Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

CodinSpace: An astronomy-inspired serious game for self-learning programming

Diogo Alexandre Casas Novas Pinto

Orientador(es) | Francisco Manuel Coelho

Évora 2025

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

CodinSpace: An astronomy-inspired serious game for self-learning programming

Diogo Alexandre Casas Novas Pinto

Orientador(es) | Francisco Manuel Coelho

Évora 2025

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	|	José Saias (Universidade de Évora)
Vogais	|	Francisco Manuel Coelho (Universidade de Évora) (Orientador) Pedro Patinho (Universidade de Évora) (Arguente)

Évora 2025

This work is dedicated to my family, whose love, support, and encouragement have been instrumental in my journey.

Preface

I welcome the reader to the master's dissertation *"CodinSpace: An astronomy-inspired serious game for self-learning programming"*. This is original work submitted in fulfillment of the requirements for obtaining master's degree in Computer Engineering at the University of Évora.

I am Diogo Pinto, a 24-year-old master's student in the *University of Évora* (UE) and a Web Developer. I decided to do this work as my master's dissertation, because recently I have been gaining interest in Game Development more than ever, and wanted to create a game that could allow anyone to learn programming basics more interactively and with less struggle. I was involved in the writing of this dissertation and development of the game *CodinSpace* from November 2023 to January 2025.

This dissertation is aimed at those who are curious about existing games and platforms for self-learning programming and their features. It is also intended for anyone interested in knowing how to implement a serious game for self-learning programming, including how to teach the programming concepts to the players, how to prepare an interpreter in the browser just for handling player code and how to transform the player code instructions into game state changes that can be immediately visualized by the player.

Acknowledgements

I would like to express my heartfelt gratitude to Professor Francisco Coelho for all the guidance in the making of this dissertation. His unwavering support played a crucial role in the completion of this work.

I deeply thank my mother, father, grandparents, brother, sister, Paulo, Marta, and my entire family for their boundless love, encouragement, and faith in me. They helped me persevere through the difficult moments and stay determined to achieve this significant milestone.

Contents

List of Figures

List of Tables

Acronyms

UE *University of Évora*

XP *Experience*

AI *Artificial Intelligence*

JSON *JavaScript Object Notation*

UUID *Universally Unique Identifier*

HTML *HyperText Markup Language*

CSS *Cascading Style Sheets*

SQL *Structured Query Language*

AR *Augmented Reality*

UI *User Interface*

GUI *Graphical User Interface*

RPG *Role-playing Game*

Abstract

Programming is a discipline that takes a lot of time and dedication to be learned. Certain concepts may be hard to grasp for students, which can lead to demotivation and loss of interest in learning to program. Additionally, when practicing, students may struggle to understand when to apply certain concepts in their code and may not receive enough feedback on the behavior of each part of the executing code. This work explores existing platforms and tools that serve as support for self-learning programming, and after analyzing core features of each one, presents the implementation of a prototype of a space-themed serious game, for self-learning programming, called *CodinSpace*. Finally, an assessment is made regarding the current state of the game and possible improvements are proposed for future work.

Keywords: Video game; Serious Game; Self-Learning; Programming; Skills

Sumário

CodinSpace: Um jogo sério com naves espaciais, para a auto-aprendizagem de programação

Programação é uma disciplina que exige muito tempo e dedicação para ser aprendida. Certos conceitos podem ser difíceis de compreender para os alunos, o que pode levar à desmotivação e perda de interesse em aprender a programar. Além disso, ao praticar podem surgir dificuldades em entender quando cada conceito deve ser usado no código e o estudante pode não estar a receber feedback suficiente sobre o que cada parte do código em execução está a fazer. Este trabalho explora plataformas e ferramentas existentes que servem de suporte à auto-aprendizagem de programação e após analisar as principais características de cada uma, apresenta a implementação de um protótipo de um jogo sério inspirado em astronomia, para auto-aprendizagem de programação, denominado CodinSpace. Por fim, é feito um balanço do estado atual do jogo e possíveis melhorias são propostas para trabalhos futuros.

Palavras chave: Video Jogo; Jogo Sério; Auto-Aprendizagem; Programação; Competências

1

Introduction

Understanding the foundational concepts of programming can be quite challenging, as it demands a very different way of thinking than the learner might be used to in other disciplines. It is common for novice programmers to feel overwhelmed and it typically requires them considerable time and effort before the learned concepts begin to make sense.

In order to truly understand how programming works, the novice programmer must not only know the syntax and the meaning behind each keyword, but most importantly, they must understand how the various code statements are sequentially arranged to reach a desired objective in a computer program. This can only be achieved through lots of practice, by repeatedly writing small computer programs that incrementally introduce the programmer to the various programming concepts.

This dissertation explores various state-of-the-art projects and their approaches to provide effective self-learning programming activities, and also presents the prototype of a space-themed serious game which attempts to create engaging and interactive challenges for self-learning programming.

A serious game can be defined as a game which uses entertainment to actively develop training for new skills and education in any area. [CKXG11] Games of this type can be very effective learning tools, especially for learning concepts or disciplines that require a very dynamic and different way of thinking, such as programming.

1.1 Objectives

This work seeks to achieve two objectives. The first objective is to conduct a review of the state-of-the-art with respect to existing tools and projects designed for self-learning programming. The second objective is to develop a serious game for interactive self-learning programming which can be easily played by complete beginners in programming, but can also present good challenges for already seasoned developers.

1.2 Motivation

One of the main difficulties in the learning process is that while practicing, the newbie programmers may find it hard to correctly understand what is going on step by step in the executing program and what is supposed to be happening in order to reach the desired program objective.

Serious games and other interactive experiences offer an opportunity to make the learning process smoother and more intuitive. By visually presenting programming concepts in level challenges and frequently giving feedback to the player in every execution step of the written programs, the player can more easily understand how each concept works and how to properly write computer programs that successfully complete tasks.

1.3 Structure of the Dissertation

This dissertation is organized to gradually introduce the reader to the theme of self-learning programming and the various features of the developed game. Each chapter and sub-chapter builds upon the last, with the content presented in an order that seemed optimal to the author.

Firstly, in the chapter two, a review is made regarding existing tools and projects for self-learning programming.

Subsequently, in chapter three, the reader is introduced to the proposal for a new serious game for self-learning programming, *CodinSpace*, including a description of the targeted devices, targeted audiences, covered JavaScript programming concepts and game genre.

After reviewing existing projects for self-learning programming and defining a clear proposal for the new serious game in the previous chapters, the implementation and features of the game prototype are explained in detail in chapter four. To complement the explanation of the prototype, at the end of the dissertation, there is a Code Examples appendix, which contains various code examples that may help the reader better understand how the game is implemented.

In chapter five, a formal and mathematical description of the *CodinSpace* game is presented. This includes the structure of the game state, the parts of the state visible to the player, how the state is initialized, the available player actions and sensors, and the conditions that determine when a level ends.

In chapter six, the results regarding the implementation of *CodinSpace* and its learning potential are presented with an overview of the achieved objectives. Additionally, the chapter proposes a series of potential improvements that could be made in future developments, in order to enhance the potential of the game as a tool for self-learning programming.

2

State of the Art

The Internet is a place where anyone can be self-taught to learn basically anything. There are millions of online resources that are good sources of knowledge, giving people unprecedented access to knowledge and resources, to learn about any area of their liking.

This chapter explores the state of the art of a specific type of existing online resources, which are serious games and gamified websites which, similarly to *CodinSpace*, aim to offer an engaging and effective environment for self-learning programming.

The serious games for self-learning programming in the state-of-the-art aim to make this learning process as productive as possible and reduce the friction to learn difficult concepts, by presenting them in an engaging and interactive way. [HNVIPRC14] Many of these games seamlessly integrate the concepts directly in the visuals and mechanics, making them more intuitive to understand and resulting in a much higher chance that the player will effectively learn them.

An article written by Matej Zapušek and Jože Rugelj in 2013 at the University of Ljubljana [ZR13] introduced a game designed to teach the concept of variable in programming. The game consists of four parts, each with its own type of exercise and questions that help the player learn about variable declaration, assignment, and data types. In addition, a 2014 conference paper by Raquel Hijon-Neira, Ángel Velázquez-Iturbide, Celeste Pizarro-Romero, and Luís Carriço [HNVIPRC14] presented

a system, called ProGames, which consisted of various serious games with 192 different exercises, for self-learning programming skills. Progames was developed with Greenfoot [Gre] Java game development environment, which allowed the creation of visual interactive experiences for the exercises. More recently, a 2020 article by Santiago Schez-Sobrino, David Vallejo, Carlos Glez-Morcillo, Miguel Á. Redondo, and José Jesús Castro-Schez [SSVGM$^+$20] proposed a serious game based on *Augmented Reality* (AR), called RoboTIC, that aimed to facilitate the learning of programming concepts for elementary school students. The game consisted of a series of 3d AR levels where the player must guide a robot to reach the goal of the stage, by assembling various pieces in a road, each with a different instruction for the robot.

There are also master's dissertations that were written at Universidade de Évora, which involved the development of serious games for self-learning programming. In 2011, José Manuel Cobiça Duarte [Dua11] published a dissertation about a study on whether serious games can be considered valid pedagogical tools and about the development of a serious game for learning *Structured Query Language* (SQL). The players learn SQL by completing various programming challenges that progressively get harder, in a 3d environment. Later, in 2016, Pedro Farias Mateus [Mat16] published a dissertation, about the development of a serious game for middle schoolers to learn basic programming. The game has its own visual programming language, which consists of assembling 2d blocks that are transformed into Python [Pyt] code.

Over the years, many serious games for self-learning programming were released, teaching programming fundamentals in different ways and in a huge variety of programming languages. However, it is important to consider that there is still a lot of potential for improvement in this field. A 2018 conference paper by Michael Miljanovic and Jeremy Bradbury [MB18], reviewed 49 serious programming games with respect to likability, accessibility, learning effect, and engagement. The authors identify a number of open problems in the literature. One of the biggest problems they state is that serious programming games are often developed independently, without learning from previous games and defined methods in the existing literature. Games that are developed without considering the existing literature are more likely to repeat mistakes made by previous game developers, which does not lead to significant improvements in the way programming concepts are taught.

By exploring the state-of-the-art and analyzing the various approaches and methodologies employed in each project, it is possible to select which are the most adequate for *CodinSpace* and ensure a consistent and meaningful gameplay. For that, the focus of this state-of-the-art chapter will be mainly oriented towards five of the most popular and well-successing serious games and platforms for self-learning programming:

CodeCombat - Online game that aims to teach self-learning programming in an enjoyable and fun experience;

CheckiO - Platform that teaches Python and TypeScript in engaging gamified programming challenges;

CodinGame - Platform that provides an impressive variety of different games that present to the player different challenges and opportunities to learn and improve programming skills;

Elevator Saga - Programming game in which players write JavaScript code to successfully transport people on elevators to the desired floors;

Scratch - High-level, block-based, visual programming language as an educational tool focused primarily on children.

2.1 CodeCombat

CodeCombat is an online game that aims for self-learning programming in an entertaining and fun experience. The game has a lot of *Role-playing Game* (RPG) elements, such as a leveling up system, dungeons, enemies, equippable items, and multiple characters, each one with its own characteristic status. CodeCombat also uses its own engine and interpreters for the player code.

In each mission, the player controls a character that must reach the end of the level, by going through lots of traps and enemies. To control the character and make it do actions, the player must write code in a language (C#, Python or JavaScript). During the mission, the game also gives hints on how the player can write better code to achieve the objectives.

After completing a level, the player gets gems, experience, items as reward, and can even get more bonus rewards if the written code is clean and organized, motivating the player to improve the current solution to the level.

Gems can be used to buy new items in the shop. These items provide bonus status for the character (e.g. more health, attack damage, etc.). Some items even have special abilities/skills that can be used as new callable functions in-game (throwAt, castDrainLife, attackRange).

There is a list of game achievements. Each time the player gets a new achievement, the game rewards gems and *Experience* (XP). Each time the player has enough experience to level up, the game will unlock new items in the shop.

The player must think logically and use the programming fundamentals adequate to each situation in order to traverse the level correctly (conditionals, loops, functions, parameters, variables, etc.).

If the player doesn't want to stick with the story mode, it is possible to challenge other players online in a competitive *Artificial Intelligence* (AI) league. Matches can be played in various game types, each with its own set of rules. To beat the opponent, the player must write code instructions that summon and control allies. The game has a wide range of functions that can be used to make each ally bot walk, attack, defend, heal, and more. The player must know when to use these functions and must write code with conditions, loops, variables, and more, to effectively manage each bot and outsmart the opponent. The player's objective depends on the type of game, so it can include defending a crystal from enemy waves, defeating the biggest number of enemy bots, gaining the most territory, and more.

2.2 CheckiO

CheckiO is a platform that teaches Python and TypeScript in engaging gamified programming challenges.

The experience is divided into various chapters represented by islands in a world map. Each chapter consists of a group of various levels, that focus on certain programming aspects and fundamentals (e.g. Strings and Integers, Lists, Dictionaries, Sets, Objects). The difficulty of the challenges can be set as easy, normal, or advanced.

In each level, the player must solve a programming problem with a code editor in the browser. The game starts by stating the problem and the rules of the challenge. If the player doesn't know how to solve a problem, it is possible to get small hints with code examples;

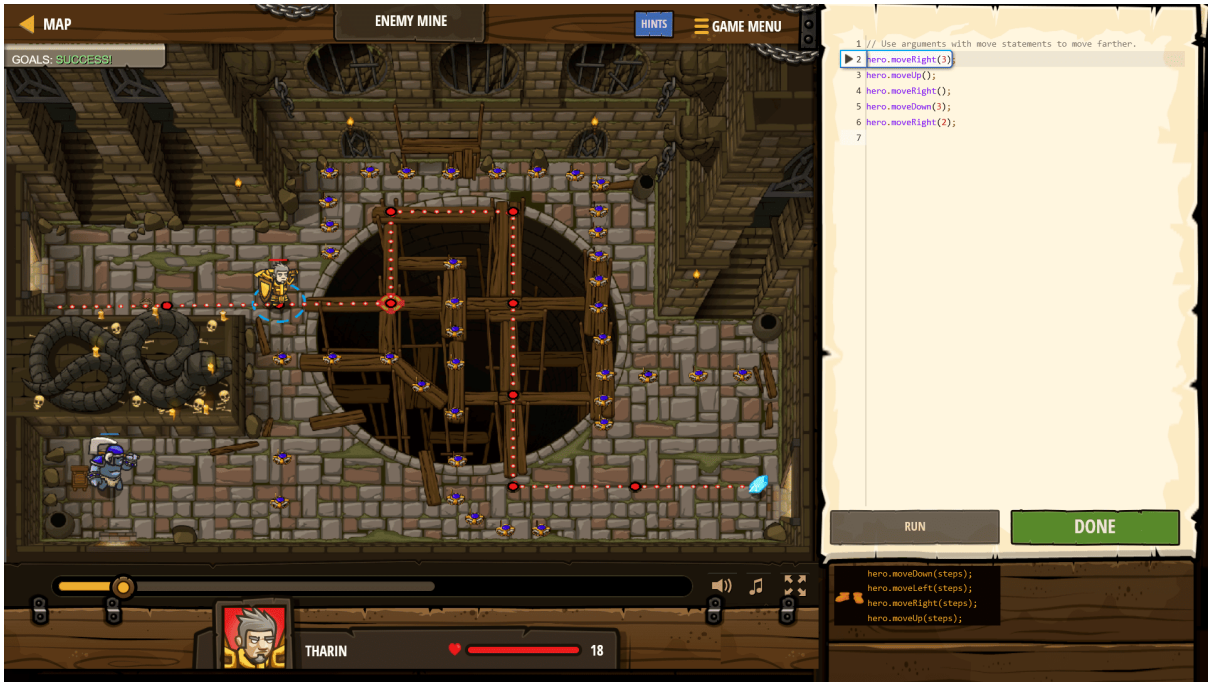

Figure 2.1: A CodeCombat level, with the level editor and available player actions on the right, and the game view on the left.

When the written code is executed, the platform will run unit tests with assertions (e.g. `assert.equals()`) to check if the program output matches the expected output. If the problem is solved correctly, the game rewards points.

The platform saves the various solutions achieved by each player. At the end of each level, three different solutions from other players will be displayed to the current player (most creative solution, most clean solution and third party solution), to help recognize what could be improved, and what other alternatives to the written code would be possible.

2.3 CodinGame

CodinGame offers an impressive variety of different games that present to the player different challenges and opportunities to learn and improve the programming skills. The challenges can be solved with 27 different programming languages (e.g. C, C++, Java, JavaScript, Lua, Python, Ruby).

During a level, the game provides lots of different information on the screen, in various windows:

Game. Displays the game state visually (the player, enemies, obstacles, etc.);

Code Editor. The player can write code here and change the programming language at any time. The editor also has syntax highlighting and code suggestions to help write better code, with the correct syntax;

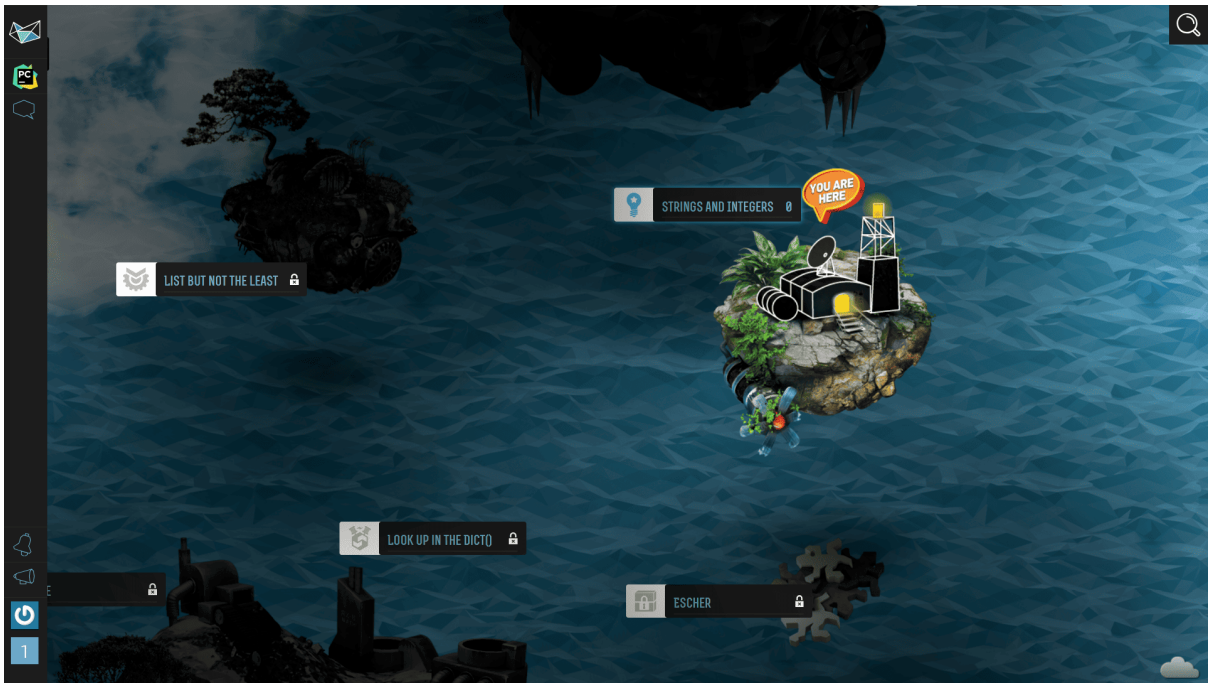

Figure 2.2: CheckiO Chapter Selection Menu

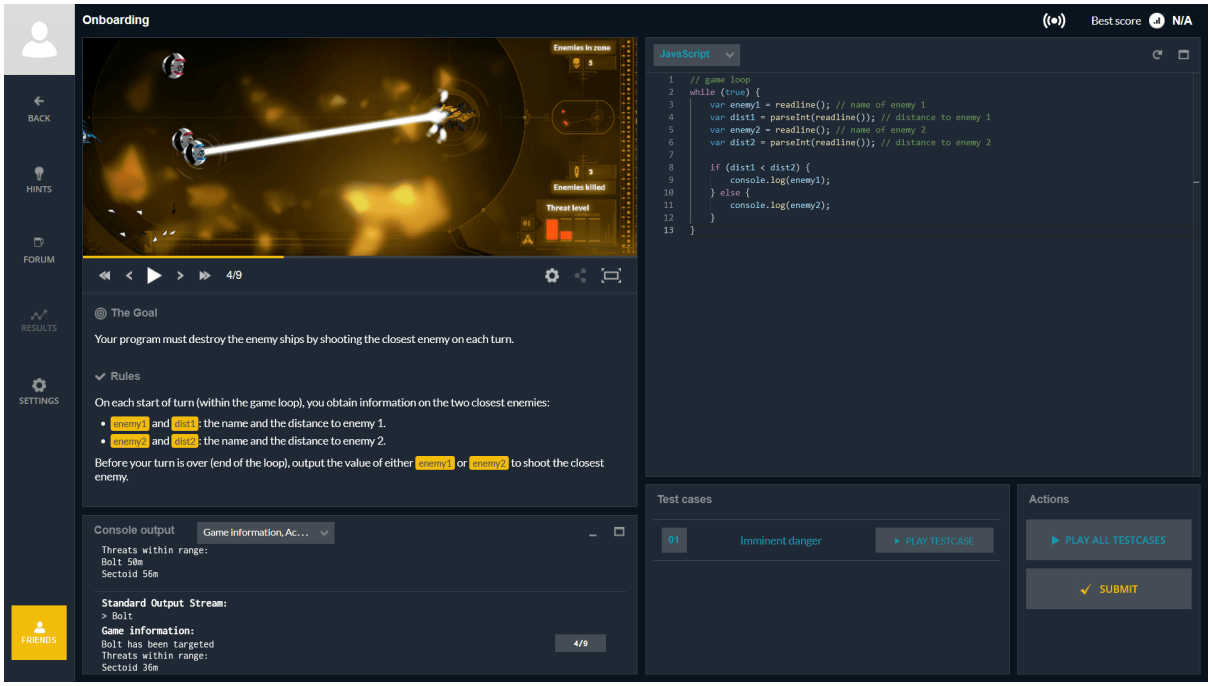

Figure 2.3: Level challenge inside CodinGame, with code editor, info text, console output and the game window

Console Output. Displays all the information of each iteration/step logged with console.log in the Standard Output Stream and also displays other relevant game insights (*e.g.* : Targeted enemy, distance to each enemy, game state, etc.). The player can rewind the game state to past steps by scrolling to previous logs and replay the animations of the game;

Test Cases. Holds every test case that will be run in the level. The player can run tests individually by selecting them here;

Actions. Contains a button to play all test cases at once and a button to submit the solution.

After completing a level, the game will display the percentage of points scored, ask the player for a level rating (so that the creator of the level can improve with feedback) and reward the player with XP.

There are various achievements that can be obtained on the different challenges, and the players can check their friends' progress and achievements on each level.

2.4 Elevator Saga

This is a programming game in which players write JavaScript code to successfully transport people on elevators to the desired floors. The algorithm written by the player must have the correct instructions written as efficiently as possible in order to complete the 18 different challenges of the game.

The website provides a documentation page where the player can learn all the possible functions to control the elevators (`goToFloor`, `stop`, `maxPassengerCount`, `getPressedFloors`, `loadFactor`, and many more).

While the code is being executed and the elevators are in movement, the game will be presenting to the player small statistics (Transported count, elapsed time, transported per second, avg waiting time, max waiting time, number of moves).

Many solutions from other players can be found on Elevator Saga Github repository. Some complete all the challenges without problems, while others are not very optimized and don't perform so well. This allows the players to compare their code with others, learn other methodologies, and find out what are the weak points of each algorithm.

2.5 Scratch

Scratch is a high-level, block-based, visual programming language as an educational tool focused primarily on children. The platform provided for the development of Scratch projects is very community-driven, counting over 164 million different projects (e.g. games, animations) and 135 million users.

Scratch is used in schools and college, because of its high potential in teaching programming concepts in a fun and interactive way.

The platform has a project editor that allows its users to create their own projects, structured in different sections:

Figure 2.4: Elevator Saga Game Website

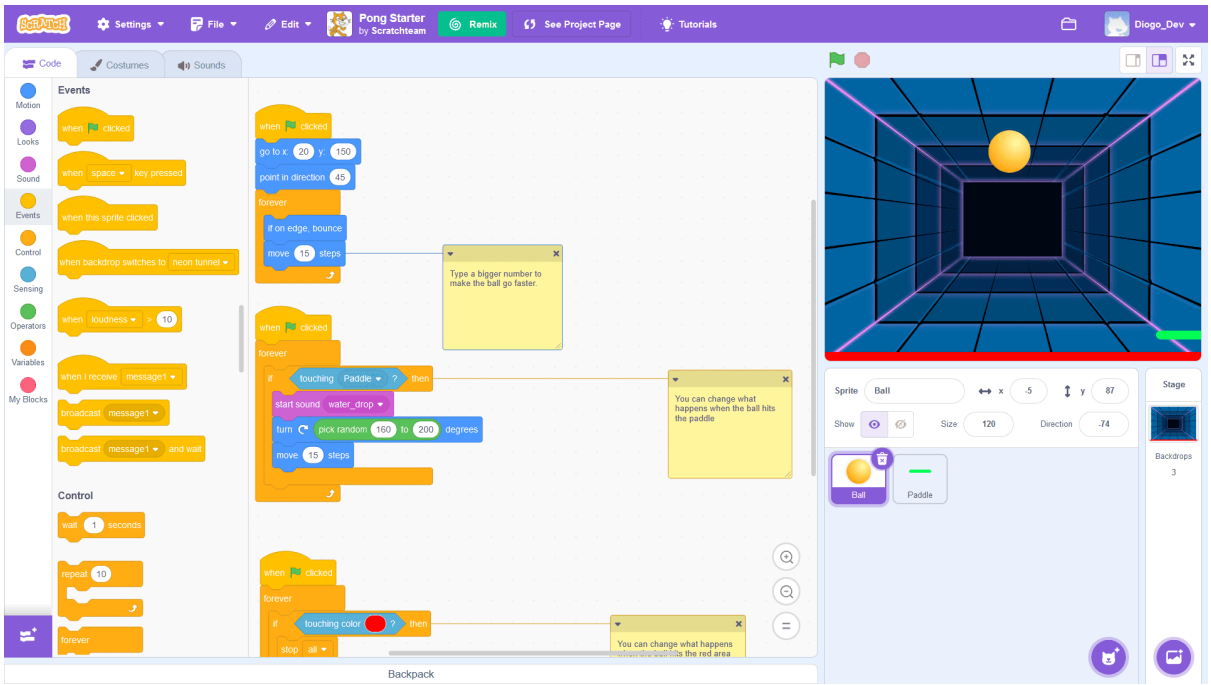

Figure 2.5: Scratch Project Editor, with blocks logic that implements a Pong game

Code

The code section has all the visual programming blocks that can be used to create the project scripts. There are nine default types of block that can be used:

Motion - Used to control the position of each sprite[1] (*e.g.* `moveXSteps`, `turnXDegrees`, `goToXY`, `changeXBy`, `setXTo`, etc.);

Looks - Control each sprite's appearance (*e.g.* `changeSizeByX`, `changeColorEffectByX`, `show`, `hide`, `switchCostumeToX`) and allows the use of say (display a bubble message above the sprite) and think (changes the state of the sprite and also displays a bubble message with the thinking state enabled on the sprite) (*e.g.* `sayXForYSeconds`,`thinkXForSeconds`, etc.);

Sound - Allows the management of all the sound in the project (*e.g.* `playSoundXUntilDone`, `stopAllSounds`, `changePitchEffectByX`, `changeVolumeByX`, `setVolumeToX`, etc.);

Events - Event listeners. If the event is fired, then the event block and all the following blocks attached to it will be executed (*e.g.* `whenGoClicked`, `whenSpaceKeyPressed`, `whenThisSpriteClicked`, `whenLoudnessGreaterThanX`, `whenIReceiveMessageX`, `broadcastMessageX`, etc.);

Control - Blocks used to control the flow of the scripts. (*e.g.* `waitXSeconds`, `repeatXTimes`, `forever`, `ifXThenYElseZ`, `waitUntilX`, etc.);

Sensing - Sensor blocks that give information about the current state of the project (*e.g.* `touchingX`, `touchingColorX`, `colorXisTouchingY`, `distanceToX`, `askXAndWait`, `keyXPressed`, `mouseX`, `mouseY`, `setDragModeDraggable`, etc.);

Operators - Can do math and logic operations (*e.g.* `xPlusY`, `xMinusY`, `XGreaterThanY`, `XAndY`, `XOrY`, `notX`, `XContainsY`, `lengthOfX`, `XModY`, `roundX`, etc.);

Variables - Blocks that allow the getting and setting of variables (*e.g.* `setVarXToY`, `changeVarXByY`, `showVariableX`, `hideVariableX`);

My Blocks - Here, users can create their own blocks, with their own purpose and logic. To create a block, the user can add various inputs (number, text, or boolean) and labels.

The user can also add extensions that provide additional functionality and new blocks.

Costumes

This section allows the user to choose and modify the sprites for the project. There are hundreds of different sprites already available and ready to be added to the project.

Sounds

This section is for all the sounds of the project. Here the user can import sounds or choose them from a huge gallery made available by Scratch and modify their speed, loudness, fading, and much more.

[1] A sprite is a 2-dimensional image or repeating loop of images that can be placed in games to represent anything visually, such as trees, clouds, characters, enemies, equipment, and much more.

Execution

This window displays the project in execution. It can help users quickly test everything about the project as they add new code blocks to the scripts. This allows for immediate feedback and an overall more visual and intuitive development experience.

Sprites

Here, the user can control which sprites are in the project and how each one will behave (which code blocks are being executed for each sprite)

2.6 Conclusion

This chapter analyzed various platforms for self-learning programming, highlighting their features and how each one managed to blend programming education with entertainment and gamification.

CodeCombat teaches C#, Python, and JavaScript, by immersing the player in an RPG where it is possible to level up, unlock new characters, and buy items that unlock skills as programming functions.

CheckiO teaches Python and TypeScript in gamified programming challenges, where concepts are split into various chapters, visually represented as islands, that can be unlocked progressively. CheckiO also stores the best player solutions for each challenge, allowing players to compare their solutions to others and learn from them.

CodinGame presents a huge collection of games for self-learning programming in over 27 different programming languages. The player interacts with each game by writing code that triggers in-game actions. Every game contains its own challenge and helps the player understand the fundamentals of programming.

Elevator Saga is a JavaScript programming game, where the objective is to write code that runs the most effective algorithm to control the game elevators and transport people to their desired floors, in as little time and moves as possible.

Scratch is a community-driven platform with its own block-based visual programming language. In Scratch anyone can create their own games, tools, and animations, by assembling code blocks together in an interactive interface. Scratch also serves as a powerful tool for learning programming and is used in schools all around the world.

Considering that the state-of-the-art regarding platforms and games for self-learning programming has been explored, and the foundational concepts behind each platform have been presented, it is now feasible to propose a new serious game for self-learning programming, drawing ideas and inspiration from these platforms.

3

Game Proposal

This dissertation proposes the development of a new serious game aimed at self-learning programming. In this chapter, we present each decision made regarding its development and outline the setting and structure of the game.

3.1 Game Name

It was decided that the most interesting and fitting name for this game is *CodinSpace* (*i.e.* Code in Space), because the game challenges revolve around the player having to write and send code to a spaceship to make it travel across space.

3.2 Chosen Game Engine

There are several modern game engines available for developers today, each with its own strengths, limitations, and intended use cases. The three engines considered for the development of *CodinSpace* were Unity, Unreal Engine, and Godot.

Figure 3.1: Godot Engine logo

Unity [Uni] is a widely used commercial game engine known for its flexibility and large community support. It is particularly popular for both 2D and 3D game development, offering strong integration with C# [Mic] as its main programming language. Unity provides a powerful visual editor, which allows developers to create the game logic without having to write code, a huge asset store with many resources from lots of different creators, and robust cross-platform deployment tools. Although it is not an open-source engine, it is free for small-scale projects and widely adopted in both indie and professional game development. It is a very popular game engine, and a very common first choice for many game developers.

Unreal Engine [Unr], developed by Epic Games [Gam], is another leading game engine, especially designed for the development of high quality 3D games and tools. It uses C++ [Wik] as its main programming language and also provides a visual scripting system called Blueprint. Unreal Engine follows a royalty based licensing model and is heavily used in AAA[1] game production. It is a powerful and constantly evolving engine with a wide range of advanced features, but it is not well suited for the development of 2D games.

