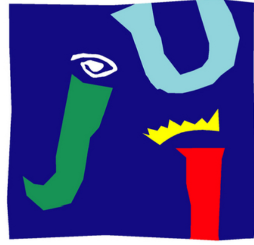


Guardians of Aeon

Development of a 2D video game combining platformer, shooter and RPG elements in Unity.

Final Degree Work Report

Gerard Roig Cléries

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 12, 2024

Supervised by: Emilo Bueso Aparici

To my father, my mother, my sisters and my brother.

Thanks to their support, I have been able to pursue my career.

Acknowledgments

First of all, I would like to thank my mentor, Emilio Sáez Soro, for providing us with an experienced and pragmatic perspective in game development.

Thanks to my final degree colleague, Miguel Ángel Álvarez Torres, for working just as hard as I have, propelling the project forward.

I also would like to thank Ada for helping me achieve my best and keeping me motivated.

And last but not least, thanks to Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

This document serves as Gerard Roig Cléries' Final Degree Work report in Video Game Design and Development.

This project involves designing, developing, and presenting a 2D video game that combines the platformer, shooter, and RPG genres using the Unity game development engine. The goal is to create a fully functional video game that seamlessly integrates the mechanics and features of these three genres into an engaging gaming experience.

Keywords

Final Degree Work, video game, 2D, Guardians of Aeon.

Contents

Contents **v**

1 Introduction **1**
1.1 Work Motivation . . . 1
1.2 Objectives . . . 2
1.3 Environment and Initial State . . . 2

2 Planning and resources evaluation **3**
2.1 Planning . . . 3
2.2 Resource Evaluation . . . 3

3 System Analysis and Design **7**
3.1 Game Concept . . . 7
3.2 Game Design . . . 8
3.3 Art Design . . . 10
3.4 Interface Design . . . 13
3.5 Sound Design . . . 15

4 Work Development and Results **17**
4.1 Work Development . . . 17
4.2 Scene Manager . . . 22
4.3 Results . . . 23

5 Conclusions and Future Work **25**
5.1 Conclusions . . . 25
5.2 Future work . . . 25

Bibliography **27**

A Source code **29**

Chapter 1

Introduction

Contents

1.1 Work Motivation 1
1.2 Objectives 2
1.3 Environment and Initial State 2

Here is an exposition outlining the initial purpose of our project, shedding light on the reasons behind our project choice and the original plans for its development. It is worth mentioning that this is a shared project with another individual.

1.1 Work Motivation

In deciding on our final degree project, Miguel Ángel Álvarez Torres and I wanted something that not only pushed our skills but also reflected our interests. We settled on Guardians of Aeon for several reasons.

Firstly, our collaboration worked well because Miguel Ángel's strengths in storytelling, art and level design matched my programming abilities and visual effects knowledge in Unity. Secondly, we wanted to prove ourselves that we were able to create a game we could be proud of.

We have worked together in several game jams in the past and we knew we were capable of doing well in a project of this magnitude. We wanted to showcast the expertise in Unity acquired throughout our four-year academic journey.

Furthermore, this joint project was mainly fueled by our shared love for diverse gaming genres, each leaving its mark on our future game. We found inspiration in titles like "Hollow Knight" [6] for its stunning art style and unique progression mechanics, "Blasphemous 2" [10] for its RPG-level design, and many others such as "Brotato" [1] for its dynamic shooter elements.

1.2 Objectives

1. Unique Art Style: We knew that a cohesive and stunning visual is what separates the good from the best games, so we decided to create the art and the visual effects from scratch. Most of the time, I personally focused on creating believable shaders for wind, glow and other effects, such as particles for plants.
2. Mechanics Progression System Across Levels: In the Forest, players learn basic mechanics like movement and wall jumping while encountering initial enemies. Transitioning to the Valley, they continue to face foes and navigate platforms with an added feature: the dash. Then, in the Chronen Archives the difficulty increases and finally, the Bridge serves as a test of all acquired skills, mixing platforming and combat mechanics.
3. Enemy AI: We wanted to create a game with at least seven different enemies with several behaviours. These include way-point movement and jumping for patrolling, and attacking, shooting or following player when detected. So a simple state machine was needed for the enemy behaviour.

1.3 Environment and Initial State

This project was a collaborative effort between Miguel Ángel Álvarez Torres and myself, where we alternated workdays. Coordination was facilitated by our proximity, as we live in the same apartment, making it easy to maintain communication. Using tools like Github for project management, we followed a structured workflow. Additionally, Discord served as our secondary communication platform if face-to-face was not possible, allowing for daily meetings. Our work routine was uneven, but we made sure to reach the expected work hours each week. Furthermore, biweekly sessions with our project supervisor provided valuable feedback and direction.

During the initial phase, my contributions revolved around the player including tasks such as implementing basic movement mechanics, wall jumping, a dash ability, camera fluidity and projectile shooting.

CHAPTER 2

Planning and resources evaluation

Contents

2.1 Planning 3
2.2 Resource Evaluation 3

This section goes into the technical aspects of the project, outlining the planning process and detailing the software and resources utilized.

2.1 Planning

In this section, the breakdown of time allocation for the project is based on various tasks. The Gantt chart (see Figure 2.1) visually represents the timeline for these tasks. The majority of the time is dedicated to technical development within Unity, involving tasks such as configuring the Unity environment, implementing movement and combat mechanics, and integrating elements into Unity. Other essential tasks include documentation, conceptualization and game design.

