Computron

Group 32

John Billingham
Brandy King
Nicolas LaCognata
Sean Simonian
Advisor: Tom Carbone

May 3, 2020

Table of Contents

1 Executive Summary

Computron is a computer based puzzle-solving video game that is focused on teaching basic computer science concepts and the process of computational thinking to beginners. There are currently many similar resources available, but they are either too technical for inexperienced programmers, or too abstract for players to apply what they learn to concepts in computer science. *Computron* bridges that gap by having players assemble code-like solutions to puzzles in a simple pseudo-language. The solution is then executed programmatically by an in-game character. This execution is displayed to the player so that they can see the flow of logic and adjust their solutions as needed. By beginning with the most basic aspect of each concept and increasing the difficulty in a stepwise fashion, *Computron* is designed to help players acclimate to increasingly complex algorithmic logic. Further, by providing feedback on solution efficiency, *Computron* also encourages players to improve upon the performance of their solution.

The main objective of this project is to help individuals who are approaching computer science for the first time by introducing them to basic concepts and encouraging them to think programmatically. The layout of the game tutorial and slow but steady increase in puzzle difficulty is specifically designed to coach players through the concepts to ensure that they understand the basic aspects of each concept. This also supports one of the secondary objectives, which is to encourage more people to pursue their interests in computer science without feeling overwhelmed with everything that beginners typically need to learn.

Computron is developed in Unity, and is playable on Windows, Mac, and Linux. The core gameplay loop has players interact with the Computron Assembly Language (CAL) and a selection of datastructures to solve puzzles. The CAL is a collection of 13 assembly-like instructions that allow players to manipulate the elements of the puzzles. The flow of the puzzles was designed with basic concepts in computer science in mind, such as sorting, logical comparisons, and arithmetic. The progression of puzzle difficulty and the order in which each game mechanic would be introduced are based off of course outlines from various beginner level courses, including Introduction to C and Computer Science I. Thorough research and testing has gone into the decisions behind these processes and defining the scope of the project.

2 Project Overview

2.1 Overview

The purpose of this project is to provide a bridge for potential computer scientists to learn and explore the fundamentals of solving problems computationally. Providing this insight is non-trivial, especially in traditional teaching environments. Our approach is to present common introductory computer science concepts in the form of puzzles and visuals in a 2D video game. This game will provide a means for those interested in computational thinking to learn in a fun and engaging environment. The players will not be expected to set up a development environment or learn the nuances of specific programming languages. All of the concepts necessary to solving the challenges should be taught to – or discovered by – the player through interactive game play. Framing this tool as a game is an important distinction, as it allows us to deliberately craft the experience of the player and deliver specific learning outcomes.

Currently, there are many game offerings that focus on teaching computational thinking, with most of them focusing on using "code" solutions for puzzle solving. However, we have found that they tend to fall far on either end of a spectrum; either the game is too abstract and the computational thinking aspect is not well emphasized, or the game is too technical and therefore inaccessible to beginner level programmers. It is our intention to provide a game that falls between these two extremes on the spectrum. We have designed our game to teach the user about different commands and how they can be used to solve the puzzles provided. This will lean away from some of the more abstract puzzle-solvers, where the movements manipulate a character spatially instead of using the sequence of commands to solve the puzzle. Conversely, we will use clear and simple language coupled with visual and audio cues to explain commands and data structures in the game, while abstracting the intricacies of language specific commands or complex details about these elements. This will help new programmers to more easily understand the concepts of computer science and use them to solve the puzzles without requiring them to be familiar with language specific syntax or the complexities of managing data structures in code. For example, we can provide a heap data structure for them to use and explain its general purpose for sorting and storing data without forcing the user to construct or maintain the heap. Taking away the technical aspects of utilizing these elements will allow players to focus on learning and understanding the concepts of the tools they are given and how to utilize those tools to solve the problem presented.

2.2 Broader Impacts

The audience that *Computron* is geared towards are people with very little technical experience relating to computer science and are just beginning to study how to program and learn computational logic. The aim of this project is to create an engaging platform for players to explore the process of approaching and solving problems computationally, build a foundation for understanding the principles of computer science, and prepare users for challenges they may face in their future studies of computer science.

At UCF, the difficulties that surround new computer science students are easily seen in the courses leading up to the Foundation Exam, perhaps most obviously with Introduction to C. Students in that class are immediately faced with two major tasks: learn to solve problems programmatically and learn the structure and syntax of the C programming language. Students often enter the class without any experience with either of these tasks, and many students end up feeling lost and discouraged early on.

Another aspect that should be considered is the unfortunate number of underrepresented groups in STEM fields, especially computer science. There are many factors that can contribute to this issue, such as individual backgrounds, ethnicity, gender, social stigmas, disabilities, or the common lack of preparation from underfunded public schools. Students who are already facing pressures related to these issues may be more inclined to give up on computer science when they feel lost in their first year.

As with any group of technically proficient individuals, it is not uncommon to find that a number of the more experienced members inadvertently reinforce a "gatekeeping" mindset towards STEM majors by supporting weed out classes and survival of the fittest. This behavior can occur with upper level computer science students or graduates who have seen the large number of highly intelligent students who give up on computer science, which could often be avoided if better education and resources were widely available.

For situations like any of those described above, a resource that can provide the tools necessary to understand and approach the concepts and challenges of computer science in an effective manner would be an invaluable resource for many individuals. Our project will provide these tools in an engaging format, enabling players of different backgrounds and experience levels to cultivate an intuitive understanding of computational thinking, while giving them a platform to apply their newfound perspective to approach and solve various problems. As a result, we hope that *Computron* and similar resources help more students of all backgrounds in successfully pursuing their passion for computer science.

While the game strives to close the bridge between logic and syntax, by focusing on the former it can benefit a broader scope of people, regardless of if they plan to become software developers or simply wish to learn the logic of computer systems. The puzzles can act as a means to spark users' interest in computer science, or just to be enjoyed by a spectrum of developers and engineers – from students to industry professionals – as the levels scale in complexity. Introducing a variety of puzzles that are solved using similar methods and mechanics will help players to begin recognizing similar patterns between different problems.

The game will have an engaging and easy to use interface to provide educational value with puzzle-based challenges. The mechanics of the game will be simple, since learning to play the game should not feel like a difficult task on its own. The scaling of puzzle difficulty will allow *Computron* to be welcoming to beginners by incrementally introducing higher complexity tools and abilities as the levels progress. By providing feedback on the basic runtime and space complexities of working solutions, players will have the ability to learn how to solve problems efficiently.

2.3 Personal Motivations

2.3.1 John Billingham

Coming from having no programming experience prior to college, I have seen and experienced the disconnect between learning the correct syntax of a programming language and being able to think and solve problems programmatically. This game has an aim to teach basic logic and computational thinking skills to its users and I believe that this can close that specified gap that many of us have felt.

2.3.2 Brandy King

With career goals of being an Instructor of computer science, I was immediately intrigued during this project proposal. As a Teaching Assistant, I have spent a lot of time working with students in introductory level computer science courses. It is often a difficult subject for students who come in with little to no experience, and I have watched many students struggle with learning a new programming language at the same time that they are learning to use that language to effectively solve problems that they have been presented. I loved the idea of an interactive game that would not only help teach people some of the basics, but also reinforce efficient problem solving with computational logic. I hope that the finished project we aim to provide will help students entering computer science classes by giving them confidence in their knowledge of concepts.

I am also very excited to learn new technologies, work with a team, and experiment with game development. As it currently stands, most of my experience with software development has been strictly course driven. Balancing full time studies, a full-time work schedule, and a family and personal life leaves me with extremely limited free time to pursue personal projects or devote time to learning things outside of lectures. Working on this project will give me the opportunity to expand my knowledge in a new programming language and to workwith a game engine. I have always been enthusiastic about video games, and I see their value as an engaging form of media. Being able to use this platform to convey instruction in computer science is a marriage of two things I am very passionate about.

2.3.3 Nicolas LaCognata

I pitched this project, and I care a great deal about it. I centered my undergraduate studies around games and simulations, and I'm excited to apply my knowledge to create a game that is both highly engaging and educational. There have been many occasions where my desire to pursue a career in game development has been viewed negatively, and I'd like to be able to point to this project as proof that the skills I have learned can provide a valuable experience to others.

The idea for this project mainly came from my experience as a Teaching Assistant for Intro to C. When I took the class, I had prior programming experience. For much of the class I checked out mentally. I drifted through the class without really seeing what was going on. Going back and seeing the challenges new programmers really face was a profound experience. I want this game to help aspiring programmers, and spark interest in our field for those on the outskirts.

2.3.4 Sean Simonian

Like many computer science students, I had very little programming experience and next to zero understanding of most computer science concepts when I first started at UCF, and I quickly found myself struggling immensely with the introductory computer science courses. The main reason I was able to get through the first year of computer science courses was because of friends who were willing to help tutor me in the areas where the lectures were not effective enough at explaining concepts to beginners. In later years, I became a much better computer science student, and found myself in the opposite role as before. I have had the opportunity to serve as a mentor to other students struggling with computer science, and often tutored several people whose professors' teaching styles failed to meet the needs of the students. By experiencing both sides of these situations, I have learned quite a bit about what beginners struggle with when learning computer science concepts, and how these concepts need to be broken down and taught in relatable ways. I am excited about the opportunity to create a tool that can help individuals build a strong foundation for understanding computer science.

Video game development was the primary reason I decided to study computer science, which motivated me to learn to create a video game for my high school senior project, and joined the Game Dev Knights club at UCF during my freshman year. Later I moved away from the goal of game development in favor of interests in software development and cyber security, so this project will provide the opportunity to more thoroughly explore game development in a real setting with a team without having to change the direction of my studies or career path.

My technical experience includes working as a software engineer for Harris Corporation supporting programs for the Federal Aviation Administration, and later working as a software developer for Texas Instruments supporting their internal Data Lifecycle Management project. I have also completed a minor in Secure Computing and Networks, have been an active member of UCF's Collegiate Cyber Defense Club for the past few years, and am currently part of the operations team for Hack@UCF to support the club. Each of these experiences has helped me build various skills that should be useful for this project. Working on this game will provide the opportunity to develop additional skills in areas where I would otherwise be lacking.

3 Legal, Ethical, and Privacy Considerations

Since this project may have real-world impacts, there are a number of potential issues and precautions to address. The fact that this project is not sponsored and is not expected to generate any revenue directly or indirectly lends to fewer potential legal issues than projects that would provide a company with monetary gain. In that regard, there are still considerations in regards to rights of ownership of original artistic work that was created for this project. All of the artwork featured in the game is completely original work done by student artists Conner Ramputi and Christian Babcock. It was created explicitly for use in *Computron* and its use or distribution elsewhere is completely reliant on individual artist consent. All music and sound for the game was designed by team member John Billingham. Its reproduction and distribution is also dependent on his express consent.

Ethical considerations were made for our game and the testing we performed during development. In order to optimize the educational capabilities of the game, we conducted tests on as many unaffiliated individuals as we could throughout the development process. We strongly encourage any future teams that take on this project to do the same and follow best practices when conducting playtesting sessions. If performing any psychological tests, it is important to create and maintain a controlled environment. For the sake of simplifying the process and minimizing liability, we have avoided performing these tests on minors and our target audience does not include minors. Since the target audience for *Computron* is primarily college freshmen with little to no prior experience with computer science topics or computational thinking, we are able to set the target age at over 18, while also expecting that the game will be useful for a number of younger students with a similar skill range.

Our team strongly supports the universal right to privacy for all people, and our product reflects that. No personally identifiable information is collected or stored about users. If future versions of this project include a web component and/or remote database, all data should be encrypted while in transit or at rest. If there is a need to create user accounts and/or collect user email addresses for scalability to verify user accounts, then those will have to be encrypted and stored securely. If ever there is a reason to collect data from users, the required information collected should be minimized and only include what is absolutely necessary.

4 Requirements

4.1 Functional Requirements

1. Be a playable video game
2. Help users learn and understand foundational computer science concepts
3. Have an easy to use 2D interface to encourage beginners
4. Present the user with various challenges and puzzles to solve
5. Provide beginner level explanations to concepts as they're introduced
6. Display visuals to help users understand concepts that are being introduced
7. Encourage user to apply concepts to solve subsequent challenges once it has been introduced
8. Supply hints to user when requested
9. Allow user to see their previous solution attempts for a level
10. Simulate the execution of player's solutions with animated visuals corresponding to the steps
11. Inform player of the accuracy and efficiency of their solution
12. Give generalized feedback on the effifciency and space complexity of a solution
13. Permit user to speed up the simulations
14. Halt solution simulation in the case of:
    - Incorrect Output
    - Runtime Error
15. Support scalability so additional levels can be added later on
16. Interface will be suitable for users with certain common disabilities:
    - Color blindness
    - Deafness
    - Epilepsy
    - Auditory stimuli sensitivities

4.2 Non-Functional Requirements

1. Demonstrate concepts visually with 2D animations simulating the execution of a user's solution to a challenge
2. Allow user to select levels
3. Design a simple pseudo-language with restricted capabilities for the user to solve challenges
4. Interpret the set of commands in a user's solution
5. Solutions must follow interpreter requirements in order to be simulated (similar to compile time errors)
6. Simulation will detect errors and halt immediately if an error is encountered (similar to runtime errors)
7. Store user's solution attempts for a level
8. Evaluate the space complexity of a solution by tracking the data structures used
9. Interface will accommodate users with certain common disabilities:
    - Components in which color is significant to gameplay will use a colorblind friendly palette
    - Components in which sound is significant to gameplay will have corresponding text and/or visual indicators to accommodate deaf users
    - Avoid presenting bright flashing colors/lights that may trigger epileptic users
    - Avoid sudden loud or unpleasant sounds that may disturb users with auditory sensitivities
10. Run on Windows and Mac computers, as well as Linux subsystems
11. Perform smoothly on standard hardware
12. Present efficiency metrics to the player after they successfully solve a puzzle
13. Personally identifiable information about users will not be collected or stored

5 Game Design

As a student sponsored project, we were faced with some very unique challenges. As a team we had to answer two very important questions:

- What exactly is the problem we wish to solve?
- What is our solution to this problem?

We quickly recognized that finding good answers to these questions was imperative to the success of our project. This section of the document will discuss our efforts in answering these fundamental questions.

5.1 Ideas

The beginning of our project was characterized by a great deal of brainstorming. We took great strides to avoid a herd mentality when discussing possible game mechanics, themes, and designs. We frequently separated to come up with ideas as individuals before returning to blend those ideas together into a more cohesive whole.

5.1.1 John Billingham

Abstracting Conditional Jumps:
Conditional jumps run the possibility of bringing complex logic that may overwhelm or confuse the player. While our instruction set is similar to assembly, we would like to minimize the complexity that is associated with assembly logic. One way that traditional higher level programming languages achieve this is by introducing nested if/else conditional statements.

These are essentially the same thing as a conditional jump. A statement is evaluated to true or false and the flow of logic (the next instruction to be executed and those after) depends on this produced boolean value. Nested if/else blocks make this flow of logic far more visually appealing. We must be careful though, by not allowing the player to create additional complexities with confusing nested logic. Limiting the number of conditional instructions that the player has access to can solve this problem. We can also just blatantly limit the number of nests that we allow with conditional logic. It is important that we do not create more complexities while trying to remove some here.

Jump Instructions as a Resource:
Distributing jump instructions as a finite resource can enforce the idea of runtime efficiency for those who play the game. The more jumps a player uses, especially those that are not necessarily needed, the more it may slow down a player's solution. A puzzle level may be able to be solved in ten different ways, but maybe there is a solution that is the fastest and most efficient in terms of runtime and space.

We can push players towards the efficient solution by distributing jump instructions scarcely. A reward system could also be introduced as a means to enforce these concepts. Players with faster solutions (fewer steps/instructions executed) could gain some sort of point based score that maybe allows them to unlock more levels down the road.

Data Structures:
Our initial inspirations and ideas for this game only contain one data structure that is used on the game board: registers. Items can be placed in and taken out of these registers and that is it. Registers support two basic functionalities:

- Item goes into register
- Item goes out of register

Computer science relies on many other equally important data structures like stacks, queues, and heaps. Rather than use just registers as components that players can use on the game board to solve a puzzle, we can bring in other structures that allow them to optimize their solution even further. This would really drive in some core computer science material, something that most educational games of this genre fail to do.

These data structures were specifically mentioned because they also all support two main functionalities like the registers above.

Stacks:

- Push
- Pop

Queues:

- Enqueue
- Dequeue

Heaps:

- Insert
- Delete

Limiting data structures to these same input/output type of mechanisms allows our Actor to easily interact and communicate with these new game components.

Puzzle levels that were solved using a naive solution earlier on may be able to be optimized at a later date after certain data structures have been introduced. We can structure our puzzle level progression system to take this into account.

Atomic Move instructions:
Having the Actor pick up an item and place it down as two separate instructions makes our instruction set feel too much like assembly code. Replacing this with one atomic move instruction abstracts these two commands, making it simpler for the player to understand its use.

The previous functionality is preserved but the steps that a player has to take to achieve it is reduced. This should help players solve levels faster by removing an unneeded intermediate step of work.

The move instruction would contain two parameters: a source and a destination. When the instruction is executed, the Actor moves to the source, picks up the data located there, and moves it to the destination specified by the instruction. This also makes our instruction set seem more like orders for our Actor to follow rather than a straight controlling of the Actor's every move.

Add/Subtract specificities:
Add and subtract instructions would work very similarly to the move instruction mentioned above. Two source registers would be specified, either to be summed or differenced, as well as a destination, where the sum or difference is placed.

The Actor does the animation work of adding or subtracting data and moving it to the correct location. This again removes additional logic that the player has to think through when solving a puzzle.

Color-based indexing system:
The player accesses instructions through a color-based indexing system. Colors represent specific data structures on the game board, while indices represent specific locations within a colored data structure. "Green sub 3" would indicate the third index within the green data structure.

This is meant to give the player more control over the locations that exist on the game board. It is also meant to drive computer science concepts of indexing and accessing different places in memory. We would conventionally use a variable and an index to access certain data structures, but the variable is replaced by a uniquely colored visualization of a data structure.

Return instruction:
Allow the user to choose a data structure to return as their output. The contents of the data structure are checked against the expected output for the puzzle level. The player chooses the data structure they wish to return by choosing that data structure's color in the output game object.

Debugging levels:
The player can step back and forth through their solution, one instruction at a time, and the Actor should support these operations to show a visual debugging process. This visual control can help players find problems within their solution, allowing them to progressively reach the end of the puzzle level.

Broken solution given initially:
A proposed puzzle type where the player must fix a broken solution that is given when the level starts. This can be used in conjunction with tutorials to teach new concepts. This enforces the functionality of certain data structures by having the player interact with a broken data structure being introduced to them.

If a player has messed with and fixed a broken data structure prior to using it in one of their future solutions, our hope is that their ability to use this data structure and think in terms of this data structure when building a solution will be increased.

5.1.2 Brandy King

Many of my ideas for *Computron* came from observing players struggle with games like *Human Resource Machine* and *7 Billion Humans*. I focused a lot on the shortcomings of those games; more specifically, I aimed to fill in what they were lacking in terms of the accessibility that they provide to inexperienced programmers. Additionally, I wanted the game to feel engaging and rewarding for players, something that is often overlooked in the development of educationally focused video games.

Declaring Variables and Data Structures:
When we were discussing the scope of the instruction set we wanted for our game, I put forward the idea of including a "declare" instruction that players would use to initialize a variable or data structure for use in their solution. This would be tied dynamically to the instructions they are able to use, as some instructions are only applicable to these items, like the Move and Copy commands. These commands have a blank field that the user has to fill with an available structure, so if they didn't declare the item, they wouldn't be able to link it to the command.

The hope was that this would reinforce the notion of instantiation of data structures for use in a program. By making the player responsible for managing the data structures, it would emphasize how a programmer is responsible for declaring and utilizing variables and data structures to handle data within an algorithm. In discussing the mechanics of how this would work, it became clear that implementing the declaring functionality as a command would be very difficult to do dynamically, so it was adapted to be its own section, separate from the solution that the player builds. This way, the variables and data structures could still be handled dynamically – both in their declaration and in their attachment to instructions in the solution.

Tutorial Levels:
It became clear very early on that many of the current game offerings in this arena did a lackluster job of providing a solid introduction to their game. Many of the games escalated in difficulty far too quickly for players new to programming to follow along with. They could complete the first few introductory puzzles, but were quickly lost when the game mechanics became more complex or increased in scope. It was evident that having a very solid tutorial aspect to our game was vital to our success in conveying concepts.

I proposed that we intersperse tutorial levels between puzzles in the flow of the game. We already knew that as players progressed through the game they would unlock access to new data structures and commands. As soon as a new item was unlocked, a tutorial level would be the next puzzle in the game. This would be a free-play level with unlimited input and no determined solution wherein the player could experiment with and explore the functionality of their newly unlocked item without the pressure of immediately needing it to solve a puzzle.

In order to “complete” each tutorial level, the player would be required to complete a short to-do list that covered the basic concepts of their new command or data structure after reading a brief description of the function and mechanics of what that item does. For example, a Heap tutorial level to-do list would include:

- Declare a Heap
- Put five inputs in the Heap
- Remove five inputs from the Heap

As each item on the to-do list is completed, it will be checked off automatically for the player. Once it is fully completed, they can elect to exit the tutorial whenever they are ready and move forward to the next puzzle, which will necessarily utilize the new element in its solution.

Limiting Certain Commands:
By limiting the capabilities of certain commands, we can better communicate to players the functions of certain elements in the game. As previously stated, the Move and Copy commands are only useful in conjunction with a variable or data structure. These commands could be in the list of available instructions for a level, but they would be dulled out and inaccessible to the user until a proper variable or data structure was declared.

Similarly, some commands will not be able to be used with certain data structures. For instance, the Add and Subtract commands can be used on variables, but we should not allow players to arbitrarily use Add or Subtract on the top element of a Heap, Stack, or Queue. The player would be required to remove that element from the structure, manipulate the data, and then put the modified data into the structure. This reinforces the core concepts of how these structures are used for storage, and will also encourage players to select the right type of structure depending on how they may need to manipulate the data in that level.

Instruction “Jumbotron”:
In *Human Resource Machine*, as the character is acting out the instructions for the player, there is an indicator in the panel containing the player’s solution that points to which command is being executed at that time. While this is very useful when trying to trace through a solution or debug a problem in the code, it is also a bit too subtle and most users that I observed didn’t notice it at all, even when they were stepping through their solution one command at a time.

From this, I got the idea of introducing a “Jumbotron” element in the User Interface of the puzzle that would show up during execution and broadcast which instruction was being executed at that time. My hope was that the text of the command the user had in their solution paired with the actions of the Actor on the screen would more directly emphasize what action was caused by each instruction. This would, in turn, create an association between the commands and the way they effect the flow of the puzzle, which was something that seemed to be lost to new users who were inexperienced at programming.

Indentation Within Conditional Blocks:
When conditional statements are presented in the games we playtested, there was no indication of which code belonged inside of the conditional block or where that block ended. This is because the conditions had to be controlled with jump statements. However, for users who are not accustomed to scoping and how code blocks work, it is very unclear where exactly a jump statement should go to end a block of code, or even to use a jump statement in the first place.

Using indentation within conditional blocks provides a much clearer picture to the user of exactly what code belongs to which piece of the logic in their solution. This is seen consistently throughout real world programming and coding syntax, where indentation and spacing are paramount to the readability of code. By adopting this convention in our pseudo-language, we can not only set up players for the way real programming languages are laid out, but also communicate to them more clearly the flow of their solution.

Points-Based Scoring of Solutions:
Making an educational game that people enjoy playing is extremely challenging. It is very important that, in addition to being instructive, our game should also be enjoyable. We can foster an early love for computer science by creating a game that players want to return to. This can be achieved by implementing game mechanics that the player can reference to see how well they are doing, and that invite the player to continue practicing their solutions.

A point-based scoring system is a popular device implemented by many games. Scoring a solution out of three stars, for example, clearly indicates to a player how well they solved the puzzle and whether or not there is room for improvement. Simple mechanics like this can be a strong motivator for players to return to previously solved levels and try a different approach in an attempt to get a better score. Encouraging this second look at a solution will stimulate the idea of reevaluating old code and making an effort to improve upon it, especially once new concepts have been learned.

Individual Puzzle Solutions Connecting to Solve Larger Puzzles:
This idea is borrowed from the game *SpaceChem*, which is a game that implements computational thinking to solve puzzles. It falls more on the abstract side of the spectrum, but it has a game mechanic where the solution to several puzzles is combined together as the input for another puzzle down the line.

In relation to our game, my idea was to have individual puzzles be part of a larger puzzle. The efficiency of each of the solutions to three smaller puzzles will dictate how well each of those pieces works together, and ultimately how well the puzzle that relies on those three pieces can be solved. This could be effectively implemented through an overall game theme where each puzzle makes sense to be one component of a larger piece. This idea of nested reliance will reinforce motivating players to revisit puzzles that they have already solved to find more efficient solutions, which can in turn lead to a better solution overall.

5.1.3 Nicolas LaCognata

My biggest inspiration for this project was *Human Resource Machine* by the Tomorrow Corporation. *Human Resource Machine* is a charming little puzzle game that tackles the problem we settled on solving, the gamification of computational thinking. I love the game and find it highly engaging. However, playtests with non-programmers clearly demonstrated the games' shortcomings when dealing with its target audience.

Basic Mechanics:
When brainstorming the basic mechanics of our game, my mind continually strayed towards *Human Resource Machine*. Through playing the game, and watching others play it, it became clear that Tomorrow Corporation's decision to stick with an assembly-like instruction set was an extremely effective mechanic.

The assembly like instructions were simple enough for non-programmers to understand, and easy to compose together to make more advanced puzzles and solutions. Even at the risk of having our game appear as a rip-off, I thought it would be wise to adapt these mechanics.

Data Structures:
After surveying many games in the problem space, we failed to find an example of a game that tackles computational thinking with the aid of Data Structures. Data Structures are such a fundamental part of being a programmer, that their omission in these games seemed incorrect.

Card Game Mechanics:
During our prototyping sessions, I came up with the idea of "Memory Cards". These memory cards would allow the player to declare register locations and data structures before their program starts running. Making the mechanics of placing these memory cards similar to a game like *Hearthstone: Heroes of Warcraft* would be a natural and compelling addition to our puzzle format. This "Memory Card" interface would be separate from the normal instruction writing interface, which helps reinforce the idea

that they are separate parts of the puzzle.

Another benefit of this design is the ability to deliver new instructions and data structures to the player in a familiar (and exceptionally game like) way, card packs. Card games deliver goodies to players in little packages whose contents appear hidden. While discussing adapting card game mechanics, we realized that having that little unboxing moment when introducing new instructions to the player could be highly compelling. Other games in our problem space often silently drop new instructions on the player as they progress. By making the new instructions appear as a reward, we can ensure that the new mechanic being introduced gets the appropriate attention from the player.

5.1.4 Sean Simonian

Much of my inspiration for creating this game comes from my experiences of being in the positions of struggling greatly with introductory computer science concepts, to later tutoring beginner computer science students. Considering common issues that beginners struggle with, as well as feedback and observations from non-technical students playtesting our game prototypes, serves as the strongest indicator of how we need to design our product in order to best serve our target audience. Like the other team members, I drew inspiration from a few games that share similar goals or mechanics as what we envisioned for our game, such as *Human Resource Machine*. As someone with less fondness and experience with the video game industry, I also spent some time focusing on the broader idea of creating a tool to benefit students learning computer science concepts, and I explored ways to pivot the basis for how we could create such a tool. Having real-world experience and passion for both software development and cyber security has greatly influenced my goals for our final product, and I hope to tie some of these concepts together in an encouraging manner for our final product.

