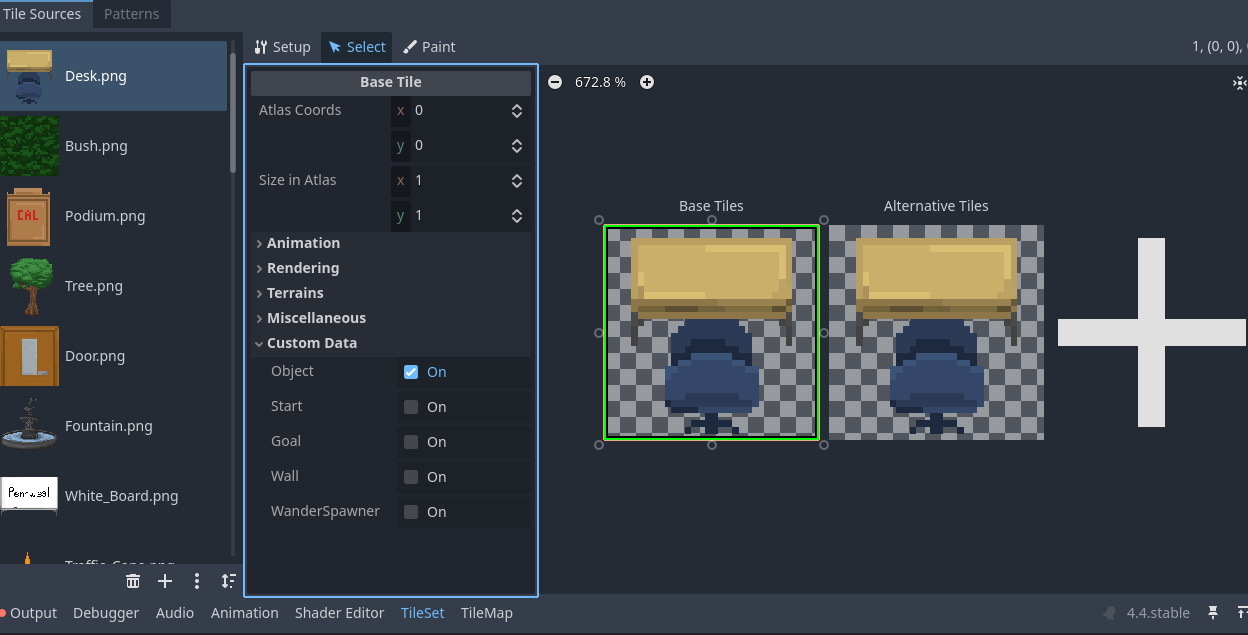
This function is another piece of our saving system, but for maps. It is as straightforward as the function name itself, \_on\_continue\_pressed. When the user clicks continue, the filepath for the .txt file that saves unlocked levels is loaded and read. The .txt file contains a number. This number represents the levels you have unlocked. If it is twenty, you have all twenty levels unlocked. If it is three, you have one through three unlocked. The map you will spawn into upon pressing continue will be based on this number, it will put you in the latest map you have unlocked. In the case that you have already beaten all levels, all levels will remain unlocked, but will spawn you at level one upon continue pressed.

mapList.gd is the script for declaring and managing the selection and model of maps. It is unique compared to others as it is our only global script. This means that it can be called by any other script in our program. It is not in a scene, and it does not have to be. Every other node is in a scene and cannot be called by any script as they are local to their own. mapList extends Node, the parent of all other subnodes. This allows it to use all the functions that a node, being the parent of every other subnode, can do. Every script besides this one has nodes attached to them.

mapList.gd tracks which map is currently active, provides functions to select maps and retrieve their type, and manage the player’s best moves. The map’s scene paths are referenced in a list that is called to produce the called map’s design. The make of each map is also stored in a list and deciphers if it is a bot map or a car map. This decides not only if the character will spawn in as a car or a bot, but is relevant for map scaling as the car sprite and respective assets are all 96x96 pixels compared to the bot’s 32x32. This also deals with what the WanderSpawner will spawn in as, cars or students. Other than these map factors, this script essentially traverses the list of maps and their individual information.