2.2 Resource Evaluation

Here, I am listing the hardware and software I have been using for the project. Although not all of it has been absolutely necessary, it has been helpful in keeping things running smoothly.

Task name	Start date	Finish date	Status	Approximate Hours Employed	December	January	February	March	April	May	June
Final Degree Project Documentation	**20.01.2024**		**In progress**	**50h about**							
Technical Proposal	20.01.2024	23.01.2024	Close	2h about							
GDD	26.02.2024	28.02.2024	Close	6h about							
Biweekly Reports	26.01.2024	31.05.2024	In progress	12h about							
Powerpoint	15.05.2024	03.06.2024	In progress	5h about							
Project Memory	01.05.2024	15.05.2024	Close	25h about							
Conceptualization and Design	**16.12.2024**		**In progress**	**25h about**							
Background Research and Concept	02.02.2024	05.03.2024	Close	5h about							
Game mechanics and systems	16.12.2023	21.03.2024	Close	10h about							
Level and scenario Design	26.02.2024	19.03.2024	Close	10h about							
Technical Development	**19.01.2024**		**In progress**	**115h about**							
Configuration of Unity Environment	19.01.2024	27.01.2024	Close	5h about							
Implementation of Movement Mechanics	19.01.2024	12.02.2024	Close	20h about							
Implementation of Combat Mechanics	12.02.2024	21.03.2024	Close	25h about							
Inventory System	26.02.2024	15.05.2024	Close	15h about							
Mission and Dialogue Systems	01.05.2024	15.05.2024	Close	10h about							
AI for Enemies and NPCs	02.03.2024	29.04.2024	Close	25h about							
Level and Scenario Implementation	13.03.2024	01.05.2024	Close	15h about							
Art Development	**26.12.2023**		**In progress**	**70h about**							
Environment Background Final Design	28.01.2024	15.05.2024	Close	30h about							
Creation of Assets and Animations	01.02.2024	21.03.2024	Close	10h about							
Design HUD and Menus	29.04.2024	15.05.2024	Close	15h about							
Design Visual Effects and Particles	26.02.2024	01.05.2024	Close	15h about							
Integration and Optimization	**04.02.2024**		**In progress**	**40h about**							
Integration of elements in Unity	30.01.2024	15.05.2024	Close	10h about							
Bug Testing and Fixing	13.02.2024	15.05.2024	Close	15h about							
Adding Details and Additional Content	29.04.2024	15.05.2024	Close	15h about							
			Total hours:	**300h about**							

Figure 2.1: My Gantt chart (made with Excel)

- **Hardware**: I have used my Asus Rog Strix 15 laptop for everything, which is composed of:
  - **CPU**: Intel Core i7-10750H
  - **GPU**: Nvidia RTX 2070 Mobile
  - **RAM**: 16 GB
- **Software**: All the software I have used is completely free.
  - **Unity** [16]: I used Unity as the main platform for developing our game. It provided all the necessary tools for creating and testing the project.
  - **Rider** (JetBrains IDE) [13]: For coding and scripting tasks, I relied on Rider, which is an IDE (Integrated Development Environment) developed by JetBrains. I chose it for its powerful features for writing code in Unity.
  - **GitHub** [9]: My colleague and I used GitHub for version control and collaboration. It allowed us to track changes and coordinate our work effectively.
  - **Overleaf** [12]: It is a great tool to edit LaTeX Templates and add lots of functionality to a formal document without downloading any program.
  - **Google Docs** [11]: We used it to draft less formal documents because it is flexible and offers real-time collaboration features.
  - **Krita** [7]: When it came to creating and refining visual assets like sprites and background images, I relied on Krita.
  - **Stable Diffusion** [15]: Stable Diffusion served as an image AI tool that I utilized for a specific task in our project, the creation of plant assets. After experimenting with various AI models and refining the selection, I ended up with a set of optimal sprite sheets for integration into Unity's sprite editor. This process involved tasks in Krita such as color correction, removing background elements, and organizing the assets for seamless integration into our game environment.

Chapter 3

System Analysis and Design

Contents

3.1 Game Concept 7
3.2 Game Design 8
3.3 Art Design 10
3.4 Interface Design 13
3.5 Sound Design 15

In this section, the various aspects of the game in which I have been involved are discussed, including its concept, game design, art design and interface design. The reasoning behind each decision and its impact on the game is also explained.

3.1 Game Concept

Guardians of Aeon follows the journey of the first space traveler from a planet called Caelum to Aeon, another planet in the same solar system. Aeon holds the ruins of an ancient civilization called the Chronen, who vanished 5,000 years ago.

The Chronen were humanoids with special abilities. They lived for a very long time and had a deep connection to nature and time itself. They could even control time to their advantage, which allowed them to build amazing things like floating cities and advanced technology. But there was a catch.

The more the Chronen messed with time, the worse things got for the entire universe.

It became clear that their powers were causing a big problem. To prevent a cosmic disaster, the Chronen had to give up their abilities. However, this caused their cities to crumble, their planet suffered, and most Chronen died.

Surprisingly, the Chronen were not completely gone. They gave up their physical forms and became guardians who could still watch over Aeon. Now their goal is to ensure that nobody ever messes with time again and repeats their mistakes.

3.1.1 Genres

Guardians of Aeon is a game that blends elements from three distinct video game genres: platformers, RPGs (role-playing games), and shooters.