Static vs Dynamic Commands:
Certain commands in the language should be static, such as "Read Input", while others should have components that can be selected from drop down menus, such

as “Jump if (blank) (blank) (blank)”. This command would have three sections to specify, so the user could specify this command to be “Jump if x = 0”.

Puzzle Checkpoints:
Break more complex challenges into multiple steps that get checked along the way. Users would be instructed to complete one step of the complete level at a time, run their solution, and if they pass then they move onto the next step, until they work up to the full solution. This would help players grasp more complex concepts that would be harder to teach as a single challenge that they would need to break down on their own.

Example: Merge Sort

1. Split an array in half recursively
2. Compare two of the sub arrays and swap them if necessary
3. Recursively compare and swap sub arrays to build back up to the full sorted array

Testing the Game’s Educational Efficacy:
We could create a set of short questions to be done on paper that require computational thinking. Before a test subject plays our game, have them work through a few of the questions. After the test subject plays our game, have them work through a few more questions from the set. Compare the results to see if they indicate whether our game helps users with computational thinking and problem solving.

Conducting official tests through the UCF Psychology department would be very beneficial if we are able to do so. Students taking courses such as Intro to Psychology are required to spend a certain number of hours as a test subject as part of their course grade. They use SONA to view available tests and sign up for a time slot, and the experiments that involve playing video games tend to fill up very quickly. Since Intro to Psychology is a general education course, many subjects are freshmen and the majority are not studying computer science or similar majors. This could provide an abundant set of candidates that match our target audience. The UCF

Psychology department may only allow these official tests to be set up and run by Psychology majors and/or Psychology graduate students doing research. If this is the case, we could look for a graduate student conducting educational research who would be interested in sponsoring these official tests.

5.2 Research and Investigations

In order to define a clear vision of what we wanted our project to be, we decided to consult research and conduct playtesting of similar games in order to inform the development of our prototypes. Once our prototypes were formed, we playtested them as well so that when we began coding the game, it would be with a clear and uniform vision in mind. The following section summarizes our findings throughout this process.

5.2.1 Research Papers

During the prototyping phase of our project, we took the time to study research centered around teaching computational thinking to beginners, with a special interest in articles focused on game-based delivery of concepts. We wanted to know how previous attempts at this approach succeeded and to ascertain which parts were not successful, what caused them to fail, and how we could avoid the same pitfalls in our own project. That way, we could make an informed decision on the progression and clarity of instruction within our own game.

One study was centered on a program designed to observe students' abilities in using Scratch to develop solutions for solving computationally based problems [1]. Scratch is a drag-and-drop block-style programming interface designed for children to help them learn how to code without requiring them to navigate the intricacies of typing and compiling code. The program consisted of a discussion of the topic coupled with a demonstration in Scratch and an emphasis on the applicable computational thinking skills required, followed by students creating their own Scratch based solutions [1]. The programs that the students developed were evaluated for their ability to decompose the problem at hand and the skill level displayed in efficient program development [1]. The study found that basic concepts like sequence were easy for students to pick up on, but that as program requirements became more sophisticated it was increasingly difficult for the participants to compose and debug their programs [1].

Applying these findings to our own sequence of instruction has informed us on the importance of properly conveying the progression of topics. It is essential that we not only deliver clear information about the mechanics of each element of our game, but also that we properly articulate and demonstrate the more complex topics, such as nested if statements or nested loops. The sequence of instructions and data structures being introduced needs to support a logical progression from simple to more complex topics, and it is necessary to ensure that players are familiar with concepts before a new one is introduced. For example, we wouldn't want to introduce new instructions in succession, but rather require the player to solve several puzzles with each element to build familiarity and comfort before a new element is introduced. Additionally, we recognize that proper conveyance of debugging practices and insightful feedback on this practice is an integral aspect of user success.

A second Scratch based research that was conducted involved freshmen level college students, which is much more in line with our project's target audience. All of the students in the study were either computer engineering, software engineering, or informational systems engineering majors, and the college where the study took place requires all engineering students to take their Computer Science I course in their second semester. The model for this study was a modification of the Computer Science I course at the university the papers' authors both taught at that added a Scratch project as part of the grading criteria for the course [2]. Aside from this addition, the course was identically structured to all other semesters the course was offered. When the performance of students in the experimental course was compared against the performance of students in the traditional course, it was found that students in the experimental version – the one supplemented with the Scratch project – performed significantly better than the others [2]. Additionally, the progress of these students was tracked over their four years of college curriculum to determine whether or not this augmented form of instruction in the core class might have lasting effects. Interestingly, it was found that while these students performed unquestionably better in a Senior Design project than their peers, the deviation of CGPA between the two groups was negligent [2]. The paper also notes that while the students enrolled in the experimental program consistently outperformed their peers, the failure rates remained the same as the traditional version of the course at this institution, over 50% [2]. Still, the results of the study indicate that the engaging platform that was provided to these students greatly enhanced their success within related courses.

The paper makes note of many key pieces of information in regards to the difficulty of the subject, but most importantly among them that "the programming concepts and language syntax may become barriers for learning programming and an inhibiting factor for motivation among students" [2]. This reinforces one of the main points our project aims to address – that students first entering programming classes are overwhelmed by needing to learn syntax as well as computational thinking, while also learning how to use them in conjunction with the concepts of the course. By acquainting players with computational thinking and algorithmic processes before they enter these introductory level courses, we hope to ease this perceived inhibition and allow students to approach their new topics and learning syntax with confidence. Additional hurdles that students encounter in learning programming include the large number of concepts that need to be mastered, the absence of instant feedback via interactive media, and a lack of strong problem-solving skills [2]. Our project aims to address all three of these issues by design. The concepts students encounter in these courses will already have been introduced to them through the course of completing our game, and problem-solving skills – especially those related to solving problems using the concepts learned in these courses – will be honed and tested as they progress through the levels. Through the animations of our character carrying out their coded solutions, our game also provides instant feedback on how well their solution actually works on the problem in question.

Another research team utilized a system they developed to create tiling patterns to teach basic programming concepts to students, such as sequence of execution, loops, and recognizing logic errors [3]. Tilings were selected because they allow creative exploration for the students while their construction is still rooted in a process that can be done computationally and followed with linear progression [3]. Similar to the simple pseudo-language that our game features, the researchers made a simple text-based language for students to use to produce tiling sequences and used it in an introductory programming course at a college, noting that using high level languages like C are often overwhelming for new programming students [3]. Indeed, using a language like C or Java to produce the same pattern that students made with their simple language in their first lesson would take a student hours of studying syntax, learning programming concepts, and then applying problem solving to produce the correct pattern. Just as in the previous study, this research was conducted on college freshmen with little to no prior programming experience majoring in fields that require computer science, which is our key demographic. Students who participated in the testing of this method of instruction were invited to fill out a voluntary survery about their perception of its efficacy. About 53% of students said that the use of the tiling program increased their knowledge in programming, as well as showing favorable agreement with statements indicating the system was engaging, fun, and increased their understanding of the programming concepts presented [3].

The method of delivery of concept for this study was of particular interest to us. By abstracting away from complex syntax and utilizing a simple textual language, the researchers were able to very effectively convey concepts, and students were comfortable using the commands provided to form their solutions to the problems they were presented with. The paper notes that these simple languages are so effective because they have a small command set and simple syntax, and are based on familiar verbage that clearly express their function [3]. This use of familiar language to teach new concepts helps users by concentrating on the function and concept of the commands instead of getting bogged down in technical terminology. Another interesting point that was made is that introductory courses are often focused on processing information that is not interesting to novice programmers, and therefore they are less motivated to complete the work or pursue fields centered around programming [3]. By abstracting the instruction of these concepts into another format that users find captivating, such as a game, we can increase the chances that students will be motivated to apply themselves to the offerings of computer science. Further, using a system that delivers immediate visual feedback is a much more rewarding form of response than waiting for a program to execute only to find the expected output is wrong, and trying to trace back through the length of the code to discover the errors. Indeed, giving immediate and meaningful feedback to a beginner in any subject is vital to their continued progress and success [3]. Our game will achieve this via the visual execution of the players solution being carried out by our character.

These first three research projects are all characterized by using some sort of visual coding format either as a supplement to, or as the tool for, delivering programming based instruction to beginners. Other research we found concentrated more specifically on the effectiveness of using games as a medium for teaching computational thinking or programming concepts to users. Documentation concentrating on this topic was of special interest to us, as it applies so directly to our goals for this project. There are already many studies based around the idea of using game-based instruction for teaching, and it has been firmly established that it is a successful medium for delivering instructional content. The real issue we are facing is whether or not it can be used effectively in regards to teaching computational thinking.

One such study we found aimed to teach computational thinking concepts using "unplugged" games to teach the elements, and then following the unplugged activities, the students would apply what they had learned to correlated "plugged-in" programming exercises [4]. The unplugged version included tangible real world objects (e.g., a deck of cards) that students would physically interact with in order to demonstrate understanding of a particular concept, and the plugged-in version would present the participants with the same object computationally and ask them to trace through a solution based on the concept and rules they were already familiar with for that particular object. The reasoning for this abstract approach is that the underlying computational concepts can be learned in any medium, and that by removing the technological applications the emphasis is specifically on the processing of information using computational thinking instead of having the participants focus too much on the technology they are using [4]. It was found that for most of the activities, a majority of the students were able to accurately solve the tasks on the plugged-in assessments with approximately 90 percent correctness [4]. However, the activity that aimed to teach students about conditionals and nested conditionals had very disappointing results, with students only solving 6 percent of the final assessment for that lesson correctly [4]. There was also a questionnaire element to this study designed to help the researchers measure the interest the students had in pursuing an IT-related job, which the students filled out both before the course and after it ended. Interestingly, the results actually indicated a slight decline in students interested in choosing a related job after completing the course, but contrastingly showed an increase in the desire to learn more about computer science, and also indicated that students enjoyed the course and felt they learned something valuable [4]. The shortcomings of this study with respect to teaching conditionals can likely be attributed to not spending enough time on the topic and the ending assessment being too complex [4].

This study in particular has greatly informed us on how we want to introduce this topic within our own project. First, we would like to introduce single conditionals in a tutorial level and then have the user solve several non-trivial puzzles using the concept, followed with another tutorial level showing them how to operate with nested conditionals and subsequent puzzles implementing that technique. Layering this instruction instead of releasing it to them all at once should help users understand the basics of conditionals before allowing them to get bogged down in more complex applications. It should also be noted that the instructors for this study were trained in teaching for only two hours a week over the course of three months [4], whereas, cumulatively, the members of our team have years of experience in teaching these concepts to novices. Still, the results of the overall effectiveness of gamification of computational thinking concepts as a solid method for instruction is very promising, and with the right approach we know that our project can be successful.

5.2.2 Playtest Results

Playtesting was vitally important to our success early on. Knowing that the main goal of our project was to fill the gap where other similar games fell short, we conducted playtesting sessions where we observed volunteers interacting with these games and took notes. By focusing on participants that had little to no coding experience, we were able to examine how the target demographic for our project interacted with material that was geared towards programmatic thinking. Because we were drawing inspiration from these games for our own project, we wanted to know precisely what about them was effective. Similarly, since it is our goal to improve upon the current offerings, we were particularly interested in which aspects of these games were confusing, cumbersome, or overwhelming to inexperienced players.

Similar Games
Human Resource Machine was the original inspiration for our project, and is well liked by programmers for its ability to have users explore algorithmic design and computational logic. It presents puzzles to the user and gives them a set of simple instructions to build a solution out of and an on screen character acts out each instruction as the sequence of the solution is run through. However, the game does not provide much in the way of informing users about the concepts presented. Rather, it simply gives the users new commands as they progress through the levels and expects them to figure out the mechanics associated with each new command through trial

and error. There is also a notable lack of useful information in regards to how they should use each instruction and what its intended purpose is. There is no tutorial element of any sort, and many of the early levels' "hint" options are completely useless.

In preliminary testing, users struggled to successfully navigate the beginner level puzzles, and none of the participants were able to complete level four. The lack of information when new instructions were presented made it very difficult for the users to understand precisely what they needed to achieve in order to successfully complete each puzzle. The users that did seek out hints from the game were met with useless text that did nothing to explain the puzzle, the commands available, or a tactic they might want to explore to nudge them towards a solution. As new instructions were made available to the users on each subsequent level, their frustration with not understanding how the commands worked together compounded and detracted from their desire to continue progressing through the game. The jump command and the registers proved to be very difficult for the users to understand without any effective messaging, and because they are such an integral part of the game, this lack of understanding early on caused a serious roadblock.

Several mechanics within the game also contributed to the difficulty that users were experiencing with understanding what they were supposed to be doing, or how they could improve when their solutions didn't work. A slider bar that increases the speed of the character acting out the instructions allows users to quickly run through their solution. One user turned the slider to max speed while they were still trying to form a solution and this took away their ability to understand where their program was going wrong. In a similar vein, all of the users were focused on the actor and what it was doing to try to figure out the issues with their solution instead of paying attention to the sequence of the solution itself and how to trace through and debug their problems. The debugging tools within the game are never explained, and most users never even attempted to utilize them. The game also allowed users to try the same incorrect solution many times without providing any kind of feedback, and even allowed one player to get stuck in an infinite loop from a poorly placed jump command.

In subsequent testing sessions, we provided missing information for users where the game does an insufficient job of providing details. We explained each command, what it does when used, how to manipulate the commands effectively, and how to work with the registers in the game. We also showed the debugging tools to the users and explained how it could be useful to trace through their solution when they encounter problems. Additionally, when users wanted a hint on the level, we provided them with clarifying information about the purpose of the puzzle and what their goal was, as well as small solution based hints, such as “you need multiple jumps in this one” and similar suggestions that nudged them toward a solution without giving away the puzzle. During these sessions, there was a noticeable increase in the comfort level the users had with manipulating the game, especially concerning the debugging tools and manipulating the commands within the solution box. There was also a definitive improvement in users’ ability to successfully solve the puzzles they were presented with.

The main take-away from *Human Resource Machine* is that messaging is incredibly important in making a game like this effective. It is very clear that new programmers need more guided instruction to be able to engage in the puzzles without being directed by someone who more clearly understands what to do. By improving the level of information provided, we were able to give users greater confidence in the skills they were learning, and in turn they were both more willing and more capable at solving various puzzles.

7 Billion Humans is the sequel to *Human Resource Machine.* It features a much more advanced instruction set and more varied puzzles, but it is plagued by the same communication issues present in its predecessor. The puzzles in this installment were less abstract than before, but the added concreteness came at a severe cost. While the game is highly engaging for those with programming experience, new players were even more lost than they had been previously. Aside from the instructions being much more complex, the puzzles now required the consideration of spatial manipulation of the on screen characters in order to be solved correctly. So, in addition to all of the same difficulties that we saw users struggle with during *Human Resource Machine*, we also observed that users were not retaining information or strategies from puzzle to puzzle.

A key observation elucidated from *7 Billion Humans* is that programming puzzle games that rely on spatial manipulation of the actor executing the instructions distract the player from the essence of the puzzle. This is incredibly important because many games in the genre rely on this mechanic. Players frequently became more concerned with counting spaces than understanding the logic needed to solve the puzzle at hand. This space counting led to an interesting approach to the puzzles we did not observe during playtests of *Human Resource Machine*. The spatial puzzle format saw an alarming rise in brute force solutions. By counting squares and adding enough if statements, players could eventually arrive at a solution that was complicated enough to pass – sometimes on accident. The idea that this tactic was successful was very shocking, as it effectively side-stepped the entire point of the game.

Our Prototypes

After we formed the initial versions of our individual prototypes, we had participants play through simple puzzles on them to test out their efficacy. Our prototypes were formed after the playtesting mentioned above, so we were aware of certain shortcomings in these types of games and made efforts to avoid similar pitfalls in our own designs.

A problem that we initially ran into was that the paper medium of our prototypes were not as immediately accessible to users. Some individuals struggled with the concepts and didn't seem as confident to begin the playtesting without receiving ample instructions first. We felt that this coincided with our previous findings in the playtesting of other games, but were wary about making a game that was heavily text-based. Even this very first step in the process informed a lot about our game design. We must be able to effectively communicate the gist of our game without losing our players interest in a wall of text.

Another interesting result of the playtesting of our early prototypes was that, once the game had been set up for them, participants were more engaged in the prototypes than they had been in the other games. During game playtesting, most users continually sought help or approval from us when we were merely operating in an observational capacity. During the prototype playtesting, though, users seemed much more confident to adjust their solutions and explore the puzzle on their own. It is unclear if the mode of delivery swayed these interactions. For one, the prototypes provided a much more hands-on experience, and therefore were inherently more engaging. Additionally, for the prototype playtests each of us filled the role of the Actor character and were responsible for completing the puzzle manipulations as we processed the participants' solutions. Because we were now essentially part of the game instead of a liason for the game, users may have subconciously avoided trying to ask for help from us as frequently as before.

After this first phase of prototype testing, we accumulated all of our individual results and designed a unified prototype. This prototype was a whiteboard-based version of the game that could be easily used and reused for playtesting. It came with a set of sample puzzles that were designed to slowly increase in complexity and make users comfortable with the progression of the game, and also featured the instantiation of our "data structures represented with cards" mechanic. For reference, we each had a list of the instruction set with short descriptions of what each command was used for.

This round of playtesting was extremely informative for us as it was a resounding success. First, the idea of using cards for the data structures worked very well with our users. By using a medium they were familiar with, it was instinctual for them to understand that they had to "play" a data structure in the puzzle to be able to use it in their solutions. Secondly, the progression of our puzzles provided a concrete base for users to understand the concepts of each puzzle and apply their solutions to similar but more complex puzzles. Also, the progression of the puzzles was found to be steady enough that users found the problems challenging, but not so disjointedly complex that they felt overwhelmed or lost when seeking out a solution. Most users were able to complete the sequence admirably. Finally, we found that our limited instruction set was extremely effective. Users were able to understand the instructions clearly and utilize them effectively for various concepts during their testing.

Early Iterations

Implementing our vision in Unity was a very involved and lengthy process. Many systems had to be designed to support the construction of our game, as well as solid foundations for the individual systems. We committed ourselves to realizing the design we had in mind and doing what it took to bring it to life instead of compromising on design ideas when they were found to be much more involved to implement than we had originally planned. Still, we wanted to be sure that our efforts were moving in the right direction, and that we weren't just adhering to what we thought the game should be if the methods were not effective in the actual execution. We continued to playtest our game at different intervals to get continuous feedback to update our goals for change and polish needed in our execution.

Many early findings reported things we expected, like that a particular game mechanic was not explained well enough or that the appearance of certain elements in the scene did not stand out enough. This kind of feedback helped reassure us that we were headed in the right direction with our end goal designs. The mechanic that was not explained well enough still had a tutorial that needed to be created for it. The elements in the scene that did not stand out well enough were awaiting replacement art from our artist team. A lot of other feedback we received, though, was not exactly what we were expecting – people thought our game was interesting, and that the concept was engaging. This, more than anything else, gave us the motivation to aggressively pursue our final design goals and make them a reality.

5.3 Prototyping Efforts

5.3.1 John Billingham

Level Structure Overview
Each level is based on an input to output format. Given an initial input, the player will be expected to use a set of given instructions to manipulate the data to match an expected output for the puzzle level they are currently on. If their created output matches the expected output for that level, they have passed the level. Otherwise, they will need to come up with a new set of instructions, maybe just slightly altered from the previous solution, to create the exact expected output. Example levels can be found in Figures 16 and 17.

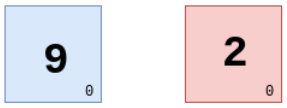
Import components of this prototype include:

- Registers
- Input
- Output

- Instructions

Registers
The puzzle game will rely on a color-based memory indexing system to access and manipulate game data. Colors will be used when accessing data structure indices, although this only applies to registers for now. Two basic blue and red registers can be seen in Figure 1.

Figure 1: Register Examples



Consider being shown these two colored boxes. The contents of the box are represented as the large, centered numbers in bold. The indices of the box are represented as the small number at the bottom right-hand corner. Indices tell you where to look in a data structure, which is redundant for now since registers only get one index.

- Figure 2 shows how to access the 9 in the blue register.
  - (called: blue sub 0)
- Figure 2 also shows how to access the 2 in the red register.
  - (called: red sub 0)

Figure 2: Register Indexing

From now on, these data structures of size 1, that only contain the index 0, will be referred to as registers. A specific register and its index is said to be a memory location.

Input
Input is data that is given to you initially. This may include one register or multiple

registers, as shown in Figure 3.

Input can also arrive in the form of an input stream. A stream of input is connected to one register and continually pushes data from the stream into the register as data is removed from it (see Figure 4). If there is no more data in the stream, nothing will be pushed to the register.

Figure 3: Input Examples

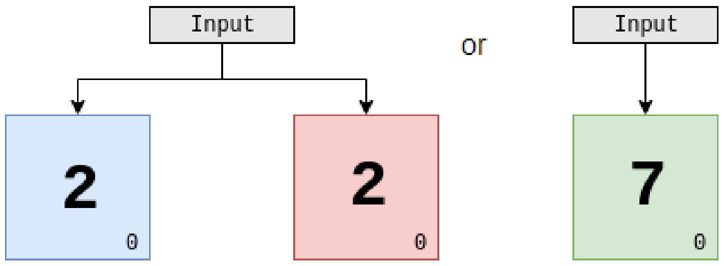
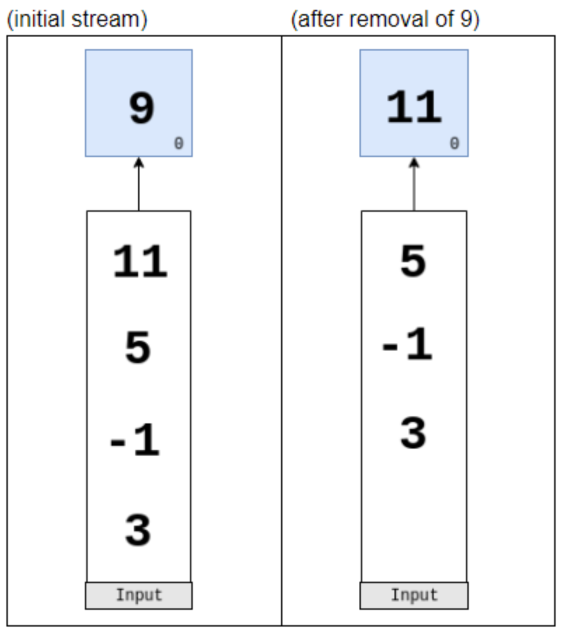


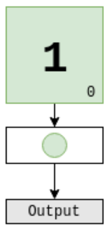
Figure 4: Input Stream



Output

The output is checked at the end of the program. It is up to the player to link the output to the correct data structure (again, only registers for now), as well as to make sure the contents of the data structure match the expected output for the level. A basic register to output linking can be seen in Figure 5.

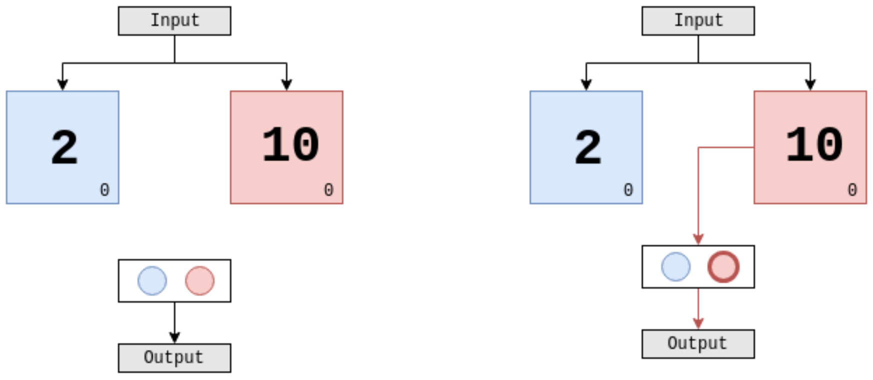
Figure 5: Register to Output



The green circle specifies that the contents of the green data structure will be returned as the output for the player's solution to the puzzle level.

Consider a scenario in which multiple data structures may be returned as output. In the following example (Figure 6), the player needs to return the register with the greater value by choosing which colored data structure to link to the output.

Figure 6: Output Choice



Players can link data structures to output by simply choosing the corresponding colored circle of the data structure they would like to return. Clearly, 10 is greater than 2 here and the red register should returned.

Instructions
The notable instructions are as follows:

- Move
- Return
- Add
- Subtract
- Jump

Instructions are clicked or dragged from a limited set of given instructions, specific to each level. When the player is ready to test their solution, each instruction will execute procedurally from top to bottom.

It is up to us to structure the levels and instructions given in each level in such a way that minimizes the amount of errors the player can create. Runtime errors are okay, but we should strive to avoid having the player feel as if they are learning syntax. We would also like to avoid confusing the player by giving a limited instruction set for each level with thorough explanations for each instruction.

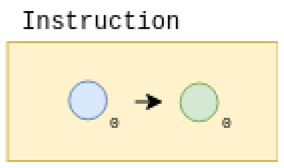
Parameters used in certain instructions (Move, Return, Add, Subtract) are chosen from a drop down menu, again minimizing the amount of errors the player can create on their own. When choosing a colored data structure and an index, only the colors of the data structures and their valid indices that are already on the puzzle board are given as options in the drop down menu.

Instruction: Move
The move instruction takes two arguments, a source register and a destination register. It's main functionality consists of three different parts:

- Move the contents of the source to the destination
- Leave the contents of the source unchanged
- Override the previous contents of the destination

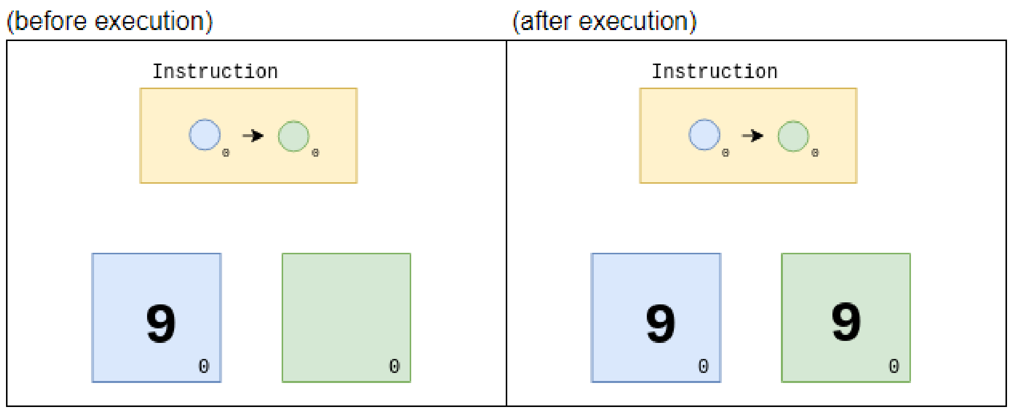
How the instruction would look in the instruction editor is shown in Figure 7.

Figure 7: Move Instruction



The contents of the two registers before and after the move instruction is received is shown in Figure 8.

Figure 8: Move Instruction In Use



Each argument (left and right side of the arrow) will have two drop down menus:

- Color of register
- Index of register

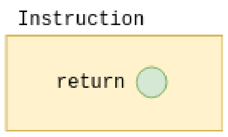
The move instruction is atomic and meant to abstract the process of manually picking up the data and placing it down.