After considering the features and requirements of this project, the selected game engine for the development of *CodinSpace* was Godot.

Godot [Goda] is an open-source, cross-platform game engine that supports both 2D and 3D game development. It has a huge community of contributors who continuously work on new features, bug fixes, and a wide range of plugins. Godot uses its own scripting language, GDScript [GDS], which is fully integrated and optimized for the engine. However, developers can also use C#.

Games in Godot are structured as trees of nodes, which are the smallest building blocks of a project. There are over 200 types of nodes, each one with a different functionality, such as sprites, sounds, tilemaps, cameras, and particle emitters. These nodes can be grouped into reusable components called scenes, which represent parts of the game (*e.g.* characters, obstacles, levels, menus). This modularity approach allows for maintainable and scalable game design.

CodinSpace was developed using Godot version 4.3, which was the most recent version available as of December 2024.

[1] In the videogame industry, games developed by major studios with high budgets and big development teams are classified as AAA (triple-A) games.

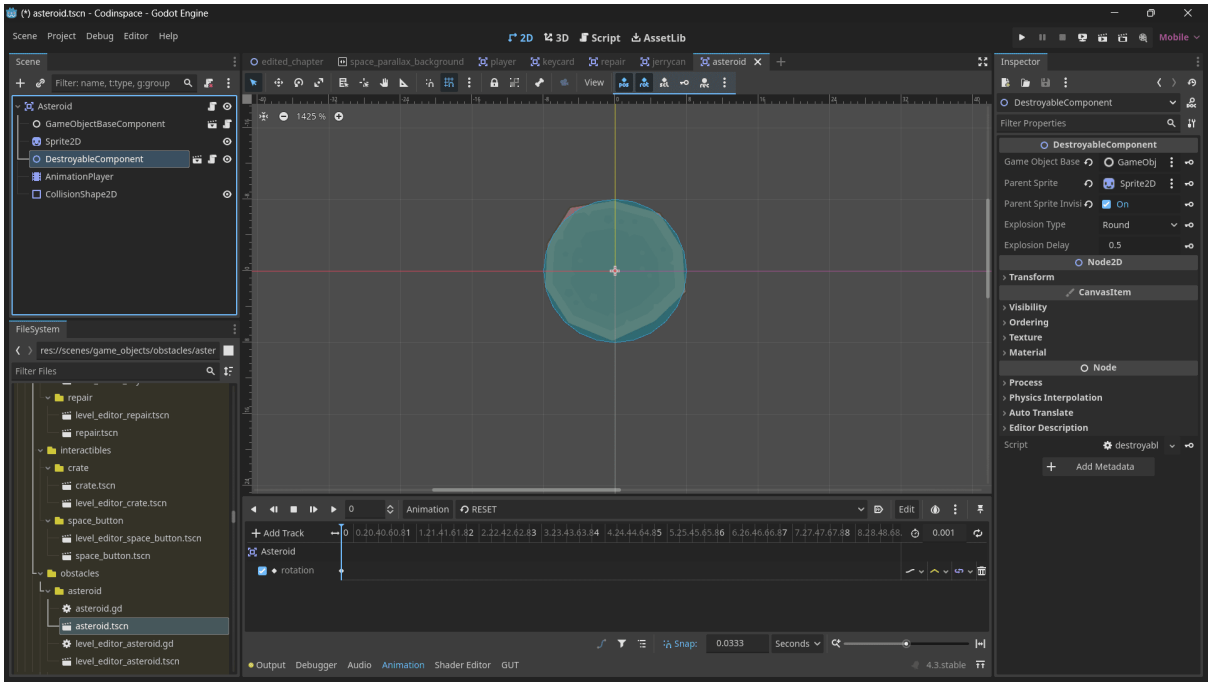

Figure 3.2: *CodinSpace* project opened in Godot

3.3 Targeted Programming Language

The programming language chosen for the challenges in *CodinSpace* is JavaScript [Moz]. JavaScript is an interpreted or just-in-time compiled programming language, commonly used for dynamic web development. It is one of the main technologies of the World Wide Web, along with *HyperText Markup Language* (HTML) and *Cascading Style Sheets* (CSS). It is used in 98.9% of all websites [W3T], to implement client-side dynamic behaviors.

Features of JavaScript:

- Dynamically typed;
- Single-threaded execution model;
- Prototype-based inheritance;
- Supports various programming styles (functional, imperative, object-oriented);
- Asynchronous operations through promises and async/await patterns;
- Dynamic generation of scripts from strings with the eval function (Used in *CodinSpace* for sending code from the game engine to the browser).

The official standard for JavaScript is the ECMAScript Language Specification (ECMA 262) [Ecm]. The ECMAScript standard was invented by Brendan Eich at Netscape in 1996, and its first appearance was in the Netscape Navigator 2.0 browser. Today, every modern browser supports ECMAScript and the specification keeps progressively receiving new publications with new features and concepts.

According to a Statista poll [Sta], JavaScript is currently the most used programming language worldwide as of August 2024, with 62.3% of the survey participants using the language. JavaScript is a very good starting point for anyone who wants to step into the world of programming. By learning

JavaScript, it becomes possible to build fully interactive websites, back-end servers, games, and much more. This makes JavaScript one of the most useful programming languages and completely mandatory for many programming jobs.

One of the main reasons for choosing JavaScript for *CodinSpace* is its availability in the browser. To execute JavaScript in a Godot game, the game just needs to be exported in Web format, and it becomes possible to execute JavaScript through an interface in Godot (JavaScript Bridge) [Godb]. This bridge is specifically designed to allow seamless communication between the Godot Engine and the browser's JavaScript context, allowing code to be sent in strings from the game.

3.4 Targeted devices

CodinSpace is designed to run on computers, directly in the Internet browser. This makes it easier to start playing the game without downloading or installing anything else. I chose the computer as the device because it is the most convenient and simple way to interact with the game functionalities. The game involves writing code, analyzing terminal logs, watching the spaceship actions, and much more, so it becomes impracticable to place all these elements on a mobile or console device without hurting the player experience and main purpose of the game.

3.5 Targeted audience

This game is meant for anyone who has zero programming knowledge, who wants to take the first step into the world of programming, and also people who already know how to program but want to challenge their skills with programming problems and puzzles that match their skill level.

3.6 Genre

CodinSpace is designed as a 2d top-down[2] space-themed puzzle serious game.

CodinSpace was created as a top-down game where the level objects and the player are positioned in a 2d matrix, to ensure that every situation displayed in the game is visually intuitive and simple to understand. The main objective of the game is to help the player learn programming, so the game view must be simple and free of confusion.

It was also decided that *CodinSpace* would be a space-themed game, because it is a very broad theme liked by a huge audience and because space, especially in science fiction, encompasses the possibility of many different scenarios that are very interesting when presented in a game (*e.g.* planets, asteroids, spaceships, lasers, signal broadcasts, etc.).

3.7 Gameplay Description

The main gameplay of *CodinSpace* consists in solving 2D puzzles that teach programming concepts through various challenges. The player must lead a spaceship from the start of the level to the defined target without colliding with the obstacles on the way. To control the spaceship, the player

[2] In top-down games, the camera angle shows the player and the areas around them from above.

Figure 3.3: Stardew Valley, a popular top-down RPG game

must write JavaScript code that periodically enables various actions and sensors that allow the spaceship to travel safely through the level. Levels gradually increase in difficulty, introducing loops, conditionals, arrays, and other concepts as the player progresses. The player can also access bonus levels, which serve as challenges branching off from the main progression route, specifically designed to practice and reinforce learned concepts.

3.8 Covered Programming Concepts

The current prototype has various challenges that cover the most important basic programming concepts, such as variables, conditionals, loops, and arrays. Some of the existing game objects, actions, and sensors were designed intentionally with the objective of teaching concepts in an engaging and entertaining way for the player.

Output - Output in *CodinSpace* is represented as space signals, which can be emitted with the `emit()` function. It was decided that outputting like this instead of using `console.log()` keeps the concept contextualized in the game environment, and the player can use the signals to open doors, crates and to complete the levels;

Variables - To open certain doors it may be required that the player declares variables, finds crates with values, performs operations with them and emits the final value as a space signal;

Conditionals - Asteroids, lasers and other obstacles can be avoided by checking spaceship sensors, with conditionals (`if`, `else`);

Loops - Repeating patterns of operations and movements of the spaceship may force the player to use loops in the code (`while`, `for`);

Arrays - In various situations, the player might have to find values from crates scattered across the level, and declare and manipulate arrays that work with these values. The arrays can then be emitted as space signals to unlock other parts of the level.

3.9 Conclusion

This chapter outlined important decisions made in the development of *CodinSpace*, including the choice of game name, engine, programming language, target devices, audience, gameplay mechanics, and covered programming concepts. Each decision was made with the goal of creating a meaningful experience for the player, making *CodinSpace* not only a valuable tool for self-learning programming, but also an engaging and enjoyable game that sparks curiosity and motivation.

4

Prototype

In this chapter, the implementation of *CodinSpace* is detailed, describing every feature and how the core components were implemented. The chapter starts with a high-level view of the project, describing what is the player's objective, what are the existing game objects, how the *User Interface* (UI) works, and what are the game menus that the player can go through. Then, we dive deeper and look at all the actions and sensors that the player can activate and the existing interpreter and game constraints. Afterwards, the most technical parts of this project are analyzed, by looking at the game state management, how the game handles advancing and rewinding the game state, and how the player JavaScript code interpretation works. Finally, it explores the chapter editor and level editor tools, which enable the community to create their own content and present their own *CodinSpace* challenges.

4.1 Player

The player takes the role of a space programmer, a fictional type of astronaut that controls the spaceship with its own programs. The player will embark on a huge adventure, traveling many light years, through lots of obstacles and dangers, with the main purpose of learning and improving programming skills and becoming the best space programmer!

Figure 4.1: The player can choose their own spaceship and customize its color

In every level, the player will be challenged to write JavaScript code to correctly control the spaceship through all the obstacles, with the objective of bringing it from the starting point, to the target zone. To ensure that the spaceship reaches the target, the player must write the programs carefully, calling the spaceship action and sensor commands with the right syntax, logic, and in the right order.

Actions allow the spaceship to perform tasks such as flying forward, turning, pressing buttons, opening doors, or emitting signals.

Sensors provide feedback about the spaceship or its surroundings. For example, sensors can detect the amount of fuel left, identify obstacles in the spaceship's path, determine whether doors are open, or calculate distances to nearby game objects.

The spaceship has various properties that the player must account for, including its position (x and y coordinates), heading (in degrees), remaining fuel, and health. In the game code, the player data is represented as a dictionary, which contains all the previous properties and other boolean values which indicate if the player is currently using the press, open or emit actions.

4.2 Level Representation

A level is represented as a 2-dimensional space on a top-down view. It contains a spaceship that the player can control, a target that is the ultimate goal of the level, and various objects (gates, buttons, coins, keycards, crates, etc.) that the spaceship can interact with.

Knowing that the main objective of the game is to serve as a tool for self-learning programming, each level must be designed with the intention of presenting or enhancing knowledge of specific programming concepts. The layout of the level, including the start position of the spaceship, the positioning and properties of each object, and the position of the target, all are extremely important to ensure that the player encounters the ideal challenge to correctly learn the concepts, without feeling overwhelmed and confused.

CodinSpace has many game elements and features present in its levels that are purposely designed to play a role in providing the best self-learning programming environment.

First of all, the code editor allows the player to write JavaScript programs to control the spaceship. The common way of controlling characters with the use of arrow keys and other keys to interact with objects as seen in many games is absent. The only way for the spaceship to move around the level is by executing JavaScript code. This ensures that the player's experience is totally focused on writing code and applying programming concepts that safely controls the spaceship from start, to the target destination.

The player can also fully control the state of the JavaScript programs, by advancing, rewinding, or

pausing execution. Pedagogically, this has huge value, because it is possible to easily re-watch the various level moments, learn from past mistakes, and improve the written code. The game state is always synchronized with the executed program state, so the spaceship and the objects always react to the actions and sensors at the same time that their respective functions are called during execution.

Game objects such as gates, asteroids, and keycards play an important role in creating challenges and difficulties that can be visually perceived and that represent programming problems. This ensures that the player's knowledge is tested by applying the programming concepts in practice.

Another important feature in the levels is the consoles section, which contains the game console and the playground console. The game console gives textual information to the player about what happened in the various steps of execution, such as what spaceship actions the player activated, which objects were opened, collected, pressed, and what values were returned by activated sensors. This console, combined with the buttons to control the program execution state, enables the player to actively analyze the outcomes of the written programs in various different ways, which can be a very productive debugging activity. On the other hand, the playground console serves as a separate tool for the player to run any JavaScript code, separate from the code written in the code editor. A playground console provides more possibilities for the player to practice programming because it is possible to quickly try code from learned concepts and see what happens, without resulting in losing the level and without changing the game state in any way.

Constraints are also a crucial part of *CodinSpace*. Levels often present various constraints, such as maximum steps, code characters, allocated memory bytes, and execution runtime. They force the player to write JavaScript programs that are concise, apply the right concepts, and do not consume too many machine resources.

Lastly, it was also decided that all objects, including the spaceship and the target, can only be placed on integer coordinates (*e.g.* (0,0), (2,6), (9,14)), never on coordinates with decimal values (*e.g.* (0.234, 4.8), (0.383, 17.32)). This was done with the purpose of avoiding harder calculations, not overwhelming players who are just learning how to program, making the game more predictable, simple, and therefore more beginner friendly.

Programatically, all the information about the level is represented in a dictionary, containing various sub-dictionaries with information about the player, constraints, target and game objects. During the execution of the player code, three additional dictionaries are created, which hold information about the code execution steps, the emitted space signals and the created console logs.

4.3 Code Editor

The code editor is where the player can write the JavaScript code that will send commands to the spaceship. When the player enters the level, the code editor will be completely empty and all the JavaScript code must be written from scratch.

The "Run" button on the bottom right corner of the code editor can be used by the player to submit the written JavaScript code to the game. Once the "Run" button is pressed, the game level flow starts and the written code is sent to a sandboxed interpreter instance in the browser, responsible for interpreting the game JavaScript code and returning the spaceship commands.

After the player code is submitted, the code editor becomes read-only, and the interpreter will have received the code for step-by-step execution, so the player will not be able to make new changes to the code unless the level is restarted.

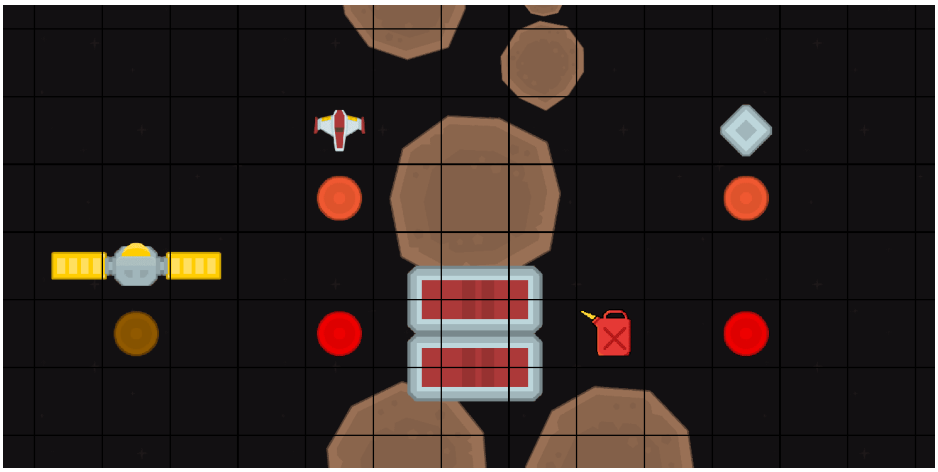

Figure 4.2: A *CodinSpace* level. The player must press the button on the left to open the gate between the asteroids and reach the target on the right. The level also includes collectible coins for points and a fuel jerrycan near the gate to refill the spaceship's tank.

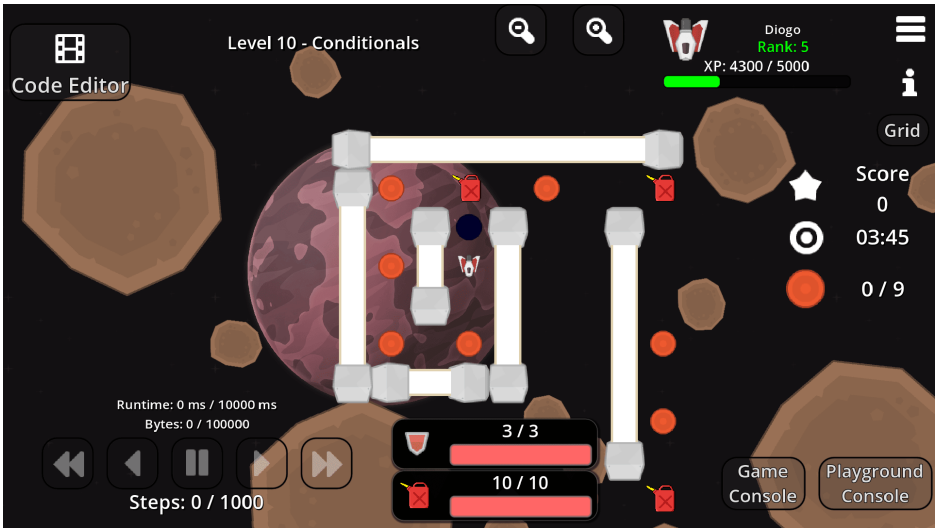

Figure 4.3: In-game view showing the level layout with a top-down camera. UI elements include the code editor button (top left), step manipulation buttons (bottom left), fuel and health indicators (bottom center), and the information panel, score, time, and coin indicators (right side). Additional buttons for game console and playground console are also shown, along with level name, zoom controls, player rank, and XP at the top.

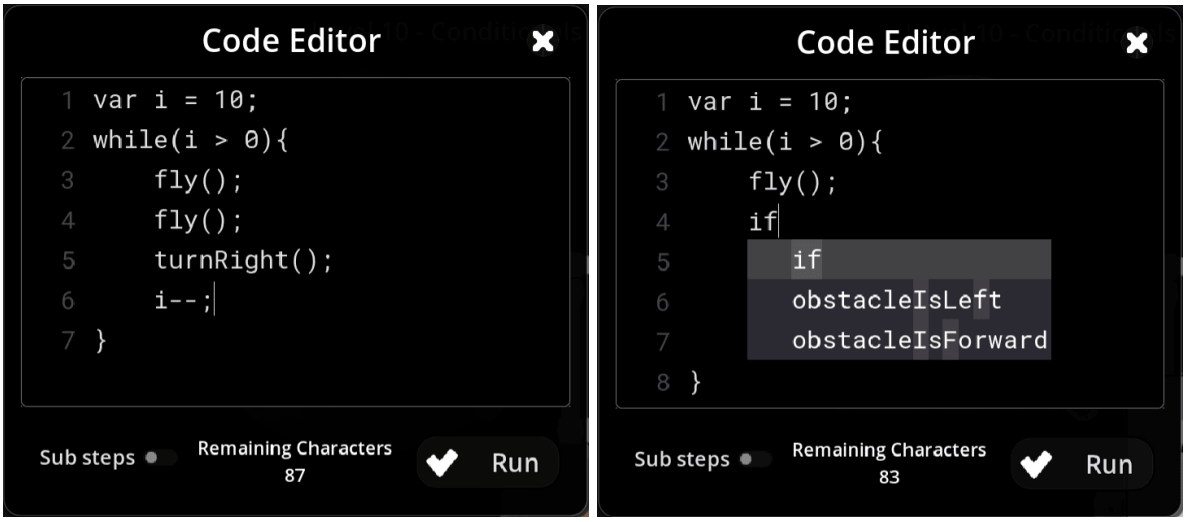

Figure 4.4: Code editor interface. The left image shows JavaScript code written by the player, along with the substeps toggle (bottom left), remaining characters indicator (bottom center), and the run button (bottom right). The right image displays a list of code suggestions.

To close the editor, the player must press the cross button in its top right corner, so that the editor does not clutter the game interface and obstruct the player vision. To reopen the code editor, the player must press the "Code Editor" button in the top left corner of the game screen.

4.3.1 Code Suggestions

While the player is writing code, the editor suggests code pieces that can be selected to autocomplete the code currently written. (`if`, `else`, `while`, `for`, `var`) This feature can be useful in cases where the player does not remember the syntax or certain keywords of the JavaScript programming language.

4.4 Program Execution Manipulation

After the player submits their JavaScript code, the interpreter begins executing it incrementally, one step at a time. In this context, a step refers to a unit of execution identified by the interpreter, such as an instruction, expression, or keyword encountered during the parsing and evaluation of the player's code. The state manipulation buttons allow the player to advance or rewind the steps of execution in the state of the game to a later or earlier state, respectively.

Each step may have new commands given by the player to the spaceship, new game object changes, or internal changes in variables. This feature allows the player to see the game state changing like a movie, because it is possible to advance or rewind the state to whatever step the player wants. This freedom is a very powerful educational aspect of the game because if the player has doubts about what a specific instruction did previously, they will not be stuck on the current execution step and can always rewind to that instruction and then return to the present step when they fully understood what happened previously.

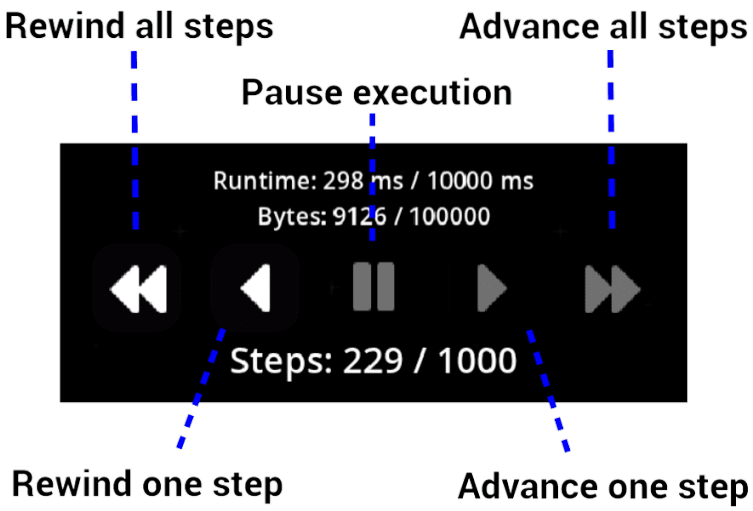

Figure 4.5: State manipulation buttons. From left to right: rewind all steps, rewind one step, pause execution (center), advance one step, and advance all steps. Near the buttons, information is displayed about the interpreter's execution steps, allocated memory, and runtime.

4.4.1 Step types

The effect of each step is obtained by requesting the JavaScript Interpreter to advance the code execution. However, while parsing the code, the JavaScript interpreter considers all keywords, expressions, and instructions to be steps. We believe that this level of detail in the program progress is, in general, too fine for a positive pedagogical effect on a beginner programmer so, by default, our choice is to show steps only for instructions. This option can be changed with the "Substeps" toggle. By default, the game registers every single step in the memory, but will only display to the player the execution steps that are considered full instructions (full statements), so that only more relevant/important steps are returned back to the game, with the spaceship commands and game changes.

There are three main types of steps in the interpretation of JavaScript code:

Instructions - Represent complete statements, such as controlling the program flow with return and break, evaluating various expressions, invoking functions, or assigning a value to a variable. Instructions are the highest level of steps, usually encapsulating various keywords and expressions. By default, when the substeps toggle is turned off, these are the only steps presented to the player, as they represent the execution moments where full instructions are completed;

Expressions - Computations made with variables, math operations (`a + b`), logical checks (`c && d`) and even function calls. At the end of execution of an expression, a value is returned (string, number, boolean, null, etc.) that can then be used in instructions, in conjunction with other expressions and operations;

Keywords - Reserved words that define specific program behavior (e.g. `var`, `let`, `if`, `for`, `while`, `function`, etc.). Before proceeding to expressions and instructions, the interpreter must first handle the keywords in them, so that they are ready to be used in the upcoming operations.

Figure 4.6: Play Substeps switch

Substeps toggle

In case the player wants to go through every single step in the execution of the written program, beyond full instructions, there is a substeps toggle that can be enabled to display this functionality. If the toggle is on, then every execution step returned by the interpreter (instructions, expressions and keywords) back to the game is as well presented to the player in the level. This toggle can be turned on and off even if the game is already running steps, so the player has the flexibility to check in more detail all the execution steps, whenever the player wants.

Toggle State	Behavior	Output Example
Off (Default)	Displays only the highest level steps, full statements (Instructions).	`let x = y + z;`
On	Displays all steps, including keywords, expressions, and instructions.	`let`, `x`, `=`, `y + z`

Table 4.1: Behavior of the Substeps Toggle

4.4.2 Advancing

It is possible to advance the state of the game to the next steps of execution. When the player presses the buttons to advance to next steps, the game starts by checking if the current step is the most recent one, or if there are other posterior steps that were already traversed by the player. If the current step is the most recent one, it means the JavaScript interpreter hasn't interpreted the next step yet, so the game requests the step to the interpreter and the value of the step is returned back to the game. All the substeps involved in this new step are then added into the completed steps list and the step changes are applied in the game. In case the current step is not the most recent one, then the next one was already interpreted, so the game just obtains the next step data from the completed steps list and applies the changes to the state, without having to interact with the interpreter. When the game is in the last step of execution, the advancing buttons are locked because there is nothing to advance beyond the last step.

Advance one step

The advance one step button has the icon of an arrow pointing to the right. When clicked, the game state progresses one step forward.

Advance all steps

The advance all steps button has the icon of 2 arrows pointing to the right. When clicked, the game state starts progressing steps forward automatically, until there are no more steps (either by level completed, error, or no more code execution steps remaining) or if the execution is paused manually by the player.

4.4.3 Rewinding

The state of the game can be rewinded by the player to previous steps of execution. When the state is rewinded, all the game elements including game objects, UI, spaceship, and projectiles are all reverted back to the previous state they had on that step. When the game is in the first step of execution, the rewinding buttons are locked because there is nothing to rewind beyond the first step.

Rewinding one step

The rewind one step button has the icon of an arrow pointing to the left. When clicked, the game state rewinds one step.

Rewinding all steps

The rewind all steps button has the icon of two arrows pointing to the left. When clicked, the game state starts rewinding steps automatically, till the game has reached the first step of execution again and everything is back to where it was at the start of the level. The automatic rewinding can also be paused manually by the player.

4.4.4 Pausing Execution

While advancing or rewinding all the steps in the execution flow, the player can decide to pause the execution at the current step, by pressing the pause execution button in the step buttons section.

The pause execution button can only be pressed if the game is currently advancing or rewinding a step. For example, if the player presses the rewind all steps button, the game starts rewinding, and by doing so, the pause execution button will become pressable.

After clicking the button, the game will first block the other buttons in the step buttons section, so that the player cannot rewind or advance again while the pause sequence is activated. Secondly, the game will finish making all the changes of the step it is moving towards. Then, the game pauses the execution on the step it just reached. At the end, the other buttons in the step buttons section become pressable again.

4.4.5 Step by step code highlighting

On each execution step, the executing instruction is highlighted in the player code inside the code editor. Every time the state is advanced or rewinded, the highlighted code also changes to the executing instruction according to that step. This feature allows the player to understand more easily what each JavaScript instruction does step by step and how the written code affects the game elements.

4.5 Game Console

The Game Console is a UI element that presents to the player parts of the program state about things such as collected objects, actions, score, emitted signals, pressed buttons, etc. Console logs

Action	Description	Notes
Advance One Step	Progress the game state one step forward.	Only works if not at the last step.
Advance All Steps	Automatically advance steps.	Stops at level completion, an error, or when manually paused.
Rewind One Step	Rewind the game state one step backward.	Only works if not at the first step.
Rewind All Steps	Automatically rewind steps.	Stops at the first step or when manually paused.
Pause Execution	Pause the current advancing or rewinding process.	Available only while advancing or rewinding steps. Pauses on the step currently being reached.

Table 4.2: Step Control Buttons Overview

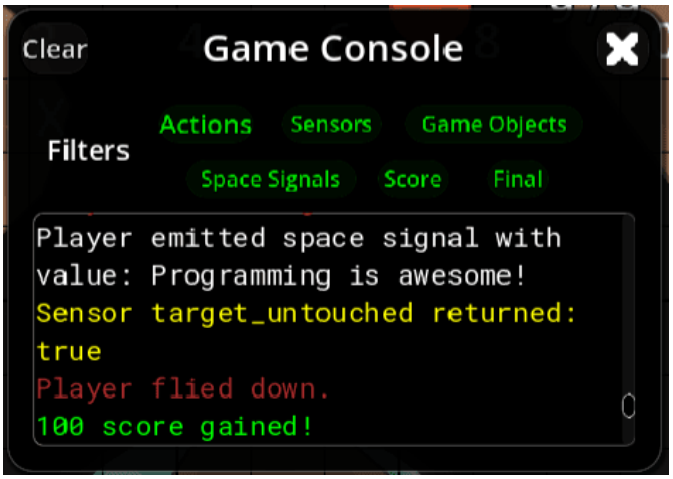

Figure 4.7: Game Console with various types of logs

are divided into different types:

Action - Player actions (*e.g.* `fly`, `turn`, `open`, `press`);

Sensor - Activated player sensors (*e.g.* `isCollected`, `obstacleIsForward`, `getHealth`);

Game Object - Game object changes (*e.g.* Button pressed, gate opened, coin collected);

Space Signal - Emitted space signals;

Score - Amount of score gained, whenever the player gains score;

Error - Errors returned by the interpreter;

Final - Final log, which indicates if the mission was completed successfully or not.

In order for the player to only see the logs that most interest them, it is possible to filter the console logs by type, when enabling or disabling the filters, respectively, to each type on their buttons. The filter buttons are located on top of the logs display of the console.

Figure 4.8: Playground console with some code already executed

In the top left corner of the console, there is a "Clear" button that, when pressed, clears all the logs from the game console.

4.6 Playground Console

Inside the game level, there is a playground console with a completely separate JavaScript interpreter running on it. The player can write small pieces of code on the playground console to test the newly learned syntax and logic, without affecting the real game in any way.

The code can be written in the input text zone below the logs display, and after it is written, the player can click the "Run" button on the right side of the input zone to submit the code into the browser's interpreter instance responsible for interpreting the playground console JavaScript code.

Various code statements can be submitted by the player one after the other, and the program state will keep memory of all the variables and logic written on all the code statements, through the entire execution. However, in case an error is thrown by the interpreter, the error log is displayed on the console, the execution environment is restarted, and the memory of the interpreter is reset.

The same as for the game console, in the top left corner of the playground console, there is a "Clear" button that, when pressed, clears all the logs from the playground console.

4.7 Game Objects

CodinSpace has a wide range of game objects, each one with its own purpose in teaching programming concepts. When combined, these game objects create many different types of interesting programming challenges.

In the obstacles there is the asteroid, gate, laser, and laser spawn. Whenever the spaceship collides with an obstacle, it takes damage, so it is very important to avoid them. When opened, the gate deals no damage to the spaceship. It is possible to open it using keycards, pressing buttons, or sending signals. The game logic revolves a lot around knowing how to write JavaScript code that

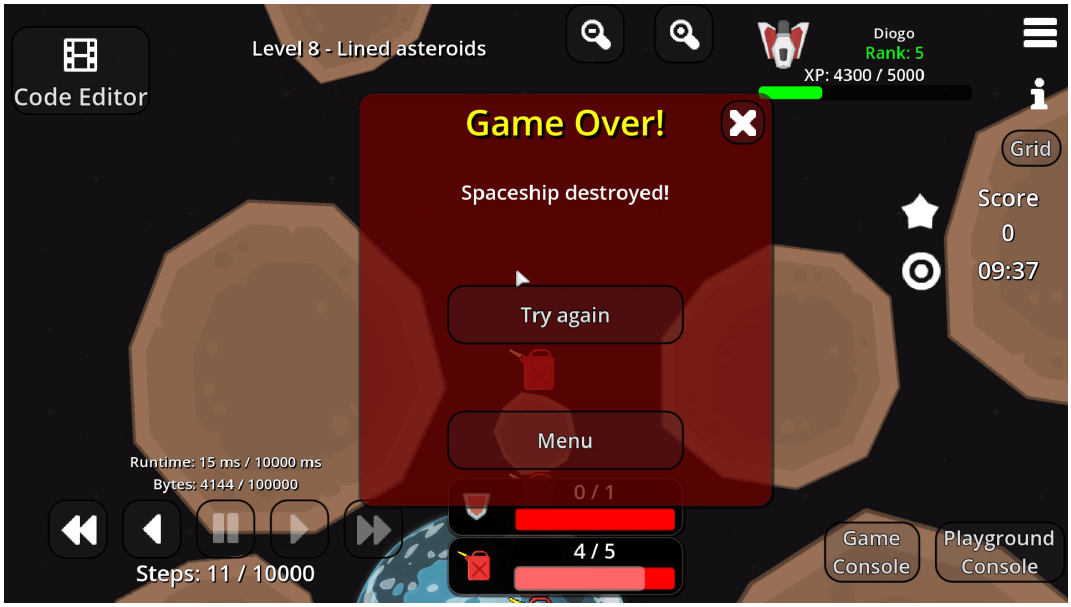

Figure 4.9: Game Over pop-up after the player collided with an obstacle

can avoid all the obstacles (or most of them), and many programming concepts are taught using them (conditionals, loops, functions).

