UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

Cavernous Expedition: Generarea procedurală a peșterilor versiunea 2024

Mihai-Cosmin Roșu

Coordonator științific:

Conf. dr. ing. Anca Morar

BUCUREȘTI
2024

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

Cavernous Expedition: Procedural generation of caves
2024 version

Mihai-Cosmin Roșu

Thesis advisor:

Conf. dr. ing. Anca Morar

BUCHAREST

2024

CONTENTS

SINOPSIS

Un joc 3D first-person singleplayer de tip puzzle și aventură în care jucătorul se află în rolul unui aventurier ce își propune explorarea unui sistem de peșteri nou descoperit. În acest sistem subteran, jucătorul se va lovi de diferite puzzle-uri, care îi vor testa modul de gândire, ce vor trebui rezolvate pentru a continua explorarea în subteran, timp în care trebuie să își asigure supraviețuirea prin căutarea resurselor pierdute de alți exploratori. Generarea procedurală se folosește de diferiți algoritmi pentru a asigura un sistem de peșteri realistic și diferit de fiecare dată, în cadrul căruia jucătorul va trebui să depășească multiple provocări pentru a explora întregul sistem subteran.

ABSTRACT

A 3D first-person singleplayer puzzle adventure game in which the player takes the role of an adventurer who sets out to explore a newly discovered cave system. In the underground system, the player will come across various puzzles, which will test their way of thinking, that have to be solved in order to continue the underground exploration, while also having to ensure their survival by searching for resources lost by other explorers. Procedural generation uses different algorithms to ensure a realistic and unique cave system each run, where the player will have to overcome multiple challenges in order to explore the entire underground system.

1 INTRODUCTION

Globally, video games form an enormous market which is constantly evolving, and with this evolution lots of new genres are popping up [1], while old genres are either forgotten or fused together to create new ones. Puzzle games usually focus on challenging the player's thinking skills [2] and require the player to spend time solving the puzzle so they can advance further. While adventure games can contain puzzles, they are more focused on exploration and action [3] and most of them require hand crafted levels. What if those two genres could be combined without the need of hand crafted levels or worlds? The solution for this is procedural generation [4]. It is a technique used to automatically produce content and is used in game development for limitless content, dynamic game worlds and reduced development resources.

Cavernous Expedition is an immersive 3D singleplayer adventure game in which the player must traverse a newly discovered underground cave system. The game challenges the player's thinking skills through a variety of interesting puzzles of multiple difficulties. Through the use of procedural generation, each run provides a distinct cave system, with different puzzles and ultimately with an unpredictable experience. Besides the puzzles, the player has to keep an eye out for resources which help him when exploring and solving puzzles.

The following chapters present the current market and game engine possibilities, the requirements of the application, what the application offers while presenting the workflow of classes, details regarding the implementation and the assessment of the results, in regards to performance and feedback. For the market research, multiple games are presented, which fall into similar categories, be it in regards to genre or theme. Then, the game engine options are presented and analyzed. In the following chapter, the requirements are presented from a game design perspective. After this, the game is presented while describing the flow and architecture of the classes, the way they interact and how they impact the game. Then, more details are written about the algorithms used and about the already presented classes. A couple of code snippets are presented, whenever a new technique or functionality is used. Lastly, the results are assessed, from a performance perspective, regarding the algorithms used, and from a user standpoint, presenting their feedback and opinion on the game.

2 MARKET RESEARCH

2.1 Similar games

In the current day, the video game market is a place where everyone can find a game title to their liking. because of the vast array of genres that are currently published. As the player preferences evolve, and the technology allows experimentation, more and more types of games will make their appearance. At the same time, the original genres are also coming back reinterpreted and more interesting than ever. As shown in Figure 1 [5], the adventure genre is situated in top 3 for all of the major gaming platforms, even taking first place on consoles. While the puzzle genre only appears on mobile, it takes second place, even placing above adventure games.

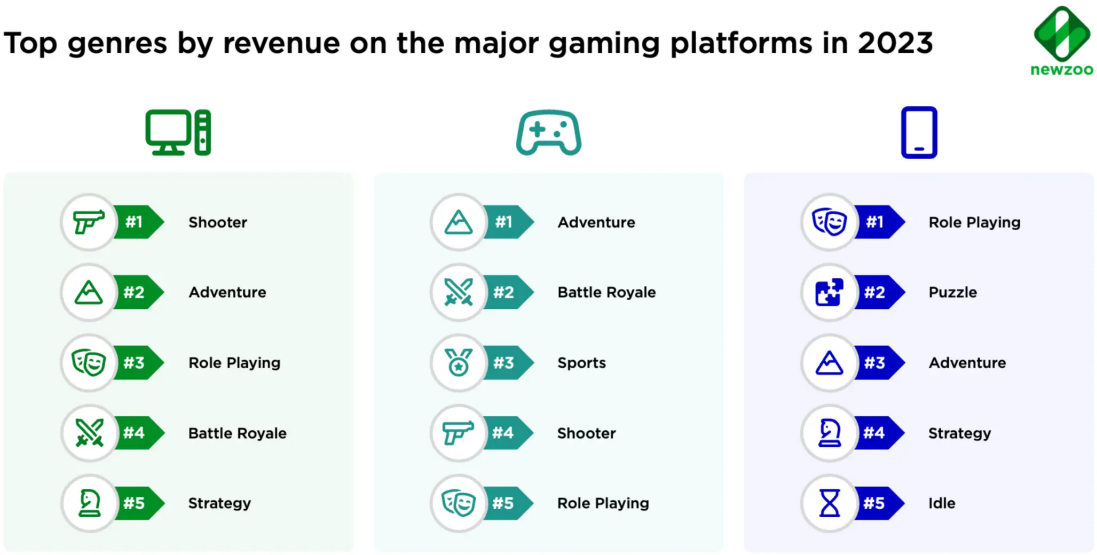

Figure 1: Most popular video game genres on the major gaming platforms (2023)

Adventure games are the ones that focus on narrative and exploration. The first ever adventure video game is considered to be Colossal Cave [6], which was released in 1976 by Will Crowther. It is a text adventure game, which means playing the game is similar to reading a book, with the major difference of being able to change the course of the story. Nowadays, adventure games became much more complex, with huge worlds and rich stories that are waiting to be discovered by the players. Most of those games now incorporate puzzles or challenges that must be solved to advance further, but in the majority of them the exploration of the world and the story still remain the main focus. An example of a game series that has the players adventure inside its world, while still challenging their thinking skills is the Portal series

[7]. Most of the game revolves around solving the puzzles with the help of the portals, but ultimately they allow the player to progress in the story and discover the world. Another example is the We Were Here series [8], which embraces even more the puzzle solving feature, focusing completely on crafting interesting puzzles for the players to solve in a two-player co-op playstyle. Still, the puzzles unlock more and more of the story as they are solved, allowing the players to advance in the game.

Procedural generation is a technique that has been used to give replayability to games for a long time. The first game to feature procedural generation is Rogue [9], released in 1980. The game is played in the CLI, and the player adventures into a procedurally generated dungeon which ensures a different experience each run. In the current day, procedural generation is much more advanced, covering not only the story, but also the graphics, being able to compete with hand-made models. A good example is No Man's Sky [10], which offers the player the opportunity to explore over 18 quintillion planets in the galaxy [11]. The players take the role of travellers exploring various planets. They have to gather resources and survive by building bases and spacecrafts with the ultimate goal of reaching the center of the galaxy. Another well known title is Minecraft [12], which has vast worlds containing fields, oceans, mountains, caves, multiple dimensions and many more, all procedurally generated. While the game is more of a sandbox, allowing the players to build anything they want, there are also elements of exploration, survival and resource gathering with the end goal of defeating a final boss.

There are also numerous games revolving around caves that use procedural generation for their environment. Looking at 2D games, one good example is Spelunky [13], which is a roguelike platformer. Here, the players must traverse procedurally generated underground dungeons that are filled with traps and enemies in hopes of finding treasure. On the other hand, Deep Rock Galactic [14] is a great example of a 3D game with procedural generation and an underground cave theme. In this game, the players take the role of various dwarves that are tasked with exploring procedurally generated caves filled with alien bugs, which they must fight against, with the purpose of collecting resources and repairing old equipment. There are also classes, each with their own perks and progression system.

While the previously mentioned games are released on PC and consoles, mobile games are also worth taking a look at. As shown in Figure 2, the mobile segment has generated more revenue than PC and consoles combined. In this category of games, one that stands out is Don't Starve [15]. While it is released on PC and consoles, it is also released on Android and iOS. Here, the player has to survive in a procedurally generated world by collecting resources to eat and to have a light source during the night, when monsters come out in dark areas. It has a 2D artstyle and features multiple biomes, caverns and oceans.

Using this research, I can conclude that even though there are plenty of adventure games that revolve around puzzles, they have hand-crafted levels and worlds. Looking at the procedural generation, there are adventure games that use this technique to offer endless replayability. There are even such games that are themed after underground caverns, present on all of the most popular platforms. But, there aren't any that combine adventure, puzzles and procedural

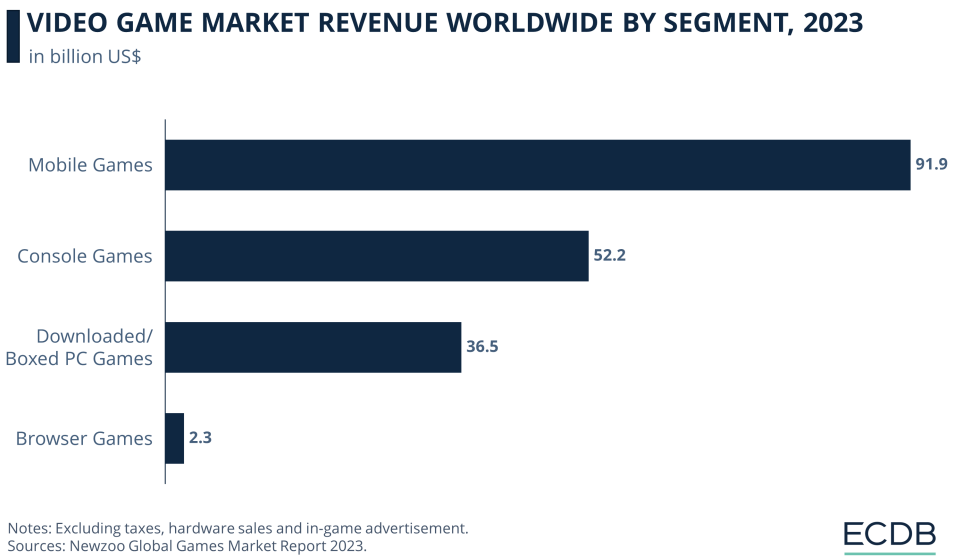

Figure 2: Video game market revenue worldwide by segment (2023)

generation so Cavernous Expedition can fill this gap in the market.

2.2 Game engines

Picking the right game engine is one of the most important steps in the development of a game. It dictates the structure of the game while offering support for various features. Nowadays, most game engines offer a multitude of features and most of them cover the basic necessities, but there are still numerous differences between them.

As shown in Figure 3 [16], Unity [17] is the most popular game engine in 2024. It offers support for complex operations such as multithreaded programming or shaders and allows 2D, 3D and virtual reality development, supporting animations, importing 3D objects and many more. It also has support for Android and iOS and can be used on Windows, macOS or Linux and uses the C# programming language. It can also be a more lightweight game engine than its competitors, and is more beginner friendly with its extensive asset store [18] that offers a multitude of free assets, from 2D or 3D objects to animations, scripts and many more. A few well known games developed in Unity are Cuphead [19] and Beat Saber [20].

In second place is the Godot game engine [21]. It is an open source game engine that was initially developed by Juan Linietsky and Ariel Manzur in 2014. Developers can use C#, C++ or Godot's own programming language, GDScript, which have native support, or they can use the GDExtension to allow them to program in any programming language they want. Godot offers support for 2D and 3D games for all platforms and allows developers to write shaders or multithreaded programs. Though it is still limited compared to its other competitors, using the fact that it is open source, the community will be able to shape it exactly how they want.

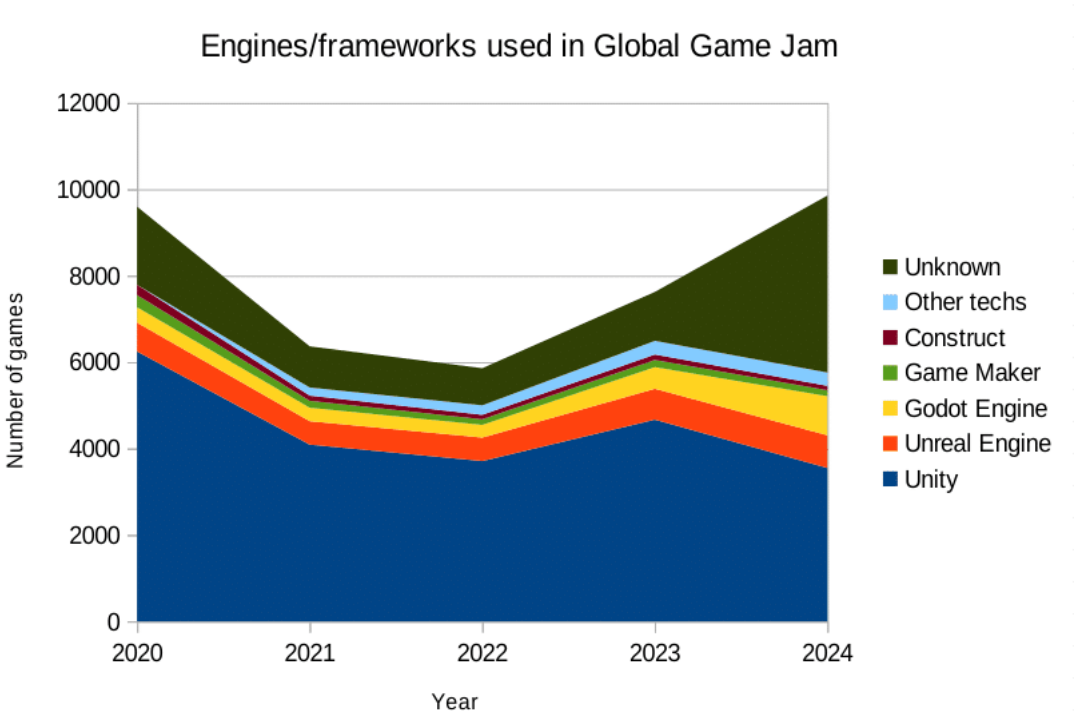

Figure 3: Engines/Frameworks used in Global Game Jam

A few examples of games developed using Godot are Brotato [22] and Endoparasitic [23].

In third place is Unreal Engine [24], developed by Epic Games. It allows developers to program in C++ or by using Blueprints, which is a graph-like visual scripting system. It mainly offers support for 3D and is well known for its games which feature outstanding graphics, but 2D games can still be developed in Unreal. Though Unreal offers support for shader programming, usually the Material node editors are used instead, and offers support for multithreading. A few well known games developed in Unreal Engine are Borderlands 3 [25] and Sea of Thieves [26].

After considering all these game engines, I chose Unity because it is more lightweight and has long term support. It also has everything needed for developing Cavernous Expedition such as support for shaders, the asset store and animation support. While Godot was also an attractive choice, the fact that it is still new to the market with the possibility of unexpected issues made me choose Unity. Unreal Engine is not really suited for Cavernous Expedition as outstanding graphics are not necessary for it.

3 REQUIREMENTS ANALYSIS

Cavernous Expedition is a first-person adventure puzzle game that takes place in an underground cave system which the player must explore by solving the puzzles that block their way. As it is a first-person game, the basic functionalities of moving and looking around must be implemented. For the adventure part, an interesting and realistic world has to be provided, in this case taking the form of a large cave system. Lastly, for the puzzles, the player must have to solve fun challenges that will test their problem solving skills.

3.1 Controls

As mentioned previously, it is important for the player to be able to traverse the world and explore the cave system, while also solving puzzles. Because of that, the controls must include:

- Player movement: The player can move around using the W,A,S,D keys, sprint by pressing Left-Shift while moving, and jump by pressing Space;
- Camera control: The player's point of view is controlled using the mouse movement;
- Interaction: The player can interact with various objects using the E key;
- Pausing: The player can pause the game by pressing the Escape key.