Instruction: Return
The return instruction will specify only one argument – which colored data structure to return as output.

- A return instruction is required for every level
- May be given, or may require the player to write it
- The data structure that is returned is compared against the expected output for the level

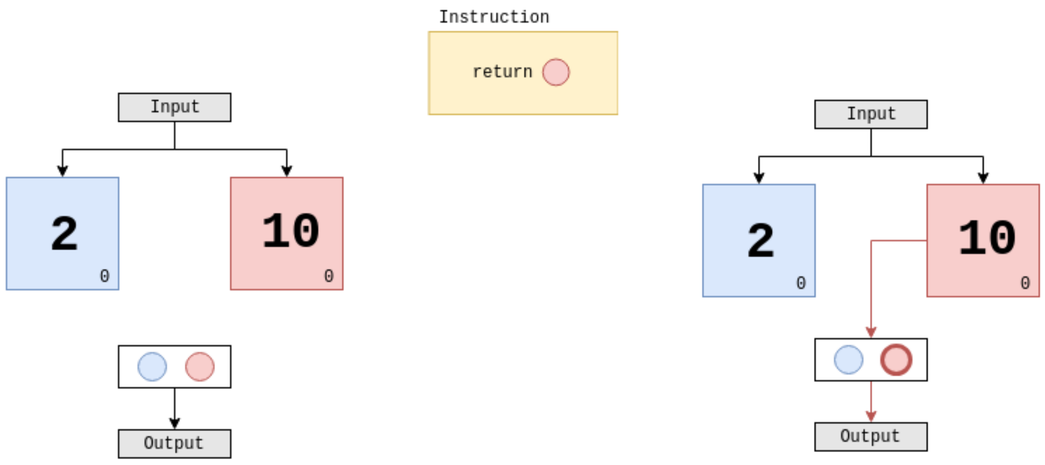
Figure 9 shows how the return instruction looks in the instruction editor. The argument is chosen from a drop down menu that allows the player to select the color of the data structure they want to return.

Figure 9: Return Instruction



Using the example from the output section (Figure 6), we can show the correct usage of the return instruction in Figure 10.

Figure 10: Return Instruction In Use



Instruction: Add
The add instruction takes three arguments:

- Two memory locations to be added
- The destination of their sum, another memory location

Figure 11 shows how the add instruction looks in the instruction editor.

When referencing the memory locations, an index must also be specified, as we've seen before. Figure 12 shows the add instruction in use by displaying the contents of the registers included in the instruction before and after it is run.

Note: The destination can be one of the source registers used, in which case it would just overwrite the data, similarly to how the move instruction functions.

Figure 11: Add Instruction

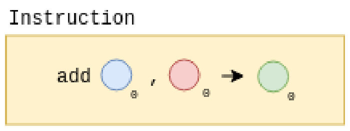
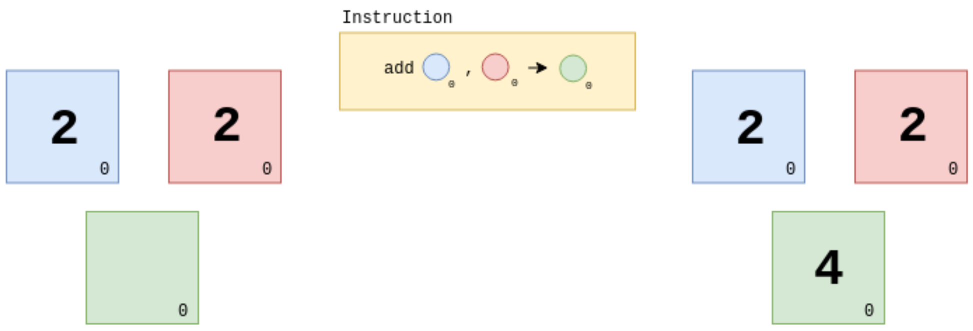


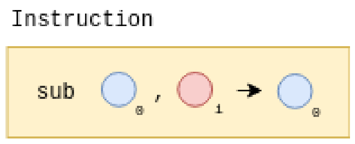
Figure 12: Add Instruction In Use



Instruction: Subtract
The subtract instruction works exactly the same as the add instruction above, although the difference of the two registers is placed into the destination, rather than the sum.

Figure 13 shows how the subtract instruction looks in the instruction editor.

Figure 13: Subtract Instruction



Instruction: Jump
The jump instruction is paired with an anchor. When a jump instruction is executed, the flow of control immediately jumps to the instruction that lays at the anchor. It is up to the user to choose where the anchor will lie after placing a Jump instruction in the instruction editor.

Figure 14 shows how the jump instruction looks in the instruction editor, while Figure 15 shows how the control flow is handled when a jump instruction is executed.

Figure 14: Jump Instruction

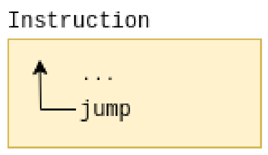


Figure 15: Jump Instruction In Use

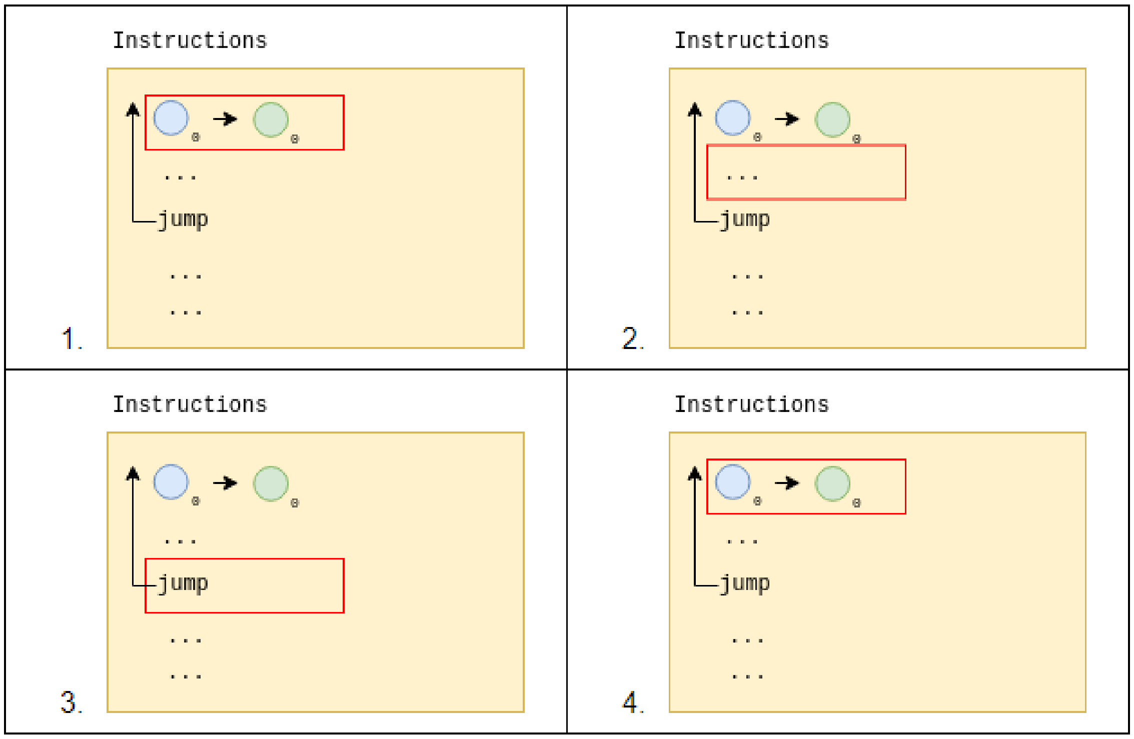


Figure 16: Example Level: Find Sum

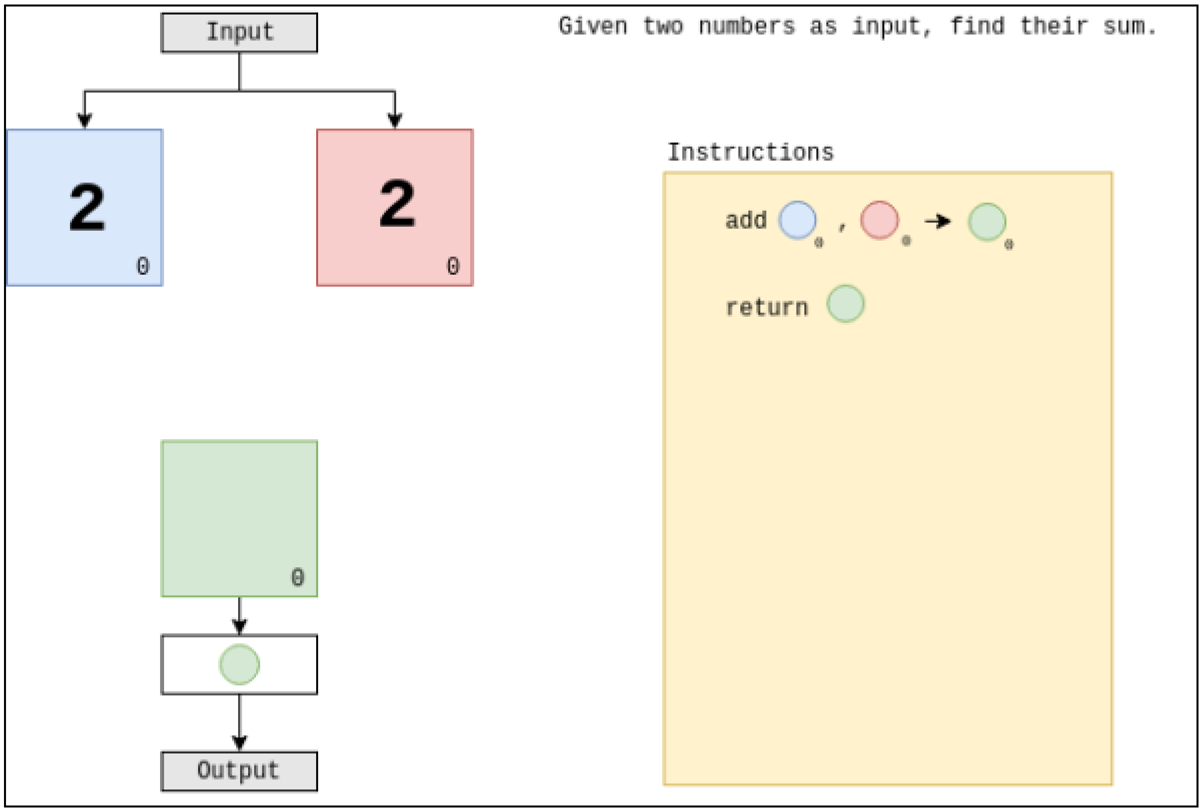
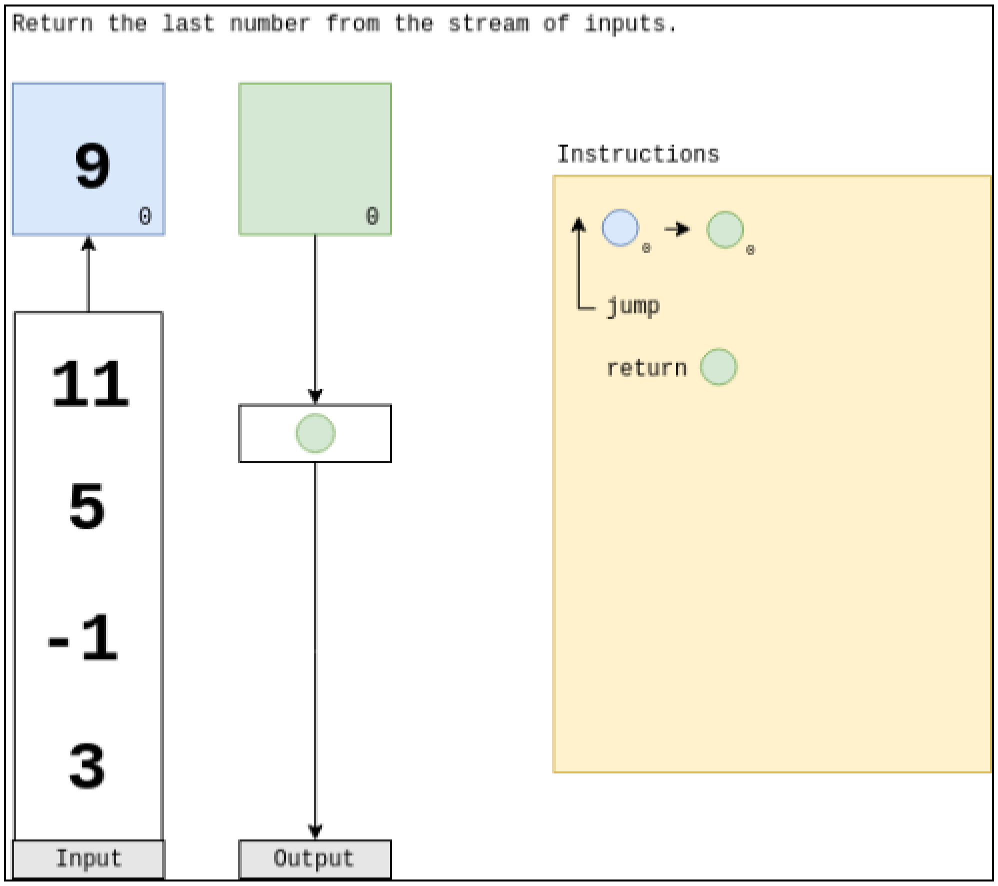


Figure 17: Example Level: Return Last


Return the last number from the stream of inputs.

5.3.2 Brandy King

My prototype is called *Desktop* (see Figure 18). Expanding on the initial game structure we formed our ideas around, my prototype adds the ability for users to declare variables and data structures. The idea is that by making the user responsible for managing these resources directly it will help reinforce the concepts associated with each element. An Actor will move around the field and carry out the commands selected by the user in the Solution area.

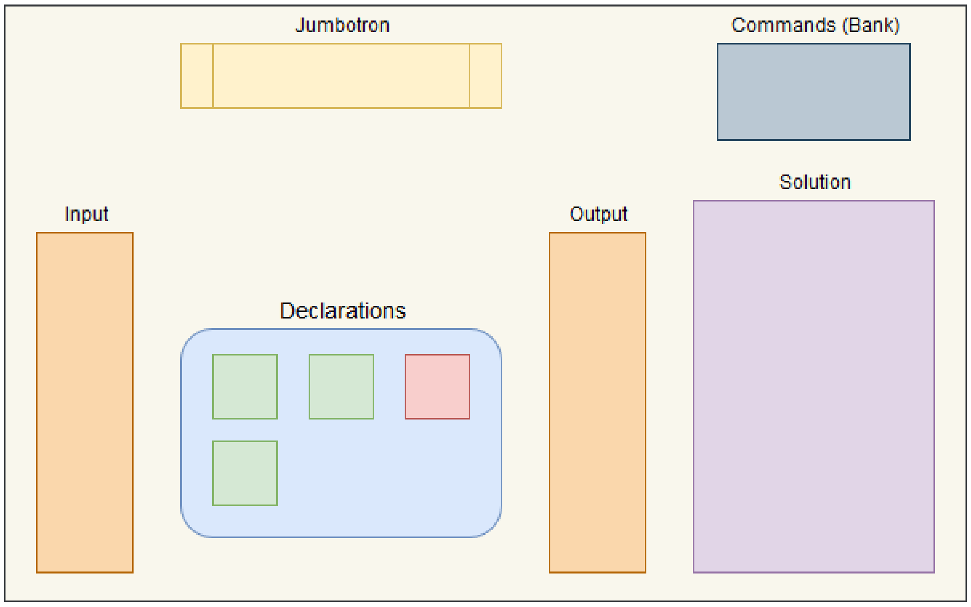


Figure 18: Sample layout of the basic *Desktop* level design

Setting

The playing field of *Desktop* takes place on top of a desk. A small figurine on the desk comes to life at the start of the game and moves around the desktop during gameplay. Each level of the game is a puzzle designed to test the user on their skills with computational thinking. Tutorial style levels, which occur whenever a new command or data structure is introduced, will feature a short to-do list for the player to complete to ensure they understand the concept before they are required to use it to solve a puzzle.

Elements

Input: The input trough will feature a queue of pieces of data that the user will need to manipulate in order to solve the puzzle. Data can only be picked up one at a time, and only the topmost piece of data can be selected - the player cannot pick up an arbitrary piece from the queue. The values of the data will be randomly generated when each level is loaded, and puzzles must be dynamic to handle any number of data elements with any value. Data cannot be placed back into the queue once it has been removed. The style of the input trough is to be determined.

Output: The output trough is the area where the manipulated pieces of data should be placed in the expected order for a correct solution. As an element is added to this queue, it will push the rest of the elements down so that the first element is always on the bottom and the most recently added element is always on top. Data cannot be retrieved from the queue once it has been placed. The style of the output trough is to be determined.

Declarations: This is the designated area for players to place their declared variables and data structures. These items cannot be used in the commands within the solution unless they have been placed here first. The variables and data structures will be stored in the commands bank, and players will drag them to the declarations area in order to declare them. When the items are dropped here they will snap into the grid. There is a limited amount of space available for the player to utilize, so more intricate puzzles will require the player to manage their declarations more closely. The declarations area will be styled after a notebook.

Commands: This is a storage area. It will consist of a listed bank of commands, as well as the variable and data structure elements, currently available to the user. The commands will appear as small rectangular cards and will be manipulated with drag-and-drop controls. The variable and data structure elements will be square shaped to make them easily differentiable from the commands. For more information on each of the structures and commands, see their respective sections below. The commands area will be styled after a stacked document sorter, with loose plans for the player to select which “level” of the sorter they are pulling from - commands or data structures. Further separating these two components will help keep the concepts discreet from one another.

Solution: This is the area where the player will drag-and-drop their commands in order to build a solution to the puzzle. If the solution length exceeds the size of the box, a scroll bar will pop up so that users can navigate through their solution. The bottom of this box will have a fixed panel with debugging buttons and a button to run the current solution. While a solution is running, an indicator will pop up in the solution box and point at which command is currently being carried out by the figurine. The debugging buttons will feature a step button that executes only the next line of code, a back button that undoes the last line of code executed, and a stop button that will halt the current running solution. Players will be able to rearrange commands within the solution box by dragging them to different places within the solution. Similar to how structures will snap to a grid within the declarations box, commands will align within the solution box in sequence. The alignment will be dynamic whenever a command is being drug through the current sequence of commands so that users can see precisely where any inserted commands will be placed. Commands that are inside of an if block will be indented to more clearly show the user what is part of the conditional logic. When if blocks are open, the subsequent commands will automatically be aligned with the indentation until the user places a command flushed left. This alignment will also be dynamic, where users can drag commands right or left to either include them in the block or mark the end of the block. The command area will be styled after a yellow legal pad.

Figurine: The figurine will be a small animated character that moves around the desktop, carrying out the commands that the user has selected in the solution box. The user will only be able to manipulate the data in the puzzle through the actions of the figurine, which are subsequently controlled by the sequence of commands in the solution box. The animations of the figurine will assist in conveying how the data is being manipulated by the commands, as well as clearly displaying how the sequence of commands in the solution is working to solve the problem. The figurine will become visibly upset when a solution fails to find the correct solution, and will cease their actions whenever a program halting error is encountered. They will also be the source of providing information to the player. If the player clicks on the figurine while they are in a halted state, the figurine can provide feedback to the player. When errors halt the figurine, it can inform players on the type of error. If the figurine is at a natural halt (ie, the solution finished running but didn't find the correct solution), it can provide tips and hints to the player that can help clarify the goals of the current puzzle. Additionally, this character will be the method of delivering newly acquired commands and structures to the player. The exact design of the figurine is to be determined.

Jumbotron: The purpose of the Jumbotron is to provide an additional level of clarity to the user. Its sole purpose is to essentially broadcast which command the figurine is currently executing. This helps address the issue with players only paying attention to the character when running their solutions and not watching the sequence being followed within the solution box. The Jumbotron will be styled after a desktop digital clock displaying the current command instead of the time, and will flash to indicate a change of command.

To-Do List: The to-do list is located in the upper left hand corner of the desk, and will only become active when the user is in a tutorial level. Its purpose is to list the specific tasks that the player needs to accomplish with their new command or data structure to demonstrate their understanding of it. Check boxes on the to-do list will automatically get ticked off once the task is completed, and players will not be allowed to advance to the next level until all tasks on the list are satisfied. During this time, the play area becomes an open play arena, with endless input for the player to manipulate. Solutions can be run as normal, but there is no expected correct solution for these levels. The player is allowed to freely manipulate data however they want in order to test out their new tools, as well as play around with all of the other commands and structures currently available to them. Once all of the items on the to-do list are achieved, a button next to the to-do list illuminates and they can choose to move forward to the next puzzle whenever they are ready to do so. The to-do list will be styled after a post-it note.

Structures

For all of the structures available to the user, they are not required to maintain the storage of data within the structure. Everything that is related to structure and storage will be handled on the back end. This allows the user to concentrate solely on the concept of the structure.

Variable: The variable structure can hold a single piece of data. If new data is stored in the variable, the previous value being held is lost. Data can be copied from or removed from the variable. The variable will be visualized as a single square.

Heap: The player selects if they want a MinHeap or MaxHeap when declaring the heap structure. Multiple pieces of data can be stored in a heap. The player is only able to copy or remove the top element from the heap, either the min or max depending on which type of heap they are using. When new elements are added to the heap, it shakes on screen as it sorts the data. The heap will be visualized as a stack of squares, but only the top element will be visible.

Stack: Multiple pieces of data can be stored in a stack. The player can only remove the topmost element from the stack. The stack will be visualized as a stack of boxes, but only the top element will be visible. When new elements are added to the stack, a new box appears on the top with the most recently added element. Once the number of elements in the stack is five additional boxes that are added don't cause the stack to visually expand, but the data will be maintained for any number of elements in the stack.

Queue: Multiple pieces of data can be stored in a queue. The player can only remove the first element from the queue. The queue will be visualized as a stack of boxes, but only the first element will be visible. When new elements are added to the stack, a new box is slid underneath the current stack, and only the first element of the queue is visible. Once the number of elements in the queue is five, additional boxes that are added don't cause the queue to visually expand, but the data will be maintained for any number of elements in the queue.

Commands

Many commands within Desktop have blank fields that the user needs to fill in. Whenever one of these commands is first played in the solution box, the available items on the board that can be used to fill that blank are outlined with a bright glow, and the player must click on the item they want to use in that blank. The user can also change the designated item by clicking on the command itself within the solution box, which will cause the items to illuminate again so the user can change the selected item. The applicable items are input and output by default but can also include variables and data structures that have been declared. The piece of data that will be manipulated is described within the relevant section for that item. If a command is not able to be used on one of these items, it will be explicitly stated for that command.

Pick Up (blank): This command will cause the figurine to pick up the applicable data element from the specified item. Any data currently being held by the figurine is lost. This command cannot be used on output.

Put Down (blank): This command will cause the figurine to put down the data they are currently holding in the specified item. The figurines hands will be empty after this executes. This command cannot be used on input.

Copy To (blank): This command will cause the figurine to make a copy of the data they are currently holding and place the copy in the specified item. The figurine will still be holding the original data after this executes. This command cannot be used on input or output.

Copy From (blank): This command will cause the figurine to make a copy of the applicable data element from the specified item and hold the copy in their hands. Any data held by the figurine before this executes is lost. There are no errors associated with this command. This command cannot be used on input or output.

Jump: This command causes the sequence of execution of commands to jump to the specified command. When it is played, the user has to drag an arrow attached to the jump command card and hook it into another instruction in the command box. There are no errors associated with this command.

Add: This command will have the figurine pick up the applicable data element from the selected item and add it to the data they are currently holding. If this command executes when the figurines' hands are empty, it causes an error. This command can only be used on variables.

Subtract: This command will have the figurine pick up the applicable data element from the selected item and subtract it from the data they are currently holding. If this command executes when the figurines' hands are empty, it causes an error. This command can only be used on variables.

Using if commands in Desktop causes subsequent commands in the solution box to be indented one level to show they are part of the conditional block. Users have to drag commands to the left to end the indentation and the conditional block.

If Less Than (blank): This command will have the figurine compare the data in their hands to the applicable data element in the specified item. If the value of the data they are holding is less than the targeted data, the commands within the if block will be executed. Otherwise, the conditional block will be skipped. If this command executes when the figurines' hands are empty, it causes an error. This command cannot be used on input or output.

If Greater Than (blank): This command will have the figurine compare the data in their hands to the applicable data element in the specified item. If the value of the data they are holding is greater than the targeted data, the commands within the if block will be executed. Otherwise, the conditional block will be skipped. If this command executes when the figurines' hands are empty, it causes an error. This command cannot be used on input or output.

If Equal To (blank): This command will have the figurine compare the data in their hands to the applicable data element in the specified item. If the value of the data they are holding is equal to the targeted data, the commands within the if block will be executed. Otherwise, the conditional block will be skipped. If this command executes when the figurines' hands are empty, it causes an error. This command cannot be used on input or output.

Needed Improvements

At the time of prototyping, Desktop does not have a way to check whether or not an empty condition might occur. This is problematic because we need some form of verification that a specified item is empty, like the input or a particular data structure. This missing piece will help bring together the type of algorithmic design and computational thinking that Desktop aims to convey to users.

5.3.3 Nicolas LaCognata

For my prototype, I spent a lot of time focusing on how we should integrate data structures into our assembly-based puzzle structure. The approach I took saw players attaching data structures to registers to "augment" them and change how the register stores and processes information. I also took a stab at defining a theme for our game.

Setting

In *Automata*, you play an Automaton stuck in a laboratory who is tasked with reprogramming itself to solve problems. When solving the puzzle, they player may be addressed by a narrator when they make a mistake. The game was to have a "Mad Science" feel to it, much like *Portal*.

Rules

For each puzzle the player is presented with a problem, and they must use the provided instructions to solve it. Every puzzle will contain the following elements:

- An Input track
- An Output track
- A set of static registers built into the test chamber
  - These registers allow the player to store and manage multiple data cubes
- A set of register augmenters
  - These augmenters turn the registers into special data structures that can make the puzzles easier/possible
- An instruction set
  - These are the instructions the Automaton can execute to solve the puzzle; when placed in the solution window, they will be executed in the order they appear
  - Some operations are invalid for certain states of the puzzle. For your safety, please avoid executing invalid operations.

