

Bachelor's Thesis Report

Development of a tool for the parallel procedural generation of worlds inside a game engine and its application to a video game

Álvaro Chuan Díaz-Maroto

Bachelor's Thesis

Bachelor's Degree in Video Game Design and Development

Universitat Jaume I

June 27, 2025



Project supervised by: Miguel Chover Sellés

To my friends and family.

Acknowledgments

Videogames have been an essential part of my life since my parents gifted me, my first videogame console, a Nintendo DS. It wasn't a surprise for my parents the day I told them I wanted to be a game developer, in fact, they helped me find this degree and prepare for it. Without them and my family, I could never have attended this university and follow my dreams, so first I would like to thank my parents for giving me this opportunity and supporting me through my whole journey here.

Apart from my family, I would like to thank all my friends, specially Lucía and Mario, sharing the same house for 3 years was definitely an experience I won't forget in my life. For you Mario, I can't stress enough how thankful I am for working with me in almost every single university project, we have cried of laughter and lost our minds over bugs way too many times. I would also like to thank Paco, even though each of us ended up in opposite sides of the Valencian Community, you've been there for me at any moment given, no matter the distance or the time we expend without talking, our friendship never changes. In addition, I could not forget to thank my good old friend Ruben, you are one of the reasons I entered this career, and you always supported my work. We made a promise a long time ago, I would code the games, and you would make the music for them. You are no longer here with us to accomplish that promise, but I still keep it in mind, I will never forget you.

I would also like to thank all the good friends that I've made along the way, Edward, Raúl, Victor, Christian, Diego, Joan, Jorge, Sergio, Alan, Mar and Alex, you are the best group of game developers and classmates that I could have ever asked for. I know that some day, BackToBits will get to something.

Last but not less important, I would like to thank you, Eva. You've been there on my highest and my lowest, you've always supported me even when I was trying to do things that were clearly out of my possibilities, your support has been essential for me to be here today.

Finally, I would also like to thank my Bachelor's Thesis Supervisor Miguel Chover for his support here and in my internship at INIT. Maria Villar for her

support and investigation on the same matter and all my internship coworkers for the great time I've spent there, I would love to work with you again in the future. Also thank Sergio Barrachina Mir for his «[L^AT_EX Template for writing the Bachelor's thesis](#)», that has been used to write this memory.

Abstract

This Bachelor's Thesis presents the development of a tool for the procedural generation of 3D environments using a GPU-parallelized version of the Wave Function Collapse algorithm, integrated within the Unity engine. The tool enables users to define custom tile sets and adjacency rules to generate worlds, offering an intuitive interface to manage tile and world creation. To validate the tool, a complete 3D top-down video game titled System Escape was developed. This game demonstrates the tool's capabilities through procedurally generated planets within a dying solar system, where the player must escape before a supernova occurs. The game integrates resource collection, combat, upgrades, and survival mechanics. The tool significantly reduces development time while ensuring creative freedom and performance scalability. Performance tests show substantial improvements over CPU-based implementations, and the modular nature of the tool makes it suitable for a wide variety of game genres. This project highlights the potential of combining GPU computing with procedural generation techniques, and aims to contribute a robust and accessible asset to the Unity development community.

Contents

Contents	v
1 Introduction	1
1.1 Project Motivation	2
1.2 Related Subjects	2
1.3 Project objectives	3
1.4 Task Planning and Scheduling	3
1.5 Expected results	5
1.6 Tools to be used	5
2 Design	7
2.1 Wave Function Collapse 2D algorithm	7
2.2 Wave Function Collapse 2D parallelization	8
2.3 Wave Function Collapse 3D algorithm	10
2.4 Wave Function Collapse 3D parallelization	13
2.5 Unity tool	13
2.6 Videogame	18
3 Tool development	37
3.1 WFC compute shader adaptation	37
3.2 Shader management in the CPU side	40
3.3 Chunks generation mode	43
3.4 User interface	45
3.5 Tool results	49
4 Game development	51
4.1 Player movement	51
4.2 Resource gathering	53
4.3 Combat	54
4.4 Upgrade system	55
4.5 Spaceship movement	56
4.6 Game loop	57
4.7 Interface	57
4.8 Sound system	58

4.9	2D art	59
4.10	3D art	60
4.11	Results	62
5	Conclusions and future work	65
5.1	Conclusions	65
5.2	Future work	66
	Bibliography	67
6	Project repositories	69

CHAPTER

1

Introduction

Índice

1.1	Project Motivation	2
1.2	Related Subjects	2
1.3	Project objectives	3
1.4	Task Planning and Scheduling	3
1.5	Expected results	5
1.6	Tools to be used	5

This project aims to develop a tool for the Unity 3D game engine that allows users to procedurally generate finite environments using a parallelized version of the Wave Function Collapse algorithm, in a simple and user-friendly manner through a customized and intuitive interface.

As an application of this tool, a 3D top-down video game will be developed, featuring a low-poly, cartoon-style aesthetic and belonging to the adventure and space exploration genres. The game will take place in a solar system whose central star is near the end of its life cycle. The player's objective will be to escape the solar system before the supernova occurs. To do so, the player will embark on a spaceship and travel across various planets within the system, facing enemies and collecting materials to upgrade both their personal equipment and their spaceship. These planets will be generated using the aforementioned tool, thereby serving as a demonstration of its applied use within a video game context.

1.1 Project Motivation

Procedural content generation is a field that is increasingly being explored within the realm of video games. Examples of this trend include titles such as *No Man's Sky*, *Minecraft*, *Astroneer*, *Terraria*, *Dwarf Fortress*, and the majority of existing roguelikes. However, all of these games have had to develop their own generation systems from scratch. This project seeks to eliminate that burden for developers by providing them with an effective and intuitive tool that enables the creation of game worlds without the need to implement such systems manually.

Regarding the game included in this project, the primary motivation is to demonstrate how the application of this tool can yield solid results and accelerate the development process, while maintaining a high standard of quality in the final product. At the same time, it aims to create a game that is engaging and appealing to a broad audience.

1.2 Related Subjects

Subject	Explanation
VJ1208 - Programming II	This course is essential for learning to use the C# programming language, which is used in Unity.
VJ1215 - Algorithms and Data Structures	Although different programming languages and algorithms are used compared to those covered in the course, algorithmic thinking and a solid understanding of data structures are crucial for developing a tool optimized for use by any user.
VJ1216 - 3D Design	The content of this course encompasses everything related to the creation of 3D assets for implementation in video games, making it fundamental for the development of the demonstration game included with the tool.
VJ1221 - Computer Graphics	To optimize the Wave Function Collapse algorithm, it is necessary to run it in parallel across multiple threads. While this could be done on the CPU, it will be executed on the GPU for better performance. Therefore, this course is essential for understanding the types of data handled by the GPU and how to work with them using shaders.

Triple Project	This integrates the courses VJ1222 - Conceptual Design of Video Games, VJ1223 - Game Art, and VJ1224 - Software Engineering. This project represents the real-world project we undertake as students during the degree and establishes the foundation for good organization and professional working practices.
VJ1227 - Game Engines	Since both the tool and the game will be developed using the Unity 3D engine, this course is key, as it teaches students how the engine works in depth.

Table 1.1: Related Subjects

1.3 Project objectives

- Develop a parallelized version of the Wave Function Collapse algorithm capable of generating procedural finite worlds.
 - Implement this parallelized version of the algorithm into a shader to shift the computational load to the GPU, allowing a faster world generation.
- Develop a Unity Tool that allows the users to use the parallelized Wave Function Collapse algorithm easily.
 - Design a logical graphical user interface that allows interaction with the tool without requiring the user to understand its internal workings.
- Create a visually and mechanically engaging video game capable of showcasing the results achieved with the tool, while also providing an interesting and appealing gameplay experience.

1.4 Task Planning and Scheduling

This planning is an approximation of what will ultimately be the actual project schedule, as accurately estimating timelines for the development of both a tool and a game involves a degree of uncertainty. The total estimated workload is 400 hours, 100 hours more than what is outlined in the academic curriculum. However, I am more than willing to dedicate this additional time, as this is a project I am truly passionate about and believe is worth the extra effort. Final time tracking will be conducted using Jira software.

Table 1.2: Scheduling

Preparation (5 hours)	
Technical proposal (5 hours)	Drafting of the technical proposal and outlining the structure of the project
Preproduction (55 hours)	
WFC Research (30 hours)	In-depth research on the implementation and parallelization of the Wave Function Collapse algorithm
Game Design Document (GDD) (25 hours)	Full drafting of the Game Design Document for the demonstration video game
Production (230 hours)	
Programming and Parallelization of the WFC Algorithm on GPU (125 hours)	Implementation of a GPU-parallelized version of the Wave Function Collapse algorithm for procedural world generation
Graphical Interface Programming for the Tool in Unity (15 hours)	Development of a custom editor for the Wave Function Collapse tool in Unity
3D Asset Modeling and Texturing (30 hours)	Creation and texturing of the 3D models to be used in the demonstration game
Programming of Core Game Mechanics (40 hours)	Implementation of the game's main mechanics, including player, world, and enemy behavior
Shader Programming for the Game (10 hours)	Development of URP shaders to enhance the game's overall visual appearance
2D Asset Design (10 hours)	Creation of the 2D assets used in the game's user interface
Post-production (35 hours)	
Publishing the Tool on the Unity Asset Store (10 hours)	Preparation of the required documentation and final adjustments for publishing the tool on the Unity Asset Store
Sound Design (5 hours)	Selection and implementation of the game's audio components
Game Polishing (20 hours)	Addition of subtle effects and quality-of-life improvements to enhance the user experience
Testing and Optimization (20 hours)	
Testing (20 hours)	Time allocated to ensure the proper functionality and performance of both the tool and the game

Documentation (45 hours)	
Biweekly Reports (10 hours)	Drafting of project progress reports
Final Project Report (30 hours)	Writing of the complete final project document
Presentation and Defense (5 hours)	Preparation of the presentation and recording of the defense video

1.5 Expected results

- A tool capable of generating finite worlds, based on a collection of user-created three-dimensional tiles and their corresponding adjacency rules.
- A user interface designed to provide intuitive and straightforward control over the tool's output.
- A visually and mechanically engaging video game that showcases the capabilities of the aforementioned tool, while also delivering a satisfying user experience in line with the quality standards expected of a vertical slice.

1.6 Tools to be used

Programming and implementation

- Unity 6
- Visual Studio Code
- GitHub Desktop

Art 2D and 3D

- Blender
- Krita

Documentation writing

- Google Docs
- Overleaf

Organization

- Jira

CHAPTER 2

Design

Índice

2.1	Wave Function Collapse 2D algorithm	7
2.2	Wave Function Collapse 2D parallelization	8
2.3	Wave Function Collapse 3D algorithm	10
2.4	Wave Function Collapse 3D parallelization	13
2.5	Unity tool	13
2.6	Videogame	18

This bachelor's thesis is composed of two separate parts, the creation of a Unity tool that allows the use of the wave function collapse algorithm parallelized in the GPU to generate worlds automatically and the creation of a video game that uses this tool to showcase its potential while delivering a satisfying player's experience.

2.1 Wave Function Collapse 2D algorithm

The WFC algorithm was originally created by **Maxim Gumin** in 2016 [Gum16]. Gumin introduced two versions of the algorithm, both oriented towards texture synthesis. The first version examined a sample texture and created new textures, maintaining the same patterns and structure as the original sample, while the other version of the algorithm achieved a similar result using a set of tiles and adjacency rules given by the user.

Both versions were built on top of the work done by **Paul Merrell's**

Model Synthesis [Mer07] but we will be only focusing on the tiled version presented by Gumin, as it is the one used in our tool.

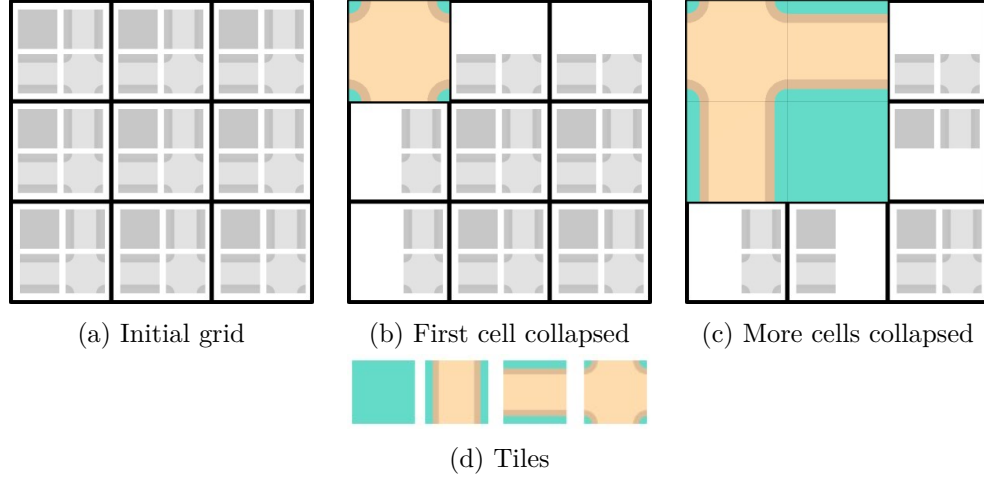


Figure 2.1: 2D WFC example

This version of the algorithm works with a given set of tiles and its adjacency rules, we will refer to these set of tiles as *tilesets* for the rest of the document. These rules indicate which tiles can be next to each other, so for each side of the cell, we will have a set of possible neighbor tiles. Once all is set, the algorithm stores the initial map as a grid of cells that can initially be any tile from the tileset. The algorithm uses the amount of possible cells of each tile as the criteria to decide which tile to collapse, we will name this criteria *entropy*. The cell with less entropy is always selected for its collapse, initially all cells have the same entropy, so a random cell is collapsed, meaning that one of its possible tiles is selected as the final tile. This change affects all cells in the grid as their possible tiles might change to accommodate the new change, cells adjacent to the collapsed one will only have as possible tiles the valid neighbors of the collapsed cell. The same rule applies to all cells towards their respective adjacent cells. This cascade effect is what gives the algorithm the Wave Function name, as all changes propagate to the whole grid. Figure 2.1 shows an example of the algorithm applied to a 2D grid.

2.2 Wave Function Collapse 2D parallelization

As explained in section 2.1, the WFC algorithm needs to propagate each cell collapse to all cells in the grid to ensure proper generation, making all cells dependent on the others of the grid. This global dependency makes the parallelization of the WFC algorithm a difficult task, as the division of the work load must be made carefully to ensure the effects of each cell collapse

are properly applied in the grid. **B. T. Brave** [Bra21] proposed the following solution for this matter.

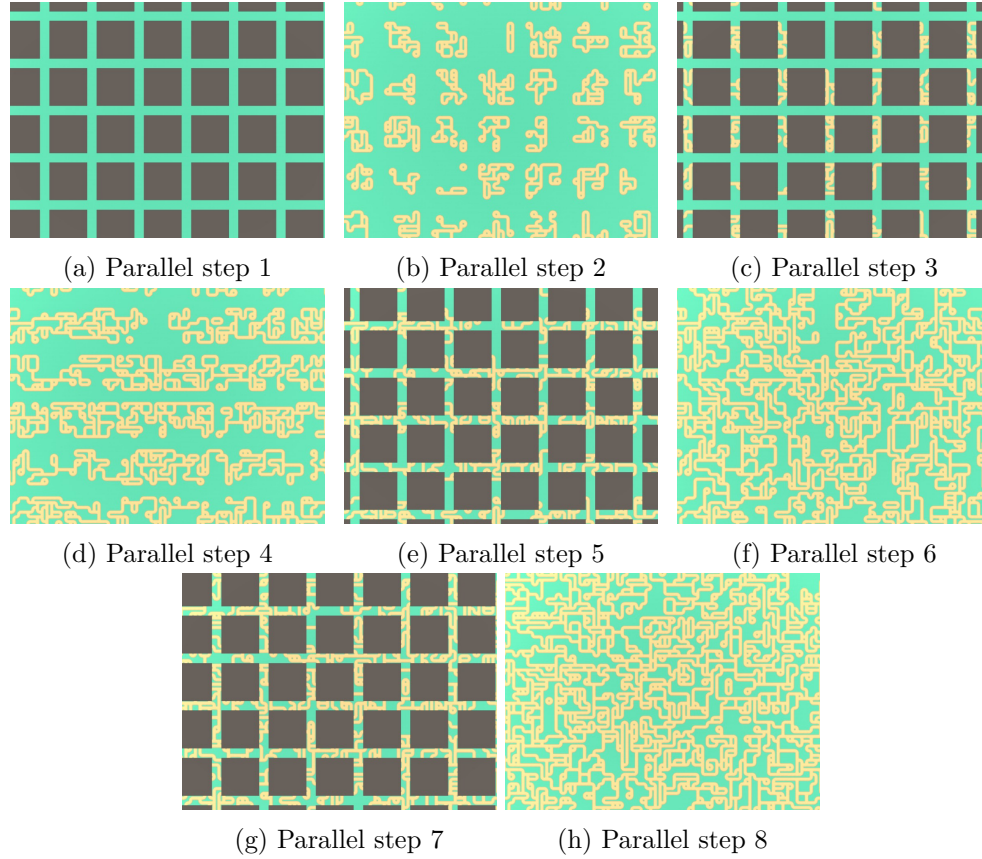


Figure 2.2: 2D WFC example

The algorithm will generate 4x4 chunks of the grid with a gap of one cell between all chunks all at once. This means that each thread will execute the original WFC algorithm within the cells contained in each chunk, limiting the propagation of changes in each chunk's region. This method avoids possible simultaneous memory readings / writings by different threads, as the cells forming the boundaries of each chunk won't be modified.