Assets/Map Creation

Continuing with maps, we will now introduce you to our game’s GUI and theming. We implemented Pennwest California pieces throughout the game. There are twenty levels, every four out of these twenty levels are meant to encompass one day at California. Hardness is increased as the days are completed. Levels one through four are equal in hardness, levels five through nine are equal in hardness, and so on. Each day starts with the character parking their car. The next level is getting to the character’s desk in a classroom. The next is making your way through a courtyard. Lastly, the player will find themselves in another classroom. Each day includes these same level designs, but, as stated, is harder than the previous.

 Map creation is easy. Booleans were created to allow the ability to link scripts to our assets. After .png’s are imported in and added to the Tileset, you select the image to change its custom data. Here, you can add collision with Object, create a start tile with Start, a goal tile with Goal, a wall with Wall, and a spawn point for a student/car with the WanderSpawner option. You can create alternate tiles to create separate, individual customizations for the same image.

To create a new map, you simply create another map scene and attach its path to the mapList function in the correct order. Then, you write whether it will be a car or a bot map. After these specifications are made, you can start drawing on any block of your choosing in TileMap. Everything must be placed on its correct layer to function properly. Collision items and spawn/goal points must be placed in the collision layer. Floor items must be placed in the background layer. There is a separate layer, the feet layer, for the overlay of the character’s footprints/tire marks over the background tiles to show where the character has walked.

Included below is a picture of a level found in ComPuzzle, level 6 to be precise, and a larger picture showing our bot/Vulcan character along with the car sprite. Blaze the bot is dressed in Vulcan attire along with the hair and beard. The car is red to follow with California’s colors.

* * **

In this level, the goal is to reach your desk through the maze with the command blocks. The command blocks are colorfully displayed at the bottom of the screen, along with the Menu and Run buttons. Each block has its individual color to add some clarity of which blocks go together for new users. Loop and End Loop are the same colors to show that they go together, they need each other. The same idea goes for If, and End If, along with the optional Else. To the right, you can see the command list. This is the list that will be rearranged, added to, and edited by the user. These are the blocks that will be run in that top-down manner to manipulate the character. There are buttons to easily switch the positions of each block or delete them. The menu and its options can be seen in the lower left corner. The simplicity of our graphics and layout allows for clear distinction between the separate elements of our game. This should allow for an easier understanding of our game’s functions.

**Differences from Design Document**

The differences between our actual game design and the proposed design in our design document are many. Our design document was meant to present us with ways to implement each feature, and what would be included in our game as a whole. We tried to follow these guidelines, but as this semester has continued on, it was obvious that we would not follow the design document to a T. Changes were required for better playability of our game. Other ideas were added/edited to make for a more cohesive game. New ideas sprung, ideas that were preferred to the initial propositions. Problems were also encountered, leading to fixes that were not mentioned in our previous document. The numerous changes that were made will be pointed out and explained in follows.

Firstly, we implemented more pseudocode function blocks than planned. There was never a solid foundation of how many blocks we would have, or exactly what each one would encompass. This was a positive move forward, allowing for more pseudocode to be learned by the player and a solid plan of what each function does. Walking forward, Turning, and Loops were all in the original plan. Now, we have added in the If, endIf, and Else, functions along with the Function block. The If function is straightforward if you are already familiar with if statements in programming. This function will act just as your normal coding if statement does. The Function block will allow the player to create their own function including whatever actions they wish. This will teach the user how to make their own functions, just like in real coding. Both of these added functions are expanded on in detail in alternative paragraphs. Essentially, definitive pseudocode function blocks were decided on and additional ones were added, differing from the limited blocks we discussed in the design document.

Buttons were also changed and added. This is not only including the added function blocks previously mentioned, but Reset was added and the functionality of the buttons changed. Reset is available at all times throughout the program. It resets the character to its starting position and clears the command list of blocks. Functionality was changed when we stated that a drag/drop would be used to choose your pseudocode blocks to create your program. Now, when you click the function blocks of your choice, it will appear in the command list itself. When it appears in the command list, arrows are provided to rearrange the order of the program. These arrows replaced the idea of dragging and dropping to rearrange your placement.

Our game has changed to place more of a focus on Pennwest California theming. Since our game revolves around a classroom as a maze, we decided to expand upon that idea and add in even more Pennwest ideas. The original game character was simply a robot, it has since changed into a Vulcan robot; holding on to our original robot character and mixing it with Pennwest California’s mascot. We have also decided to make the maps even more CalU related by changing the level design to map out a day at Cal. You will manipulate a car to park when you arrive at California. Then, you will go to your desk in a classroom. Next is making your way through a courtyard. Lastly, another classroom level. CalU’s colors and logo will be seen throughout each level. The progression of level hardness is still kept.