3.2 Game mechanics

For Cavernous Expedition to be an adventure puzzle game, multiple game mechanics must be implemented. They will shape the game and offer it its own feeling, which will differentiate it from other similar games.

3.2.1 Dynamically generated cave system

The most important feature of the game is the cave procedural generation. This feature is responsible for creating the world in which the game takes place and ensures a different experience each run. The following characteristics represent important parts of this feature:

- Various cave types: The player will encounter different types of caves from massive caverns to narrow tunnels, each with its own distinct shape and form;
- Resource placement: Inside the caves the player will find various resources which will be needed for an easier time during the underground exploration;

- Puzzles: Each cave will contain one puzzle, which will require the player to find objects or hints needed to solve it. While the majority of the puzzles can be solved just using the items in close proximity, there are also puzzles that can have their solution in neighboring caves.

3.2.2 Resource management

For the game to give the feel of a real adventure, the players will have to search for resources and manage their hunger and light levels. The resources that can be found inside the caves are:

- Batteries: The player is equipped with a flashlight on their helmet, which in time loses its strength due to its power consumption. Because of this, the player must search for batteries lost by other adventurers which they can use to increase the strength of their flashlight. If the light level reaches its minimum, the player is only able to see one meter ahead, which will make the exploration considerably more difficult;
- Food and water: To survive, the player must search for food and water sources to keep their hunger in check. As the caves don't have water sources or natural food, the player must search for other adventurers' lost items (represented as apples and water bottles). The hunger level determines the movement speed of the player, meaning that the more hungry they are the slower they will be able to move.

3.2.3 Challenges and exploring

The main components of the gameplay are exploring and puzzle solving. The puzzles are of variable difficulty and challenge the player through various tasks, such as:

- Pressing multiple buttons: The player will find multiple buttons placed on the ground that must be held pressed at the same time. As the player can only keep one button pressed at once by themselves, they must search for heavy objects such as rocks which can be placed on the buttons to keep them pressed;
- Matching symbols: The player will find a machine with multiple buttons labeled using various letters from the alphabet and must match the correct letters by coloring them in the same color through interacting with the buttons. Hints that specify the correct pairs of letters will be found on the walls of the puzzle cave room;
- Ordering symbols: The player will find a machine with multiple buttons labeled using various letters placed in a straight line and must swap them to achieve the correct order. The buttons can be swapped by interacting with them and hints specifying the correct order can be found on the walls of the puzzle cave room;
- Assembling a key: The player will find a machine with a white crafting board and a key hole. They must search the puzzle room for key fragments that can be picked up

and placed on the white board. Once all the fragments have been placed on the white board, the player will be able to pick up the key and use it to unlock the key hole;

- Hitting crystals: The player will find multiple crystals in the puzzle room which emit a certain sound when hit (interacted with), each in a different pitch. The player must find the puzzle machine which contains a hint regarding the order in which the crystals must be hit, which takes the form of multiple musical notes placed at different heights, each suggesting a certain pitch;
- Showcasing gems: The player will find multiple pillars in the puzzle room, each with a symbol representing a certain gem. The player must find those gems in any of the already opened rooms, and place them on top of the correct pillars. Once the correct gem is placed on top of a pillar, it will begin spinning, and can't be interacted with anymore.

Each cave has tunnels connected to it, which contain a gate, blocking the way. Once the cave room's puzzle is completed, all the gates associated with that cave open, allowing the player to progress further.

The exploration component of the game is showcased with the help of a minimap, which allows the player to see how much of the cave system they have discovered and what it looks like. The minimap is also useful to help the player understand their position and retrace their steps if needed.

4 THE PROPOSED SOLUTION

Cavernous Expedition takes place in an underground world which stands out through its cave system, with multiple tunnels linking together the caverns. Each cave has its own challenges and can take various forms from massive caverns to narrow passages. The player takes the role of an explorer who wants to discover the mysteries hidden in the depths of the newly found cave system. Their purpose is to map the entire cave system and discover what treasures lie inside.

To achieve the described game with the previously mentioned requirements, multiple classes were used, which work together as shown in Figure 4.

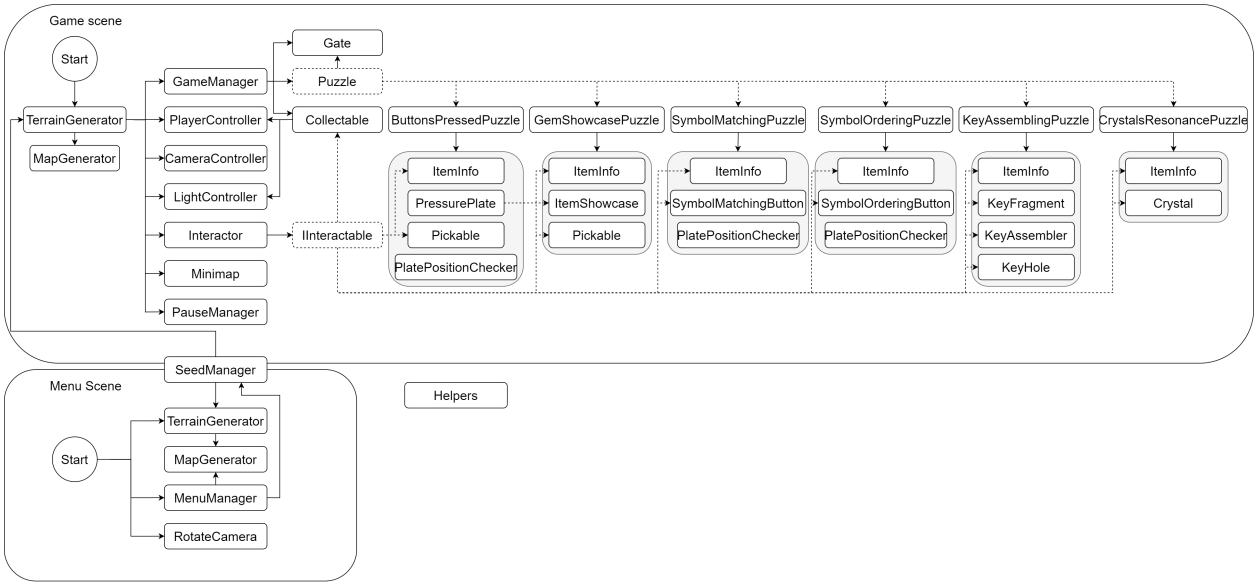

Figure 4: Workflow

4.1 Terrain

The terrain is one of the key points of the game, as everything revolves around the cave system which must be explored by the player. To achieve a realistic cave system, two important algorithms were used: Cellular Automaton [27] and Marching Cubes [28]. The two classes that use these algorithms are MapGenerator, which uses Cellular Automaton to generate a 2D cave system; and TerrainGenerator, which gets the 2D cave system from MapGenerator, creates a 3D volume out of it and uses Marching Cubes to generate the terrain. To achieve a more realistic terrain, two noise maps obtained using Perlin Noise [29] were used to create

bumps of various sizes on the floor and ceiling. The result is a blocky 3D cave system which can be seen in Figure 5.

Figure 5: Example of a generated cave system

For it to feel like a real cave system as much as possible, it was colored using different nuances of brown with a noise texture to give it a bit of imperfections. The texture was applied using Triplanar Mapping [30] to avoid places where the texture would be stretched.

With the terrain generation procedure in place, a seed is needed to dictate what the Cellular Automaton and the Perlin Noise generate so that by using the same seed, the same world is generated, while using different seeds mean obtaining completely different worlds. This seed is taken by the TerrainGenerator class from the SeedManager class.

4.2 Player and Camera

When the terrain is done generating, the player is spawned in one of the caves. Then, the PlayerController and CameraController scripts handle the basic functionalities of moving and turning. The PlayerController allows the player to move using the W,A,S,D keys, as well as sprint using the Left-Shift key and jump using the Space key. Through those functionalities the player can traverse the terrain and work towards solving the puzzles.

The CameraController is responsible for rotating the camera by using the mouse movement and to move the camera along with the player. The camera represents the player's point of view, so it uses perspective projection to achieve a first person effect.

4.3 Interactables

To solve the puzzles and collect resources, the player must have a way to interact with certain objects. This is done through the Interactor class which allows the player to press the E key when they want to interact with something. For the interaction to be successful, the player must look at the object they want to interact with, or else no interaction occurs. Also, for the player to know what the interaction will do, when looking at an object which they can interact with, an interaction description is shown at the top side of the screen which tells them what that interaction does.

The interaction effect depends of what object the player interacts with, as each interactable object implements the IInteractable interface to specify the effect.

4.4 Puzzles

The puzzles represent the main element of gameplay in Cavernous Expedition. They represent the challenges the player must overcome while navigating through the caverns and are the main obstacle stopping the player from exploring. The class responsible for choosing which puzzles to spawn is the GameManager.

After the terrain is done generating, it randomly (based on the seed) selects one puzzle from the pool of puzzles for each cave, and prepares everything for each chosen puzzle. Between the caves, there are multiple gates placed, which block the player's path and must be opened through completing the puzzles. These gates use the Gate class and are placed by the GameManager inside the tunnels that connect the caverns. All the puzzles extend the abstract Puzzle class which offers basic functionalities that each puzzle needs, such as what happens when the puzzle is completed, and also stores various information needed by each puzzle. Each puzzle also uses the InfoButton class, which implements the IInteractable interface, allowing the player to interact with it with the purpose of finding out more details about that puzzle. Other than that, each puzzle has a completion indicator represented as a red cube, which, when the puzzle is completed, turns green.

There are multiple puzzles, as mentioned in the previous section, each using various classes to achieve the required functionalities.

4.4.1 Multiple buttons pressed

This puzzle tasks the player with holding multiple buttons pressed at once. Because the player can only be at one place at a time, and the buttons can be found anywhere in the puzzle's cave, the player must find objects heavy enough to hold the buttons pressed when placed on top of them. The player will find in this cave a machine with multiple orange indicators,

multiple buttons and multiple (one less than the buttons) rocks. The puzzle itself uses the ButtonsPressedPuzzle script which extends the Puzzle class and is responsible for placing the buttons and the rocks, checking how many buttons are pressed and changing the indicators' color to yellow when their corresponding button is pressed. The buttons use the PressurePlate class which checks if a certain object is placed on top of them. The PlatePositionChecker script is also used to verify if the buttons are placed correctly (not in the walls or ground). The rocks have their models taken from the LowPoly Rocks [31] package from the Unity asset store. They use the Pickable class, which implements the IInteractable interface, allowing the player to pick them up and place them where needed. The puzzle can be seen in Figure 6.

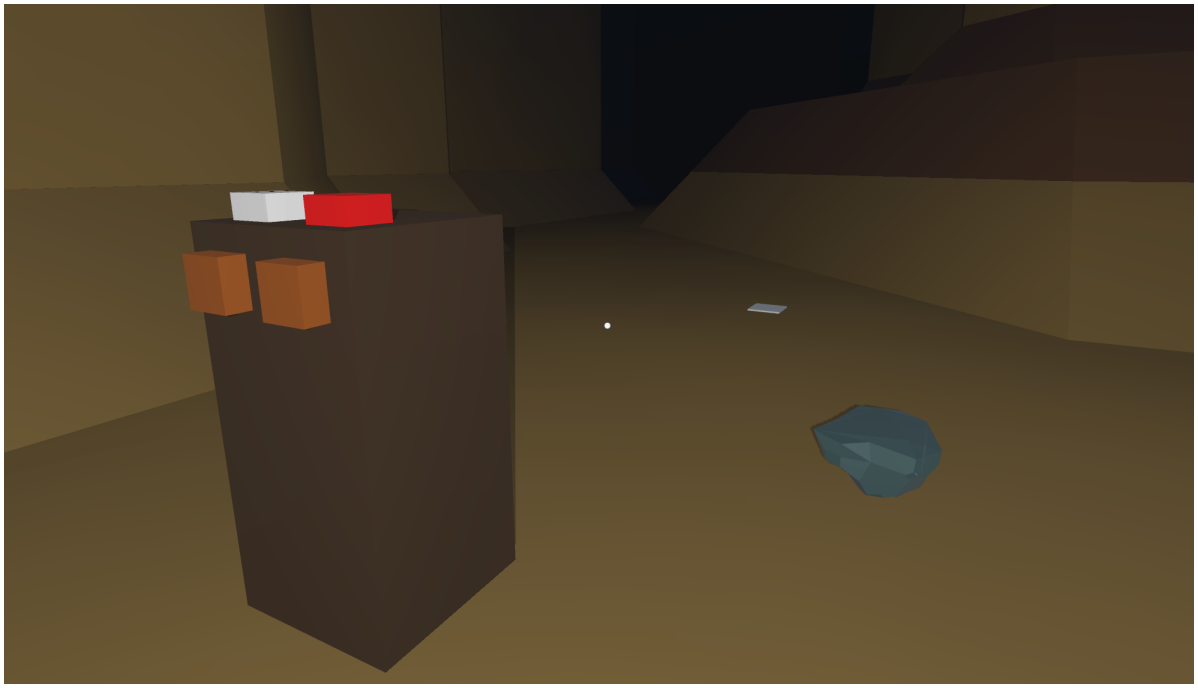

Figure 6: Puzzle with two buttons which must be pressed simultaneously

4.4.2 Symbol matching

This puzzle tasks the player with finding the correct way to match multiple symbols. They have to form pairs by coloring the symbols in the same color. The player will find in this cave a machine with multiple buttons placed in two vertical columns, each with a letter inscribed on it, and multiple hints which take the form of wooden signs attached to the walls. The puzzle uses the SymbolMatchingPuzzle script which extends the Puzzle class and is responsible for placing hints and monitoring each button's current color. The buttons use the SymbolMatchingButton class, which implements the IInteractable interface, allowing the player to interact with the buttons in order to change their color. The hints have the correct pairs written on them and use the PlatePositionChecker class to verify if the hint is placed correctly (not in the walls). The puzzle can be seen in Figure 7.

Figure 7: Puzzle with four symbols which must be matched

4.4.3 Symbol ordering

For this puzzle, the player needs to order multiple symbols in a certain way. They have to swap the symbols until the correct order is achieved. In this cave, the player will find a machine with multiple buttons placed in a straight line, each with a letter inscribed on it, and multiple hints which take the same form as the hints from the previous puzzle. The puzzle uses the SymbolOrderingPuzzle script which extends the Puzzle class and is responsible for placing hints and monitoring the current order of the symbols. The buttons use the SymbolOrderingButton class, which implements the IInteractable interface, allowing the player to interact with the buttons. By interacting with two different buttons, they will swap places. The hints have information on them regarding the way in which the symbols have to be ordered, such as "X is first", or "X is before Y", where X and Y are symbols. They also use the PlatePositionChecker class to verify if their placement is valid. The puzzle can be seen in Figure 8.