In order to generate the remaining cells, this previous step will be repeated three more times, each time with an offset of two cells to the right, two cells to the top and two cells to the right and top respectively. In each step, each chunk region is reset to the original tile possibilities for each cell to ensure a clean generation. Even though some of the work done by previous steps is being discarded, some of the previous work is kept due to the offset, allowing to follow the adjacency rules of the remaining cells and generating a new area in each step. Figure 2.2 shows a visual representation of this method extracted

from the **B. T. Brave's** website.

2.3 Wave Function Collapse 3D algorithm

The tool aims to generate 3D environments, so a 3D version of the algorithm is needed. The 3D version of the WFC algorithm builds on top of the original 2D algorithm, adding a new dimension, meaning that the cell grid becomes a tridimensional matrix and each cell will have six adjacent cells instead of four. Apart from that, the algorithm remains the same, each time a cell collapses its possible tiles into a single tile, all its adjacent cells will update their possible tiles and so on.

While the algorithm doesn't change much between its 2D and 3D versions, these two new adjacent cells increment the difficulty of setting the adjacency rules for each tile, as their six faces will need their own set of possible neighbor tiles specified by the user. While this could be done by hand, it is unreasonable to do it, specially when tilesets contain large amounts of tiles.

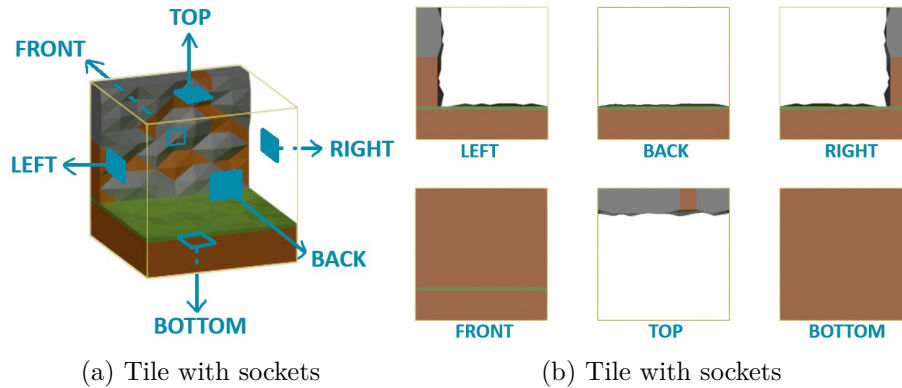


Figure 2.3: Sockets example

To solve this problem and according to the previous work of **María Villar** [Lóp25], a new element must be added to each face of the tiles. This element will be named *socket* and will be used to automatically generate the possible neighbors of each tile face for each tile of the tileset. Sockets have several properties that help define the adjacency rules of the tile face they are attached to. Figure 2.3 shows a visual representation of the sockets each tile would have.

These socket properties vary depending on the axis alignment of the face the socket is attached to, so a distinction between horizontal (back, right, front and left) and vertical (bottom and top) faces is made. Both type of faces have the most important property of the sockets, the socket type, as for a tile to be considered as a possible neighbor of another tile face, their corresponding

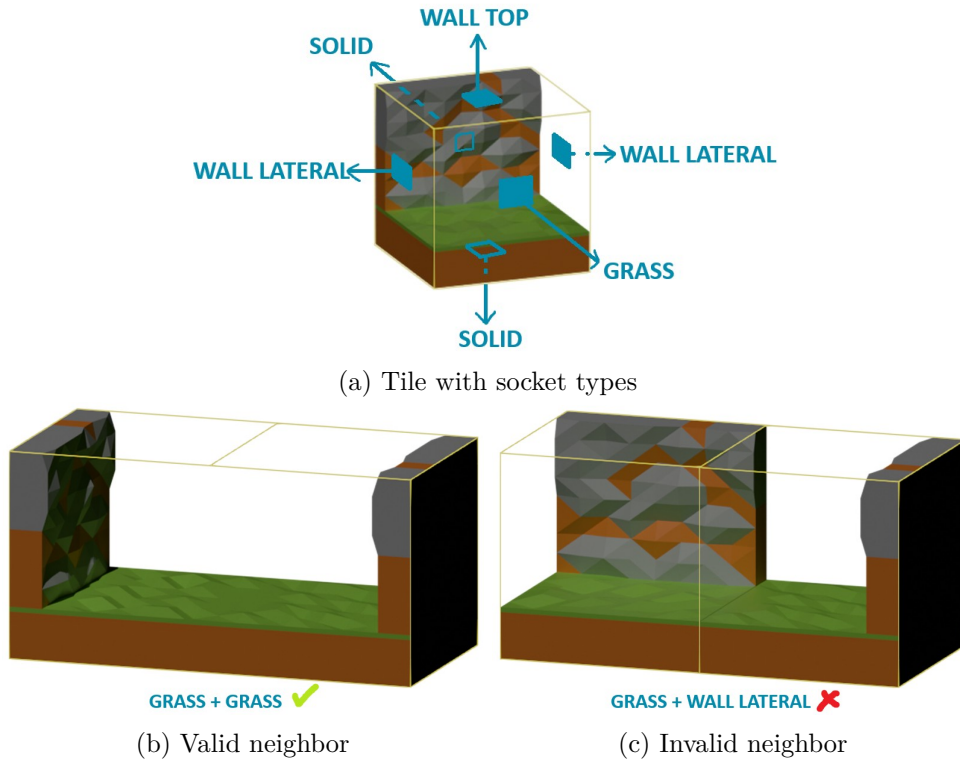


Figure 2.4: Socket type rule example

sockets of the facing tiles must share the same socket type. Figure 2.4 shows an example of the socket type rule being applied.

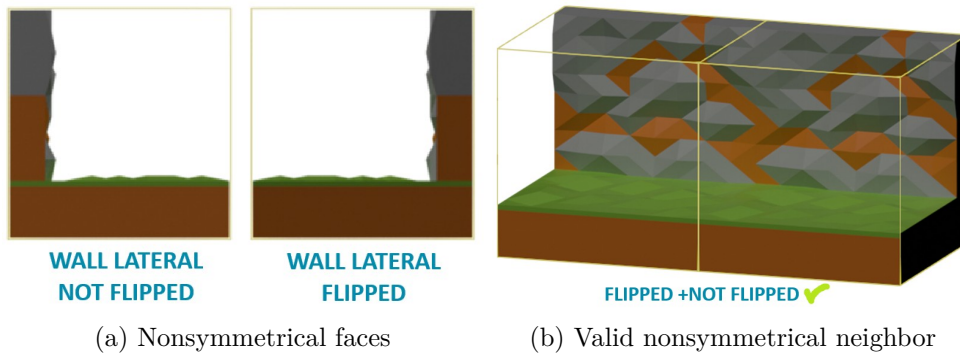


Figure 2.5: Nonsymmetrical horizontal socket configuration

In addition, they must follow some extra rules based on the properties not shared across horizontal and vertical faces:

- **Horizontal faces:** Must both have the property symmetrical selected,

meaning that the orientation of the face isn't considered or one be marked as not-flipped and the other as flipped. As seen in figure 2.4a both faces, left and right have the same socket type assigned, but they are not symmetrical as seen in figure 2.3b, so a distinction is between them is needed, that distinction is the flipped property. Figure 2.5 shows a visual representation of this rule.

- **Vertical faces:** Must be both marked as rotationally invariant, meaning that their rotation isn't considered determining their connectivity or share the same rotation index. Figure 2.6 shows a visual representation of this rule when two sockets are not rotationally invariant.

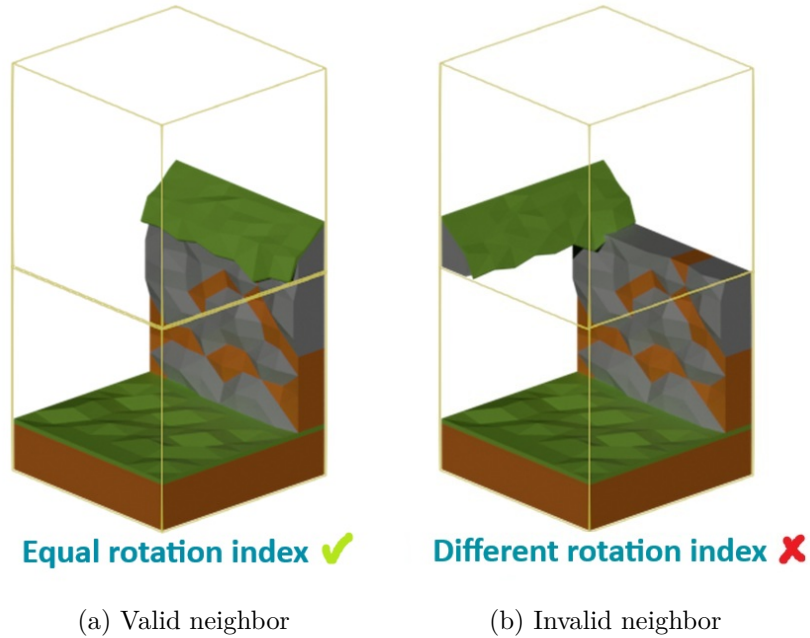


Figure 2.6: Non-rotationally invariant vertical socket configuration

Face	Name	Symmetry	Flipped
LEFT	WALL LATERAL	No	Flipped
FRONT	GRASS	Yes	-
RIGHT	WALL LATERAL	No	Not Flipped
BACK	SOLID	Yes	-

Table 2.1: Sockets example horizontal faces

Face	Name	Rotationally invariant	Rotation index
UP	WALL TOP	No	0

DOWN	SOLID	Yes	-
------	-------	-----	---

Table 2.2: Sockets example vertical faces

Tables 2.1 and 2.2 show a possible configuration for the tile shown in Figure 2.3.

2.4 Wave Function Collapse 3D parallelization

The parallelization of the WFC 3D algorithm follows the same procedure as its 2D version explained in section 2.2 but applied in a tridimensional grid layer by layer. This method divides the desired map grid into horizontal layers and applies the same logic as the 2D version of the algorithm layer by layer. Starting from the bottom layer of the grid, it divides the layer into 4x4 chunks, leaving a gap of one tile between chunks, resetting each chunk's area and applying the WFC algorithm to those areas and repeating the process 3 more times with different offsets to cover the whole grid layer. Upon the end of the process, incompatibilities are checked and in case of them, the process is restarted for the current layer until no incompatibilities are found and continuing to the next layer. Once all layers are completed, the whole map is generated following all the adjacency rules established by the users via the sockets.

Although it would be theoretically possible, to use 3D chunks and a 3D division of the map with a 3D offset, this would always result in irresolvable states due to the lack of lower layer foundations, in other words, the moment a chunk in an upper layer collapses a cell into a tile that covers what it has beneath it, all tiles from chunks under that would have to collapse into solid (non-usable) tiles, but because they are being generated at the same time, this update won't arrive on time, therefore generating an irresolvable state. There's theoretically a chance that all chunks that generate a tile that needs lower tiles to collapse on a solid tile concur with all lower chunks generating solid tiles in the coordinates need, but it's such a lower probability that is not even considered.

2.5 Unity tool

2.5.1 Introduction

General description

This unity tool allows developers to generate procedural and customizable worlds for their games. It uses the 3D Wave Function Collapse algorithm

parallelized with the GPU, only requiring the user to design predetermined tiles for their tilesets and configure their sockets to be able to generate a full working world.

Objectives

The objective of this tool is to reduce the amount of work required to generate worlds for a game. It eliminates the need of handcrafting each level, as the tool is capable of generating completely new procedural worlds while maintaining the same tileset given by the user.

Scope

The tool includes the options to set the size of the world both on the X, Y, and Z axes, generate as many tiles and sockets as needed and create tile variations automatically, all integrated within Unity's editor UI.

Main audience

The main audience for this Unity tool are game developers that need a fast and reliable way of generating worlds and artist that want to showcase their modular assets.

2.5.2 System Requirements

Functional Requirements

- The tool must be able to generate a map of any given dimensions, using any given tileset and following its tile's socket configuration.
- The tool must be able to configure as many tiles and tilesets as the user specifies.
- The tool must be able to manage as many socket types as the user specifies.

Non-Functional Requirements

- The tool must be efficient to reduce performance impact in the editor.
- The tool must work with the latest versions of Unity.

Dependencies

- Unity 4.2 or higher.
- Compute shader compatible deployment platform.

2.5.3 System architecture

Architecture diagram

Figure 2.7 shows the architecture diagram used for the tool.

Design patterns

The tool architecture applies various software design patterns to ensure a seamless integration with the Unity Engine. The **Command Pattern** is utilized implicitly through Unity's Undo and Redo system, allowing user actions, such as object creation or parameter modification, to be tracked and reversed. The two main UI scripts function as **Facade components**, abstracting complex interactions between user input, asset generation, and scene manipulation into a clean interface. Additionally, object and asset instantiation follows the **Factory Pattern**, enabling the creation of game objects and assets based on the parameters defined by the user.

2.5.4 User interface design

Prototypes

Figure 2.8 shows a mockup of the main windows of the Unity tool integrated into Unity's custom editor windows. They use all design and fonts provided by Unity to maintain the visual aspect of the engine and fit in.

Navigation flow

The navigation flow in the tool will be separated into 2 main processes. First, the user will need to create all the tiles needed for their project and set their adjacency rules by adjusting the socket properties of each tile face. The left window shown in Figure 2.8a will be used for this purpose, as it will allow the user to set all the parameters explained in section 2.3 and create a tile asset with the bottom button. In case, the user selects an already existing tile asset, the window will show its data and the button will modify the asset instead of creating a new one.

Once the user has finished creating the tile set, it will navigate to the second window shown in figure 2, where the generation parameters will be set. Once all parameters have been set, the bottom button will allow the user to create a WFC Generator game object in the active scene. In case an already existing WFC Generator game object is selected, the window will show its data and the button will modify the existing game object instead of creating a new one.