On the other hand, the collectibles are the jerrycan, keycard, repair kit, and coin. The jerrycan replenishes the spaceship fuel tank, the keycard unlocks gates or crates, the repair kit restores the spaceship shield points, and the coin gives score to the player. The game goal of these objects is to force the player to write more complex code to make turns and detours to other locations of the level, in order to keep the spaceship working and also to gain extra score.

The interactives are the crate and the button. The crate can be opened by the player with the `open()` action, and will return a value that can be stored in a variable. The value can then be emitted in a signal to unlock gates that require it. The button can be pressed by the player with the `press()` action, in order to open gates and unlock certain crates. By having interactives in the game, it is possible to teach the player programming concepts such as variables, conditionals, and the use of arrays.

The target is the main object of each level. The player must overcome all the challenges of the level in order to reach the target. Sometimes, as with the gate or the crate, the target may require certain things to be done first, such as pressing specific buttons, collecting keycards, or emitting signals.

4.7.1 Obstacles

Obstacles are game objects that cause difficulties for the player, making them take other precautions to overcome the obstacle.

Asteroid

The player will often come across asteroids that will obstruct the path to the objective. If the player collides with an asteroid, depending on its size, the spaceship can take some damage, or be totally destroyed. The asteroid is also destroyed on impact with the spaceship.

Figure 4.10: Asteroid

Figure 4.11: Gate

To avoid asteroids, the player must find the open spaces between them and use the sensors to detect if the spaceship is on the direction of any asteroid, therefore, using conditionals to decide what direction the ship should take. It can often happen that the player must repeat the use of sensors for a certain number of steps, thus incentivizing the use of `while` loops.

The properties that define the asteroid are the id, 2d position, color, radius, and damage.

Gate

The gate is a type of obstacle that blocks the player access to other sections of the level. In order to open them, the player must fulfill a list of requirements. These requirements may involve pressing specific buttons, collecting certain keycards, or emitting specific signals.

The purpose of the gate is to force the player to check if a certain number of conditions is fulfilled, using the `if` and `else` statements.

The properties that define the gate are the id, 2d position, color, a flag to know if it is open, a flag to know if it must be opened manually or can be opened with button or signal, a flag indicating if it is presented horizontally or vertically, and a damage value that will be inflicted on the player when colliding. The requirements are an array of required game object ids, each one with the required property name and for that property the required value.

Laser

The laser, same as with the asteroid is an obstacle that obstructs the player's path to the objective. While designing the levels it was decided that sometimes lasers positioned in straight lines can show desirable paths more easily instead of only having asteroids obstructing the way. This clear definition of the path that the spaceship should take makes it more intuitive and perceptible to understand what programming concepts should be used to successfully beat the level.

The properties that define the laser are the id, 2d position, color, direction, and damage.

Laser Spawn

The laser spawn is an obstacle that for now only serves the aesthetic purpose of being the emitter of lasers. When the player collides with a laser spawn, the spaceship will take damage, but the laser

Figure 4.12: Laser

Figure 4.13: Laser Spawn

spawn does not get destroyed.

The properties that define the laser spawn are the id, 2d position, color, direction, and damage.

4.7.2 Collectibles

Collectibles are game objects that the player can collect and that benefit the player in some way (unlock a certain part of the level, increases the score, recovers health, etc.).

Jerrycan

The jerrycan is a collectible that replenishes the player's spaceship fuel tank by a certain amount of points. Each jerrycan can have its own number of fuel points, so the player may have to pay attention to which jerrycan to pickup, so that the spaceship always has enough power.

The properties that define the jerrycan are the id, 2d position, color, fuel quantity, score gained on collection, and a flag indicating if it is collected or not.

Keycard

The keycard is a type of collectible that can be used to unlock specific crates or gates. The existence of keycards adds another challenge to the game, because instead of having the player write code just to reach the target directly, they must first collect certain keycards, which will unlock sections of the level that lead to the target.

The properties that define the keycard are the id, 2d position, color, score gained on collection, and a flag indicating whether it is collected or not.

Figure 4.14: Jerrycan

Figure 4.15: Keycard

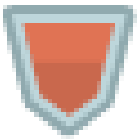

Figure 4.16: Repair Kit

Repair Kit

Repair kits are collectibles that recover the player's spaceship shield points. The player will often encounter many obstacles that are hard to avoid and can damage the spaceship. To make sure that the mission is not failed, it is important that the player collects the repair kits whenever the spaceship is damaged. This concern for repairing the spaceship motivates the player to check the sensors frequently for repair kits nearby and the current status of the spaceship's health.

The properties that define the repair kit are the id, 2d-position, color, shield points gained on collection, score gained on collection, and a flag indicating if it is collected or not.

4.7.3 Interactives

Interactives are game objects that the player can interact with, in order to unlock a new area or complete a puzzle.

Crate

Crates are interactives that can be opened by the player and may contain important messages that the player can emit to open gates and unlock new areas. Similarly as with the gate, the crate can have requirements that must be fulfilled to be opened.

The main purpose of the crate is to make the player use variables for storing data and later using those data on a different part of the level, with function calls.

The properties that define the crate are the id, 2d-position, color, score gained on opening, a flag indicating if it is opened or not, the contained value, a flag indicating if it can be manually opened or must be with a button or signal, and lastly the requirements to open. The requirements are defined in the same way as with the gate, using the array of required game objects.

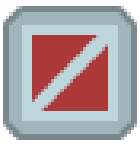

Figure 4.17: Crate

Figure 4.18: Button

Figure 4.19: Target

Button

Buttons can be pressed by the player to perform various actions and changes on the level. They can be bound as requirements of other game objects, such as gates or crates, which may require a series of buttons to be turned on/off in order for them to be unlocked.

The properties that define the button are the id, 2d-position, color, score gained on pressing and a flag indicating if it is pressed or not.

4.7.4 Target

Reaching the target is the player's main objective. In order to complete a level, the player must go through all the obstacles and challenges with the objective of reaching the target point, which is normally located in a difficult access area. There can be only one target per level.

The properties that define the target are the 2d position, color, and requirements. The requirements are defined in the same way as with the gate and the crate, using the array of required game objects.

4.8 Tooltips

Tooltips are small pop-ups that appear whenever the player hovers the mouse over a game object (keycard, spaceship, asteroid, target). Each tooltip gives useful information about that object, for example, when a gate is hovered, the appearing popup will indicate the id of the gate, the requirements, and if it is opened or not.

Some properties presented in the tooltip are common to most game elements, for example, the id is a property that identifies a game object, so every time a game object is hovered, the id will appear. On the other hand, there are properties that can be very specific to one type of game element, for example, when we hover a jerrycan, the points of fuel that the jerrycan replenishes appear as a property in the tooltip and when we hover a crate or a gate, the requirements to open appear on the tooltip (buttons that need to be pressed, keycards that need to be collected, signals that need to be emitted).

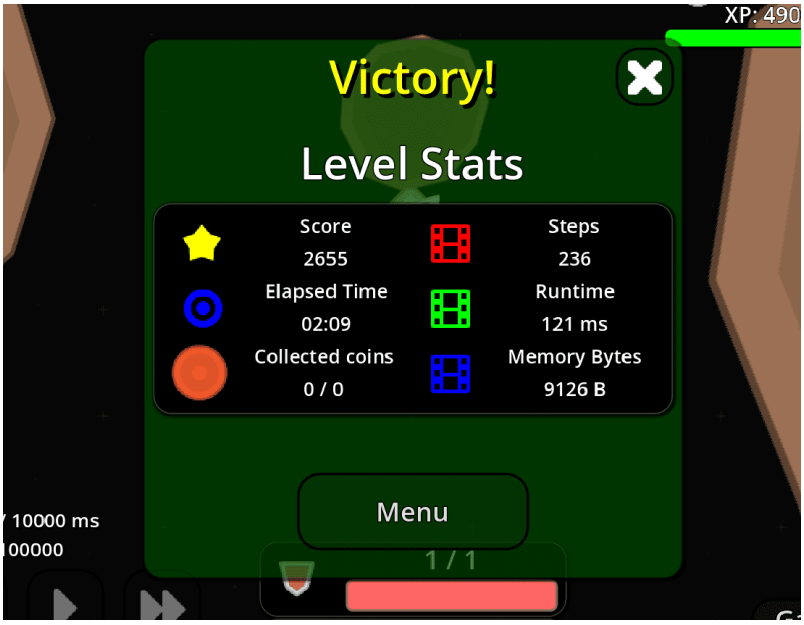

Figure 4.20: Victory pop-up that appears after the player reaches the target

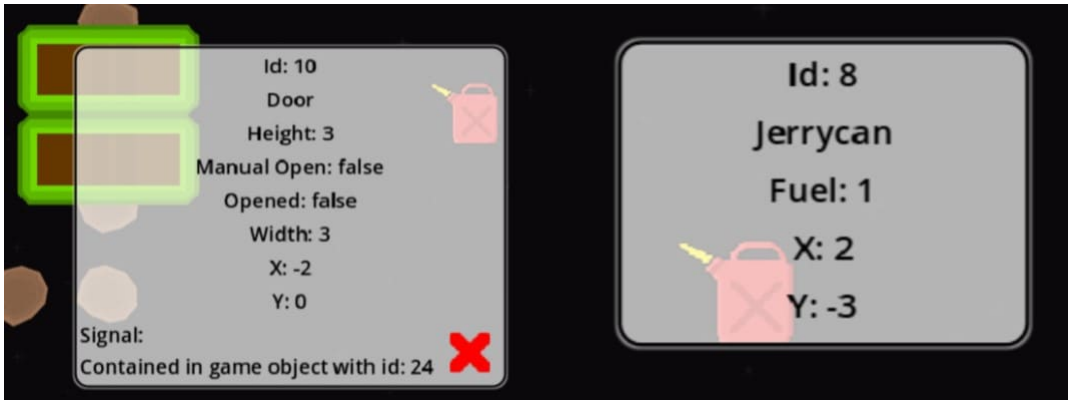

Figure 4.21: Tooltips displaying information about a hovered gate and jerrycan

Table 4.3: Game Objects and Their Functionalities

Game Object	Category	Functionality
Asteroid	Obstacle	Obstructs the path to the objective. On collision it gets destroyed and damages the player.
Gate	Obstacle	Blocks access to other sections of the level. Can be opened by fulfilling requirements such as pressing buttons or collecting keycards.
Laser	Obstacle	Obstructs the path to the objective. On collision damages the player, but is indestructible.
Laser Spawn	Obstacle	Aesthetic laser emitter that damages the player upon collision but is indestructible.
Jerrycan	Collectible	Replenishes the spaceship's fuel tank by a specific amount of fuel points.
Keycard	Collectible	Used to unlock gates or crates, enabling access to new areas.
Repair Kit	Collectible	Restores spaceship shield points, helping the player survive collisions with obstacles.
Crate	Interactives	Can be opened to retrieve important data or messages required to unlock gates or areas.
Button	Interactives	Pressing it activates specific actions such as unlocking gates or crates.
Target	Target	The main objective of the level. The player must overcome challenges and meet requirements to reach it.

4.8.1 Sensor Information Pop-up

The sensor information pop-up is an information pop-up that is displayed in the game's UI whenever a sensor is activated. The pop-up tells the player the name of the sensor that was activated and the value it returned.

4.9 Info Section

The info section is a small display that gives information to the player about specific programming topics or tips to complete the current level. This is a very important part of the game, because it displays a summary of the key programming concepts introduced in the current level.

While playing a level, the player can open the info section by pressing the button on the right side of the screen and close it by pressing the cross button inside the section.

All the information inside the info section is written in Markdown, allowing the use of headings, lists, links, and much more. Godot does not provide any node that allows the use of Markdown, so the game uses the MarkdownLabel [Dae]. This plugin includes a custom Godot node that extends the `RichTextLabel` from Godot to use Markdown instead of BBCode (Bulletin Board Code).

Figure 4.22: Info section

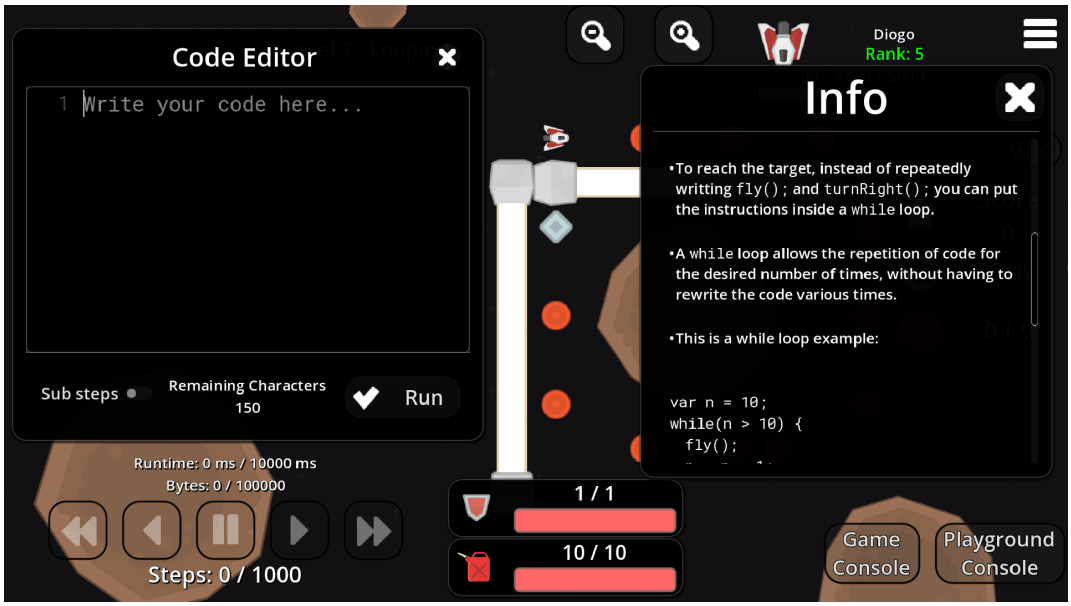

Figure 4.23: Player game view with the code editor and info section opened

Figure 4.24: Main Menu

4.10 Menus

4.10.1 Main Menu

When the player opens *CodinSpace*, the game starts off by displaying the main menu. In this menu, the player has four buttons, play, editor, profile, settings, and credits. By clicking on the play button, the game opens the menu to select the campaign or community chapter and levels. By clicking on the editor button, the main menu for the chapter and level editors opens. On the other hand, the profile button opens the player profile and the settings button opens the game settings. Finally, the credits button opens the game credits.

4.10.2 Profile

The player profile displays information about the player name, earned experience, current rank, and selected spaceship. The name and the spaceship type and color can be changed at any time by the player when opening the profile. This small customization gives the player a bit more choice on how they should appear inside the level, thus making them feel more connected with the game which now has something customized to their liking.

4.10.3 Settings

Allows the player to adjust the sound volume of the game with 3 different sliders, the master volume slider, the music volume slider, and the sound effects (SFX) volume slider, respectively.

4.10.4 Credits

The credits identify the game developer Diogo Pinto, the thesis advisor Professor Francisco Coelho and contains a list of asset credits, with information about all the used assets (asset name, owner and website link).

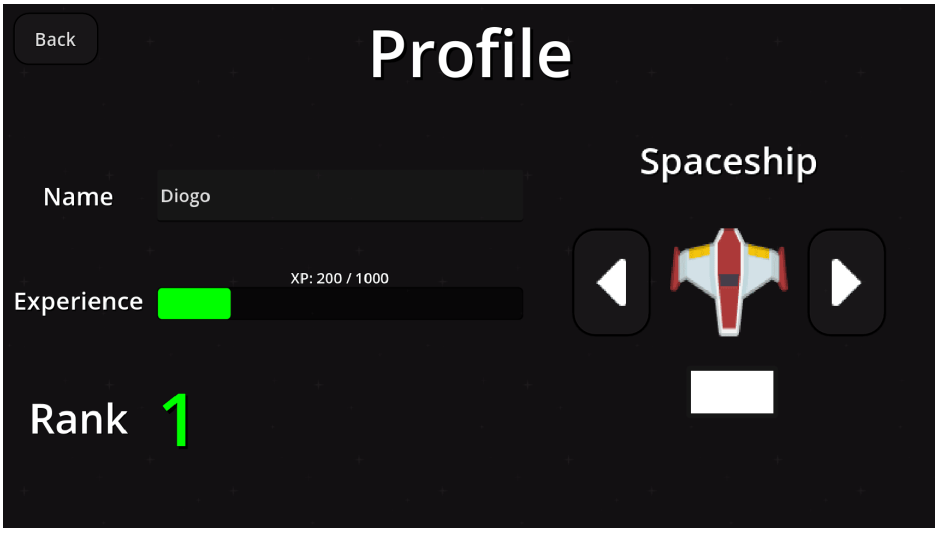

Figure 4.25: Player profile

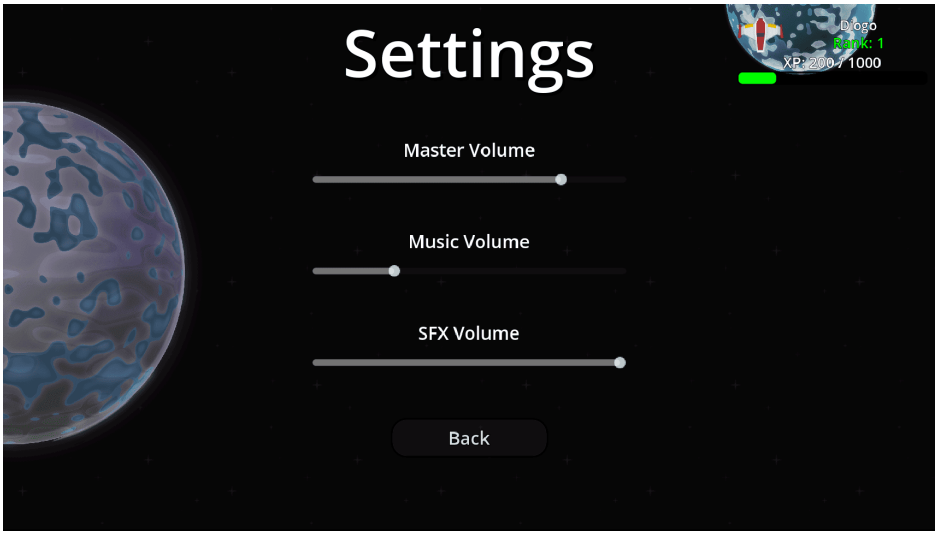

Figure 4.26: Settings

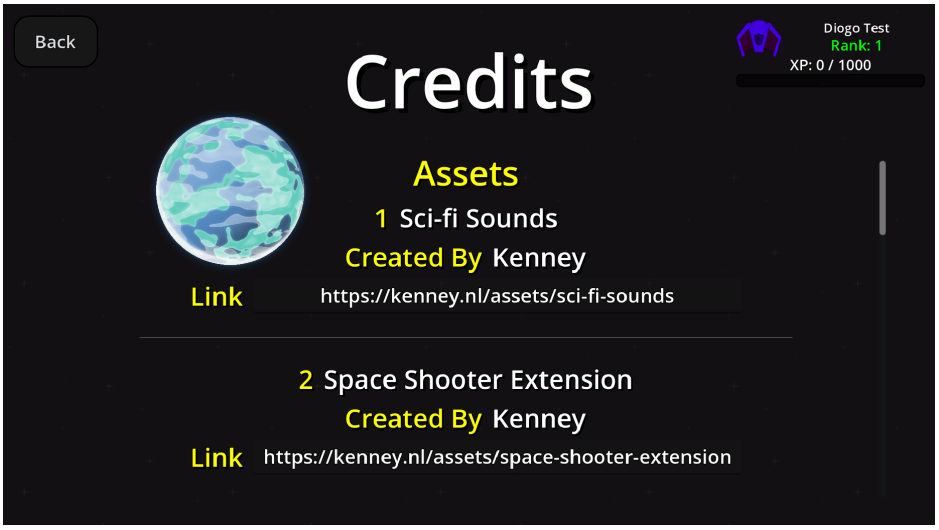

Figure 4.27: Credits

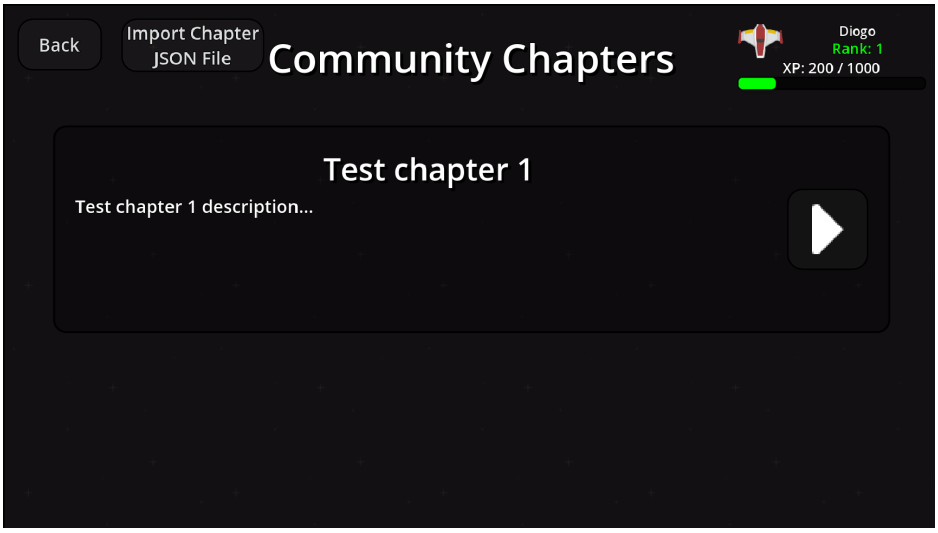

Figure 4.28: Community Chapters Selection

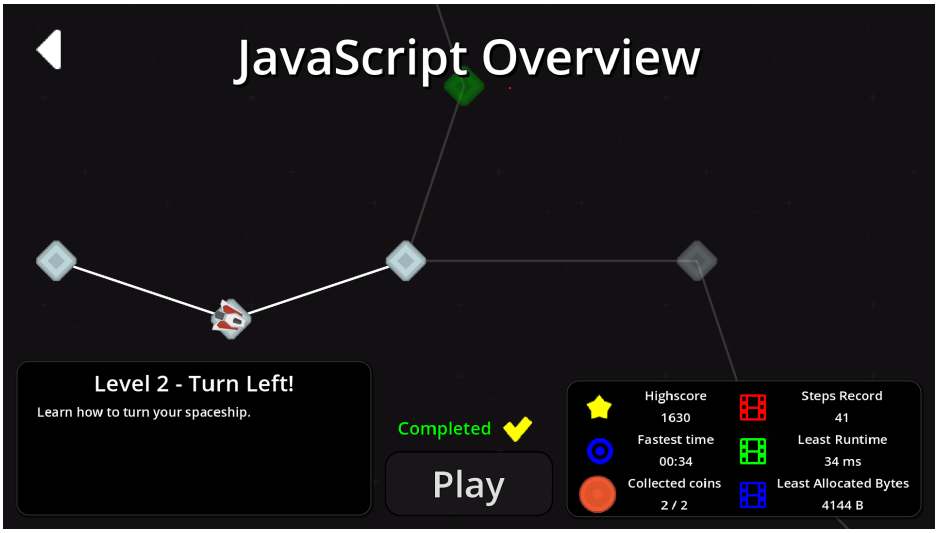

Figure 4.29: Level Selection

4.10.5 Chapter Selection

The chapter selection menu is where the player can choose the game chapter. It starts by presenting the list of existing chapters and the player can see their name and description. There is a "Back" button to return to the main menu. By selecting a chapter, the level selection menu will open, with all the levels contained in that chapter.

4.10.6 Level Selection

Displays every level within a selected chapter, using a graph structure in which each node represents a level. The player can select a level by traveling with the spaceship across the graph until the node is reached. However, it is only possible to travel to the nodes that have been unlocked. Nodes that are locked have gray colored paths connecting to them. On the other hand, nodes that are unlocked, therefore playable, have white color paths connecting to them.

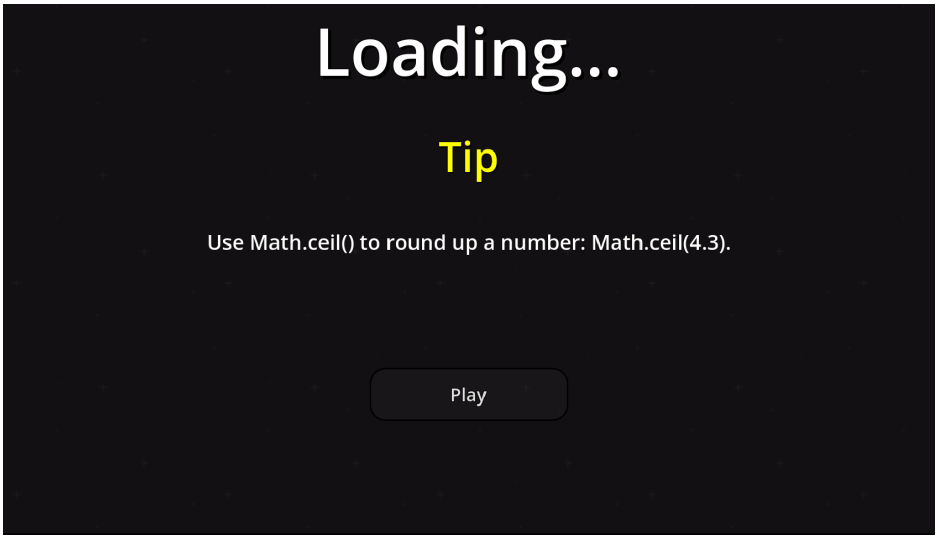

Figure 4.30: Loading Screen

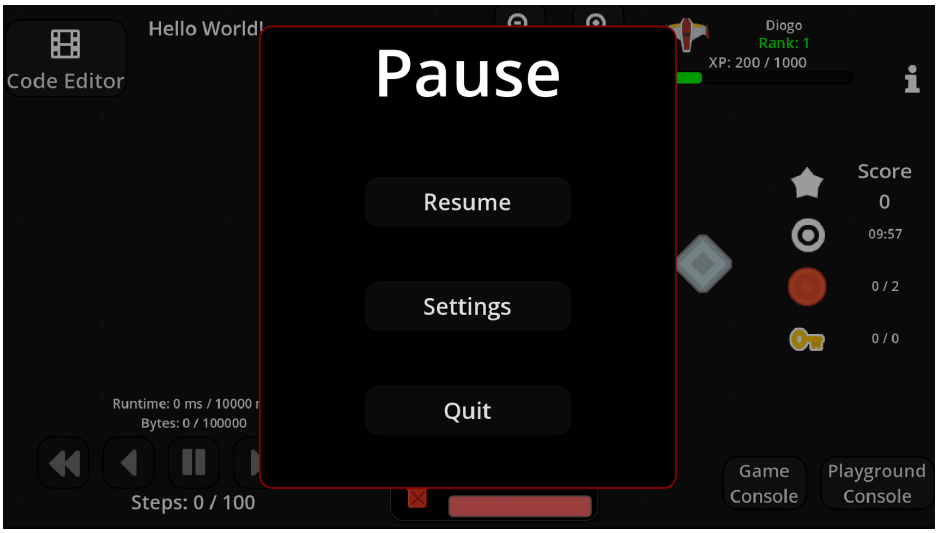

Figure 4.31: Pause Menu

4.10.7 Loading Screen

Small section that serves as a transition between the menus and the level that the player is about to start. It presents useful programming tips to the player that are chosen randomly every time the loading screen appears.

4.10.8 Pause Menu

By pressing the pause button in a level, the game will pause and display the pause menu. This menu has three buttons, respectively, the "Continue" button, the "Settings" button, and the "Quit" button. By pressing the "Continue" button, the game will be resumed. By pressing the "Settings" button, the settings menu will appear. By pressing the "Quit" button, a confirmation modal will appear, where the player can press the "Yes" button to quit to level, or the "No" button to go back to the pause menu.

Figure 4.32: Chapter Editor Main Menu

Figure 4.33: Level Editor Main Menu

4.10.9 Chapter Editor Main Menu

Menu that lists to the player, in a table, every chapter that was created on the chapter editor. Each row on the chapters table has the name of the chapter, an "Edit" button that when clicked opens the chapter editor for that specific chapter, and has a "Delete" button that deletes the chapter. There is also a "New Chapter" button, which can be used to create a new chapter on the chapter editor.

4.10.10 Level Editor Main Menu

Displays to the player, in a table, each level made using the level editor. Each row contains the name of the level, an "Edit" button to open the editor for that particular level, and a "Delete" button to remove the level. There is also a "New Level" button for creating a new level in the editor.

4.11 Level Constraints

Every level in *CodinSpace* has various constraints that the player must always fulfill, up to the end of the level. This adds another set of challenges to the game, because the player must write code that does not only take into account the game objects around the spaceship, but also the constraints that must stay fulfilled. In case any of these limits are reached, the game automatically ends with a game over message explaining that the player failed because one constraint was broken.

4.11.1 Game Constraints

Constraints that are set directly on the elements of the game.

- **Maximum Health** - Maximum health that the player spaceship is allowed to have. The player starts with this health value, and each time a repair is collected, the health can never go above the value of this constraint;
- **Maximum Fuel** - Maximum fuel that the player spaceship fuel deposit can have. The fuel deposit of the spaceship starts with this fuel value, and every time a jerrycan is collected, the fuel can never go above the value of this constraint;
- **Maximum time to complete level** - Time allowed for the player to complete the level. When the player starts the level, a timer in the UI starts counting down from this value, and if it reaches 0, the player loses the level because the allowed time for that level is over;
- **Maximum number of characters allowed** - Maximum number of characters that can be written in the code editor, for the current level. Once the player reaches the limit of this constraint, they cannot write more characters unless some code is erased. This constraint forces the player to write code that is more compact and less repetitive, which helps fulfill the pedagogical objective of using loops and functions to reuse code.

4.11.2 JavaScript Interpreter Constraints

Constraints that are defined within the JavaScript interpreter.

- **Maximum Interpreter Steps** - Limits the number of steps that the interpreter can go through. This forces the player to write code that runs in as few steps as possible;
- **Maximum Interpreter Runtime** - Maximum runtime allowed for execution of the player code. In the game interpreter, this time counter is incremented at every step, but on the playground console interpreter, the time counter is reset every time the player submits new code. This constraint is not only good to make the player write code that uses up less runtime, but also prevents the occurrence of infinitely running programs because the interpreter stops advancing steps once the constraint is reached;
- **Maximum Interpreter Memory Bytes Allocation** - Limits the number of memory bytes that can be allocated inside the interpreter. Used not only to force the player to write code that allocates less memory, but also to prevents memory bombs (programs that infinitely allocate memory till the machine freezes completely).

4.12 Singletons

In programming, singleton is a design pattern, which consists of having a class with only one instance that is globally accessible through the entire application (a singleton). In Godot, a singleton [Godc] is a node that is instanced once at the start of the game (is set by the developer to be auto-loaded) and is used to declare global functions or store global variables that need to persist between scenes and that are frequently accessed in different components of the game (used for example in: player data management, save/load system, scene transitioning, game state management, etc.).

CodinSpace uses several singleton scripts, each responsible for managing a specific part of the game's logic. One key use case is game state management, as the player advances or rewinds the execution steps, the game state is constantly changing. All game components (such as game objects, the UI, the spaceship, etc.) must be updated to reflect these changes. In addition, certain behaviors must be repeated to preserve a consistent gameplay experience (*e.g.* spaceship and object animations should replay, console logs should be added or removed, and sensor pop-ups should appear or disappear depending on whether the execution is being advanced or rewinded).

To support this dynamic behavior, *CodinSpace* centralizes game state data in singletons (*e.g.* player state, console state, steps state). This makes it easy for different parts of the game to access current state information without relying heavily on Godot signals to propagate updates. As a result, state transitions are smoother and more reliable, since the necessary data is always available through the state singletons.