4.4.4 Assembling a key

This puzzle requires the player to assemble a key and use it to unlock the puzzle's mechanism by turning the key inside the key hole. They have to search around the cave in order to find key fragments, which are used to craft the key. In this cave, the player will find a machine with a white board and a key hole, and three key fragments. The puzzle uses the KeyAssembling-Puzzle script which extends the Puzzle class and is responsible for placing the key fragments around the cave room. The fragments use the KeyFragment class, which implements the

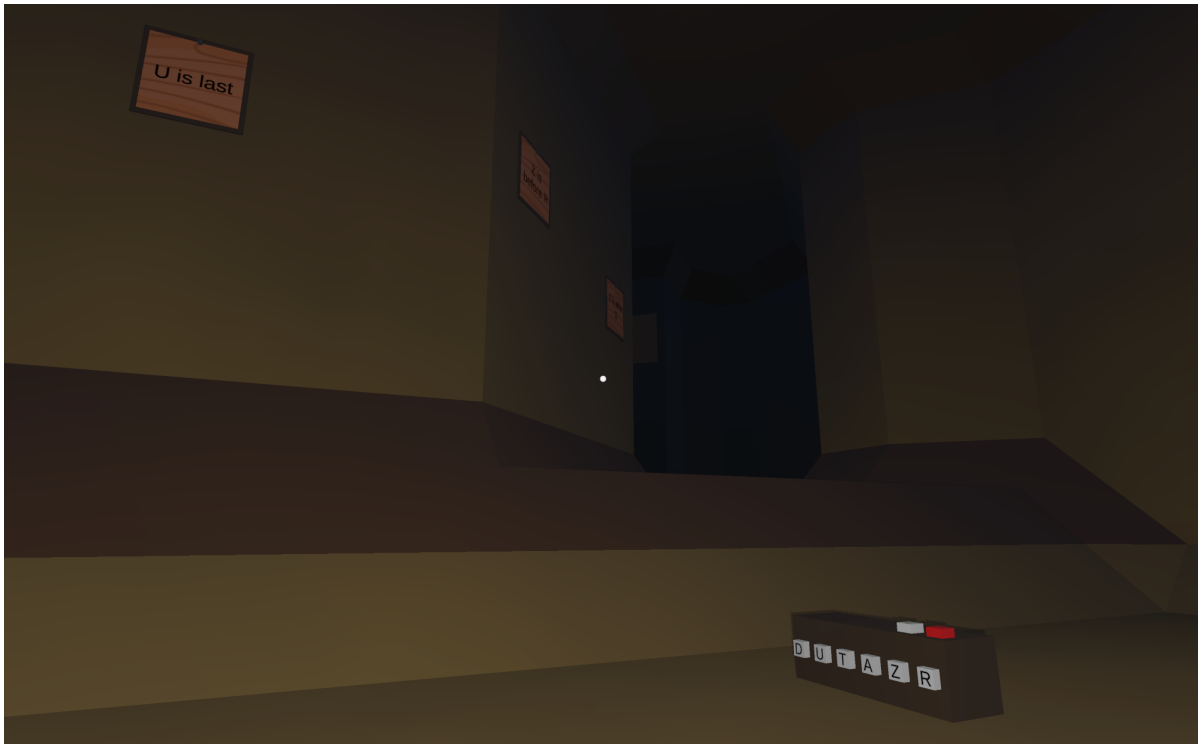

Figure 8: Puzzle with six symbol which must be ordered

IInteractable interface, allowing the player to pick them up and store them. The white board uses the KeyAssembler class, which also implements the IInteractable interface, allowing the player to place the currently stored key fragments onto the board, and, when all three have been placed, to pick the key up. Lastly, the key hole uses the KeyHole class which implements the same IInteractable interface and allows the player to use the key to complete the puzzle. The puzzle can be seen in Figure 9.

4.4.5 Crystal resonance

For this puzzle, the player must hit multiple crystals in a certain order for them to play a specific tune. They will find in this cave multiple crystals, of different sizes and colors and a machine with a hint containing musical notes which represent the correct order in which to hit the crystals. The puzzle uses the CrystalResonancePuzzle script which extends the Puzzle class and is responsible for placing the crystals inside the cave room and for storing the current note sequence produced by the player, while also updating the hint. The crystals have their models taken from the Translucent Crystals [32] package from the Unity asset store. They use the Crystal class, which implements the IInteractable interface, allowing the player to interact with them to produce a hit. The hint is updated constantly to reflect the current state of the sequence of notes, by coloring the musical notes green if correct, or red otherwise. The puzzle can be seen in Figure 10.

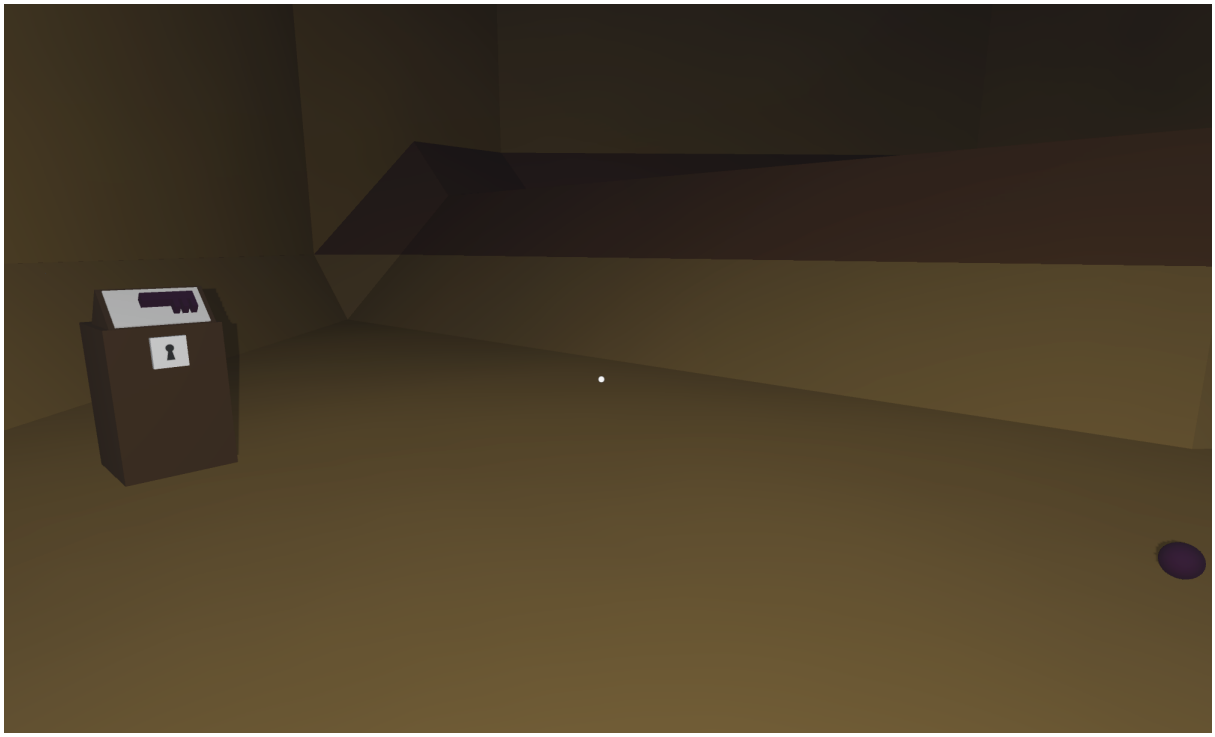

Figure 9: Puzzle with a key which must be assembled

4.4.6 Gem showcase

This puzzle tasks the player with finding various types of gems which must be placed on their specific showcase spot. The player will find in this cave multiple pillars, each with a hint representing a drawing of their corresponding gem, and throughout the previously explored caves, they will find multiple gems of various shapes and forms. The puzzle uses the GemShowcasePuzzle scripts which extends the Puzzle class and is responsible for placing the pillars in the cave room and the gems in any of the already accessible cave rooms. The pillars have their model taken from the Low Poly Pillar Set [33] from the Unity asset store. They use the ItemShowcase script which extends the PressurePlate class in order to use its functionality of checking if a certain object is placed on top of them, and adding a new functionality of spinning the item placed on top (if it is the correct one). The gems have their models taken from Simple Gems Ultimate Pack [34], also from the Unity asset store. They use the Pickable class, allowing the player to pick them up and place them where needed. The puzzle can be seen in Figure 11.

4.5 Resources

Another feature of the game is the resource management, which makes the player pay attention to other items, other than the ones required by the puzzles. The GameManager script is responsible for placing the resources, one of each in every cave room. All the resources use the Collectable class, which implements the IInteractable interface, allowing the player

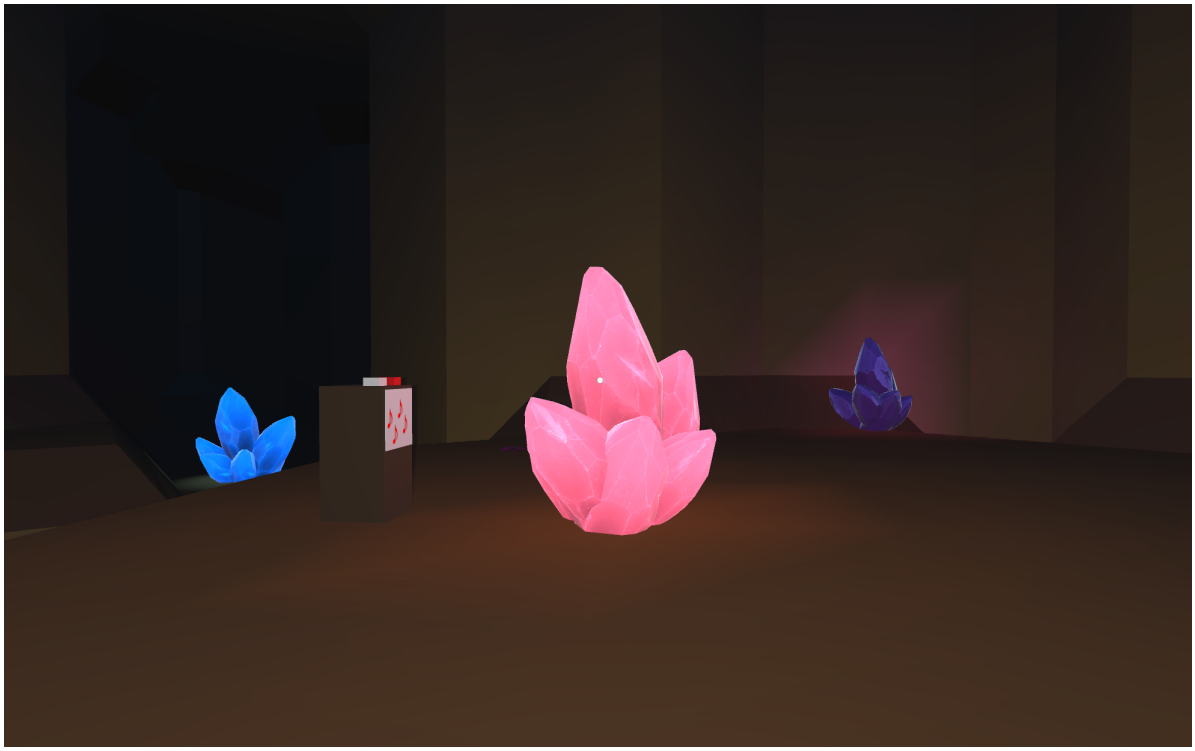

Figure 10: Puzzle with four crystals which must be hit in a specific order

to collect said items to increase their hunger and light levels. This class interacts with the PlayerController scripts, as it is responsible for the hunger level, and with the LightController class, which is responsible for the light level and for the intensity of the player's flashlight. The resources present in the game have their models taken from the Survival Kit Lite [35] package from the Unity asset store and they are, for hunger, apples and water bottles, and for light, batteries, which can be seen in Figure 12.

4.6 User interface (UI)

There are multiple UI elements used to relay information to the player. They are the minimap, the levels of hunger and light represented using sliders, colored orange and yellow, respectively, a crosshair, and an interaction text. The minimap represents one of the goals of Cavernous Expedition, more specifically mapping the entire cave system. The Minimap script is responsible for the autocompletion of the map and draws on the minimap around the player's position, which is also marked using the color red. To complete the map the player has to move around the cave system, and the minimap will gradually get completed. The hunger and light sliders reflect the state of player's hunger level, and the state of the flashlight's battery, respectively. The interaction text only appears when the player is looking at an interactable object. These elements have been placed taking into consideration the player, who should see as much of the game as possible, with UI elements only occupying a small portion of the screen. This is why the minimap is placed in the upper-right corner, the hunger and light level in the upper-left

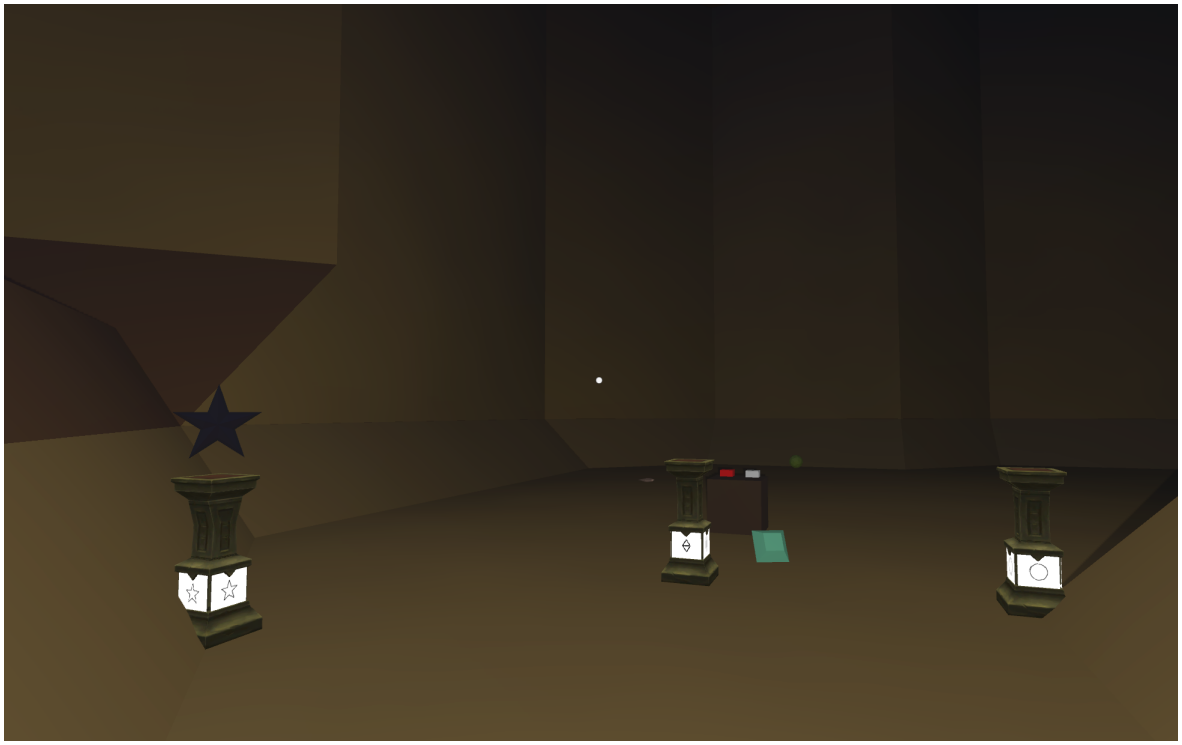

Figure 11: Puzzle with three showcase pillars which must have the correct gems placed on top

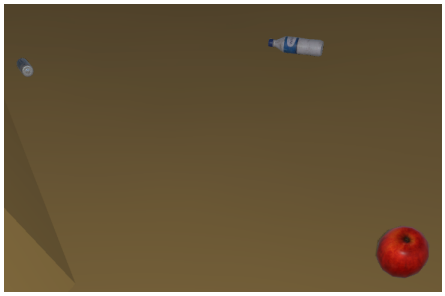

Figure 12: Resources (apple, water bottle, battery)

corner, the interaction text in placed on the top side, in the center, and the crosshair is a small dot placed in the middle of the screen. The in-game UI elements can be seen in Figure 13.

Another part of the UI is the pause menu. The menu puts a grey transparent overlay on the whole screen and multiple buttons and sliders on top of it. The pause menu is handled by the PauseManager class, which is responsible for pausing the game when the player presses the Escape key. Once the game is paused, the player will be able to resume the game, restart the game (using the same seed), change the sound effects' volume and the mouse sensitivity, and go back to the main menu.

Figure 13: The in-game UI

4.7 Main menu

Everything that was talked about in this chapter until this point refers to the game scene, where the actual game takes place. But besides this scene, there is the menu scene, which is the first scene the player sees when opening the game. The main menu presents the title of the game, an input field in which to write a seed, if needed, and three buttons, one which starts the game, one which opens the options tab, revealing two sliders, controlling the sound effects' volume and the mouse sensitivity, and one that closes the application. To the right side of the buttons, a cave map can be seen. In the background, there can be seen a randomly chosen terrain, in which a camera slowly rotates. The terrain is generated in the same way as in the game scene, through the TerrainGenerator and MapGenerator classes, and the camera is rotated using the RotateCamera script. Lastly, the UI is controlled through the MenuManager script, which also handles the cave map, which shows the player what the current seed (chosen randomly, or by the player) would generate. To obtain this map, the MenuManager uses the MapGenerator class as well. The menu can be seen in Figure 14.