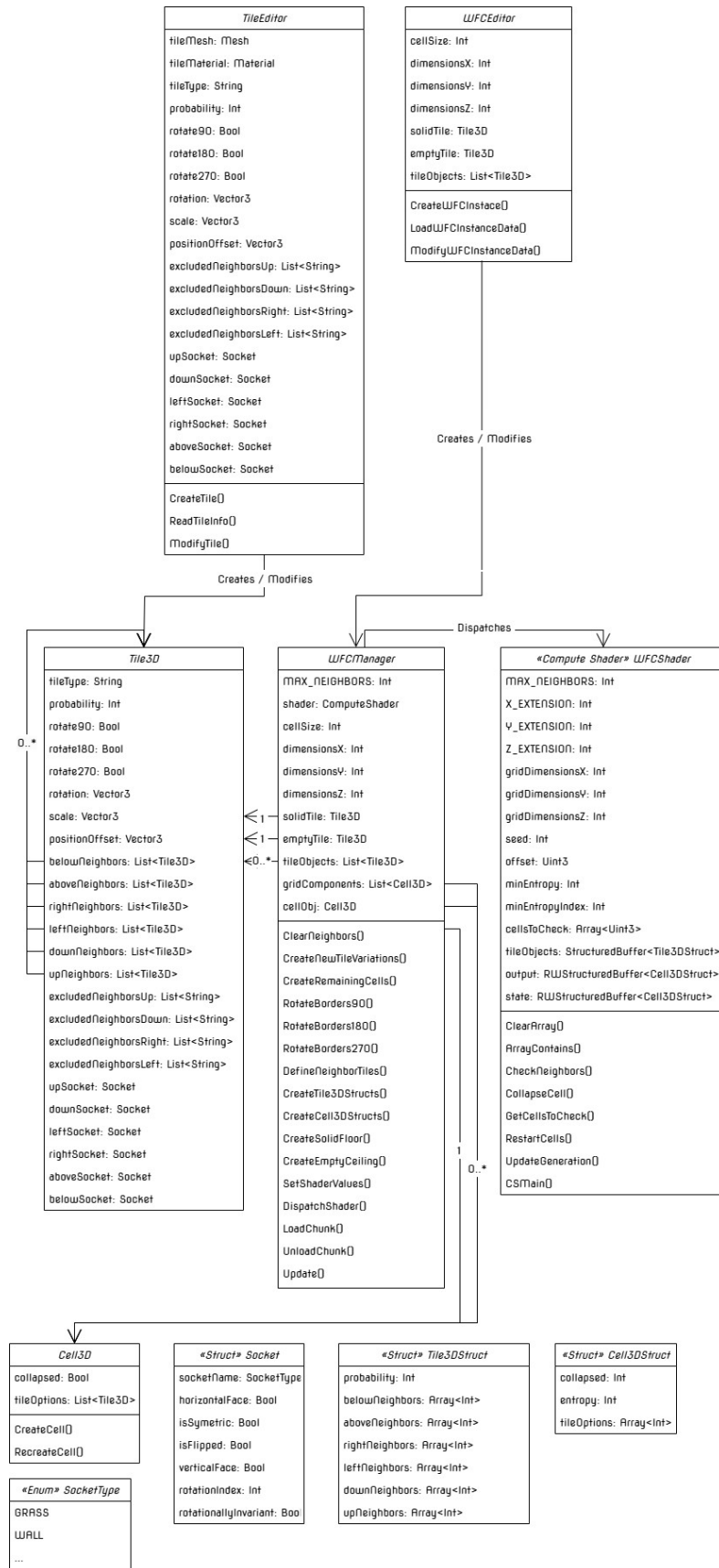
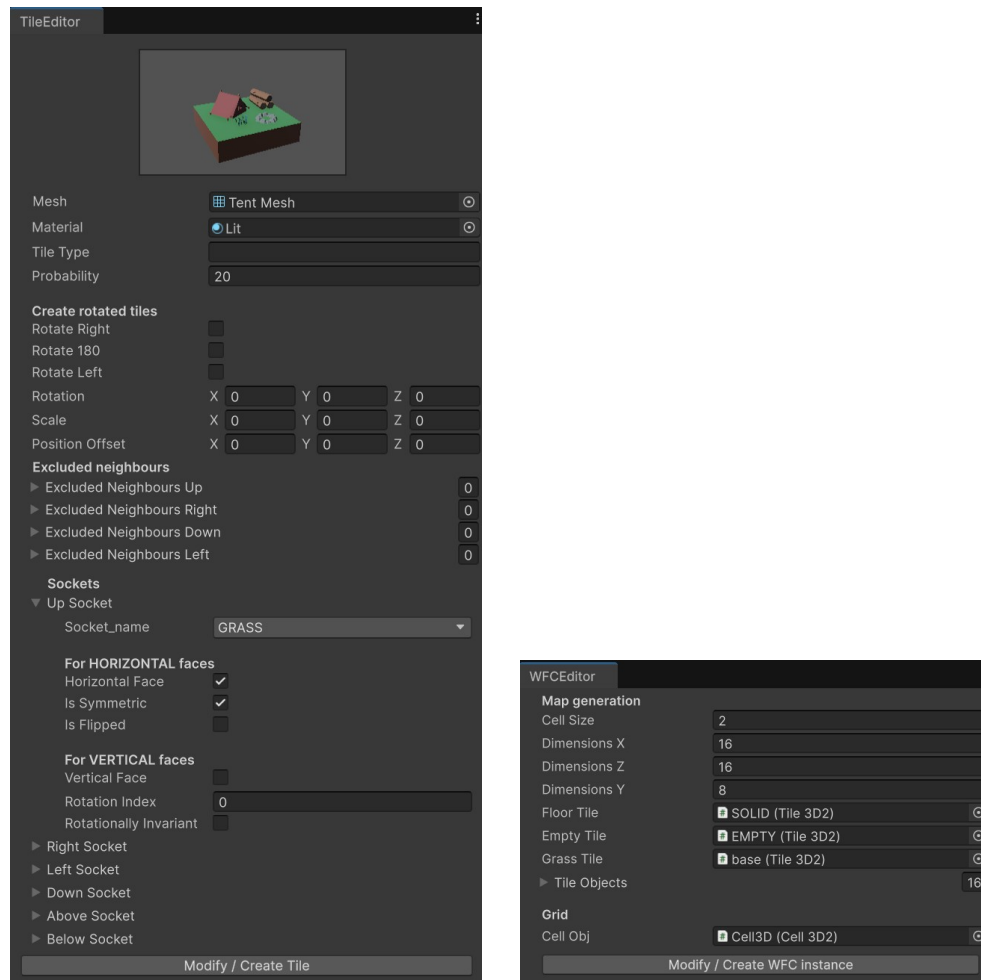


Figure 2.7: Architecture diagram



(a) Tile editor

(b) WFC editor

Figure 2.8: Tool windows mockup

2.5.5 Test and validation

Test plan

The test for this tool will be done, testing it directly in a test scene. In this scene, several tile sets will be defined, and maps of different dimensions will be generated to test for failures.

Accepting criteria

For the tool to be accepted, all tests explained before must be passed. That meaning that the tool must be capable of generating a map of any given size with any given tileset and socket configuration.

2.6 Videogame

2.6.1 Introduction

Title

System escape

Game concept

System escape is a 3D top-down adventure game based in a solar system which star is near the end of its lifetime. As the star starts expanding before its final collapse, all the astronauts in the planets leave their planets in a rush for their lives, except for you. As a stranded astronaut you'll have to find your way out of the solar system before the supernova erases the system from existence, gathering resources to build a spaceship and adventuring into the other planets in the system, fighting enemies and collecting better materials to upgrade your equipment until it's powerful enough to bring you out of the danger area.

Game's objective

The main objective of the game is to showcase the power of the Unity tool explained before as all the planets within the game will be procedurally generated with the tool, with different biomes, therefore different tilesets. In terms of gaming experience, the game aims to present the player with a fast-paced adventure centered in exploration and time and resource management.

Platforms

The only platform where the game will be available is PC.

Target audience

The target audience of System Escape are players that like both space exploration and replayable experiences like Astroneer or The Outer Wilds.

2.6.2 Game mechanics

Player controls

- Player / Spaceship movement: WASD or Left Joystick.
- Accelerate / Decelerate spaceship: Shift / LeftControl or Right Trigger / Left Trigger.
- Jump / Fly: SPACE or South Button.
- Collect / Attack / Interact: Left Click or West Button.
- Use gadget: Right Click or East Button.

- Switch between gadgets: Q / E or Left Button / Right Button.
- Open mission pad: TAB or Select Button.
- Pause menu: ESCAPE or Options Button.
- Zoom in / Zoom out: Mouse Wheel or Dpad Up / Down.

Game rules

Winning condition

- Escape the solar system before the supernova occurs.

Losing conditions

- The supernova occurs while you're in the solar system.
- The planet you are in is eaten by the star.
- Your life meter comes to 0.
- You run out of oxygen for more than 5 seconds.
- Your spaceship is totally destroyed.

Objectives

- Escape the solar system before the supernova.
- Upgrade your equipment to overcome each new planet's challenge.
- Upgrade your spaceship to finally escape the planet.
- Gather resources to upgrade your equipment.
- Explore the planets to discover new materials.

Restrictions

- You can't escape the solar system until your spaceship isn't equipped with the interstellar engine.
- You can't enter a planet that has been eaten by the star.
- You can't enter the star.
- You can't get an upgrade on both your spaceship and equipment without the right materials.
- You can't get an upgrade on both your spaceship and equipment that requires previous upgrades without unlocking them.
- You can't gather materials that require an equipment upgrade to be gathered without unlocking the upgrade.

Game systems

Health

- Your max health is 100 life points.

- If your health reaches 0 you'll die, losing the game.
- Your health will decrease after each hit received by an enemy.
- Your health will start recovering at a rate of 1 life point each 0.5 seconds.


Oxygen





- Your initial max oxygen level is 60L.
- Your oxygen level decreases at a rate of 0.5L per second.
- Your oxygen level will increase at a rate of 15L per second when you're near your spaceship or near space wrecks.
- Your max oxygen level can be increased with upgrades to your oxygen tank.
- Planet conditions can alter the consumption rate of oxygen.

Combat

- Your player will aim automatically at the closest target when a weapon gadget is equipped.
- Your attack range and damage will vary depending on the weapon gadget equipped.
- Enemies will follow the player when entering their vision range.
- Enemies will attack the player when entering their attack range.
- Some enemies might have a nest that will keep spawning them at a defined rate.
- Enemy nests can be destroyed by attacking them.
- Each enemy and nest will have a set amount of health that when reduced to 0 will kill or destroy them.

The Table 2.3 describes all the enemies in the game:

Name	Planet	Description	Stats	Reference
Arrakworm	Colis	A tiny worm that moves under the sand of the dessert, it can detect its preys thanks to the vibrations in the sand. When nearby a prey, it jumps out of the sand and attacks.	<ul style="list-style-type: none"> • Life points: 50 • Damage per hit: 5 • Detection range: 10m • Attack range: 2m 	

Rhinosite	Colis	Medium size armored bugs that live in the desert. They're not so smart, they'll charge at anything as soon as they see it.	<ul style="list-style-type: none"> Life points: 100 Damage per hit: 10 Detection range: 10m Attack range: 6m 	
Culex	Phobos	Big flying bugs similar to mosquitoes that shoot toxic liquid and move in flocks.	<ul style="list-style-type: none"> Life points: 100 Damage per hit: 10 Detection range: 10m Attack range: 8m 	
Dionaea	Phobos	Carnivore plant that has various mouths, generally hunts Culex, but it'll attack anything nearby by launching one of the mouths and trying to bite anything within its attack range.	<ul style="list-style-type: none"> Life points: 200 Damage per hit: 15 Detection range: 10m Attack range: 5m 	
Gaculex	Regio	An evolution of the Culex found in Phobos. Instead of shooting liquid, it fills an area with toxic gas. The gas damages the player each second.	<ul style="list-style-type: none"> Life points: 200 Damage per hit: 20 Detection range: 10m Attack range: 4m 	




Victorite	Regio	An ancient bird similar to a dinosaur that attacks its preys by diving with its claws on them.	<ul style="list-style-type: none"> • Life points: 300 • Damage per hit: 20 • Detection range: 10m • Attack range: 5m 	
Edwatcher	Platum	A biped monster, similar to a velociraptor that, thanks to the lack of atmosphere of its planet, has mutated into a silent assassin that stalks its victims while being almost invisible and then stabs them with its claws.	<ul style="list-style-type: none"> • Life points: 400 • Damage per hit: 25 • Detection range: 10m • Attack range: 1m 	
Beltrax	Platum	Similarly to the Edwatcher, the Beltrax species are lone predators that attack their victims, climbing onto them.	<ul style="list-style-type: none"> • Life points: 400 • Damage per hit: 20 • Detection range: 10m • Attack range: 1m 	

Table 2.3: Enemies

Equipment

- The player will start with basic equipment.
- The equipment can be upgraded at any moment with the necessary materials.
- Equipment might have previous equipment unlocks needed for unlocking them.

- Equipment upgrades will increase the stats of the player like oxygen capacity, attack resistance, movement speed and material capacity.

The diagram in Figure 2.9 shows the equipment upgrades tree with its costs and dependencies:

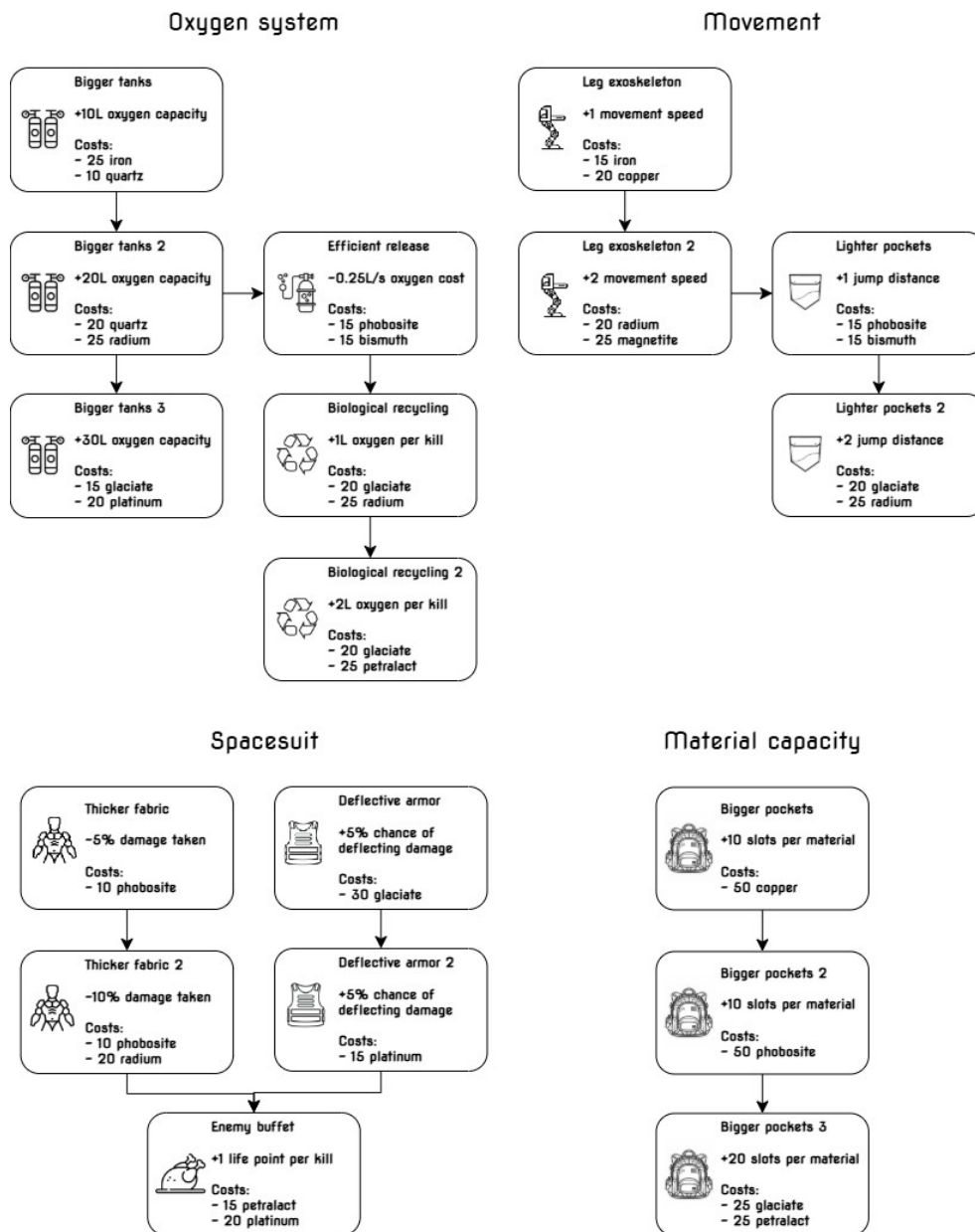


Figure 2.9: Equipment upgrades tree

Spaceship

- The spaceship is meant to be a transport medium to travel between planets.
- The spaceship can be upgraded at any time with the necessary materials.
- Spaceship upgrades might need previous spaceship upgrades to be unlocked before unlocking them.
- Spaceship upgrades will affect the spaceship stats like max velocity, acceleration, deceleration, and the ability to land in new planets.

The diagram in Figure 2.10 shows the spaceship upgrades tree with its costs and dependencies.