Singletons are also used in *CodinSpace* for other global functionalities (*e.g.* common UI logic, sound effects, save/load operations, string formatting) and data (*e.g.* volume settings, player progress, spaceship customization, etc.) that need to be accessed across multiple scenes.

4.13 Strings Organization

During development, to improve readability and simplicity of the code, I made the decision to separate all strings from the code, into their own separate singleton `Strings.gd`. It contains every single string that is used in *CodinSpace*, stored in constants.

The code is folded in regions, which is a new feature introduced in Godot 4.0. They are used in this singleton to better organize the numerous strings and to easily remember which part of the game each one belongs to.

If the strings were scattered across the game files, the developer would have to go through every file and search for that specific string, which would be much more time consuming and confusing. By having all the strings declared in only one location as constants, it becomes much easier to make new changes to the game, because the developer only needs to change the string value associated to the constant, which remains the same through all game files.

I decided to explain this decision before proceeding to the more technical chapters of the dissertation, so that it becomes clear to the reader the reason why the code is structured the way it is and why the strings are not directly present in the code.

```
#region Resource Paths
...
#endregion

#region State Sections
const SECTION_NAME := "sectionName"
const SUBSECTION_NAME := "subsectionName"
const SUBSECTION := "subsection"
...
#endregion

...

#region Actions
const ACTIONS := "actions"
const ACTION := "action"
const FLY := "fly"
...
const EMIT := "emit"
#endregion

#region JavaScript Interpreter
const JS_GAME_INTERPRETER_INITIALIZATION := "gameInterpreter = new Interpreter
    (\"\", initFunc);"
const JS_CONSOLE_INTERPRETER_INITIALIZATION := "consoleInterpreter = new
    Interpreter(\"\");"
const JS_GAME_INTERPRETER_APPEND_CODE := "gameInterpreter.appendCode(\"{code}\");"
...
#endregion
...
```

Figure 4.34: String constants from the Strings Singleton

```
1  func advance_step() -> void:
2    var step := await StepsStateManager.fetch_next_step_data()
3
4    update_game_state(step)
5    update_global_state()
6
7    if StepsStateManager.check_current_step_is_last():
8      if StepsStateManager.get_objective_achieved():
9        victory.emit()
10     else:
11       game_over.emit()
```

Figure 4.35: `advance_step()` function from the `GlobalStateManager.gd` singleton

4.14 Advancing and Rewinding the Game State

As presented in the chapter on the execution flow of the player code, the game allows the player to advance, rewind, and pause the game state, which is a core feature and crucial to the self-learning aspect of the game. Now we will delve into more detail and look at how these functionalities really work and how the game manages to keep the state always updated, using the state management singletons previously defined.

4.14.1 Advancing

It is possible to advance the state by one step, by pressing the advance one step button in the level UI. By doing so, the function `advance_one_step()` will be called, which will set the `playing_one_step` flag to `true`, disable all the state manipulation buttons except pause and start a one-second cooldown timer that does not allow any other step to be made in that time period (to not break the game, by making it too fast). Afterwards, the `advance_step()` function from the Global State Manager is called and it is awaited till the advance operation is completed.

The `advance_step()` function starts by requesting the next step data, by calling the `fetch_next_step_data()` function from the Steps State Manager and then storing the result in a step variable, which will be used afterwards to update the game state.

Inside the `fetch_next_step_data()` there is a loop that repeatedly keeps getting the next steps of execution, till the current step is an end of statement (not a substep), then returns the step data. When the player is currently at the most recent step, the function `step_forward()` from the JavaScript Execution Handler singleton is called, which will request to the game sandbox interpreter instance the next step and process the received data. Once the JavaScript Execution Handler singleton has finished the operation and stored the step data in the completed steps array, the `fetch_next_step_data()` function just needs to get the step data from the dictionary. In case the player advances to a step that has already been interpreted before (the player rewinded to a previous step before and now is advancing to a step that has already been processed by the interpreter), the function just needs to get the step data from the completed steps array.

As stated in a previous chapter, if the player has the fetch sub-steps option enabled in the code editor, the game will be displaying every single step of execution, not only the steps at the end of statements. In this case, if the `fetching_interpreter_sub_steps` flag is `true`, this function will

```
func fetch_next_step_data() -> Dictionary:
  var current_step_is_sub_step := true
  var current_step = null
  while current_step_is_sub_step:
    if check_current_step_is_most_recent():
      await JavascriptExecutionHandler.step_forward_game_interpreter()
    current_step = get_current_completed_step()
    current_step_is_sub_step = current_step[Strings.SUB_STEP]
    increment_current_step_id()
    if get_fetching_interpreter_sub_steps() or current_step[Strings.
        EXECUTION_TERMINATED]:
      return current_step
  return current_step
```

Figure 4.36: `fetch_next_step_data()` function from the `StepsStateManager.gd` singleton

not keep fetching next steps until the end of the statement, but instead will only fetch one step and immediately return its value.

After the step data is returned back to the `advance_step()` function, the game will use those data to update the state (actions, sensors, code to be highlighted, errors, etc.). Finally, the `playing_one_step` flag is switched back to `false` and the advance operation is completed.

In case the player presses the advance all steps button instead, the game will set the `playing_all_steps` flag to `true` and repeatedly call the `advance_one_step()` function from the buttons UI every second, until the final step is reached or the player presses the pause button.

4.14.2 Rewinding

By pressing the rewind one step button in the level UI, the player begins the operation of rewinding the state by one step, which begins with the `rewind_one_step()` function. This function works similarly to the `advance_one_step()` function, being the only differences that the `rewinding` flag is set to `true` alongside the `playing_one_step` flag and that the function `rewind_step()` from the Global State Manager is called.

Next, the `rewind_step()` function requests the previous step data, calling the `fetch_previous_step_data()` function from the Steps State Manager, and then storing the result in the step variable.

The `fetch_previous_step_data()` function works very similarly to the `fetch_next_step_data()` function as well, having a `while` loop that keeps decrementing the current step id and getting the previous steps of execution from the completed steps array until it reaches an end of the statement step or it reaches the initial step.

When the `fetching_interpreter_sub_steps` flag is `true`, the function will immediately return the first step that it accesses, even if it is not the end of a statement.

As with the advance all steps button, when the rewind all steps button is pressed, the game sets the

```
1  func fetch_previous_step_data() -> Dictionary:
2    var current_step_is_sub_step := true
3    var current_step = null
4    while get_current_step_id() > 0 and current_step_is_sub_step:
5      decrement_current_step_id()
6      current_step = get_current_completed_step()
7      current_step_is_sub_step = current_step[Strings.SUB_STEP]
8      if get_fetching_interpreter_sub_steps() == true:
9        return current_step
10   return current_step
```

Figure 4.37: `fetch_previous_step_data()` function from the `StepsStateManager.gd` singleton

`playing_all_steps` flag to `true` and periodically calls the `rewind_one_step()` function every second, until the initial step is reached or the pause button is pressed.

4.14.3 Updating the Game State

After advancing or rewinding to a different step of execution, the game state must be updated, so that every change contained in that step is reflected in the game. The state can be updated by calling the function `update_game_state()` from the Global State Manager and passing the step data as a parameter.

First of all, the function starts by updating the highlighted code range dictionary, the used game interpreter memory bytes, and the game interpreter runtime, with the values contained within the step data dictionary. Next, every space signal that was activated previously is disabled because space signals must only remain active in the step they were created.

Subsequently, the function checks if the step dictionary "`content`" entry is not `null`, which may in certain steps, contain a dictionary with data about an action or sensor that was triggered by the player, in the code. If the content is a sensor, the function emits a signal which tells the game to present a pop-up in the player UI, with information about the activated sensor (name and value of the sensor). If the content is an action, the function applies the action to the game with the `do_step_action()` function from the Actions singleton and calls the `update_game_objects()` function from the Game Objects State Manager, so that every game object that was influenced by the action, is updated to its new state (*e.g.* player doing the "`press`" action nearby a button, may make the button state change to `pressed`).

After the state sub-dictionaries on each state management singleton are updated, the `update_global_state()` function in the Global State Manager is called, which will fetch the updated dictionaries from each state management singleton and update the main state dictionary in the Global State Manager. This dictionary is then used to send an updated copy of the entire state to the code interpreter in the browser, which will be used to make the player sensor and action functions work properly, with updated information about the current state of the game.

Finally, after having the game state updated with the current step, the game always checks if is currently in a state where the mission can be terminated, which can happen by having the player reach the target or by failing the level. If the mission is terminated, then the game will not allow the player to request next steps on the game interpreter, and instead only allows to rewind and advance to steps that have been interpreted before and are in the completed steps array.

```
1  func update_game_state(step: Dictionary) -> void:
2    var step_content = step[Strings.CONTENT]
3
4    var rewinding := StepsStateManager.get_rewinding()
5
6    if StepsStateManager.get_current_step_id() == 0:
7      set_highlighted_code_range(null)
8      set_game_interpreter_memory_bytes(0)
9      set_game_interpreter_runtime(0)
10   else:
11     set_highlighted_code_range(step[Strings.HIGHLIGHTED_CODE_RANGE])
12     set_game_interpreter_memory_bytes(step[Strings.USED_MEMORY_BYTES])
13     set_game_interpreter_runtime(step[Strings.RUNTIME])
14
15   if rewinding:
16     check_game_finished(step)
17
18   SpaceSignalsStateManager.disable_all_space_signals()
19   if step_content != null:
20     var content_type: String = step_content[Strings.TYPE]
21     if content_type == Strings.SENSOR:
22       Sensors.sensor_activated.emit(step_content[Strings.NAME], step_content[
           Strings.VALUE])
23     elif content_type == Strings.ACTION:
24       Sensors.sensor_disabled.emit()
25       if rewinding:
26         GameObjectsStateManager.update_game_objects()
27         Actions.do_step_action(step_content)
28       else:
29         Actions.do_step_action(step_content)
30         GameObjectsStateManager.update_game_objects()
31   else:
32     Sensors.sensor_disabled.emit()
33
34   if not rewinding:
35     check_game_finished(step)
```

Figure 4.38: `update_game_state()` function from the `GlobalStateManager.gd` singleton

4.15 JavaScript Code Interpretation

For the purpose of having functioning JavaScript code written by the player, there must be a JavaScript interpreter to process the written code and return back to the game the needed information. Then it was decided that the interpretation of the code would be done in the player's browser, using the Godot `JavaScriptBridge` Singleton for the interaction between the game and the browser.

Initially, the interpretation of the JavaScript code was done directly in the browser's JavaScript execution context, sending the code as a string in the `eval()` function of the `JavaScriptBridge` singleton to be immediately executed in the browser. However, this was a very unsafe approach, because malicious scripts could be easily run not only on the console, but on the game aswell, greatly compromising the consistency of the game. To address this problem, I searched for a way to isolate the game code execution, from the browser's Javascript code execution context. The main solution identified was the use of NeilFraser's JS-Interpreter instances.

This section starts by presenting the NeilFraser's JS-Interpreter, highlighting its key features, and in what ways it is used in *CodinSpace* to execute player code. Next, it explores in detail the interaction between the game and the JS-Interpreter instances, explaining this in four parts. First, it presents the JavaScript Execution Handler singleton, responsible for managing all the data received from the JS-Interpreter instances and the data to send next. Second, it describes the JavaScript Function Calls singleton, which contains all the JavaScript functions calls needed to interact with the JS-Interpreter, as strings ready to be sent through the JavaScript Bridge. Third, it outlines how the game JS-Interpreter instance is initialized, which files are sent as strings to be executed in the browser's context and how the player code is sent to the interpreter. Finally, it provides a step-by-step explanation of the algorithm used to advance steps in the interpreter execution, with pseudo-code.

4.15.1 NeilFraser's JS-Interpreter

The NeilFraser's JS-Interpreter is a sandboxed JavaScript interpreter that is created in a completely isolated environment from the browser's JavaScript context. It comes with very powerful features that make it perfectly suited for use in *CodinSpace*, respectively:

- Step by step code execution (makes it possible to easily advance and rewind the game state, step by step);
- Information about the specific characters of code executed on each step (used in the game code highlight);
- Creation of global variables and functions that cannot be overridden once created (used to define game functions that can be used by the player such as `fly()`, `turnLeft()`, `obstacleIsForward()`);
- Detection of excessive memory use (used to restrict the maximum amount of memory that the player is allowed to allocate per level);
- Creation of multiple interpreter instances running separately (used to separate the game code execution, from the playground console code execution);
- Information about the current state of the interpreter (running step, waiting for an asynchronous function call or execution finished).

To execute the JavaScript code sent from the game, two separate JS-Interpreter instances were created, the game interpreter instance and the console interpreter instance. The game interpreter instance is responsible for executing the code written by the player in the game code editor, which contains the JavaScript instructions necessary to activate actions and sensors that will allow the spaceship to travel across the level and reach the target. On the other hand, the console interpreter instance is responsible for executing the code written in the playground console.

The main reason to separate the interpretation of playground code from game code is that the two will deal with very different instructions and execution contexts that should never be mixed with each other. This allows the player to freely write and execute code in the playground console, without ever affecting in any way the game's current state and the interpretation of game code.

4.15.2 Interaction between the game and the JS-Interpreter instances

In order to have a functioning game that responds to the instructions written by the player in JavaScript, many data must be exchanged frequently between the game and the JS-Interpreter instances. For the game to effectively send and retrieve that data, it must go through a series of steps and use the `JavaScriptExecutionHandler` and `JavaScriptFunctionCalls` singletons, created specifically to manage this interaction with the interpreters.

JavaScript Execution Handler

The `JavaScriptExecutionHandler` is a singleton responsible for managing all the flow of data between the game and the JavaScript interpreters. It has functions to initialize the interpreters and handle the advancing of execution steps.

JavaScript Execution Handler Functions List

`start_game_interpreter(player_code: String)` - Initializes a new game interpreter instance, then appends the "`player_code`" to its queue of pending code to be executed;

`start_console_interpreter()` - Initializes a new console interpreter instance;

`get_console_interpreter_is_executing()` - Returns `true` if the console interpreter is currently executing a new step, `false` otherwise. Used for the proper functioning of the playground console;

`set_console_interpreter_is_executing(value: bool)` - `console_interpreter_is_executing` variable is set to "`value`";

`step_forward_game_interpreter()` - Handles the advancing of one step forward in the game interpreter;

`step_forward_console_interpreter()` - Handles the advancing of one step forward in the console interpreter.

JavaScript Function Calls

During the early development phase, to execute a JavaScript function, the `JavaScriptExecutionHandler` would send the respective code as String parameter of the

```
func advance_game_interpreter_step():
  return await JavaScriptBridge.eval(Strings.JS_GAME_INTERPRETER_STEP)

func game_interpreter_append_code(code: String) -> void:
  var formatted_player_code := Strings.format_code(code)
  var javascript_player_code := Strings.JS_GAME_INTERPRETER_APPEND_CODE.format(
    {Strings.CODE: formatted_player_code}
  )
  await JavaScriptBridge.eval(javascript_player_code)
```

Figure 4.39: Functions from the `JavaScriptFunctionCalls` Singleton

`JavaScriptBridge.eval()` method. This quickly became problematic because it required to explicitly mix GDScript code with JavaScript functions in String form. To address this problem, it was decided to create a singleton to associate GDScript functions with JavaScript calls. Furthermore, by convention, a GDScript function name would be almost the same as its JavaScript counterpart, the difference being the use of snake_case for the former and camelCase for the latter.

Each of the GDScript functions starts by performing the necessary string transformations to the JavaScript code string, then the `JavaScriptBridge` `eval()` method is called, with the string passed as a parameter. The code is then interpreted by the JS-Interpreter, and the result is returned to the GDScript function. Finally, the function returns the result to the `JavaScriptExecutionHandler`, which will process the received information.

It is also important to note that to have a more consistent and clear code structure, each GDScript function was given the same name as its JavaScript counterpart, with the only difference that the GDScript functions were written in snake_case and JavaScript in camelCase.

`JavaScriptFunctionCalls` Singleton Functions List

`initialize_base_javascript_code()` - Gets the code from the three JavaScript files (`gameCode.js`, `gameFunctions.js`, `strings.js`), necessary for the correct execution of the player code in the interpreters, and sends it as a string to the browser with `JavaScriptBridge` `eval()`;

`update_state_in_game_interpreter()` - Updates the game state dictionary copy that is in the game JS-Interpreter instance, with the most recent game state dictionary data, so that every JavaScript action and sensor functions work properly and do their calculations correctly with updated information (asteroid positioning, collected game objects, player health, etc.);

`_ready()` - When the singleton is fully loaded, calls the `initialize_base_javascript_functions()` function and connects a signal that will call the `update_state_in_game_interpreter()` function whenever the game state changes;

`get_game_interpreter_step_value()` - Checks if there is data about an action or sensor activated by the player in the current step. If there is, then the data is returned as a *JavaScript Object Notation* (JSON) file to be processed by the game, otherwise it returns `null`;

`check_game_interpreter_current_step_is_end_of_statement()` - Returns `true` if the current step in the game interpreter is the last step of a statement, returns `false` otherwise;

`check_console_interpreter_current_step_is_statement()` - Returns `true` if the current step in the console interpreter is the last step of a statement, returns `false` otherwise;

`advance_game_interpreter_step()` - Orders the game interpreter to advance one step;

`advance_console_interpreter_step()` - Orders the console interpreter to advance one step;

`get_game_interpreter_highlighted_code_range()` - Returns the dictionary with start and end characters of the currently executed code on that step;

`get_game_interpreter_current_rough_value_memory_bytes()` - Returns the number of memory bytes that are currently allocated in the game interpreter instance;

`get_console_interpreter_current_rough_value_memory_bytes()` - Returns the number of memory bytes that are currently allocated in the console interpreter instance;

`get_game_interpreter_status()` - Returns the current status of the game interpreter instance, which can be 0 (The interpreter has finished executing all code), 1 (The interpreter has pending code to run and is prepared to advance to the next step), 2 (The interpreter currently does not have pending code to run, but there is a pending `setTimeout`/`setInterval` task which will provide code in the future), and 3 (The interpreter's execution is blocked by an asynchronous call);

`get_console_interpreter_status()` - Returns the current status of the console interpreter instance. Works similarly to `get_game_interpreter_status()`;

`get_game_interpreter_value()` - Returns the value that the game interpreter currently has set in its output property. This property is always getting updated to the most recent log, error, returned value and statement evaluation, being sometimes a useful source of information to the game about what is currently going on inside the interpreter in any specific step of execution;

`get_console_interpreter_value()` - Returns the value that the console interpreter currently has set in its output property. Works similarly to `get_game_interpreter_value()`;

`initialize_game_javascript_interpreter()` - Initializes the game JS-Interpreter instance, clearing all its context and memory of previous executions;

`initialize_console_javascript_interpreter()` - Initializes the console JS-Interpreter instance, clearing all its context and memory of previous executions;

`game_interpreter_append_code(code: String)` - Appends the JavaScript code string "`code`" to the game interpreter instance, adding it to the queue of pending code to run;

`console_interpreter_append_code(code: String)` - Appends the JavaScript code string "`code`" to the console interpreter instance, adding it to the queue of pending code to run.

Initialization of the game JS-Interpreter instance

Once the player starts a level, the state is initialized and the `initialize_base_javascript_code()` function is called, sending files with code to be executed on the browser, in order to prepare it with all the necessary functions and variables to receive player code.

```
func initialize_base_javascript_code() -> void:
  var javascript_strings: String = Global.get_javascript_code(Strings.
      PATH_JAVASCRIPT_STRINGS_JS)
  var javascript_game_functions: String = Global.get_javascript_code(Strings.
      PATH_JAVASCRIPT_GAME_FUNCTIONS_JS)
  var javascript_game_code: String = Global.get_javascript_code(Strings.
      PATH_JAVASCRIPT_GAME_CODE_JS)

  await JavaScriptBridge.eval(javascript_strings, true)
  await JavaScriptBridge.eval(javascript_game_functions, true)
  await JavaScriptBridge.eval(javascript_game_code, true)
```

Figure 4.40: `initialize_base_javascript_code()` function from the `JavaScriptFunctionCalls` Singleton

```
var gameInterpreter = new Interpreter('', initFunc);
var consoleInterpreter = new Interpreter('');
```

Figure 4.41: The two JS-Interpreter instances being created

The browser runs three distinct JavaScript files:

1. `strings.js` - Stores every constant text string that is used on the browser and on the JS-Interpreter instances;
2. `gameFunctions.js` - Has every action and sensor game function that can be used by the player. The functions are all wrapped inside another function, that will be used to initialize the game JS-Interpreter;
3. `gameCode.js` - Contains functions that give useful information to the game about the code execution. Most of these functions are called and awaited by the game, with the `JavaScriptBridge eval()`. Subsequently, the returned values are sent back to the game. This file also has a function to update the state variable on the browser, with the current value of the state dictionary of the game. The entire dictionary is sent from the game as a string and then converted to a JSON object on the game interpreter instance.

When the player submits written code for execution in the code editor, by pressing the "Run" button, the `start_game_interpreter()` function is called, instantiating the game JS-Interpreter and appending the player code to it. The code is now ready to be interpreted step by step.

```
# Initialize the JavaScript Interpreter and append player code
func start_game_interpreter(player_code: String) -> void:
  await JavascriptFunctionCalls.initialize_game_javascript_interpreter()
  JavascriptFunctionCalls.game_interpreter_append_code(player_code)
  code_started_executing.emit()
```

Figure 4.42: `start_game_interpreter()` function from the `JavaScriptExecutionHandler` Singleton

Advancing execution steps in the game interpreter instance

As explained in the "Advancing and Rewinding Game State" chapter of the dissertation, when the player requests the game to advance the state, if the current step is the most recent one, then it means the next step has not been interpreted yet by the game interpreter instance. When this happens, the Steps State Manager will request the JavaScript Execution Handler singleton to advance execution steps in the game interpreter, by calling its `step_forward_game_interpreter()` function.

This is the `step_forward_game_interpreter()` function in pseudocode:

```
function step_forward_game_interpreter():
    var current_step_is_statement = false
    var execution_terminated = false
    while not current_step_is_statement:
        var id = get_interpreter_steps_count()

        # Shows in Game Over popup
        var message = null
        # Shows in Game Console
        var error = null

        var content = get_step_value()
        if content != null:
            process_step_content(content)

        # Interpreter reached end of execution, or encountered error (Code 0)
        if get_interpreter_status() == 0:
            execution_terminated = true
            message = get_interpreter_stopped_message()
            error = get_interpreter_stopped_error()

        if not execution_terminated:
            advance_interpreter_step()
        current_step_is_statement = check_current_step_is_statement()

        var highlighted_code_range = get_highlighted_code_range()

        if exceeds_memory_limit() or exceeds_runtime_limit() or exceeds_step_limit
            ():
            execution_terminated = true
            handle_constraint_violation()
            message = get_constraint_violated_message()
            error = get_constraint_violated_error()

        save_step_data(
            id,
            execution_terminated,
            highlighted_code_range,
            content,
            error,
            message,
            used_memory,
            current_step_is_statement,
            elapsed_runtime
        )
        increment_interpreter_steps_count()

        if execution_terminated or fetch_substeps_enabled():
            break
```

The `step_forward_game_interpreter()` function manages the process of advancing execution steps. Its algorithm consists of a loop that continuously requests new steps of execution to the

Step	Action	Condition/Check
1. Initialize variables	Initialize necessary variables like `current_step_is_statement` and `execution_terminated`.	None
2. Get current step value	Retrieve the current step value using `get_step_value()`.	`content != null`
3. Check interpreter status	Check the status of the interpreter (`get_interpreter_status()`).	If `status == 0`, terminate execution with message and error.
4. Advance interpreter step	Advance the interpreter by one step.	If no termination or constraint violation.
5. Check constraints	Ensure that memory, runtime, and step limits are not exceeded.	If limits are exceeded, terminate execution and handle the constraint violation.
6. Save step data	Save the current step data for the game state update.	Always executed at the end of the loop.
7. Increment step count	Increment the interpreter's step count.	Always executed at the end of the loop.
8. End or continue loop	If a full statement is reached or a constraint is violated, stop the loop.	If `execution_terminated` or `fetch_substeps_enabled()` is true, exit the loop.

Table 4.4: Step-by-Step Logic in `step_forward_game_interpreter()`

interpreter until a full statement (a full step) is reached. In each iteration, the function checks the interpreter status, advances one step, and validates the max memory, runtime and step constraints. At the end of each iteration, the new step data is added to the completed steps array and the step count is incremented. The function stops earlier if the player is fetching all substeps, the interpreter reaches the end of execution, encounters an error, or violates any constraints After the function execution is finished, the game state is updated with the changes contained in the new step data.

4.16 Level and Chapter definition

Campaign levels are defined in their own JSON files, which have all the information needed to render and start a level, respectively the id, music id, constraints, player initial state, target, info section text markdown and game objects initial state.

A campaign chapter is defined in its own folder, containing a levels folder that contains the JSON file of each level inside that chapter and a `chapter.json` file that has information about the chapter such as its name, description, position of the level nodes, and order of completion of the levels (connections between the level nodes).

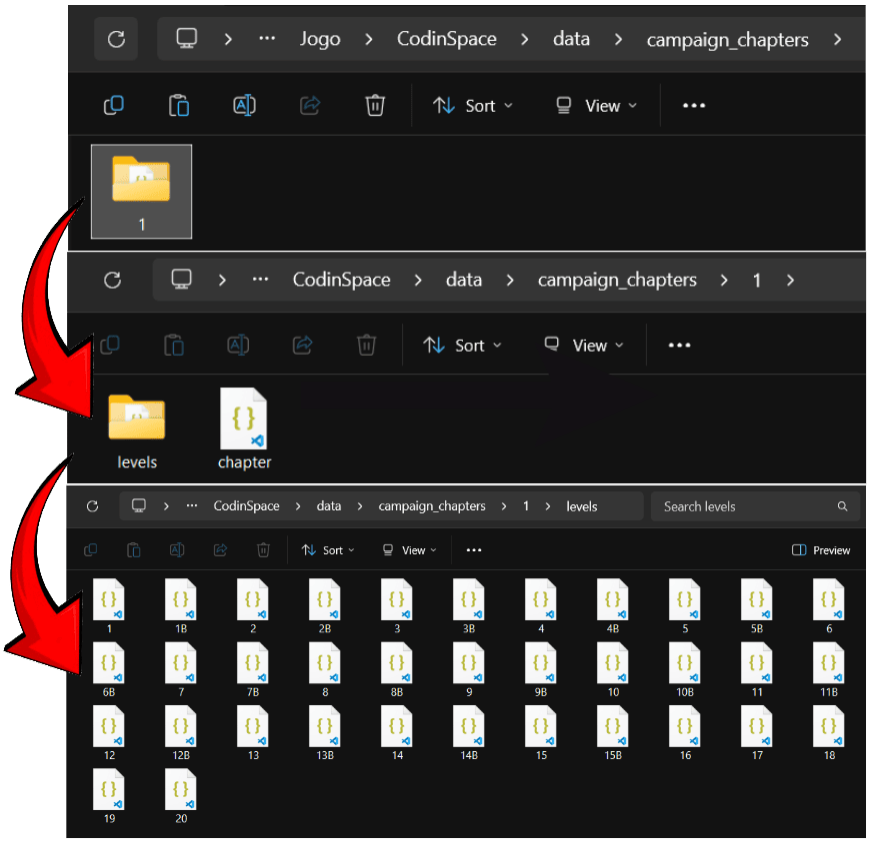

Figure 4.43: Files and folders structure

```
{
  "uuid": "8ap7t0x7-c8ak-j9rz-0r66s3n6del5"
  "constraints": {
    "maxCodeCharacters": 180,
    "maxFuel": 20,
    "maxGameInterpreterMemoryBytes": 100000,
    "maxGameInterpreterRuntime": 10000,
    "maxGameInterpreterSteps": 500,
    "maxHealth": 1,
    "maxMinutes": 10
  },
  "musicNumber": 4,
  "name": "Level 9 - Two Paths",
  "description": "Travel to the button, then return to the door. A new case of while loops.",
  "placeablesIdCount": 38,
  "player": {
    "heading": 0,
    "spaceSignalRange": 20,
    "x": -3,
    "y": -2
  },
  "target": {
    "color": "#ffffff",
    "requirements": {},
    "x": 2,
    "y": 0
  },
  "gameObjects": {
    "asteroids": {
      "1": {
        "color": "#ffffff",
        "damage": 1,
        "radius": 1,
        "x": -12,
        "y": -7
```

Figure 4.44: Level definition in JSON

Figure 4.45: Chapter definition in JSON

4.17 Community Driven Game Content

To ease the process of creating new chapters and levels, not only for the developer but also to the game community and to allow easy expansion of the game beyond its original content, it was decided that it is important to give the community tools to create new chapters and levels.

4.17.1 Level Editor

Using the level editor, it is possible to create new levels with ease and in an intuitive way. It was not only used to create all the existing levels inside the game, but is also available for the community members to create their own game content and levels.

The level editor is divided into various sections, each with its own utilities:

Level View

Camera view of the level that is being created, allowing the creator to edit the level by directly making visual changes on it. The level may sometimes be much larger than the screen size, so the camera can be zoomed in/out and dragged around the level, making it easier to navigate around the level.

There is also a grid that can be enabled, to help with positioning the placebles in the right place. The grid divides each level coordinate and has on top and left the value of the x and y axis, respectively.

Figure 4.46: Level Editor Tools

Select, copy and remove placeables

Three important tools of the level editor are the selector, the eyedropper and the remover. The selector is used to select a specific placeable (player, game object, target) from the level view and investigate its details. By pressing the placeable with the selector equipped, the placeable details window is displayed. The eyedropper copies a placeable and all its properties, then allows it to be cloned in other parts of the level, without having to redefine all its properties for each clone placeable. The remover is a tool that allows the removal of placeables from the level by clicking them.

Tool	**Purpose**
Selector	Selects placeables on the level and shows their details, such as position, type, and attributes.
Eyedropper	Allows the cloning of placeables and their properties.
Remover	Removes placeables from the level.

Table 4.5: Overview of Tools in the Level Editor

Level Details

The level details display is positioned on the left side of the screen and has general information about the level, such as name, description, and music. The level creator can also define here the value of each constraint, respectively the interpreter constraints (maximum memory bytes, maximum execution steps, maximum runtime) and the game constraints (maximum player health, maximum player fuel). On the bottom of the sidebar there is also an "Export" button to open the export level window.

The level details can be opened or closed by pressing the "Level Details" button located on the top left side of the screen.

Placeables List

Displays in a grid with various tabs every placeable that can be added to the level. By selecting a placeable from the grid, it becomes possible to spawn the placeable inside the level view, just by clicking on the grid tiles where we want to place it.

Placeable Details Section

Whenever a placeable is clicked with the selector tool, the placeable details section is opened. This section displays information about the placeable, such as name, id, color, is opened, is pressed, and many other properties, depending on the type of placeable that is selected. Each of these properties can be modified, changing the behavior and appearance of the placeable.