Another change is the absence of audio. In our design document, it was proposed that we would include music. We decided against this so that more focus could be placed on the quality of the game logistics as a whole and the art since every asset is hand drawn from our team. The quality of the game and the user’s experience was more important to us than a tune played over it all. At the same time, this absence is accurate to real classrooms, which do not have music in them. To fit the California Campus theme, you will be met with silence, just as you would in a real parking lot/classroom/library.

A change following the decision against audio was the lack of a settings menu. This cut was decided for multiple reasons. The biggest reason being that it would be a seemingly unnecessary addition. We planned to include audio options in this segment along with scaling options. As we are not including audio in our game, there would be no need for an audio option. Screen scaling also became unnecessary as our game will naturally scale to the size of your screen, allowing you to customize the size by dragging the window at its edges. Another thing that could be put into a settings menu would be controller settings. Our game uses so little keyboard/mouse input that it would be pointless to include the option to change these. The character movements rely on the pseudocode, which MUST be selected with a mouse, no changing that. Any other option that may be found in a settings menu can either be changed on the user’s device itself(like brightness) or is so insignificant that settings would be a pointless inclusion.

It was stated in our document that we would include ‘easter eggs.’ We did not define what this would encompass. The basic idea was that there could be little things you could find if you took a certain path in the maze, or completed the level in the least amount of moves, something like a surprise prize you could win. We did not implement this idea primarily due to time constraints. As the deadline for our project grew closer, we decided it would be better to move our focus into polishing up our game rather than adding extra unnecessary things. It was in our better interest to put more work into testing the game mechanics and working on good map designs and assets with the last of our time. The closest thing to an easter egg we have is a hidden twenty-first level, a graduation walk.

Our implementation timeline is slightly different than specified in our document. As the project began, the details realized, the understanding of our project and its timeline grew. In the document, our implementation timeline began in January with ‘Implementation/Design.’ This is accurate, the implementation and design began, no changes. Next, in February, ‘Implementation/Level Design.’ Implementation is true, but level design did not begin until March. ‘Implementation/Design’ is more fitting. March follows with ‘Implementation/Testing.’ I think a more accurate view of this would be ‘Implementation/Level Design/Testing.’ Still implementing new features, while also creating a sort of level design plan, and testing things out as we went. Next to last is April, with ‘Testing/Revisions.’ April includes more than just testing and revisions, the implementation does not stop until the end of April. April more so includes ‘Implementation/Testing/Revisions.’ We are implementing new/better features until we are out of time, to get as much working time out of this semester as possible. We want to make sure our game is as good as it can be, so implementing does not stop until it must. Testing and revisions are also worked through April until we are out of time, trying to find any problems before the deadline. Lastly, May, with a finished program and presentation, is correct. Though this is a small implementation timeline edit from our Design document to now, it felt important to include as the growth of our project led to new realizations and target goals.

**Challenges during Implementation**

During the creation of this project, we had several challenges while creating and designing. One of the first ones we encountered was actually learning the programming language and environment of Godot. It was our first time using a game engine, so it took some time to become familiar with. We spent our winter break learning how to work in the environment by following their tutorial projects and watching videos about Godot. Doing these certainly helped our understanding, but this was an ongoing process/challenge throughout the entire project. The videos/tutorials we watched could not encompass all of the things we would be implementing, not even close. With each feature we wanted to implement came the learning and researching of how. As our understanding grew, old code needed to be replaced and redone as it was realized that better methods could be used.

The next problem we faced was getting the character to interact with the environment. We originally had the robot and the map interact using the built-in functions, such as collision. This had to be changed once we implemented the walking function. We learned that the function we were using to move the character’s sprite would ignore the built-in collision element, so we needed to create our own collision system. We made our walking function so that the character would move x or y pixels depending on the tile size. The walking feature would use this info to look one tile ahead in the direction it is facing and look at that tile’s data. It used this information to determine if it was possible to walk onto that tile or not.

We had a problem getting the robot to know where it was compared to the map. We discovered this problem when switching maps; the robot would sometimes walk through walls or get blocked by invisible walls. That was the first indication of what the problem was, the robot was getting incorrect map data. The cause of the problem was the translation from the global position (the exact x and y value of the robot’s location) to local position (the map’s x, y values). We learned that if the map’s position does not start at global (0,0), then the translation from global to local would not line up. The fix was rather easy, we just made the maps all be at (0,0), then stretch the maps to all be the same size. This of course would lead to another problem, map stretching.