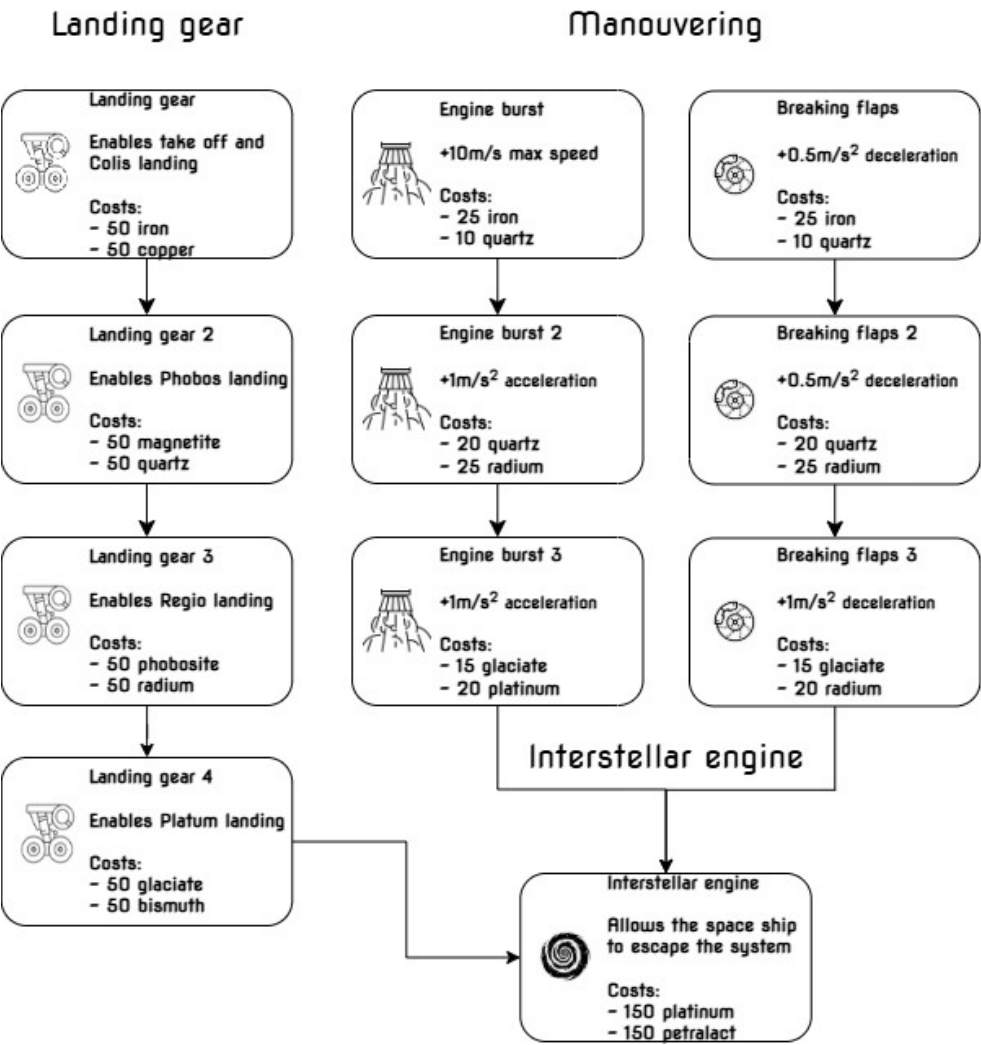


Figure 2.10: Spaceship upgrades tree

Gadgets

- The player will start with 2 simple gadgets, a light sword and a gathering tool.
- Gadgets can be upgraded and crafted at any moment with the necessary materials.
- Gadgets upgrades and crafting might have previous gadgets unlocks needed for unlocking them.
- Only one gadgets might be selected at a time.
- There might be some gadgets that once activated remain activated for a period of time even when not equipped, like guardian drones.

The diagram in Figure 2.11 shows the gadgets upgrades tree with its costs and dependencies:

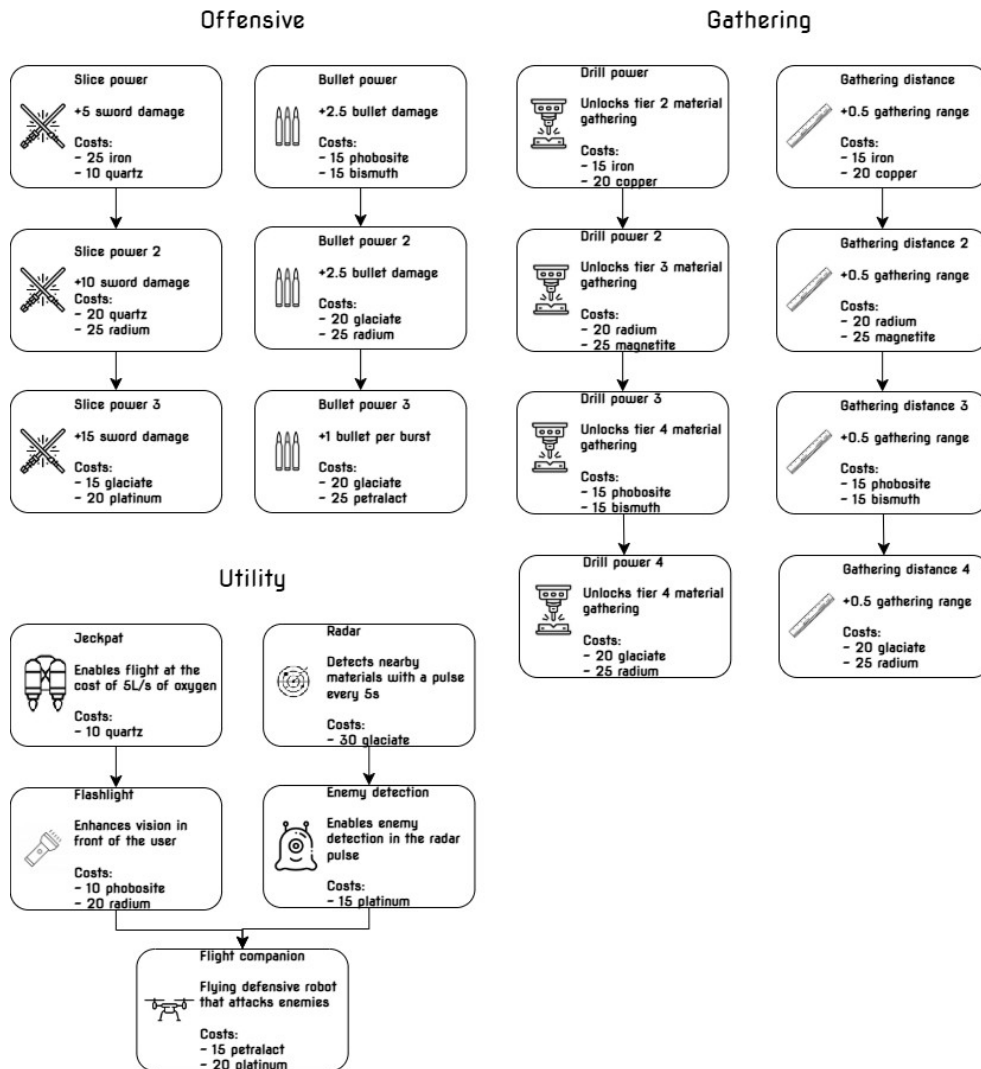

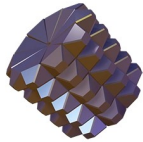


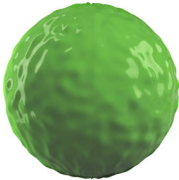
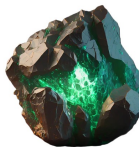


Figure 2.11: Gadgets upgrades tree

Materials

- Materials are distributed in each planet.
- Materials are categorized in different tiers.
- Each tier requires an upgrade for the player's gathering tool.

The Table 2.4 describes all the materials in the game:

Name	Tier	Description	Reference
Iron	1	Iron rock formations shaped like cubes	
Copper	1	Copper rock formations shaped like stars	
Magnetite	2	Solid rods that form clusters	
Quartz	2	Pink crystals that form short sharp layers	
Phobosite	3	Spherical containers made out of thick liquid merged with rocks	
Radium	3	Glowing green rocks merged with basic rocks	


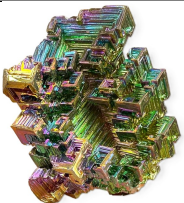


Glaciate	4	Icicles full of gas bubbles that form sharp clusters	
Bismuth	4	Erratically shaped cubes that merged one with each other forming prisms	
Platinum	5	Shiny metallic rocks formed in the side of mountains	
Petralact	5	Slimy transparent substance formed after biological matter suffers the effects of no atmosphere for too long	

Table 2.4: Materials

Solar system

- The solar system is composed by the main star and 5 planets.
- The star will grow, will destroy a planet every 20 minutes.
- Planets are orbiting around the star at a constant speed.
- Planets are distributed in equidistant circular orbits around the central star.
- Planet destruction order is determined by the closeness to the star, meaning that the closest planets will be destroyed first and further ones last.
- Each planet contains new materials and materials from planets closer to the star.
- Each planet has its own set of enemies and biome conditions that might affect the player's stats differently.

The composition of the solar system will be specified in the section 2.6.4.

Progression

- The player will start its journey in the closest planet to the star, acting as a tutorial.
- The player will have to learn to gather materials and fight basic enemies to fix the spaceship before leaving the first planet.
- Once exited the first planet, the player will have to go through every single planet, gathering new resources and unlocking new upgrades and gadgets to face the new challenges and unlocking every spaceship upgrade until reaching crafting the interstellar engine and leaving the solar system.

Physics and game logic

- Game objects will work using rigid body physics.
- Planets will have a set size, measured in tiles generated by the unity tool, explained before.
- Planets will be divided in memory persistent chunks, loading only the ones near the player.
- Planets are procedurally generated each time the player enters them for the first time.
- The solar system will have world bounds that will prevent the player from going too far into space without the interstellar engine, in case the player has unlocked it, the game will end, and a victory screen will be shown with a summary of the things done by the player.

2.6.3 Story and narrative

General synopsis

As part of the Interstellar Space Exploration Agency (ISEA), you and your comrades of the James Web 3 mission are adventuring into an unexplored solar system to study the effects of a dying star in its neighbor planets. According to the information that you were given by the ISEA, the mission was completely safe, but suddenly a new emergency announcement arrives to your team. *“CAUTION: The Star Supernova is imminent, leave the system at all cost. ESTIMATED REMAINING TIME: 0’ 100” 00”’ ”*. As the messages arrives, your team starts panicking and leaves the planet, without noticing that you were left. Now, alone and stranded, you’ll have to find your way out of the system.

Main character

- **Name:** Clark Jackson
- **Description:** Clark Jackson is a 35 years old man, 1.80 meters tall that served to the US Navy in his 20s to finally afford college and becoming an

aerospace engineer for the ISA. He's a passionate for space exploration and a talented engineer, part of the James Web 3 mission.

- **Motivations:** Clark Jackson's main motivation is to undercover the secrets from space to contribute to society once he comes back from the mission. He has a special interest in the development of new green energy technologies, and believes that a Dyson Sphere might be the ultima energy source for humanity.

Story and world background

The game takes place in the year 2050, when humanity has recently discovered the interstellar engine. This invention allows the spaceship to travel between solar systems. With the barriers of space travel distance broken, a new aerospace agency was created, the Interstellar Space Exploration Agency (ISEA) with the main objective of researching new solar systems. One of the missions directed by the ISEA is the James Web 3, a mission to study the effects of a dying star on its system's planets. Our main character, Clark Jackson, is selected as part of the crew for this mission. Once all members were prepared, they were deployed in the closest planet "*Mercum*", the closest to the star "*Andromedae*". The mission was supposed to be completely safe as the star's life span was still somewhat far from its end, but there was a miscalculation and the crew had to abandon the system as soon as possible. Unfortunately, Clark Jackson was left behind in the escape, so now he'll have to find its way out alone before the final supernova.

Missions and story progression

In terms of story progression and missions, the game features a single mission, escape the "*Andromedae system*" before the star dies and erases the system from existence. You'll have to keep moving from planet to planet, adapting to the environment, so the story progression is tied to the choices made by the player.

2.6.4 Level design

Andromedae system map

The game takes place in the "*Andromedae system*". The system is composed by 5 different planets that orbit around the main star in equidistant circular orbits at a constant speed. The star will grow at a continuous speed, destroying near planets each 20 minutes.

Planets

The following table 2.5 the characteristics of each of the planets (for a more detailed explanation of enemies and materials, see section 2.6.2 and 2.6.2):

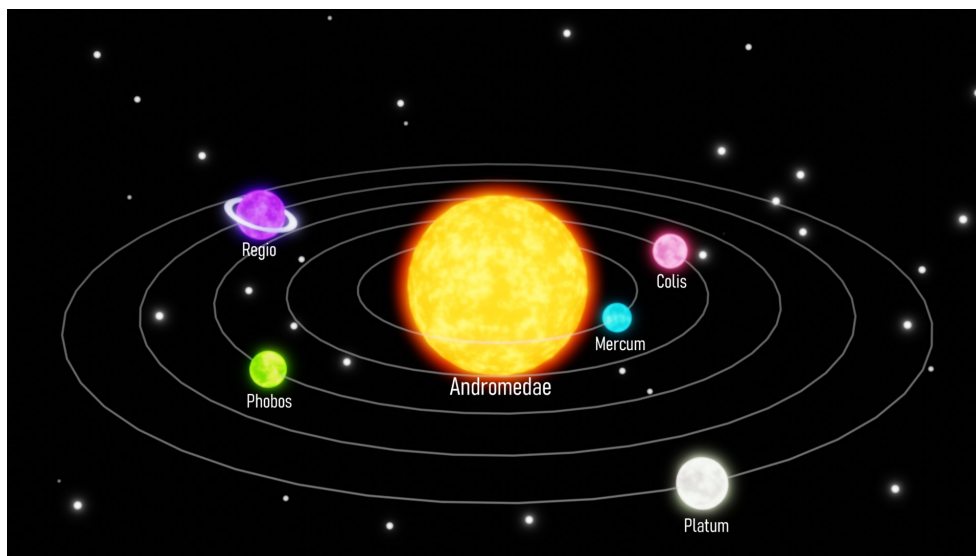


Figure 2.12: Andromedae system map

Name	Description	Size	Difficulty
Mercum	Initial planet that serves as a tutorial for the player. Mainly composed by water and earth-like terrain with a clear atmosphere that doesn't affect oxygen consumption. Tier 1 materials can be found with no enemies in sight	Small	Easy
Colis	Pinkish desertic planet that is mainly made out of sand and rocky terrain with a sandy atmosphere. Tiers 1 and 2 materials can be found here, accompanied by terrestrial enemies	Medium	Easy
Phobos	Corrosive planet that is mainly made out of rocky terrain and toxic water. Its corrosive atmosphere doubles the oxygen consumption of the player, and toxic water damages the player when in contact. Tiers 1, 2 and 3 materials can be found here, accompanied by both terrestrial and flying enemies	Medium	Medium

Regio	Gaseous planet that is mainly made out of islands separated by the thick gas of the atmosphere. The atmosphere makes is difficult to see the surroundings. Falling into a pit results in the death of the player. Tiers 1, 2, 3 and 4 materials can be found here, accompanied by flying enemies	Large	Medium-high
Platum	Moon-like planet that is mainly made out of mountains. The lack of an atmosphere has made the enemies of this planet evolve into more dangerous versions. Tiers 1, 2, 3, 4 and 5 can be found here accompanied by terrestrial and flying enemies	Large	High

Table 2.5: Planets

Difficulty curve

Each planet presents the player with new movement challenges and new enemies that gradually require the player to get better upgrades and use new equipment. This results in a constant rise of difficulty through the planets, with the addition of the planet destruction that ultimately forces the player to travel to newer and more difficult planets.

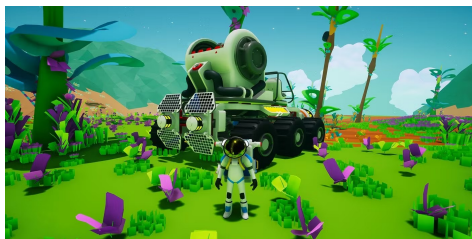
2.6.5 Art and visual style

Art direction

System escape’s art direction falls directly into the 3D low poly category, with a zoomable 3:4 view controlled by the player. It’s main visual reference is the game “*Astroneer*” developed by *System Era Softworks*, released in 2016. Figure 2.13 shows a set of in game captures that show the color full, low poly art of “*Astroneer*” that inspires the whole art style of this game.

Character and enemy design

The main character design is mainly based on a space suit, all his body is covered by it, even his face is not visible due to the anti - UV ray visor. The suit is mainly made out of white fabric. Figure 2.14 shows a character from “*Astroneer*” as a reference.



(a) Atrox from “*Astroneer*”



(b) Desolo from “*Astroneer*”



(c) Novus from “*Astroneer*”



(d) Sylva from “*Astroneer*”

Figure 2.13: Art references from “*Astroneer*”



Figure 2.14: Design reference for Clark Jackson from “*Astroneer*”

Enemy designs are detailed in the enemy table at section 2.6.2.