- Platformer elements: The game utilizes two-dimensional environments, requiring players to navigate them by jumping between platforms and overcoming obstacles
- RPG elements: Players interact with an NPC (non-playable character) for information and they are encouraged to uncover hidden knowledge by exploring the world.
- Shooter elements: Players engage in battles against enemies using various weapons and abilities to overcome them, adding a layer of challenge to the game.

3.2 Game Design

3.2.1 Mechanics and Game Feel

Being the main programmer, the majority of the mechanics have been implemented and tweaked. Gradually the main focus shifted into smoothing the gameplay experience and giving an organic feel, so I decided to develop all character movement script with physics. This translates into applying forces to do any kind of movement instead of changing its position directly, which could be made in a 2D game but it did not feel right.

- Character Movement: To implement an only physics-based approach, a force is applied with a certain acceleration when a movement input is pressed. It is important to limit the maximum speed so that it cannot snowball.
- Character Jump: A vertical force is applied with a bigger acceleration when two conditions are met, the character is on the floor[1] and the jump input is pressed.

[1] In Unity, I chose to do a box ray-cast to know if the player is in contact with its adjacent walls.

- Character Wall Jump: An opposite force to the wall and a vertical force are applied when two conditions are met, the character is on the wall and the jump input is pressed. Then, the input from the player is disabled for a few frames to give time to detach from the wall.

- Dash: A force in the direction of the movement is applied when the dash input is pressed. There is a small cool-down with this mechanic.

The other non-movement related mechanics are:

- Shoot: There are several weapons (see table 3.2.3) which the player can shoot with. Instead of reloading the weapons there is an over-heat system implemented. When the weapon is fully heated, it needs to cool down for a couple of seconds.

- Heal: There is a health system for the player and the enemies. When the player takes damage it can take a potion to heal himself and fill the health-bar. A Brackeys tutorial [4] was followed.

- Inventory: It is used to keep track of the weapons (see table 3.2.3) and potions the player has. It is used to change weapons or use health potions.

- Interact: This input can be used near the NPC to interact with him.

3.2.2 Level Design

The adventure begins in a **forest**. Players learn the core mechanics: movement, jumping, interacting, climbing walls and shooting. Here, they will face their first enemies and encounter the a friendly NPC, a guide who offers assistance and gives context.

Progressing through a **valley**, players have to prove their combat skills against enemies while also navigating platforming sections. This area marks the introduction of the dash ability, adding a new dimension to movement.

The journey leads to the Chronen Archives, an ancient **library** contained in a mountain. Here, the guardians of the Chronen civilization are protecting their knowledge. The players must again test their skills as they navigate new platforms, all while avoiding projectiles and shooting enemies.

Finally, the game ends in a dangerous **bridge** where all the learned experience is applied with increased difficulty.

3.2.3 Weapons

Weapon	Type	Damage (per bullet)	Fire Rate	Overheating	Range
Sting	Gun	10	300	Low	Short
Elemental	Gun	30	150	Medium	Medium
Firelight	Shotgun	60	35	High	Short
Drowner	Machine Gun	28	600	Very High	Medium
Stronghold	Rifle	70	90	Medium	Long
Frigid	Bow	100	60	Low	Medium

3.2.4 Enemies

Every enemy has a unique AI which causes them to behave differently. Some of them have a basic state machines with two states, patrol and attack.

- Minion: Patrols between two way-points. When the player is close enough, follows the player and attacks him.
- Jumper: Similar to the minion, it has a patrol state and an attack state. However, it moves by jumping, so it can avoid obstacles.
- Octopus: It is a flying creature that follows the player using the A* algorithm. I followed a Brackeys tutorial [2], and after downloading a free resource, I applied this path-finding script to all levels.
- Scorpion: A static creature that shoots venom projectiles with gravity to the player. The initial angle of the projectile is calculated using physics.
- Jellyfish: A peaceful creature that only hurts the player when touched. It also follows a fixed path.
- Radar: An enemy radar that disintegrates the player when scanned.
- Ghost: An enemy that goes through walls and deals damage to the player when touched.

3.3 Art Design

In our project, the 2D art design of the game was primarily accomplished using Krita, a powerful digital painting software. We also used Krita for our environment backgrounds.

On the one hand, the tile palettes were designed and then imported to Unity through sprite-sheets. I created two tile-map grids for two different sizes, one for the general sized tiles and a smaller one for the platform creation. Then, manually adjusted each

tile collider to synchronize it to its visual.

On the other hand, all the animations were carefully programmed and revised to show what the player and the enemies are doing all the time.

I have also created a total number of 176 different plants using Stable Diffusion AI. First, I experimented with several image generation AI models to determine which produced the best results. Then, approximately 50 images were generated to find the most suitable ones. I selected the top 10 images and edited them using Krita:

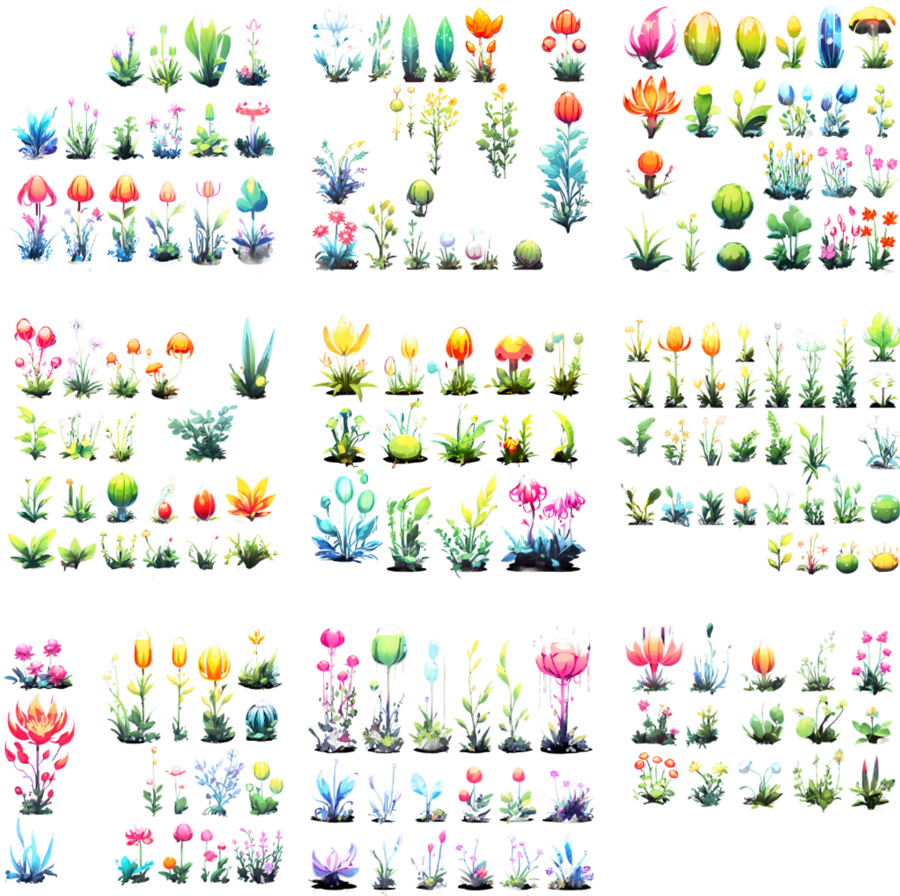


Figure 3.1: Optimal sprite sheets edited in Krita

This is the edition in Krita step-by-step:

- Corrected the color of the plants and removed any malformed ones.
- The majority of the time was spent deleting and retouching all the plants.
- Used the selection tool to remove the background.
- Organized them for use in Unity's sprite editor.
- Crafted plant assets from AI-generated images, refining and curating a selection of 10 optimal sprite sheets in Krita for integration into Unity's sprite editor (see Figure 3.1). Each has between 10 and 20 plants.

In order to create a material and assign a shader to each plant, I had to implement a custom Editor script for Unity to allow the division of all plants into individual sprites.

Once I had 176 plants I classified them into colors and shapes. The main reason was to use different plants for different level themes:

- Forest: green
- Valley: orange and yellow
- Library: red, purple and pink
- Bridge: blue

3.3.1 Shaders and particles

Custom shaders were crafted in order to simulate natural phenomena and enhance visual aesthetics, including wind simulation for plant movements and glow effects. These last ones were used in multiple particle effects and in character and enemies' eyes.

- Wind Shader: A significant focus was placed on developing a dynamic wind shader to simulate realistic plant movements. By using the Shader Graph editor, I customized parameters such as wind intensity, direction and speed to evoke lifelike movements. This idea was taken from a Sasquatch B Studios tutorial [14].
- Glow Shader: Parallel to the wind shader, I dedicated efforts to implementing a glow shader designed to enhance specific visual elements, such as petals and eyes. Extensive research into glow shader methodologies guided my approach. This shader was carefully calibrated to impart luminosity to selected elements, making them stand out. For the eyes, a sprite-sheet mask was needed, this last concept was developed by following a Brackeys tutorial [3].

3.4 Interface Design

3.4.1 Visual System

First of all, to improve the overall organic feel of the character camera some adjustments were made by:

- Implementing Camera Smoothness: Unity built-in algorithms were used to ensure smooth camera movements, reducing jarring transitions.
- Adjusting Camera Parameters: Fine-tuned various camera settings such as field of view, depth, and distance from the player character.
- Limiting Camera Movement: To prevent the camera from straying beyond the game world boundaries, implementation of constraints or boundaries, restricting its movement to predefined areas.

Visually, the interface design is also formed by the Game User Interface (UI) in-game and the menus. The UI is composed of the character's health bar and a smaller bar for the weapon over-heat status. Each bar is edited with their own script. The player has access to these scripts and can modify their values.