For the game scene to obtain the seed from the main scene, the SeedManager class was used, which stores the currently chosen seed and is not destroyed when loading the game scene, so that the TerrainGenerator can get the correct seed.

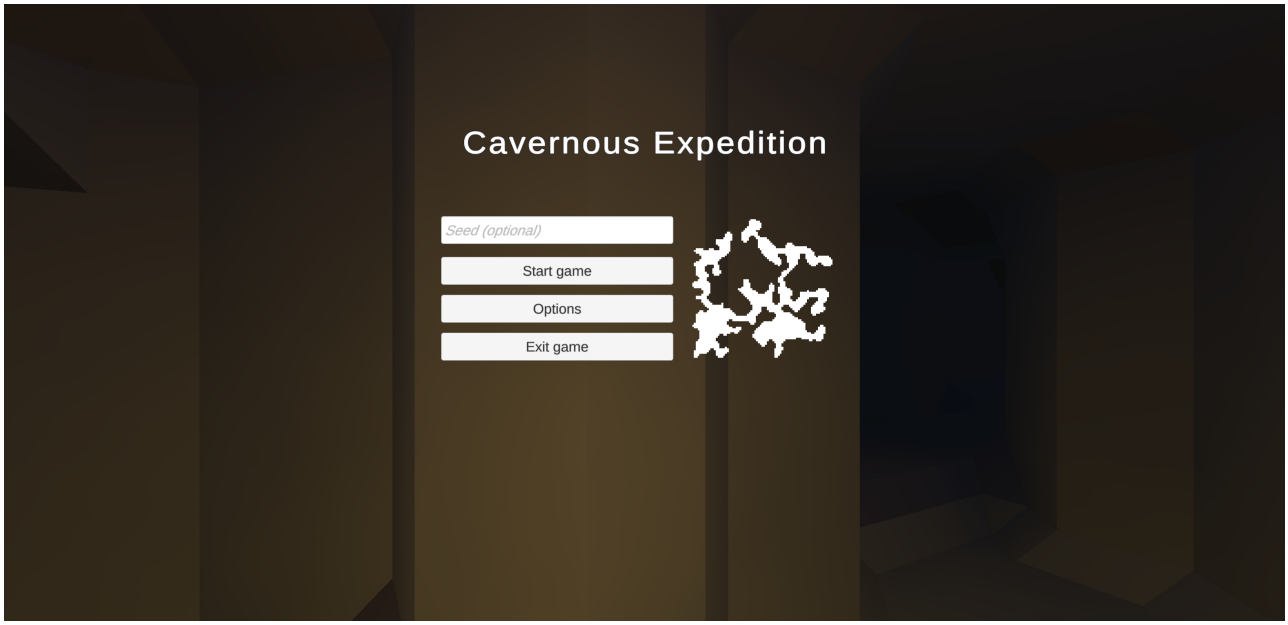

Figure 14: The main menu

4.8 Others

There are multiple sounds which can be heard throughout the game, such as the player walking, the crystals being hit, the buttons being pressed and the puzzles being completed. All these sounds have been taken from Pixabay [36].

There is also one more script to be mentioned, which is the Helpers class. As its name implies, it is a helper class which provides various utilities frequently required by the other classes.

5 IMPLEMENTATION DETAILS

As mentioned previously, Cavernous Expedition was developed using the Unity game engine, more specifically version 2023.2.2f1. Because Unity offers a multitude of features and utilities, there were many its components and functionalities used during the development of the game. The majority of them are basic features that most game engines offer, but still have to be covered so that everything makes sense.

5.1 Preliminaries

Unity offers features in the form of editor functionalities, such as utilities or ready-to-use components, but there are also features in the form of code libraries and namespaces.

5.1.1 The editor

This category contains the majority of features used that affect the Unity editor.

- Scene: A scene is a place where a user can place and configure objects. A game can contain a single scene, or multiple scenes through which the game can switch;
- Object hierarchy: Inside a scene, all objects form a hierarchy, meaning that certain objects can be placed as childs of other object which influences the way they are moved (as the parent object moves, so does the child object);
- Components: Each object can hold multiple components, which are essentially scripts/classes which define the behaviour of that object;
- Prefab: Prefabs are a way of storing a configured object to be used as a reusable asset;
- Layers: Each object has a layer associated with it, which can be used to differentiate between multiple objects in the scene;
- Animations: The animation system used by Unity offers the possibility to create a sequence of changes to an object, which are stored as a state in the animation controller, and can be played when needed;
- UI: Unity offers multiple UI elements which can be used to display various things on the screen, such as text, images, sliders, buttons, input fields and many more. All of these must be placed inside a Canvas object, which is rendered either on the screen, or in world space.

5.1.2 The code

The features presented here are essentially already prepared scripts that Unity offers to simulate a variety of functionalities, such as collisions, physics and many more.

- GameObject: This is the class that defines an object, whether present in the scene or stored as a prefab. It offers various information about said object and allows searching for other components held by the object;
- Transform: This is the only component that has to be on every object. It specifies the object's position, rotation and scale;
- Collider: A collider is a script that defines the shape of the object to be used by the physics system. There are multiple basic shapes of colliders (box, sphere, etc.), but complex objects can also have correctly fitted colliders with the help of mesh colliders, which use the mesh to determine the shape;
- Rigidbody: This script is used to simulate the forces applied on the object. It allows the object to be affected by gravity, but also other forces which can be applied through collisions or manually in code;
- Mesh filter and renderer: These are two scripts that work hand in hand. The mesh filter holds the object's mesh information, which is used by the mesh renderer to render said mesh;
- Audio listener and source: As their names imply, the audio listener allows the player to hear the sounds played by the game using the audio source;
- Animator: This script is used to play and switch between animations. This can be done by creating a state machine defining the order in which the animations must be played, or manually through the code;
- MonoBehaviour: This is a built-in class which offers lifecycle methods such as the Start method, which is called in the first frame of the game, or the Update method, which is called every frame, and many more. Every class that has wants to implement behaviour must extend this class;
- Random system: Unity provides its own pseudo-random number generator which is state based. This means that the state dictates what number is generated and makes seeded random [37] behaviour possible.

5.2 Terrain generation

For the terrain generation to be realistic, multiple algorithms were used, each having a significant role in the achieved result, but two of them stand out.

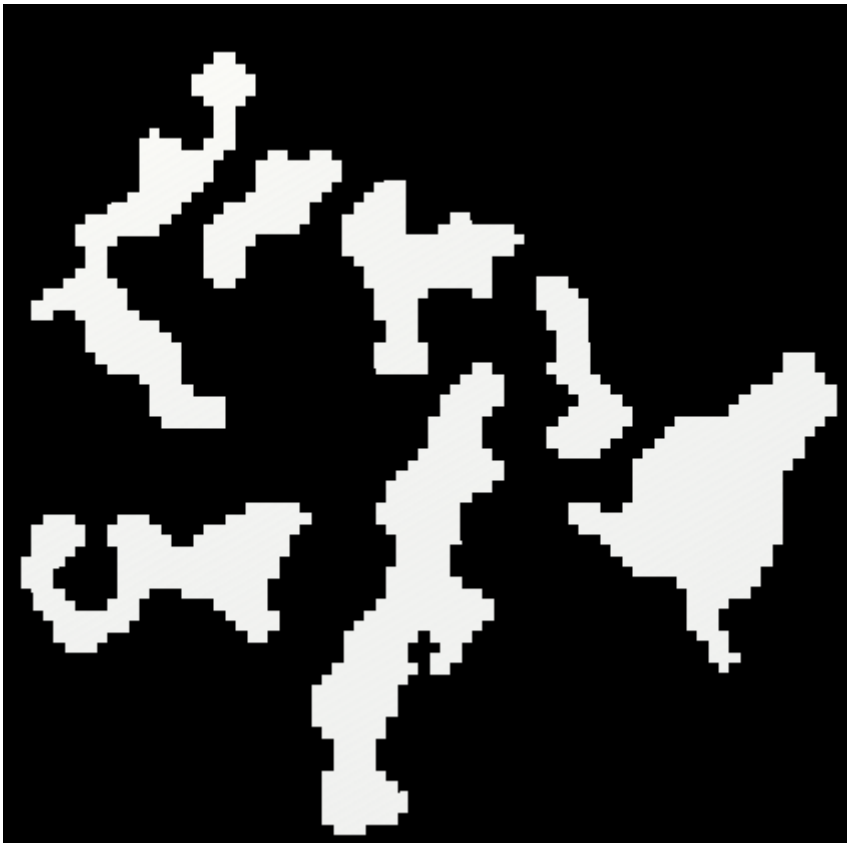

Figure 15: Cave system after Cellular Automaton

5.2.1 Cellular Automaton

The first algorithm, Cellular Automaton, is responsible for generating a 2D cave system and is implemented by the MapGenerator class. This class doesn't extend the MonoBehaviour class, meaning that the other scripts have to create an instance of this class when needed. A cellular automaton is a collection of cells on a grid that have a finite number of states, and use a set of rules based on the states of the neighboring cells to evolve over a number of steps. In this case, the grid represents the map of the cave system and is represented through a 2D array of integers with a size of 80x80. Each cell has two states: wall (-1) and cave (1). The rule used is based on the number of neighboring cells that are walls and specifies that if this number exceeds a threshold of 4, the cell is also a wall, otherwise the cell is a cave. Before starting, the random system's seed is set using the InitState method, which sets the state of the pseudo-random number generator. In the beginning of the algorithm, the cells' states are randomly chosen to achieve a fill percentage of about 47%, and 5 steps are performed with the specified rule. After the steps are completed, the map consists of multiple caves of various sizes, some being too small to be useful, so they are eliminated by changing their state from caves to walls. The same happens for small walls inside caves, their state is changed from walls to caves. This is done by determining the size of each cave/wall and comparing it with a size threshold (in this case 75). One such cave system obtained after the Cellular Automaton is completed can be seen in Figure 15.

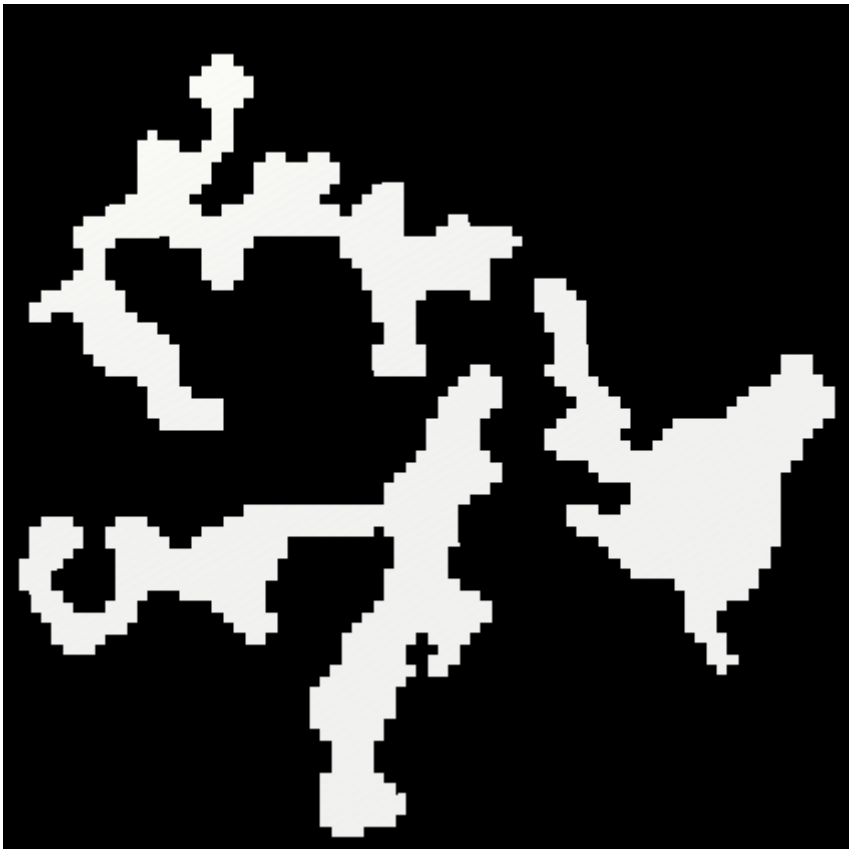

Figure 16: Cave system partially connected

5.2.2 Ensuring connectivity between caves

For the caves to be usable they need to be connected through tunnels that allow the player to navigate between them. To do this, for each cave without a connection, the closest cave is found and connected to. With just this, the caves form multiple groups, but there are still groups not connected to each other (as shown in Figure 16), so to solve this the previously mentioned method is used in a modified way. Firstly, the largest cave is chosen as the main cave, and the caves are split into those connected to the main cave, and those not connected to it. Then, for each cave not connected to the main cave, the closest cave connected to the main one is searched. The connection that is actually chosen is the one with the smallest distance between all found possible connections, and then the algorithm is applied again until there are no more caves not connected to the main cave. The reason that only the smallest connection is chosen is to avoid unnecessarily long tunnels and to achieve a more realistic cave system. The result can be seen in Figure 17.

The actual tunnels are composed of multiple small circles (in this case with a radius of 2). The number of circles that form the tunnel is determined by the length of the vector between the two ends of the tunnel, rounded up. To create the tunnel, the circles are placed on the vector in an equidistant way and for each cell of the map it is checked whether it is inside one of the circles, in which case its state is changed from wall to cave.

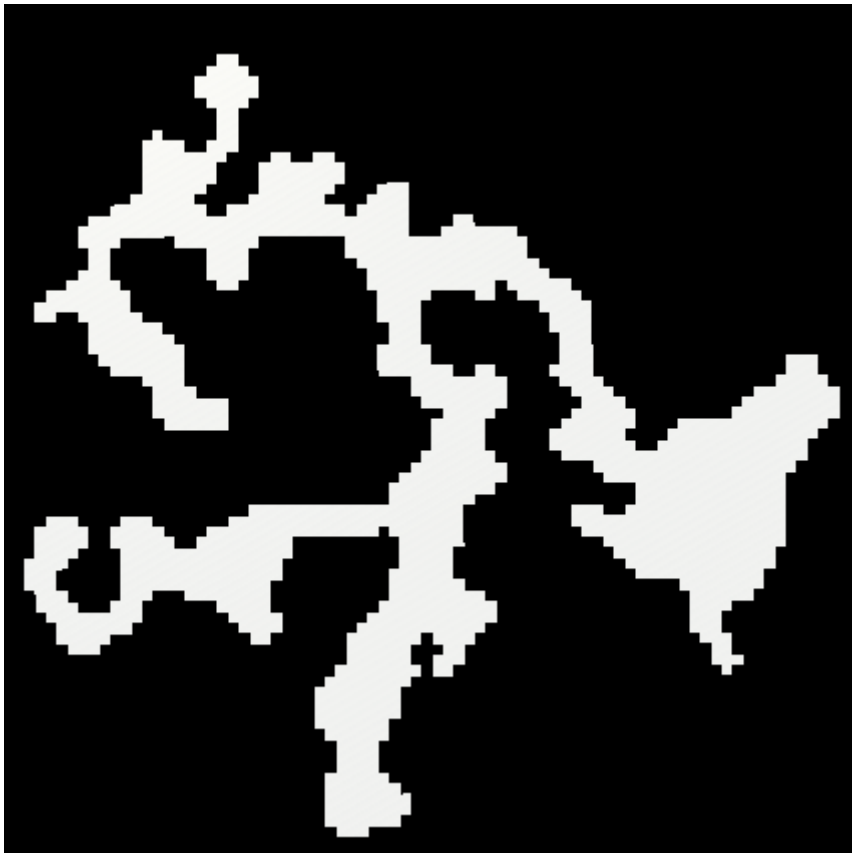

Figure 17: Cave system fully connected

5.2.3 Preparing the data volume

Before generating the terrain mesh, a data volume must be prepared. The data volume can be seen as a 3D array, for which I used a 3D RenderTexture with a format that allows each element to store a 32 bit float (UnityEngine.Experimental.Rendering.GraphicsFormat.R32_SFloat [38]). To prepare this data volume I used a compute shader [39]. Compute shaders are essentially programs that run on the GPU, outside of the normal rendering pipeline and use the massive parallel capabilities of the GPU. For the purpose of preparing the data volume, for each element of the volume, a thread determines its value. The compute shader is dispatched by the TerrainGenerator class, after the 2D map is done generating.