Environment and scenarios



Figure 2.15: Generic world chunk generated by the Unity Tool

As explained in section 2.6.4 System Escape’s world is divided into 5 different planets, each of them with a different set of tiles and color palette. The main color of each planet’s palette can be seen in the “*Andromedae system*” map shown in figure 2.12 and a more in depth explanation of the orography of each planet can be found in section 2.6.4. As a secondary effect caused by the use of the wave function collapse algorithm in the Unity tool explained at the beginning of this document, all planets are structured in chunks made out of cubic tiles. Figure 2.15 shows an example of a generic chunk generated by the tool that resembles how the planet’s structure will be. Keep in mind that the tile set will fit both the color palette and description explained in the section mentioned before.

UI and UX



Figure 2.16: Mission pad from “*Astroneer*”

All UI and UX in System scape will be integrated within a pad that the character will have hanging from his belt. In order to access the inventory,

check health and oxygen levels, manage upgrades or even enter the pause menu, the player will have to stop and open take out this device. This means that the UI is fully integrated into the game world, being diegetic. Figure 2.16 shows the reference pad from "*Astroneer*".

To provide the players with a better game experience, sounds will be re-produced to indicate important information, such as low oxygen levels or low health levels, and changing the gadget in use won't require the use of the pad.

2.6.6 Sound and music

Soundtrack style

System Escape's soundtrack style is based on the "*Astroneer*" and "*Minecraft*" soundtracks, both featuring "*Ambient music*" that brings a feeling of joy and exploration but with a deep familiar and cozy touch. In System Escape's soundtrack, the sound design aims to achieve this exploration feeling but changes the familiarity and coziness for an insecure and tense background. To do this, the soundtrack will be purely instrumental, faster and more intense than the referenced games, a closer approximation is the song "*Pig Step*" added in the latest "*Nether Update*" for "*Minecraft*".

Key sound effects

Some sounds are needed for the basic functioning of the game:

- UI click sound
- UI change selection sound
- Damage taken sound
- Sword swing sound
- Gun burst shot sound
- Enemy attack sound (one for each enemy)
- Low health sound
- Low oxygen sound
- Planet eaten by the star alert sound
- Spaceship engine sound
- Enemy death sound
- Material collected sound
- Gathering sound
- Death sound
- Oxygen refill sound

2.6.7 User interface

Main and in-game menus

The game features a single non-in-game menu, the main menu, where the player will have the option of starting a new run, changing the sound and graphical settings of the game and exiting the game. This menu will be composed by the title of the game at the top left side, all menu buttons under the title and a real-time render of the “*Andromedae system*” as the background. New submenus as the settings menu will appear on the right side of the screen once activated. For the in-game menus, the player will have 5 menus incorporated into the characters pad as explained in section 2.6.5. One of the menus will show the general state of the player and its inventory. Another 3 will show the progression trees for the Equipment, Spaceship and Gadgets upgrades as shown in sections 2.6.2, 2.6.2 and 2.6.2. Finally, the last menu will show the pause menu with the options to resume, quit and return to the main menu.

Indicators

As explained in the section 2.6.7, the character general state menu will also display 2 bars, one for the current health of the player and another one for the current oxygen level of the player.

Accessibility options

All menus in the game will be interactive with the mouse and game pad, navigating between all the available buttons by hovering over them with the mouse or navigation between them with the D-pad of any game pad.

2.6.8 Technical aspects

Game engine

The game will be fully developed in Unity 6.

Programming languages

C# will be the main programming language used for the game scripts, along with HLSL for shader creation.

IA and NPC behavior

As the game only features enemy NPCs, the AI used for them will be a basic enemy behavior system with a navigation mesh. Enemies will follow the player if it enters their vision range, until reaching their attack range, they'll attack in the direction of the player.

Tool development

Índice

3.1	WFC compute shader adaptation	37
3.2	Shader management in the CPU side	40
3.3	Chunks generation mode	43
3.4	User interface	45
3.5	Tool results	49

This chapter shows an in-depth explanation of the development and implementation of the tool in Unity, including the whole process and decisions made along the way. It also explains the modifications made to it to better fit the tool's purpose, along with the results obtained after finishing both the development and implementation.

3.1 WFC compute shader adaptation

One of the main advantages of GPU computation lies in its capability to execute a massive number of threads in parallel, making it particularly well-suited for algorithms that can be decomposed into independent tasks. For this reason, the WFC algorithm, specifically, its 3D parallelized version as explained in Section 2.3 has been adapted to run on the GPU using a compute shader written in HLSL.

Compute shaders operate in a specialized programmable shader stage designed for general purpose computations. Unlike traditional shaders, they

are not part of the graphics rendering pipeline and must be explicitly dispatched. When dispatching a compute shader, the number of thread groups to be launched needs to be specified, and each thread group runs a fixed number of threads, defined by the `[numthreads(x, y, z)]` directive in the HLSL code. In this implementation, as explained in Section 2.4, each layer of the map is generated in parallel by dividing it into smaller chunks, each of which is handled by a separate thread group.

Due to the limitations of GPUs, the limited number of temporal registers available per thread group, this implementation restricts each group to 2x2 threads. This is a trade-off to maintain high performance, as exceeding the register budget per group can lead to register spilling and a significant drop in performance.

Each thread in a compute shader is uniquely identified by a 3D coordinate known as the `DispatchThreadID`, which is automatically assigned by the GPU at dispatch time. This ID combines the thread's position within its group and the group's position within the global grid of thread groups. Specifically, the global ID is computed as:

$$\text{DispatchThreadID} = \text{GroupID} \times \text{NumThreads} + \text{GroupThreadID} \quad (3.1)$$

Where `GroupID` is the index of the thread group, `NumThreads` is the number of threads per group in each dimension, and `GroupThreadID` is the thread's local index within its group. This system allows each thread to independently compute and operate on a unique region of the data. In this WFC adaptation, the shader is designed to process a 4x4 region of the map, with the origin of each region determined by its corresponding `DispatchThreadID` and an additional offset that controls the offsets needed to leave a gap of one cell between chunks while also displacing the generation to cover all the layer as explained in section 2.2.

That data is accessed via a structured buffer that stores the map in the form of a cell struct array, each cell struct containing the information of its state, collapsed or not, its entropy and an array of its possible tiles. Unlike C# classes, structs can't contain other structs as properties, so the possible tiles saved as indexes of another structured buffer that contains an array of tile structs, each of them containing the parameters of each tile in the tileset, these parameters being the arrays of compatible neighbor tile for each of the tile's faces, this ones being also saved as indexes of this same buffer. In addition to these two buffers, a third one is used to track any error occurred in any thread of the dispatch.

Once the initial index of the map area covered by the thread is calculated, the shader calculates the indexes of the rest of the cells in the area to ease later access to them. Then, all that cells are reset to their default values to eliminate the effects of previous dispatches, preparing the area for the WFC generation. From that point, for each cell in the area covered by that execution of the shader, the neighbors of each cell are updated based on their adjacent cells and its possible tiles, saving the index of the cell with less entropy at the end of the update. This index is used to access the cell that will be collapsed, meaning that a random tile from its possible tiles is selected as the final tile. This loop is repeated until all cells in the area are collapsed, or an incompatibility is encountered. Figure 3.1 shows a diagram representing the shader processes.

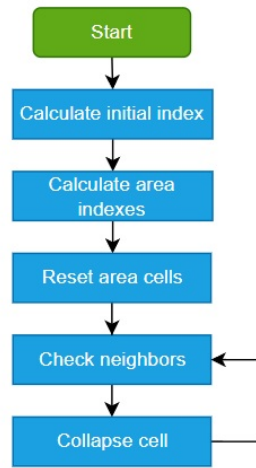


Figure 3.1: WFC compute shader diagram

The specific implementation of the neighbor check and tile collapse are broke down as follows:

- **Check Neighbors:** This operation is done once for each of the cells every time a cell is collapse and at the start. The way it is done is by going through all cells in the area and for each of them check if their array of possible tiles is still valid. To do that, it is checked tile by tile in the array of possible options if that tile appears at least in one of the list of neighbors in the corresponding direction of any of the possible tiles in the neighbor cells. With that checked, a new array is built with only the new possible tile for the cell we are checking.
- **Collapse cell:** This operation is done after all neighbors are checked for all cells in the area. The cell with entropy (less possible tile options)

is chosen to be collapse. Collapsing is choosing a random tile from its possible tiles array, this is done using the seed given by the CPU modifying it based on the index of the cell we are collapsing and then obtain the modulus of the that seed and the number of possible tiles for the cell we are collapsing, which gives us an index that will be the chosen tile. If the array is empty, that means we've reached an incompatibility, this will update the state buffer to indicate it.

3.2 Shader management in the CPU side

As explained above, the compute shader is in charge of the WFC logic, but it needs a method to prepare the data before loading it into the GPU buffers, manage the dispatches and retrieve the data from the buffers once the dispatches are done. This is done by a C# script that runs in the editor when the tool needs to be used. The script is divided into three phases, data preparation, shader dispatching and data retrieval.

3.2.1 Data preparation phase

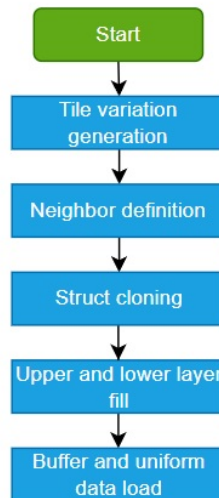


Figure 3.2: Data preparation phase diagram

The data preparation phase is in charge of making the data arrangements necessary for it to be used in the shader, as it needs all neighbors to be pre-calculated for every face of every tile. Before calculating the neighbors, the tool integrates a method that allows the automatic generation of tile variations, meaning that all versions of tiles containing symmetries are generated automatically, freeing the end-user from the burden of creating variations for each tile by hand.

For example, if the user defines a wall tile, all 3 variations of this tile will be calculated by creating a copy of the original tile, rotating the mesh inside the game object 90° , 180° and 270° respectively and also switching the adjacency rules of each face of the cell to match the new rotation. Figure 3.3 shows this example visually.

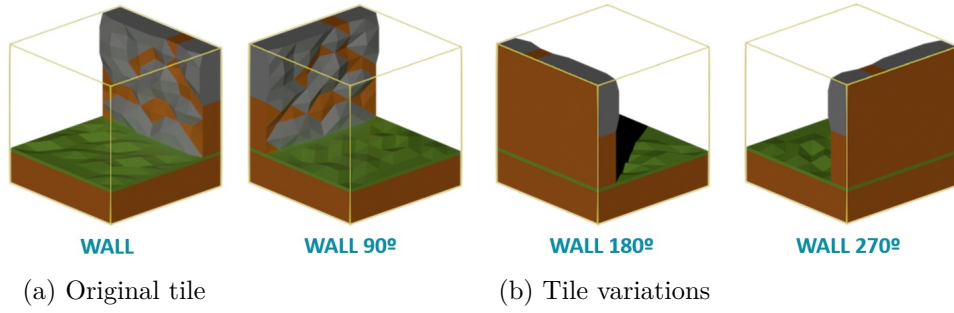


Figure 3.3: Automatic tile variation generation

After the tile variation generation is done, the process of defining neighbors begins. It uses the sockets and their parameters explained in section 2.3 to achieve this goal. It goes through all the tiles in the tileset and for each of them, it checks one by one all the other tiles in the tileset, comparing each face's socket with its counterpart, the top face of the cell being checked against the bottom face of the cell being checked with, right face with left face, front with back, etc. In addition to the socket rules, a new parameter has been added to allow the user to define exception rules. These rules prevent neighboring of tiles that would be compatible by the socket rules, but the user doesn't want them to occur. To do so, the user can add a type to each tile and exclude any tile type on any face of a tile, preventing compatibility on that tile face for those specific types.

Once all neighbors are precalculated for each tile face, the 3D grid of the map is fully generated based on the dimensions specified by the user and filled with uncollapsed cells that have all tiles in the tileset as a possibility. These cells and tiles are stored as instances of the `Cell()` and `Tile()` classes, which is convenient when working within the CPU, but as explained in section 3.1, the compute shader used for the WFC generation can't handle C# classes and uses structs instead. This requires an extra process in the data preparation phase, the generation of structs based on the corresponding class instances previously used in the preparation phase. It consists of cloning the grid and tiles in the tileset into arrays of cell and tile structs respectively with the same data so it can be pushed into the buffers declared for the shader.

Finally, all tilesets have two mandatory tiles added by the tool, the solid and empty tiles, these tiles are used to fill the air parts of the map and the

map parts beneath the surface. To ensure the upper and lower limits of the map are properly filled and don't have voids, the upper and lower layers of the map are collapsed into empty and solid tiles before sending the data to the buffers.

This buffers, along with another uniforms needed by the shader such as the X, Y and Z dimensions of the grid, the offset vector and the seed are loaded into the GPU before dispatching the shader. The seed is used for the random tile collapse as the GPU can't natively generate random numbers, so a seed is needed that is modified by the `DispatchThreadID` and the cell to be collapsed index. Figure 3.2 shows a visual diagram of this phase.

3.2.2 Dispatch control phase

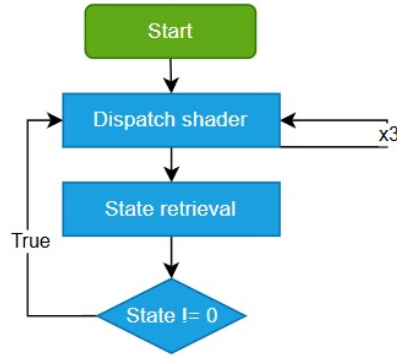


Figure 3.4: Dispatch control phase diagram

The dispatch control phase is in charge of performing the necessary dispatches needed to generate a map based on the dimensions, tiles and tileset specifies by the user. This phase follows what was explained in section 2.4, dividing the map into one tile thick layers and handling the dispatches needed to generate each layer. As explained in section 2.2 for each layer, four dispatches of the shader are done, each of them with a randomly generated seed and a new offset. These offsets are (0, layer, 0), (2, layer, 0), (0, layer, 2) and (2, layer, 2) to properly cover for all tiles in the layer and ensure proper propagation of the changes across the layer. After the dispatches, the state buffer data is retrieved and check, this data represents the amount of incompatibilities found during the dispatch on all threads, if different from zero, something failed and the layer is redispached until zero incompatibilities are found. Once the final layer reaches this zero incompatibilities state, the dispatch control phase finishes, as a proper map has been generated. Figure 3.4 shows a visual diagram of this phase.

3.2.3 Data retrieval phase

The data retrieval phase is the last phase needed for completing the map generation. At this the whole grid of the map is fully collapsed with no errors, so the original cell structs are replaced by the data retrieved from the shader buffer and the necessary tile game objects are instantiated into the scene as children of the original grid cells. Finally, the hierarchy is modified, eliminating all cell parent objects, empty tile and solid objects, leaving a clean hierarchy that the user can understand and manipulate if needed. The way this phase works is just a loop that instantiates the correct tile on each cell based on the struct information. Figure 3.5 shows the complete diagram of the tool parallel mode.

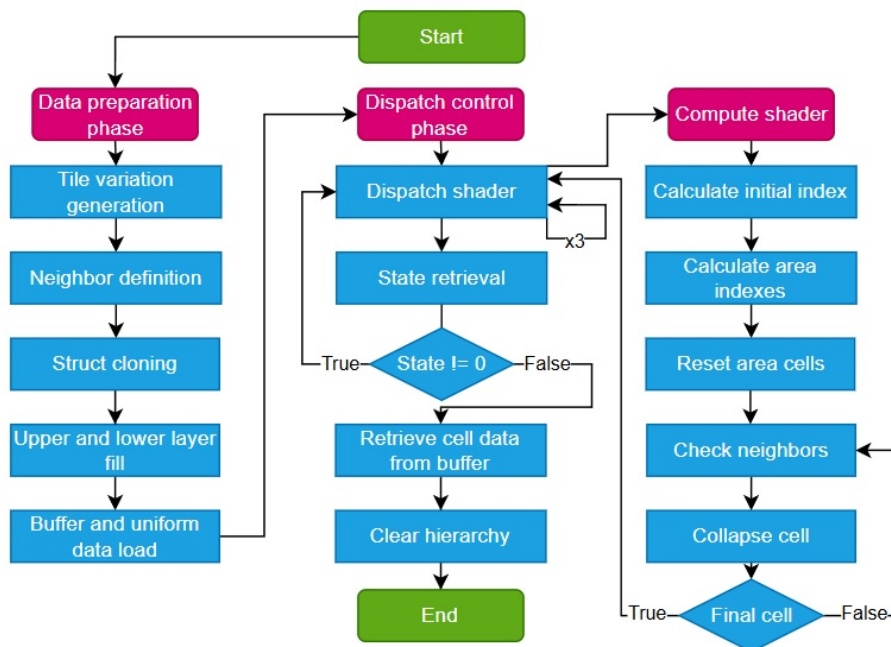


Figure 3.5: Complete parallel mode diagram

3.3 Chunks generation mode

The original idea of this project was to only implement the parallel approach as it seemed to be the perfect solution for the problem, but after weeks of tweaking the shader and trying new options, a problem that couldn't be solved was found with that approach. Parallel generation works really fast until a certain threshold is met, 16 by 16 is the last map size where the generation times are reasonable. This is because as the map size increases, the odds of

finding incompatibilities increase and for each incompatibility the tool needs to restart the process.