Instruction Set

INPUT	**Summary:** Allows your automata to grab an item from the input track. It places it in its hands. **Notes:** Your automata will drop any item currently in their hands to accept the new input, losing the old data forever. If the input track is empty, the automata will do nothing **Faults:** None
OUTPUT	**Summary:** Allows your automata to place the item in its register into the output track. **Notes:** None **Faults:** None
SUBMIT	**Summary:** Allows your automata to submit its output for evaluation. **Notes:** This should always be the last instruction in your program. **Faults:** Output Empty Error Output Incorrect Error
JUMP	**Summary:** This type of jump is unconditional. When your automata reads this instruction, it will move its instruction counter to the corresponding anchor, effectively skipping forward or backward in your program. **Notes:** This instruction can be used to create loops. **Faults:** Output Empty Error Output Incorrect Error

JUMP IF NULL	**Summary:** This type of jump is conditional. When your automata reads this instruction, it will move its instruction counter to the corresponding anchor, if and only if its hands are empty. **Notes:** This instruction can be used to exit loops when there is no more input. **Faults:** None
JUMP IF LESS X	**Summary:** This type of jump is conditional. When your automata reads this instruction, it will move its instruction counter to the corresponding anchor, if and only if the data in its hand is less than the data in register X. **Notes:** This instruction can be used to exit loops or select branches of logic. **Faults:** Hand Empty Error Register Empty Error
JUMP IF GREATER X	**Summary:** This type of jump is conditional. When your automata reads this instruction, it will move its instruction counter to the corresponding anchor, if and only if the data in its hand is less than the data in register X. **Notes:** This instruction can be used to exit loops or select branches of logic. **Faults:** Hand Empty Error Register Empty Error

MOVETO X	**Summary:** This instruction allows your automata to place the item in its hand into a register on the board, specified by the argument X. **Notes:** This instruction overwrites any data in the target register, and empties your automata's hand. **Faults:** Hand Empty Error
COPYTO X	**Summary:** This instruction allows your automata to copy the item in its hand into a register on the board, specified by the argument X. **Notes:** This instruction overwrites any data in the target register, but your automata retains a copy of the data. **Faults:** Hand Empty Error
MOVEFROM X	**Summary:** This instruction allows your automata to take an item from a register and keep it in its hand. **Notes:** This instruction overwrites any data the automata is holding and removes the data from the target register. **Faults:** None
COPYFROM X	**Summary:** This instruction allows your automata to copy an item from a register and keep it in its hand. **Notes:** This instruction overwrites any data the automata is holding, but the data also remains in the target register. **Faults:** None

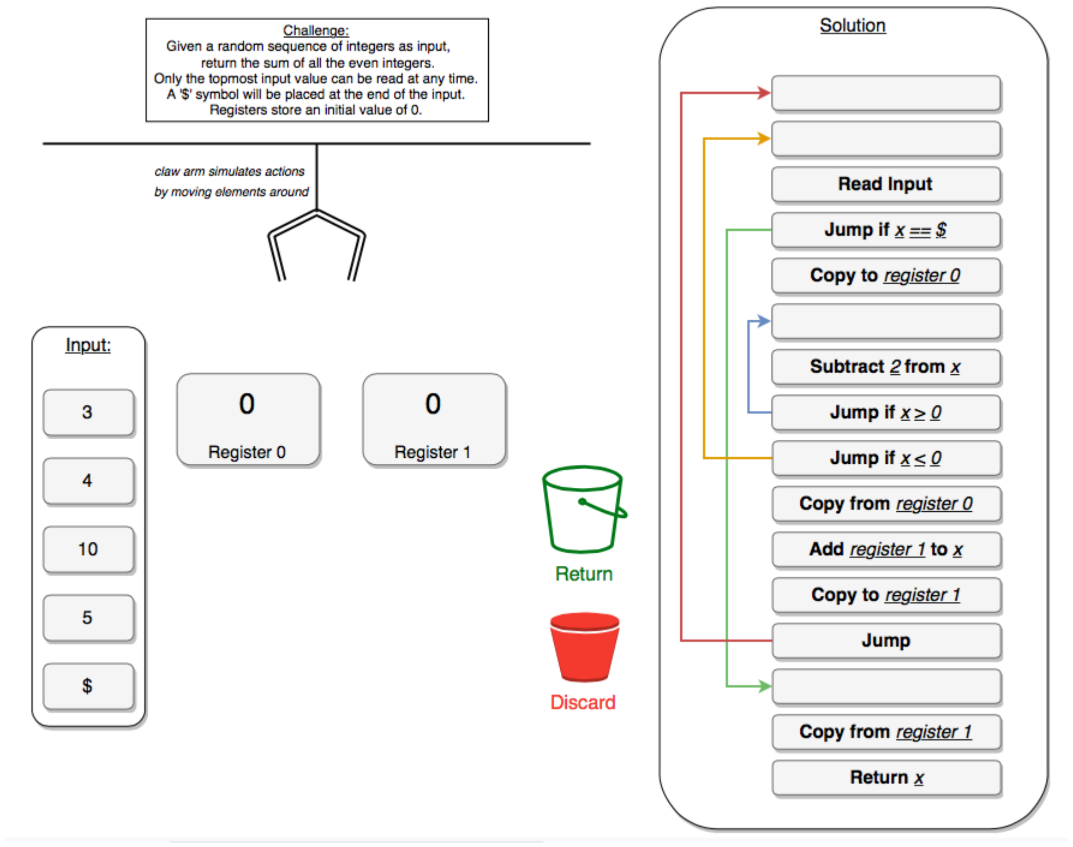
ADD X	**Summary:** This instruction allows your automata to add the value stored in a register to the value stored in its hand. **Notes:** The result of the operation is stored back in the automata's hand. **Faults:** Register Empty Error
SUBTRACT X	**Summary:** This instruction allows your automata to subtract the value stored in a register to the value stored in its hand. **Notes:** The result of the operation is stored back in the automata's hand. **Faults:** Register Empty Error

5.3.4 Sean Simonian

Paper Prototype 1.0

My initial paper prototype, shown in Figure 19, was built on the established game structure of manipulating input and output by executing instructions in a player's solution. The objective for each level is displayed at the top left of the screen, the solution space is on the right side, and the remaining area is utilized for visualizing the command execution. This prototype includes a robotic claw that can move horizontally across the top of the main game UI area, and can extend vertically to pick up or drop input values.

Figure 19: Paper Prototype 1.0


Challenge:
Given a random sequence of integers as input, return the sum of all the even integers.
Only the topmost input value can be read at any time.
A '$' symbol will be placed at the end of the input.
Registers store an initial value of 0.
claw arm simulates actions by moving elements around
Input:
3
4
10
5
$
0
Register 0
0
Register 1
Return
Discard
Solution
Read Input
Jump if x == $
Copy to register 0
Subtract 2 from x
Jump if x ≥ 0
Jump if x ≤ 0
Copy from register 0
Add register 1 to x
Copy to register 1
Jump
Copy from register 1
Return x

Solution Space:

The solution space on the right side of the screen is where the player constructs their code-like solution to satisfy the puzzle objectives. The player must use this space to place a series of instructions that will execute in order.

Instructions/commands:
The instructions that are available in this prototype are:

- Read Input
  - The claw picks up the top most item from the input.
- Jump [If ___]
  - This instruction will point to another line in the solution to jump to while executing.
  - Optionally, the player may include an If *(condition)* instruction so that the jump will only occur if a condition is met.
- Copy ___ ___
  - Copies the current value in the claw to a specified register, or copies the value from a register to the claw.
  - The first blank can be set to either "to" or "from"
  - The second value can be set to a specific register
  - Example: *Copy to register 0*
- Add ___
  - Add the value of a specified register to the current value in the claw.
- Subtract ___
  - Subtract the value of a specified register from the current value in the claw.
- Return
  - Place the current value held by the claw into the return bin, and stop executing the instructions.

The Claw:
As each step of the player's solution is executed the claw manipulates the data on the board, and the corresponding instructions are highlighted in the solution space. The claw does all of the work manipulating data as it simulates the player's solution to the puzzle, and it can only hold at most one value at any time. By having the claw

hold a value and manipulate is as specified in the solution, this renders a separate "current value" box unnecessary. The claw can pick up and put down data elements, or it can combine the element its currently holding with another element to add or subtract values.

Input:
Input arrives in a vertical stack that acts like a conveyor belt. The claw will navigate over the input stack, pick up the topmost item, and proceed to manipulate the data.

Registers:
Registers are located in the middle of the game board, and they can be used to store data that needs to be set aside while the claw manipulates other data elements.

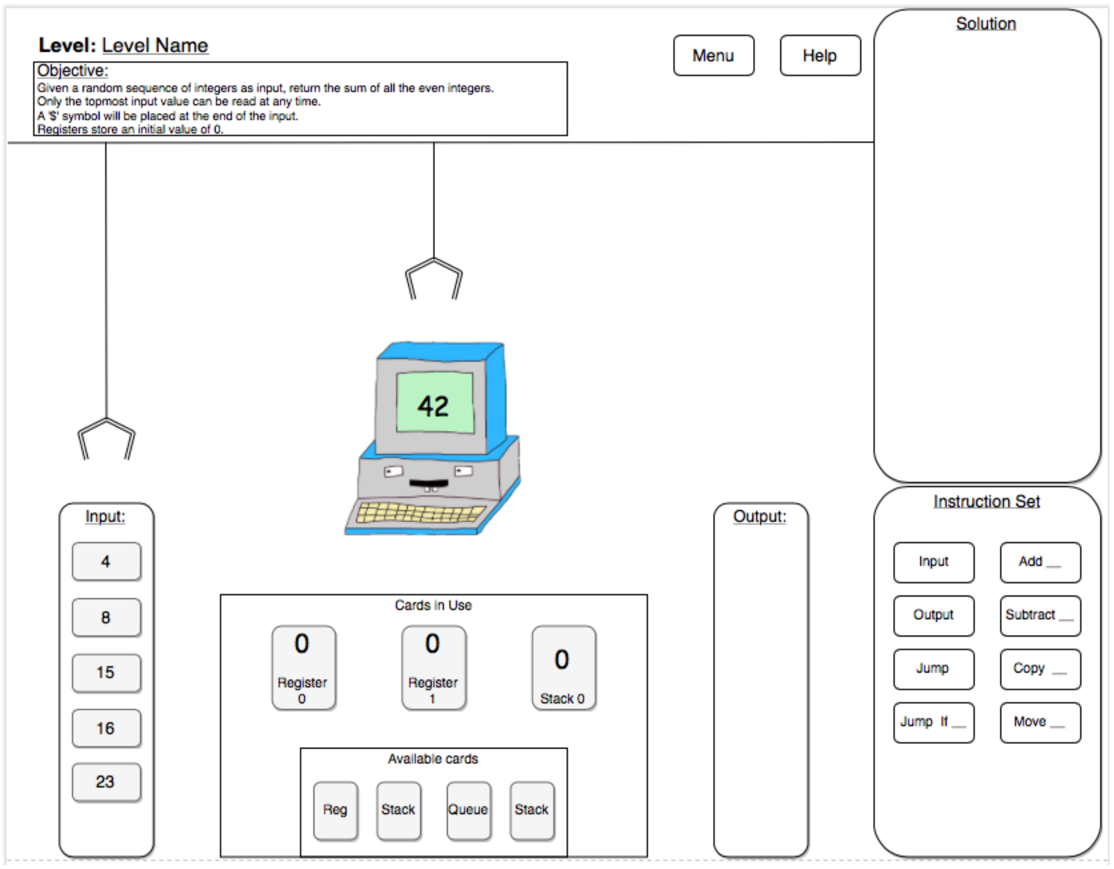
Discard Bin:
Sometimes the current value being held by the claw is no longer needed, so it can be discarded in a trash can towards the bottom right of the screen. This gets rid of the value and can free up space for the claw or registers to hold useful elements.

Puzzle Description:
Every puzzle has a clearly defined objective that should result in a single value to return upon completion, similar to a return value of a function in traditional programming. This prototype includes a "return" bin next to the discard bin, and the claw will place the final return value in the return bin to end the code execution. If the return value matches the expected value, then the solution was successful, otherwise the solution is incorrect.

Paper Prototype 2.0: Puzzle Scene Design Concept
After multiple iterations of prototype testing and team discussions to determine what works and what should be changed, we shared a solid view of how the puzzles should work, the components we wanted to include in the puzzles, and the complete instruc-

tion set. This prototype, shown in Figure 20, was designed to present a possible view of what the entire level will look like, where the components should be placed, and how they interact with each other on screen.

Figure 20: Paper Prototype 2.0


Objective:
Given a random sequence of integers as input, return the sum of all the even integers.
Only the topmost input value can be read at any time.
A '$' symbol will be placed at the end of the input.
Registers store an initial value of 0.

Menu Bar:
The menu bar stretches across the top of the screen. This contains the name of the current level along with some buttons to the right. The Menu button will pause the game and bring up a pause menu, which will present the player with options to exit the puzzle, change game options, or resume the game. The Help button can be pressed to give useful hints or tips for solving the puzzle if the player is struggling.

Level Description:
Below the top menu bar is a section for the the description of the current level ob-

jectives. This will contain text that explains the current puzzle in a concise manner.

Solution Space:
Similar to previous prototypes, the solution space on the right side is where the player constructs a solution to solve the puzzle objectives. This space includes a scrollbar on the right side to navigate long solutions.

Instruction Set:
Below the solution space is the set of instructions the player can use for the current level. The player can drag and drop these instructions into the solution space to construct their solution. Dragging an instruction from the set creates a new instance of that instruction, meaning that the instruction does not get removed from the instruction set. If the player clicks on an instruction, a description of that instruction will appear on the screen to remind the player what the instruction does and how to use it.

Game Space:
The majority of the screen will be the game space, which contains the remaining elements which are explained below. The game space is where the remaining components interact to visually simulate the steps as the player's solution is executed.

The Claw:
As each step of the player's solution is executed, the claw moves data between game components. The claw moves horizontally along the top of the game space, and can expand and retract to reach components. The claw is only used for transportation of data between two points, it does not hold onto anything when it is idle. This means that any data that was picked up will be put down somewhere else before moving to the next step. Data is not modified while it is in the claw.

Computer:
The central figure of this prototype design is a sentient computer as a sort of Actor. The computer holds the current value that is displayed on its monitor, and it performs computations and manipulates data. The claw can deliver data to the computer or pull data from the computer to move to another game component, such as a data structure or output.

Input:
Input is received from a vertical stack or conveyor belt system on the bottom left of the game space. The claw can pick up the topmost input value and move it somewhere else.

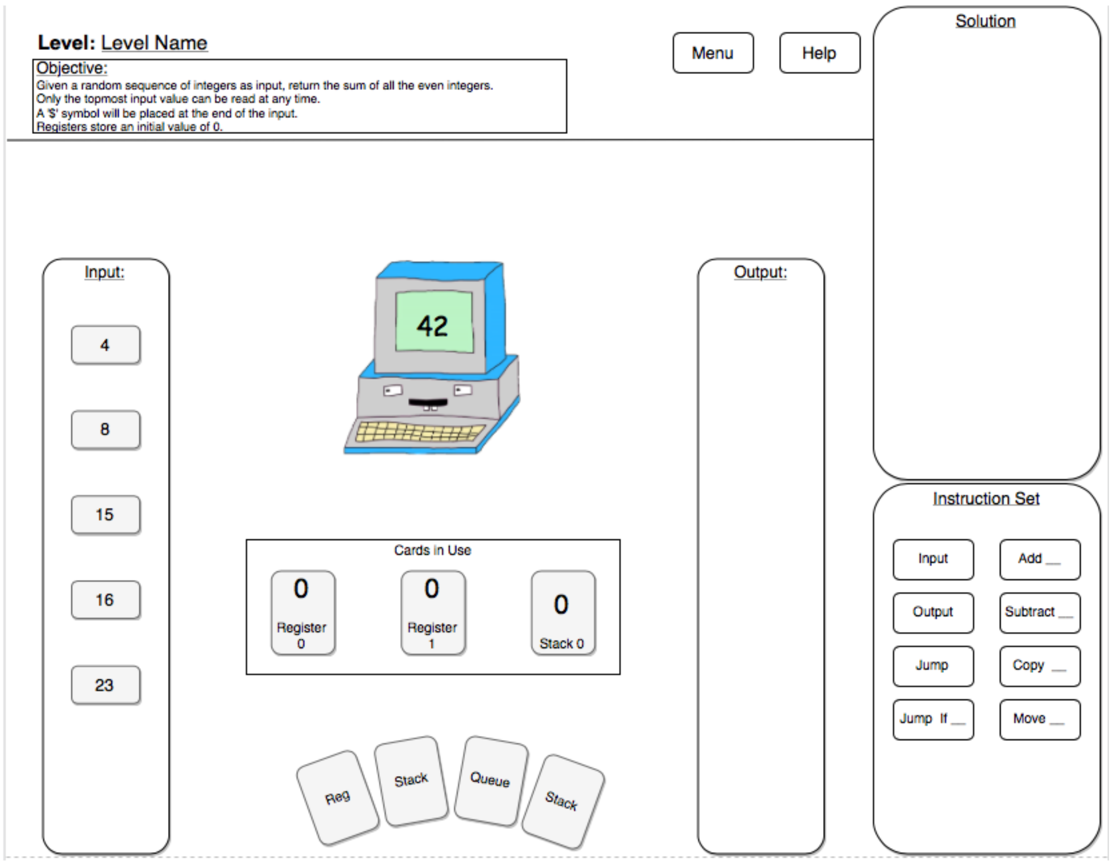
Output:
Output is delivered to a vertical stack or conveyor belt system on the bottom right of the game space. The claw can place a value onto the top of the output, and any prior output values slide downward or off the screen.

Card Space:
The card space is at the bottom of the game space between input and output, and it displays the data structure cards that are available or in use. The player can "play" a card by dragging and dropping an available card to the "in use" section. Cards being used will display their contents, or if the data structure is too large to display, the player can click on the card to enlarge it and view the contents. Players can also click on a card to get a description of the data structure and how to use it.

5.4 Final Unified Prototype: The Rise of Computron

After thoroughly analyzing several iterations of prototypes and many rounds of brainstorming on specific aspects and features of gameplay and game components, the team designed a unified final prototype to develop in Unity. This unified prototype, shown in Figure 21, serves as the foundation for the game to be built on, so while many components will be added and changed throughout the development process, the overall game flow and mechanics will not deviate far from this prototype. The initial Unity prototype is shown in Figure 22.

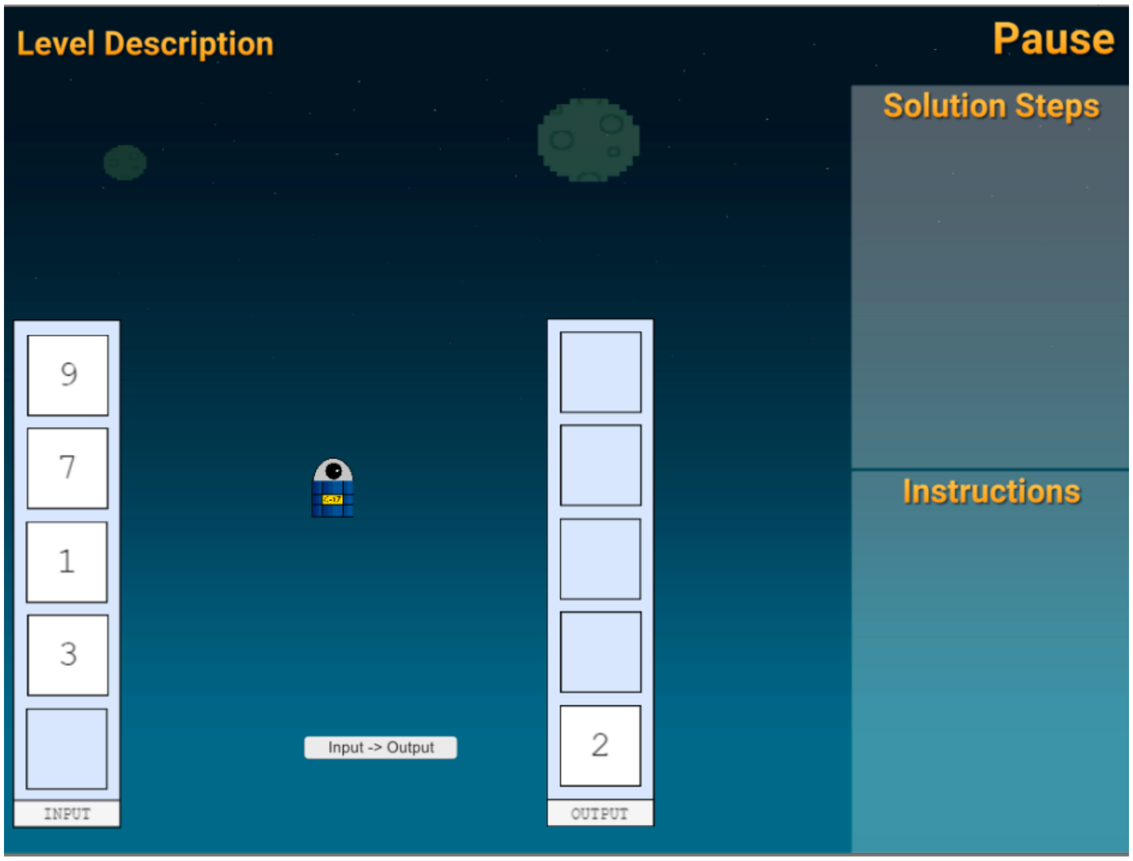
Figure 21: Unified Prototype



Level: Level Name

Objective:
Given a random sequence of integers as input, return the sum of all the even integers.
Only the topmost input value can be read at any time.
A '$' symbol will be placed at the end of the input.
Registers store an initial value of 0.

Figure 22: Unity Prototype



5.4.1 Puzzle Scene

Menu Bar:
The menu bar spans the top of the screen of the game puzzle scene to serve as a container for a few other elements of the user interface.

Level Description:
The level description provides a description of the current level objectives. This contains text that explains the current puzzle and requirements in a concise manner. The level description should fit in a relatively small space, yet clearly define what is expected from the player. Although beginner players may get confused at certain points during the game, such confusion should be caused by facing unfamiliar Computer Science concepts, and thus alleviated by attempting the puzzles and observing the results. Generally players should *not* be confused by what is expected from them, which is why the phrasing of level descriptions should be carefully considered and

tested.

Menu Button:
The Menu button pauses the game and displays a pause menu over the puzzle scene, which presents the player with options to exit the puzzle scene, change certain game options, such as volume, and resume the game.

Help Button:
The Help button can be pressed if the player is struggling or needs help approaching the current puzzle. This displays relevant hints or tips for solving the puzzle so the player can better understand how to arrive at an appropriate solution.

Solution Space:
The solution space on the right side of the screen is where players must utilize available instructions to construct a solution to the current puzzle. Players can drag instructions into the solution space, rearrange them, or remove them as they attempt to create a solution that satisfies the level objectives. While the player's solution is executing, the instruction that corresponds to the current step is highlighted to help the player understand what causes the actions simulated in the game space.

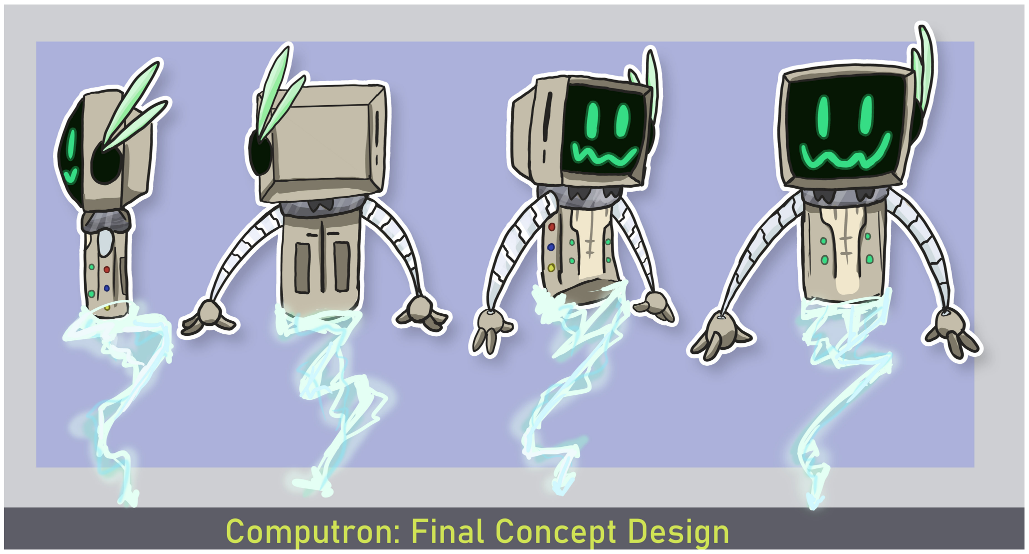
Instruction Set:
Below the solution space on the right side of the screen is the instruction set, which provides all the instructions that the player can use for the current level. These instructions can be dragged and dropped into the solution space to construct a solution for the current level. The instructions do not get removed from the instruction set when the player drags them into the solution space, instead this creates a new instance of the instruction, so that a single instruction can be used multiple times in a solution. If the player clicks on an instruction, the description of that instruction appears on the screen to remind the player what the instruction does and how to use it.

Game Space:
The game space is the largest element in the puzzle scene and takes up the majority of the total screen space, since this is where most of the action happens to visually

demonstrate how logical instructions are executed and how the components interact. The game space contains a number of key components as described below.

Computron:

Figure 23: Computron Concept Sketch



The central figure of each level, Computron is the main actor for the game. Computron is a robotic computer character that moves around to certain components of the game space to simulate the steps of the player's solution. Computron can pick up input values, store and manipulate the data, transfer it to and from other components such as registers or data structures, and bring data to the output stack. Computron has a computer monitor that displays the current value being held, and shows changes being made to the current value.

Input:
Input values are received from a vertical stack on the bottom left of the game space. This input stack works similarly to a conveyor belt delivering items. Computron may only pick up the topmost item from input, like popping a stack. When the top element of input is removed, the second element becomes the new top element, and all the items on the conveyor belt slide upwards by one space. No items can be pushed onto the input stack.

Output:
Output values are delivered to a vertical stack, similar to the input stack, on the bottom right of the game space. When outputting a value, Computron pushes it onto the top of the output stack, and any items in the stack slide down by one position, and the bottom element slides off the screen. No items can be removed from the output once they are pushed.

Card Space:
At the bottom of the game space between input and output is where data structure cards are stored and used. Each puzzle may include a small set of playing card objects, each containing a specific data structure. The possible cards are Register, Stack, Queue, and Heap. For the initial game state, the cards are not in use but are available to put into use if the player chooses. The player can do this by dragging a card from their "hand" up to a card slot above the available cards. When a card is in a slot, it becomes active and can be used and referenced by the instructions. Cards in use will display their contents, or if the data structure is too large to display, the player can click on the card to enlarge it and view the current values in the data structure. Clicking on a card that is either available or in use also provides a description of the corresponding data structure type and how to use it.

5.5 Puzzle Level Sequence

The tutorial sequence of *Computron* will introduce players to the game mechanics and allow players to acclimate to the puzzle solving workflow. The ordering of the tutorial sequence is crafted to control the rate at which players learn, and provide a smooth difficulty curve while constantly presenting new mechanics and concepts to keep player interest high.

Level 1: Ins and Outs

Learning Objective: Teach players the very basics of our user interface through a simple puzzle.

Problem: There's only a single input to output here. I hope you can handle it!

Available Instructions:	**Expected Solution:**
• INPUT • OUTPUT	1. INPUT 2. OUTPUT

Level 2: Ins and Outs pt. 2

Learning Objective: Ensure that the player understands the basic mechanics of commanding Computron by requiring them to chain the previous puzzle's solution. In addition, prepare the player to understand the value of loops.

Problem: Oh boy! Can you do that again five whole times?

Available Instructions:	**Expected Solution:**
• INPUT • OUTPUT	1. INPUT 2. OUTPUT 3. INPUT 4. OUTPUT 5. INPUT 6. OUTPUT 7. INPUT 8. OUTPUT 9. INPUT 10. OUTPUT

Level 2a (Optional): Alternator

Learning Objective: Demonstrate a nuanced mechanic of the input instruction. It is not apparent from the previous puzzles, but executing two input commands in succession will cause Computron to discard the first.

Problem: I don't like the first and third number. Just give me the other two.

Available Instructions:	**Expected Solution:**
• INPUT • OUTPUT	1. INPUT 2. INPUT 3. OUTPUT 4. INPUT 5. INPUT 6. OUTPUT

Level 2b (Optional): Triplicate

Learning Objective: Have players incorporate the idea of skipping over inputs in conjunction with the power of using loops.

