Computron

John Billingham, Brandy King, Nicolas LaCognata, and Sean Simonian

Department of Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — Computron is a computer-based puzzle-solving video game that aims to teach computational thinking as well as some basic computer science concepts to users. The aim of the game is to be accessible and engaging for inexperienced players who are interested in learning programming, while still being instructional. In doing so, it aspires to make the transition into introductory level courses more manageable.**

***Index Terms* — Computer science education, courseware, educational technology, electronic learning.**

I. Introduction

Every semester, hundreds of students enroll in their first introductory computer science course at UCF. These students are about to face two very difficult tasks that they must do simultaneously: master the syntax of their first programming language and learn to think computationally in order to solve novel problems.

Computron is a puzzle-solving video game that aims to better prepare users for programmatic thinking. By gamifying the educational process, *Computron* remains accessible to inexperienced programmers while still conveying educational subject matter. The stepwise increase in difficulty through the in-game puzzle progression is designed to help players acclimate to increasingly complex algorithmic logic. By providing feedback on solution efficiency and brevity, *Computron* also encourages users to improve upon their solutions.

II. Overview

Currently, there is no shortage of offerings on the market for individuals who want to use a video game interface to break into programming. The issue, however, lies in the deficit in the market; most games available now are either too abstract to easily apply their lessons to programming (*Baba is You*) or too technical for beginners to learn (*Codingame*). There are a handful of games that bridge that gap by being clear about their connection to programming without requiring knowledge of syntax. However, even the games that stray away from the technical aspects of computational thinking have too steep of a learning curve for players new to coding to find useful. A complete lack of practical instructions or tutorials of any kind make games like *Human Resource Machine* a good fit for individuals looking to hone their programming skills, but a bad match for those looking to break into coding.

This is where *Computron* comes in. By giving the player control of a very simple assembly-like command language to build solutions, the difficulty of trying to master syntax is abstracted without losing the meaning of how instructions work together to build an algorithmic solution. The assembled solution is then executed on screen by a character, acting out each step of their program in sequence and showing how the data in the puzzle changes based on the solution the player has built. This visual representation of what their solution does computationally will help users clearly trace through the flow of logic and puzzle manipulation that they have set up.

While this template is quite similar to *Human Resource Machine*, there are significant improvements necessary in order to make *Computron* a viable offering for beginners. The first key component is a hint system that can actually help players who need a nudge in the right direction. Current offerings that even have hint systems are at best comedic but unhelpful, and at worst completely unrelated to the puzzle or a solution. The hint system in *Computron* is designed to push players towards a solution while being cautious not to directly give the answer away. The second key component, and arguably the most important, is the implementation of a tutorial system that introduces players to new commands and structures as they encounter them. After playtesting *Human Resource Machine* during our research phase, we found this to be the biggest downfall as players who weren't already technically minded became stuck and were forced into frustration since nothing had ever been explained to them in regards to how different elements of the game were meant to operate.

Another crucial difference between *Computron* and games like *Human Resource Machine* is the introduction of data structures. We introduce players to the concepts of registers, heaps, stacks, and queues so that they can explore solving problems with these staples of computer science without requiring that they keep up with maintaining the structures in code. We achieve this by giving the player "memory cards" of different data structure types, which act as a sort of black box for them to use in their solutions to certain puzzles. This way they can focus on learning the general concepts that govern these structures without having to get hung up on the intricacies of their maintenance in code.

By including these tools designed to help users understand the concepts but still encourage them to explore

the different commands and come to a solution on their own, *Computron* hopes to instill a sense of curiosity and desire for exploration in its users when it comes to computational thinking.

III. Research and Design

Preliminary stages of development involved research with existing programming games in order to understand what usable approaches were and where innovation needed to occur. By observing novice programmers attempt games designed around programming, we were able to make more informed design choices about our own project. Focusing on aspects that caused users to become frustrated or give up became a driving motivating factor in defining what *Computron* should be capable of.

The next logical step for us was to develop paper prototypes of our game and begin trying it out on potential players. We took feedback into account and modified the prototypes over several iterations. This way, we could refine the feature set of *Computron* before we ever began coding the project. This was instrumental to our success as not only did we eliminate the need to rework or replace existing code when changes were made to the overall design, but it also put us in the position of starting the coding of the project with a well-defined end goal. That means we were able to take details into consideration during early stages of coding and account for them from the beginning of our implementation instead of having to force them in later once frameworks had already been well established.