To turn the 2D cave system generated previously into a 3D data volume, the map is copied on multiple levels of the data volume, one on top of the other to achieve a given height (in this case a height of 8). The lowest and highest levels are considered to be completely filled with walls. Lastly, to add a bit more realism, two noise maps are used to add bumps on the floor and ceiling. Both maps are generated using the Mathf.PerlinNoise [40] method with the seed as offset. To add them to the data volume, for each 3D point that is inside the cave system (has a value of 1 in the initial 2D map) its value is modified by the corresponding noise map's value multiplied by a noise multiplier (for the floor the multiplier is 2, and for the ceiling 3). The compute shader HLSL code can be seen in Figure 18.

```
RWStructuredBuffer<int> map2d;
RWStructuredBuffer<float> floorNoise;
RWStructuredBuffer<float> ceilingNoise;
int2 mapSize;

RWTexture3D<float> map3d;
int height;

float3 worldSize;

float noiseCeilingHeight;
float noiseFloorHeight;

[numthreads(8,8,8)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    if (id.x >= mapSize.x || id.y >= height || id.z >= mapSize.y)
        return;

    if (id.y == 0 || id.y == height - 1)
    {
        map3d[id] = -1;
    }
    else if (map2d[id.x * mapSize.y + id.z] == 1)
    {
        if (id.y < height / 2)
            map3d[id] = id.y - floorNoise[id.x * mapSize.y + id.z] * noiseFloorHeight;
        else
            map3d[id] = height - (ceilingNoise[id.x * mapSize.y + id.z] * noiseCeilingHeight) - id.y;
    }
    else
    {
        map3d[id] = -1;
    }
}
```

Figure 18: Compute shader used to generate a 3D volume out of a 2D map

5.2.4 Marching Cubes

The second important algorithm, Marching Cubes, is responsible for generating a triangle mesh from a given data volume. It works by forming cubes out of adjacent 3D points of the data volume, and iterating (marching) over them. These cubes dictate what triangles should the mesh contain. Each corner of a cube can have two possible states, inside or outside the mesh. In this case, this is determined by its value, if it is lower than 0, the corner is inside the mesh, otherwise it is outside. As a cube has 8 corners, it means that there are a total of 256 possible configurations of a cube, but the majority are mirrored versions of one another, which leave 15 unique cases that can be seen in Figure 19 [41].

To implement the Marching Cubes algorithm, I used another compute shader which is also dispatched by the TerrainGenerator class. This time, for each cube there is a thread which determines its configuration and adds the necessary triangles to a buffer. To turn each configuration into the corresponding triangles I used a precalculated look-up table [42]. For the CPU to be able to get the triangles calculated on the GPU, a ComputeBuffer [43] is needed, which is created using the Append compute buffer type and is set accordingly using the SetBuffer method. On the GPU side, an AppendStructuredBuffer is used to store the triangles. To represent a vertex, which needs a position and a normal, I used a Vertex struct comprising of two float3s on the GPU side, and a VertexInfo struct comprising of two Vector3s on the CPU side. Even though the Append method of the buffer is atomic, since a triangle

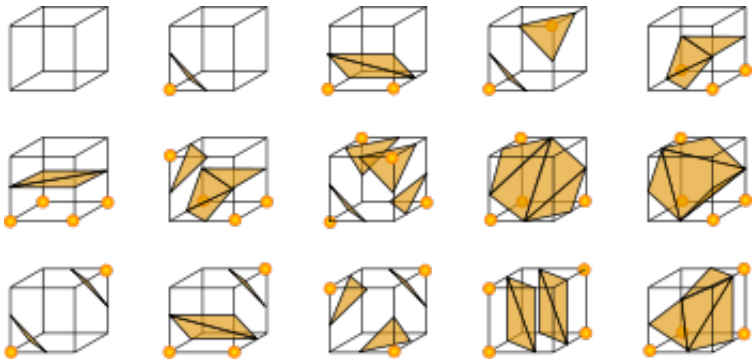

Figure 19: Marching cubes cases

is formed of three vertices, I would need to append three times, which wouldn't be atomic anymore. To solve this, I used a struct comprising of three Vertex elements, and I append this structure instead of three separate Vertex items. Since the buffer is instantiated using the size of a VertexInfo struct, on the CPU side, which means six floats, a triangle struct is considered to be three such structs. All of these structs can be seen in Figure 20.

```
struct Vertex
{
    float3 position;
    float3 normal;
};

struct Triangle
{
    Vertex a;
    Vertex b;
    Vertex c;
};
```

```
5 references
public struct VertexInfo
{
    public Vector3 position;
    public Vector3 normal;
}
```

Figure 20: GPU structs compared with CPU struct

Another important detail is the way the vertices' positions are calculated. The Marching Cubes algorithm places the vertices along the edges of the respective cube. To keep things simple, I always placed the vertices in the middle of their respective edge, which gives a more blocky terrain. To achieve the desired size for the terrain, when the vertices are calculated, a multiplier is used (in this case 8). With the positions calculated, there are only the normals left. To calculate the normals, I used the following formula to determine the perpendicular vector to a plane determined by three points, where N is the normal and A, B and C are the three points:

$$N = (B - A) \times (C - A) \tag{1}$$

Lastly, after the compute shader finishes calculating all the vertices/triangles, they must be taken back from the GPU. In the case of Append ComputeBuffers, because their size is unknown, firstly another ComputeBuffer, this time of type Raw, must be used to get their size from the GPU using the CopyCount method. Afterwards, the CPU can get the elements

of the Append ComputeBuffer by specifying the number of elements to take using the GetData method. After this, through the use of a dictionary, as seen in Figure 21, the duplicate vertices are eliminated to optimize the size of the mesh. Once only the necessary vertices remain, a MeshFilter and MeshRenderer components are added to the game object, and then the vertex array, the triangle array and the normals array are added to a newly created mesh, which is then sent to the MeshFilter.

```
triangleCount = 0;
foreach (VertexInfo vertex in vertexArray)
{
    int existingIndex;
    if (vertexDictionary.TryGetValue(vertex, out existingIndex))
    {
        meshTriangles.Add(existingIndex);
    }
    else
    {
        vertexDictionary.Add(vertex, triangleCount);
        meshVertices.Add(vertex.position);
        meshNormals.Add(vertex.normal);
        meshTriangles.Add(triangleCount);
        triangleCount++;
    }
}
```

Figure 21: Mesh optimization

5.2.5 Material

For the terrain material I chose to use a Surface Shader [44], which is a way of writing shaders that interact with lighting, and is much easier to use than low level vertex/pixel shaders. Using a Surface Shader, one can modify properties of the surface, such as albedo color, which is what I needed.

I wanted to make the caves have a beige-brown look that would get darker with height, so that the ceiling would have a dark brown color, while the floor would have more of a beige color. To achieve this, I defined a number of colors (nuances of brown) for the shader to use, and used the Input structure to get the surface's world position and normal. The normal was used to determine if the surface was a floor or a ceiling by checking its y value, and afterwards, the position's y value was used to split the coloring between two colors. In case the surface was a wall, two colors were chosen, one brighter and one darker, and the surface's color was the result of linearly interpolating between the two chosen colors, based on the position's y value (height). The shader's main function, surf, can be seen in Figure 22.

To add some imperfections to the material, a noise texture was used. To avoid texture stretching, I used Triplanar Mapping, which solves the mentioned issue. Triplanar Mapping works by mapping the texture three times along the three planes (xOy, xOz and yOz), and calculating each mapping's influence using the surface's normal. Lastly, the three mappings are blended together using the calculated influences. After mapping the noise texture, I used

```
void surf (Input IN, inout SurfaceOutputStandard o)
{
    float4 noise = triplanarMapping(IN.worldPos, IN.worldNormal, textureScale, _NoiseTex);

    if (IN.worldNormal.y > 0.5) {
        if (IN.worldPos.y < minHeight)
            o.Albedo = lerp(_Ground1, _Ground1Dark, noise.r);
        else
            o.Albedo = lerp(_Ground2, _Ground2Dark, noise.r);
    } else if (IN.worldNormal.y < -0.5) {
        if (IN.worldPos.y > maxHeight)
            o.Albedo = lerp(_Ceiling1, _Ceiling1Dark, noise.r);
        else
            o.Albedo = lerp(_Ceiling2, _Ceiling2Dark, noise.r);
    } else {
        float4 wallColorLight = lerp(_Ground1, _Ceiling1, (IN.worldPos.y - minHeight) / (maxHeight - minHeight));
        float4 wallColorDark = lerp(_Ground1Dark, _Ceiling1Dark, (IN.worldPos.y - minHeight) / (maxHeight - minHeight));
        o.Albedo = lerp(wallColorLight, wallColorDark, noise.r);
    }

    o.Smoothness = _Glossiness;
}
```

Figure 22: Surface shader

it to interpolate between the initially chosen color and a darker tone of it.

5.3 Player controller

Controlling the player implies being able to look and move around. For that, the logic is split into two different scripts, CameraController and PlayerController which interact with various game objects under the Player game object, and various components under said game objects.

5.3.1 Camera controller

The Camera Controller handles everything related to the camera. The camera itself is a game object with a Camera component configured to use a perspective projection and default settings. The controller rotates the camera using the movements of the mouse, which are received using the Input.GetAxisRaw method. The movements received from the mouse are multiplied with a modifiable sensitivity value and with Time.deltaTime, which takes into account the framerate, in order to achieve a constant camera rotation regardless of framerate. Then, these values are added/subtracted to/from the X and Y rotations, which are then used to set the rotation of the camera. To achieve a realistic first-person controller, the X rotation, which rotates the camera up and down, is restricted to a maximum of 90 degrees. Another game object affected by the camera controller is the Orientation game object, which is an empty (no other components other than a Transform) game object used to determine the direction the player is facing. It is only rotated around the Y axis, unlike the camera which is rotated around the X and Y axes. This can be seen in Figure 23.

```
float mouseX = Input.GetAxisRaw("Mouse X") * Time.deltaTime * xSensitivity;
float mouseY = Input.GetAxisRaw("Mouse Y") * Time.deltaTime * ySensitivity;

yRotation += mouseX;
xRotation -= mouseY;
xRotation = Mathf.Clamp(xRotation, -90f, 90f);

transform.rotation = Quaternion.Euler(xRotation, yRotation, 0);
orientation.rotation = Quaternion.Euler(0, yRotation, 0);
```

Figure 23: Camera controller

5.3.2 Player movement

The Player Controller handles everything related to the movement of the player. The movement uses the physics system provided by Unity, so the Player object has a Rigidbody and also a Collider (more specifically a Capsule Collider). To move the player, the Player Controller gets the player's input with Input.GetAxisRaw method, which returns a float specifying in which direction to move the player, based on the W,A,S,D keys. These values are multiplied with the Orientation object's forward and right vectors in order to add a force in that direction. The actual force is added in the FixedUpdate method, which runs at a constant framerate, so the movement is handled the same regardless of the framerate. To jump, an Impulse type force is applied upwards, and to sprint, the movement speed used is increased.

To achieve a more realistic movement, different techniques are used depending if the player is on the ground (this is checked using raycasts with Physics.Raycast [45]), in the air, or on a slope. This techniques include changing the drag while in the air and disabling the use of gravity while on a slope to avoid the player slowly falling down the slope when no input is received. While moving on a slope, the direction in which the player moves is also changed by projecting the original direction onto the slope, to avoid the player running into the slope. When the player is moving or running, a walking or running sound is being played.

5.3.3 Interaction system

For the player to be able to interact with other objects in game, the Interactor script is used to send a raycast in the direction the player is looking, and if it hits an object with a component that implements the IInteractable interface, call its Interact method. This means that any object that should do something when the player interacts with it should have a component that implements the IInteractable interface, which has a single method, Interact, and two variables, one that specifies if that object can be interacted with in that moment, and one that is a string containing an interaction prompt, telling the player what that interaction would do. This prompt is displayed by the Interactor class in a textbox on the screen.

5.4 Game manager

The game manager is the one that coordinates everything related to the gameplay, such as spawning the player, the puzzles, the gates and the resources. The GameManager script is initialized, and then enabled by the TerrainGenerator class after the terrain is done generating. The game manager also follows the Singleton design pattern, in order for its instance to be unique and accessible from anywhere. It holds various information, such as all the positions used by the puzzles, the puzzle items, etc, and also references to the scripts held by the player.

The player is spawned using the Instantiate method in a randomly picked cave room, which is also considered the starting room. When spawning the player, a point is randomly picked from the 2D room and is converted to the coordinates of the 3D world using the MapToWorld method from the TerrainGenerator class. Then, for the player to not be spawned in the air, a raycast is sent downwards in order to get the position of the ground, which is the position where the player is actually spawned, with an offset to avoid them spawning in the ground. This can be seen in Figure 24.

```
if (Physics.Raycast(terrainGenerator.MapToWorld(playerPosition), Vector3.down, out RaycastHit hit))
{
    validPoint = true;
    GameObject player = Instantiate(playerPackage, hit.point + Vector3.up * 3, Quaternion.identity);
    cameraController = player.GetComponentInChildren<CameraController>();
    playerController = player.GetComponentInChildren<PlayerController>();
    lightController = player.GetComponentInChildren<LightController>();
    pauseManager = player.GetComponentInChildren<PauseManager>();
}
```

Figure 24: Spawning the player

After the player is spawned, for every cave room, starting with the one where the player was spawned, a puzzle is randomly picked and placed inside the cave, with one of each resource and with the gates in the tunnels connected to the cave. As the cave will open all its gates upon its puzzle completion, to iterate over the caves in an order similar to that in which the player will navigate through them, a Queue is used. When a cave room is popped from the queue, its neighboring caves are pushed into the queue, similar to a BFS algorithm [46].

When placing the puzzle, to avoid it facing the wall, the best direction for the puzzle to look at is picked out of the north, west, south and east directions. This is done by sending a raycast in each direction and calculating the distance to the wall hit by the raycast, picking the direction with the maximum distance. Then, to avoid the puzzle being in a slope or wall, a check method is used to verify if the puzzle is overlapping with the ground. This is done using the Physics.OverlapSphere method [47] by calling the IsPlacedCorrectly method from the Puzzle class, which uses multiple spheres inside the puzzle object to check for overlapping with colliders that are on the ground layer. This can be seen in Figure 25.

After the puzzle is successfully placed, the necessary gates are placed into the tunnels and rotated so that they block the passage. Once the gates have been placed, one of each resource

```
1 reference
public struct PuzzlePositionChecker
{
    public GameObject obj;
    public float radius;
}

1 reference
public bool IsPlacedCorrectly()
{
    for (int i = 0; i < positionCheckers.Length; i++)
    {
        Collider[] col = Physics.OverlapSphere(positionCheckers[i].obj.transform.position, positionCheckers[i].radius, groundLayer);
        if (col.Length > 0)
            return false;
    }

    return true;
}
```

Figure 25: Checking if a puzzle is placed correctly

is spawned in the cave. To avoid items spawning into each other, their spawning points are stored. Once every cave had its puzzle, gates and resources spawned, the puzzles are enabled.

5.5 Puzzles

All the puzzles have their respective class extend the Puzzle class, which also extends the MonoBehaviour class, allowing the puzzles to use methods such as Update, Start, etc. It contains an abstract method, named InitPuzzle which is called when the puzzle is enabled. This method is used to spawn the necessary objects for the puzzle. Another method which is used by all the puzzles is the PuzzleCompleted method, which is called by each puzzle when the completion condition is achieved, and is responsible for opening the gates and changing the color of the puzzle's indicator. These two methods can be seen in Figure 26. Besides those, the Puzzle class offers the IsPlacedCorrectly method in Figure 25 and a method used for placing hints on the walls.

```
7 references
public abstract void InitPuzzle();

6 references
public void PuzzleCompleted()
{
    completionIndicator.material.color = Color.green;
    audioSource.volume = PlayerPrefs.GetFloat("sfx");
    audioSource.Play();
    foreach (Gate gate in gates)
        gate.OpenGate();
}
```

Figure 26: InitPuzzle and PuzzleCompleted methods

The gates use the Gate class which has a single method, called OpenGate, which is called by the parent puzzle when it is completed. The method moves the gate downwards, into the ground, until it is not seen anymore, when the gate is destroyed (deleted from the scene).