Problem: I only want every third item. If you give me anything else, I'll cry. Well, the Computron equivalent of crying.

Available Instructions:	**Expected Solution:**
• INPUT • OUTPUT • JUMP • JUMP IF NULL	1. INPUT 2. JUMP IF NULL [EOP] 3. INPUT 4. INPUT 5. OUTPUT 6. JUMP 1

Level 3: Hip Hop

Learning Objective: Introduce the concept of jumping and looping.

Problem: Someone dumped a bunch of garbage in my input! Throw it all in the output, I guess.

Available Instructions:	**Expected Solution:**
• INPUT • OUTPUT • JUMP • JUMP IF NULL	1. INPUT 2. JUMP IF NULL [EOP] 3. OUTPUT 4. JUMP 1

Level 4: Flip Flop

Learning Objective: Introduce variable declaration, and present the player with their first significant challenge.

Problem: I want both of these items, but I don't like the order. Give them to me in reverse.

Available Instructions:	Expected Solution:
• INPUT • OUTPUT • JUMP • JUMP IF NULL • MOVE TO X • MOVE FROM X **Available Cards:** • Register x2	1. Play Register [0] 2. INPUT 3. MOVE TO [0] 4. INPUT 5. OUTPUT 6. MOVE FROM [0] 7. OUTPUT

Level 5: Flip Flop Hop

Learning Objective: Allow players to experiment with the new register mechanic, and guide them to generalize their previous solution.

Problem: Keep the pairs together, but give them to me in reverse. Just like before but, y'know, more times.

Available Instructions:	Expected Solution:
• INPUT • OUTPUT • JUMP • JUMP IF NULL • MOVE TO X • MOVE FROM X **Available Cards:** • Register x2	1. Play Register [0] 2. INPUT 3. JUMP IF NULL [EOP] 4. MOVE TO [0] 5. INPUT 6. OUTPUT 7. MOVE FROM [0] 8. OUTPUT 9. JUMP 2

Level 6: Triceraflops

Learning Objective: Expand player's understanding of the power of registers, and start guiding them toward seeing the value of stacks.

Problem: Since you've got pairs down, wanna try it with triples? You better.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • MOVE TO X • MOVE FROM X **Available Cards:** • Register x4	**Expected Solution:** 1. Play Register [0] 2. Play Register [1] 3. INPUT 4. MOVE TO [0] 5. INPUT 6. MOVE TO [1] 7. INPUT 8. OUTPUT 9. MOVE FROM [1] 10. OUTPUT 11. MOVE FROM [0] 12. OUTPUT

Level 6a (Optional): Triceraflops Hops

Learning Objective: Force the player to generalize their previous solution for an arbitrary number of inputs.

Problem: So you reversed a triple once, but can you do it again and again?

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • MOVE TO X • MOVE FROM X **Available Cards:** • Register x4	**Expected Solution:** 1. Play Register [0] 2. Play Register [1] 3. INPUT 4. JUMP IF NULL [EOP] 5. MOVE TO [0] 6. INPUT 7. MOVE TO [1] 8. INPUT 9. OUTPUT 10. MOVE FROM [1] 11. OUTPUT 12. MOVE FROM [0] 13. OUTPUT 14. JUMP 3

Level 6b (Optional): Quanta-Flop

Learning Objective: Move towards the idea of using multiple registers to track the data as a cumbersome task, reinforce the incoming usefulness of a stack.

Problem: These four numbers are all backwards! Place them into the output box in ascending order.

Available Instructions:

- INPUT
- OUTPUT
- JUMP
- JUMP IF NULL
- MOVE TO X
- MOVE FROM X

Available Cards:

- Register x4

Expected Solution:

1. Play Register [0]
2. Play Register [1]
3. Play Register [2]
4. INPUT
5. MOVE TO [0]
6. INPUT
7. MOVE TO [1]
8. INPUT
9. MOVE TO [2]
10. INPUT
11. OUTPUT
12. MOVE FROM [2]
13. OUTPUT
14. MOVE FROM [1]
15. OUTPUT
16. MOVE FROM [0]
17. OUTPUT

Level 7: Fat Stacks

Learning Objective: Introduce players to the Stack data structure, and demonstrate its FILO property by having them reverse the input stream.

Problem: There's a lot of inputs here. Can you put them all to the output in reverse order?

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • MOVE TO X • MOVE FROM X **Available Cards:** • Register x5 • Stack x2	**Expected Solution:** 1. Play Stack [0] 2. INPUT 3. JUMP IF NULL [6] 4. MOVE TO [0] 5. JUMP 2 6. MOVE FROM [0] 7. JUMP IF NULL [EOP] 8. OUTPUT 9. JUMP 6

Level 8: Comparatron

Learning Objective: Introduce players to conditional jumps, and show that jumps can be used to select branches of logic.

Problem: I only want the greatest - of every pair of items, that is.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X **Available Cards:** • Register x5 • Stack x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. JUMP IF NULL [EOP] 4. MOVE TO [0] 5. INPUT 6. JUMP IF GREATER [0], 8 7. MOVE FROM [0] 8. OUTPUT 9. JUMP 1

Level 9: Big Fish

Learning Objective: Allows players to generalize the concept of maximization shown in the previous problem to a more robust situation, and continue making them comfortable with using jumps to select branches of logic.

Problem: Give me the biggest number you can find!

Available Instructions and Cards	Expected Solution
Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X **Available Cards:** • Register x5 • Stack x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. MOVE TO [0] 4. INPUT 5. JUMP IF NULL 9 6. JUMP IF LESS [0], 4 7. MOVE TO [0] 8. JUMP 4 9. MOVE FROM [0] 10. OUTPUT

Level 9a (Optional): Two Fish

Learning Objective: Allows players to challenge themselves with a more difficult variant of the previous problem.

Problem: I'm greedy, give me the two biggest numbers this time.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X **Available Cards:** • Register x5 • Stack x2	**Expected Solution:** 1. Play Register [0] 2. Play Register [1] 3. Play Register [2] 4. INPUT 5. MOVE TO [0] 6. INPUT 7. MOVE TO [1] 8. INPUT 9. JUMP IF NULL 21 10. JUMP IF LESS [0], 18 11. MOVE TO [2] 12. MOVE FROM [0] 13. JUMP IF LESS [1], 15 14. MOVE TO [1] 15. MOVE FROM [2] 16. MOVE TO [0] 17. JUMP 8 18. JUMP IF LESS [1], 8 19. MOVE TO [1] 20. JUMP 8 21. MOVE FROM [0] 22. JUMP IF GREATER [1], 28 23. MOVE TO [2] 24. MOVE FROM [1] 25. OUTPUT 26. MOVE FROM [2] 27. JUMP 30 28. OUTPUT 29. MOVEFROM [1] 30. OUTPUT

Level 9b (Optional): Red Fish

Learning Objective: Originally, this level was designed to extend the concepts of the previous level. However, upon implementation in the game it was found that the solution was actually deeply difficult at this stage in the process, so the requirements were changed for it to be the last level available and it can only be unlocked if every other star in the game is earned. When all of the abilities of the game are unlocked, the challenge becomes much simpler and the goal becomes testing the player to put their resources together and find the simplest solution.

Problem: Give me the three largest numbers in the input.

Available Instructions:

- INPUT
- OUTPUT
- JUMP
- JUMP IF NULL
- JUMP IF LESS X
- JUMP IF GREATER X
- JUMP IF EQUAL X
- MOVE TO X
- MOVE FROM X

Available Cards:

- Register x5
- Stack x2
- Queue x2
- Heap x2

Expected Solution:

1. Play Heap [0]
2. INPUT
3. JUMP IF NULL 6
4. MOVETO [0]
5. JUMP 2
6. MOVEFROM [0]
7. OUTPUT
8. MOVEFROM [0]
9. OUTPUT
10. MOVEFROM [0]
11. OUTPUT

Level 10: Little Fish

Learning Objective: Encourage players to explore the concept of minimization.

Problem: I feel bad for only caring about big inputs. Give me the smallest one you can find this time.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X **Available Cards:** • Register x5 • Stack x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. MOVETO [0] 4. INPUT 5. JUMP IF NULL 9 6. JUMP IF GREATER [0], 8 7. MOVE TO [0] 8. JUMP 4 9. MOVE FROM [0] 10. OUTPUT

Level 11: Equal Fish

Learning Objective: Encourage players to explore the concept of equality.

Problem: I really like that first number. Give me every one of them that you find.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X **Available Cards:** • Register x5 • Stack x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. MOVETO [0] 4. INPUT 5. JUMP IF NULL 10 6. JUMP IF EQUAL [0], 8 7. JUMP 4 8. OUTPUT 9. JUMP 4 10. MOVE FROM [0] 11. OUTPUT

Level 12: Heap Fish

Learning Objective: Introduce players to their second data structure, the Heap.

Problem: Now I want it all - give me everything you find, and do it from largest to smallest.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X **Available Cards:** • Register x5 • Stack x2 • Heap x2	**Expected Solution:** 1. Play Heap [0] 2. INPUT 3. JUMP IF NULL 6 4. MOVETO [0] 5. JUMP 2 6. MOVEFROM [0] 7. JUMP IF NULL [EOP] 8. OUTPUT 9. JUMP 6

Level 13: Reverse Fish

Learning Objective: Show players the power of combining data structures to accomplish a new task.

Problem: Scratch that - reverse it. Give them all to me, smallest to largest.

Available Instructions:

- INPUT
- OUTPUT
- JUMP
- JUMP IF NULL
- JUMP IF LESS X
- JUMP IF GREATER X
- JUMP IF EQUAL X
- MOVE TO X
- MOVE FROM X

Available Cards:

- Register x5
- Stack x2
- Heap x2

Expected Solution:

1. Play Heap [0]
2. Play Stack [1]
3. INPUT
4. JUMP IF NULL 7
5. MOVE TO [0]
6. JUMP 3
7. MOVE FROM [0]
8. JUMP IF NULL 11
9. MOVE TO [1]
10. JUMP 7
11. MOVE FROM [1]
12. JUMP IF NULL [EOP]
13. OUTPUT
14. JUMP 11

Level 14: Pivot Fish

Learning Objective: Introduce players to the Queue data structure and highlight their usefulness of keeping things in order.

Problem: Pivot time! Save the first item and output everything smaller. Then give me the pivot and all the ones that were larger. Keep 'em in the order you find 'em!

Available Instructions:	**Expected Solution:**
• INPUT	1. Play Register [0]
• OUTPUT	2. Play Queue [1]
• JUMP	3. INPUT
• JUMP IF NULL	4. MOVE TO [0]
• JUMP IF LESS X	5. INPUT
• JUMP IF GREATER X	6. JUMP IF NULL 12
• JUMP IF EQUAL X	7. JUMP IF LESS [0], 10
• MOVE TO X	8. MOVE TO [1]
• MOVE FROM X	9. JUMP 5
Available Cards:	10. OUTPUT
	11. JUMP 5
• Register x5	12. MOVE FROM [0]
• Stack x2	13. OUTPUT
• Queue x2	14. MOVE FROM [1]
• Heap x2	15. JUMP IF NULL [EOP]
	16. OUTPUT
	17. JUMP 14

Level 15: Copy Constructor

Learning Objective: Introduce players to the copy to and copy from instructions, allowing them to make duplicates of data values.

Problem: I like this number so much – I want two of them!

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X • COPY TO X • COPY FROM X **Available Cards:** • Register x5 • Stack x2 • Queue x2 • Heap x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. COPY TO [0] 4. OUTPUT 5. COPY FROM [0] 6. OUTPUT

Level 16: ALU

Learning Objective: Introduce players to the add instruction, allowing them to perform addition and, by extension, multiplication on data values.

Problem: These numbers are good, but they could be twice as good. Give me the double of each!

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X • COPY TO X • COPY FROM X • ADD X **Available Cards:** • Register x5 • Stack x2 • Queue x2 • Heap x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. JUMP IF NULL [EOP] 4. COPY TO [0] 5. ADD [0] 6. OUTPUT 7. JUMP 2

Level 17: Mega-Multiplier

Learning Objective: Encourage players to achieve multiplication through repeated addition.

Problem: Why stop at double? Let's make it the quadruple of each number!

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X • COPY TO X • COPY FROM X • ADD X **Available Cards:** • Register x5 • Stack x2 • Queue x2 • Heap x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. JUMP IF NULL [EOP] 4. COPY TO [0] 5. ADD [0] 6. COPY TO [0] 7. ADD [0] 8. OUTPUT 9. JUMP 2

Level 18: Ultra-Multiplier

Learning Objective: Encourage players to achieve exponential multiplication through repeated addition and storage of the results.

Problem: I want MORE! Give me the number times sixteen this time.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X • COPY TO X • COPY FROM X • ADD X **Available Cards:** • Register x5 • Stack x2 • Queue x2 • Heap x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. JUMP IF NULL [EOP] 4. COPY TO [0] 5. ADD [0] 6. COPY TO [0] 7. ADD [0] 8. COPY TO [0] 9. ADD [0] 10. COPY TO [0] 11. ADD [0] 12. OUTPUT 13. JUMP 2

Level 19: Mister Negative

Learning Objective: Introduce players to the subtract command and encourage them to explore the idea of negation.

Problem: I ate too much and need to balance out. For every number here, give me its negative equivalent.

Available Instructions: • INPUT • OUTPUT • JUMP • JUMP IF NULL • JUMP IF LESS X • JUMP IF GREATER X • JUMP IF EQUAL X • MOVE TO X • MOVE FROM X • COPY TO X • COPY FROM X • ADD X • SUBTRACT X **Available Cards:** • Register x5 • Stack x2 • Queue x2 • Heap x2	**Expected Solution:** 1. Play Register [0] 2. INPUT 3. JUMP IF NULL [EOP] 4. COPY TO [0] 5. SUBTRACT [0] 6. SUBTRACT [0] 7. OUTPUT 8. JUMP 2

6 System Design

6.1 Developer Tools

When brainstorming the requirements for *Computron*, a great deal of thought was put into the selection of developer tools. As the team has learned, choosing the wrong tool can be a very expensive mistake. Two of the most vital choices we made, the engine and the version control system, are documented in this section.

6.1.1 Game Engine

Making games from scratch is a long and difficult practice. Like most small development teams, we decided very early on that making a custom framework or engine would be inappropriate for the scope of the project. The team opted to instead make use of a commercial game engine. Using a commercial game engine puts a huge set of tools at our disposal, and allows for rapid prototyping and iterations. However, choosing which game engine to use is still a non-trivial issue. In the realm of commercial game engines, there are two major options: Unreal Engine 4 (developed by Epic Games) and Unity (developed by Unity Technologies). Both engines are excellent choices, and we carefully weighed their positive and negative attributes.

Unreal Engine 4 is a high power tool. Written and scripted in C++, games shipped with Unreal can push the limits of the player's computer hardware. In addition Unreal is open source, includes better support for version control, and provides a very mature visual scripting system to aid in producing game logic. However, concerns were raised that Unreal would be too powerful for the needs of our project. Unreal's higher graphical fidelity is irrelevant in a sprite-based 2D application, and the engine's generally steeper learning curve threatened to greatly slow our progress in the project's early stages. In addition, Unreal's huge C++ code base can often result in very long compile times, especially for older development machines.

Unity is also an extremely advanced tool. Written in C++ and scripted with C#, games made with Unity can generally be prototyped and iterated much faster than Unreal competitors. Unity is easier to learn, has better documentation and more tutorials, and has vastly superior support for 2D titles. However, Unity presents its own set of drawbacks as well. Poor support for modern version control systems, the lack of a visual scripting tool, closed source engine code, and a dated pricing model were all worrisome issues.

After extensive discussions, the team unanimously decided that *Computron* will be built using Unity, version 2019.2.12.f1. Improved support for 2D titles and superior ease of use were the deciding factors.

6.1.2 Version Control System

Choosing Unity as our game's engine made choosing the right version control system for *Computron* an extremely difficult process. The Unity Editor's heavy use of binary files, and its tendency to generate lots of temporary files between program runs made integrating traditional version control systems like Git or Subversion difficult. We considered many solutions to this problem.

Our first consideration was Unity's own solution to it's difficult relationship with version control systems, Unity Collaborate. Unfortunately, even a cursory glance over its features proved it to be an insufficient solution to the teams requirements. Unity Collaborate lacks many basic features of any modern version control system, including:

1. Branching and Merging.
2. Any out of Editor interface to the repository.
3. Single file reverts/restores.
4. Support for more than 3 team members without paying money.
5. Access to more than 1GB of server space without paying money.
6. Ability to host a dedicated server.

Instead of surveying further, the team decided to move forward with Git as our version control system. Using Git worked extremely well during the first sprint, up until our first major integration. It then became abundantly clear that Git support was much worse than we had originally evaluated it to be. Even after configuring the editor to use plain-text meta files and assets, Git was entirely unable to merge non-conflicting changes in a single Unity scene. Workarounds existed to make Git compatible with these scene merge conflicts, but the process did not work consistently on both Mac and Windows development machines, of which our team is equally using both.

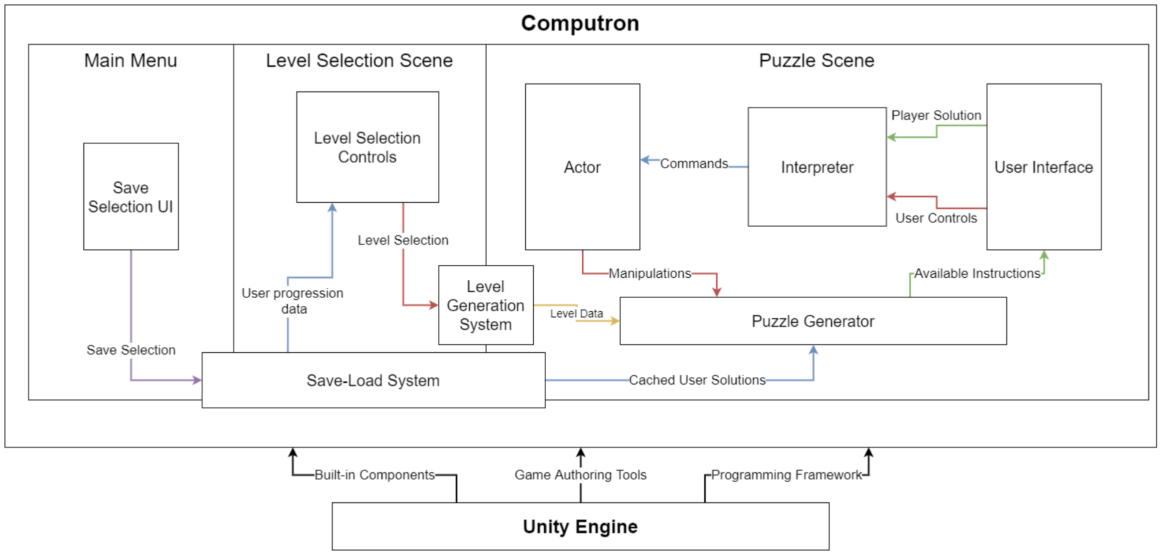
In place of hacking together a Git integration, the team decided to migrate to Perforce. Perforce is a centralized version control system that is currently the dominant version control system in the game development industry. Perforce is free for small student teams, and Unity has minor built in support. Up until recently, Unity's Perforce integration was a premium only feature, but it has now been quietly released for the free tier of the editor. Unlike Git, Perforce does not have a standard (and free) cloud platform for hosting repositories. Perforce administrators are expected to find their own server space. *Computron's* perforce repository is hosted on a free Amazon Web Service EC2 Linux server. The AWS server has proved highly accessible, configurable, and performant.

Our Perforce configuration has proven stable on both Windows and Mac development machines, and has allowed us to collaborate without any major incidents. Perforce has an extensive feature set for working with the binary files often found in game repositories, and allows users to lock files and force sequential edits when making breaking changes. At major milestones, the team pushes a copy of the current build of the project to the Git repository.

6.2 Overall Project Structure

At the highest level, our game is organized into several major systems that communicate between three Unity Scenes: the Save-Load system, the Level Generation System, the Puzzle System, the Puzzle User Interface, the Interpreter, and the Actor. A graphical representation of the relationship between these systems is show in figure 24.

Figure 24: Computron System Overview



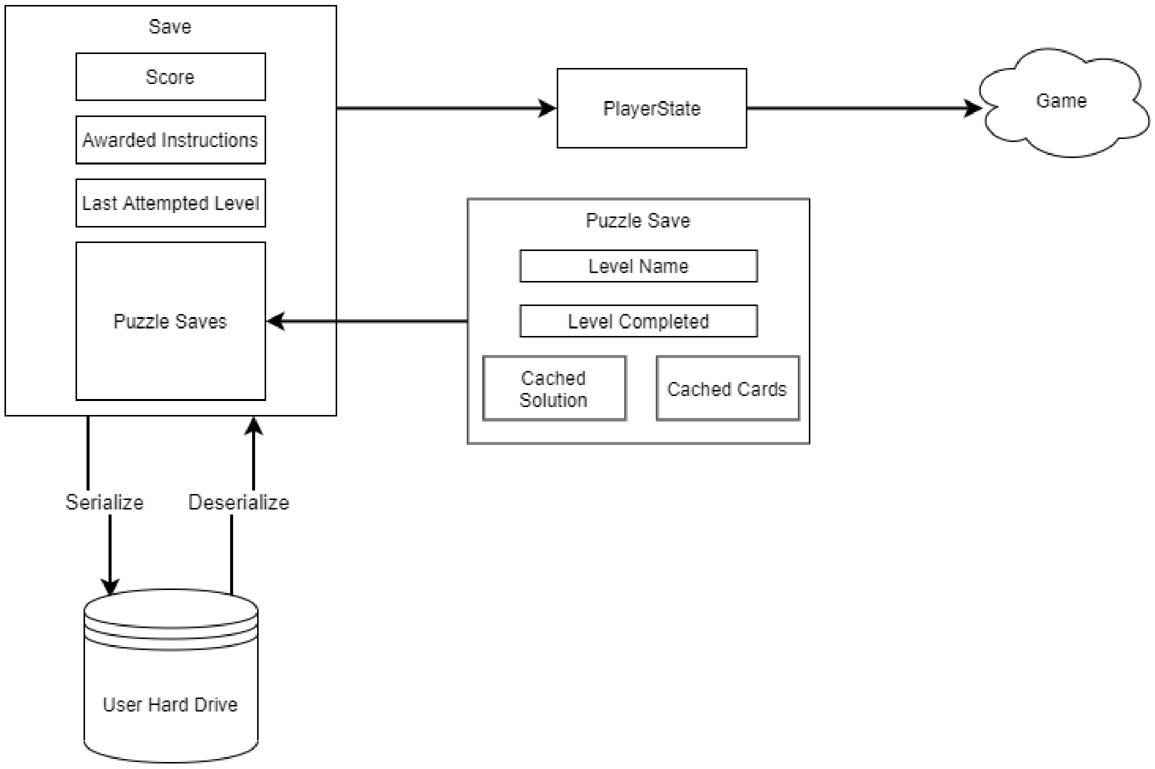
A more detailed explanation of all of these systems can be found throughout the remainder of this section.

6.3 Save Load System

6.3.1 Overview

The Save-Load system is the backbone of *Computron*. It is responsible for serializing and deserializing all the relevant player data needed to reconstruct play sessions. The system currently supports up to three player save slots to store game data. This restriction is an artificial one, as we only chose to allow the UI of the main menu to address three slots. The underlying mechanisms of the system can handle arbitrarily many save slots, as long as the player is given an interface to select them. An overview is provided by 25.

Figure 25: Save-Load System Overview



As hinted to above, the only place in which the player actively interacts with the Save-Load system is at game startup. After hitting play, the player will be presented with a choice of three save slots, and the necessary UI to create, delete, or select a save in any of the three slots.

6.3.2 Data

In this section you will find an enumeration of all the information required to restore user play sessions. Due to how this information is generated and captured, changes to the game's level layout can corrupt these save files and cause unexpected behavior in the level selection scene and puzzle scenes. It is highly recommended that developers clear outdated save files when adding, removing, or otherwise altering levels.

The top-level save object is simply called a Save. Below is an enumeration of the data captured in a save:

- Player Score
- Awarded Instructions
- Last Attempted Level
- A collection of puzzle-specific saves for each puzzle the player has attempted

Top-level Saves defer the responsibility of tracking per-puzzle player data to a specialized serializable object cleverly know as a Puzzle Save. Puzzle Saves are generated and maintained on a per-level basis. When a player visits a level for the first time, a new entry is added to their save file for that level. This entry includes the following information:

- The level's name
- Whether the level has been completed
- A list of what instructions the player left in the solution window
- A list of what cards the player left in the play area
- Which challenge stars the player has earned

6.3.3 Scene Traversal

After all this data is collected, and loaded (from either a binary or JSON save file) it is transferred to a PlayerState GameObject which travels between all scenes. This PlayerState object provides an interface for level generation systems to read and write from player session data to configure the scenes and report significant game events.

In the level select scene PlayerState is used extensively. PlayerState is queried to color the level graph to reflect node availability, restrict node access to enforce player progression, and update the scene's heads up display.

In the puzzle scene, it was imperative that players be able to revisit puzzles and see the solution they left there. PlayerState is used to regenerate those saved solutions.

6.4 Level Generation System

6.4.1 Overview

Player save information is not the only information to traverse scenes. In order to allow 25 different puzzles to be displayed on the same Unity scene, *Computron* makes heavy use of an enumeration of puzzle data, and a Puzzle Generator that processes that data to configure the puzzle scene for the selected puzzle.

This information is embedded in all of the GameObjects used to create the level select graph. The nodes of the graph store vital information for both the front end and the back end of this scene. The root GameObject of each node contains all of the data required to maintain the level graph. This includes an adjacency list of neighbors, components required to render and draw lines between the nodes, and an integer value specifying the required score to make that node traversable. Attached as a payload to each of these nodes is a child GameObject that contains the enumeration of all the data required to generate the puzzle level that node represents.

6.4.2 Data

The enumeration of all the data required to generate a level can be found via the public members of the Puzzle Data class. This data includes:

- Puzzle name
- Puzzle prompt
- Puzzle hints
- Which Output Generator to use
- Number of each card type allowed in the puzzle
- Which challenge stars are available for the level
- The score threshold for each enabled challenge star
- A flag to generate random input
- A seed for the input random number generator
- A seed for the card address random number generator
- The number of inputs to generate for this puzzle

- (Optional) an explicit input sequence to use in place of random numbers
- Instruction filtering flags to enable or disable parts of the CAL
- A Tutorialization script to specify an interactive tutorial sequence

Manipulating the data structure above and correctly connecting a prefabricated map node to the level graph are the only two steps required to add a basic puzzle to the game.

6.4.3 Scene Traversal

In order to feed puzzle information to the puzzle scene, that data object of a selected level node is extracted from the instance of the level graph and persisted to the Puzzle Scene for puzzle generation.