Once we had a finalized prototype that would encompass our desired features, we worked together to pin down all of the details – the memory card system, the instruction set, and the run of puzzles that we wanted to include. From here, we took this final prototype version back out to potential players and tested yet again to ensure that we were addressing the issues we had previously encountered, that we were making necessary improvements where other offerings fell short, and that our finished product was effective, playable, and engaging. The refinement of these core concepts of our game meant that we were ready to begin implementing our design.

IV. Implementation

Computron was developed in Unity with C#, and version control was handled via Perforce with releases deployed to the GitHub repository for the project at each major milestone.

There are only three scenes within the project – the main menu, level select, and puzzle scene. The game works by using one scene to load the contents of another, depending on user selection. The successful completion of a puzzle level awards the player with a score based on several metrics, which then allows them to progress to other levels in the game.

A. Main Menu Scene

The menu scene is a straightforward implementation that allows users to load and delete existing save files, or to control music and sound volumes in the options menu. The save system in *Computron* gives users three slots for save files, each of them maintaining separate game file data. Behind the scenes, these save slots instantiate a persistent game object that is not destroyed between scenes, storing important game progress data specific to that save slot. This data is saved automatically when a user exits the game so that the next time the game is loaded, they can continue where they left off on that file.

B. Level Select Scene

The level selection scene is a graph-based map of level nodes (Fig. 1). The progression through this map is restricted with a minimum score requirement to unlock each additional puzzle node. These minimums have been established to mandate that a player cannot do just the bare minimum to get through the game and may need to revisit some solutions or optional levels in order to get a better overall score and continue moving forward. The difficulty progression of the puzzles themselves was designed during our prototyping phase and focuses on introducing a basic concept, building on it, and incorporating it with other concepts.

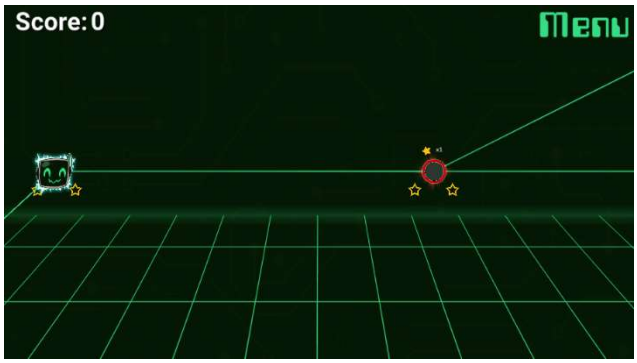

Fig. 1. A screenshot of the level selection scene, showing a locked puzzle node.

A locked node is red and has a minimum unlock score greater than the players' current score. The node also shows

outlines for how many stars (either two or three depending on the puzzle type) are available to be earned. These nodes are blocked and cannot be traveled to until the minimum score requirement is met. When a node becomes unlocked, it turns blue and is able to be reached in navigation. When a level is completed, it turns gold and the acquired stars are filled in.

Control of the scene is primarily through mouse controls, allowing zooming, panning, click-based A* navigation, and click-based level selection. Hovering over a puzzle gives some basic information, such as puzzle name, description, and stars available or earned.

C. Puzzle Scene

The puzzle scene is a single instance in the project with many standard elements and objects, like the static user interface elements, the background, the character, and so on. Because so much of the puzzle is the same between different levels, we made the design choice to have a single version of the scene. It contains all of the static elements and acts as a "base version" of the puzzle. Then we populate what changes between puzzles dynamically whenever a new level is selected by the player. The puzzle-specific data of each level – the input stream, puzzle description, award metrics, solution output, etc. – are loaded dynamically in the scene when the player enters a level (Fig. 2).

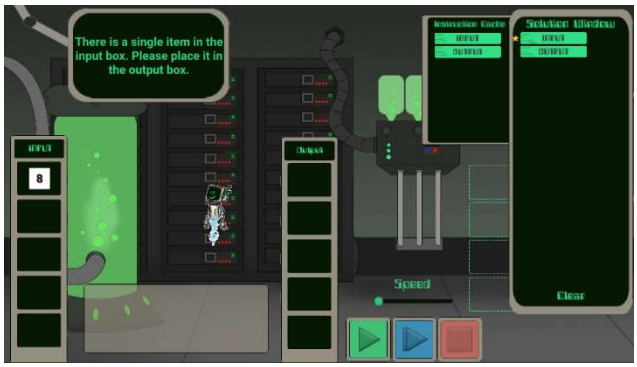

Fig. 2. A screenshot of the first puzzle in the game, where a user has already put a solution in and is ready to execute it.

In each puzzle, the player is given a problem description that tells them how to handle the input data to solve that puzzle. An instruction cache of all the commands that the player has available to solve the problem at hand holds draggable representations of each command in the language of the game. The player then drags any number of these commands into the solution window, arranging them in the order they wish for the commands to be executed. A small control panel allows them to run a full solution, step through a solution one command at a time, or halt a currently running solution.