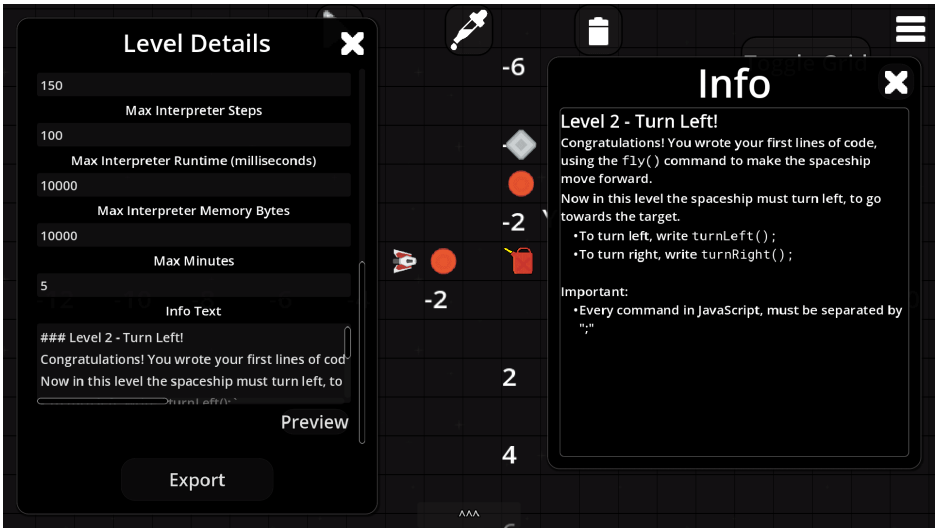

Figure 4.47: Level details and the info section preview

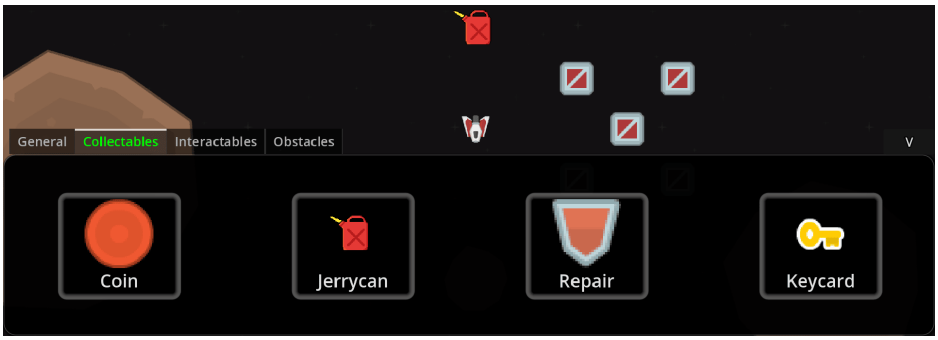

Figure 4.48: Placeables List, containing various game objects that can be placed inside the level

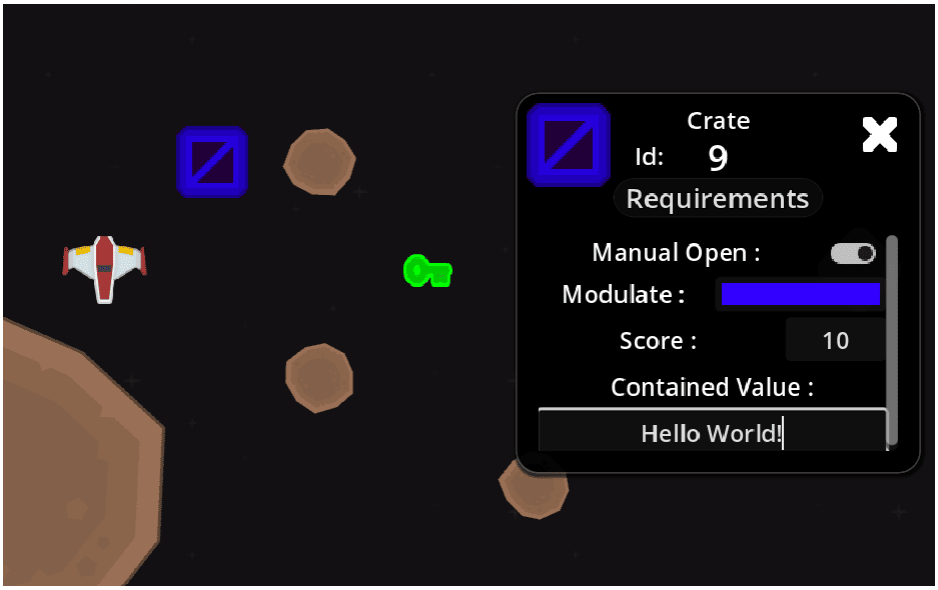

Figure 4.49: Placeable Details

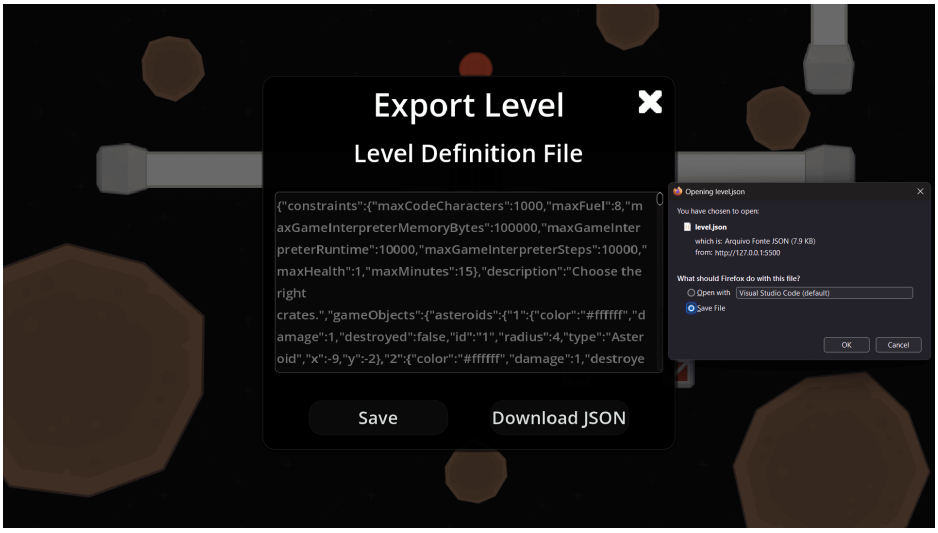

Figure 4.50: Export Level Modal

Exporting Levels

After successfully creating a level in the level editor, the player can not only keep the level saved on the browser storage, but can also download it to a JSON file, making it possible to share the level with other players in the community. In order to export the level, the player must press the "Export" button on the level details, therefore opening the export window. Inside the export window, the player can either save the level locally on the game by clicking the "Save" button, or download the level definition JSON file by pressing the "Download JSON" button. The export window also shows the preview of the JSON text, allowing the creator to see how the JSON of the level is defined.

4.17.2 Chapter Editor

Chapters can be created in the chapter editor, a tool designed to make it easy to assemble different levels together into a chapter and define them sequentially. In the chapter editor, it is possible to place level nodes in any position, each one being associated with a level from the list of player-made levels. The nodes are then connected to each other according to the list of node connections, creating a graph that can be navigated by the player spaceship in a specific order.

Finally, the chapter is given a name and description, then it is saved to the browser storage or is exported, so that it becomes possible to share the chapter and its levels with the *CodinSpace* community.

Chapter View

Same as the level editor, the chapter editor also has a camera view that allows the creator to intuitively visualize every change made to the level nodes or its connections instantly. It also has a camera that can be zoomed and dragged, and a grid which shows the boundaries between each coordinate and helps to determine the correct positioning of every level node in the chapter view.

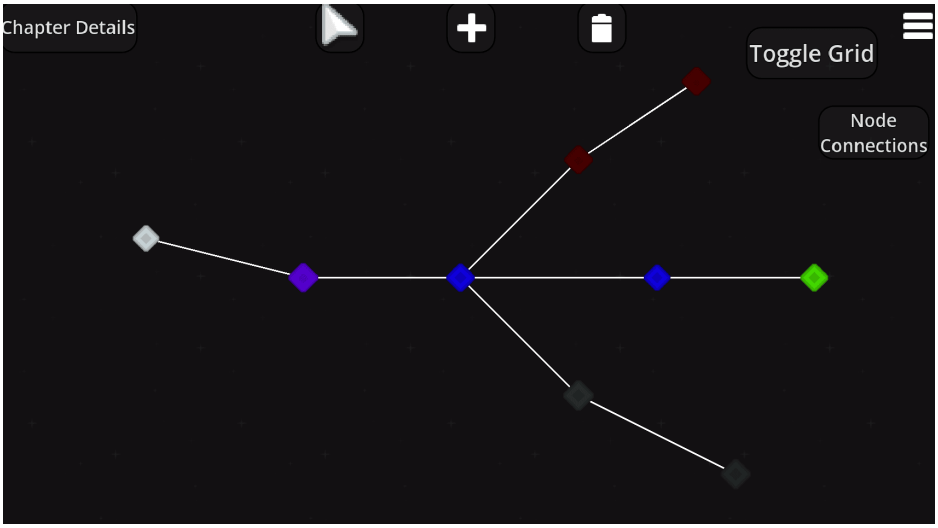

Figure 4.51: Chapter Editor

Figure 4.52: Chapter Editor Tools

Select, insert and remove level nodes

The selector, inserter and remover are three tools which can be activated by pressing their respective buttons on the top of the screen. The selector can be used to select a specific level node from the chapter view and obtain its details. By pressing a level node with the selector equipped, the level node details window is displayed. On the other hand, the inserter is a tool that allows the insertion of new level nodes to the chapter view by clicking the coordinates where we want to place them. Lastly, the remover is a tool that allows the removal of placed level nodes from the chapter view, by clicking them.

Chapter Details

The chapter details display is located on the left side of the screen and is where the creator can define the chapter name and description. On the bottom of the chapter details display there is also an "Export" button that when pressed opens the export chapter window.

The chapter details can be opened or closed by pressing the "Chapter Details" button located on the top left side of the screen.

Level Node Details

By selecting a level node from the chapter view with the selector tool, the level node details window is opened. This window displays information about the node, such as the *Universally Unique Identifier* (UUID), name and description of the level that is linked to it, the color of the node, and a flag that indicates if it is the first level node of the chapter, so that the game knows where to initially place the player spaceship in the chapter graph when presenting the level selection screen.

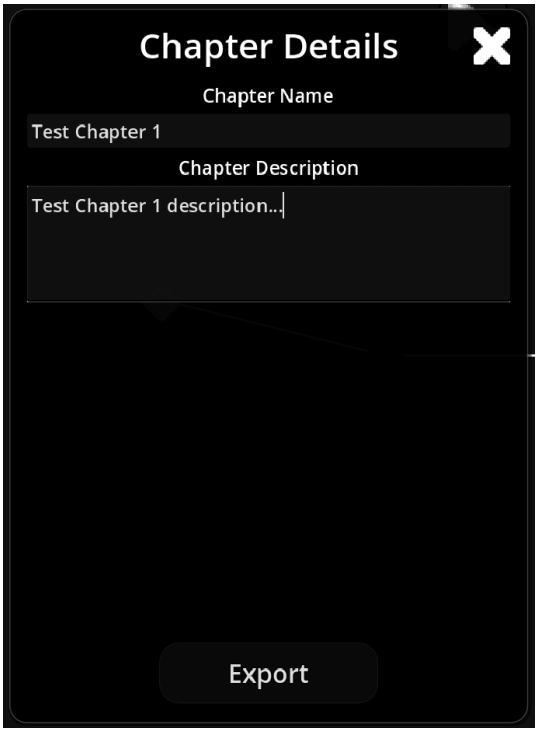

Figure 4.53: Chapter Details

The level node details window also has a plus icon button that opens the "Link level to node" window for that specific node.

Linking existing levels to nodes

In order to assemble a fully working chapter, the nodes used to build the chapter graph must all be associated to existing levels from the player-made levels list. To achieve this, the creator can open the details of a specific level node, and there press the plus icon button. By doing so, the "Link level to node" window opens and displays a list with all the existing levels that can be bound to that node. Each level entry in the displayed list has the name and description of the level and a plus icon button, that can be pressed to link the level to the node. Once the level is linked to the level node, then the level node details window is refreshed, presenting the UUID, name and description of the newly selected level. Now whenever the player's spaceship travels to that specific node, the selected level appears.

Node Connections

To connect the level nodes in the graph, there must be associations between the nodes of each connection, indicating the id of the first and second nodes for each connection. These connections can be created on the node connections window, which can be opened by pressing the "Node Connections" button on the right side of the screen.

The node connections window presents to the creator a list of all the existing connections in the chapter and a "New" button that creates a new entry in the list for a new connection. Each connection in the list has two input boxes, one for the id of the first node and another for the id of the second node. By having the creator fill these ids, the connection is created and becomes visible

Figure 4.54: Level node details window

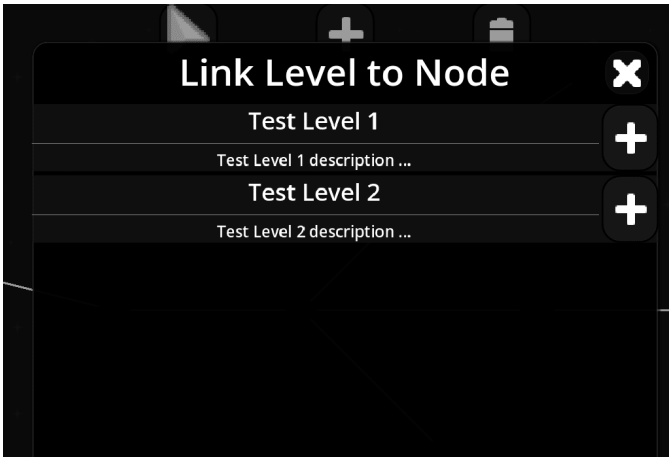

Figure 4.55: Link level to node window, with the list of custom levels that can be associated to the node

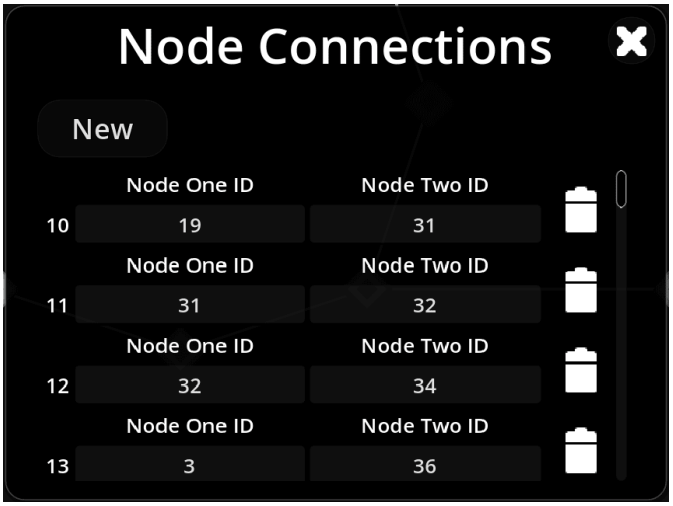

Figure 4.56: Node connections window

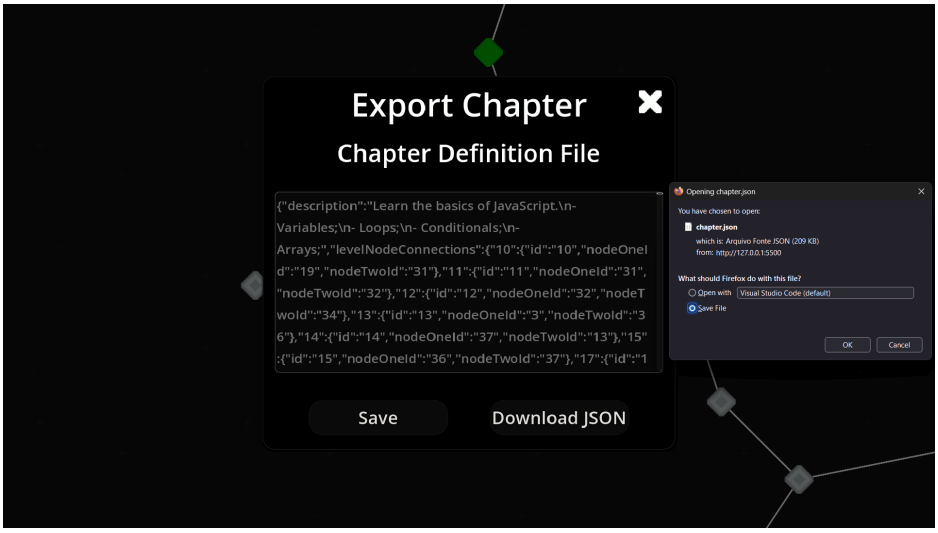

Figure 4.57: Export Chapter Modal

in the chapter view, represented by a line between the two nodes. Each entry in the connections list also has a trash icon button, which when clicked, removes the connection bound to that entry.

By allowing for the creation of custom connections between each node, the creator can creatively build sequences of levels that may branch into different topics and difficulties, with simplicity.

Exporting Chapters

Exporting chapters works in the same way as exporting levels in the level editor. By clicking the "Export" button in the chapter details, the export chapter window appears, allowing the creator to either save the chapter locally in the game or download the chapter definition JSON.

When exporting a chapter, the JSON definition of each level contained in it is added to the exported file, so that when importing this chapter later, every level is included without having to import them separately.

Figure 4.58: Button to import level JSON file, in the community level selection menu

Figure 4.59: Importing a chapter JSON file from the file explorer into CodinSpace

4.17.3 Importing chapters and levels

The player can import chapters and levels made by the community, by pressing the "Import Chapter JSON File" or "Import Level JSON File" buttons, located on the community chapters and levels selection menus, respectively. When the button is pressed, the file explorer of the local machine opens, and the player is prompted to select an existing chapter/level JSON file.

In case the player imports a chapter, the chapter is added to the community chapters list, and all its levels are inserted in the game and added to the community levels list. If the player just imports an individual custom level, then it is added to the community levels list.

To make the community content creation more versatile and diverse, the chapter creators can not only link levels that were made by them to the chapter but can also link any imported level from the community to the chapter, making it possible to create chapters with "mixes" of levels from various community creators.

4.18 GitHub repository

The *CodinSpace* Godot project can be accessed in the following GitHub repository:

`https://github.com/DiogoPinto123/ue-codinspace`

To run *CodinSpace*, you just need to open a local web server inside the folder `CodinSpaceHTML5`, and access the web server with your desired browser.

5

Formal Game Description

This section provides a formal, mathematical description of the key elements in a *CodinSpace* level:

Game State - The essential information about the game;

Game View - The part of the game state that is visible to the player;

Game Setup - How the game starts, i.e., its initial state;

Player Actions - Description of the player actions and how they define the next state;

Player Sensors - Description of the player sensors and what information they provide;

Conditions to finalize game - Definition of the player victory and game over conditions;

Progression of Play - How the game progresses to the next state.

5.1 Data Types Notation

Considering the high diversity of data types used in the game, it is important to represent them in the formal game description with clarity to the reader. To achieve that, it was decided that the right decision was to define them with a mathematical notation where each data type values are split into different sets:

- $\mathbb{Z}$: The set of integers:
$$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$$
- $\mathbb{T}$: The set of text strings:
$$\mathbb{T} = \{\dots, \text{"}a\text{"}, \text{"}heading\text{"}, \text{"}open\text{"}, \dots\}$$
- $\mathbb{B}$: The boolean set, containing *true* and *false*:
$$\mathbb{B} = \{\textit{true}, \textit{false}\}$$
- $\mathbb{V}$: The set of 2D vectors, defined as pairs of integers:
$$\mathbb{V} = \{(x, y) \mid x \in \mathbb{Z}, y \in \mathbb{Z}\}$$
- $\mathbb{D}$: The set of dictionaries. A dictionary is represented as a mapping from keys to values:
$$\mathbb{D} = \{\text{"key1"} \rightarrow \text{"value1"}, \text{"key2"} \rightarrow \text{"value2"}\}$$
- **Lists**: A list of elements of type X is denoted as $[X]$. For example, $[\mathbb{Z}]$ represents the set of lists of integers:
$$[\mathbb{Z}] = [1, 2, 3, 4]$$

5.2 Game State

The set of states is:
$$S = P \times G \times C \times L \times X \times E \times O$$

Where:

- P is the set of all possible player states;
- G is the set of all possible game object states;
- C is the set of all possible constraint states;
- L is the set of all possible console states;
- X is the set of all possible step states;
- E is the set of all possible space signal states;
- O is the set of all possible state properties.

The properties in O are:

- h: $\mathbb{D}$ - Range of characters from the player code that correspond to the code that is executed in the current step. Represented as a dictionary with two entries, one is for the starting character and another is for the last character of the range;
- t: $\mathbb{D}$ - Dictionary containing the target position;
- m: $\mathbb{Z}$ - Level music number;
- y: $\mathbb{T}$ - Message that is presented whenever the game finishes and the player didn't achieve the objective. It can be different, depending on the way that the player lost the game (*e.g.* error, constraint exceeded, spaceship destroyed);
- r: $\mathbb{Z}$ - Interpreter runtime used, till the current step (sum of all the currently interpreted steps runtime);
- b: $\mathbb{Z}$ - Memory bytes allocated in the interpreter, at the current step;
- n: $\mathbb{T}$ - Level name;
- i: $\mathbb{T}$ - Level info text;
- z: $\mathbb{Z}$ - Player score;
- f: $\mathbb{Z}$ - Final player score in the level, which is the combination of the score obtained in the level and the bonus score obtained from the remaining time;
- u: $\mathbb{Z}$ - Remaining minutes for the player to complete the level;
- v: $\mathbb{Z}$ - Remaining seconds of the current minute.

The `GlobalStateManager` singleton manages all state management singletons and keeps track of the entire state of the game in the state dictionary. This dictionary is sent to the JS-Interpreter instance in the browser at every step, in order to give updated information about the game to the JavaScript functions that need these data to work properly (`get_health`, `obstacle_forward`, `is_collected`, and many other functions use data from the current state of the game to make decisions and calculate their outputs).

During a level, all the game state data from each execution step is stored in a dictionary. This dictionary is managed by seven state management singletons, each one responsible for creating, reading, updating, and deleting (CRUD) the data of a specific section of the state.

State Functions

If $p \in P$, $g \in G$, $c \in C$, $l \in L$, $x \in X$, $e \in E$, $o \in O$ we also define the following functions:

Update global state

$$\begin{gathered} \textit{update\_global\_state}(s) = s' \quad \text{such that} \\ s'_c = c, \quad s'_x = x, \\ s'_p = p, \quad s'_g = g, \quad s'_e = e \end{gathered}$$

Updates the global state with the updated values of each sub state.

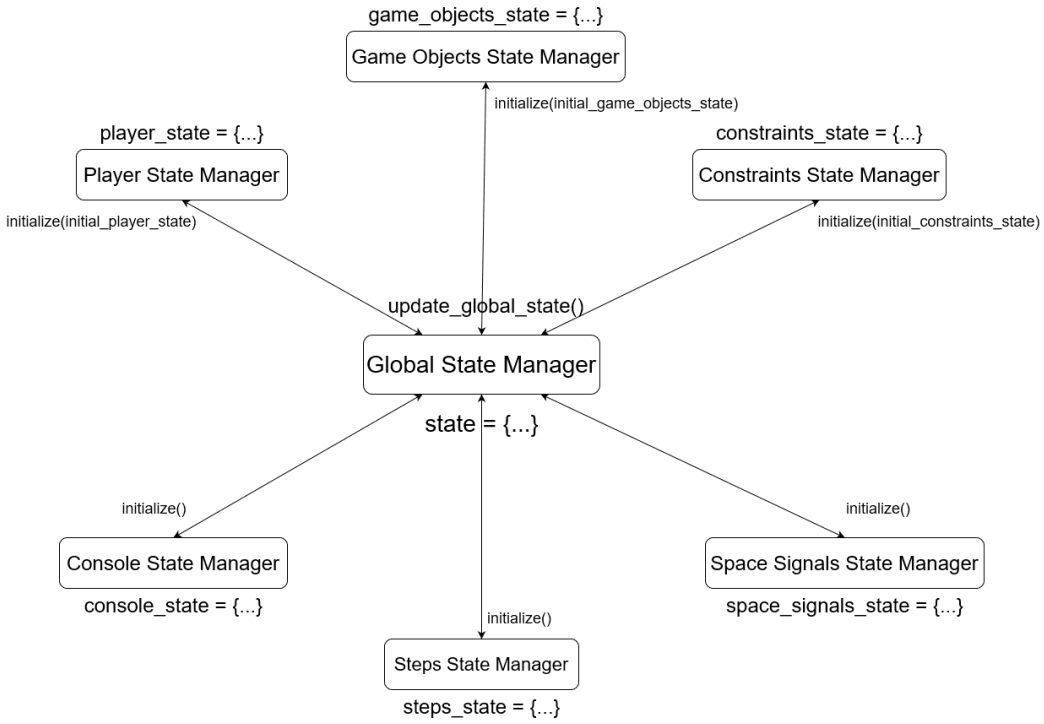

Figure 5.1: Diagram presenting all the seven state management singletons

Advance step

$$\textit{advance\_step}(s) = s' \quad \text{such that}$$
$$s' = \textit{update\_game\_state}(s,\ step)\ \times\ \textit{update\_global\_state}(s)$$

Advances one JavaScript execution step in the player code. The updated state s' is computed by fetching the next step data, updating the game state, and updating the global state.

Rewind step

$$\textit{rewind\_step}(s) = s' \quad \text{such that}$$
$$s' = \textit{update\_game\_state}(s,\ step)\ \times\ \textit{update\_global\_state}(s)$$

Rewind one JavaScript execution step in the player code.

Update Game State

$$\textit{update\_game\_state}(s, n) = s' \quad \text{such that}$$
$$s' = \textit{process\_step}(s, n)\ \wedge\ \textit{E.disable\_all\_space\_signals}(s_e)$$
$$\wedge\ \textit{process\_actions}(s', n)\ \wedge\ \textit{check\_game\_finishes}(n)$$

Updates the state s with all the changes of step n.

- **Process Step:**

$$\textit{process\_step}(s, n) = \begin{cases} \textit{initialize\_highlighted\_code\_range()} & \text{if } \textit{S.current\_step\_is\_first()} \\ \\ \textit{update\_highlighted\_code\_range}(n_h) & \\ \wedge\ \textit{update\_interpreter\_memory\_bytes}(n_b) & \\ \wedge\ \textit{update\_interpreter\_runtime}(n_r) & \text{otherwise.} \end{cases}$$

- **Process Actions:**

$$\textit{process\_actions}(s', n) = \begin{cases} \textit{Sensors.sensor\_activated.emit}(s, v) & \\ \wedge\ \textit{L.add\_sensor\_log}(s, v), & \text{if } n_t = \text{Sensor} \\ \\ \textit{Actions.do\_step\_action}(c) & \\ \wedge\ \textit{G.update\_game\_objects()}, & \text{if } n_t = \text{Action} \end{cases}$$

Highlighted code range

$$\textit{highlighted\_code\_range}(s) = s_h$$

Returns the dictionary with the highlighted code range s_h.

- $s_h \in \mathbb{D}$: Highlighted code range.

Update highlighted code range

$$\textit{update\_highlighted\_code\_range}(s, v) = s' \quad \text{such that} \quad s'_h = v$$

Updates the highlighted code range s_h to the value v.

- $s_h \in \mathbb{D}$: Highlighted code range;
- $v \in \mathbb{D}$: New highlighted code range value.

Target

$$\textit{target}(s) = s_t$$

Returns the target info s_t.

- $s_t \in \mathbb{D}$: Target info.

Target position

$$\textit{target\_position}(s) = \begin{pmatrix} t_x \\ t_y \end{pmatrix} \quad \text{such that} \quad (t_x, t_y) = s_t$$

Returns the target position (t_x, t_y).

- $s_t \in \mathbb{D}$: Target info;
- $(t_x, t_y) \in \mathbb{V}$: Target position.

Target color

$$\textit{target\_color}(s) = s_{t,c}$$

Returns the target color $s_{t,c}$ from the target s_t.

- $s_{t,c} \in \mathbb{T}$: Target color.

Target requirements

$$\textit{target\_requirements}(s) = s_{t,r}$$

Returns the target requirements $s_{t,r}$ from the target s_t.

- $s_{t,r} \in \mathbb{D}$: Target requirements.

Music number

$$\textit{music\_number}(s) = s_m$$

Returns the music number s_m of the current level.

- $s_m \in \mathbb{Z}$: Music number.

Game over message

$$\textit{game\_over\_message}(s) = s_o$$

Returns the game over message s_o.

- $s_o \in \mathbb{T}$: Game over message.

Update game over message

$$\textit{update\_game\_over\_message}(s, v) = s' \quad \text{such that} \quad s' = s \text{ with } s'_o = v$$

Updates the game over message s_o to the value v.

- $s_o \in \mathbb{T}$: Game over message;
- $v \in \mathbb{T}$: New game over message value.

Interpreter runtime

$$\textit{interpreter\_runtime}(s) = s_r$$

Returns the interpreter runtime s_r.

- $s_r \in \mathbb{Z}$: Interpreter runtime.

Update interpreter runtime

$$\textit{update\_interpreter\_runtime}(s, v) = s' \quad \text{such that} \quad s' = s \quad \text{with} \quad s'_r = v$$

Updates the interpreter runtime s_r to the value v.

- $s_r \in \mathbb{Z}$: Interpreter runtime;
- $v \in \mathbb{T}$: New interpreter runtime value.

Increment interpreter runtime

$$\textit{increment\_interpreter\_runtime}(s, v) = s' \quad \text{such that} \quad s'_r = s_r + v$$

Increments the interpreter runtime s_r by v.

- $s_r \in \mathbb{Z}$: Interpreter runtime;
- $v \in \mathbb{Z}$: Interpreter runtime increment value.

Interpreter memory bytes

$$\textit{interpreter\_memory\_bytes}(s) = s_b$$

Returns the allocated interpreter memory bytes s_b.

- $s_b \in \mathbb{Z}$: Allocated interpreter memory bytes.

Update interpreter memory bytes

$$\textit{update\_interpreter\_memory\_bytes}(s, v) = s' \quad \text{such that} \quad s' = s \quad \text{with} \quad s'_b = v$$

Updates the allocated interpreter memory bytes s_b to the value v.

- $s_b \in \mathbb{Z}$: Allocated interpreter memory bytes;
- $v \in \mathbb{T}$: New allocated interpreter memory bytes value.

Level name

$$\textit{level\_name}(s) = s_n$$

Returns the level name s_n.

- $s_n \in \mathbb{T}$: Level name.

Info text

$$\textit{info\_text}(s) = s_i$$

Returns the info text s_i.

- $s_i \in \mathbb{T}$: Info text.

Score

$$\textit{score}(s) = s_z$$

Returns the player score s_z.

- $s_z \in \mathbb{Z}$: Player score.

Increase score

$$\textit{increase\_score}(s, v) = s' \quad \text{such that} \quad s'_z = s_z + v$$

Increases the player score s_z by v points.

- $s_z \in \mathbb{Z}$: Player score;
- $v \in \mathbb{Z}$: Points.

Decrease score

$$\textit{decrease\_score}(s, v) = s' \quad \text{such that} \quad s'_z = s_z - v$$

Decreases the player score s_z by v points.

- $v \in \mathbb{Z}$: Points.

Final score

$$\textit{final\_score}(s) = s_f$$

Calculates and returns the final player score s_f, by joining the player score s_z with the bonus score obtained from the remaining time.

- $s_f \in \mathbb{Z}$: Final score.

Elapsed time

$$\textit{elapsed\_time}(s) = s_e$$

Returns the elapsed time s_e since the beginning of the level.

- $s_e \in \mathbb{T}$: Elapsed time.

5.2.1 Player State

The set of player states is:

$$P = \{(x, y, s, h, f, r, e, o, j)\}$$

Where:

- $x, y \in \mathbb{Z}$: Player position on the level;
- $s \in \mathbb{Z}$: Player heading in degrees;
- $h \in \mathbb{Z}$: Player health;
- $f \in \mathbb{Z}$: Player fuel;
- $r \in \mathbb{Z}$: Player space signal range;
- $e \in \mathbb{B}$: Player is currently emitting a space signal;
- $o \in \mathbb{B}$: Player is currently opening;
- $j \in \mathbb{B}$: Player is currently pressing.

Player State Manager Singleton

The `PlayerStateManager` singleton is responsible for managing all the player spaceship properties and the player state at each step of execution. The singleton holds information about the player, such as their position (`Vector2` value with coordinates `x` and `y`), and other integer values for the heading, health, fuel, and space signals range. There are also various boolean properties, which indicate if the player is opening, pressing or emitting, at the current step of execution.

Player State Functions

If $p \in P$, we also define the following functions:

Update player state

$$\textit{update\_player\_state}(p, v) = p' \quad \text{such that} \quad p' = v$$

Updates the player state p to the value v.

- $v \in P$: New value.

Player heading

$$\textit{player\_heading}(p) = p_s$$

Returns the player heading p_s from the player state p.

- $p_s \in \mathbb{Z}$: Player heading.

Update player heading

$$\textit{update\_player\_heading}(p, v) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_s = v$$

Updates the player heading p_s to the value v.

- $p_s \in \mathbb{Z}$: Player heading;
- $v \in \mathbb{Z}$: New heading value.

Player position

$$\textit{player\_position}(p) = (p_x, p_y)$$

Returns the player position (p_x, p_y) from the player state p.

- $p_x \in \mathbb{Z}$: Player position X coordinate;
- $p_y \in \mathbb{Z}$: Player position Y coordinate.

Update player position

$$\textit{update\_player\_position}(p, v) = p' \quad \text{such that} \quad p'_x = v_x \ \wedge \ p'_y = v_y$$

Updates the player position (p_x, p_y) to the value v.

- $p_x \in \mathbb{Z}$: Player position X coordinate;
- $p_y \in \mathbb{Z}$: Player position Y coordinate;
- $v \in \mathbb{V}$: New position value.

Player pressing

$$\textit{player\_pressing}(p) = p_j$$

Returns the boolean property p_j (pressing) from the player state p.

- $p_j \in \mathbb{B}$: Pressing.

Update player pressing

$$\textit{update\_player\_pressing}(p, v) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_j = v$$

Updates the boolean property p_j (pressing) to v.