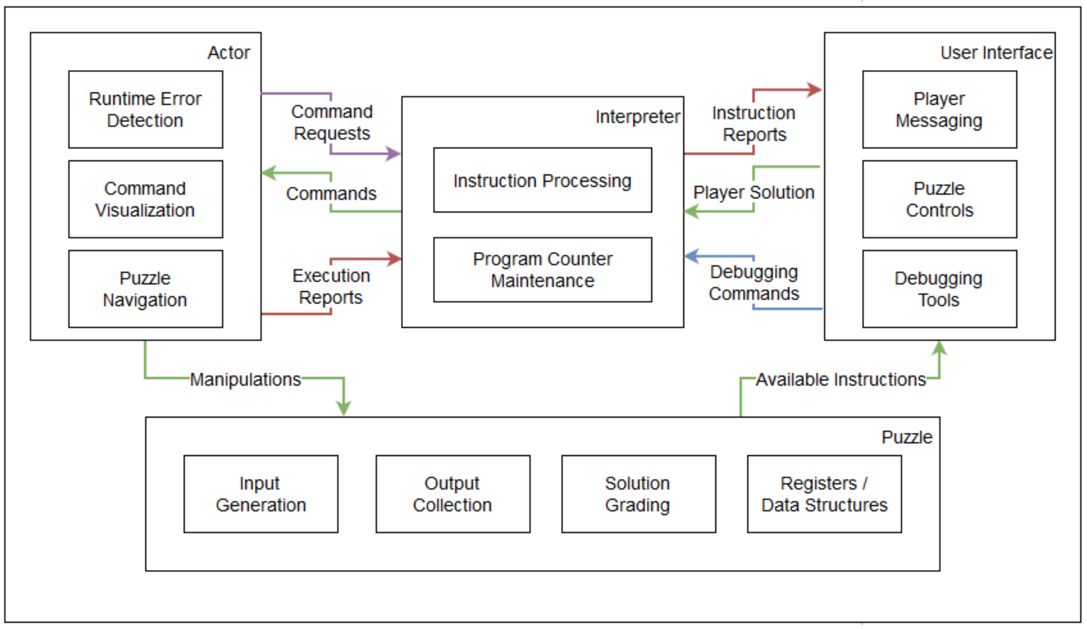
6.5 Puzzle System

6.5.1 Overview

Our puzzle system will be responsible for formally defining the nature and functionality of the puzzles in our game.

The following diagram, Figure 26, gives a general overview of the components of the puzzle system and how they communicate with one another.

Figure 26: Puzzle System Overview



The responsibilities of the puzzle system are as follows:

- Present input to the player
- Determine the expected output the player is responsible for generating
- Offer allowed operations the player can perform to solve the puzzle
- Load an initial puzzle scene specific to each puzzle level
- Save a puzzle scene state for a specific level when the player leaves
- Load saved puzzle scene states for each level a player has previously attempted

Here are the main components of the Puzzle Scene that rely on the Puzzle System:

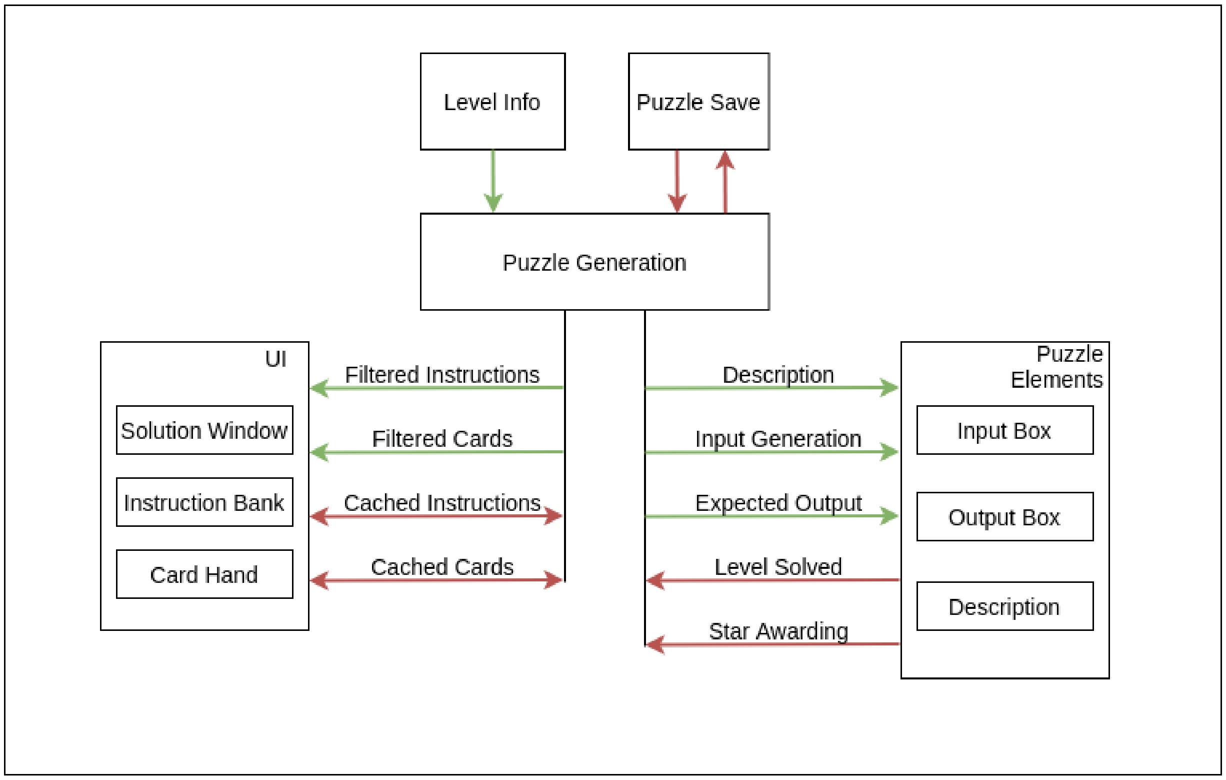
- Input / Output (I/O)
  - Input Box
  - Output Box
  - I/O Data (numbers)
- Game Cards
  - Player Card Hand
  - Card Slots
- Instruction Area
  - Instruction bank

Note that while the Game Cards and the Instruction Area fall under the User Interface System, there are certain parts of these components that rely on the Puzzle System's level generation functionalities. Only that relationship will be mentioned here. Refer to the User Interface System for a more detailed look at these game components.

6.5.2 Puzzle Level Generation

When the player chooses a level, the puzzle system controls which game objects get loaded in to the puzzle scene. This process is handled by a puzzle generation script as seen in Figure 27.

Figure 27: Puzzle Level Loading/Saving



The puzzle generator uses two different kinds of data: initial puzzle data and saved puzzle data.

The initial puzzle data is the information that the scene will always need to load:

- Input stream
- Expected output
- Level description
- Allowed instructions
- Allowed cards
- Number of each allowed card
- Hints
- Earnable awards
- Award requirements

This information varies from level to level. It is retrieved from a puzzle data script attached to a game object that travels from a selected map node from the level selection scene. This can be thought of as a base state of the puzzle level. If the player has never attempted the level, only this data will be used when generating it.

Saved puzzle data is information that the puzzle generator needs to load from a previous attempt at the puzzle level. It includes:

- Instructions left in solution window
- Which instructions have been seen before
- Cards left in card played area
- Earned awards
- Was the puzzle solved

All of this data comes from a player state script which controls the player's save state that is in use.

6.5.3 Puzzle Level Caching

When the player decides to leave a puzzle level (via exiting the game, returning to the level menu, etc.), the puzzle system saves the state of the current level by using the player state interface to write to the correct save file. This flow of information can be seen in 27.

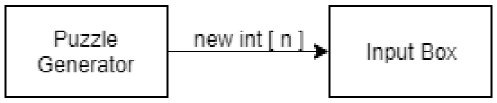
All of the data that is saved is the same as the data that is loaded from previous puzzle attemps:

- Instructions left in solution window
- Which instructions have been seen before
- Cards left in card played area
- Earned awards
- Was the puzzle solved

6.5.4 Input

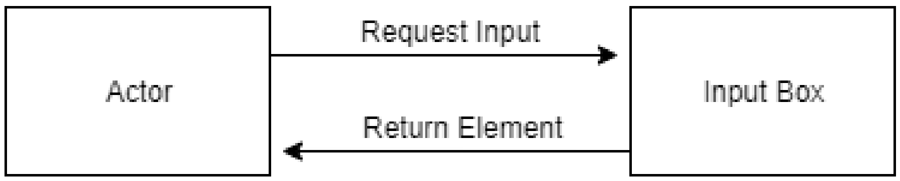
The Puzzle System is in charge of filling the puzzle Scene's input box with data that is necessary for the chosen level, shown by Figure 28. The puzzle generator can either randomly generate an array of n numbers or use a developer specified input stream to fill the input box with corresponding data.

Figure 28: Puzzle System: Input Overview



The puzzle system is responsible for giving the Actor the correct data from the input box when it is needed. The Actor can make an input request to the input box. The system will simply return the top-most element from the input box to the Actor entity, along with the numerical data that is associated with it behind the scenes (see Figure 29).

Figure 29: Input-Box/Actor Communication



Note that if the Actor makes an input request to an empty input box, the puzzle system will return NULL.

The input box has a fixed number of input items that can fit inside (5 in our examples). Sometimes the size of the input stream can be larger than the size of the input box itself. In this case, the input box works as a queue. If we had an input stream of size six, the sixth element would be hidden until the top-most element of the input box is removed by the Actor. All of the elements below will move up a slot and the hidden sixth element would then appear at the bottom, as shown in Figure 30 and Figure 31.

Figure 30: Input-Box Overloading (before)

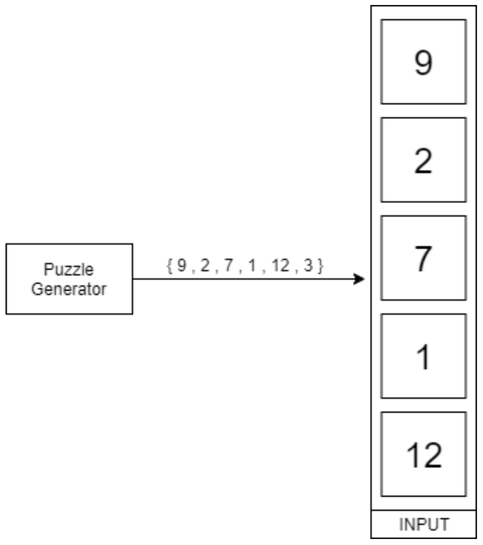
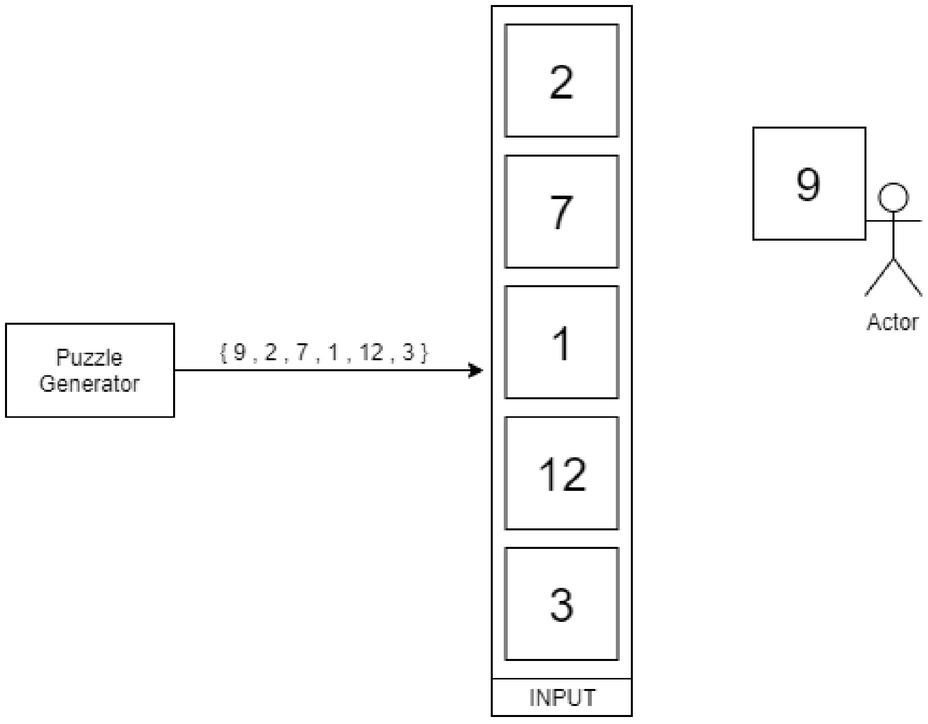


Figure 31: Input-Box Overloading (after)



If the input stream is the same size as the input box, or if the Actor has removed enough numbers from a larger stream and there are no more slots to fill, the input numbers behave only slightly differently. Figure 32 and Figure 33 show how the numbers below will move up a slot, like before, but the bottom slot will be left empty.

Figure 32: Input-Box Same Size (before)

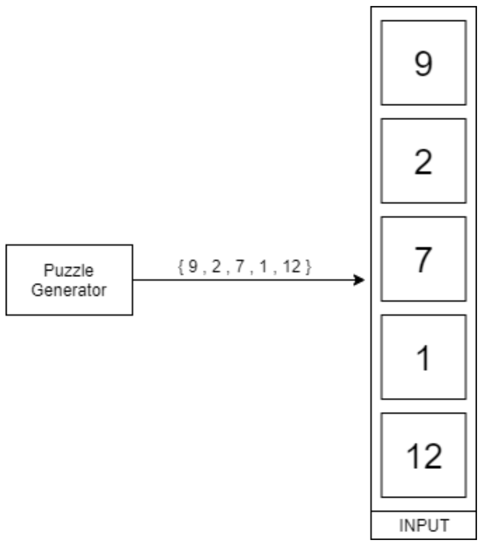
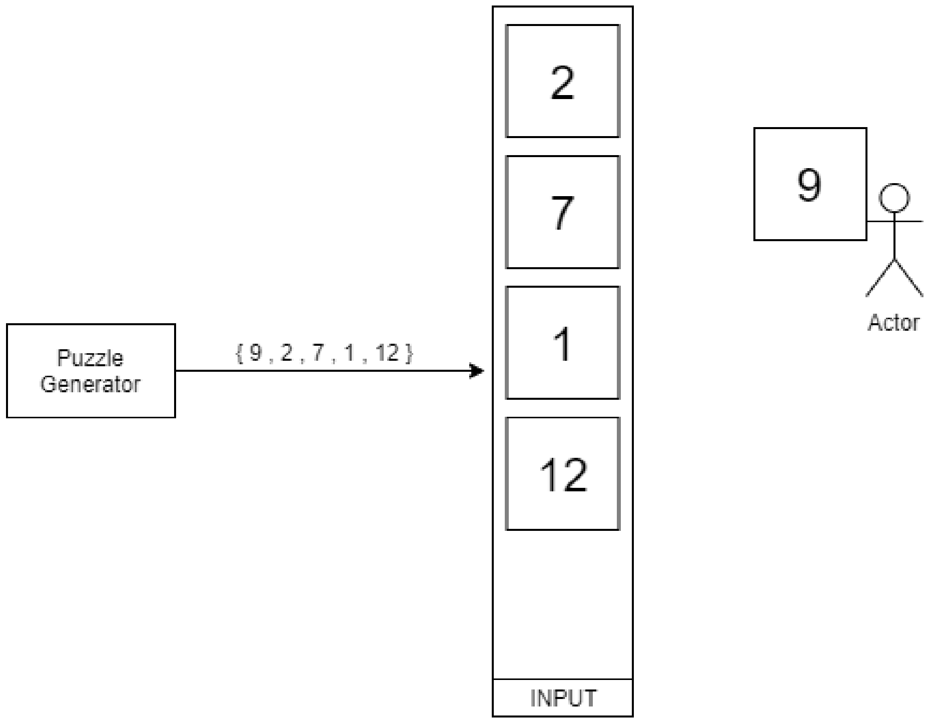


Figure 33: Input-Box Same Size (after)



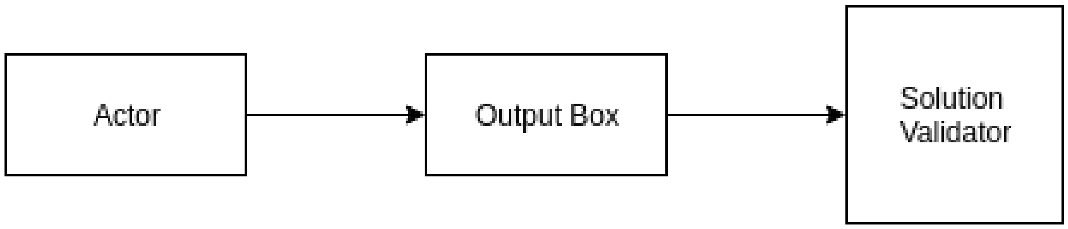
6.5.5 Output

In regards to output, the Puzzle System handles the interaction points between:

- The output box
- The Actor
- The output validator

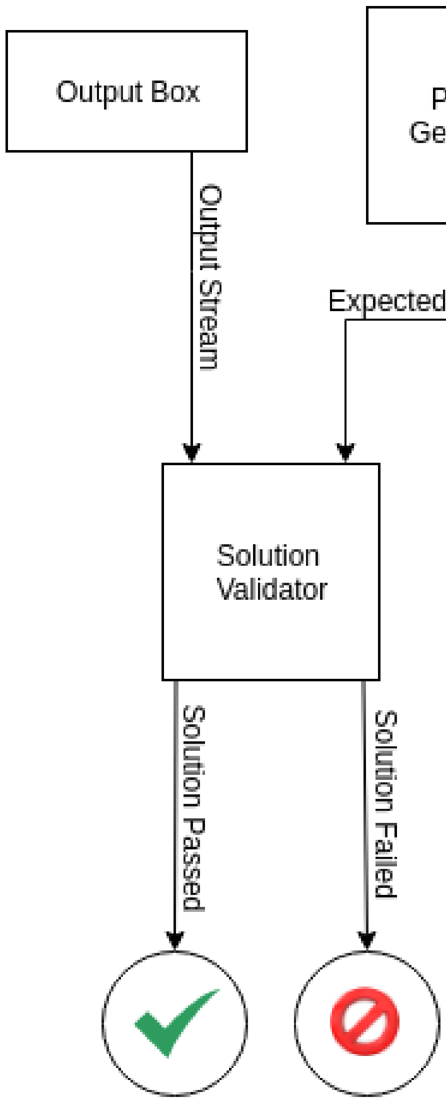
The Actor, after receiving an output instruction from the Interpreter, places the data item from its hands to the output box. After all of the instructions are run, the entire contents of the output box are submitted for validation, as shown in Figure 34.

Figure 34: Output Interaction Points



A solution generation script generates the expected output based on the input that was generated for that puzzle. An output validation script takes the contents of the player's output box, their solution, and compares it against the expected output. If the contents match, the player has solved the puzzle. If the contents do not match, the player's solution did not solve the puzzle. The purpose for this confirmation of output matching only is to allow the player to configure different algorithms that reach the same solution, and inherently present a challenge to find the ideal solution to garner the best possible score. Figure 35 shows the decision process of validating a solution.

Figure 35: Output Validation



6.5.6 Game Cards

The Puzzle System makes use of the puzzle generator in order to tell the user interface which game cards are placed in certain positions on the game board when beginning or loading a level. Allowed cards are retrieved from the current level's puzzle data object. Card positions are pulled from the player state save file. A card position not only indicates if a card is in the player's card hand or the puzzle board's card slots, but also the position within either of those areas.

The card hand is filled during initialization. The puzzle data object will contain the following:

- Number of register cards
- Number of stack cards
- Number of queue cards
- Number of heap cards

The initial content of the card hand is based on this data, although the locations of these cards may be changed later on. Cards can move between two different puzzle scene areas:

- Card hand area
- Card played area

Any update to a card's position will be reflected when returning to that level. When a level is exited, all of the cards locations on the game board are saved to that level's save file. When a level is loaded at another point in time, the card objects will be instantiated like above, but also placed at their saved locations.

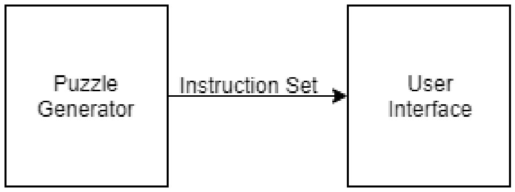
6.5.7 Instructions

The puzzle scene tells the instruction editor component of the User Interface:

- Which instructions the player can use for the current level
  - These will initially appear in the instruction bank
- Which instructions were left in the editor during a previous attempt at the puzzle level
  - These will appear where they were left in the editor previously

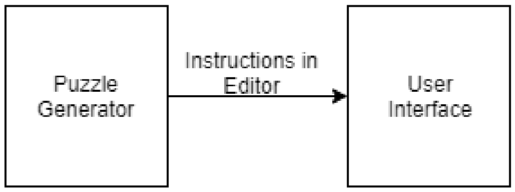
The puzzle level's puzzle data object contains the information that the User Interface needs to present the player with the appropriate instructions for a given puzzle level. The puzzle generator reads the puzzle data and passes the instruction set off to the User Interface as shown in Figure 36.

Figure 36: Loading Instruction Set



The player's save file tells the user interface which instructions were left in the editor (Figure 37). The initial state of the puzzle level leaves the editor empty. Otherwise, the editor is populated with the same instructions that were left during a previous attempt. This also includes final solutions that actually solved the puzzle level.

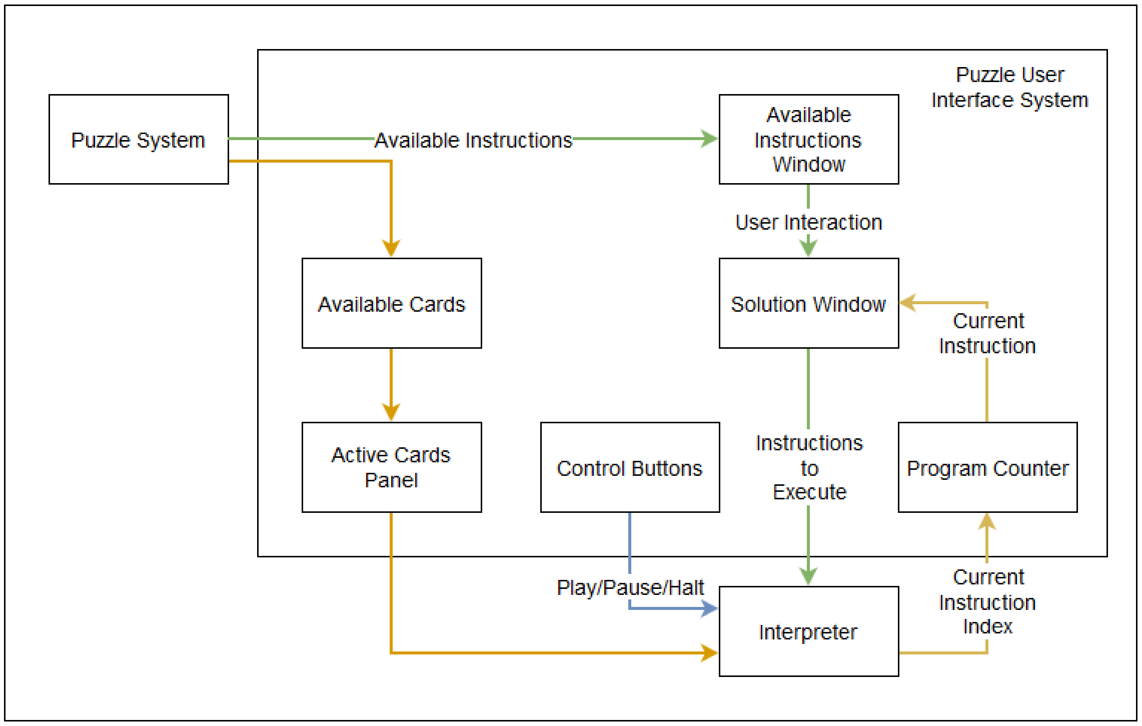
Figure 37: Loading Instruction Editor State



6.6 User Interface System

The User Interface (UI) is one of the most important pieces of *Computron*, as it is responsible for providing players with the primary means of interacting with the game in an efficient and enjoyable way. The UI presents the user with controls to navigate between scenes and solve puzzles, and it is the main source of text-based information that enables the user to learn and proceed with the game. An overview of the major components of the UI for the puzzle scene can be seen in Figure 38.

Figure 38: UI System Overview



6.6.1 Information Dissemination

Controlling the flow of information to the player presented both technical and design issues to the team. Providing too many hints, prompts, and explanations would quickly turn the game into a glorified textbook; however, providing too few would make the game feel frustrating and inaccessible to our target audience. The User Interface was tasked with providing the right amount of information to the player in an easy to follow sequence.

The UI system communicates with other major game systems to get relevant information, and the UI is responsible for delivering this information to the player at the appropriate times. The data-driven approach of the puzzle system allows it to provide the UI with unique puzzle data for each level of the game. At the start of each level, the UI retrieves from the puzzle system the data that is used to fill out the level's description, prompts, and hints.

6.6.2 UI Control System

The puzzle scene contains many UI elements that can be grouped together based on common features, and during development it became clear that certain game states such as tutorialization should trigger similar behaviors (such as focusing, disabling, highlighting, etc.) for entire groups of UI elements at once. With the significant amount of UI elements in the puzzle scene plus the need for special UI behaviors triggered by the game state, it became necessary to implement a system for giving similar UI elements shared abilities, and to create a centralized means of controlling all UI elements from a single controller. This need was filled by the UI Control System that consists of two primary classes: UIController and ControllableUIElement, with multiple subclasses that inherit from ControllableUIElement.

The ControllableUIElement class serves as an interface for the UIControl and UIControlGroup classes. The UIControl class provides generic UI elements with a few simple methods to bring the object in or out of focus and control whether the object is enabled for the player to use or not. The UIControlGroup class is used for controllable UI elements that also contain child objects that are controllable UI elements; it serves as a sort of middle manager. A UIControlGroup can be controlled with the behaviors of a ControllableUIElement, but the UIControlGroup also tracks all of its child objects with a ControllableUIElement component. When a UIControlGroup method is called, the parent group object also has its children run their equivalent method. Although the generic UIControl class is appropriate for most individual UI objects, there are some special UI elements that need slightly modified behaviors. The CardUIControl, JumpCommandUIControl, and CardCommandUIControl classes inherit from the UIControl class and override the standard methods to fit their special needs.

While all of the ControllableUIElement objects have been granted shared capabilities, they still need to be controlled by a master component. The UIController class creates a centralized authority that coordinates and controls all ControllableUIElements. The UIController finds all ControllableUIElement components and has many methods to control these UI elements or groups during runtime based on the game state.

6.6.3 Instruction Awarding

Upon opening a level in the puzzle scene, the Puzzle Generator tells the User Interface which instructions, if any, are being encountered for the first time. Whenever a level is introducing the player to instructions they have not seen before, a sequence of “instruction awarding” is triggered to aid the player. The instruction awarding sequence presents a prominent UI text window that informs the player that new instructions are available, and each new instruction is represented as a “card” object within the instruction awarding window. The UI Controller brings the instruction awarding panel into focus, while blurring and darkening other portions of the scene. The player must click on an instruction card to reveal its contents, which includes the name of the new instruction and a description of how it works. Once the player has clicked on every new instruction, they are able to continue with the level.

6.6.4 Tutorials

Tutorials are designed to provide novice players with a high level of guidance necessary for them to learn the basics of how to play the game and familiarize them with puzzle mechanics. If a level includes both a tutorial and new instructions, then the tutorial sequence will begin as soon as instruction awarding has been completed. Tutorials provide guidance via a specialized UI text window, and call on the UI Controller to focus and highlight particular UI elements that the player should interact with to complete the tutorial. These tutorials often have multiple steps to complete, so as soon as one step is completed, the UI text changes and the UI Controller focuses on different UI elements that are relevant to the next step. Once the tutorial sequence is complete, the player can continue with normal puzzle gameplay.

6.6.5 Solution Building and Execution

A fairly complex custom group of game objects that serves a significant role in the puzzle scene is what we call a "dynamic scroll view". The puzzle scene contains two instances of dynamic scroll views: the Instruction Cache and the Solution Window.