As this threshold seemed too low, an alternative chunk generation mode was created. For this mode, a separate C# script is used along with a modified version of the compute shader to accommodate the needs of this new approach.

This approach is similar to the parallel version but adds a few steps to the algorithm:

- **Chunk subdivision:** The first step in this mode is to divide the original grid into 4x4x4 cubic chunks and based on the number of chunks, generate as many offset coordinates as needed, this coordinates indicate the position of the bottom left cell of the area that will be obtained as a sub-grid.
- **Subgrid copy:** For each of the offsets obtained in the last step, an area of 3x3x1 chunks is copied to a smaller grid, and an equal subgrid is created to save the indices of the original grid that corresponds to the ones in the subgrid. This is done to always load the same amount of data to the GPU buffer, as in the parallel approach, the whole grid had to be loaded into the buffer.
- **Dispatch layer by layer on the center chunk of the subgrid:** Now that the subgrid loaded in the buffer, a single dispatch per layer is done with a group size of 1x1x1, and a thread count of one. This eliminates parallelization, but eliminates the problem of concurrent memory accesses to cells surrounding the area. Being able to update that area, reduces the amount of incompatibilities to almost none, so the profit obtained is larger than the performance sacrifice.
- **Shader working area update:** As a modification of the shader, it is now needed to update an area one tile bigger than the area that is going to be modified, meaning that a loop of update neighbors is run once for the whole working area and then the generation starts in the smaller area corresponding to the chunk, this is the reason why the subgrid obtained before is bigger than the chunk generated, because the algorithm needs the information about the cells surrounding the area that is going to be generated to prevent incompatibilities.
- **Backtracking:** When the information is received back in the CPU, a counter has been added to control the attempts for each chunk, if it exceeds a limit of 100 attempts, the last chunk is regenerated, trying to find a new solution that might fit better the next chunk.

In addition to these changes, the function in charge of cleaning the hierarchy has been modified to group all cells in chunks and assign coordinates to the names of these chunks to present the results of the generation to the end-user in a clean form.

This approach is capable of generating a map that greatly surpasses the limitations of the parallel approach at the cost of performance, but with a linear increase in time consumption instead of exponential time consumption as the parallel approach had. This approach is not perfect either as even with the backtracking implemented it might end up finding an incompatibility due to not updating the whole map after each cell collapse, but gives satisfactory results almost every time if the tile set is properly configured. Figure 3.6 shows a complete chunk mode diagram.

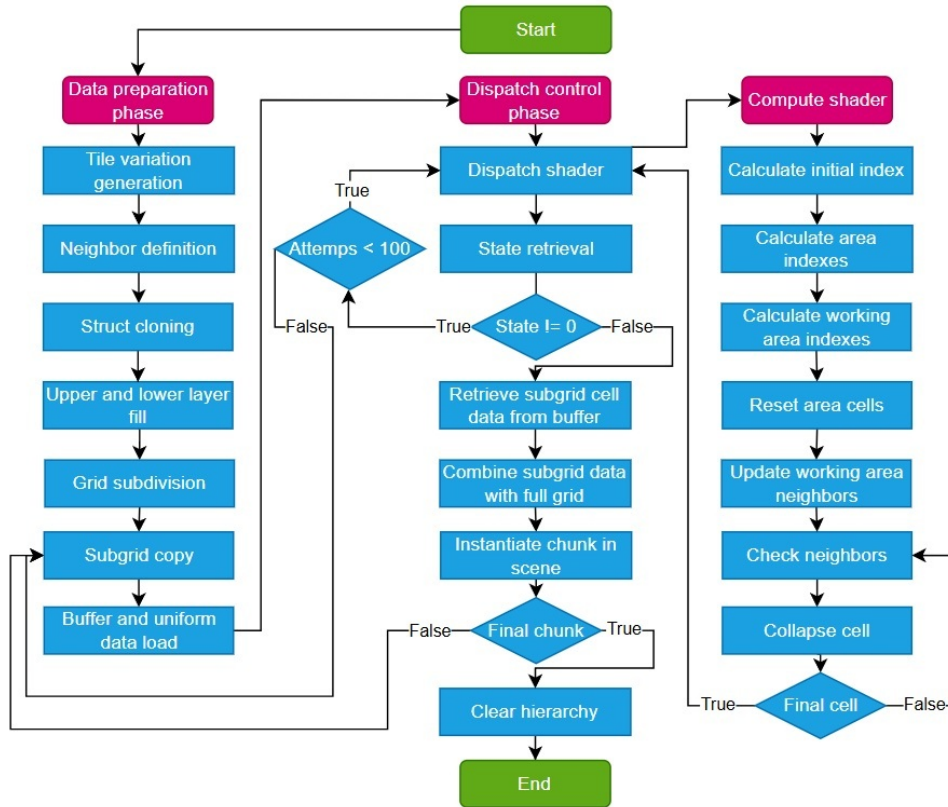


Figure 3.6: Complete chunk mode diagram

3.4 User interface

As a tool, the user needs an interactive interface that abstracts all the logic explained above and gives the user the ability to use the tool easily. For that reason, two different interfaces have been created, one for managing different

tileset and tiles and another one to control the actual generation using those tileset.

The class `Tileset()` has been created as a scriptable object to be able to store the data of each tileset, that data being the tiles contained in that tileset, the socket types created for that tileset, the tile types created for that tileset and the size of the tiles contained in the tile set.

3.4.1 Tilset editor UI

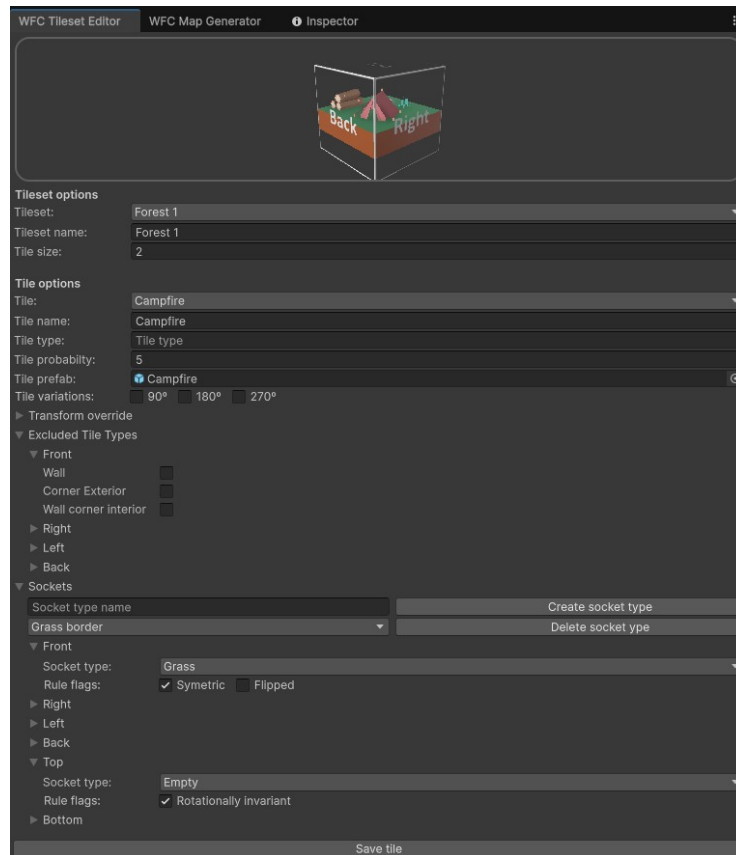


Figure 3.7: Final tileset editor UI

Figure 3.7 shows the final result of the interface created for the tool. This interface has been created using Unity's UI Toolkit and is paired with a C# script that allows the use and creation of the tool's interface as a window of Unity and manages the different aspects of the tool.

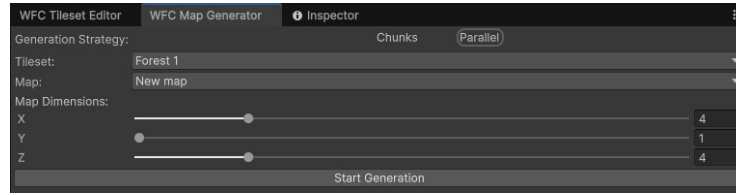
- The tool automatically reads the data in the Asset's folder of the user looking for the directories that it has established and if it doesn't find them it automatically creates them.

- The tool lets the user load an existing tileset or create a new one with the dropdown at the top of the window and allows the modification of the tileset name and tile size at any moment given.
- The tool loads automatically all the tiles of a tileset, allowing the user to switch between the existing ones or create a new one.
- When a tile is selected, the user can rotate the 3D model of the tile, zoom in and out and see an indicator that tells the user which face corresponds to which socket. This was done by instantiating an object in the scene, adding a camera and rendering the view of that camera into the editor, the same method as unity uses (both camera and object can't be seen by the user).
- The user can change the name of the tile, the type of the tile, its probability and the prefab attached to that tile, the variations wanted and the transform override in case his models are not centered properly.
- The selection of excluded types for each socket has been converted into a procedural set of checkmarks to make sure the user always enters information of existing types and gives a more intuitive interaction.
- Sockets now have two sections where the user can create as many socket types as he wants with just a text field and a button and eliminate any of the existing at any moment. If there are any sockets using an eliminated socket type, the script goes through all the tiles, fixing it.
- All sockets are now as simple to configure as clicking the interface a few times to change all the parameters needed.

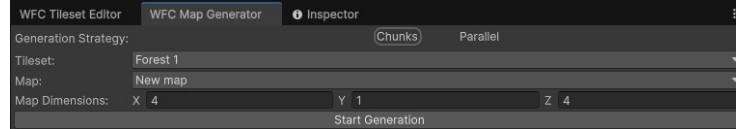
Finally, all work done in this window is saved as the user presses the save button at the bottom of the window to make sure that the user is sure about the changes he has done. Once saving is occurring, the script automatically renames the objects, modifies the data directories if needed and moves the objects to their corresponding new folder to keep the project of the user organized.

3.4.2 Map generation UI

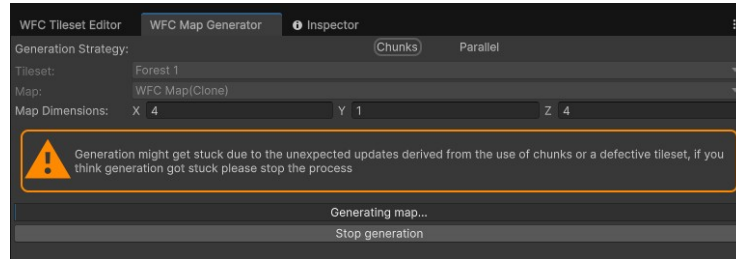
Figure 3.8 shows the interface of the map generation window of the tool, this interface is simpler than the tileset editor one because once the user has prepared the tileset, generating the map only requires the tileset he wants to use, the object he wants the map to be child of and the generation mode that he will be using.



(a) Parallel mode



(b) Chunk mode



(c) Interface while generating

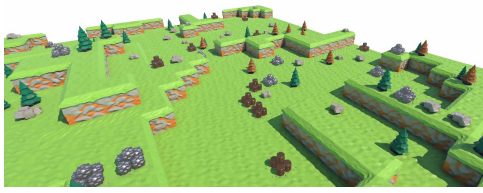
Figure 3.8: Map generation UI

This interface was created using the same methods as in section 3.4.1, using Unity's UI Toolkit and attaching a C# Script to it, that manages the scripts needed for the generation.

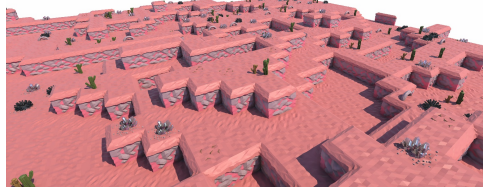
- The user can change the tileset used for the generation at any moment, except while generating a map.
- The user can select the parent object of the map or generate a new one, if the tool detects that the selected one contains a map it will reset that map.
- The user can select the chunk generation or the parallel generation, depending on which one is more convenient for his purpose.
- The user can select any map size, the tool makes sure the parameters entered are valid and limits the input with slider if parallel mode is selected.
- Once generation of the map has started, the user is given an alert that generation might get stuck, so a progress bar and a stop button are shown to him in case he needs them.

Based on the parameters set by the user, the tool calls the parallel or chunk version of the algorithm and gives them the parameters needed to output a result based on the user input.

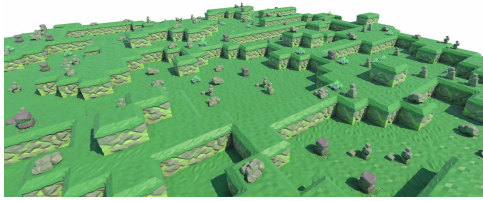
3.5 Tool results



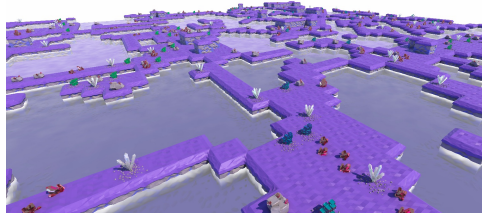
(a) Mercum tileset



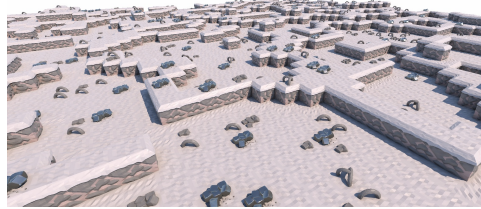
(b) Colis tileset



(c) Phobos tileset



(d) Regio tileset



(e) Platum tileset

Figure 3.9: Tool results obtained for the System Scape game

Map Size	CPU version	Parallel mode	Chunk mode
16x3x16	14s	0.8s	3s
32x3x32	140s	-	6s
64x3x64	1400s	-	12s

Table 3.1: Time improvements over CPU implementation

The tool has achieved the expected results marked at the start of this project, it's a great alternative to other procedural generation approaches and

greatly simplifies the use of the Wave Function Collapse algorithm to the end-users, allowing them to create maps with complete freedom on how they look. The interface is intuitive and filled with tooltips to help the user navigate through its options.

Even though the tool results are good and according to what we expected from it, several problems were encountered along the way, being the main one that the compilation times of the shader are too long, limiting the maximum amount of tiles to 50 including its variations, it is still a wide range, but it's a limit after all. Figure 3.9 show the worlds created for the game System Scape, also part of this project, each scenery has a unique tileset with different tiles and a different map size, proving the tool capable of generating any size environments with any tileset given by the user while giving great results. On the other hand, table 3.1 shows a great improvement over the CPU version of the tool, proving that taking workload to the GPU is effective.

Game development

Índice

4.1	Player movement	51
4.2	Resource gathering	53
4.3	Combat	54
4.4	Upgrade system	55
4.5	Spaceship movement	56
4.6	Game loop	57
4.7	Interface	57
4.8	Sound system	58
4.9	2D art	59
4.10	3D art	60
4.11	Results	62

This chapter shows the development and implementation of the video game System Scape, which serves as a demonstration of the tool capabilities.

4.1 Player movement

4.1.1 Input gathering

As the game is meant to be played with controller or keyboard and mouse, the newest Unity input system has been used to map then input of the player in the most generic way. This system allows the use of any type of controller without the need of having a specific input mapping for each controller type.