- $p_j \in \mathbb{B}$: Pressing;
- $v \in \mathbb{B}$: New boolean value.

Player opening

$$\textit{player\_opening}(p) = p_o$$

Returns the boolean property p_o (opening) from the player state p.

- $p_o \in \mathbb{B}$: Opening.

Update player opening

$$\textit{update\_player\_opening}(p, v) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_o = v$$

Updates the boolean property p_o (opening) to v.

- $p_o \in \mathbb{B}$: Opening;
- $v \in \mathbb{B}$: New boolean value.

Player emitting

$$\textit{player\_emitting}(p) = p_e$$

Returns the boolean property p_e (emitting) from the player state p.

- $p_e \in \mathbb{B}$: Emitting.

Update player emitting

$$\textit{update\_player\_emitting}(p, v) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_e = v$$

Updates the boolean property p_e (emitting) to v.

- $p_e \in \mathbb{B}$: Emitting;
- $v \in \mathbb{B}$: New boolean value.

Player health

$$\textit{player\_health}(p) = p_h$$

Returns the player health p_h from the player state p.

- $p_h \in \mathbb{Z}$: Player health value.

Update player health

$$\textit{update\_player\_health}(p, v) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_h = v$$

Updates the player health p_h to the value v.

- $p_h \in \mathbb{Z}$: Player health value;
- $v \in \mathbb{Z}$: New player health value.

Check player has health

$$\textit{player\_has\_health}(p) = a \quad \text{such that} \quad a = \begin{cases} \textit{true} & \text{if} \quad \textit{player\_health}(p) > 0 \\ \textit{false} & \text{otherwise} \end{cases}$$

Determines whether the player has health remaining.

- $a \in \mathbb{B}$: Boolean value indicating whether the player has health remaining.

Damage player

$$\textit{damage\_player}(p, a) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_h = \max(0, p_h - a)$$

Decreases the player health p_h from the player state p, by amount a.

- $p_h \in \mathbb{Z}$: Player health value;
- $a \in \mathbb{Z}$: Amount of health points to decrease.

Heal player

$$\textit{heal\_player}(p, a) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_h = \min(m, p_h + a)$$

Increases the player health p_h in the player state p, by amount a. The new health value cannot be greater than the maximum health value m.

- $p_h \in \mathbb{Z}$: Player health value;
- $a \in \mathbb{Z}$: Amount of health points to increase;
- $m \in \mathbb{Z}$: Maximum health points.

Player fuel

$$\textit{player\_fuel}(p) = p_f$$

Returns the player spaceship fuel p_f from the player state p.

- $p_f \in \mathbb{Z}$: Player spaceship fuel value.

Update player fuel

$$\textit{update\_player\_fuel}(p, v) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_f = v$$

Updates the player spaceship fuel p_f to the value v.

- $p_f \in \mathbb{Z}$: Player spaceship fuel value;
- $v \in \mathbb{Z}$: New player spaceship fuel value.

Check player has fuel

$$\textit{player\_has\_fuel}(p) = a \quad \text{such that} \quad a = \begin{cases} \textit{true} & \text{if } p_f > 0 \\ \textit{false} & \text{otherwise} \end{cases}$$

Determines whether the player spaceship has fuel p_f remaining.

- $p_f \in \mathbb{Z}$: Player's spaceship fuel value;
- $a \in \mathbb{B}$: Player's spaceship has fuel.

Increase player fuel

$$\textit{increase\_player\_fuel}(p, a) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_f = \min(m, p_f + a)$$

Increases the player spaceship fuel p_f from the player state p, by amount a. The new fuel value cannot be greater than the maximum fuel value m.

- $p_f \in \mathbb{Z}$: Player spaceship fuel value;
- $a \in \mathbb{Z}$: Amount of fuel to increase;
- $m \in \mathbb{Z}$: Maximum fuel value.

Decrease player fuel

$$\textit{decrease\_player\_fuel}(p, a) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_f = \max(0, p_f - a)$$

Decreases the player spaceship fuel p_f from the player state p, by amount a.

- $p_f \in \mathbb{Z}$: Player spaceship fuel value;
- $a \in \mathbb{Z}$: Amount of fuel to decrease.

Player space signal range

$$\textit{player\_space\_signal\_range}(p) = p_r$$

Returns the player spaceship space signal range p_r from the player state p.

- $p_r \in \mathbb{Z}$: Player spaceship space signal range.

Update player space signal range

$$\textit{update\_player\_space\_signal\_range}(p, v) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_r = v$$

Updates the player spaceship space signal range p_r to the value v.

- $p_r \in \mathbb{Z}$: Player spaceship space signal range;
- $v \in \mathbb{Z}$: New player space signal range value.

Increase player space signal range

$$\textit{increase\_player\_space\_signal\_range}(p, a) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_r = \min(m, p_r + a)$$

Increases the radius p_r of space signals emitted by the player, by amount a.

- $p_r \in \mathbb{Z}$: Player spaceship space signal range;
- $a \in \mathbb{Z}$: Radius increase value.

Decrease player space signal range

$$\textit{decrease\_player\_space\_signal\_range}(p, a) = p' \quad \text{such that} \quad p' = p \quad \text{with} \quad p'_r = \max(0, p_r - a)$$

Decreases the radius p_r of space signals emitted by the player, by amount a.

- $p_r \in \mathbb{Z}$: Player spaceship space signal range;
- $a \in \mathbb{Z}$: Radius decrease value.

5.2.2 Constraints State

The set of constraint states is:

$$C = \{(h_{\text{max}}, f_{\text{max}}, s_{\text{max}}, t_{\text{max}}, m_{\text{max}}, b_{\text{max}}, d_{\text{max}})\}$$

Where:

- $h_{\text{max}} \in \mathbb{Z}$: Maximum player health;
- $f_{\text{max}} \in \mathbb{Z}$: Maximum player fuel;
- $s_{\text{max}} \in \mathbb{Z}$: Maximum code characters allowed;
- $t_{\text{max}} \in \mathbb{Z}$: Maximum number of execution steps the JavaScript interpreter can take;
- $m_{\text{max}} \in \mathbb{Z}$: Maximum memory bytes that can be allocated in the JavaScript interpreter memory.
- $b_{\text{max}} \in \mathbb{Z}$: Maximum runtime the JavaScript interpreter can take (per level session on the game interpreter, but per executed code on the playground console interpreter);
- $d_{\text{max}} \in \mathbb{Z}$: Maximum time allowed (in minutes).

Constraints State Manager Singleton

The `ConstraintsStateManager` singleton holds all the data about the level constraints such as the max player health, max interpreter steps, and max interpreter memory bytes. Every constraint is defined at the start of the level in this singleton's state dictionary with the `initialize()` function. Once the data is set on the dictionary, the game can only retrieve the values from the singleton, without performing any changes (it becomes read-only).

Constraints State Functions

If $c \in C$, we also define the following functions:

Max health

$$\textit{max\_health}(c) = c_{h_{\text{max}}}$$

Returns the max player health $c_{h_{\text{max}}}$ from the constraints state c.

- $c_{h_{\text{max}}} \in \mathbb{Z}$: Max player health.

Update max health

$$\textit{update\_max\_health}(c, v) = c' \quad \text{such that} \quad c' = c \quad \text{with} \quad c'_{h_{\text{max}}} = v$$

Updates the max player health $c_{h_{\text{max}}}$ in the constraints state c to the value v.

- $c_{h_{\text{max}}} \in \mathbb{Z}$: Max player health;
- $v \in \mathbb{Z}$: New max health value.

Max fuel

$$\textit{max\_fuel}(c) = c_{f_{\text{max}}}$$

Returns the max player fuel from the constraints state c.

- $c_{f_{\text{max}}} \in \mathbb{Z}$: Max player fuel.

Update max fuel

$$\textit{update\_max\_fuel}(c, v) = c' \quad \text{such that} \quad c' = c \quad \text{with} \quad c'_{f_{\text{max}}} = v$$

Updates the max player fuel $c_{f_{\text{max}}}$ in the constraints state c to the value v.

- $c_{f_{\text{max}}} \in \mathbb{Z}$: Max player fuel;
- $v \in \mathbb{Z}$: New max fuel value.

Max code characters

$$\textit{max\_code\_characters}(c) = c_{s_{\text{max}}}$$

Returns the maximum number of code characters allowed $c_{s_{\text{max}}}$, from the constraints state c.

- $c_{s_{\text{max}}} \in \mathbb{Z}$: Max code characters.

Update max code characters

$$\textit{update\_max\_code\_characters}(c, v) = c' \quad \text{such that} \quad c' = c \quad \text{with} \quad c'_{s_{\text{max}}} = v$$

Updates the maximum number of code characters $c_{s_{\text{max}}}$ in the constraints state c to the value v.

- $c_{s_{\text{max}}} \in \mathbb{Z}$: Max code characters;
- $v \in \mathbb{Z}$: New max code characters value.

Max interpreter steps

$$\textit{max\_steps}(c) = c_{t_{\text{max}}}$$

Returns the number of maximum interpreter steps allowed $c_{t_{\text{max}}}$, from the constraints state c.

- $c_{t_{\text{max}}} \in \mathbb{Z}$: Max interpreter steps.

Update max interpreter steps

$$\textit{update\_max\_steps}(c, v) = c' \quad \text{such that} \quad c' = c \quad \text{with} \quad c'_{t_{\text{max}}} = v$$

Updates the maximum number of interpreter steps allowed $c_{t_{\text{max}}}$ in the constraints state c to the value v.

- $c_{t_{\text{max}}} \in \mathbb{Z}$: Max interpreter steps;
- $v \in \mathbb{Z}$: New max interpreter steps value.

Max interpreter runtime

$$\textit{max\_runtime}(c) = c_{b_{\text{max}}}$$

Returns the maximum interpreter runtime allowed $c_{b_{\text{max}}}$, from the constraints state c.

- $c_{b_{\text{max}}} \in \mathbb{Z}$: Max interpreter runtime.

Update max interpreter runtime

$$\textit{update\_max\_runtime}(c, v) = c' \quad \text{such that} \quad c' = c \quad \text{with} \quad c'_{b_{\text{max}}} = v$$

Updates the maximum interpreter runtime allowed $c_{b_{\text{max}}}$ in the constraints state c to the value v.

- $c_{b_{\text{max}}} \in \mathbb{Z}$: Max interpreter runtime;
- $v \in \mathbb{Z}$: New max interpreter runtime value.

Max interpreter memory bytes

$$\textit{max\_memory\_bytes}(c) = c_{m_{\text{max}}}$$

Returns the maximum interpreter memory bytes allowed $c_{m_{\text{max}}}$, from the constraints state c.

- $c_{m_{\text{max}}} \in \mathbb{Z}$: Max interpreter memory bytes.

Update max interpreter memory bytes

$$\textit{update\_max\_memory\_bytes}(c, v) = c' \quad \text{such that} \quad c' = c \quad \text{with} \quad c'_{m_{\text{max}}} = v$$

Updates the maximum interpreter memory bytes allowed in the constraints state c to the value v.

- $c_{m_{\text{max}}} \in \mathbb{Z}$: Max interpreter memory bytes;
- $v \in \mathbb{Z}$: New max interpreter memory bytes value.

Max minutes

$$\textit{max\_minutes}(c) = c_{d_{\text{max}}}$$

Returns the maximum level duration in minutes, from the constraints state c.

- $c_{d_{\text{max}}} \in \mathbb{Z}$: Max level duration in minutes.

Update max minutes

$$\textit{update\_max\_minutes}(c, v) = c' \quad \text{such that} \quad c' = c \quad \text{with} \quad c'_{d_{\text{max}}} = v$$

Updates the maximum level duration $c_{d_{\text{max}}}$ in the constraints state c to v minutes.

- $c_{d_{\text{max}}} \in \mathbb{Z}$: Max level duration in minutes;
- $v \in \mathbb{Z}$: New max level duration value.

5.2.3 Console State

The set of console states is:

$$L = \{(f, h)\}$$

Where:

- $f \in \mathbb{D}$: A dictionary that indicates if each log filter is enabled or disabled. Its keys are a, s, g, p, v, and u, corresponding respectively to Action, Sensor, Game object, Space signal, Score, and Final logs; each value is a boolean.
- $h \in [\mathbb{D}]$: A list of dictionaries, where each dictionary represents a console log.

The dictionary f is defined as:

$$f = \{(a, s, g, p, v, u)\}$$

Where:

- $a \in \mathbb{B}$ – Action logs are to be displayed in the console;
- $s \in \mathbb{B}$ – Sensor logs are to be displayed in the console;
- $g \in \mathbb{B}$ – Game object logs are to be displayed in the console;
- $p \in \mathbb{B}$ – Space signal logs are to be displayed in the console;
- $v \in \mathbb{B}$ – Score logs are to be displayed in the console;
- $u \in \mathbb{B}$ – Final logs are to be displayed in the console.

The list h contains the following properties for each console log:

$$h = \{(i, m, c, t)\}$$

Where:

- $i \in \mathbb{Z}$ – Step Id;
- $m \in \mathbb{T}$ – Log message;
- $c \in \mathbb{T}$ – Log color;
- $t \in \mathbb{T}$ – Log type.

Console State Manager Singleton

The `ConsoleStateManager` singleton manages the game console data. It provides functions to add console logs of each type, enable and disable log filters, delete console logs, and list the existing console logs according to the selected filters.

Console logs are represented in the singleton as dictionaries, that contain the id of the step in which they were created, their color, their type and their strings to be displayed in the console. Each time a console log is created, it is added to the console logs list in the console state dictionary.

Filters are represented as boolean values, that when turned `false`, remove all logs associated to the filter from the console and when turned `true`, re-add all the logs of that type back.

Console State Functions

If $l \in L$, we also define the following functions:

Filters

$$\mathit{filters}(l) = l_f$$

Returns the filters dictionary l_f from the console state l, containing information about which filters are enabled or disabled.

- $l_f \in \mathbb{D}$: Filters.

Toggle filter

$$\textit{toggle\_filter}(l, n) = l' \quad \text{such that} \quad l' = l \quad \text{with} \quad l'_{f,n} = \neg l_{f,n}$$

Toggles the state of the filter with name n in the filters dictionary l_f to active or inactive.

- $n \in \mathbb{T}$: Filter name;
- $l_{f,n} \in \mathbb{B}$: Current filter value.

Console logs

$$\textit{console\_logs}(l) = l_h$$

Returns the list of all console logs l_h recorded up to the current step of execution, from the console state l.

- $l_h \in [\mathbb{D}]$: List of console logs.

Filtered console logs

$$\textit{filtered\_console\_logs}(l) = \{\, c \in l_h \mid l_f(t_c) = \textit{true} \,\}$$

Returns the list of console logs x that pass the enabled filters. Here, t_c is the type of the console log c, and $l_f(t_c) \in \mathbb{B}$ indicates whether logs of that type are enabled.

- $x \in [\mathbb{D}]$: List of filtered console logs;
- $c \in \mathbb{D}$: A console log;
- $t_c \in \mathbb{T}$: Type of console log c.

Add console log

$$\textit{add\_console\_log}(l, m) = l' \quad \text{such that} \quad l' = l \quad \text{with} \quad l'_h = l_h + m$$

Adds the console log m to the list of console logs l_h.

- $l_h \in [\mathbb{D}]$: Current console logs list;
- $m \in \mathbb{D}$: Console log to be added, having the properties:
  - $m_i \in \mathbb{Z}$: Step Id;
  - $m_m \in \mathbb{T}$: Log message;
  - $m_c \in \mathbb{T}$: Log color;
  - $m_t \in \mathbb{T}$: Log type.

Delete console logs by step

$$\textit{delete\_console\_logs\_by\_step}(l, i) = l' \quad \text{such that} \quad l'_h = \{\, m \in l_h \mid m_i \neq i \vee m_t = f_u \,\}$$

Removes all console logs created at step i from the logs list l_h, except for logs of the final type f_u. This is used when a step is rewinded, ensuring that logs from future steps do not remain in the console.

- $i \in \mathbb{Z}$: Id of the step to be removed;
- $m \in \mathbb{D}$: A console log under consideration;
- $l_h \in [\mathbb{D}]$: The current list of console logs;
- $l'_h \in [\mathbb{D}]$: The updated list after removal;
- $f_u \in \mathbb{T}$: The designated final log type.

5.2.4 Steps State

The set of steps states is:

$$X = \{(g, i, c, o, a, r, f, m, t, v)\}$$

Where:

- $g \in \mathbb{Z}$: Number of steps that have been interpreted by the interpreter. This value increases every time a new step is interpreted and stays the same when the player rewinds or advances to steps that were already interpreted;
- $i \in \mathbb{Z}$: Current step id. When the player advances or rewinds the state of the game, this value will also increase or decrease respectively, to always indicate what is the step where the player is currently at;
- $c \in \mathbb{D}$: List containing the dictionaries with data of each interpreted step by the interpreter;
- $o \in \mathbb{B}$: Game is executing one step;
- $a \in \mathbb{B}$: Game is executing all steps in a certain direction (rewinding/advancing);
- $r \in \mathbb{B}$: Game state is being rewinded;
- $f \in \mathbb{B}$: Game is fetching every interpreter step, not only the steps at the end of each statement;
- $m \in \mathbb{B}$: Level is in a finished state, which can be victory or game over;
- $t \in \mathbb{Z}$: Id of the execution step where the mission was terminated;
- $v \in \mathbb{B}$: Player finished the level with victory.

Steps State Manager Singleton

The `StepsStateManager` singleton exists for the purpose of managing all execution flow operations and each step data, having also helpful flags that indicate in which state of execution the game is currently situated.

During execution, the data of each interpreted step is stored in a completed steps dictionary and each step is represented as a separate dictionary.

This singleton has various boolean properties that determine the flow of the game, such as to indicate if the game state is advancing one step, advancing all steps, rewinding, fetching sub-steps, if the mission was terminated or if the objective of the level was achieved.

There are also two integer properties that keep track of the current step id in the interpreter and in the game.

Steps State Functions

If $x \in X$, we also define the following functions:

Update steps state

$$\textit{update\_steps\_state}(x, v) = x' \quad \text{such that} \quad x' = v$$

Updates the steps state x to the value v.

- $v \in X$: New steps state value.

Fetch previous step data

$$\textit{fetch\_previous\_step\_data}(x) = x_{c,k} \quad \text{such that}$$

$$x_{c,k} = \begin{cases} x_{c,j} & \text{if } j = \max\{0 \leq i < x_i \ \wedge \ (\neg x_{c,i,s} \ \vee \ x_f)\} \\ \texttt{null} & \text{if no such } j \text{ exists} \end{cases}$$

Returns the data for the previous execution step x_{c_k} from the completed steps list x_c, starting from the current step id x_i. If the player does not want to see substeps, the execution will be rewinded until a full step is found. If the player wants to see substeps ($x_f = \textit{true}$), the function stops early and just returns the previous substep $i-1$. If the first step is reached, no more rewinding will be done, and the current step is returned.

- $x_i \in \mathbb{Z}$: Current step id;
- $x_{c,k} \in \mathbb{D}$: Previous step data;
- $x_{c,j,s} \in \mathbb{B}$: Completed step x_{c_j} is a substep;
- $x_f \in \mathbb{B}$: Interpreter is fetching substeps.

Fetch next step data

$$\textit{fetch\_next\_step\_data}(x) = x_{c_k} \quad \text{such that}$$

$$x_{c,k} = \begin{cases} x_{c,j} & \text{if } j = \min\{x_i < i < x_c \wedge (\neg x_{c,i,s} \vee x_f \vee x_{c,i,t})\} \\ \texttt{null} & \text{if no such } j \text{ exists} \end{cases}$$

Returns the data for the next valid step $x_{c,k}$ from the completed steps list x_c, starting from the current step id x_i, and iterating forward until a valid non-sub-step is found or the end of the list is reached. If the current step is the most recent step and the mission is not terminated, the interpreter performs an external step-forward operation. The iteration halts early if the interpreter is fetching sub-steps ($x_f = \text{true}$) or if the step execution is terminated ($x_{c,i,t} = \text{true}$).

- $x_i \in \mathbb{Z}$: Current step id;
- $x_{c,k} \in \mathbb{D}$: Next valid step data;
- $x_{c,i,s} \in \mathbb{B}$: Completed step x_i is a sub-step;
- $x_f \in \mathbb{B}$: Interpreter is fetching sub-steps;
- $x_{c,i,t} \in \mathbb{B}$: Execution was terminated at completed step x_i.

Interpreter steps count

$$\textit{interpreter\_steps\_count}(x) = x_g$$

Returns the interpreter steps count x_g, from the steps state x.

- $x_g \in \mathbb{Z}$: Interpreter steps count.

Increment interpreter steps count

$$\textit{increment\_interpreter\_steps\_count}(x) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_g = x_g + 1$$

Increments the interpreter steps count x_g by one.

- $x_g \in \mathbb{Z}$: Interpreter steps count.

Current step id

$$\textit{current\_step\_id}(x) = x_i$$

Returns the current step id x_i.

- $x_i \in \mathbb{Z}$: Current step id.

Update current step id

$$\textit{update\_current\_step\_id}(x, v) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x_i = v$$

Updates the current step id x_i to value v.

- $x_i \in \mathbb{Z}$: Current step id;
- $v \in \mathbb{Z}$: New step id value.

Increment current step id

$$\textit{increment\_current\_step\_id}(x) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_i = x_i + 1$$

Increments the current step id x_i by one.

- $x_i \in \mathbb{Z}$: Current step id.

Decrement current step id

$$\textit{decrement\_current\_step\_id}(x) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_i = x_i - 1$$

Decrements the current step id by one.

- $x_i \in \mathbb{Z}$: Current step id.

Check current step is first

$$\textit{current\_step\_is\_first}(x) = c \quad \text{such that} \quad c = \begin{cases} \textit{true} & \text{if } x_i = 0 \\ \textit{false} & \text{otherwise} \end{cases}$$

Checks if the current step is the first.

- $x_i \in \mathbb{Z}$: Current step id;
- $c \in \mathbb{B}$: Boolean which indicates if the current step is the first.

Check current step is the most recent

$$\textit{current\_step\_is\_most\_recent}(x) = c \quad \text{such that} \quad c = \begin{cases} \text{true} & \text{if } x_i = x_g \vee (x_m \wedge x_i = x_t) \\ \text{false} & \text{otherwise} \end{cases}$$

Checks if the current step is the most recent.

- $x_m \in \mathbb{B}$: Mission terminated;
- $x_i \in \mathbb{Z}$: Current step id;
- $x_t \in \mathbb{Z}$: Id of the step where the mission was terminated;
- $x_g \in \mathbb{Z}$: Current interpreter step id;
- $c \in \mathbb{B}$: Current step is the most recent.

Check current step is last

$$\textit{current\_step\_is\_last}(x) = c \quad \text{such that}$$

$$c = \begin{cases} \textit{true} & \text{if } \textit{mission\_terminated}(x) \wedge \textit{current\_step\_is\_most\_recent}(x) \\ \textit{false} & \text{otherwise} \end{cases}$$

Checks if the current step is the last.

- $x_i \in \mathbb{Z}$: Current step id;
- $c \in \mathbb{B}$: Boolean which indicates if the current step is the last.

Check player reached max steps

$$\textit{reached\_max\_steps}(x) = r \quad \text{such that}$$

$$r = \begin{cases} \textit{true} & \text{if } \textit{current\_step\_id}(x) = \textit{max\_game\_interpreter\_steps}() \\ \textit{false} & \text{otherwise} \end{cases}$$

Checks if the player has reached the maximum steps limit, defined in the constraints state.

- $r \in \mathbb{B}$: Boolean which indicates if the player has reached the maximum steps limit.

Completed steps

$$\textit{completed\_steps}(x) = x_c$$

Returns the list of completed steps x_c from steps state x.

- $x_c \in [\mathbb{D}]$: List of completed steps data.

Completed step by id

$$\textit{completed\_step\_by\_id}(x, i) = \begin{cases} x_{c,i} & \text{if } i \geq 0 \\ \texttt{null} & \text{otherwise} \end{cases}$$

Returns the completed step that has id i, from the completed steps list.

- $x_{c,i} \in \mathbb{D}$: Completed step with id i;
- $i \in \mathbb{Z}$: Completed step id.

Current completed step

$$\textit{current\_completed\_step}(x) = x_{c,x,i}$$

Returns the completed step that has the same id as the current step.

- $x_i \in \mathbb{Z}$: Current step id;
- $x_{c,x,i} \in \mathbb{D}$: Current step data.

Step runtime by id

$$\textit{step\_runtime\_by\_id}(x, i) = \begin{cases} x_{c,i,r} & \text{if } i \geq 0 \\ \texttt{null} & \text{otherwise} \end{cases}$$

Returns the interpreter runtime of the completed step with id i.

- $i \in \mathbb{Z}$: Completed step id;
- $x_{c,i,r} \in \mathbb{Z}$: Interpreter runtime of step with id i.

Add completed step

$$\textit{add\_completed\_step}(x, s) = x' \quad \text{such that} \quad x'_c = x_c + s$$

Adds the step data s to the list of completed steps x_c.

- $x_c \in [\mathbb{D}]$: List of completed steps data;
- $s \in \mathbb{D}$: New step data.

Playing one step

$$\textit{playing\_one\_step}(x) = x_o$$

Returns the value of the boolean property x_o (playing one step), which is true if the game is currently advancing or rewinding a step.

- $x_o \in \mathbb{B}$: Playing one step.

Update playing one step

$$\textit{set\_playing\_one\_step}(x, v) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_o = v$$

Updates the boolean property x_o (playing one step) to the value v.

- $x_o \in \mathbb{B}$: Playing one step;
- $v \in \mathbb{B}$: New boolean value.

Playing all steps

$$\textit{playing\_all\_steps}(x) = x_a$$

Returns the boolean property x_a (playing all steps), which is true if the player pressed the buttons to advance or rewind all steps.

- $x_a \in \mathbb{B}$: Playing all steps.

Update playing all steps

$$\textit{update\_playing\_all\_steps}(x, v) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_a = v$$

Updates the boolean property x_a (playing all steps) to the value v.

- $x_a \in \mathbb{B}$: Playing all steps;
- $v \in \mathbb{B}$: New boolean value.

Rewinding

$$\textit{rewinding}(x) = x_r$$

Returns the boolean property x_r (rewinding), which is true if the player pressed buttons to rewind the execution.

- $x_r \in \mathbb{B}$: Rewinding.

Update rewinding

$$\textit{update\_rewinding}(x, v) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_r = v$$

Updates the boolean property x_r (rewinding) to the value v.

- $x_r \in \mathbb{B}$: Rewinding;
- $v \in \mathbb{B}$: New boolean value.

Fetching interpreter sub steps

$$\textit{fetching\_interpreter\_sub\_steps}(x) = x_f$$

Returns the boolean property x_f (fetching interpreter sub steps).

- $x_f \in \mathbb{B}$: Fetching interpreter sub steps.

Update fetching interpreter sub steps

$$\textit{update\_fetching\_interpreter\_sub\_steps}(x, v) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_f = v$$

Updates the boolean property x_f (fetching interpreter sub steps) to the value v.

- $x_f \in \mathbb{B}$: Fetching interpreter sub steps;
- $v \in \mathbb{B}$: New boolean value.

Mission terminated

$$\textit{mission\_terminated}(x) = x_m$$

Returns the value of the boolean property x_m (mission terminated), which indicates if the mission is over or not.

- $x_m \in \mathbb{B}$: Mission terminated.

Update mission terminated

$$\textit{update\_mission\_terminated}(x, v) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_m = v$$

Updates the boolean property x_m (mission terminated) to the value v.

- $x_m \in \mathbb{B}$: Mission terminated;
- $v \in \mathbb{B}$: New boolean value.

Objective achieved

$$\textit{objective\_achieved}(x) = x_v$$

Returns the boolean property x_v (objective achieved), which indicates if the mission was completed successfully or not.

- $x_v \in \mathbb{B}$: Objective achieved.

Update objective achieved

$$\textit{update\_objective\_achieved}(x, v) = x' \quad \text{such that} \quad x' = x \quad \text{with} \quad x'_v = v$$

Updates the boolean property x_v (objective achieved) to the value v.

- $x_v \in \mathbb{B}$: Objective achieved;
- $v \in \mathbb{B}$: New boolean value.

5.2.5 Game Objects State

The set of game object states is:

$$G = \{(c, k, a, d, j, r, l)\}$$

Where:

- $c \in \mathbb{D}$: The dictionary representing the state of coins;
- $k \in \mathbb{D}$: The dictionary representing the state of keycards;
- $a \in \mathbb{D}$: The dictionary representing the state of asteroids;

- $d \in \mathbb{D}$: The dictionary representing the state of doors;
- $j \in \mathbb{D}$: The dictionary representing the state of jerrycans;
- $r \in \mathbb{D}$: The dictionary representing the state of repair items;
- $l \in \mathbb{D}$: The dictionary representing the state of lasers.

Game Objects State Manager Singleton

The `GameObjectsStateManager` singleton is responsible for managing the state of every game object inside the level. The state manager has functions to get values from the state, perform operations on any game object (*e.g.* open, press, collect, destroy) and to check on every step if any game object state can be changed.

Game Objects State Properties

The game objects state dictionary contains every game object section dictionary. Each section dictionary key is the game object type in plural (*e.g.* `coins`, `doors`, `keycards`) and inside there are the info dictionaries of each created game object of that type. Inside each game object dictionary can exist many different properties, depending on the type of game object, that will completely define the game object state on the level (*e.g.* `position`, `pressed`, `opened`, `destroyed`, `xpCounted`). The key for every game object dictionary entry is defined by the current number of game objects in the section.

These are the possible properties for the game objects:

- $x \in \mathbb{Z}$: Game object position on the X-axis;
- $y \in \mathbb{Z}$: Game object position on the Y-axis;
- $u \in \mathbb{B}$: Collectible game object is collected;
- $d \in \mathbb{B}$: Destroyable game object is destroyed;
- $e \in \mathbb{B}$: Game object experience has already been counted on the player profile;
- $p \in \mathbb{B}$: Pressable game object is pressed;
- $b \in \mathbb{B}$: Pressable game object is currently being pressed;
- $w \in \mathbb{B}$: Openable game object is a switch;
- $o \in \mathbb{B}$: Openable game object is open;
- $m \in \mathbb{B}$: Openable game object can be opened manually;
- $z \in \mathbb{T}$: Game object color;
- $r \in \mathbb{Z}$: Game object radius;
- $g \in \mathbb{Z}$: Obstacle game object damage;
- $h \in \mathbb{Z}$: Health that collectible game object replenishes;
- $f \in \mathbb{Z}$: Fuel that collectible game object replenishes;

- $s \in \mathbb{Z}$: Score that game object can give;
- $n \in \mathbb{Z}$: Angle in degrees where the game object is heading;
- $l \in \mathbb{B}$: Game object sprite is displayed horizontally;
- $t \in \mathbb{T}$: Value contained inside crate game object;
- $q \in \mathbb{D}$: Game object requirements.

Game Objects State Functions

If $g \in G$, we also define the following functions:

Check game objects state has section

$$\textit{game\_objects\_state\_has\_section}(o, n) = a \quad \text{such that} \quad a = \textit{true} \text{ if } n \in G$$

a is true if there is a game objects section with name n.

- $n \in \mathbb{T}$: Section name;
- $a \in \mathbb{B}$: Game Objects state has section.

Section by name

$$\textit{section\_by\_name}(o, n) = s \quad \text{such that} \quad s = \begin{cases} o_n & \text{if } n \in o \\ \texttt{null} & \text{otherwise} \end{cases}$$

Returns the section with name n from the game objects state o.

- $n \in \mathbb{T}$: Section name;
- $s \in \mathbb{D}$: Section.

Game object

$$\textit{game\_object}(o, n, i) = o_{n,i}$$

Returns the game object with id i from the section n of the game objects state o.

- $n \in \mathbb{T}$: Section name;
- $i \in \mathbb{Z}$: Game object Id;
- $g \in \mathbb{D}$: Game object.

Game object by id

$$\textit{game\_object\_by\_id}(o, i) = g \quad \text{such that} \quad g = \begin{cases} o_{n,i} & \text{if there exists } n \in o \text{ such that } i \in o_n \\ \texttt{null} & \text{otherwise} \end{cases}$$

Returns the game object with id i from the game objects state o.

- $i \in \mathbb{Z}$: Game object Id;
- $g \in \mathbb{D}$: Game object.

Update game object

$$\textit{update\_game\_object}(o, n, i, v) = o' \quad \text{such that} \quad o' = \begin{cases} o \text{ with } o_{n,i} = v & \text{if } n \in o \land i \in o_n \\ o & \text{otherwise} \end{cases}$$

Updates the game object with id i from the section n to the value v.