The map stretching was a problem that was difficult to solve. The way we decided to do the scaling was to get a standard/max size in both the x and y direction and divide that by the map’s actual tile dimensions. That would give a scalar in both axes. Then, we use the built-in scale function that will stretch the image by whatever amount you decide. When we did this with the map, we noticed that it would not change the robot’s scale nor the pixel on which the bot would spawn. This would cause the robot to appear on a different tile than it would actually be. Since the scaling only actually enlarges the image, and not the actual tile data, the robot would be in the correct spot, but it would not line up with the map view. To solve this, we multiplied the spawn point by the scalar, which would give the correct location for the robot to spawn on. Once we had the correct spawn tile, we had to also scale up the robot to match, but that was fairly simple. Now, our game will not only stretch to fit any map size we make while completing correct robot placement and walking, but it will scale the game to your preferred screen size.

When we were implementing the Loop feature, we encountered several problems. The main problem we had was nesting Loops. Whenever we nested a Loop and it reached the first endLoop, it would immediately exit all Loops and continue off from the most recent endLoop it found. An added bonus was that it would crash after getting an unexpected endLoop. This problem was originally solved by creating a Loop and endLoop counter. If both counters were equal, the loop could end, if they were not, then it would make it go again. Once an endLoop was seen, it would increase the endLoop counter, and every time a new Loop ran, it would increase the Loop’s counter. This was our first solution. It worked, but as we made more progress, we found a better way to fix this problem. We ended up making a process node function, which would call the Loop function whenever it was requested, and it would create new Loop calls instead of being recursive as we originally had. Once that was implemented, the Loops were working, but still not fully functional.

We continued having problems with the Loop blocks when it was discovered that if they were not terminated, the program would crash at the end of the command list. This was because it would look for an endLoop, and because it did not read an endLoop, it would try to read the next node. It was the same problem you could see in C: reading an array or a file and not checking if at end, and trying to continue to read another element. We solved this problem by not allowing the user to use the Run button if the Loops and endLoops are not equal. We kept track of this by using a counter we would increase when there was a Loop and decrease when there was an endLoop. If the counter was 0, then they were balanced, and we would allow the user to run their program. We would also use this counter to restrict users from adding too many endLoops, there would have to be more than 0 Loops to enable the endLoop block. This solution also worked for when a user would delete a node. Since we do not want to remove anything from the command list or prompt an error if they were just trying to fix something, the counter and disabling the run button worked perfectly for our needs.

**Use of Software Engineering Principles**

Many software engineering principles can be found throughout our entire project. We followed an object-oriented approach; it would be impossible to create this game otherwise. Our modularity can be seen with each .gd file and its included functions performing their own unique purposes. There are separate scripts for our UI handling, game logic, and the input handling. There are also separate scripts for the robot and its functionality, along with a script specifically for the pseudocode block’s logic system. Having everything separated and organized maintains clarity and reduces complications. It also allowed us to have a much easier time building on to the existing pieces and subsequently knowing where to search for bugs in their allocated areas. All of these modules and functions communicate with each other. They are independent pieces working together cohesively.

Inheritance can be seen all throughout our program. We used many built-in classes/nodes from Godot, including CharacterBody2D, Node2D, Node, PopupPanel, GridContainer, and many more. In all of these cases, the script that extends these nodes inherits all of the properties that come along with them. This allowed for a lot of freedom to benefit from this game engine and override its methods. For example, our \_ready() method is a built-in method we override to customize how the robot behaves when the scene starts while still benefiting from the functionality provided by Godot’s base classes. This can be seen in many other cases in our script, as well.

The code for our looping block system used to include recursion. The loop function would call itself for as many iterations as the user input. This way of processing the loop was initially a good handle when it was the only pseudocode block we had. As things grew with the inclusion of If statements, a central process function was seen to be a better fit. This allowed for better reuse and modularity as our program grew. Instead of having a process node in each If’s and Loop’s functions, one was created that could be called in both scenarios.

As everything is in separate modulars with distinct functions, our project could be easily scalable. If we wanted to add on to this game or change features, it would be easy to do so. Everything is in its right place and has its own purpose so adding onto old code or adding in new scripts would be easy to connect with our previous methods. Having a scalable project is the goal, even if we no longer want to add on anything, because that points to having implemented organized and well-purposed code.