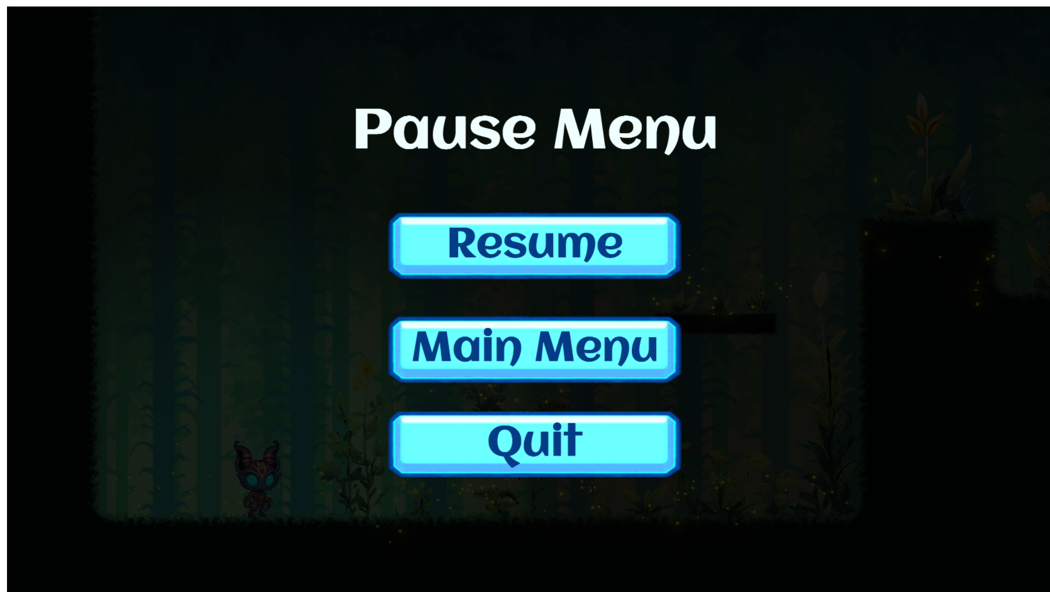


Figure 3.2: Game Pause Menu

On the contrary, the Inventory menu is a round wheel where the player is able to choose a weapon and heal.

There is also a main menu where the player can choose between the four levels, go to the settings or leave the game. Finally, there is a pause menu (see Figure 3.2), where the player is able to pause mid game, go back to the menu or leave the game.

3.4.2 Control System

In this version the game is only for PC, so all the inputs are keyboard and mouse related. These are the keybindings for each mechanic:

- Character Movement: A/left arrow is used to go left. D/right arrow is used to go right.
- Character Jump and Wall Jump: Space-bar.
- Dash: Left Shift.
- Shoot: Mouse Left Click/Left Control
- Inventory Access: Press Q and choose by moving the mouse from the wheel menu.
- Interact: E Key.

3.5 Sound Design

3.5.1 Audio Manager

I developed an Audio Manager, which handles all sound-related functionalities in the game. This includes managing sound effects, music, and ambience sounds.

I searched for copyright-free sounds on Freesound [8]. I found suitable sound files that match our game's requirements and can be freely used without legal issues.

3.5.2 Sound effects

I added 3D spatial sound effects for enemies that change based on the player's position relative to the enemies, providing audio cues that help players sense the direction and distance of threats.

I also added other sound effects for enemies, such as attacks, and sounds for button clicks, navigation, and other UI elements.

3.5.3 Music and Ambience

All the music was downloaded from the compositor Scott Buckley [5]. His music is licensed under the Creative Commons Attribution 4.0 law.

Chapter 4

Work Development and Results

Contents

4.1 Work Development **17**
4.2 Scene Manager **22**
4.3 Results **23**

All the work has been explained in general terms, so in this chapter the main focus of my work development will be discussed. It will go a little more into detail about specific topics and then the final results will be showed.

Most graphs and scripts have been written by me, so I have designed the organization with no coordination inconveniences. My main source of information has been Brackeys and Unity Documentation.

4.1 Work Development

4.1.1 Main Character

I will start explaining the most important components and scripts the main character has and for what reason:

- Character Movement Script:
  - Singleton Instantiation: The script includes a Singleton pattern implementation to ensure there is only one instance of this class, which provides a global point of access to the instance.

  - The Update method is used to handle player input and update the character's state, while the FixedUpdate method is responsible for executing the character's movement logic within the physics system.
  - Collision Detection: Methods such as isGrounded, isWallLeft, and isWallRight are implemented to detect collisions with the ground and walls, allowing the character to respond appropriately to environmental obstacles.
  - Gizmos Visualization: The OnDrawGizmos method is utilized to visualize the ray-casts used for collision detection in the Editor (see Figure 4.1).
  - A dash mechanic is implemented, allowing the character to perform a quick dash in the specified direction with a cool-down period between dashes.

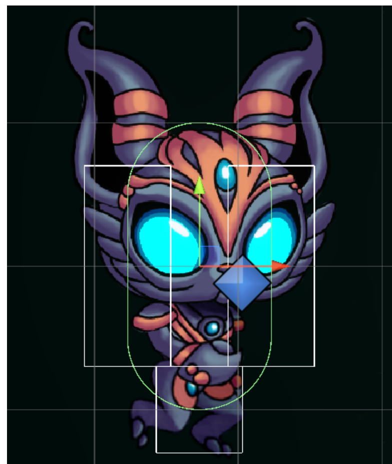


Figure 4.1: White Boxes are the ray-casts representation

- Character Shoot Script:
  - The CharShoot script serves as the central controller for the character's shooting mechanics in the game. Its primary function is to manage the firing of bullets based on player input while also overseeing the mechanisms associated with overheating, which impacts the character's ability to shoot rapidly.
  - Functionalities to regulate the rate of fire, ensuring that the character cannot shoot continuously without pause. It uses a co-routine to handle the cool-down period between successive shots.
  - Additionally, the CharShoot script introduces an overheating system, which imposes restrictions on shooting when the character's weapon becomes excessively heated. This feature adds a strategic element to gameplay, requiring players to manage their shooting frequency effectively to avoid overheating.

- Bullet Behavior Script:
  - On the other hand, the BulletBehavior script defines key attributes of the bullets, including their appearance, speed, and lifetime, which significantly impact the dynamics of the game's combat system.
  - Upon instantiation, the BulletBehavior script determines the visual characteristics of each bullet, such as its sprite and material, based on predefined parameters. This allows for visual variety and distinction among different types of bullets.
  - Moreover, the script assigns different behaviors to various types of bullets. For example, bullets may travel at different speeds or be subject to gravitational forces, influencing their trajectory and interactions with game elements.
  - Upon collision, appropriate actions are triggered, such as applying damage to enemy entities or initiating sound effects.