The root object of a dynamic scroll view is a frame that contains a modified scroll view with many more features than a normal Unity scroll view. One of the key features of the dynamic scroll view is the ability for players to add, remove, and reorder its contents during runtime. The dynamic scroll view is meant to contain modified UI buttons with custom code to give the buttons drag and drop behaviors. These drag and drop behaviors allow the player to click and drag the object, move it around with the mouse, then drop it into an object with the Instruction Container script which allows it to catch draggable objects. The special objects with the drag and drop behaviors became the foundation of the UI instruction objects that are used in the Instruction Cache and the Solution Window.

When draggable instruction objects are dragged over a dynamic scroll view, a blank slot spawns in the scroll view content panel as a placeholder for where the button would be placed if it is dropped. If there are other objects in the scroll view's content panel, dragging the instruction object up or down over the scroll view will also move the placeholder slot to indicate where the object will snap to when dropped. Instruction objects can also be dragged from one dynamic scroll view and dropped in another, which will become the instruction object's new parent. This capability is what allows instructions from the instruction cache to be placed into the solution window.

In order to complete a level, the player must use the available instructions to construct a solution that satisfies the level prompt. The set of available instructions varies for each puzzle, so the Puzzle System tells the UI System which instructions should be included in each level. For each available instruction, the UI system spawns a corresponding instruction prefab inside the instruction cache scroll view. These instruction objects are primarily UI elements but they also store necessary information for the instruction's operation, which will be necessary later in the solution building process. When an object in the instruction cache is dragged, it creates a clone of itself to leave in its place so the cache cannot run out of instructions. If an instruction object is dropped anywhere other than the solution window, it destroys itself.

Once the player is satisfied with the solution they've constructed, they can click on the play button in the control panel at the bottom of the puzzle scene to begin the sequence of events that allows Computron to simulate execution of the solution. When the play button is invoked, the Interpreter system scans through the instruction objects in the solution window and extracts the data stored in the instruction objects. The UI play button also calls the UI Controller to disable the instruction cache and solution window until the simulation is halted, since the player should not be allowed to interact with these UI elements while the solution is executing.

The control panel contains a halt button, which signals to the Interpreter to immediately stop execution and reset. The halt button also tells the UI Controller to enable the instruction cache and solution window so that the player can interact with these UI elements again.

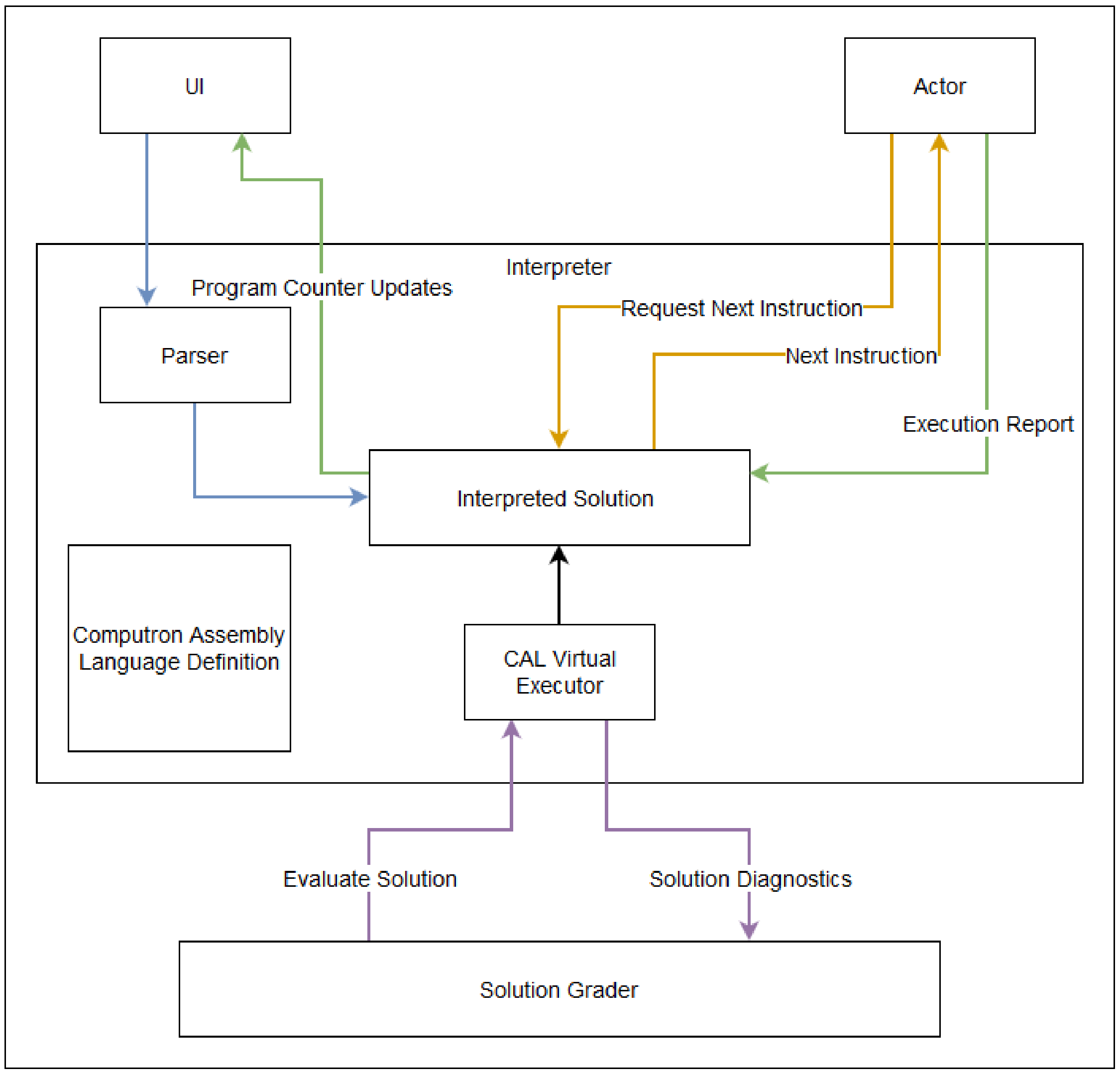
During execution the UI system also communicates with the Interpreter to capture and display certain runtime information to the player. As the player's solution is executed, the Interpreter tracks execution steps with a program counter and sends the program counter data to the UI system. The UI uses this information to update a program counter indicator object that shows the player which instruction is currently being simulated. Other common runtime communications that the UI gets from the Interpreter include errors encountered when running a simulation, or the results of a completed simulation.

6.7 Interpreter System

6.7.1 Overview

As the invisible interface between the player's instructions window and the Actor's execution logic, the Interpreter's responsibilities are considerably abstracted from normal gameplay programming. More specifically, the Interpreter is responsible for defining the game's assembly-like language, extracting player instructions from the user interface, marshaling the release of commands to the Actor for execution, and performing runtime analysis of player submissions. See Figure 39 for a look at the Interpreter's system diagram.

Figure 39: Interpreter System Overview



6.7.2 Language

In order to keep the game accessible to inexperienced users, we developed a simple pseudo-language to manage the manipulation of the puzzles. Below are the 13 instructions that make up the CAL.

- INPUT
  The INPUT command instructs Computron to approach the puzzle's Input Box and retrieve the next value. If Computron is currently holding a value, he will discard it. If the Input Box is empty, Computron will retrieve and hold a NULL value.
- OUTPUT
  The OUTPUT command instructs Computron to approach the puzzle's Output Box and deposit his current value. If Computron is not holding anything, this instruction will be considered invalid and raise a runtime error. Otherwise Computron will deposit the value in the output box.
- JUMP
  The plain JUMP command instructs the interpreter to unconditionally move the program counter to a new line in the player's solution. This instruction has no effect on Computron.
- JUMP IF NULL
  The JUMP IF NULL is a conditional jump instruction, which is only activated if the Actor is not holding any value. This instruction is often used to break out of loops.
- JUMP IF LESS [X]
  The JUMP IF LESS [X] is a conditional jump instruction, which is only activated if the value the Actor is holding is less than the value stored in player-specified memory card X. If memory card X is empty, or if Comptron is currently holding a null value, this instruction will be considered invalid and raise a runtime error.
- JUMP IF GREATER [X]
  The JUMP IF GREATER [X] is a conditional jump instruction, which is only activated if the value the Actor is holding is greater than the value stored in player-specified memory card X. If memory card X is empty, or if Comptron is currently holding a null value, this instruction will be considered invalid and raise a runtime error.

- JUMP IF EQUAL [X]
  The JUMP IF EQUAL [X] is a conditional jump instruction, which is only activated if the value the Actor is holding is equal to the value stored in player-specified memory card X. If memory card X is empty, or if Comptron is currently holding a null value, this instruction will be considered invalid and raise a runtime error.
- MOVE TO [X]
  The MOVE TO [X] command instructs Computron to move the value currently being held into the player-specified memory card X. If Computron's hands are currently empty, this instruction will be considered invalid and raise a runtime error.
- MOVE FROM [X]
  The MOVE FROM [X] command instructs Computron to remove the value currently being stored in the player-specified memory card X. If the memory card is empty, Computron will retrieve and hold a NULL value. If Computron is currently holding a value, it will be overwritten.
- COPY TO [X]
  The COPY TO [X] command instructs Computron to make a copy of the value currently being held and place it into the player-specified memory card X. Computron will retain a copy of the number. If Computron's hands are currently empty, this instruction will be considered invalid and raise a runtime error.
- COPY FROM [X]
  The COPY FROM [X] command instructs Computron to retrieve a copy of the value currently being stored in in the player-specified memory card X. The memory card will return the value that it holds. If the memory card is empty, Computron will retrieve and hold a NULL value. If Computron is currently holding a value, it will be overwritten.
- ADD [X]
  The ADD [X] command instructs Computron to add the value stored in player-specified memory card X to the value currently held. If either Computron's hands or memory card X are empty, this instruction will be considered invalid and raise a runtime error. Otherwise, Computron will perform the addition and overwrite his current value with the result.
- SUBTRACT [X]
  The SUBTRACT [X] command instructs Computron to subtract the value stored in player-specified memory card X from the value currently held. If

either Computron's hands or memory card X are empty, this instruction will be considered invalid and raise a runtime error. Otherwise, Computron will perform the subtraction and overwrite his current value with the result.

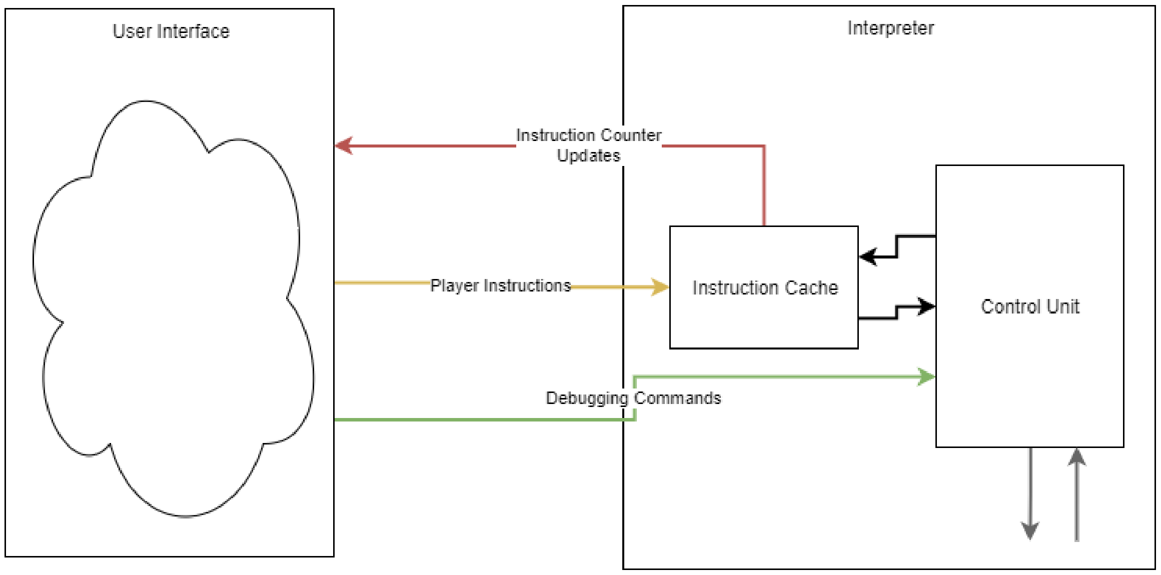
6.7.3 Program Counter

To facilitate proper simulation of the player's solution, the Interpreter will need to maintain a program counter that iterates and jumps through the player's code appropriately. To ensure that these updates happen correctly, the Interpreter will need to rely on reports from the Actor that state the validity of instructions and the result of conditional expressions. Updates to the program counter are forwarded to the Puzzle's UI system to maintain a visual representation of the program counter while the solution executes.

6.7.4 UI Interface

In addition to updating the program counter, the Interpreter is closely tied to the Puzzle UI's primary control panel. The Start, Step, and Halt buttons all tie directly to the Interpreter's control interface. Figure 40 illustrates the nature of these interactions.

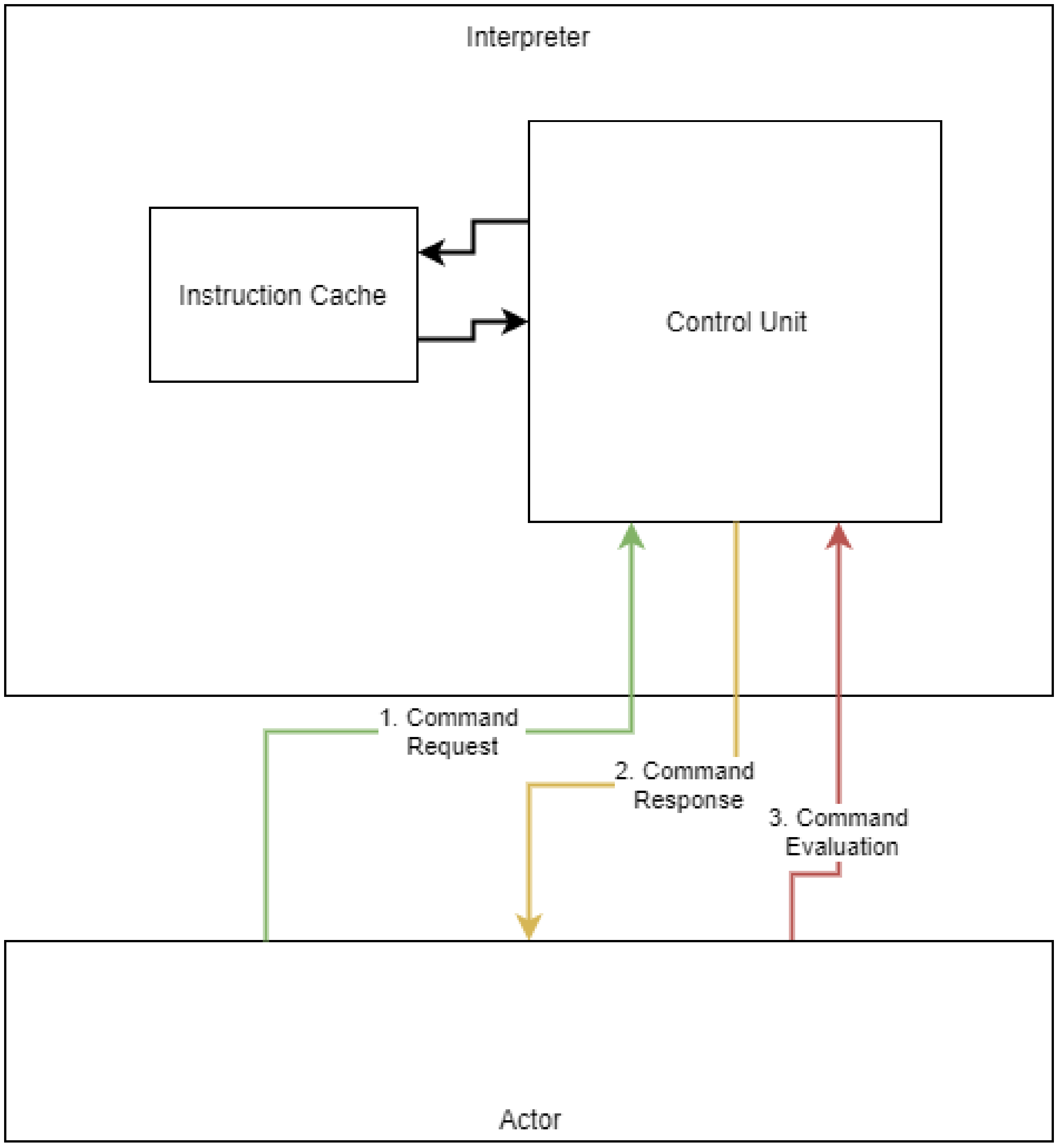
Figure 40: Interpreter/UI Interface Overview



6.7.5 Actor Interface

The Interpreter's Actor interface is responsible for making instructions available to the Actor on demand. In order to ensure the program counter is updated properly, the Interpreter and Actor engage in a three part handshake, shown in 41.

Figure 41: Interpreter/UI Interface Overview



The stages of the handshake are:

1. Command Request
   The first step of the Actor control exchange is the Command Request. The Actor will issue a Command Request to the Interpreter when the next player instruction is needed.
2. Command Response
   The second step of the exchange is the Command Response. This response will be a data structure for the Actor to parse, including the next instruction's opcode and argument.
3. Command Evaluation
   The last step of the exchange is the Command Evaluation. This stage is the Actor's chance to report runtime errors to the Interpreter if the instruction is deemed invalid and report the result of a conditional expression if the instruction included one.

6.7.6 CAL Virtual Machine

The Actor's implementation of the CAL is tightly coupled to the physical state of the puzzle. In addition, it performs many non-executing tasks to visualize the language. These conditions made the simulation unsuitable for conducting performance and correctness evaluations. To support those functions of solution grading the Interpreter manages a Computron Assembly Language Virtual Machine (CALVM) object. The CALVM allows the interpreter to simulate player solutions rapidly, and inspect runtime data like error states and cycle count easily. This also allows the game to reserve puzzle simulation explicitly for displaying player solutions.

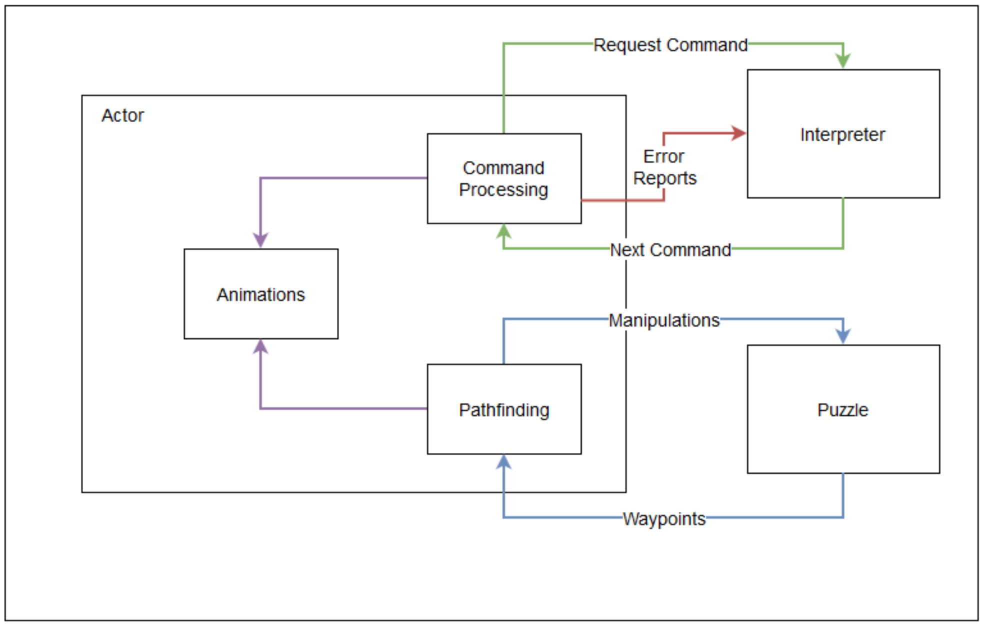
6.8 Actor System

6.8.1 Overview

The Actor is responsible for simulating the player's solutions to the puzzles. The Actor will be the primary source of non-verbal information, communicating to the player through movement and animations. The Actor's name is Computron (for whom the game is named), and it will be seen moving around the Puzzle Scene. Computron will be tasked with manipulating the data elements of the current puzzle based on the sequence of commands that the player constructs in the Solution Pane of the User Interface.

Figure 42 provides a high-level outline of the Actor System, all of its internal processes, and how it interacts with the other core systems of the game.

Figure 42: Actor Component Overview



The responsibilities of the Actor are as follows:

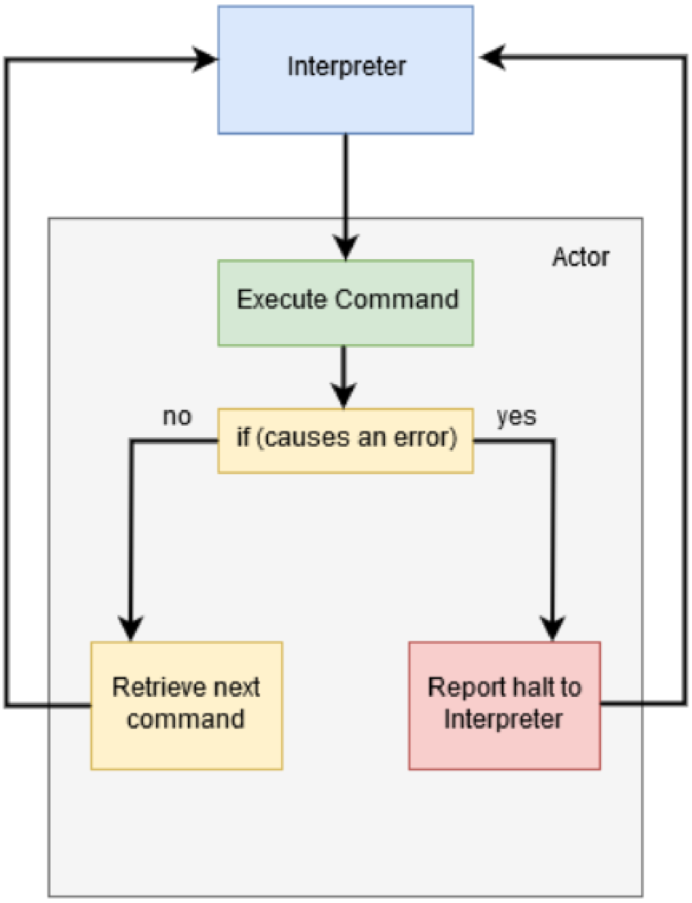
- Retrieve commands from the Interpreter System
- Alert the Interpreter to halt execution when an invalid command is attempted
- Move to the correct waypoints based on the command received
- Properly handle data manipulation in the puzzle scene (move and update data elements as specified by the current command)
- Communicate the flow of execution to the player through visuals
- Illustrate to the player how data changes in response to commands
- Tell the player about runtime errors in their code

6.8.2 Interpreter Interactions

Effective integration with the Interpreter is a key component of the functionality of the Actor. The Actor relies completely on the command sequence from the Interpreter in order to illustrate the execution of the solution. If the player is using the step button to execute their solution one command at a time, the Actor waits for a signal from the Interpreter to grab the next instruction. Otherwise, the play button has been used and the Actor and Interpreter engage in that sequence of instruction retrieval and signaling automatically. The Actor will continue to retrieve the next command in sequence from the Interpreter until the sequence terminates or an invalid command is attempted. If the command can be executed, the Actor will carry out the instruction visually so that the player can understand the flow of execution and the logic behind their solution. If the command causes an error in execution, the Actor will alert the Interpreter to halt execution. The flow of this process between the Actor and Interpreter can be seen in Figure 43.

As mentioned in Section 6.7.5, there is a three step handshake between the Actor and the Interpreter when processing instructions. To summarize, the Actor requests the next command, the Interpreter responds with a data structure of that command, and the Actor then reports the results of the execution of that command to the Interpreter. Again, this communication flow can be seen in Figure 41 located in that section.

Figure 43: Interactions between the Interpreter and Actor

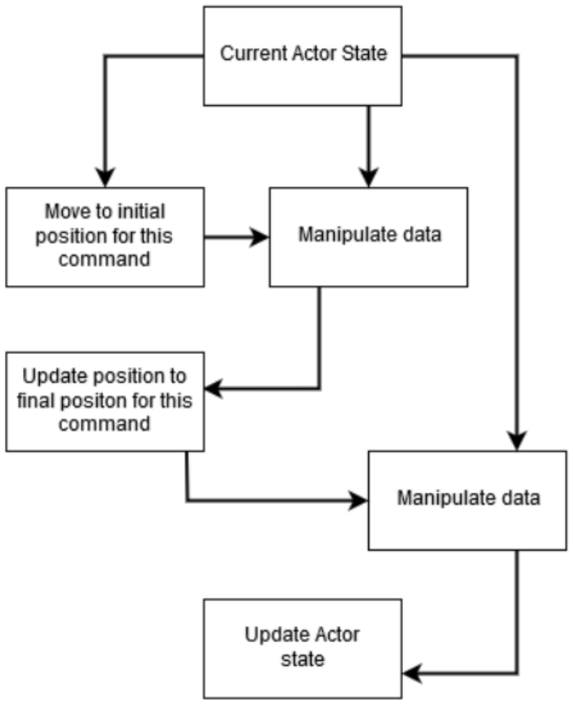


6.8.3 Processing Commands

The processing of commands is the core function of the Actor. Interfacing with the Interpreter is only the first part of command processing; the majority of that processing occurs independently within the Actor once a command has been retrieved. Figure 44 shows in detail how the Actor internally processes commands.

After fetching a command, the Actor will then execute a series of movements and actions to carry out that command. This includes moving to the proper waypoint to initiate the instruction, manipulating data from the Puzzle Scene object the Actor is currently at, moving to a second waypoint to continue execution, and manipulating data at that second Puzzle Scene object. Depending on what the current command is and the status of the Actor when the new command is retrieved, some of these four steps may not be applicable. Parsing each instruction separately and successfully

Figure 44: The state diagram followed by the actor when processing commands


Current Actor State
Move to initial position for this command
Manipulate data
Update position to final positon for this command
Manipulate data
Update Actor state

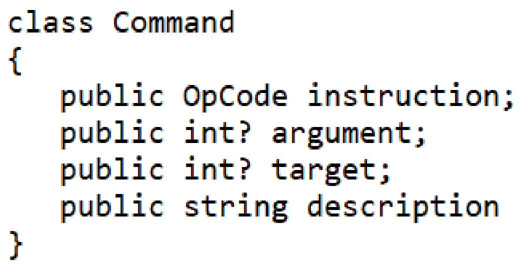
regardless of the current state is vital. Commands that cause errors must be reported accurately back to the Interpreter so that execution can be halted when they are encountered, and no false reporting can be tolerated. If the command won't cause an error until it is partially completed, the partial execution of that command must be shown. Additionally, commands that can complete without causing errors should always be carried out exactly as specified. This is integral to the process of the game because even if the command is the wrong selection at the time, it still needs to be executed properly in the puzzle to show the player the error in their logic. Of course, correct commands that work towards the solution also need to be executed properly to eventually reach the solved state of the puzzle.