**User’s Manual(Usage, etc)**

The Game

ComPuzzle is a 2D character-in-a-maze game, except the character is a robot and is moved with the arrangement of provided pseudocode blocks, and the maze is various places that can be seen on Pennwest California’s campus. The user’s aim is to reach the end goal which will be placed somewhere in the maze. These pseudocode blocks will include functions such as taking a step and turning either way. Other pseudocode blocks will include statements such as an If, Loop, and Function. The user will move around and string together multiple commands to create a program that the character will follow – ideally to the goal.

When you first load our game files, you will be presented with the options of ‘Start’ or ‘Quit’. To continue on to our game, you will click the start button. Here, you will be brought into another menu screen, with the options ‘New Game,’ ‘Continue,’ ‘Level Select,’ and ‘Tutorial.’ Once you click ‘New Game,’ you will be spawned in level 1. This is where you will learn the basics of how to play. You will notice 3 main sections on the screen, the Map your character has spawned into, the Function Selector at the bottom portion of the screen, and the Command List to the right.

Map:

The map is where your character spawns. This is the place to plan out your path to the goal. You will notice that the maps all have tiles placed everywhere. These tiles have certain coordinates on the map. These coordinated locations will be unbeknownst to the user, but they are used in the logic of our script. Each tile is equal to one step for the character; this is a helpful visualization for the user to calculate the number of steps you will need to take to get from the start to the goal. Each map will have a defined starting location (the tile you spawn on) and a goal tile (the tile you want to reach). There may be obstacles throughout the map that will force you to take an alternate path to the goal, to make the game a little harder.

Function Selector:

The Function Selector is located on the bottom of the screen, it holds the functions that you will use to create your program. These functions will be what moves your character throughout the maze. In the Function Selector, you will see ten buttons with labels encapsulating their general purpose. When you click on a function, a copy of that function will be added to the Command List on the right, showing your current program and each block’s order. The Function Selector includes the following buttons: ‘Take Step,’ ‘Turn Left,’ ‘Turn Right,’ ‘Loop,’ ‘End Loop,’ ‘If,’ ‘Else,’ ‘End If,’ ‘Func,’ and ‘Menu’. The Menu Button opens the Menu List, where you get the options, ‘Reset,’ ‘Map Select,’ ‘Clear List,’ ‘Main Menu,’ ‘Quit,’ and ‘Next Level.’ Reset will stop the execution of the program and reset the Robot to the starting tile. Clear List will also stop the execution (if running) and will remove all functions selected. Map Select will exit the maps and bring you to the map selector. Main Menu returns you to the first screen you see when you start the game. Next Level will bring you to the next level if you have it unlocked. Quit, the last one in this menu, will exit the game entirely. The Run Button (far right in Function Selector) will start reading the Command List and will disable all of the Function Selector Buttons, meaning no more function buttons or pseudocode blocks can be accessed while the command list is being read, with the exception of Main Menu.

Command List:

The Command List is where you build your Program. It is located on the right side of the screen. When you click a Function is the Function Selector area, it will be created in the Command List. Each Function will have four unique characteristics when brought to the Command List: an up arrow, a down arrow, a delete button, and an index number. The up and down button will move the Function up or down the Command List. This allows the user to arrange the blocks in any order they please. The delete button will delete the Function from the command list. The index number shows you the placement of each function block within the program list. When run is hit, the command list will read and process the program in a top-down order -- starting at function index 1 and continuing until the end of the list.

Functions:

Our game has eight built in functions as previously mentioned, they are: ‘Take Step,’ ‘Turn Left,’ ‘Turn Right,’ ‘Loop,’ ‘End Loop,’ ‘If,’ ‘Else,’ ‘End If,’ and ‘Func’. While it may seem complicated, these functions are relatively simple.

Take Step:

The ‘Take Step’ function is one of the simplest blocks, but the most critical. This function will make the character take one single step in whichever direction it is facing.

Turn Left:

The ‘Turn Left’ function is another one core to our game. It will make the character turn to **ITS** left, or counterclockwise, whichever helps you visualize it better.

Turn Right:

The ‘Turn Right’ function is the third of the core functions. Opposite to ‘Turn Left,’ it will make the character turn to **ITS** right, or clockwise, whichever helps you visualize it better.