The Bullet Behavior Script was programmed in a way it could be re-used for every bullet type. When the weapon is swapped in the inventory, another bullet is chosen.

Once the health system and the overheat system was fully functional, I created a script to visually handle an HP bar and another one to handle the overheat bar, both part of the UI.

I also added a Laser Behavior Script in case we needed it. At its core, the script uses a LineRenderer component to render the laser beam, providing a visual representation of a projectile's path. The LineRenderer is configured to start at the position of the bullet spawn point.

This script performs a ray-cast along the direction of the laser beam to detect any obstacles or targets in its path. If the raycast intersects with an object, the destination point of the laser beam is adjusted to terminate at the point of intersection, ensuring that the laser beam is visually obstructed by solid objects in the environment.

4.1.2 Enemies

The enemies AI, as I mentioned before, have a unique script that contains all its behavior. However, all the enemies have a common Enemy script to deal damage to the player or receive damage.

The most different AIs are definitely from the octopus and the scorpion:

- Octopus: I wanted to create an enemy that followed the player and damaged him. The first version was simple, the octopus simply went in a straight line through walls. Then I realized that I needed some sort of algorithm to avoid obstacles, this is the moment I remembered the A* algorithm (see Figure 4.2). However, I knew that for only one enemy, my own A* implementation was not worth the huge investment of time. So instead, I followed a Brackeys A* tutorial [2].

- Scorpion: This enemy was supposed to shoot projectiles with gravity in a static way. Nevertheless, I challenged myself to shoot the character and update the launch angle each time the character moved. To achieve this I ended up using the physics equations of the parabolic movement[1].

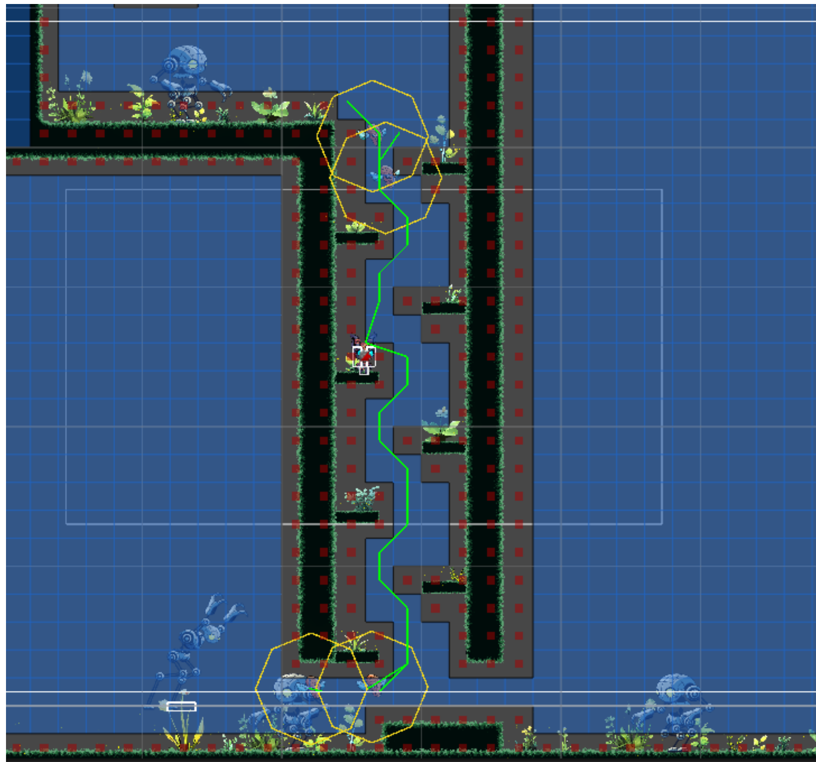


Figure 4.2: Octopus A* Algorithm

[1]
$$y = y_0 + v_{0_y} t - \frac{1}{2} g t^2 \tag{4.1}$$

4.1.3 Shading and Particles

Once all the plants were added to a scene it felt lifeless. The organic feel I was looking for did not match my expectations with a collection of static plants. That is when I stumbled across shaders and particles.

In order to add life to the game I wanted to create a wind shader effect to the plants and also add a jiggle effect when the player touched them. To achieve this I researched how to develop my own Wind Shader Graph in Unity (see Figure 4.5) and I found a comprehensible tutorial [14] explaining it in detail.

However, the game still needed something to pop even more, which is why I implemented a glow shader. This glow shader was applied to multiple types of particles and some bullets that I also developed. Then I played with its color settings to ambient each level in a different set of colors (see Figure 4.3).

Figure 4.3: Particle effect with pink glow shader

By recycling the glow shader I decided to give glow in some other places, like the main character and the enemy eyes (see Figure 4.4). This was done by applying a mask, which I had to manually craft, in Krita. After creating all the emission masks I added them to their respective Sprite Editor and added the glow material.

Figure 4.4: Eyes with yellow and blue glow.
Particle effects with yellow glow and venom projectile with green glow.

4.2 Scene Manager

4.2.1 Implementing Inventory

First, I developed an inventory system that allows players to easily access their weapons throughout the game at any time, allowing them to switch their weapons as needed. This system also keeps track of whether the dash ability has been obtained or not. The inventory is maintained after death.