The command passed from the Interpreter to the Actor needs to have several fields:

- Command Opcode
- Command Argument
- Command Destination
- Command Description

The associated opcode for the command is an integer value unique to that command, represented by an enumerator value. Each command in the language has a corresponding opcode. By reading these values, the Actor will know which instruction type it needs to process. The argument for that command will provide additional information to the Actor, such as which Memory Card the Actor will be interacting with when it is processing a command at the Registers Waypoint. This field is nullable for commands that don't require this information. The command destination variable is a number associated with jump type commands that informs the Interpreter of where to update the Program Counter to when a jump is followed. This field is also nullable so that it doesn't effect command types that don't require it. Finally, a description of each command is included as part of the package, and is used in instruction awarding to inform the player of what each command's function is. Because this information needs to be easily passed between the Actor and Interpreter systems, it has been packaged as a class (as seen in Figure 45) for easier communications.

Figure 45: The form of the class used for packaging the fields of the Command



```
class Command
{
   public OpCode instruction;
   public int? argument;
   public int? target;
   public string description
}
```

6.8.4 Processing Commands

The Actor system runs on a nested state machine. The inner level state machine is activated when a new command is received, and informs the Actor of how to behave in response to the command that is being executed. This state machine has a state for each of the 13 instructions in the CAL. Because each command has different intricacies involved in its successful execution, it was deemed necessary to allow explicit reactions to each situation. With that in mind, there are several general states that the Actor will find itself in while it is processing commands. For this reason, the command processing state machine was wrapped inside of a more general state machine that is used to dictate shared actions. The outer level state machine controls the five main action states of the Actor:

- Idle
- Processing
- Moving
- Reporting
- Halted

Idle
The Idle state is the default position of Computron. In this state, the character sits at the resting waypoint and simply awaits for a change in state to signal it to take a different action.

Processing
The Processing state is used for command retrieval and parsing. This is triggered by the Interpreter when a player hits either the step or play button. The Actor then retrieves the next command from the Interpreter. If the command requires movement of the character, the Moving state is invoked. Finally, the current command state is invoked from the inner level state machine for the Actor to simulate the specific execution of that command.

Moving
The Moving state is only invoked if the command being processed requires movement of the character. The Actor then checks its current position against the target position for that command and if there is a large enough difference between the two positions, the position of the character is updated. The appearance of smooth movement is achieved through only checking to update the position when absolutely necessary and using the SmoothDamp feature in the Vector3 class to make motion end less abruptly.

Reporting
The Reporting state is invoked at the end of each command being processed. This is third part of the handshake with the Interpreter system, and involves the Actor relaying important information to the Interpreter. The first aspect is detecting an error state and accurately reporting that to the Interpreter, as well as invoking its own Halted state. If no errors were encountered, the Actor then verifies whether or not

it was in stepping mode. If so, it updates its applicable control variables and awaits the next signal from the Interpreter to continue processing. If the next instruction in line is the submit instruction, though, the Actor automatically moves forward and completes the puzzle so that the player isn't required to trigger this action. If the Actor is not in the stepping mode, it passes a validation report to the Interpreter which signals the Interpreter to update the program counter appropriately and then signal the Actor to retrieve the next instruction.

Halted
The Halted state is called only when an error is encountered during process. To be clear, this is the equivalent of a runtime error in the player's proposed solution, like trying to store or output a null value. Depending on what the current instruction is and the state of applicable variables, the Actor is able to report the type of error to the player via on-screen messaging.

6.8.5 Player Messaging

The Actor is also a key component of communication with the player. Through the visual execution of the instructions that the player has selected, the Actor conveys how the player's solution to the puzzle functions. This includes not just moving around the board and manipulating the data elements, but also providing visual cues to the user via detailed animations. For example, the data elements will be visibly picked up and held by the Actor. If that data is copied to another location, the Actor will be seen producing a copy of that data and placing it at the location specified. Data that is incremented or decremented via commands will be changed according to those commands so that the player sees and understands how the data elements are being manipulated.

Aside from effective visual communication, the Actor will also provide information to the player via text when applicable. For the most part, this will be explicitly in response to runtime errors that are encountered during execution. Because a certain runtime error is always the same no matter what instance of the attempted puzzle solution causes it to occur, the Actor can provide generic messaging prompts to the player about these errors independently of the puzzle itself. As previously seen in Figure 44, the Actor processes the success of commands completely independently and reports those errors back to the Interpreter so that the Interpreter can halt execution. It is because of this independent evaluation that the Actor does not need to rely on input from any other systems in order to provide successful and relevant messaging on these errors.

In addition to error messaging, the Actor can also give puzzle specific hints to the player when requested. The hints are stored as part of each puzzle's data and are cycled through any time the player clicks on the character while it is stationary. These hints are designed to encourage the player towards a solution without giving away too much. The point is to continue to challenge the player to reach the solution on their own, but at the same time provide some guidance on where they need to concentrate their thinking.

7 Budget and Financing

As a student-sponsored team, Group 32 has no financial support. Fortunately all of the tools we have decided to use to make *Computron* are either free or available to students free of charge. Unfortunately, the nature of the licenses for some of our tools is highly restrictive, so careful monitoring of resource usage was imperative to keep our costs at zero.

Below is a table of the software tools we are using, the tool's cost to the team, and any licensing risks we had to monitor for that tool.

Software	**License Cost**	**License Restrictions**
Unity	\$0.00	No dark mode, limited VCS integration
Perforce VCS (Helix Core)	\$0.00	
Helix Swarm	\$0.00	
AWS Server (Perforce hosting)	\$0.00	30GB of hard drive space, 750 hours of compute time
Git	\$0.00	

8 Milestones

This section of the document describes the major milestones of our project that we set as regular interval goals for our success.

8.1 Big Prototype Party (Monday, Oct 14th)

Summary

Since this project does not have a sponsor, the onus is on us to determine the exact nature of the problem we would like to solve. To ensure that our efforts are directed effectively for the remainder of our project, our first month has been dedicated to researching and playtesting. By this date, every member of the team should have a solid foundation of the problem space we are working in, and have a prototype of a solution.

Deliverables

Every member of the team must have a prototype of our final game. This prototype should fit the following criteria:

- The prototype is interactive.
- The prototype is engaging to work with.
- The prototype represents a wide slice of the game's mechanics.
- The prototype makes the player think computationally.
- The rules of the prototype are concise and written out.

8.2 Game Pitch (Friday, October 18th)

Summary

This milestone culminates one of the most critical stages of our project. Now that we've had the time to demonstrate non-abstractly what our problem space is and how we'd like to solve it, we can converge on a unified solution. By this milestone the team should be able to confidently answer the following questions, and develop the documentation necessary to capture our answers:

1. What will our final product look like?
2. What are the learning objectives of our game?
3. What kind of puzzles will our game have?
4. How will those puzzles work?
5. How will those puzzles achieve our learning objectives?
6. How will we order these puzzles to ensure that there is an appropriate learning curve?
7. What evidence do we have that our learning curve will be engaging and effective for our target audience?
8. What games did we draw inspiration from?

Deliverables

This milestone's deliverables are still non-code items. These are items that we will rely on for the remainder of our project, and inform all future efforts. Specifically, we must deliver:

- A unified prototype of our final game.
- A write up of our prototype, its capabilities, and its design philosophy.
- Diagrams/concept drawings of game mechanics not demonstrated by the prototype.
- Research Papers that support our design choices.
- Playtesting findings that support our design choices.
- Any other artifacts necessary to answer the questions above.

8.3 Paper Prototype 2.0 (Monday, October 28th)

Summary

Rapid iteration is an important part of successfully designing a game. By this milestone, our original prototype should be placed in the hands of as many playtesters as possible. The group should use the data from this playtesting to inform any design changes we make to our prototype.

Deliverables

- A new prototype of our final game, with documentation supporting our changes to our original prototype.
- Expanded design documentation to capture the vision of our final game including:
  - Detailed system diagrams that describe how the mechanics of our game interact.
  - An explanation of who will be the lead on certain areas of our game.

8.4 Hello Game (Friday, Dec 13th)

Summary

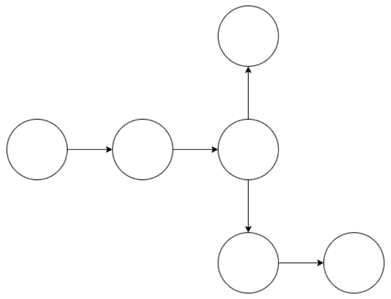
This milestone marks our the delivery of our proof of concept prototype. The teams' advisor will evaluate the prototype and provide recommendations for moving forward. The prototype will show an extremely basic vertical slice of our final game.

Deliverables

The only deliverable for this milestone will be the Hello World prototype. This prototype should meet the following requirements:

- Usability
  The prototype shall show the entire scene flow of our game. A user should be able to move between the Main Menu, Puzzle Selection, and Puzzle Scenes at will. The game should respond to these requests gracefully, and never leave the user trapped in a scene.
- Game Mechanics
  The foundations of the game's major mechanics should be in place and interactable. The expected functionalities of the demo are specified below by scene.
  - Main Menu
    The Main Menu scene should display the game's working title, *Computron*, as well as a functional play button that will bring the user to the Puzzle Selection Scene. The scene should also have a placeholder background image.
  - Puzzle Selection Scene
    The Puzzle Selection Scene should display a placeholder level graph, with selectable nodes. Each node should transfer the player to the Puzzle Scene. The player should also be able to return to the main menu.

Figure 46: Sample level graph



  - Puzzle Scene
    The Puzzle Scene will be the most involved portion of our prototype. Desired mechanics are listed below by discipline:

    UI: The Puzzle UI will need to allow the player to perform the basic puzzle solving operations. It should present the player with an instruction set (*Input, Output*) from which they may drag instructions into a solution window. The Puzzle UI should present controls to allow the player to begin and terminate the simulation of their solution. When the player's solution is being executed, the UI should prohibit alterations to the solution window and display a program counter to mark the current instruction being executed.

Level Design: The level design of the Puzzle Scene will present all of the static elements that will be present in every puzzle of our game. It will display an input box with randomly generated contents, an output box that can accept objects from the Actor, and a simple background plate.

Puzzle Logic: Working in tandem with the level design, the puzzle scene's logic system should be operational. Puzzle Logic is responsible for loading the scene with the appropriate puzzle information. This includes a randomly generated input box, the level prompt, an expected output, and the logic to read the output box upon program completion to grade the player's solution.

Interpreter: The Interpreter is the glue that binds the UI controls to the Actor system. For this milestone, the Interpreter should be able to access and process the player's solution stored in the solution window. When the player starts the simulation, the Interpreter should process the instructions (ensuring none are malformed) and make them available to the Actor upon request. When the Actor requests an instruction, the interpreter will update its program counter accordingly and report the counter's new position to the UI. The Interpreter's PC calculation should be fully operational for the Input, Output, and Unconditional Jump commands.

Actor: For this milestone, the Actor should be able to query the Interpreter for instructions to execute and interact with static level elements. Upon receiving an input command, the Actor should move to the input box and remove at item. The currently held item should be displayed on the Actor. Upon receiving an output command, the Actor should move to the output box and deposit the currently held item. If the Actor does not have an item in hand, it should report a fault to the Interpreter so that simulation can be halted. If the Actor sees two input commands back to back, it should discard its currently held item and attempt to take from the input box again.

- Art
  For this milestone, programmer art placeholders are acceptable. However, the team should have a plan in place to acquire quality art assets.

8.5 Waterfall Method (Thursday, December 5th)

Summary

This milestone marks the deadline for Senior Design I documentation submission. This document must be complete, professionally bound, and delivered to HEC-345 by 1:00PM.

Deliverables

As per the requirements of this course, our final design document will describe all aspects of the project including:

1. An Executive Summary
   The Executive Summary will give a high level overview about the purpose of the project and how the project will fulfill that purpose. The Executive Summary should be written in language accessible to non-technical professionals.
2. A section detailing project significance
   This section should echo our Executive Summary in greater detail, with more specific language.
3. A section describing our requirements
   This section should include an exhaustive list of our project's functional and non-functional requirements.
4. A section detailing our game's design process
   This section should describe the team's efforts in discovering project requirements, and relate the game mechanics chosen to the projects goals.
5. A section detailing our game's technical design
   This section should describe the layout of the game's code systems. It should include detailed technical discussions and diagrams to guide development.

6. The project's milestones
   The milestones will be sets of requirements to ensure the project remains on track.

8.6 Open to Interpretation (Friday, January 10th)

Summary

"Open to Interpretation" will mark the project's first Alpha release. In this state, the game should be functional enough to place in front of a playtester with minor developer intervention. All basic game mechanics will be fully functional, and observing gameplay should produce relevant observations for the development team.

Deliverables

This milestone's deliverable is a stable Alpha release of our game. The requirements of the Alpha release are as follows:

- Player experience
  The player experience should be polished enough to allow an individual with no experience with our project to pick up and play it. All controls presented to the player should work as expected, free of any noticeable bugs or glitches. Menus should allow for proper flow between the game's scenes, and incomplete or non-functional menus should be marked as such.
  Specifically, the player should be able to perform the following operations:
  1. Launch the game from outside the Unity Editor.
  2. Start the game from the main menu, and enter the puzzle selection scene.
  3. Upon first run, only the first level of the game should be unlocked.
  4. Subsequent levels will only be unlocked after the player completes a puzzle.
     – Exiting a puzzle without solving it will not unlock new puzzles
  5. Puzzle unlocking will have an associated visual when returning to the level select screen.
     – Player controls will be disabled and player focus should be drawn to the unlocked levels. (e.g. Camera zoom, simple animations, etc.)

6. After selecting a puzzle, the player should be transferred to the puzzle scene. The game should have at least two distinct levels.
7. In the puzzle scene, the following mechanics should be operational.
   - The instruction set pane shows only the subset of instructions chosen to be available for that puzzle.
   - Instructions can be dropped into the solution pane and manipulated in an intuitive way
   - If the puzzle includes Memory Cards, the cards are displayed at the bottom of the screen and are interactable.
   - Move and Copy instructions are not usable until at least one Memory card is in play.
   - Memory Cards can be placed on, removed from, and re-ordered in the play area.
   - A Memory Card cannot be removed if it is referenced by an instruction in the solution window. Instructions blocking card removal are highlighted when the player's request fails.
   - Computron is capable of interacting with the Inbox, Outbox, and registers.

- Available Puzzles
  Puzzles 3, 4 of the tutorial sequence specified in Section **5.5**.
- Available Mechanics
  - Level Selection
    The level selection scene allows players to chose which puzzle they'd like to attempt. Choosing a level should transfer the player to the Puzzle Scene with the relevant puzzle data loaded. The player will be restricted in which puzzles are available, and have more unlocked as they successfully complete them.
  - Basic instruction writing
    Player can click and drag the INPUT, OUTPUT, MOVETO, MOVEFROM, JUMP, and JUMP IF NULL instructions into the instruction pane. Jump instructions allow player to click-and-drag to set jump anchor.
  - Basic Memory Card Manipulation
    Player is presented and can interact with a hand of memory cards on levels that require them. Cards can be played, reorganized, and removed. MOVETO and MOVEFROM instructions are active if and only if there is at least one Memory Card on the board.

- Basic Solution Grading
  After a player completes a puzzle, they are presented with a score sheet that provides metrics on the par for:
  - Instruction count
  - Solution Runtime
  - Memory Card cost

  After they are done viewing this screen, they will be transferred back to the puzzle selection scene where they will see new levels unlock.

8.7 Tutorializing (Wednesday, February 12th)

Summary

"Tutorializing" will mark the Computron's second Alpha release. For this release, the tutorial system will be greatly expanded to allow for more hands-off playtesting.

Deliverables

This milestone's deliverable is another stable Alpha release of our game. The expected new features are listed below:

- Available Puzzles
  Puzzles 1 through 9 of the tutorial sequence specified in Section **5.5**.
- Available Mechanics
  - Level Selection
    The level selection menu should meet the following requirements:
    - Level selection menu supports a strong ordering of tutorial levels.
    - Level selection menu allows for optional paths that do not impede player progression
    - The player's performance on puzzles (number of stars) should be recorded and restored between play sessions.
    - Advanced levels should be restricted from play until the player earns the requisite number of points to unlock them.

- Instruction Writing
  The puzzle writing and interpreting system should support the following instructions:
  * INPUT
  * OUTPUT
  * JUMP
  * JUMP IF NULL
  * JUMP IF LESS
  * JUMP IF GREATER
  * MOVETO X
  * MOVEFROM x
- Memory Card Manipulation
  Player memory card interactions should be well polished. Players should find playing and removing cards an intuitive part of the puzzle solving process. In addition, the following cards should be operational:
  * Register
  * Stack

8.8 Breaking Beta (Friday, February 28th)

Summary

"Breaking Beta" is Computron's first Beta release. At this point, the game's minimum viable product should be feature complete, leaving ample time to rework and refine mechanics based on intense playtesting.

Deliverables

- Available Puzzles
  Puzzles 1 through 12 of the tutorial sequence specified in Section **5.5**.
- Available Mechanics
  - Instruction Introductions
    Messaging should be added to attract the player's attention to new instructions when they are made available. Upon first receipt of a new instruction,

the User Interface should display an interactive menu that allows players to unbox their new command.

- Instruction Writing
  In game documentation of how instructions work is available from the puzzle scene.
- Memory Card Manipulation
  In game documentation of how a memory card works is available from the puzzle scene.

8.9 FIEA Playtest (Monday, March 2nd)

Summary

The team's advisor has offered to allow *Computron* to be playtested by students at the Florida Interactive Entertainment Academy (FIEA). Playtesters from FIEA will span both technical and non-technical roles in game development, and will be able to offer valuable insights. It will be imperative to capture these insights, and use them to guide the final stages of the game's development.

Deliverables

For this playtest, the team should be able to present a high quality build of the game. Fielding a high quality demo will be very important in allowing the team to gather effective observations from the playtesters. The team should arrive to the playtest with:

1. A stable build of the game.
2. A set of exit survey questions to track player experience.
3. Several computers capable of playing the game to allow increased playtesting bandwidth.

After the playtest, the team should compile player feedback and evaluate which parts of the game will need more attention.

8.10 Quick Math (Friday, March 20th)

Summary

“Quick Math” will be the second Beta release of Computron. It will feature an expanded instruction set and new puzzles that make use of mathematical principles.

Deliverables

- New Instructions
  Computron should support four brand new instructions:
  - ADD X
  - SUBTRACT X
  - BUMP++
  - BUMP--
- New Puzzles
  This release should feature three new puzzles that exercise the new problem spaces made accessible by the expanded instruction set:
  1. Adding Machine
     This puzzle should introduce players to the mechanics of the new ADD instruction. Puzzle should ask players to add pairs of numbers in the input and output their sum.
  2. Subractinator
     This puzzle should introduce players to the mechanics of the new SUBTRACT instruction. Puzzle should ask players to subtract pairs of numbers in the input and output their difference.
  3. Hexiplier
     This puzzle should show players an application of the new ADD instruction. For each number in the input, the puzzle should expect that number times 6 as output.

8.11 Release Candidate (Friday, April 10th)

Summary

Nearing the end of our project, milestone requirements will largely be determined by the state of the game and the results of playtests. For this milestone, the team should have most of the concerns raised from the FIEA playtest fully remedied.

Deliverables

A fun release candidate of *Computron*.

8.12 Mission Accomplished (Wednesday, April 15th)

Summary

"Mission Accomplished" marks the final release of *Computron*. The project should be in a state that accomplishes the goals set forth in this document.

Deliverables

A game the team is proud of.

9 Project Summary and Conclusion

9.1 Summary

The early stages of our project mainly consisted of research and testing. When we first began, we knew the main idea of our project – an educational video game that taught players computer science concepts – but we didn't have a clear definition on how that idea would be realized or the steps we needed to take to get there. We knew we didn't want to jump into designing the game and its mechanics without a clearly defined plan and endpoint in mind. As a team we combined efforts in researching and brainstorming to determine a general concept that our game would be based around. With a central idea defined, we each developed a paper prototype mock-up of what our version of the game would be and playtested it on people inexperienced with programming. The unified prototype was borne of our combined efforts and the information gathered from this initial research and development phase of *Computron*.

After the unified prototype was formed, we each produced an identical whiteboard version of the game and tested it on as many people as we could. The testing went exceptionally well and we all agreed that we were ready to move forward with coding the game systems and mechanics. With a clear picture now forming as we began laying down the skeleton code to instantiate the basic layout of our game, we continued to fine tune the definitions and expectations of each aspect of our game and what the end result product should look like. This consisted not just of the physical appearance and feel of the game, but also the functionality of each component and how it should interact with other core components of the game. Additionally, the progression of puzzles that was designed for use with the prototype tested so well that it was adopted directly as the tutorial sequence for *Computron*. Minor changes and additional levels were later added to the sequence, but the progression was found to be extremely effective.

Through frequent playtesting, we were able to refine a comprehensive design of our game. Working to implement key features was a driving focus, and informed the priority of minor features and supporting functionality. As a team, we strove to continuously keep our end goal in mind, and always maintained the priority of each sprint to reflect the end goal of the project as a whole. Through this focus and determination, we are able to now deliver an exhaustive and satisfying first pass at *Computron.*

9.2 Future Considerations

We realize that six months is an incredibly tight turnaround time for a fully realized video game project, and are well aware that there is potential for improvement and iteration on our final version of *Computron.* It is our dearest hope that future students will take interest in the groundwork we have laid here and aspire to expand upon the current offerings and capabilities offered herein. To this end, we would like to present a summary of possible improvements to guide others towards success in their efforts.

As it currently stands, *Computron* has only a basic implementation of tutorial sequences implemented. We would like to see future iterations of the project expand upon the number of provided tutorials. Keep in mind that a tutorial punctuating every level of the main path becomes more annoying than helpful, so it would also be beneficial to ensure there are enough buffer levels between newly introduced instructions or cards, and thus, tutorials. The frequency of introducing new game elements and interrupting the player to deliver instructions versus the progression of puzzles and challenge presented in solving each one is a delicate balance, and requires extensive testing and refinement. Proceed with caution.

In addition to expanding the tutorials available to players, it would behoove the next team to adopt this project to refine the tutorial construction system to be more user friendly. As it stands now, building a tutorial is a fairly involved process. In order to enable the streamlining of expansion on *Computron* in any further iterations, it is integral to construct a tutorial building system that is intuitive and effortless. By streamlining this process, we can collectively open up the possibility of introducing *Computron* as course software and allow professors to customize the puzzles provided to fit course materials.

In the same sense, it could be beneficial to the application of *Computron* to lend greater support to WebGL builds from Unity. In its current state, the game experiences some scaling errors based on monitor resolution and whether or not the user is in full-screen mode. As we are currently only set to support deployment to an independent desktop build for the main three operating systems, including better web deployment stability could be largely beneficial. In addition to web-based access, a system that implements a database to keep track of leaderboard scores could be used in conjuction with Introduction to Programming classes in order to not only provide students with an incentive to practice programming concepts, but also to allow professors to track the progress of students enrolled in their courses. This insight could help lecturers focus on topics students struggle with and allow them to spend less time on topics that students handle well.

Additionally, there are some minor tweaks that could be made to enhance the player experience. First, the visuals of the game have some room for improvement. Currently there are animations missing on the Computron character. The artists we had on board were able to get sprite sheets into the game for animations but they were not implemented because the art style of the character and the puzzle scene contrasted. They were in the process of collaborating to transfer the sprite sheets to the puzzle scene art style, but unfortunately did not make it into the build in time. Also, in game objects like data cubes are extremely basic and could be much more impactful with a bit of polish. Second, more levels and card awarding can go a long way to improve player investment in *Computron*. Currently, only newly acquired instructions are being distributed via the awarding system. Expanding this to include cards would be immensely beneficial to players in conjuction with the expanded tutorial system. More levels in the progression sequence could help space the time between tutorials – as previously mentioned – as well as increase the incentive for players to continue practicing their computational thinking skills. A complementary feature of this would be including tooltips on each instruction type and card type to remind players of the description of an element whenever it is appropriate to do so (e.g., on right click, double left click, etc.). Furthermore, future developers might aspire to construct a "card book" in the image of a card collector, where each card has an assigned slot in the book, and players can look through it at their leisure and review important information about each game element.

Some possible "nice to have" features for future iterations of *Computron* might include deeply clever star requirements, allowing multiple solutions to be saved for each puzzle, and the opportunity to submit more efficient solutions to the developers. For current star requirements, players are simply measured against the metrics for the best solutions that the developers could come up. It would be nice for future teams to revisit those solutions and see if they can find more clever ways to get better ratings for the awards metrics. Similarly, if a player happens to get a better score than the metrics set so far, it would be beneficial to submit that solution to the development team so that the game could be updated to reflect a better solution. This would, of course, involve more iteration or explanation. Again, the idea of an online leaderboard comes to mind for claiming and reporting solution efficiency. Additionally, some metrics of efficiency, instruction count, and cycle count may not all be attainable on a single pass at a puzzle solution. It would be advantageous to allow users to solve multiple solutions for each puzzle in the game. This way, an optimally time efficient solution and an optimally space efficient solution could both be stored for future reference.

9.3 Conclusion

The extensive amount of research and playtesting that went into this project before development began was very informative of our design. Being able to change major components of the game without having to redesign or replace code was very beneficial, and we were able to develop a very clear idea of what we expected out of our project before we ever began implementing it. Running through our actual game on paper and being able to refine details that players found confusing or unintuitive helped shape the final vision of *Computron* well ahead of the first line of code ever being written.

Minimizing the interactions between the four core systems of the game also helped us to reduce the spaces where problems might arise. Each major system of the game only communicates with two other systems, limiting the potential for errors to occur. While a considerable amount of work had to be done when building parts of the game that closely relied on interfacing with another system, we were largely able to work independently on our respective systems. This allowed a lot of progress to happen simultaneously and no one was ever left waiting for someone to finish something before they could work on their own system.

The early playtesting results have shown a lot of promise and positive feedback. Many players reported that the concept of the game is interesting and that the game is engaging to play. Most early constructive feedback was in response to systems that had not been fully completed, with some quality of life suggestions that have since been incorporated into our final design. With all major systems of the game in their current state, the majority of feedback revolved around polish requests for game visuals, with some suggestions for expansion on mechanics. These have been summarized in the Future Considerations section of this document. With that in mind, it is our opinion that this Computron 1.0 verision is an immense success as we were able to meet and exceed all of project goals.

10 References

[1] J. Cheon and K. Kwon. “Exploring problem decomposition and program development through block-based programs”, In *International Journal of Computer Science Education in Schools*, April 2019

[2] D. Topalli and N. E. Cagiltay. “Improving programming skills in engineering education through problem-based game projects with Scratch”, In *Computers and Education*, 2018

[3] H. Liang et al. “Students’ perception on the use of visual tilings to support their learning of programming concepts”, In *International Conference on Teaching, Assessment and Learning for Engineering*, 2013

[4] L. Leifheit et al. “Programming Unplugged: An Evaluation of Game-Based Methods for Teaching Computational Thinking in Primary School”, In *European Conference on Games Based Learning*, 2018

11 Appendices

11.1 Appendix A: List of Figures

List of Figures

11.2 Appendix B: Contact Information

We wish to be available to any prospective teams who take on the task of iterating on this project. For that reason, we have chosen to disclose our email addresses so that those who follow can contact us with questions about design choices, more specific guidance on improvements, and general project inquiries.

Name:	**Email Address:**
Nicolas LaCognata	nicolacognata@gmail.com
Brandy King	brandy.king.85.15@gmail.com
John Billingham	jebillingham3@knights.ucf.edu
Sean Simonian	seansimonian@knights.ucf.edu

Additionally, we created an email for the project in order to manage contact with potential artists and other project-related correspondence. If future developers would like to take over this account, we are happy to share the login credentials.