Every puzzle has an information button which uses the InfoButton class, extending the IInteractable interface. When interacted with, it pauses the game and displays a textbox on the screen with a brief description of the puzzle. An animation is also played, simulating the

button being pressed, while a button clicking sound is also being heard.

All the puzzles, when spawning an item, use the list held by the GameManager to check whether that spot was already used to spawn another item. Then, after the item is successfully spawned, its chosen position is added to the list.

5.5.1 Multiple buttons pressed

This puzzle uses the ButtonsPressedPuzzle script which extends the Puzzle class to override the InitPuzzle method. It starts by spawning the buttons in randomly chosen points from the cave room. Again, raycasts sent downwards are used to place the buttons on the ground. Once a button is placed, the PlatePositionChecker script, present on the button prefab, is used to check whether the button overlaps with the terrain. As shown in Figure 27, this is done by using the Physics.OverlapBox method [48], which is similar to its sphere counterpart, but uses a box to check for overlapping.

```
2 references
public bool CheckPosition()
{
    Collider[] cols = Physics.OverlapBox(transform.position + transform.up * boxOffset, boxSize / 2, transform.rotation, groundLayer);
    return cols.Length == 0;
}
```

Figure 27: Code used to check if plate-like objects are overlapping with the terrain

After the buttons have been placed, the rocks are also being spawned inside the cave room, in a similar manner, but without the PlatePositionChecker script. Once they are placed as well, the puzzle is prepared to be solved.

The buttons, which use the PressurePlate class, constantly check is there is something on top of them, using the Physics.OverlapBox method inside the Update method. Once something is detected on top of them, it is checked if it is the player, using the Player layer, or a rock by checking if the object's name contains the word Rock. If any of these two conditions are met, the class sets a boolean to true. The ButtonsPressedPuzzle class monitors the booleans of the spawned buttons, and updates the puzzle's indicators accordingly. Once all the child buttons have their booleans set to true, the puzzle is completed and calls the PuzzleCompleted method.

The rocks use the Pickable class which extends the IInteractable interface to implements the Interact method. When the player interacts with Pickable object, the class stores a reference to the Interactor and locks it to prevent other interactions until the object is placed down, disables the rigidbody and the collider components of the object and places the object in front of the player constantly inside the Update method. To avoid the object obstructing the player's point of view, its material's transparency is changed, to turn the object see-through. When the player wants to place the object, the Pickable script uses a raycast when the player presses the E key, and places the object on the point hit by the raycast, with a small offset to avoid the object being placed inside other objects, also unlocking the Interactor.

5.5.2 Symbol matching

This puzzle uses the SymbolMatchingPuzzle script which similarly extends the Puzzle class. In the InitPuzzle method, it starts by shuffling a list containing the letters of the English alphabet, which is used to form the pairs by considering two consecutive letters in the shuffled list a pair. Then, the hints are placed by using the PlaceHint method from the Puzzle class. This method chooses a point randomly from the cave room, and sends raycasts towards north, west, south and east and stores the hits in an array. Then, the array is iterated through to search for a place where the hint can be placed correctly. To verify that, the hint prefab also uses the PlatePositionChecker script to check for overlapping. The first raycast hit that is on a wall, where the hint can be placed correctly is chosen. The hint prefab also uses a canvas set in world space to display text inside the world, on the hint object. This text is also changed by the PlaceHint method. A snippet of this code can be seen in Figure 28.

```
foreach (RaycastHit hit in hits)
{
    if (hit.collider == null) continue;

    Vector2 mapHintPoint = terrainGenerator.WorldToMap(hit.point);
    Vector2Int mapHintPointInt = new Vector2Int((int)mapHintPoint.x, (int)mapHintPoint.y);
    if (!region.points.Contains(mapHintPointInt)) continue;

    if (Helpers.LayerInLayerMask(hit.collider.gameObject.layer, groundLayer) && !usedPlaces.Contains(hit.point))
    {
        GameObject spawnedHint = Instantiate(hint, hit.point, Quaternion.identity, transform);
        spawnedHint.transform.LookAt(hit.point + hit.normal);
        spawnedHint.transform.Rotate(new Vector3(90, 0, 0));
        spawnedHint.transform.position += spawnedHint.transform.up * 0.1f;
        spawnedHint.transform.position += offset;
        if (spawnedHint.GetComponent<PlatePositionChecker>().CheckPosition() == false)
        {
            Destroy(spawnedHint);
            continue;
        }

        TMP_Text spawnedHintText = spawnedHint.GetComponentInChildren<TMP_Text>();
        spawnedHintText.text = hintText;

        placedHint = true;

        return hit.point;
    }
}
```

Figure 28: Snippet of code used to place hints

As the puzzle allows a maximum of 4 pairs, the puzzle's prefab has 8 buttons. In case the number of pairs picked is less than 4, the unnecessary buttons are disabled by using the SetActive method from the GameObject class. Then, the letters for each column are shuffled and assigned to the buttons. The puzzle's class also stores the current colors of the buttons, divided in columns, and also all the possible colors for the buttons (in this case 6).

The buttons use the SymbolMatchingButton class, which implements the IInteractable interface. In the Interact method, the button's color is changed to the the next unused color, which is determined by checking the currently used colors in the parent puzzle's class. After a color is picked, the parent's colors are updated and the CheckSymbols method from the parent's class is called. This method checks if the pairs have the same color, in which case the PuzzleCompleted method is called. The same animation simulating the button being pressed and the same button clicking sound are being played upon interaction.

5.5.3 Symbol ordering

This puzzle uses the SymbolOrderingPuzzle script, extending the Puzzle class. The InitPuzzle method starts by shuffling a list of letters from the English alphabet, from which the first couple are used for the puzzle. Then, hints are placed using the PlaceHint method from the puzzle class. After the hints are spawned, as the previous puzzle, this one's prefab has the maximum number of buttons (in this case 6), and has to disable the unnecessary buttons in case the number of letters is less than maximum. Afterwards, the used letters are again shuffled in another list, which will be used to assign the letters to the buttons. The puzzle's class also stores the current order of the letters and stores a reference to a button, which is used by the button's script.

The buttons use the SymbolOrderingButton class, implementing the IInteractable interface. In the Interact method, if no other button was pressed, the script will update the parent puzzle's stored button reference. If another button has been pressed, the two buttons get swapped by updating their text and the parent puzzle's order. This can be seen in Figure 29. At the end of the interaction, the parent puzzle's CheckSymbols method is called, which checks if the current order of letters is correct, in which case it calls the PuzzleCompleted method. This time, multiple animations are used. When the first button is pressed, it remains pressed to help the player remember what button was pressed. When the second button is pressed, it plays the animation of it being pressed, and the first button becomes unpressed as well through another animation. On each interaction, the button clicking sound is also being played.

```
2 references
public void Interact(Interactor interactor)
{
    if (parentPuzzle == null || parentPuzzle.isCompleted)
        return;

    audioSource.volume = PlayerPrefs.GetFloat("sfx");
    audioSource.Play();

    if (!parentPuzzle.isSelected)
    {
        animator.Play("ButtonStayPressed");
        parentPuzzle.selectedButton = this;
        parentPuzzle.isSelected = true;
    }
    else
    {
        if (parentPuzzle.selectedButton.index == index)
            return;

        animator.Play("ButtonPressedAnimation");
        parentPuzzle.selectedButton.animator.Play("ButtonUnpress");

        Helpers.SwapArrayElements(parentPuzzle.currentOrder, parentPuzzle.selectedButton.index, index);
        SetLetter(parentPuzzle.currentOrder[index], index);
        parentPuzzle.selectedButton.SetLetter(parentPuzzle.currentOrder[parentPuzzle.selectedButton.index], parentPuzzle.selectedButton.index);

        parentPuzzle.isSelected = false;
        parentPuzzle.CheckSymbols();
    }
}
```

Figure 29: Interact method from SymbolOrderingButton class

5.5.4 Assembling a key

The KeyAssemblingPuzzle script used by this puzzle also extends the Puzzle class. Its InitPuzzle method starts by choosing a random color for the key, using the Random.ColorHSV method. Then, the three key fragments are spawned inside the cave room using the same method with raycasts sent downwards. The puzzle's class also stores the currently picked keys as GameObject references and a boolean specifying if the key has been picked.

The key fragments use the KeyFragment class, implementing the IInteractable interface. They use rigidbodies and colliders to be able to fall to the ground after they are spawned, and to be interacted with. The Interact method updates the parent puzzle's list of currently picked keyfragments and changes the key fragment's material color transparency to 0, making the key fragment invisible and still on the ground. Its rigidbody and collider are also disabled.

The board onto which the fragments must be placed uses the KeyAssembler class, also extending the IInteractable interface. When the player interacts with it, the picked key fragments are placed on the board in predetermined spots, and also they become children of the board object by changing their Transform's parent to the board's Transform. If all the fragments are on the board, the key will be picked, updating the boolean in the parent puzzle's class. Lastly, after the key has been picked, the board object is destroyed and because the key fragments are now its children, they get destroyed as well. The Interact method can be seen in Figure 30.

```
2 references
public void Interact(Interactor interactor)
{
    if (parentPuzzle == null)
        return;

    if (currentFragments[0] == false || currentFragments[1] == false || currentFragments[2] == false)
    {
        foreach (GameObject keyFragment in parentPuzzle.pickedKeyFragments)
        {
            KeyFragment keyFragmentScript = keyFragment.GetComponent<KeyFragment>();
            currentFragments[keyFragmentScript.index] = true;

            keyFragment.transform.position = placements[keyFragmentScript.index].transform.position;
            keyFragment.transform.rotation = placements[keyFragmentScript.index].transform.rotation;
            keyFragment.transform.parent = gameObject.transform;
            keyFragmentScript.SetColor(parentPuzzle.keyColor);
        }

        parentPuzzle.pickedKeyFragments.Clear();
    }
    else
    {
        parentPuzzle.pickedKey = true;
        Destroy(gameObject);
    }
}
```

Figure 30: Interact method from KeyAssembler class

The key hole uses the KeyHole class, also extending the IInteractable interface. Its Interact method checks if the key was picked up, in which case the puzzle is completed by calling the parent puzzle's class' PuzzleCompleted method. Here, an animation is played simulating the key being introduced into the key hole and then rotated.

5.5.5 Crystal resonance

The CrystalsResonancePuzzle script used by this puzzle also extends the Puzzle class. The InitPuzzle method begins by picking a number of crystals and choosing the same number of pitches. The pitches are chosen as an offset, starting from 0, going up to 1 in increments. Then, the pitches are shuffled to obtain the correct sequence of pitches, stored separately, then are once again shuffled to obtain the pitches which will be assigned to the crystals. After choosing the pitches, the crystals are spawned in random places inside the cave room, using the same raycast downwards method. The puzzle's hint features a number of musical notes, which are textboxes inside a canvas that uses world space to render. As the number of notes is set to the maximum in the prefab, some of them are disabled if needed. All the remaining ones are placed at a different height to signify the correct pitch order, and their color is changed to red. The puzzle's class stores the number of correct hits and offers the CrystalHit method, seen in Figure 31, to be used by the crystals. This method checks whether the current hit comes from the correct crystal, in which case the note is colored green, and if was the last note, the puzzle is completed using the PuzzleCompleted method. If the current hit comes from the wrong crystal, the notes are all reset to red. Because the hit, while wrong currently, could be correct as the first hit, the method also has a recall parameter, which is used for wrong hits to avoid infinite recursion.

```
2 references
public void CrystalHit(float pitch, bool recall = true)
{
    if (Mathf.Abs(correctPitches[correctHitsCount] - pitch) < 0.0001f)
    {
        keynotes[correctHitsCount].color = Color.green;
        correctHitsCount++;

        if (correctHitsCount == crystalCount)
        {
            isCompleted = true;
            PuzzleCompleted();
        }
    }
    else
    {
        correctHitsCount = 0;
        foreach (TMP_Text keynote in keynotes)
            keynote.color = Color.red;

        if (recall)
            CrystalHit(pitch, false);
    }
}
```

Figure 31: CrystalHit method from CrystalsResonancePuzzle class

The crystals use the Crystal class, implementing the IInteractable interface. The Interact method is used to call the CrystalHit method from the parent puzzle's class, with the crystal's pitch. It also plays a sound using the PlayOneShot method from the AudioSource class, but only after configuring the source's pitch accordingly to the crystal's pitch and the volume to the player's volume setting.

5.5.6 Gem showcase

This puzzle uses the GemShowcasePuzzle script, which extends the Puzzle class. The InitPuzzle starts with choosing the number of gems. Then, for each one, a random gem is chosen, then its pillar is spawned in a random place inside the cave room. Each pillar has a hint placed inside the prefab, which is an image inside a world space canvas. The image's texture is changed to the gem's corresponding image. Then, the actual gem has to be spawned. As the puzzle can spawn the gems anywhere in the accessible caves, a list of cave rooms is stored in the Puzzle class, and it represents the caves that had been processed by the GameManager at the moment this cave was being processed. As the GameManager used a BFS algorithm, that list can be used as a list of currently accessible rooms. The gems are spawned in a random number between 1 and 3 in a randomly chosen place inside a randomly chosen cave from the list of accessible caves.

The pillars use the ItemShowcase script, which extends the PressurePlate class. This is because the pillars need to check if they have the correct item placed on top of them, but also want to spin the item afterwards. By extending the PressurePlate class, the checking part can be preserved by calling base.Update, then adding the spinning. Once the correct gem is placed on the pillar, in holds the boolean that specifies if it is pressed on true, and disables the rigidbody and collider in order to spin the item. The actual spinning is another animation that is played on loop.

The gems use the Pickable class, which has been presented previously.

5.6 User interface (UI)

The in-game UI is comprised of the minimap, the levels of hunger and light, the interaction prompt and a crosshair. The minimap is actually an image whose texture is changed during the game, the levels are sliders, the interaction prompt is a textbox and the crosshair is an image of a circle. All of these are placed inside a Canvas object rendered on screen.

For the minimap, the Minimap class is responsible of changing the image's texture. This is done using a compute shader, seen in Figure 32, which receives the texture it has to modify, which uses the UnityEngine.Experimental.Rendering.GraphicsFormat.R32G32B32A32_SFloat format [38], and various data such as the actual map, the map seen by the player, the map size and the player's position. It uses those to render the portion of the map seen by the player in white, and the rest in black, with the player's position in red. Each thread is responsible for one of the pixels of the texture and the shader is dispatched inside the Update method of the Minimap class. The script also calculates and stores the portion of the map seen by the player. A point of the map has been seen by the player if the distance between it and the player is less than a certain value, in this case 5.

Both of the sliders for hunger and light levels are handled by the scripts that use that level. In

```
#pragma kernel CSMain
#define INSIDE float4(1.0, 1.0, 1.0, 1.0)
#define OUTSIDE float4(0.0, 0.0, 0.0, 1.0)
#define PLAYER float4(1.0, 0.0, 0.0, 1.0)

RWTexture2D<float4> minimapTexture;

RWStructuredBuffer<int> map;
RWStructuredBuffer<int> seen;
int2 mapSize;
float2 player;

[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint2 id2d = uint2(id.x, id.y);
    if (distance(player, float2(id.x, id.y)) < 1)
        minimapTexture[id2d] = PLAYER;
    else
        minimapTexture[id2d] = (map[id.x * mapSize.y + id.y] == 1 && seen[id.x * mapSize.y + id.y] == 1) ? INSIDE : OUTSIDE;
}
```

Figure 32: Compute shader used for minimap

the case of the hunger level, it is handled by the PlayerController script, which uses the hunger level to determine the movement speed of the player. In holds a variable which determines the current hunger level, which is also used to constantly update the slider's value, and is constantly decreased by a fixed value inside the FixedUpdate method. In the case of the light level, it is handled by the LightController script, which uses the light level to change the strength of the player's flashlight. This will be described in more details in a later section. The light slider is handled similarly to the hunger one, its value is changed constantly by the variable representing the light level, held by the LightController.