Loop:

The ‘Loop’ function is where things may start getting confusing. Essentially ‘Loop’ will repeat ALL functions that are positioned within the loop block and its ‘End Loop’ block. There will be a small counter (with an up and down arrow) in the top left of the loop function when in the Command List. That number controls how many times you will repeat the functions inside the loop before it exits and continues on to the rest of the Program. If there are multiple loops in your Program, the ‘End Loop’ block that your ‘Loop’ belongs to may get confusing. Each Loop will have exactly ONE End Loop assigned to it. If you put a Loop inside another Loop and BEFORE the first Loop’s end Loop, that is called nesting. When you nest a Loop, you need to end the inner Loop before you can end the outer Loop. Think of it as a Russian nesting doll, you can put one inside another, but the ones inside need the top and bottom parts to be put together correctly. The same thing applies to loops, they (the inside loop) need to be closed before you close the bigger one (outside loop).

End Loop:

The ‘End Loop’ function signals that it is the end of the section of Functions that you want to repeat. The loop’s items will be done running. There is only ONE ‘End Loop’ per ‘Loop’. Each ‘End Loop’ is assigned to its closest ‘Loop’ and cannot be shared. If a Loop already has an ‘End Loop’, then it will attach to the next closest. You cannot have an ‘End Loop’ without a present ‘Loop.’

If:

The ‘If’ function will run all functions between the ‘If’ and ‘End If’ or ‘Else’ ONLY when the condition assigned to it is true. When the condition is true, the functions underneath will be run until the Program encounters either ‘End If’ or ‘Else.’ Once the Program finds one of those two, it will ignore all other functions until the ‘End If,’ where the program will continue as normal. If the original condition happened to be false, then the program will instead ignore all of the functions between the ‘If’ and either ‘End If’ or ‘Else’ where the Program will once again continue normally.

Else:

The ‘Else’ function is the only extra/optional function here, and is used alongside the ‘If’ function. It must be placed within the ‘If’ and the ‘End If’ functions. As previously discussed, ‘If’ will test a condition and give a true or false. If the condition is false, it will ignore all of the functions until the ‘End If’ function was found. There is a caveat to that though, and that is this ‘Else’ function. ‘Else’ is Similar to ‘If’ but will only perform its functions when the ‘If’ gives a false value to its condition. If the optional ‘Else’ is used and the condition is false, then all functions placed between ‘Else’ and ‘End If’ will be run. If the original condition was true, and the ‘Else’ function was utilized, then the program will instead ignore all functions between the ‘Else’ and ‘End If.’

End If:

Similarly to the ‘End Loop,’ the ‘End If’ signals the end of the ‘If’ function. There must be one and only one ‘End If’ per ‘If.’ The ‘End If’s’ attach to the closest ‘If’ so long as that ‘If’ does not already have an ‘End If’ assigned. There cannot be an ‘End If’ without the presence of an ‘If.’

Func:

The Func Button will bring up a menu where you will have the ability to create a function, load a function, or delete a function. When you select Create, a window will pop up where you have the character in a 15x15 room with no obstacles. Here, you can design your function in peace. It works very similarly to the regular command list, but there is no objective. The func creator comes with the nine standard nodes, the run button, and a save button. When you select save, the creator will prompt for a name and a nickname. The name is the full name of your custom function, and the nickname will be a max of seven characters and will be hyphenated. The real name will be the one displayed in the function selector, while the nickname will be the one shown when it is added to the command list. Once you confirm the name, it will bring you back to the selector where you can now select and load it into your program. If you do not name it, it will default to ‘TEMP’ for both full name and nickname. When you have some functions created, you will be able to select the function, then press either the load or delete button. Load will add the function into the command List, while delete will remove that function from the save and it will be gone forever.

Playing:

Now that you have an understanding of the pieces of our game, it is finally time to learn how to play. The objective is to get your character to the goal, which is shown to be either a golden tile or an empty parking space surrounded by yellow lines. You will use the command blocks that were explained previously to control the character’s movements. These blocks will appear in a list to the right upon clicking them, which can be manipulated if need be. This includes rearranging the blocks, deleting any, changing the index on a loop, or changing the condition on an if statement. These blocks will be run from top to bottom. Once you are satisfied with your list you may hit the run button. Upon hitting the run button, each command will be executed one-by-one, you will see the color change to know which block is being processed. During runtime, you cannot edit the list. If you want to stop this execution, you can reset your character or clear the list. You will watch your character follow the commands. Once all commands are processed, a check will be made to see if the robot landed on the goal tile. If you did not reach the goal, you will be reset to the starting location. Your command list will still be fully intact, and you will notice some convenient marks on the ground to show where those commands led you. You can rearrange or add whatever you need to try again, there is no punishment for failing. Once you do reach the goal, you have beaten the level! You will see the Victory Screen, where you can go to the next level, replay the same level, go to the map selector, or quit. Whichever option you choose is fine, because the game will save your progress, and the following level will still be unlocked the next time you load in. As you progress through the game, you will find that the levels will get harder, and the path may be more difficult to get through. You will also find wandering people or cars that can sometimes get in the way. You will have to learn how to avoid the obstacles and navigate the maze. Mastering these levels will help you learn how to think creatively and solve problems just as programmers do. After you have beaten all of the levels, you will be given a diploma and attend the hidden 21st level—graduation! In this level, you will simply grab your diploma and walk in graduation.

**References**

*Cargo-Bot*. (2025). Github.io. https://i4ds.github.io/CargoBot/?state=1

*LightBot*. (2017). Lightbot.com; LightBot Inc. https://lightbot.com/

Scratch Foundation. (2019). *Scratch - About*. Mit.edu. https://scratch.mit.edu/about

**Glossary**

* Pseudocode: A simple representation of code in an English-like, readable form.
* Node: Basic building block in Godot’s scene system. Every element in a scene, such as sprites, buttons, and animations, is a node.
* Scene: A collection of nodes that form a functional unit in the game, such as a character, layout, or button.
* Function/Method: In programming, modules of code that accomplish a specific task.
* Tween: A function in Godot to animate properties of nodes smoothly throughout movements.
* Variable: abstract storage location for a value that is associated with a chosen name, used in programming.
* Increment: Increasing a variable/value by another value.
* Decrement: Decreasing a variable/value by another value.
* Loop: A control structure in programming that repeatedly executes a segment of code until the specified condition is met. In our game’s case, the loop will repeat until the counter has run out.
* Nested Loop: A loop inside of another loop.
* If Statement: A conditional statement in programming that performs the specified function/code when the condition is true.
* Else: An optional part of an if statement that will perform the specified function/code when the original if statement’s condition is not true.
* End If: A block that marks the end of an If statement.
* Match statement: Likened to the switch/case statement in other programming languages, a control flow statement in Godot that checks a variable against several possible values and executes code based on its match.
* Scaling: The process of adjusting the size of the game elements to fit different screen sizes
* Object-oriented programming: Programming paradigm that organizes code into objects, focuses on reusability and modularity.
* Classes: User-defined data types in object-oriented programming that serve as blueprints for creating objects.
* Objects: Individual instances of a class.
* Modularity: Refers to the concept of dividing complex systems into smaller modules.
* Recursion: When a function calls itself from within its own code.
* Inheritance: A concept in object-oriented programming where a class inherits properties and behaviors of another class.

**Appendix**

Team Details and Individual Contributions

Work was completed as a group effort. Ideas, designs, execution, and testing were completed by the ComPuzzle team exclusively. Different sections were completed by each member, these sections being the member’s main focus. While each member had their own section; testing, debugging, and checking the team’s work was completed by all throughout the project’s entirety. Here is a list of each member and the areas they led:

Cameran McGill: Lead Programmer

Abigail Dehart: Lead Design & User Manual

Brandon Mastin: Secondary Programmer

Brianna Dulik: Specification/Analysis & Presentation

Workflow Authentication

I, Abigail Dehart, agree with not only the authenticity of this paper, but of the project ComPuzzle as a whole; The content, design, and development of our displayed work is original and created by our team.

 4/20/2025

I, Brianna Dulik, agree with not only the authenticity of this paper, but of the project ComPuzzle as a whole; The content, design, and development of our displayed work is original and created by our team.

 4/20/2025

I, Brandon Mastin, agree with not only the authenticity of this paper, but of the project ComPuzzle as a whole; The content, design, and development of our displayed work is original and created by our team.

 4/20/2025

I, Cameron McGill, agree with not only the authenticity of this paper, but of the project ComPuzzle as a whole; The content, design, and development of our displayed work is original and created by our team.

A black and white logo

Description automatically generated 4/20/2025

**Writing Center Report**

Met with Caroline Krofcheck on 4/18/2025 at 2:30PM, confirmation email sent to Chen via KRO83903@pennwest.edu.