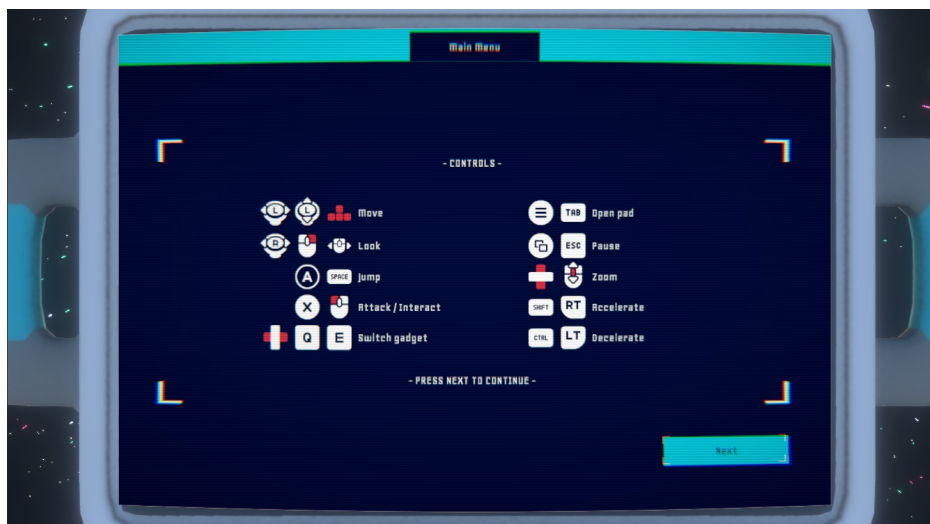


Figure 4.1: Controls of the game shown in the UI

These inputs have their different event calls attached to the player controller class, which is in charge of managing these inputs and translate them into actions based on the state of the input at that moment. Figure 4.1 shows the final controls panel inside the game's UI, showing the mapping of both game pad and keyboard and mouse actions.

4.1.2 Player



Figure 4.2: Player using the jetpack

The player controller script translates the vector obtained from the left axis stick or WASD and sets the speed of the player to the result of that direction

multiplied by the player speed, avoiding accelerations, decelerations and allowing proper air control without inertia. This approach is rather non-realistic but gives the player a more comfortable control over the player movement.

Where inertia is applied is in the jump and jetpack implementations, when the input attached to these actions is first pressed, an impulse force is applied upwards to the player, leaving him in a free fall state once the inertia of this force is gone. Then, if the player has the jetpack upgrade and pressed the button again, a continuous upwards force will be applied to the player until the player stops holding the button down or runs out of fuel for the jetpack. Figure 4.2 shows a capture of the player using the jetpack.

Apart from that, collisions are handled in a classic way, using Unity's rigid body physics.

4.1.3 Camera

As before, the player controller script translated the vector obtained from the right axis stick or mouse delta and sends it to the camera controller script, which rotate the camera accordingly while maintaining the player in sight. To do this, the camera controller uses a ray cast to avoid collisions of the camera with other objects in the scene and the actions to zoom in and out have been mapped to the d-pad and mouse wheel, which both modify the effective distance of the ray cast. Figure 4.3 shows the camera set in different angles and with different zoom configurations to show the results of the custom camera controller.



(a) Zoomed out

(b) Zoomed in

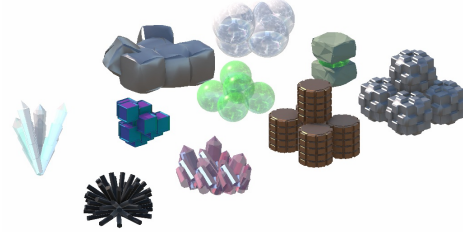
Figure 4.3: Camera controller examples

4.2 Resource gathering

The resource system has been implemented by creating a set of prefabs, each representing a type of resource that can be obtained by the player. These resources require the player to have certain upgrades to be able to obtain and require a certain amount of time for gathering them.



(a) Material gathering



(b) In-game Materials

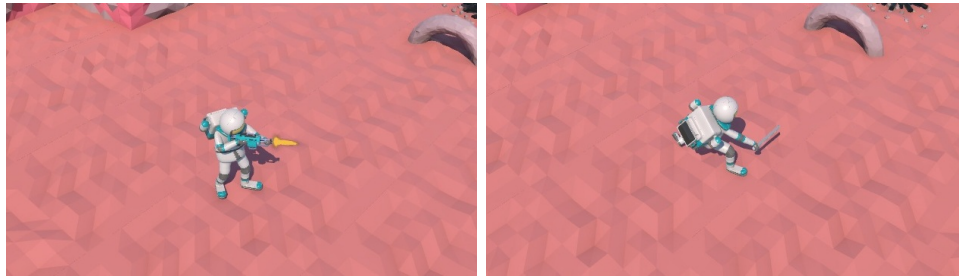
Figure 4.4: Gathering system example

Once the player is in range of gathering a material, if he holds down the interaction button, a countdown will start in the material that will change the visual state of it until breaking it down, once it's completely gathered, it's added to the players inventory and if the player stops interacting mid-way through the gathering process, the material returns to its original state, resembling the block behavior of *"Minecraft"*.

Additionally, the way the gathering range is managed is by using a sphere trigger with a radius equivalent to the gathering range of the player that detects when a material enters or leaves the range. Once they enter or leave the range, materials are added or removed from the list of available materials in the `GameManager()` singleton class, which controls almost every aspect of the game that needs to be shared across game objects. When gathering materials, the closest one to the player from the ones in range is chosen. In addition, when gathering a material, a line renderer is used for aesthetic purposes and to indicate which material is being gathered. Figure 4.4 shows the player gathering a material in the planet Regio and all materials in the game.

4.3 Combat

The combat system implements two weapons, a sword and a rifle, these work in the same way, a sphere trigger is used to detect enemies that enter the area of the rifle, as it is the weapon with larger range. Once the player decides to attack, all enemies are ordered by distance to the player and the closer one is selected as the target. In case of using the rifle, a burst shot is made, meaning that the enemies will receive the damage amount of the rifle three times with an interval of 0.1 seconds, resembling as it was shot three times. In case the player is using the sword, an additional check is done to make sure that the closest enemy to the player is within the sword range and then damage is applied to the enemy. Figure 4.5 shows the player equipped with



(a) Rifle attack

(b) Sword attack

Figure 4.5: Combat system weapon showcase

different weapons and performing the attack animations.

Even though the implementation is rather simple, this methods in addition with different animation set for each weapon, attack and hit animations, automatic rotation of the player's model towards the target and sound effects makes the combat feel alright. Originally, the game had 10 enemies planned but, due to the addition of the chunk mode to the Unity tool also made in this project, the time for making the game had to be reduced. This reduction of time has forced the elimination of almost all enemies from the game, except from one simple enemy. The game is still enjoyable as the enemies were just an obstacle to make the task of escaping the planet more challenging, but are not necessary for the game to still be an enjoyable and complete experience. This enemy consists of a static structure that absorbs health from the player when he gets in its range.

4.4 Upgrade system

The upgrade system works by having a scriptable object that stores the type of upgrade as a type of an enum, if it has been obtained or not, its costs in materials and a list containing the previous upgrades that need to be unlocked before obtaining it. There's an instance of this scriptable object saved in the project files for each of the upgrades existing in the game and detailed in figures 2.9, 2.10 and 2.11 with their corresponding dependencies and costs. Figure 4.6 shows the in-game interface used for obtaining the upgrades.

The way upgrade unlocking is managed:

- When an upgrade is obtained, the upgrade object is passed to the `Game-Manager()` singleton so it can handle its unlock.
- First, the system checks if all previous upgrades required for the new one are obtained.

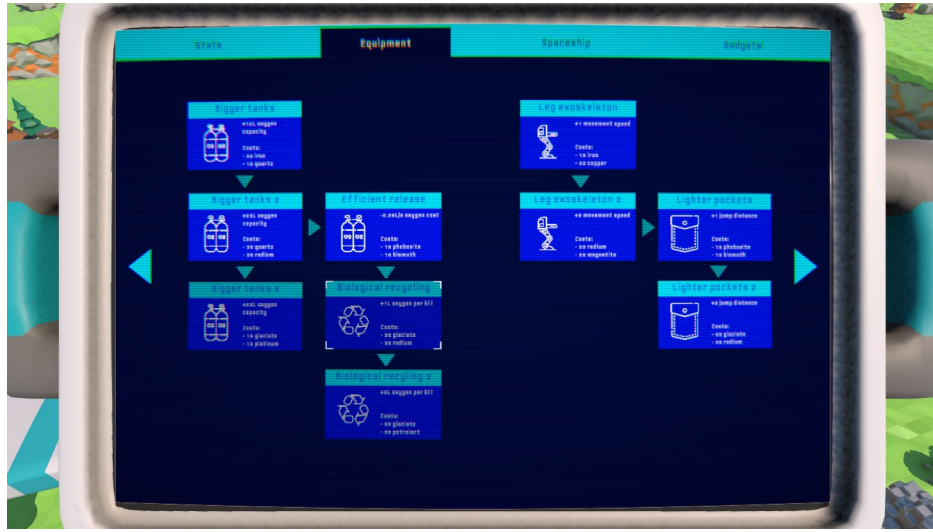
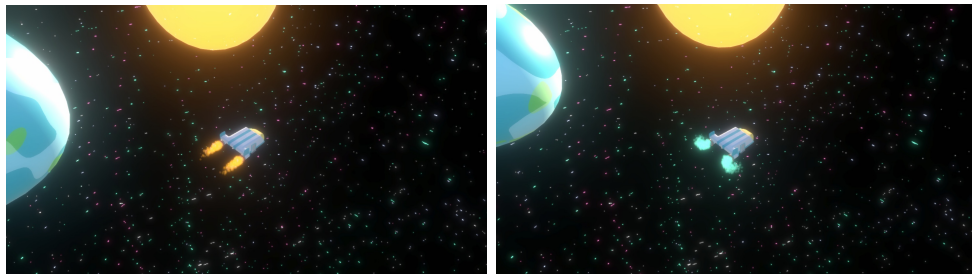


Figure 4.6: Upgrade UI panel used in the game

- Then it checks if the player has enough resources to get the upgrade.
- Finally, if all the criteria has been met, the system applies the upgrade, modifying the corresponding variables with the corresponding data depending on the upgrade type specified in the upgrade.

4.5 Spaceship movement



(a) Spaceship acceleration

(b) Spaceship deceleration

Figure 4.7: Spaceship accelerating and decelerating

The movement of the spaceship is implemented similarly to tank controls. The input is taken using the same method explained in section 4.1, this time, the left stick axis horizontal input and A/D are taken to rotate the spaceship on its Y axis while the right and left trigger axis as well as shift and control keys are used to accelerate and decelerate the spaceship respectively. This acceleration is constant, and no friction is applied, this is done to better represent the movement in space. To help the player navigate through space,

the spaceship movement system is tied to the X and Z axis, avoiding vertical movement as it isn't needed in the game and would only interfere with the objective of moving between planets. In addition, particle systems have been placed on the spaceship engines to indicate when the spaceship is accelerating or not, as it can be difficult if no celestial bodies appear on camera. Figure 4.7 shows the spaceship movement, accelerating and decelerating in space.

4.6 Game loop

The main game loop is managed by the singleton class `GameManager()`, it updates the rotation of all planets even while the player isn't watching them and also the scale of the main star in the game, as if this star gets big enough to overlap with the player or the planet where the player has landed it results in the death of the main character and a game over state. All the start in the game orbit at equidistant circular orbits around the main star, so the star scale increases linearly at a rate calculated, giving the player an 20-minute time span between the destruction of each planet.

This class also manages, the oxygen consumption and regeneration coroutines, the health regeneration coroutines, combat, scene loading and game resets. These implementations are rather simple as they only update values if certain criteria is met or overtime, the design of these systems can be found in sections 4.3 for combat, 2.6.2 for the oxygen system and 2.6.2 for the health system.

4.7 Interface

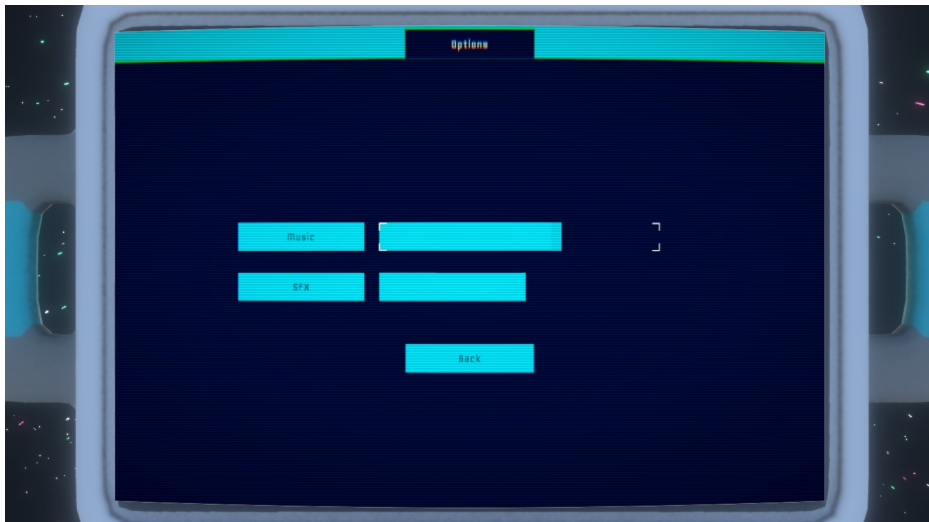


Figure 4.8: Options tablet screen

The main interface of the game is the tablet carried by the main character, in this screen, the user can buy all the upgrades needed, check its inventory and access the pause and main menus. This interface is controlled by the singleton class `HUDManager()` in charge of keeping the interface updated with the actions of the player and loading or unloading the different elements necessary to navigate the game menus.



Figure 4.9: Game world HUD

Each time, the player accesses the games interface, the tablet flies in front of the screen to make it look like the player is actually using the screen. This interface is implemented thought a screen space camera canvas to allow the use of camera effects on the tablet screen rendering, this will have an in depth explanation in section 4.9. It can be navigated by both clicking on the elements and using a game pad. In addition to this main interface, some HUD elements are rendered in the game world to show interaction hints to the player, these are controlled by the `PlayerController()` class as it was easier to manage them in that way. Figures 4.8 and 4.9 show two examples of the table screen interface and the game world HUD just explained, another tablet screen menu can be found in Figure 4.6.

4.8 Sound system

The sound system of the game uses a similar implementation as the other general systems of the game, like the interface or the game loop management. It has been implemented through a singleton class `SoundManager()` that is accessible from any script of the game and ensures that only one instance of it is being loaded. This allows for seamless sound transitions between scenes and avoids sound error like music overlapping.

The class integrates dictionaries made with enums that link each enum key to an audio clip, making it extremely easy to make the calls for a sound to play. It also creates the audio source components necessary at any time to ensure that enough sources are loaded while not saturating the system. The final version of the game implements 13 different sound effects that can be played at any time and 7 music tracks that are played on different scenes of the game. These audios were taken from free sources under the Creative Commons 0 license to avoid copyright infringements.