The interaction prompt is a textbox whose actual text is changed by the Interactor class when it detects the player looking at an interactable object. When the player is not looking at an interactable one, the prompt is set to an empty string, so it doesn't show anything. In case the player is holding an object with the Pickable class, it will change the prompt when the Interactor is locked.

5.6.1 Pause and end menu

Pausing inside the game is possible with the help of the PauseManager script. It detects when the player presses the Escape key and pauses the game. This is done by having a variable that specifies if the game is paused or not, which is changed when the Escape key is pressed. The scripts affected by this are the PlayerController, CameraController, Interactor and Pickable. All of these check if the game is paused, by accessing the variable of the PauseManager through a reference from the GameManager, and ignore all inputs received while the game is paused.

The pause menu contains multiple buttons, such as Resume, Restart, Back to menu and sliders for the sound effects' volume and mouse sensitivity. The buttons have on-click callbacks attached to them, the Resume button disables the pause, the Restart button reloads the scene using the SceneManager class provided by Unity and the Back to menu button uses

the same SceneManager to load the menu scene. The sliders also have callbacks, which are called when their values are changed. These values are also stored so that after the game is closed and reopened, the previous values are restored. The method used to store the values will be detailed in a later section. The sliders, when their values are changed, update the values stored, while the scripts that play sounds use the stored volume value, and the CameraController uses the stored mouse sensitivity.

The game is completed when all puzzles are completed. When this happens, the GameManager will put the game on permanent pause, meaning the player cannot unpause, and show a textbox informing the player that they completed all the puzzles. The player will be able to press on a single button, taking them back to the main manu.

5.7 Lighting

The lighting inside the game is all turned off, with the exception of the player's flashlight and the crystals, which also emit light. The player's light is handled by the LightController class, which uses the light level to change the player light's strength. This light is actually a point light object, which actually emits light in all directions. Even though Unity offers a spot light, which would be closer to the effect of a flashlight, after testing both, I liked the point light better. The LightController changes the range of the point light based on the light level to increase the strength of the light.

5.8 Resource management

All the resources in the game use the Collectable class, which implements the IInteractable interface. The Interact method calls the CollectedItem method from the GameManager and specifies what resource it is. That method, based on the type of resource received, calls the Consume method from the PlayerController class, which increases the hunger level when a water bottle or an apple was consumed or the UseBattery method from the LightController, which increases the light level when a battery is used.

5.9 Main menu

The main menu is placed in a different scene. In this scene, there is also a terrain generated using the TerrainGenerator script, which in this scene, instead of enabling the GameManager, it placesthe camera in a random spot inside the caves, which is then rotated around through the RotateCamera class. The TerrainGenerator takes its seed from the SeedManager class, which has the only purpose of storing a seed. Initially it also generates a random seed, to be used by the TerrainGenerator, but will be able to be changed later. It also uses the

DontDestroyOnLoad method from the Object class to allow the object with the SeedManager script to be passed to the game scene, with the stored seed. The SeedManager also follows the Singleton design pattern, to ensure that a single SeedManager script is active at any time. This is important because when the player goes back to the main menu, from the game scene, the SeedManager which is not destroyed when the scene is changed, would remain active, along with the SeedManager from the menu scene. Through the Singleton design pattern, one of them will get destroyed.

The main menu features a title, multiple buttons, an input field and an image of the map inside a Canvas object which is rendered on screen. The buttons are used to start the game, open an options tab, and exit the game. The input field is used to allow the player to type their desired seed, while the image reflects the map of the introduced seed, or a random map if no seed was introduced. Everything related to the main menu is handled through the MenuManager class. It attaches on-click callbacks to the buttons and handles generating the map of the given seed.

The start button changes the scene using the SceneManager class, the options button disables the current buttons and displays two sliders, for the sound effects' volume and mouse sensitivity. These work similar to the ones from the pause menu. There also appears a button to return from the option tab. Lastly, the exit button closes the game using the Application.Quit method.

The input field has a callback which is triggered when the text held by it is changed. When this happens, the SeedManager's seed is updated and the MenuManager creates a new MapGenerator class with the new seed to generate the 2D map. Then, a compute shader, seen in Figure 33 is dispatched which uses the map to generate a texture for the main menu image. The texture also uses the UnityEngine.Experimental.Rendering.GraphicsFormat.R32G32B32A32_SFloat format [38]. Each thread is responsible for one pixel and the color is generated based on the values in the map. Since the cave has the value of 1, it will be white, and the walls, having the value of -1, which is turned to 0, will be transparent.

```
#pragma kernel CSMain

RWTexture2D<float4> tex;
RWStructuredBuffer<int> map;
int2 mapSize;

[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    if (id.x >= mapSize.x || id.y >= mapSize.y)
        return;

    uint2 id2d = uint2(id.x, id.y);

    float val = map[id.x * mapSize.y + id.y];
    if (val < 0.0)
        val = 0.0;

    tex[id2d] = float4(val, val, val, val);
}
```

Figure 33: Compute shader used to display the map inside the menu

5.10 Others

There are multiple animations used throughout the game, but all of them are simple states inside their respective animator. Every animator begins in an empty state and has its state changed through code. Every other state can have no transitions or a transition to the empty state, which happens after that animation clip is completed. An example of one of the animators can be seen in Figure 34.

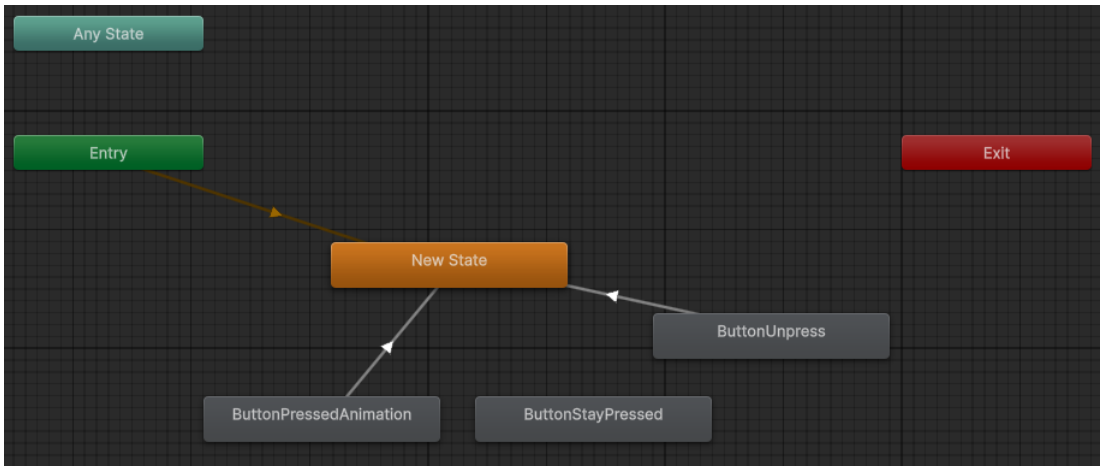

Figure 34: Button animator

The player's settings are stored using the PlayerPrefs class [49] provided by Unity. It allows the game to store data between game sessions. This means that the player's settings are remembered when the game is reopened. The settings include the sound effects' volume and the mouse sensitivity. When one of the sliders handling these settings is modified, it also updates the value stored in PlayerPrefs. If there hasn't been any value stored, a default value is assigned. When one of the scripts needs to use the volume or sensitivity, they get that value directly from the PlayerPrefs class.

There are also multiple sounds played throughout the game. Since all of them are sound effects, there was no need to use audio mixers, so every sound is played by its respective script when needed, through the PlayOneShot method from the AudioSource class.

6 RESULTS

As Cavernous Expedition is a video game, the users' feedback is extremely important for the development of the game. At the same time, because the main feature of the game is the terrain generation, it is just as important for the implemented algorithms to be efficient so that the game runs great, without performance issues. These two topics are presented in the following sections.

6.1 Algorithm performance

To determine the performance of the terrain generation, the two algorithms responsible for it have been timed when using different map sizes. Even though the game has a fixed map size, it is important to measure the efficiency for higher dimensions in regards to future developement. In Figure 35 can be seen the time plots for the Cellular Automaton and Marching Cubes while using different map sizes, starting from 40x40 going up to 240x240 in increment of 40.

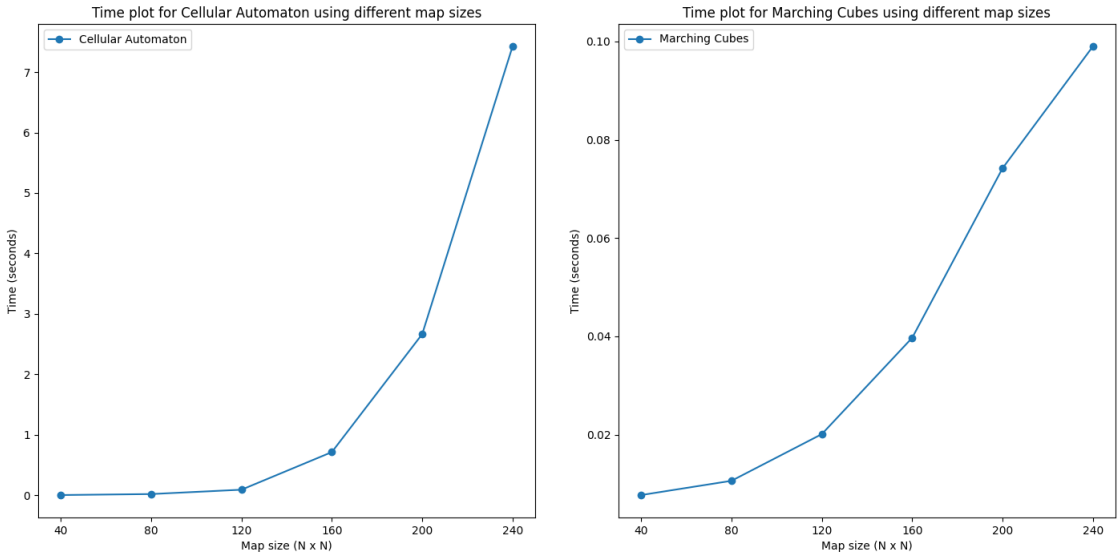

Figure 35: Time plots for Cellular Automaton and Marching Cubes using different map sizes

A comparison between the two can be seen in Table 1. It shows that while both algorithms perform well at small map sizes, once the map size reaches 160x160, the Cellular Automaton takes almost one second to complete, which is considerably longer than the Marching Cubes, and will most likely make the player notice this waiting time. Once a the map size reaches

200x200, the Cellular Automaton takes even a couple seconds to complete. On the other hand, the Marching Cubes, even at high map sizes, performs really well, taking just under a tenth of a second for a 240x240 map. This is because the Cellular Automaton is run on the CPU, while the Marching Cubes is run on the GPU, being able to use the massive parallel power of it. While Cavernous Expedition uses 80x80 maps, this means that if the game is to include customizable map sizes in the future, an optimization will be needed for the Cellular Automaton. This can include smarter rules which would need fewer steps, or a parallel implementation which can be run on the GPU.

Map size	**Cellular Automaton (seconds)**	**Marching Cubes (seconds)**
40x40	0.003	0.0077
80x80	0.0196	0.0106
120x120	0.0915	0.0201
160x160	0.7136	0.0397
200x200	2.6652	0.0742
240x240	7.425	0.099

Table 1: Time comparison between Cellular Automaton and Marching Cubes

It is also important for the game to have a good framerate throughout the game. While the 80x80 map size used by Cavernous Expedition gives around 400 frames per second, it is important to test how well the game performs for bigger maps. As shown in Figure 36, the higher the map size is, the lower the FPS value is. Even though for the computer I tested this on even a 240x240 map size would have more than enough frames per second, for other computers it might pose a problem. Because of this, for future development a chunk based system could be implemented, which would reduce the number of objects active in the scene, increasing the framerate.

6.2 Feedback assessment

To determine what other people think of Cavernous Expedition, I asked a couple of volunteers to play the game and then leave their impressions through a form. In order to receive genuine feedback, they played the game by themselves, with no involvement from me, other than giving them a short explanation of the game. The majority of them played for about 10-15 minutes, which was expected considering the chosen size for the map, and the average number of puzzles present in a run.

In order to assess if the terrain generation was producing varied and realistic results, I asked them to rate the variedness and the realism on a scale of 1 to 5. As shown in Figure 37, the users consider the terrain to be quite varied from run to run, but there is still some room for improvement. On the other hand, in Figure 38, the responses are more varied, and show that the users considered the cave system to be somewhat realistic, but it isn't in its best shape.

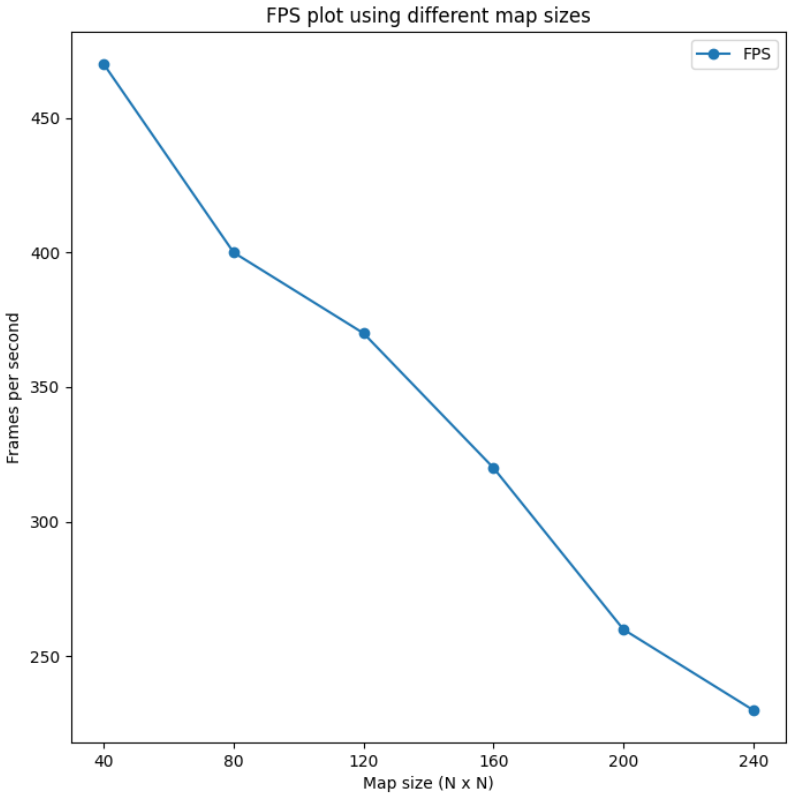

Figure 36: FPS plot using different map sizes

Taking this feedback into consideration, some future improvements could include cave systems divided into multiple levels, giving the world some elevation. For this, the caves generated by the Cellular Automaton could be placed at different heights, which could allow the player to climb up or down through the tunnels. From an aesthetic perspective, more realism could be achieved from placing stalactites and stalagmites inside the caves, while the caves themselves could have different themes, similar to biomes. This would also open up possibilities for biome specific puzzles.

Regarding the gameplay, the volunteers were asked how intuitive were the puzzles. As shown in Figure 39, while some found the puzzles to be very intuitive, some struggled a bit to understand what some puzzles required. This means that the information system, through which a puzzle description would be shown on screen when the information button on the puzzle was pressed, is not enough for all the puzzles. Another possibility is that the hints used for some of the puzzles could be confusing to some. In the future, there could be a special tab in the menu in which all the puzzles would be described in more detail, taking into consideration explaining what the hints mean and how they help the player to solve the puzzle, though this has to be done carefully to still leave the thinking part to the player, as the puzzles must test the player's thinking skills.