4.2.2 Level Transition and Checkpoints

Next, I implemented the mechanics for level transitions. In other words, creating the logic that determines how players progress from one level to the next, using triggers such as reaching a certain point or interacting with an NPC. Incorporating checkpoints is necessary for saving player progress after death.

4.2.3 Resetting Enemies and Pause Logic

Finally, I included a mechanism to reset enemies, involving a system that respawns or resets enemies' states under certain conditions, such as when a player re-enters an area or dies. Additionally, I fixed some bugs in the enemy pathfinding system (specifically the A* algorithm) to ensure enemies reset their destination and any pre-existing path.

Implementing a pause functionality was also convenient to allow players to halt the game and resume where they left off, adding control over their play sessions.

4.3 Results

The project successfully achieved its core objectives outlined in the Introduction. A unique and cohesive art style was established through Krita and custom shaders. The 2D art assets, environments, and even the plant life contribute to a distinct aesthetic.

Moreover, there is a progression system effectively implemented. Players begin by grasping fundamental movement mechanics like jumping and wall-climbing. As they progress, the dash ability adds a new dimension to movement. Finally, the last levels challenge players by requiring mastery of all learned skills.

The enemy AI also offers a satisfactory level of variety and challenge. Enemies show diverse behaviors, ranging from basic patrolling and attacking to more complex pathfinding utilized by the octopus or the projectile-shooting behavior of the scorpion.

Figure 4.5: Wind Shader Graph

CHAPTER 5

Conclusions and Future Work

Contents

5.1 Conclusions **25**
5.2 Future work **25**

5.1 Conclusions

At the start of the project, my team mate and I had plenty of time to develop our tasks. These last days, on the other hand, we have had to organize ourselves diligently to balance the labor practices with the game development.

This project has been a significant learning opportunity, helping us improve our skills in programming, art design, and level design. Through communication and task delegation, we learned valuable lessons in project management, which are essential in game development. The resulting game reflects our collaborative efforts.

While we successfully achieved our initial goals, we also learned the importance of managing time and iteratively refining game difficulty.

5.2 Future work

There is plenty of room for growth in this project. One way to go is expanding the game's world with new levels and obstacles, opening up the chance to introduce different types

of enemies and maybe even a big boss battle. We could also dive deeper into enemy AI, making them smarter and more challenging to deal with.

Another path to explore is bringing the game to mobile platforms, which could help it reach a wider audience. But the skills I picked up while creating the game's art style, such as shader development, will definitely come in handy for future projects.

Bibliography

[1] Blobfish. Brotato. https://store.steampowered.com/app/1942280/Brotato. Accessed: 2023-12-16.

[2] Brackeys. A* implementation from a resource. https://youtu.be/jvtFUfJ6CP8. Accessed: 2024-04-03.

[3] Brackeys. Glow shader. https://youtu.be/WiDVoj5VQ4c. Accessed: 2024-03-21.

[4] Brackeys. Health bar tutorial. https://youtu.be/BLfNP4Sc_iA. Accessed: 2024-02-24.

[5] Scott Buckley. Free music. https://www.scottbuckley.com.au/tag/free-music/. Accessed: 2024-05-10.

[6] Team Cherry. Hollow knight. https://www.hollowknight.com. Accessed: 2023-12-16.

[7] Krita Foundation. Krita. https://krita.org/. Accessed: 2024-12-16.

[8] Freesound. Free sfx. https://freesound.org/. Accessed: 2024-05-04.

[9] Inc. GitHub. Github. https://github.com/. Accessed: 2024-01-29.

[10] The Game Kitchen. Blasphemous 2. https://www.blasphemous2game.com. Accessed: 2023-12-16.

[11] Google LLC. Google docs. https://www.google.com/docs/about/. Accessed: 2023-12-16.

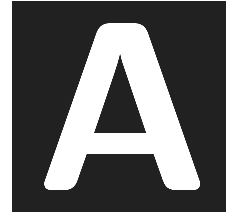
[12] Overleaf. https://www.overleaf.com/. Accessed: 2024-05-01.

[13] JetBrains s.r.o. Jetbrains rider. https://www.jetbrains.com/lp/dotnet-unity/. Accessed: 2024-01-19.

[14] Sasquatch B Studios. Wind shader. https://youtu.be/Ctbqax1XRiE. Accessed: 2024-03-15.

[15] CompVis Team. Stable diffusion. https://github.com/CompVis/stable-diffusion. Accessed: 2024-03-09.

[16] Unity Technologies. Unity. https://unity.com/. Accessed: 2024-01-19.