- $n \in \mathbb{T}$: Section name;
- $i \in \mathbb{Z}$: Game object Id;
- $v \in \mathbb{D}$: New game object value.

Game objects with attribute

$$\textit{game\_objects\_with\_attribute}(o, a) = l \quad \text{such that}$$

$$l = \{g \in o_{n,i} \mid n \in o, i \in o_n, a \in \text{attributes}(g)\}, \forall n : o_n \in o$$

Returns the list of game objects with the attribute a.

- $a \in \mathbb{T}$: Attribute name;
- $l \in [\mathbb{D}]$: Game objects list;
- attributes(g): Set of attributes of game object g;
- $n \in o$: Section name.

Check requirement fulfilled

$$\textit{requirement\_fulfilled}(o, r) = f \quad \text{such that} \quad f = \begin{cases} \textit{true} & \text{if } r_i = \text{null or } r_p = c \lor r_v = g_{r,b} \\ \textit{false} & \text{otherwise} \end{cases}$$

Checks if the requirement r is fulfilled.

- $r \in \mathbb{D}$: Requirement;
- $r_i \in \mathbb{Z}$: Requirement id;
- $r_p \in \mathbb{T}$: Requirement property name;
- $r_v \in \mathbb{T}$: Requirement property value;
- $g_{r,b} \in \mathbb{T}$: Contained value in game object g;
- $f \in \mathbb{B}$: Requirement is fulfilled.

Check requirements fulfilled

$$\textit{requirements\_fulfilled}(o, l) = f \quad \text{such that}$$

$$f = \begin{cases} \textit{true} & \text{if } \forall r \in l, \textit{requirement\_fulfilled}(o, r) = \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}$$

Checks if every requirement in l is fulfilled.

- $l \in \mathbb{D}$: Requirements;
- $f \in \mathbb{B}$: Requirements are fulfilled.

Update game objects

$$\textit{update\_game\_objects}(o) = o' \quad \text{such that} \quad o' = f(o)$$

Updates the state of any game object that is in interaction with the player (being touched, pressed, collected, opened or receiving a space signal) in the current step.

- $f : G \to G$: A function that applies updates based on spaceship interactions with game objects (e.g., touched, pressed, collected, opened, or receiving signals).

Collect Game Object

$$\textit{trigger\_collect}(o, n, g) = o' \quad \text{such that}$$

$$o' = \begin{cases} o_{n,g} & \text{if } g \in \textit{game\_objects\_with\_attribute}(o, Collectible) \wedge (\neg g_c \vee \neg g_d) \\ o & \text{otherwise} \end{cases}$$

Updates the game object g as collected (used in keycards, coins, and jerrycans) and sends a signal to the game indicating that the game object was collected. If the execution is currently being rewinded, the operation is reverted;

- $n \in \mathbb{T}$: Section name;

- $g \in \mathbb{D}$: Game object;
- $g_c \in \mathbb{B}$: Game object g is collected;
- $g_d \in \mathbb{B}$: Game object g is destroyed.

Touch game object

$$\textit{trigger\_touch}(o, n, g) = o' \quad \text{such that}$$

$$o' = \begin{cases} o_{n,g} & \text{if } g \in \textit{game\_objects\_with\_attribute}(o, Obstacle) \wedge \neg g_d \\ o & \text{otherwise} \end{cases}$$

Handles a player touch on game object g (used in obstacles). If the program execution is currently being rewinded, the operation is reverted;

- $n \in \mathbb{T}$: Section name;
- $g \in \mathbb{D}$: Game object;
- $g_d \in \mathbb{B}$: Game object g is destroyed.

Press game object

$$\textit{trigger\_press}(o, n, g) = o' \quad \text{such that}$$

$$o' = \begin{cases} o_{n,g} & \text{if } g \in \textit{game\_objects\_with\_attribute}(o, Pressable) \wedge (\neg g_d \vee g_p) \\ o & \text{otherwise} \end{cases}$$

Updates the game object g as pressed (used in buttons). If the program execution is currently being rewinded, the operation is reverted;

- $n \in \mathbb{T}$: Section name;
- $g \in \mathbb{D}$: Game object;
- $g_p \in \mathbb{B}$: Game object g is pressed.

Open game object

$$\textit{trigger\_open}(o, n, g) = o'$$

$$o' = \begin{cases} o_{n,g} & \text{if } g \in \textit{game\_objects\_with\_attribute}(o, Openable) \wedge \neg g_o \\ o & \text{otherwise} \end{cases}$$

Updates the game object g as opened (used in keycards, coins, and jerrycans) and sends a signal to the game indicating that the game object was opened. If the program execution is currently being rewinded, the operation is reverted;

- $n \in \mathbb{T}$: Section name;
- $g \in \mathbb{D}$: Game object;
- $g_o \in \mathbb{B}$: Game object g is opened.

Destroy game object

$$\textit{trigger\_destroy}(o, n, g) = o'$$

$$o' = \begin{cases} o_{n,g} & \text{if } g \in \textit{game\_objects\_with\_attribute}(o, Destroyable) \land \neg g_d \\ o & \text{otherwise} \end{cases}$$

Sets the game object g as destroyed (used in asteroids, the player, and the target). If the program execution is currently being rewinded, the operation is reverted;

- $n \in \mathbb{T}$: Section name;
- $g \in \mathbb{D}$: Game object;
- $g_d \in \mathbb{B}$: Game object g is destroyed.

Count game object xp

$$\textit{count\_game\_object\_xp}(o, n, g) = s'$$

$$o' = \begin{cases} o_{n,g} & \text{if } g \in \textit{game\_objects\_with\_attribute}(o, XP) \land \neg g_e \\ o & \text{otherwise} \end{cases}$$

Increments the xp of the player in the main state m by the xp points specified in the game object g. The player xp is a value that only goes up and cannot ever be rewinded, so the `xpCounted` flag was added to make sure that the xp of this particular game object is not counted twice when rewinding and advancing.

- $n \in \mathbb{T}$: Section name;
- $g \in \mathbb{D}$: Game object;
- $g_e \in \mathbb{B}$: Game object g XP is counted.

Collected game objects count by section

$$\textit{collected\_game\_objects\_count\_by\_section}(o, n) = c$$

Returns the number of collected game objects that are in section n.

- $n \in \mathbb{T}$: Section name;

- $c \in \mathbb{Z}$: Collected game objects count.

Game object type count

$$\textit{game\_objects\_in\_section}(o, n) = c$$

Returns the number of game objects that are in section n.

- $n \in \mathbb{T}$: Section name;
- $c \in \mathbb{Z}$: Game objects count.

Check game object can be changed

$$\textit{can\_be\_changed}(o, g) = c$$

Returns the boolean property c which is *true* if the game object g can be changed in the current execution step.

- $n \in \mathbb{T}$: Section name;
- $c \in \mathbb{B}$: Game object g can be changed.

5.2.6 Space Signals State

The set of space signals states is:

$$E = \{(s, d)\}$$

Where:

- $s \in \mathbb{Z}$: Step ID;
- $d \in \mathbb{D}$: Dictionary of space signals for step s.

The dictionary D for each step contains entries corresponding to individual space signals:

$$D = \{(i, p) : i \in \mathbb{Z}, p \in \mathbb{D}\}$$

Where:

- $i \in \mathbb{Z}$: Space signal ID;
- $p \in \mathbb{D}$: Dictionary containing the properties of the space signal.

Space Signals State Manager Singleton

The `SpaceSignalsStateManager` singleton has functions that manage the level space signals, such as getting, creating, enabling, and disabling space signals.

Space Signals State Properties

The space signals dictionary contains as entries one dictionary per step and within each step Dictionary there are the info dictionaries of each space signal that was created in that step. A space signal info dictionary has information on the position where the space signal was created, the range, if it is currently being emitted, and the value of its message.

$$E = \{(s_i, \{s_{i,j} : j \in J_i\}) : i \in I,\, j \in J_i\}$$

Where:

- $i \in I$: Step index, where I is the set of all step indices;
- J_i is the set of space signal indices associated with step i, and each step s_i contains a dictionary of space signals $s_{i,j}$;
- Each space signal $s_{i,j}$ is represented by the following properties:

  $$s_{i,j} = (x, y, v, r, e, c)$$

  Where:

  - $x \in \mathbb{Z}$: Position of the space signal on the x-axis;
  - $y \in \mathbb{Z}$: Position of the space signal on the y-axis;
  - $v \in \mathbb{T}$: Message value of the space signal;
  - $r \in \mathbb{Z}$: Range of the space signal;
  - $e \in \mathbb{B}$: Space signal is enabled;
  - $c \in \mathbb{B}$: Space signal is currently being emitted.

Space Signals State Functions

If $e \in E$, we also define the following functions:

Current step space signals

$$\textit{current\_step\_space\_signals}(e) = \begin{cases} e_i & \text{if } e \text{ has key } i \\ \emptyset & \text{otherwise} \end{cases}$$

Returns the dictionary of emitted space signals e_i in the current step i, from the space signals state e.

- $e_i \in \mathbb{D}$: Dictionary of emitted space signals in step i.

Space signal

$$\textit{space\_signal}(e, u, i) = e_{u,i}$$

Returns the dictionary of the space signal with id i emitted at step with id u.

- $u \in \mathbb{Z}$: Step id;
- $i \in \mathbb{Z}$: Space signal id;
- $n \in \mathbb{D}$: Space signal.

Space signal position

$$\textit{space\_signal\_position}(e, d) = \begin{pmatrix} p_x \\ p_y \end{pmatrix} \quad \text{such that} \quad p = e_{u,d}$$

Returns the space signal d position.

- $d \in \mathbb{D}$: Space signal;
- $p \in \mathbb{V}$: Space signal position.

Enable space signal

$$\textit{enable\_space\_signal}(e, u, i) = e' \quad \text{such that} \quad e'_{u,i,e} = \textit{true}$$

Enables the space signal with id i in step with id u

- $u \in \mathbb{Z}$: Step id;
- $i \in \mathbb{Z}$: Space signal id.

Disable space signal

$$\textit{disable\_space\_signal}(e, u, i) = e' \quad \text{such that} \quad e'_{u,i,e} = \textit{false}$$

Disables the space signal with id i in step with id u

- $u \in \mathbb{Z}$: Step id;
- $i \in \mathbb{Z}$: Space signal id.

Disable all space signals

$\textit{disable\_all\_space\_signals}(e) = e' \quad \text{such that} \quad \forall u \in e \land \forall i \in e_u, \quad \textit{disable\_space\_signal}(e, u, i)$

Disables every space signal.

- $u \in \mathbb{Z}$: Step id;
- $i \in \mathbb{Z}$: Space signal id.

Emit space signal

$$\textit{emit\_space\_signal}(e, v, p, j, r) = \begin{cases} e' = e & \text{if } r = \textit{false}, \\ e' = e \text{ with } e_{u,i} = \textit{false} \quad \forall i \in e_u & \text{otherwise} \end{cases}$$

Emits a space signal with v as message, on the 2d coordinates p, with a range of j. If r is true (rewinding), the action is reverted, so the space signal created on this step is removed.

- $v \in \mathbb{T}$: Space signal message;
- $p \in \mathbb{V}$: Position;
- $j \in \mathbb{Z}$: Range;
- $r \in \mathbb{B}$: Rewinding.

5.3 Game View

The game view is the part of the state s which is shown to the player.

If $s \in S$:

$$\begin{gathered} \textit{view}(s) = V_s = \\ = \textit{view\_P}(s_p) \times \textit{view\_G}(s_g) \times \textit{view\_C}(s_c) \times \textit{view\_L}(s_l) \times \\ \times \textit{view\_X}(s_x) \times \textit{view\_E}(s_e) \end{gathered}$$

Player state view

The view of the player state $g \in P$, includes the entire player state.

$$\textit{view\_P}(p) = p$$

Game Objects State View

The view of the game objects state $g \in G$, includes all game object sections (c, k, a, d, j, r, l), excluding the properties e, t, and n of each game object o within the sections.

$$\mathit{view\_G}(g) = \{(c', k', a', d', j', r', l') : c' = \{o \setminus \{e, t, n\} \mid o \in c\},\ k' = \{o \setminus \{e, t, n\} \mid o \in k\}, \dots\}$$

Constraints state view

The view of the constraints state $c \in G$, includes the entire constraints state.

$$\mathit{view\_C}(c) = c$$

Console state view

The view of the console state $l \in L$, includes all filters and includes the console logs which were created in the current or previous execution steps.

$$\mathit{view\_L}(l) = \big(f, \{(i, m, c, t) \in h : i \leq X.\mathit{current\_step\_id}(x)\}\big)$$

Steps state view

The view of the steps state $x \in X$, includes all properties except g (the number of interpreted steps) and t (the id of the termination step).

$$\mathit{view\_X}(x) = x \setminus \{g, t\}$$

Space signals state view

The view of the space signals state $e \in E$, includes all space signals that are enabled.

$$\mathit{view\_E}(e) = \{(s_i, \{s_{i,j} : j \in J_i\}) \in E : s_{i,j} = (x, y, v, r, e, c), e = \mathit{true}\}$$

- $v \in \mathbb{T}$: Space signal message;
- $p \in \mathbb{V}$: Position;
- $j \in \mathbb{Z}$: Range;
- $r \in \mathbb{B}$: Rewinding.

- $(s_i, \{s_{i,j} : j \in J_i\}) \in E$ refers to a step s_i containing a set of space signals $s_{i,j}$ for step i in the space signals state;
- $s_{i,j} = (x, y, v, r, e, c)$ represents a space signal in step i with the properties x, y, v, r, e, and c;
- $e = \mathit{true}$ filters only space signals that are currently enabled.

5.4 Game Setup

The game setup is the definition of the initial state of the game when the player just started the level.

$$\begin{gathered} \mathit{setup}() = s_0 = \\ = \mathit{setup\_P}() \times \mathit{setup\_G}() \times \mathit{setup\_C}() \times \mathit{setup\_L}() \times \\ \times \mathit{setup\_X}() \times \mathit{setup\_E}() \times \mathit{setup\_O}() \end{gathered}$$

- $s_0 \in S$: Initial state.

Setup global state properties

$$\begin{gathered} \mathit{setup\_O}() = o_0 = \\ = \mathit{initialize\_interpreter\_memory\_bytes}() \times \mathit{initialize\_game\_over\_message}() \times \\ \times \mathit{initialize\_interpreter\_runtime}() \times \mathit{initialize\_highlighted\_code\_range}() \times \\ \times \mathit{initialize\_score}() \times \mathit{initialize\_time}() \end{gathered}$$

Initializes the global state properties o;

- $o_0 \in \mathbb{D}$: Initial global state properties.

$$\mathit{initialize\_interpreter\_memory\_bytes}() = o_{b_0} = 0$$

Initializes the interpreter memory bytes count o_b as 0;

- $o_{b_0} \in \mathbb{B}$: Initial interpreter memory bytes count.

$$\mathit{initialize\_game\_over\_message}() = o_{y_0}$$

Initializes the game over message o_y as an empty string;

- $o_{y_0} \in \mathbb{T}$: Initial game over message.

$$\mathit{initialize\_interpreter\_runtime}() = o_{r_0} = 0$$

Initializes the interpreter runtime o_r count as 0;

- $o_{r_0} \in \mathbb{Z}$: Initial interpreter runtime.

$$\mathit{initialize\_time}() = \begin{cases} o_{u_0} = \mathit{C.max\_minutes}() & \wedge \\ o_{v_0} = 0 \end{cases}$$

Initializes the level timer minutes o_u, to the max minutes value in the constraints state dictionary, and initializes the seconds value o_v to 0;

- $o_{u_0} \in \mathbb{Z}$: Initial remaining minutes;
- $o_{v_0} \in \mathbb{Z}$: Initial remaining seconds of the current minute.

Setup player state

$$\textit{setup\_P}(i) = p_0 = (x_0, y_0, s_0, h_0, f_0, r_0, e_0, o_0, j_0)$$

such that

$$\begin{aligned} x_0, y_0 &= i_x, i_y, \\ s_0 &= i_s, \\ h_0 &= \textit{initialize\_player\_health}(), \\ f_0 &= \textit{initialize\_player\_fuel}(), \\ r_0 &= 0, \\ e_0 &= \textit{false}, \\ o_0 &= \textit{false}, \\ j_0 &= \textit{false}. \end{aligned}$$

Initializes the player state dictionary with the value i and sets the other properties to their default values;

- $p_0 \in P$: Initial player state;
- $i \in \mathbb{D}$: Player starting point data from the JSON level definition. It has the player position (i_x, i_y) and the player heading i_s.

Initialize player health

$$\textit{initialize\_player\_health}() = p_{h_0} = \textit{C.max\_player\_health}()$$

Initializes the player health p_h to the max player health value defined in the constraints.

- $p_{h_0} \in \mathbb{Z}$: Initial player health.

Initialize player fuel

$$\textit{initialize\_player\_fuel}() = p_{f_0} = \textit{C.max\_player\_fuel}()$$

Initializes the player spaceship fuel p_f to the max fuel value defined in the constraints.

- $p_{f_0} \in \mathbb{Z}$: Initial player fuel.

Setup Game Objects State

$$\textit{setup\_G}(i) = g_0, \quad \text{where } \forall s \in i,\ g_{s_0} = \textit{initialize\_section}(i_s)$$

Initializes the game objects state dictionary g with the value i.

- $g_0 \in G$: Initial game objects state dictionary;
- $i \in \mathbb{D}$: Data of each section of game objects in the JSON level definition.

Initialize Section

$$\textit{initialize\_section}(i_s) = g_{s_0} \quad \text{where } \forall o \in i_s,\ s_{o_0} = \textit{initialize\_game\_object}(o)$$

Initializes a section s using the input data i_s.

- $g_{s_0} \in \mathbb{D}$: Initial section state dictionary;
- $i_s \in \mathbb{D}$: Input data for the section, containing all objects and their properties;
- *initialize_game_object*(o): Initializes a single game object o.

Initialize Game Object

$$\textit{initialize\_game\_object}(o) = o_0$$

Initializes a single game object o.

- $o_0 \in \mathbb{D}$: Initial game object state dictionary;
- $o \in \mathbb{D}$: The input object with its attributes (e.g., position, type).

Setup constraints state

$$\textit{setup\_C}(i) = c_0 = (h_{\text{max}}, f_{\text{max}}, s_{\text{max}}, t_{\text{max}}, m_{\text{max}}, b_{\text{max}}, d_{\text{max}})$$

such that

$$
\begin{aligned}
h_{\text{max}} &= i_{h\,\text{max}}, \\
f_{\text{max}} &= i_{f\,\text{max}}, \\
s_{\text{max}} &= i_{s\,\text{max}}, \\
t_{\text{max}} &= i_{t\,\text{max}}, \\
m_{\text{max}} &= i_{m\,\text{max}}, \\
b_{\text{max}} &= i_{b\,\text{max}}, \\
d_{\text{max}} &= i_{d\,\text{max}}.
\end{aligned}
$$

Initializes the constraints state dictionary with the value i;

- $c_0 \in C$: Initial constraints state;
- $i \in \mathbb{D}$: Loaded constraints data from the JSON level definition. It has all the data needed to initialize each one of the constraints.

Setup console state

$$
\textit{setup\_L}() = l_0 = \begin{cases} \textit{initialize\_filters}() & \wedge \\ l_{h_0} = \emptyset \end{cases}
$$

Initializes the console state dictionary l.

- $l_0 \in L$: Initial console state;
- $l_{h_0} \in \mathbb{D}$: Initial console logs list

$$
\textit{initialize\_filters}() = l_{f_0} = (a_0, s_0, g_0, p_0, v_0, u_0)
$$

such that

$$
\begin{aligned}
a_0 &= \textit{true}, \\
s_0 &= \textit{true}, \\
g_0 &= \textit{true}, \\
p_0 &= \textit{true}, \\
v_0 &= \textit{true}, \\
u_0 &= \textit{true}
\end{aligned}
$$

Initializes the console filters l_f by setting all filters as enabled:

- $l_{f_0} \in \mathbb{D}$: Initial console filters

Setup steps state

$$\textit{setup\_X}() = x_0 = (g_0, i_0, c_0, o_0, a_0, r_0, f_0, m_0, t_0, v_0)$$

such that

$$\begin{aligned} g_0 &= 0, \\ i_0 &= 0, \\ c_0 &= \emptyset, \\ o_0 &= \textit{false}, \\ a_0 &= \textit{false}, \\ r_0 &= \textit{false}, \\ f_0 &= \textit{false}, \\ m_0 &= \textit{false}, \\ t_0 &= 0, \\ v_0 &= \textit{false}. \end{aligned}$$

- $x_0 \in X$: Initial steps state;
- $g_0 \in \mathbb{Z}$: Number of steps interpreted by the interpreter, initialized to 0;
- $i_0 \in \mathbb{Z}$: Current step ID, initialized to 0;
- $c_0 \in \mathbb{D}$: List of completed steps, initialized as an empty list;
- $o_0, a_0, r_0, f_0, m_0, v_0 \in \mathbb{B}$: Flags indicating execution states.

$$\textit{initialize\_interpreter\_steps\_count}() = x_{g_0} = 0$$

Initializes the interpreter steps count x_g as 0;

- $x_{g_0} \in \mathbb{Z}$: Initial interpreter steps count.

$$\textit{initialize\_current\_step\_id}() = x_{i_0} = 0$$

Initializes the current step id x_i as 0.

- $x_{i_0} \in \mathbb{Z}$: Initial step id.

$$\textit{initialize\_completed\_steps}() = x_{c_0} = \emptyset$$

Initializes the completed steps list x_c as an empty list.

- $x_{c_0} \in [\mathbb{D}]$: Initial completed steps list.

Setup space signals state

$$\mathit{setup\_E}() = e_0 = \emptyset$$

Initializes the space signals state dictionary e as empty:

- $e_0 \in E$: Initial space signals state.

5.5 Actions

Actions are functions that the player can utilize to transform the game state. This includes the spaceship or other game objects (*e.g.* moving the spaceship, pressing a button, etc.). The player can call the JavaScript functions that trigger these actions on the spaceship. These functions are supported by the game engine through the use of the JavaScript Bridge interface [Godb], which allows for the communication between the game and the browser's JavaScript context.

The player writes the code to beat the level, which contains the function calls to control the spaceship. After the player writes and submits their code, the game prepares the browser to execute it. To achieve this, the game sends through the JavaScript Bridge, to the browser, a string with the implementation of each game function, which includes the action and sensor functions. The browser then initializes a sandboxed JavaScript interpreter, JS-Interpreter, which isolates the code execution from the browser's context for security purposes. Finally, the JS-Interpreter receives and executes the code containing the implementation of each game function, and is then fully prepared to receive JavaScript code written by the player.

Fly

$$\mathit{fly}(s) = \mathit{action}(s, p, fly) = s' \quad \text{such that}$$
$$\begin{cases} s'_p = s_p + s_h \quad \wedge \\ s'_f = s_f - 1 \quad \text{if } s_f > 0, \\ s'_f = 0 \quad \text{otherwise} \end{cases}$$

The spaceship flies one coordinate in the direction that is headed. The ship loses one fuel point.

- $s' \in \mathbb{D}$: Updated State;
- $s_h \in \mathbb{Z}$: Heading direction of the spaceship;
- $s_f \in \mathbb{Z}$: Fuel of the spaceship;
- $s_p \in \mathbb{V}$: Position of the spaceship.

Turn Left

$$\mathit{turn\_left}(s) = \mathit{action}(s, p, turn\_left) = s' \quad \text{such that}$$

$$s'_h = (s_h + 90°) \mod 360° \qquad \text{(update heading counterclockwise)}$$

The spaceship turns 90 degrees counterclockwise.

- $s_h \in \mathbb{Z}$: Current heading of the spaceship.

Turn Right

$$\textit{turn\_right}(s) = \textit{action}(s, p, turn\_right) = s' \quad \text{such that}$$

$$s'_h = (s_h - 90° + 360°) \mod 360° \qquad \text{(update heading clockwise)}$$

The spaceship turns 90 degrees clockwise.

- $s_h \in \mathbb{Z}$: Current heading of the spaceship.

Open

$$\textit{open}(s) = \textit{action}(s, p, open) = s' \quad \text{such that}$$

$$\begin{cases} \forall g \in \text{nearby\_objects}(s_p), & \\ s'_{g,o} = \text{open} & \text{if } g_t = \text{openable} \wedge \textit{requirements\_fulfilled}(g, s), \\ s'_{g,o} = s_{g,o} & \text{otherwise} \end{cases}$$

Any openable game objects near the spaceship (*e.g.* crates, doors) will receive a request to be opened. If the player did not meet the openable's requirements, then the command will be ignored, and the openable game object will stay closed.

- $g_o \in \mathbb{B}$: Object is open;
- $g_t \in \mathbb{T}$: Type of object;
- $s_p \in \mathbb{V}$: Position of the spaceship.

Press

$$\textit{press}(s) = \textit{action}(s, p, press) = s' \quad \text{such that}$$

$$\begin{cases} \forall g \in \text{nearby\_objects}(s_p), & \\ s'_{g,p} = \text{pressed} & \text{if } g_t = \text{pressable}, \\ s'_{g,p} = s_{g,p} & \text{otherwise} \end{cases}$$

Any pressable game objects adjacent to the spaceship (*e.g.* buttons) will be pressed if they are within range.

- $g_p \in \mathbb{B}$: Object is pressed;
- $g_t \in \mathbb{T}$: Type of object (whether it can be pressed);
- $s_p \in \mathbb{V}$: Position of the spaceship.

Emit

$$\textit{emit}(s, m) = \textit{action}(s, p, emit, m) = s' \quad \text{such that}$$

$$s'_s = s_s \cup \{(s_p, m)\}$$

The spaceship emits a space signal with the message passed as argument of the function.

- $s_s \in [\mathbb{D}]$: List of signals emitted by the spaceship;
- $m \in \mathbb{T}$: Message to be emitted;
- $s_p \in \mathbb{V}$: Position of the spaceship.

5.6 Sensors

Sensors can be activated by the player to obtain information about the spaceship's surroundings or about the spaceship itself. This information is useful to make important decisions in the code that will help control the spaceship more safely, by making it avoid detected obstacles and find useful collectibles.

Similarly to actions, sensors are also activated by calling JavaScript functions. However, the big difference is that unlike actions, sensors never change the state of the spaceship or any game object and only return information. This information can then be used in other parts of the written program, to help decide which actions or sensors should be called next.

Get health

$$\textit{get\_health}(s, p) = s_{p,h}$$

Returns the current player health points.

- $s_{p,h} \in \mathbb{Z}$: Player health points.

Get fuel

$$\textit{get\_fuel}(s, p) = s_{p,f}$$

Returns the current player fuel points.

- $s_{p,f} \in \mathbb{Z}$: Player fuel points.

Distance to game object

$$\textit{distance\_to}(s, p, i) = \sqrt{(s_{p,x} - s_{g,i,x})^2 + (s_{p,y} - s_{g,i,y})^2}$$

Calculates and returns the distance between the spaceship and the game object with the specified id i.

- $i \in \mathbb{Z}$: Game object id;
- $(s_{p,x}, s_{p,y}) \in \mathbb{V}$: Player spaceship position;
- $(s_{g,i,x}, s_{g,i,y}) \in \mathbb{V}$: Position of game object with id i.

Check game object is collected

$$\textit{is\_collected}(s, i) = s_{i,c}$$

Checks if the game object id i is already collected. Returns *true* if the game object is collected, *false* otherwise.

- $i \in \mathbb{Z}$: Game object id;
- $s_{i,c} \in \mathbb{B}$: Game object with id i is collected.

Check game object is pressed

$$\textit{is\_pressed}(s, i) = s_{i,p}$$

Checks if the game object with id i is pressed. Returns *true* if the button is currently pressed, *false* otherwise.

- $i \in \mathbb{Z}$: Game object id;
- $s_{i,p} \in \mathbb{B}$: Game object with id i is pressed.

Check game object is open

$$\textit{is\_open}(s, i) = s_{i,o}$$

Checks if the game object with id i is open. Returns *true* if the game object is currently open, *false* otherwise.

- $i \in \mathbb{Z}$: Game object id;
- $s_{i,o} \in \mathbb{B}$: Game object with id i is open.

Check obstacle is forward

$$\textit{obstacle\_is\_forward}(s, p) = \exists i \in O \text{ such that } (s_{p,x,f}, s_{p,y,f}) = (s_{i,x}, s_{i,y})$$

Checks if there is an obstacle in the same direction of the spaceship, on the next coordinate. Returns *true* if there is an obstacle forward on the next coordinate and *false* otherwise.

- $i \in \mathbb{Z}$: Id of the game object currently being checked;
- $(s_{i,x}, s_{i,y}) \in \mathcal{V}$: Position of the object that is currently being checked;
- $(s_{p,x,f}, s_{p,y,f}) \in \mathbb{V}$: Position ahead of the spaceship.

Check obstacle is left

$$\textit{obstacle\_is\_left}(s, p) = \exists i \in O \text{ such that } (s_{p,x,l}, s_{p,y,l}) = (s_{i,x}, s_{i,y})$$

Checks if there is an obstacle to the left of the spaceship, on the next coordinate. Returns *true* if there is an obstacle to the left on the next coordinate, and *false* otherwise.

- $i \in \mathbb{Z}$: Id of the game object currently being checked;
- $(s_{i,x}, s_{i,y}) \in \mathcal{V}$: Position of the object that is currently being checked;
- $(s_{p,x,l}, s_{p,y,l}) \in \mathbb{V}$: Position to the left of the spaceship.

Check obstacle is right

$$\textit{obstacle\_is\_right}(s, p) = \exists i \in O \text{ such that } (s_{p,x,r}, s_{p,y,r}) = (s_{i,x}, s_{i,y})$$

Checks if there is an obstacle to the right of the spaceship, on the next coordinate. Returns *true* if there is an obstacle to the right on the next coordinate, and *false* otherwise.

- $i \in \mathbb{Z}$: Id of the game object currently being checked;

- $(s_{i,x}, s_{i,y}) \in \mathcal{V}$: Position of the object that is currently being checked;
- $(s_{p,x,r}, s_{p,y,r}) \in \mathbb{V}$: Position to the right of the spaceship.

Check target is untouched

$$\mathit{target\_untouched}() = u$$

Checks whether the target is untouched by the spaceship.

- $u \in \mathbb{B}$: Target is untouched.

5.7 Conditions to finalize game

The conditions for victory or game over are defined based on the game state $s \in S$, and are checked at each step of execution.

Victory Condition: The player wins the game when they successfully reach the target.

$$\mathit{victory}(s) \iff \neg \mathit{Sensors.target\_untouched}(s)$$

Game Over Conditions: The game ends in failure when one of the following conditions is met:

$$\mathit{game\_over}(s) \iff \begin{cases} \neg \mathit{P.player\_has\_health}(s_p) \quad \vee \\ \neg \mathit{P.player\_has\_fuel}(s_p) \quad \vee \\ (\mathit{X.current\_completed\_step}(s_x)_e \wedge \neg \mathit{X.rewinding}(s_x)) \quad \vee \\ (s_v = 0 \wedge s_u = 0) \end{cases}$$

5.8 Progression of Play

Given a state $s \in S$, the progress (next state) that results from action a, by player $p \in P$ is:

$$\mathit{progress}(s, p, a) = \begin{cases} \mathit{action}(s, p, a) & \text{if} \quad \neg \mathit{victory}(s) \quad \wedge \quad \neg \mathit{game\_over}(s) \\ s & \text{otherwise} \end{cases}$$

The state only progresses if it is not an ending state (victory, game over) and the action is legal.

6

Conclusions and Future Work

Overall, the project was completed with great results. The objective of creating a serious game for self-learning programming was achieved, and now *CodinSpace* is one more learning tool for anyone who wants to learn programming and face new challenges in a fun and interactive way. The game in its current state has a first chapter with a set of levels that serve as an introduction to JavaScript and to programming basics in general. It also provides tools for anyone to create their own fully playable chapters and levels, with almost total freedom to change many of the properties of each game object and the existing interpreter and game constraints.

However, it is important to recognize that the project is still far away from what its real potential can be. There is a lot of future work that can be done in order to truly have a complete tool for self-learning programming.

One of the improvements is the addition of interpreters for other programming languages, instead of only JavaScript. By allowing players to complete the challenges in different programming languages, the game would reach a wider audience because people who are not interested in learning JavaScript would have other alternatives to play the game. The game would also become much more replayable, because once the players beat the game in a certain programming language, they could start a new campaign using a different one.

Another feature to consider for future updates would be a new mode in which levels are played with a visual programming interface, similar to Scratch [Scr], as explored in the state-of-the-art chapter of this dissertation. This would be ideal for a younger audience and for players who would prefer to start learning programming in a simpler way, without having to explicitly write the code.