4.9 2D art

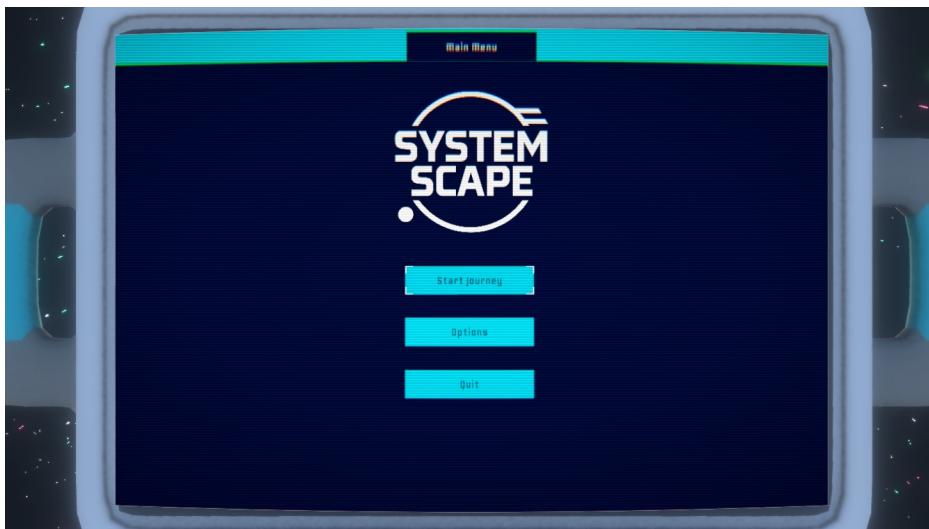
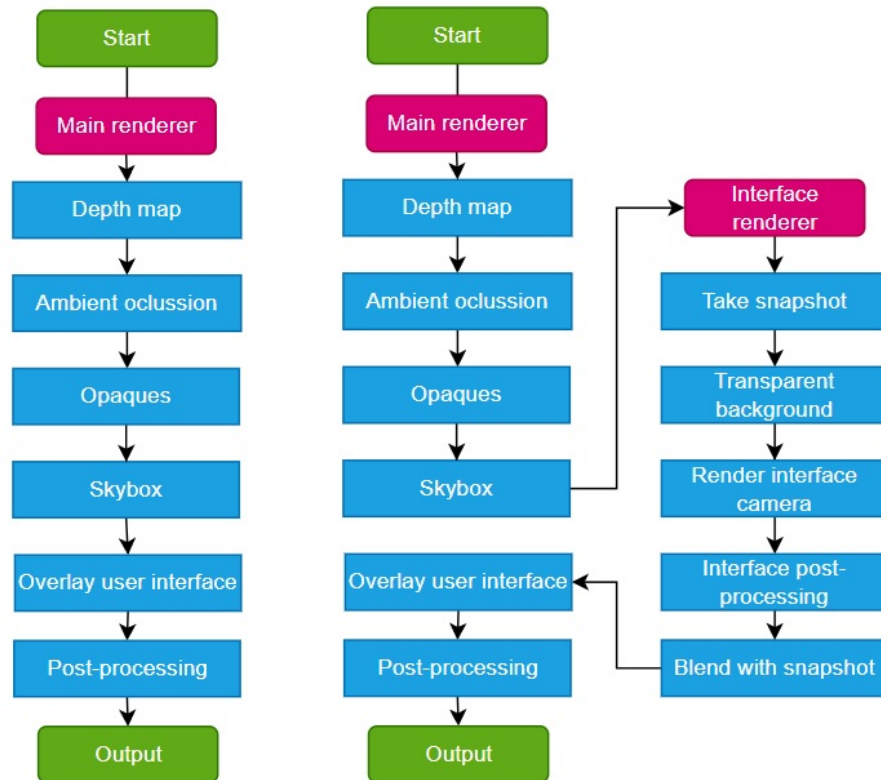


Figure 4.10: Main menu with CRT filter

The 2D art implemented in the game is rather limited, as all elements in the game except from the interface are 3D elements integrated directly into the game world. Figures 4.8, 4.6 and 4.10 show the aesthetic chosen for the interface and 2D elements of the game, resembling the old era of computers, with simple buttons and limited colors. This aesthetic is intentional as the mechanisms the user will be interacting with in the game, are from a space corporation so they are meant to be easy to use and functional, not beautiful.

To enhance this old computer aesthetic, a CRT effect has been added. To achieve this effect, a post-processing volume has been used for the game interface elements. This has been done by using a separate camera that only renders the game's interface and applies the post-processing to it. This theoretically simple task required the use of different URP renderers, as by default Unity applies any post-processing of the camera stack to the whole image, not only what one of the cameras render. This is due to the way the render pipeline works, as post-processing is applied after all elements in the game

are rendered, independently of what camera is supposed to be affected by the post-processing. This custom render for the interface renders only what the interface camera sees, applies the post-processing as that camera is done rendering everything on it and then inserts the resulting rasterized image on top of the final main camera render, using alpha clipping to merge these two renders. Figure 4.11 shows a diagram of the process.



(a) Original render pipeline

(b) Modified render pipeline

Figure 4.11: Custom render features diagram

4.10 3D art

All the 3D elements of the game have been modeled and texturized using Blender. The game 3D assets are composed by 60 tiles, 12 for each planet, 10 materials, one spaceship, one main character model, one pad model, a rifle and a drill. In addition, particle systems and a dome have been created to encapsulate the explorable area of each planet with custom materials that resemble a force field and custom materials for both the star and the planets and the space sky box have been created. Figures 4.12, 4.13 and 4.14 show the assets used in the game, the solar system with custom procedural planet

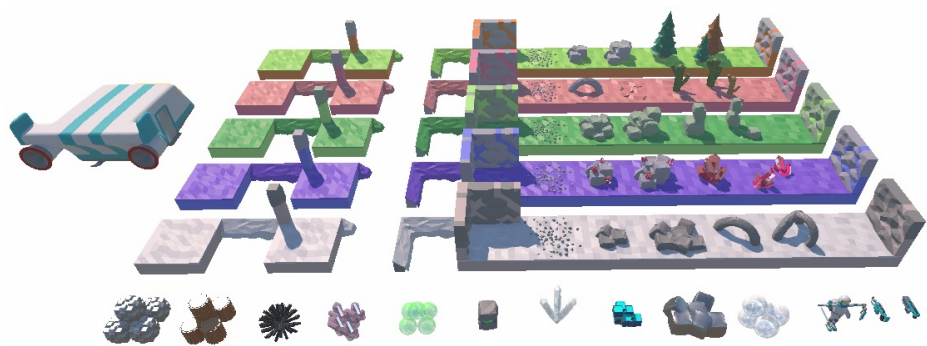


Figure 4.12: Game assets

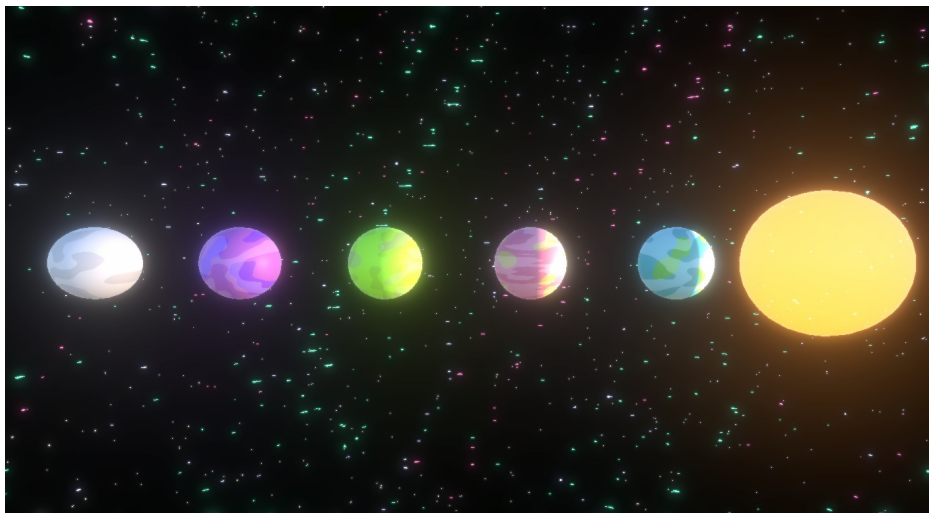


Figure 4.13: Space scene assets

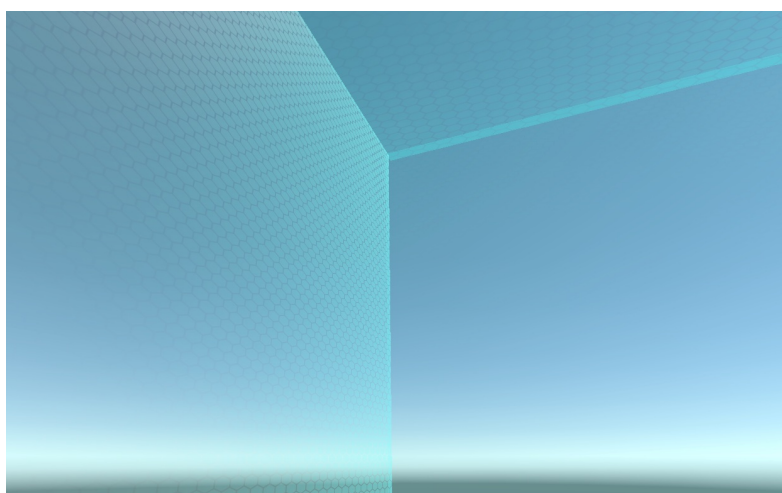


Figure 4.14: Dome asset

materials and the dome being used in the scenes. In addition to those assets, animations have been implemented using mixamo for their generation, having 2 different animation trees, one for the sword and one for the rifle and drill which include idle, running, jumping, attacking and getting hit animations.

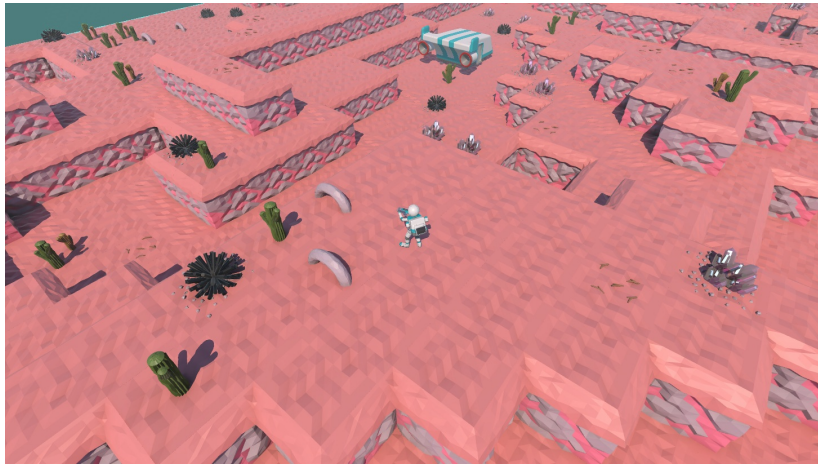
4.11 Results

Even though all enemies that were planned haven't been implemented due to complications in the development process, the game has resulted in a satisfactory experience that can be enjoyed by the player at the same time that it cover the main objective of the game, showcasing the WFC generation tool capabilities. The different planets with different tileset and sizes showcase the flexibility of the tool and its use in a real project, in addition, the lack of time while developing the game has further showcased the convenience of the tool as without it, maps of this size and quality wouldn't have been possible due to time limitations. Figures 4.15 and 4.16 showcase the different planets made for the game, showing the final results of both the tool use and the final game.

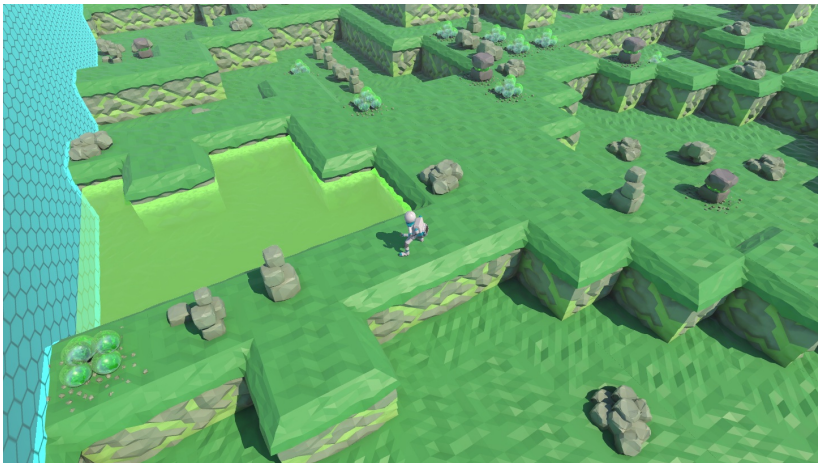


(a) Mercum planet

Figure 4.15: Final game results 1



(a) Colis planet



(b) Phobos planet



(c) Regio planet

Figure 4.16: Final game results 2



(a) Platium planet

Figure 4.17: Final game results 3

Conclusions and future work

Índice

5.1	Conclusions	65
5.2	Future work	66

This chapter shows the conclusions obtained from this project and the future work planned for it.

5.1 Conclusions

The main objective of this project was to create a tool that allows users to generate procedural worlds using the parallelized version of the Wave Function Collapse algorithm easily without having to worry about its implementation. The tool has achieved this goal, offering the user a nicely integrated interface within the editor that allows him to create as many tilesets as needed, modify all the socket types to their will and all of that while having a visual representation that helps them identify which parameters they are modifying.

The tool has also proved to have significantly higher performance than the usual implementations, thanks to taking the work load from the CPU and translating it to the GPU to take advantage of their parallel capabilities. Even though the parallel approach ended up having its inconvenient due to the way the algorithm works, a solution was found that, under a limitation on the map size, works as expected.

On the other hand, the problems encountered along the way of creating the parallel approach led to the creation of the chunk mode. This mode allows the user to generate larger maps with a linear temporal cost. Exploring this method, helped to better comprehend the problems behind the Wave Function Collapse algorithm and how solve them, but at the same time, this implementation wasn't expected in the original planning so it delayed the development of the game, resulting on its lack of the original planned enemies.

Speaking about the game, System Scape has achieved the goal of being an enjoyable experience while showcasing the power of the tool. Its variety of planets along with different biomes and map sizes demonstrate the power of the tool while providing an interesting and varied exploration experience to the players, challenges them to escape the astral system in time.

In general, the project has come to a satisfactory result for both the Unity tool and the System Scape game.

5.2 Future work

As explained above in section 5.1 we considered that the tool is at a great state and really useful for user that need procedural generation of levels for the creation of their games, but there's still room for improvement, reason why the tool hasn't been published on the asset store yet.

The main line of future work on the tool will be as follows:

- Develop a parallel version that works only on CPU side for users that prefer it.
- Optimize the compute shaders to allow for a bigger tile count than 50.
- Write a full documentation PDF to add to the tool package.
- Publish the tool in the Unity's Asset Store.

As for the game System Scape, the only future work left is to implement the enemies originally planned for the game, as they will add a nice touch of difficulty to the game, incentivizing the player to rush their way through planets.

Bibliography

- [Bra21] B. T. Brave (2021). *Infinite Modifying In Blocks*. Accessed: 2025-06-14.
URL <https://www.boristhebrave.com/2021/11/08/infinite-modifying-in-blocks/>
- [Chr24] R. Christie, B. Kitchen, W. Tumilowicz, S. Hooper y B. C. Wünsche (2024). *Procedurally Generating Large Synthetic Worlds: Chunked Hierarchical Wave Function Collapse*. En *2024 39th International Conference on Image and Vision Computing New Zealand (IVCNZ)*, págs. 1–6. IEEE.
- [Gum16] M. Gumin (2016). *Wave Function Collapse Algorithm*. <https://github.com/mxgmn/WaveFunctionCollapse>. Accessed January 2025.
URL <https://github.com/mxgmn/WaveFunctionCollapse>
- [Kim19] H. Kim, S. Lee, H. Lee, T. Hahn y S. Kang (2019). *Automatic generation of game content using a graph-based wave function collapse algorithm*. En *2019 IEEE Conference on Games (CoG)*, págs. 1–4. IEEE.
- [Lóp25] M. V. López y M. Chover (2025). *Procedural Generation of 3D Maps with Wave Function Collapse: Optimization and Advanced Constraints*. En *Proceedings of Eurographics 2025*. The Eurographics Association.
URL <https://diglib.eg.org/server/api/core/bitstreams/28237711-7df0-49ef-a6d2-b65aa163bb75/content>
- [Mar19] Marian42 (2019). *Infinite procedurally generated city with the Wave Function Collapse algorithm*. <https://marian42.de/article/wfc/>. Accessed January 2025.
- [Mer07] P. Merrell (2007). *Example-Based Model Synthesis*. En *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07)*, págs. 105–112. ACM, New York, NY, USA.
URL https://paulmerrell.org/wp-content/uploads/2022/03/model_synthesis.pdf

- [Odd22] OddMax (2022). *GitHub - oddmax/unity-wave-function-collapse-3d: Implementation of wave function collapse approach for Unity in 3d space.* <https://github.com/oddmax/unity-wave-function-collapse-3d>. Accessed January 2025.
- [San19] A. Sandhu, Z. Chen y J. McCoy (2019). *Enhancing wave function collapse with design-level constraints.* En *Proceedings of the 14th International Conference on the Foundations of Digital Games*, págs. 1–9.
- [Stå18] O. Stålberg y R. F. Games (2018). *EPC2018 - Oskar Stålberg - Wave Function Collapse in Bad North - YouTube.* <https://www.youtube.com/watch?v=QxzG4fyt7TU>. Accessed January 2025.
- [Stå22] O. Stålberg y Konsoll (2022). *Konsoll 2021: Oskar Stålberg - The Story of Townscaper - YouTube.* <https://www.youtube.com/watch?v=6fQ2pIKN5zI>. Accessed January 2025.
- [Tri] W. Tristan. *Using Wave Function Collapse algorithm for 2D and 3D level generation.* <https://tristan-w.github.io/WFC-2D-3D>. Accessed January 2025.

CHAPTER 6

Project repositories

Here are the links to the repositories of both the Wave Function Collapse Tool and the video game System Scape.

[Wave Function Collapse Unity Tool](#)

[System Scape videogame](#)