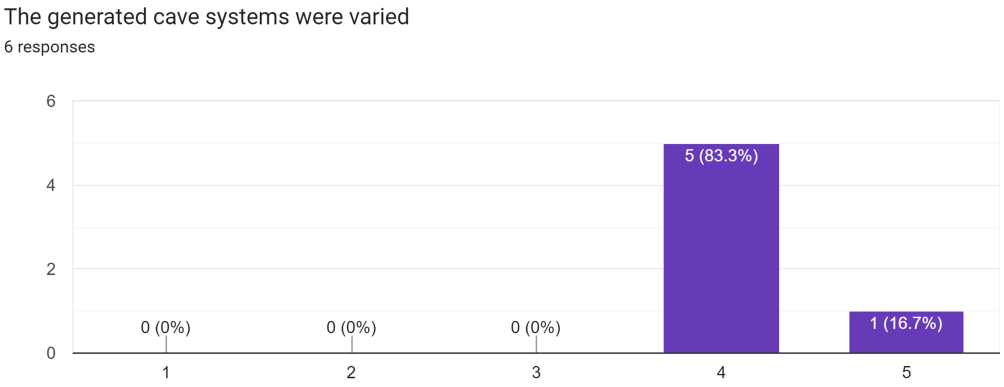

Figure 37: Terrain variedness answers

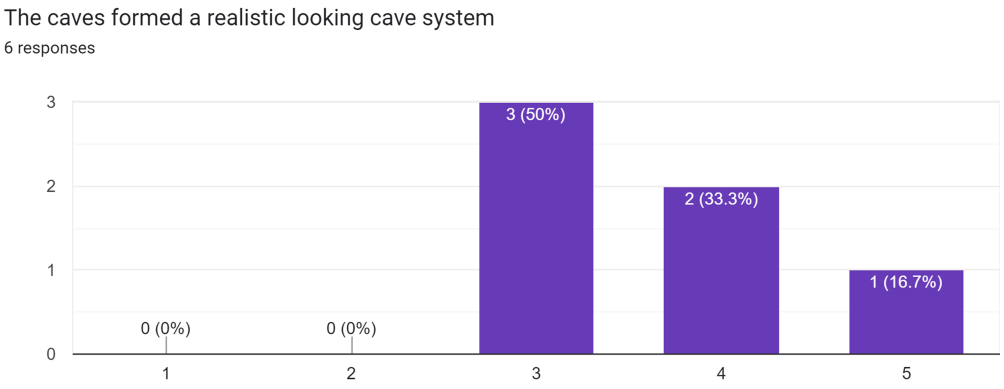

Figure 38: Terrain realism answers

When asked about the quantity of resources, everyone answered that there were enough. While this could mean that the current number of resources, one of each for every cave room, is just enough, it could also mean that there are too many. Looking from another perspective, the decrease rate for the hunger and light levels could be too slow, diminishing the need to search for resources.

As the controls are also important, in Figure 40 can be seen that the users were mostly satisfied with the current controls, which they found intuitive. In the future, key binding could be added so that everyone can have the desired controls.

Regarding the menus, the majority of the volunteers found them easy to navigate and had no problem going through them. For them, future improvements could include a more professional look, but their functionality is good.

The participants were also asked to rate the game by giving it an overall score. As shown in Figure 41, they thought the game was good, but not perfect. This shows that the users

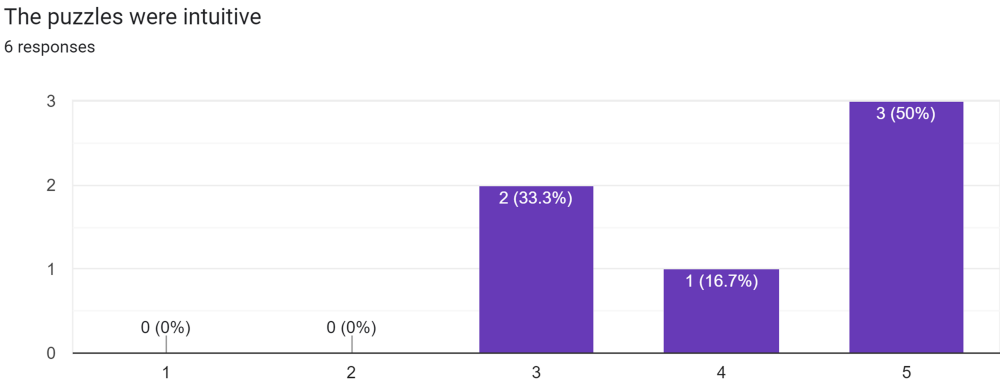

Figure 39: Puzzle intuitiveness answers

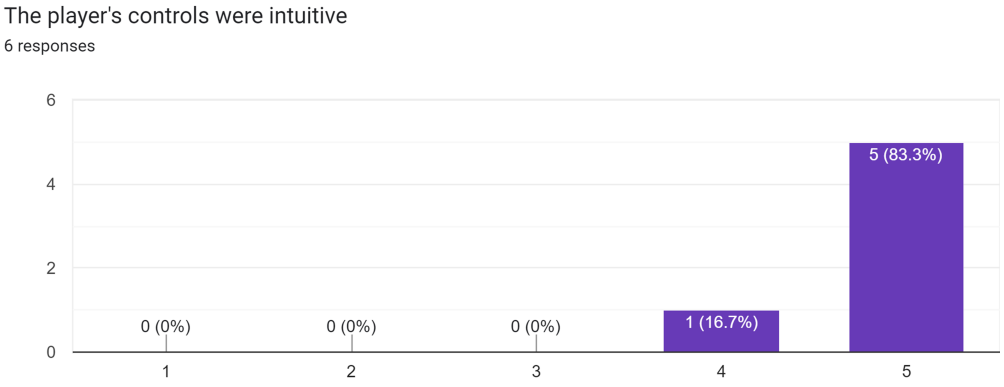

Figure 40: Player controls answers

believe the game still needs improvements. Beside the already mentioned possible future changes, the game would benefit from a more customizable map, meaning more options other than the seed, and from a greater selection of puzzles. The current types of puzzles are not nearly enough for the game to feel replayable, as all 6 othem could appear on a single map. The diversity of puzzles can also be improved, with more complex puzzles or puzzles that use the actual terrain for their solution, strengthening the connection between the caves and the puzzles.

Lastly, the volunteers were asked what features they would like to see in the future. One suggestion that stands out is the addition of co-op play, which would give the game the opportunity to feature more interesting puzzles. This also affirms the market research conducted in an earlier chapter, where it was shown that puzzle games often include co-op play to be able to construct immersive and complex puzzles that require communication to solve.

Other suggestions include the addition of player-versus-player, which could take the form of a timed competition, such as two or more players trying to solve the same map in as little time

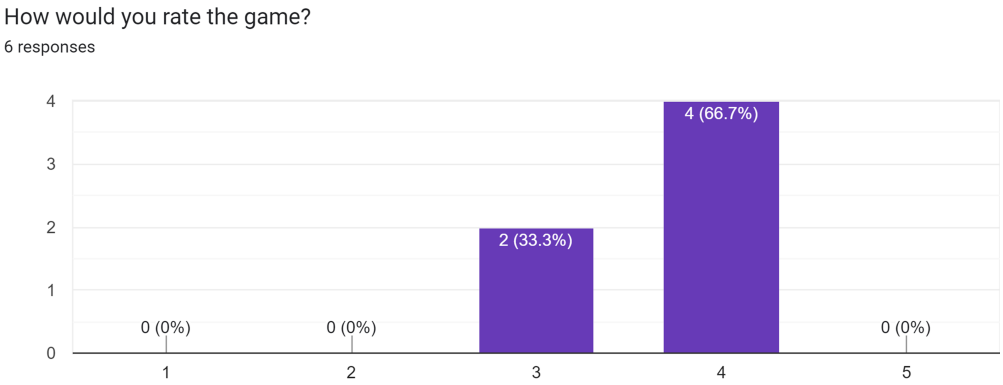

Figure 41: Overall score

as possible. Another interesting suggestion is the addition of combat, which could be against various monsters that would dwell inside the caves, turning the game's focus from the puzzles towards a survival perspective.

The rest of the volunteers also suggested improvements such as more decorations, which would also help with navigating the cave system without relying so much on the minimap, more puzzle diversity and, overall, more things to do.

7 CONCLUSIONS

After analyzing the market of adventure-puzzle and procedurally generated games, Cavernous Expedition managed fulfill the game design requirements, using procedural generation to create a realistic and varied cave system which the player explores while solving various puzzles, with each run being a different experience.

As shown in the previous chapter, the game is far from perfect and can benefit from multiple improvements. On the technical side, some of the algorithms used could be implemented more efficiently, maybe even profiting from the GPU's massive parallel performance. On the other hand, the design side is also in need of future changes. While the important ones include a greater variety of puzzles and a better customization for the terrain generation, the others must also be taken into consideration. A greater level of details would also be a huge plus for the world generation, giving it more realism, while a different generation technique, involving elevated caves, would improve the player's immersion. Lastly, adding the feature of multiplayer, be it in the form of co-op or PvP, would greatly increase the replayability rate.

BIBLIOGRAPHY

[1] Charlie Hall. Nine new genres that defined the decade in games. https://www.polygon.com/features/2019/11/8/20943571/new-game-genres-decade-in-review-2020-battle-royale-fortnite-dark-souls. Last accessed: June 24, 2024.

[2] Machinations. How to design a puzzle game. https://machinations.io/articles/how-to-design-a-puzzle-game. Last accessed: June 24, 2024.

[3] Joe Keeley. What are adventure games and how have they evolved? https://www.makeuseof.com/what-are-adventure-games/. Last accessed: June 24, 2024.

[4] Aoi Ikeda. Unraveling the mysteries of procedural generation in gaming. https://tokengamer.io/unraveling-the-mysteries-of-procedural-generation-in-gaming/. Last accessed: June 24, 2024.

[5] Rocket Brush Studio. Most popular video game genres in 2024: Revenue, statistics. https://rocketbrush.com/blog/most-popular-video-game-genres-in-2024-revenue-statistics-genres-overview. Last accessed: June 24, 2024.

[6] HistoryofInformation.com. Adventure or colossal cave adventure is the first computer text adventure game. https://www.historyofinformation.com/detail.php?id=2020. Last accessed: June 24, 2024.

[7] Valve. Portal 2. https://store.steampowered.com/app/620/Portal_2/. Last accessed: June 24, 2024.

[8] Total Mayhem Games. We were here. https://totalmayhemgames.com/games/we-were-here/. Last accessed: June 24, 2024.

[9] Michael Toy and Glenn Wichman. Rogue. https://www.pcjs.org/software/pcx86/game/other/1985/rogue/. Last accessed: June 24, 2024.

[10] Hello Games. No man's sky. https://www.nomanssky.com/. Last accessed: June 24, 2024.

[11] Hilton Webster. How many planets are actually in no man's sky? https://www.thegamer.com/no-mans-sky-how-many-planets/. Last accessed: June 24, 2024.

[12] Mojang. Minecraft. https://www.minecraft.net/en-us. Last accessed: June 24, 2024.

[13] Mossmouth. Spelunky. https://spelunkyworld.com/original.html. Last accessed: June 24, 2024.

[14] Ghost Ship Games. Deep rock galactic. https://www.deeprockgalactic.com/. Last accessed: June 24, 2024.

[15] Klei Entertainment. Don't starve. https://www.klei.com/games/dont-starve. Last accessed: June 24, 2024.

[16] GameFromScratch.com. Game engine popularity in 2024. https://gamefromscratch.com/game-engine-popularity-in-2024/. Last access: June 24, 2024.

[17] Unity Technologies. Unity game engine. https://unity.com/. Last accessed: June 24, 2024.

[18] Unity Game Engine. Unity asset store. https://assetstore.unity.com/. Last accessed: June 24, 2024.

[19] Studio MDHR. Cuphead. https://www.cupheadgame.com/. Last accessed: June 24, 2024.

[20] Beat Games. Beat saber. https://www.beatsaber.com/. Last accessed: June 24, 2024.

[21] Juan Linietsky and Ariel Manzur. Godot. https://godotengine.org/. Last accessed: June 24, 2024.

[22] Seaven Studio Blobfish. Brotato. https://godotengine.org/showcase/brotato/. Last accessed: June 24, 2024.

[23] Deep Root Interactive. Endoparasitic. https://godotengine.org/showcase/endoparasitic/. Last accessed: June 24, 2024.

[24] Epic Games. Unreal engine. https://www.unrealengine.com/en-US. Last accessed: June 24, 2024.

[25] Gearbox Studio. Borderlands 3. https://borderlands.2k.com/. Last accessed: June 24, 2024.

[26] Microsoft Studios Rare. Sea of thieves. https://www.seaofthieves.com/. Last accessed: June 24, 2024.

[27] Eric W. Weisstein. Cellular automaton. https://mathworld.wolfram.com/CellularAutomaton.html. Last accessed: June 24, 2024.

[28] William Lorensen and Harvey Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21:163–, 08 1987.

[29] Liam Cubicle. What is perlin noise? https://www.occasoftware.com/blog/perlin-noise. Last accessed: June 24, 2024.

[30] Martin Palko. Triplanar mapping. https://www.martinpalko.com/triplanar-mapping/. Last accessed: June 24, 2024.

[31] SnowFiend Studios. Lowpoly rocks. https://assetstore.unity.com/packages/3d/environments/lowpoly-rocks-137970. Last accessed: June 24, 2024.

[32] SineVFX. Translucent crystals. https://assetstore.unity.com/packages/3d/environments/fantasy/translucent-crystals-106274. Last accessed: June 24, 2024.

[33] 3dfancy. Low poly pillar set. https://assetstore.unity.com/packages/3d/environments/dungeons/low-poly-pillar-set-20825. Last accessed: June 24, 2024.

[34] AurynSky. Simple gems ultimate animated customizable pack. https://assetstore.unity.com/packages/3d/props/simple-gems-ultimate-animated-customizable-pack-73764. Last accessed: June 24, 2024.

[35] Rowntec Assets. Survival kit lite. https://assetstore.unity.com/packages/3d/props/tools/survival-kit-lite-92549. Last accessed: June 24, 2024.

[36] Pixabay. Pixabay. https://pixabay.com/. Last accessed: June 24, 2024.

[37] Statistics How To. Random seed: Definition. https://www.statisticshowto.com/random-seed-definition/#excel. Last accessed: June 24, 2024.

[38] Unity Documentation. Unity - scripting api: Graphicsformat. https://docs.unity3d.com/ScriptReference/Experimental.Rendering.GraphicsFormat.html. Last accessed: June 24, 2024.

[39] Unity Documentation. Unity - manual: Compute shaders. https://docs.unity3d.com/Manual/class-ComputeShader.html. Last accessed: June 24, 2024.

[40] Unity Documentation. Unity - scripting api: Mathf.perlinnoise. https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html. Last accessed: June 24, 2024.

[41] Matthew Fisher. Marching cubes. https://graphics.stanford.edu/~mdfisher/MarchingCubes.html. Last accessed: June 24, 2024.

[42] Paul Bourke. Polygonising a scalar field. https://paulbourke.net/geometry/polygonise/. Last accessed: June 24, 2024.

[43] Unity Documentation. Unity - scripting api: Computebuffer. https://docs.unity3d.com/ScriptReference/ComputeBuffer.html. Last accessed: June 24, 2024.

[44] Unity Documentation. Unity - manual: Writing surface shaders. https://docs.unity3d.com/Manual/SL-SurfaceShaders.html. Last accessed: June 24, 2024.

[45] Unity Documentation. Unity - scripting api: Physics.raycast. https://docs.unity3d.com/ScriptReference/Physics.Raycast.html. Last accessed: June 24, 2024.

[46] GeeksForGeeks. Breadth first search or bfs for a graph. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. Last accessed: June 24, 2024.

[47] Unity Documentation. Unity - scripting api: Physics.overlapsphere. https://docs.unity3d.com/ScriptReference/Physics.OverlapSphere.html. Last accessed: June 24, 2024.

[48] Unity Documentation. Unity - scripting api: Physics.overlapbox. https://docs.unity3d.com/ScriptReference/Physics.OverlapBox.html. Last accessed: June 24, 2024.

[49] Unity Documentation. Unity - scripting api: Playerprefs. https://docs.unity3d.com/ScriptReference/PlayerPrefs.html. Last accessed: June 24, 2024.