New game objects that behave differently and have other functionalities could also be added to the game. The new game objects would teach other programming concepts that are not currently in the game, or reinforce already existing concepts in different approaches.

The game should be playtested by various groups of people, with different programming skill levels (*e.g.* high school students, college freshmen without programming knowledge, college undergraduates with programming knowledge, etc.). With playtesting, we can get useful feedback on possible improvements and check if *CodinSpace* is an effective tool for self-learning programming.

Finally, a *CodinSpace* community website could also be created, specifically dedicated to host community chapters and levels uploaded by players and to display global player rankings about the various statistics of each level. This would require the creation of a front-end page for user interaction, a back-end server, and a database to store all the information about the players, chapters, and levels.

A

Code Examples

A.1 Attributes Properties Info

```
enum ATTRIBUTES {
  COLLECTABLE,
  OBSTACLE,
  DESTROYABLE,
    ...
  OPENABLE,
  PRESSABLE,
  GRABBABLE,
  REQUIRES,
  RANDOMLY_COLORIZED,
  REACTABLE_TO_SPACE_SIGNAL,
}

var ATTRIBUTE_PROPERTIES := {
  ATTRIBUTES.COLLECTABLE: [
    {
      Strings.NAME: Strings.COLLECTED,
      Strings.DEFAULT_VALUE: false
    }
  ],
  ...
  ATTRIBUTES.PRESSABLE: [
    {
      Strings.NAME: Strings.PRESSED,
      Strings.DEFAULT_VALUE: false
    },
    {
      Strings.NAME: Strings.CURRENTLY_BEING_PRESSED,
      Strings.DEFAULT_VALUE: false
    }
  ],
  ATTRIBUTES.OPENABLE: [
    {
      Strings.NAME: Strings.OPENED,
      Strings.DEFAULT_VALUE: false
    }
  ],
  ATTRIBUTES.RANDOMLY_COLORIZED: [
    {
      Strings.NAME: Strings.COLOR,
      Strings.DEFAULT_VALUE: Global.get_random_color_code
    }
  ]
}
```

A.2 Placeable Types Info

```
var PLACEABLE_TYPES_INFO := {
  Strings.PLAYER: {
    Strings.SECTION_NAME: Strings.PLAYER,
    Strings.LEVEL_EDITOR_TAB: Strings.LEVEL_EDITOR_GENERAL,
    Strings.LEVEL_EDITOR_SCENE: preload(Strings.PATH_LEVEL_EDITOR_PLAYER_SCENE),
    Strings.SPRITE: preload(Strings.PATH_PLAYER_PNG),
  },
    ...
  Strings.COIN: {
    Strings.SECTION_NAME: Strings.GAME_OBJECTS,
    Strings.SUBSECTION_NAME: Strings.COINS,
    Strings.ATTRIBUTES: [ATTRIBUTES.COLLECTABLE, ATTRIBUTES.XP, ATTRIBUTES.SCORE],
    Strings.POINTS: 100,
    Strings.XP: 100,
    Strings.SCENE: load(Strings.PATH_COIN_SCENE),
    Strings.LEVEL_EDITOR_TAB: Strings.LEVEL_EDITOR_GAME_OBJECTS,
    Strings.LEVEL_EDITOR_SCENE: preload(Strings.PATH_LEVEL_EDITOR_COIN_SCENE),
    Strings.SPRITE: preload(Strings.PATH_COIN_PNG),
  },
  ...
  Strings.CRATE: {
    Strings.SECTION_NAME: Strings.GAME_OBJECTS,
    Strings.SUBSECTION_NAME: Strings.CRATES,
    Strings.ATTRIBUTES: [ATTRIBUTES.OPENABLE, ATTRIBUTES.REQUIRES],
    Strings.SCENE: load(Strings.PATH_CRATE_SCENE),
    Strings.LEVEL_EDITOR_TAB: Strings.LEVEL_EDITOR_GAME_OBJECTS,
    Strings.LEVEL_EDITOR_SCENE: preload(Strings.PATH_LEVEL_EDITOR_CRATE_SCENE),
    Strings.SPRITE: preload(Strings.PATH_CRATE_PNG),
  },
    ...
  Strings.ASTEROID: {
    Strings.SECTION_NAME: Strings.GAME_OBJECTS,
    Strings.SUBSECTION_NAME: Strings.ASTEROIDS,
    Strings.ATTRIBUTES: [ATTRIBUTES.OBSTACLE, ATTRIBUTES.DESTROYABLE, ATTRIBUTES.
        DESTROYABLE_ON_PROJECTILE_IMPACT, ATTRIBUTES.DESTROYABLE_ON_TOUCH,
        ATTRIBUTES.SCALABLE],
    Strings.SCENE: load(Strings.PATH_ASTEROID_SCENE),
    Strings.LEVEL_EDITOR_TAB: Strings.LEVEL_EDITOR_GAME_OBJECTS,
    Strings.LEVEL_EDITOR_SCENE: preload(Strings.PATH_LEVEL_EDITOR_ASTEROID_SCENE),
    Strings.SPRITE: preload(Strings.PATH_ASTEROID_PNG),
  },
  Strings.DOOR: {
    Strings.SECTION_NAME: Strings.GAME_OBJECTS,
    Strings.SUBSECTION_NAME: Strings.DOORS,
    Strings.ATTRIBUTES: [ATTRIBUTES.OBSTACLE, ATTRIBUTES.OPENABLE, ATTRIBUTES.
        FLIPPABLE, ATTRIBUTES.SCALABLE, ATTRIBUTES.REQUIRES, ATTRIBUTES.
        REACTABLE_TO_SPACE_SIGNAL],
    Strings.SCENE: load(Strings.PATH_DOOR_SCENE),
    Strings.LEVEL_EDITOR_TAB: Strings.LEVEL_EDITOR_GAME_OBJECTS,
    Strings.LEVEL_EDITOR_SCENE: preload(Strings.PATH_LEVEL_EDITOR_DOOR_SCENE),
    Strings.SPRITE: preload(Strings.PATH_DOOR_PNG),
  },
}
```

A.3 Step Forward in the JavaScriptExecutionHandler

```
func step_forward() -> void:
  var current_step_is_statement := false
  while not current_step_is_statement:
    var id := StepsStateManager.get_game_interpreter_steps_count()

    var time_step_start := Time.get_ticks_msec()

    var execution_terminated := false
    var message = null
    var error = null

    var content = await JavascriptFunctionCalls.get_game_interpreter_step_value()

    if await JavascriptFunctionCalls.get_game_interpreter_status() == 0:
      execution_terminated = true
      var interpreter_value = await JavascriptFunctionCalls.
          get_game_interpreter_value()
      if interpreter_value != null:
        message = Strings.GAME_OVER_ERROR
      else:
        message = Strings.GAME_OVER_EXECUTION_TERMINATED
      error = interpreter_value

    if not execution_terminated:
      await JavascriptFunctionCalls.advance_game_interpreter_step()
    current_step_is_statement = await JavascriptFunctionCalls.
        check_game_interpreter_current_step_is_statement()

    var used_memory_bytes = await JavascriptFunctionCalls.
        get_game_interpreter_current_rough_value_memory_bytes()
    if not execution_terminated and used_memory_bytes >= ConstraintsStateManager.
        get_max_game_interpreter_memory_bytes():
      execution_terminated = true
      message = Strings.GAME_OVER_MEMORY_EXCEEDED
      error = Strings.GAME_OVER_MEMORY_EXCEEDED

    var highlighted_code_range = await JavascriptFunctionCalls.
        get_game_interpreter_highlighted_code_range()

    var time_step_end := Time.get_ticks_msec()
    var elapsed_step_time := time_step_end - time_step_start
    var previous_step_runtime = StepsStateManager.get_step_runtime_by_id(id - 1)
    var runtime = null
    if previous_step_runtime == null:
      runtime = elapsed_step_time
    else:
      runtime = previous_step_runtime + elapsed_step_time
    if not execution_terminated and runtime >= ConstraintsStateManager.
        get_max_game_interpreter_runtime():
      execution_terminated = true
      message = Strings.GAME_OVER_MAXIMUM_INTERPRETER_RUNTIME_EXCEEDED
      error = Strings.GAME_OVER_MAXIMUM_INTERPRETER_RUNTIME_EXCEEDED

    if StepsStateManager.get_game_interpreter_steps_count() >=
        ConstraintsStateManager.get_max_game_interpreter_steps():
      execution_terminated = true
      message = Strings.GAME_OVER_MAXIMUM_STEPS_EXCEEDED
      error = Strings.GAME_OVER_MAXIMUM_STEPS_EXCEEDED

    var step := {
      Strings.ID: id,
```

```
      Strings.EXECUTION_TERMINATED: execution_terminated,
      Strings.HIGHLIGHTED_CODE_RANGE: highlighted_code_range,
      Strings.CONTENT: content,
      Strings.ERROR: error,
      Strings.MESSAGE: message,
      Strings.USED_MEMORY_BYTES: used_memory_bytes,
      Strings.SUB_STEP: not current_step_is_statement and content == null,
      Strings.RUNTIME: runtime
    }

    StepsStateManager.add_completed_step(step)
    StepsStateManager.increment_game_interpreter_steps_count()

    if execution_terminated or StepsStateManager.get_fetching_interpreter_sub_steps
        ():
            return
```

A.4 Global State Manager singleton

```
var state := {}

func initialize_state() -> void:
  state = {}
  if Global.get_selected_level_uuid() == Strings.EMPTY_STRING:
    Global.set_playing_campaign(true)
    Global.set_selected_chapter_uuid(Strings.DEFAULT_CHAPTER_UUID)
    Global.set_selected_level_uuid(Strings.DEFAULT_LEVEL_UUID)

  var chapter_uuid: String = Global.get_selected_chapter_uuid()
  var level_uuid: String = Global.get_selected_level_uuid()
  if Global.get_playing_campaign():
    state = Global.get_campaign_level_data(chapter_uuid, level_uuid)
  else:
    state = Global.get_community_level_data(level_uuid)

  ConstraintsStateManager.initialize(state[Strings.CONSTRAINTS])
  ConsoleStateManager.initialize()
  StepsStateManager.initialize()
  PlayerStateManager.initialize(state[Strings.PLAYER])

  if not state.has(Strings.GAME_OBJECTS):
    state[Strings.GAME_OBJECTS] = {}
  GameObjectsStateManager.initialize(state[Strings.GAME_OBJECTS])

  SpaceSignalsStateManager.initialize()
  initialize_highlighted_code_range()
  initialize_game_over_message()
  initialize_game_interpreter_runtime()
  initialize_game_interpreter_memory_bytes()
  initialize_score()
  initialize_time()
  update_global_state()
  state_initialized.emit()

...

func advance_step() -> void:
  var step := await StepsStateManager.fetch_next_step_data()

  update_game_state(step)
  update_global_state()
```

```
43
44    if StepsStateManager.check_current_step_is_last():
45      if StepsStateManager.get_objective_achieved():
46        victory.emit()
47      else:
48        game_over.emit()
49
50  ...
51
52  func update_game_state(step: Dictionary) -> void:
53    var rewinding := StepsStateManager.get_rewinding()
54    if StepsStateManager.get_current_step_id() == 0:
55      initialize_highlighted_code_range()
56      set_game_interpreter_memory_bytes(0)
57      set_game_interpreter_runtime(0)
58    else:
59      set_highlighted_code_range(step[Strings.HIGHLIGHTED_CODE_RANGE])
60      set_game_interpreter_memory_bytes(step[Strings.USED_MEMORY_BYTES])
61      set_game_interpreter_runtime(step[Strings.RUNTIME])
62
63    if rewinding:
64      check_game_finished(step)
65
66    SpaceSignalsStateManager.disable_all_space_signals()
67
68    var step_content = step[Strings.CONTENT]
69    if step_content != null:
70      var content_type: String = step_content[Strings.TYPE]
71      if content_type == Strings.SENSOR:
72        var sensor_name: String = step_content[Strings.NAME]
73        var sensor_returned_value := str(step_content[Strings.VALUE])
74        Sensors.sensor_activated.emit(sensor_name, sensor_returned_value)
75        if not rewinding:
76          ConsoleStateManager.add_sensor_log(sensor_name, sensor_returned_value)
77      elif content_type == Strings.ACTION:
78        Sensors.sensor_disabled.emit()
79        if rewinding:
80          GameObjectsStateManager.update_game_objects()
81          Actions.do_step_action(step_content)
82        else:
83          Actions.do_step_action(step_content)
84          GameObjectsStateManager.update_game_objects()
85    else:
86      Sensors.sensor_disabled.emit()
87
88    if not rewinding:
89      check_game_finished(step)
90  ...
```

A.5 Player State Manager singleton

```
1  var player_state := {}
2
3  func initialize(initial_player_state: Dictionary) -> void:
4    player_state = {}
5    set_player_state(initial_player_state)
6    initialize_player_health()
7    initialize_player_fuel()
8    set_player_pressing(false)
9    set_player_opening(false)
10   set_player_emitting(false)
```

```
11
12 func get_player_state() -> Dictionary:
13   return player_state
14
15 func set_player_state(value: Dictionary) -> void:
16   player_state = value
17
18 ...
19
20 func get_player_position() -> Vector2:
21   return Vector2(player_state[Strings.X], player_state[Strings.Y])
22
23 func set_player_position(value: Vector2) -> void:
24   player_state[Strings.X] = value.x
25   player_state[Strings.Y] = value.y
26
27 ...
28
29 func get_player_emitting() -> bool:
30   return player_state[Strings.EMITTING]
31
32 func set_player_emitting(value: bool) -> void:
33   player_state[Strings.EMITTING] = value
34
35 ...
36
37 func check_player_has_health() -> bool:
38   return get_player_health() > 0
39
40 func damage_player(health_points: int) -> void:
41   var max_player_health: int = ConstraintsStateManager.get_max_player_health()
42   player_state[Strings.HEALTH] = clamp(player_state[Strings.HEALTH] - health_points
       , 0, max_player_health)
43
44 ...
45
46 func check_player_has_fuel() -> bool:
47   return get_player_fuel() > 0
48
49 func increase_player_fuel(fuel_points: int) -> void:
50   var max_player_fuel: int = ConstraintsStateManager.get_max_player_fuel()
51   player_state[Strings.FUEL] = clamp(player_state[Strings.FUEL] + fuel_points, 0,
       max_player_fuel)
52
53 func decrease_player_fuel(fuel_points: int) -> void:
54   var max_player_fuel: int = ConstraintsStateManager.get_max_player_fuel()
55   player_state[Strings.FUEL] = clamp(player_state[Strings.FUEL] - fuel_points, 0,
       max_player_fuel)
56
57 func get_player_space_signal_range() -> int:
58   return player_state[Strings.SPACE_SIGNAL_RANGE]
59
60 ...
```

A.6 Console State Manager singleton

```
1 var console_state := {}
2
3 enum CONSOLE_LOG_TYPE {
4   ACTION,
5   SENSOR,
```

```
6     GAME_OBJECT,
7     SPACE_SIGNAL,
8     SCORE,
9     FINAL
10  }
11
12  func initialize() -> void:
13    console_state = {}
14    console_state[Strings.CONSOLE_LOGS] = []
15    initialize_filters()
16    console_initialized.emit()
17
18  func initialize_filters() -> void:
19    console_state[Strings.FILTERS] = {
20      Strings.ACTIONS: true,
21      Strings.SENSORS: true,
22      Strings.GAME_OBJECTS: true,
23      Strings.SPACE_SIGNALS: true,
24      Strings.SCORE: true,
25      Strings.FINAL: true
26    }
27
28  func get_filters() -> Dictionary:
29    return console_state[Strings.FILTERS]
30
31  func toggle_filter(filter_name: String) -> bool:
32    var filter = not console_state[Strings.FILTERS][filter_name]
33    console_state[Strings.FILTERS][filter_name] = filter
34    return filter
35
36  func get_console_logs() -> Array:
37    return console_state[Strings.CONSOLE_LOGS]
38
39  func get_filtered_console_logs() -> Array:
40    var filtered_console_logs := []
41    var filters := get_filters()
42    for console_log: Dictionary in get_console_logs():
43      var type = console_log[Strings.TYPE]
44      if filters[type]:
45        filtered_console_logs.push_back(console_log)
46    return filtered_console_logs
47
48  func add_console_log(console_log: String, color: String, type: String) -> void:
49    console_state[Strings.CONSOLE_LOGS].push_back({
50      Strings.STEP_ID: StepsStateManager.get_current_step_id(),
51      Strings.LOG: console_log,
52      Strings.COLOR: color,
53      Strings.TYPE: type
54    })
55
56  func add_sensor_log(sensor_name: String, sensor_value: String) -> void:
57    var console_log := Strings.LOG_SENSOR.format({Strings.SENSOR_NAME: sensor_name,
        Strings.SENSOR_VALUE: sensor_value})
58    add_console_log(console_log, Strings.COLOR_YELLOW, Strings.SENSORS)
59
60  ...
61
62  func add_player_is_pressing_log() -> void:
63    add_console_log(Strings.LOG_PLAYER_IS_PRESSING, Strings.COLOR_BLUE, Strings.
        ACTIONS)
64
65  func add_player_is_opening_log() -> void:
66    add_console_log(Strings.LOG_PLAYER_IS_OPENING, Strings.COLOR_BLUE, Strings.
```

```
      ACTIONS)

func add_player_emitted_space_signal_log(value) -> void:
  print(value)
  var console_log := Strings.LOG_PLAYER_EMITTED_SPACE_SIGNAL.format({Strings.VALUE:
      value})
  add_console_log(console_log, Strings.COLOR_WHITE, Strings.SPACE_SIGNALS)

...

func add_error_log(error_message: String) -> void:
  add_console_log(error_message, Strings.COLOR_RED, Strings.FINAL)

func delete_console_logs_by_step_id(step_id: int) -> void:
  var console_logs: Array = console_state[Strings.CONSOLE_LOGS]
  console_state[Strings.CONSOLE_LOGS] = console_logs.filter(
    func(console_log: Dictionary):
      return console_log[Strings.STEP_ID] != step_id and console_log[Strings.TYPE]
          != Strings.FINAL
  )
```

A.7 Constraints State Manager singleton

```
var constraints_state := {}

func initialize(initial_constraints_state = null) -> void:
  constraints_state = {}
  if initial_constraints_state != null:
    constraints_state = initial_constraints_state

...

func set_max_player_health(value: int) -> void:
  constraints_state[Strings.MAX_HEALTH] = value

...

func get_max_game_interpreter_steps() -> int:
  return constraints_state[Strings.MAX_GAME_INTERPRETER_STEPS]

func set_max_game_interpreter_steps(value: int) -> void:
  constraints_state[Strings.MAX_GAME_INTERPRETER_STEPS] = value

func get_max_game_interpreter_runtime() -> int:
  return constraints_state[Strings.MAX_GAME_INTERPRETER_RUNTIME]

func set_max_game_interpreter_runtime(value: int) -> void:
  constraints_state[Strings.MAX_GAME_INTERPRETER_RUNTIME] = value

...

func set_max_minutes(value: int) -> void:
  constraints_state[Strings.MAX_MINUTES] = value
```

A.8 Space Signals State Manager singleton

```
var space_signals_state := {}

func initialize() -> void:
```

```
  space_signals_state = {}

func get_space_signals_state() -> Dictionary:
  return space_signals_state

func get_current_step_space_signals() -> Dictionary:
  var step_id := StepsStateManager.get_current_step_id()
  if StepsStateManager.get_rewinding():
    step_id += 1
  var step_id_string := str(step_id)
  if space_signals_state.has(step_id_string):
    return space_signals_state[step_id_string]
  return {}

...

func enable_space_signal(step_id: String, space_signal_id: String) -> void:
  if space_signals_state[step_id][space_signal_id][Strings.ENABLED]:
    return
  space_signals_state[step_id][space_signal_id][Strings.ENABLED] = true
  space_signals_state[step_id][space_signal_id][Strings.CURRENTLY_BEING_EMITTED] =
      true
  space_signal_emitted.emit(space_signal_id)

...

func emit_space_signal(value, position: Vector2, signal_range: int, rewinding: bool
    ) -> void:
  var step_id = StepsStateManager.get_current_step_id()
  if rewinding == true:
    step_id += 1
  step_id = str(step_id)

  var space_signal_id := Strings.ZERO

  if not space_signals_state.has(step_id):
    space_signals_state[step_id] = {}

  if StepsStateManager.check_current_step_is_most_recent():
    var step_space_signals_count: int = space_signals_state[step_id].size()
    space_signal_id = str(step_space_signals_count)
    space_signals_state[step_id][space_signal_id] = {
      Strings.VALUE: value,
      Strings.X: position.x,
      Strings.Y: position.y,
      Strings.RANGE: signal_range,
      Strings.ENABLED: true,
      Strings.CURRENTLY_BEING_EMITTED: true
    }
    space_signal_created.emit(space_signal_id)
    return
  enable_space_signal(step_id, space_signal_id)
```

A.9 Game Objects State Manager singleton

```
var game_objects_state := {}

func initialize(initial_game_objects_state: Dictionary) -> void:
  game_objects_state = initial_game_objects_state
  for section_name: String in game_objects_state.keys():
    initialize_game_objects_section(game_objects_state[section_name], section_name)
```

```
7
8  func initialize_game_objects_section(section: Dictionary, section_name: String) ->
       void:
9    var game_object_type := Strings.get_game_object_type_by_section_name(section_name
       )
10   var attributes: Array = Constants.PLACEABLE_TYPES_INFO[game_object_type][Strings.
       ATTRIBUTES]
11   for game_object_id: String in section.keys():
12     initialize_game_object(section_name, game_object_id, game_object_type,
         attributes)
13
14 func initialize_game_object(section_name: String, id: String, type: String,
       attributes) -> void:
15   var game_object := get_game_object(section_name, id)
16   game_object[Strings.ID] = id
17   game_object[Strings.SECTION_NAME] = section_name
18   game_object[Strings.TYPE] = type
19
20   for attribute in attributes:
21     if not Constants.ATTRIBUTE_PROPERTIES.has(attribute):
22       continue
23     for property in Constants.ATTRIBUTE_PROPERTIES[attribute]:
24       var property_name: String = property[Strings.NAME]
25       if not game_object.has(property_name):
26         if typeof(property[Strings.DEFAULT_VALUE]) == TYPE_CALLABLE:
27           game_object[property_name] = property[Strings.DEFAULT_VALUE].call()
28         else:
29           game_object[property_name] = property[Strings.DEFAULT_VALUE]
30
31 func update_game_objects() -> void:
32   Sensors.check_reactable_game_objects()
33   for collectable: Dictionary in Sensors.check_collectable_game_objects():
34     collect_game_object(collectable)
35   for obstacle: Dictionary in Sensors.check_touchable_obstacles():
36     touch_obstacle(obstacle)
37   for pressable: Dictionary in Sensors.check_pressable_game_objects():
38     press_game_object(pressable)
39   for openable: Dictionary in Sensors.check_openable_game_objects():
40     open_game_object(openable)
41
42 func collect_game_object(game_object: Dictionary) -> void:
43   var rewinding := StepsStateManager.get_rewinding()
44   var game_object_type: String = game_object[Strings.TYPE]
45
46   if not check_game_object_can_be_changed(Constants.ATTRIBUTES.COLLECTABLE,
       game_object):
47     return
48
49   game_object[Strings.COLLECTED] = not rewinding
50
51   if game_object_type == Strings.JERRYCAN:
52     if not rewinding:
53       game_object[Strings.PLAYER_FUEL_BEFORE_CHANGE] = PlayerStateManager.
           get_player_fuel()
54       PlayerStateManager.increase_player_fuel(game_object[Strings.FUEL] + 1)
55     else:
56       PlayerStateManager.set_player_fuel(game_object[Strings.
           PLAYER_FUEL_BEFORE_CHANGE])
57   elif game_object_type == Strings.REPAIR:
58     if not rewinding:
59       game_object[Strings.PLAYER_HEALTH_BEFORE_CHANGE] = PlayerStateManager.
           get_player_health()
60       PlayerStateManager.heal_player(game_object[Strings.HEALTH] + 1)
```

```
61      else:
62        PlayerStateManager.set_player_health(game_object[Strings.
              PLAYER_HEALTH_BEFORE_CHANGE])
63    count_game_object_xp(game_object)
64
65    if not rewinding:
66      game_object[Strings.CHANGED_AT_STEP] = StepsStateManager.get_current_step_id()
67      GlobalStateManager.increase_score(game_object[Strings.SCORE])
68      ConsoleStateManager.add_game_object_log(game_object, Strings.COLLECTED)
69    else:
70      GlobalStateManager.decrease_score(game_object[Strings.SCORE])
71    collectable_collected.emit(game_object[Strings.SECTION_NAME], game_object[Strings
          .ID])
72
73  ...
```

A.10 Steps State Manager singleton

```
1   var steps_state := {}
2
3   func initialize() -> void:
4     steps_state = {}
5     set_playing_one_step(false)
6     set_playing_all_steps(false)
7     set_rewinding(false)
8     set_objective_achieved(false)
9     set_mission_terminated(false)
10    set_fetching_interpreter_sub_steps(false)
11    initialize_game_interpreter_steps_count()
12    initialize_current_step_id()
13    initialize_completed_steps()
14
15  func get_steps_state() -> Dictionary:
16    return steps_state
17
18  func set_steps_state(value: Dictionary) -> void:
19    steps_state = value
20
21  func fetch_previous_step_data() -> Dictionary:
22    var current_step_is_sub_step := true
23    var current_step = null
24    while get_current_step_id() > 0 and current_step_is_sub_step:
25      decrement_current_step_id()
26      current_step = get_current_completed_step()
27      current_step_is_sub_step = current_step[Strings.SUB_STEP]
28      if get_fetching_interpreter_sub_steps() == true:
29        return current_step
30    return current_step
31
32  func fetch_next_step_data() -> Dictionary:
33    var current_step_is_sub_step := true
34    var current_step = null
35    while current_step_is_sub_step:
36      if check_current_step_is_most_recent() and not get_mission_terminated():
37        await JavascriptExecutionHandler.step_forward()
38      current_step = get_current_completed_step()
39      current_step_is_sub_step = current_step[Strings.SUB_STEP]
40      increment_current_step_id()
41      if get_fetching_interpreter_sub_steps() or current_step[Strings.
            EXECUTION_TERMINATED]:
42        return current_step
```

```
43    return current_step
44
45  func get_current_completed_step() -> Dictionary:
46    var current_step_id: int = get_current_step_id()
47    return steps_state[Strings.COMPLETED_STEPS][current_step_id]
48
49  func add_completed_step(step: Dictionary) -> void:
50    steps_state[Strings.COMPLETED_STEPS].push_back(step)
51
52  func get_playing_one_step() -> bool:
53    return steps_state[Strings.PLAYING_ONE_STEP]
54
55  ...
56
57  func get_mission_terminated_step_id() -> int:
58    return steps_state[Strings.MISSION_TERMINATED_STEP_ID]
59
60  func set_mission_terminated(value: bool) -> void:
61    if value == true and not steps_state.has(Strings.MISSION_TERMINATED_STEP_ID):
62      steps_state[Strings.MISSION_TERMINATED_STEP_ID] = StepsStateManager.
          get_current_step_id()
63    steps_state[Strings.MISSION_TERMINATED] = value
64
65  func get_objective_achieved() -> bool:
66    return steps_state[Strings.OBJECTIVE_ACHIEVED]
67
68  ...
```

Bibliography

[BS09] Brenda Brathwaite and Ian Schreiber. *Challenges for Game Designers*. Charles River Media, 2009.

[Che] CheckiO. Checkio - coding games and programming challenges for beginner and advanced. `https://checkio.org`
Accessed: (09/03/24).

[CKXG11] António Coelho, Enrique Kato, João Xavier, and Ricardo Gonçalves. Serious game for introductory programming. In Minhua Ma, Manuel Fradinho Oliveira, and Madeiras Pereira João, editors, *Serious Games Development and Applications*, pages 61–71, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Coda] CodeCombat. Codecombat - coding games to learn python and javascript. `https://codecombat.com`
Accessed: (21/02/24).

[Codb] CodinGame. Codingame - coding games and programming challenges to code better. `https://www.codingame.com`
Accessed: (12/03/24).

[Dae] Daenvil. Markdownlabel - a custom godot node that extends richtextlabel to use markdown instead of bbcode. `https://github.com/daenvil/MarkdownLabel`
Accessed: (05/04/24).

[Dua11] José Manuel Cobiça Duarte. Serious game - ferramentas lúdicas ou pedagógicas? Master's dissertation, Universidade de Évora, Évora, Portugal, 2011. `http://hdl.handle.net/10174/14081`

[Ecm] EcmaScript. Ecmascript 2025 language specification. `https://tc39.es/ecma262/#sec-intro`
Accessed: (28/11/24).

[Gam] Epic Games. Epic games homepage. `https://www.epicgames.com/site/home`.

[GDQ] GDQuest. Learn gdscript from zero. `https://gdquest.github.io/learn-gdscript/`
Accessed: (04/12/24).

[GDS] GDScript. Gdscript reference - godot engine. https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html.

[Goda] Godot. Godot engine - free and open source 2d and 3d game engine. https://godotengine.org/
Accessed: (08/09/24).

[Godb] Godot. Javascriptbridge - godot engine 4.3 documentation. https://docs.godotengine.org/en/stable/classes/class_javascriptbridge.html
Accessed: (02/12/24).

[Godc] Godot. Singletons (autoload) - godot engine 4.3 documentation. https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html
Accessed: (02/12/24).

[Gre] Greenfoot. Greenfoot | about greenfoot. https://www.greenfoot.org/overview
Accessed: (28/11/24).

[HNVIPRC14] Raquel Hijon-Neira, Ángel Velázquez-Iturbide, Celeste Pizarro-Romero, and Luís Carriço. Serious games for motivating into programming. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8, 2014. 10.1109/FIE.2014.7044111.

[Kos14] Raph Koster. *Theory of Fun for Game Design*. O'Reilly Media, 2014.

[Mat16] Pedro Farias Mateus. Jogo para aprender a programar. Master's dissertation, Universidade de Évora, Évora, Portugal, November 2016. http://hdl.handle.net/10174/19804

[MB18] Michael Miljanovic and Jeremy Bradbury. A review of serious games for programming. *Conference: Proc. of the 4th Joint Conference on Serious Games (JCSG 2018)*, 11 2018.

[Mic] Microsoft. C# - a modern, open-source programming language. https://dotnet.microsoft.com/en-us/languages/csharp.

[Min] Minoqi. Minos uuid generator.
https://github.com/Minoqi/minos-UUID-generator-for-godot
Accessed: (21/04/24).

[Moz] Mozilla. Javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript
Accessed: (25/01/24).

[Pyt] Python. Welcome to python.org. https://www.python.org/
Accessed: (28/11/24).

[Scr] Scratch. Scratch - imagine, program, share. https://scratch.mit.edu
Accessed: (16/03/24).

[SSVGM+20] Santiago Schez-Sobrino, David Vallejo, Carlos Glez-Morcillo, Miguel Á. Redondo, and José Jesús Castro-Schez. Robotic: A serious game based on augmented reality for learning programming. *Multimedia Tools and Applications*, 79(45):34079–34099, 2020. https://doi.org/10.1007/s11042-020-09202-z

[Sta] Statista. Most used programming languages among developers worldwide as of 2024. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
Accessed: (17/08/24).

[Uni] Unity. Unity real-time development platform | 3d, 2d, vr ar engine. https://unity.com/.

[Unr] Unreal. Unreal game engine. https://www.unrealengine.com/.

[W3T] W3Techs. Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript
Accessed: (28/11/24).

[Wes] Butch Wesley. Gut - godot unit testing. https://github.com/bitwes/Gut
Accessed: (19/05/24).

[Wik] Wikipedia. C++. https://en.wikipedia.org/wiki/C%2B%2B.

[Wol] Magnus Wolffelt. Elevator saga - the elevator programming game. https://play.elevatorsaga.com
Accessed: (13/03/24).

[ZR13] Matej Zapušek and Jože Rugelj. Learning programming with serious games. *EAI Endorsed Transactions on Serious Games*, 1(1), 3 2013. 10.4108/trans.gbl.01-06.2013.e6.

UNIVERSIDADE DE ÉVORA
INSTITUTO DE INVESTIGAÇÃO
E FORMAÇÃO AVANÇADA

Contactos:
Universidade de Évora
Instituto de Investigação e Formação Avançada — IIFA
Palácio do Vimioso | Largo Marquês de Marialva, Apart. 94
7002 - 554 Évora | Portugal
Tel: (+351) 266 706 581
Fax: (+351) 266 744 677
email: iifa@uevora.pt

---

email: m54062@alunos.uevora.pt
diogoalexandrecnp@gmail.com