The character of the game, Computron, carries out the commands in sequential order. This visual representation of the solution being executed shows players explicitly the order of events that they have designed, and how the sequence of their commands alters the data in the puzzle. If an error is encountered, like trying to store or output a null value, the simulation halts and the player is informed about the cause of the error. The feedback on error states informs users of runtime errors to encourage the development of good habits in designing an algorithm. Logic errors are allowed, as we want the players to have a lot of freedom with designing and exploring solutions, but as soon as a player tries to output a value that is not expected, an error state is reached. This is done to limit the number of commands between where a mistake was made and the attempt at outputting the incorrect data, which will hopefully make debugging and correcting the error easier on the player.

The tutorial system of the game kicks in automatically the first time a player enters certain key levels. When a new instruction is awarded or a new data structure is unlocked, a short tutorial will run before the player is given the full puzzle to solve. These tutorials serve to explain basic game mechanics, how to use new instructions, and what newly acquired data structures do. By implementing them only on key levels, we are able to deliver instruction to a player about how to use new tools without interrupting gameplay with a tutorial on every single level. This helps players fend for themselves for the levels following the introduction of a new command or structure, as they will have to apply what they learned to new situations and puzzles in order to be successful. Additionally, players have the option to escape or replay a tutorial at any time.

The hint system for the game is a simple mechanic of several prewritten responses that give general hints on the puzzle. These are designed to be generic enough to not give anything away, but rather encourage the player to think more about certain aspects of the puzzle that may be giving them trouble and help push them towards a solution. The player can click on Computron to receive a hint if they feel the need, and the hints are simply cycled through with each new click.

When a player successfully completes a level, they are graded on their solution against several metrics. The efficiency star is awarded if the solution is computationally inexpensive. This is measured by processing the solution in a virtual machine that tests the solution on arbitrary test cases. The instruction count star is awarded if the solution meets or beats the length of the most efficient solution the

developers could come up with. For levels with memory cards available, the memory star is awarded if the player uses only the minimum amount of memory required to solve the puzzle. This is in line with the instruction count star in the sense that it is measured against the most efficient solution the developers could come up with.

V. Technical Details

Following are the high-level block diagrams and explanations of the major systems in each scene.

In Fig. 3., you can see how the player data per puzzle is managed by the save system in the game. The data for each puzzle is stored in that save file – whether the player has attempted the puzzle or not. When a player first enters new level or successfully completes a puzzle, the saved state of that puzzle is overwritten for that file. This means that if a player decides to revisit a previous level and attempt to earn more stars, all of their current progress on that level is also cached in the save memory of their file. This includes the solution window state of the commands that they last used to successfully complete the puzzle. This will allow players to go back to and revise solutions without having to start over and can also help them examine previous solutions if they need to reference them for a new puzzle level they are trying to solve. Because of the design choice to dynamically load puzzle data into a single scene, we are able to easily cache changed states of puzzles and load them accurately for each player file. If a puzzle has not yet been attempted, the original unused state is loaded. Similarly, the first time they load a particular puzzle level may trigger the awarding of new instructions and/or a tutorial sequence. These only trigger on the first visit to that puzzle, and so the state is updated after the first visit as well.

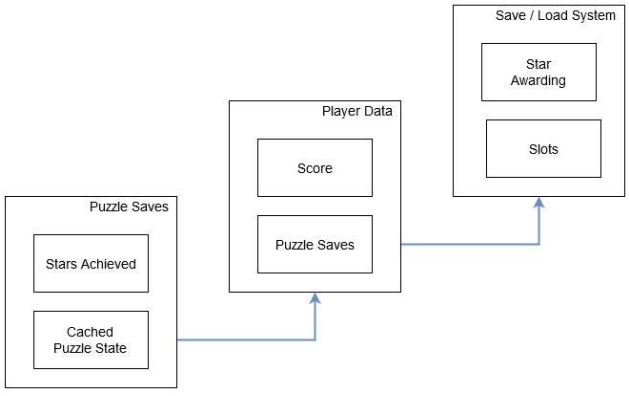

Fig. 3. Block Diagram for the Main Menu scene.

Aside from the puzzle data on each save file, there is also player-specific data. The total score a player has earned, as well as from which levels they earned which stars is saved to their particular instance of a save file.

The load file screen of the menu reads the saved data from a JSON file that is specific to the save slot selected. If that slot has not been previously used or if the data there has been deleted, the slot is marked as empty and will initialize a new save file. The loading system also has game data apart from the player-specific data, which contains information about the star awarding standards for each level. Because this data does not change between players it is stored in a different system from the save files and is used across all saves.

The level selection scene supports mouse controls for navigation, as well as the ability to look ahead of the currently available path and zoom in and out. The actual movement of the character icon on the map is done through direct travel from node to node – there is no free movement involved. By keeping the movement on direct paths, it helps reinforce the way the levels are connected to each other in content. Optional paths branch off at random intervals and include deeper exploration of a particular topic or combining two topics to learn something new.

In Fig. 4, it is more clearly shown how the saved data from a player file is used to populate information in the game throughout different scenes. The player score is displayed on a HUD in this scene, and the stars they have achieved on each level as well as information from their saved puzzle states are used to keep the map current.

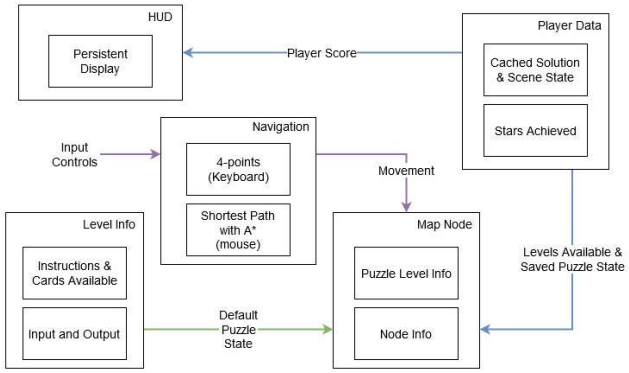

Fig. 4. Block diagram for the Level Select scene

When a puzzle is loaded from the level selection scene, the saved file data is used to populate the dynamic elements of the scene. The puzzle scene is composed of four major components – the actor, the interpreter, the user interface, and the puzzle system. These systems interface with each

other in order to carry out the core function of *Computron*, as seen in Fig. 5.

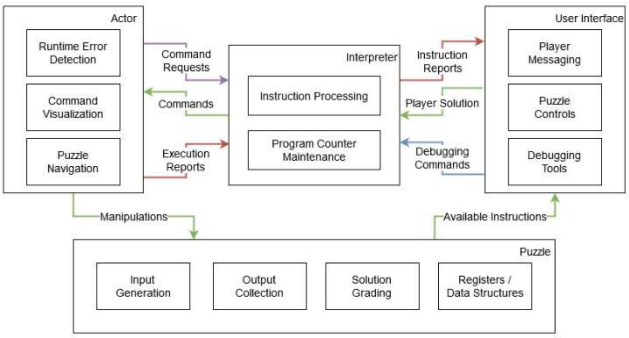

Fig. 5. Block diagram for the Puzzle scene

The Actor system controls all of Computron's actions in the puzzle scene. The system grabs the prepared solution from the Interpreter system and processes them one by one. Each command tells Computron where to move to in the puzzle and how to manipulate data at that point to help visualize the execution of the solution. If the commands have led to an error state, Computron reports that to the player via on-screen text prompts as well as informing the Interpreter system that an error has been encountered. The navigation of Computron in the scene is also managed by this system.

The Interpreter system processes the commands assembled in the solution window by the player and translates them into their algorithmic equivalent. This includes the addresses for jump instructions to jump to in the solution and the addresses of which memory cards are being used, if any. This system also maintains the program counter of the solution as it is being processed.

The User Interface system is in control of allowing the player to interact with the puzzle. It manages all of the static elements of the scene, which includes the majority of the player-interactive pieces of the game. The solution window and instruction cache are maintained by this system and are the means by which the player builds their solution. The actual button controls to play, step, or halt the execution are part of this system, as well as the hint system, tutorial system, and instruction awarding.

The Puzzle system is in control of all of the dynamic elements of the puzzle scene – those that vary between different puzzles. It populates the input data, processes the output, grades the player solution, and maintains the data structures used by players in the game. Additionally, this system relays relevant level-specific data to the User Interface system so that it can accurately build the level, like which instructions the player can use for this puzzle, and the number and type of which cards are available to be used.

VI. 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 our 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 *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 system.

The early playtesting results have shown a lot of promise and positive feedback. Many players have 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. We are excited to unveil *Computron* for you very soon.

Acknowledgement

The authors wish to acknowledge the work from student artists Conner Ramputi and Christian Babcock, who were instrumental in bringing Computron to life, and John Billingham, who created the auditory soundscape of music and effects that bring forth the relaxed atmosphere.

The authors also wish to extend their sincerest thanks to all of the students, mentors, and peers that did playtesting and reported back to us with feedback. Your efforts have been immensely informative and are deeply appreciated.