APPENDIX 

Source code

Character Movement

```
1
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public class CharMovement : MonoBehaviour
7  {
8      //Singleton instantiation
9      private static CharMovement instance;
10     public static CharMovement Instance
11     {
12         get
13         {
14             if (instance == null) instance = GameObject.FindObjectOfType<CharMovement>();
15             return instance;
16         }
17     }
18
19     private Rigidbody2D rb;
20     private Vector2 input;
21     private Animator animator;
22
23     [SerializeField] private float moveMaxSpeed;
24     [SerializeField] private float acceleration;
25
26     [SerializeField] private float jumpForce;
27     [SerializeField] private bool isJumping;
28     [SerializeField] private Vector2 jumpCastSize;
29     [SerializeField] private float jumpCastDistance;
```

```
30      [SerializeField] private Vector2 wallCastSize;
31      [SerializeField] private float wallCastDistance;
32      [SerializeField] private LayerMask groundLayer;
33      [SerializeField] private Collider2D dashCollider;
34      [SerializeField] private Collider2D playerCollider;
35
36      private float jumpTimer = 0f;
37      private float jumpTimerLimit = 0.2f;
38      private bool isWallJumping = false;
39
40      [SerializeField] private float dashSpeed;
41      [SerializeField] private float dashCooldown;
42      private float dashTimer = 0f;
43      private float dashTimerLimit = 0.1f;
44      private bool isDashing = false;
45      private Vector2 dashDirection;
46      private float dashCounter = float.PositiveInfinity;
47
48      public bool isPlayerRight = true;
49
50      private void Start()
51      {
52          rb = GetComponent<Rigidbody2D>();
53      }
54
55      private void Update()
56      {
57          input.x = Input.GetAxisRaw("Horizontal");
58          input.y = Input.GetAxisRaw("Vertical");
59
60          //Jump
61          if (jumpTimer < jumpTimerLimit)
62              jumpTimer += Time.deltaTime;
63          else
64          {
65              isWallJumping = false;
66          }
67
68          if (Input.GetButtonDown("Jump") && jumpTimer >= jumpTimerLimit)
69          {
70              jumpTimer = 0f;
71              if (isGrounded())
72              {
73                  Jump(0);
74              }
75              else if (isWallLeft())
76              {
77                  Jump(1);
78              }
79              else if (isWallRight())
80              {
81                  Jump(-1);
82              }
83          }
```

```
        //Dash
        if (dashTimer < dashTimerLimit)
            dashTimer += Time.deltaTime;
        else
        {
            playerCollider.enabled = true;
            dashCollider.enabled = false;
            isDashing = false;
        }

        if (dashCounter < dashCooldown)
            dashCounter += Time.deltaTime;

        if (Input.GetKeyDown(KeyCode.LeftShift) && dashCounter >= dashCooldown)
        {
            dashCounter = 0f;
            Dash();
        }

        //Check if going right
        if (rb.velocity.x > 0.1f)
        {
            isPlayerRight = true;
        }
        else if (rb.velocity.x < -0.1f)
        {
            isPlayerRight = false;
        }
    }

    private void FixedUpdate()
    {
        Move();
    }

    private void Move()
    {
        float speed = input.x * moveMaxSpeed;

        //Apply Force to Character
        float movement = (speed - rb.velocity.x) * acceleration;

        if (!isWallJumping && !isDashing)
        {
            rb.AddForce(movement * Vector2.right, ForceMode2D.Force);
        }
        else if (isDashing)
        {
            rb.velocity = new Vector2(dashDirection.x * dashSpeed, dashDirection.y * dashSpeed * 0.5f);
        }
    }


```

```
    private void Jump(int wall)
    {
        float force = jumpForce;
        if (rb.velocity.y < 0)
            force -= rb.velocity.y;

        if (wall == 0)
        {
            rb.AddForce(Vector2.up * force, ForceMode2D.Impulse);
        }
        else
        {
            isWallJumping = true;
            rb.AddForce(Vector2.up * force * 0.75f, ForceMode2D.Impulse);
            rb.AddForce(Vector2.right * jumpForce * wall * 0.75f, ForceMode2D.Impulse);
        }
    }

    private bool isGrounded()
    {
        if (Physics2D.BoxCast(transform.position, jumpCastSize, 0, -transform.up, jumpCastDistance, groundLayer))
        {
            return true;
        }
        return false;
    }

    private bool isWallLeft()
    {
        RaycastHit2D hitInfo = Physics2D.BoxCast(transform.position, wallCastSize, 0, -transform.right, wallCastDista
        if (hitInfo.collider != null)
        {
            if (hitInfo.collider.gameObject.GetComponent<PlatformEffector2D>() != null)
            {
                return false;
            }
            return true;
        }
        return false;
    }

    private bool isWallRight()
    {
        RaycastHit2D hitInfo = Physics2D.BoxCast(transform.position, wallCastSize, 0, transform.right, wallCastDistan
        if (hitInfo.collider != null)
        {
            if (hitInfo.collider.gameObject.GetComponent<PlatformEffector2D>() != null)
            {
                return false;
            }
            return true;
        }
        return false;
    }
```

```
192
193     private void OnDrawGizmos() //Visualize RayCasts
194     {
195         Gizmos.DrawWireCube(transform.position-transform.up * jumpCastDistance, jumpCastSize);
196         Gizmos.DrawWireCube(transform.position-transform.right * wallCastDistance, wallCastSize);
197         Gizmos.DrawWireCube(transform.position+transform.right * wallCastDistance, wallCastSize);
198     }
199
200     private void Dash()
201     {
202         dashTimer = 0;
203         isDashing = true;
204         playerCollider.enabled = false;
205         dashCollider.enabled = true;
206         Vector2 dir = new Vector2(input.x, input.y).normalized;
207         if (dir.sqrMagnitude>0)
208         {
209             dashDirection = dir;
210         }
211         else
212         {
213             if (isPlayerRight)
214                 dashDirection = new Vector2(1,0);
215             else dashDirection = new Vector2(-1, 0);
216         }
217     }
218 }
```

Character Shoot

```
1
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public class CharShoot : MonoBehaviour
7  {
8      [SerializeField] float fireRate = 300f;
9      [SerializeField] private GameObject bullet;
10     [SerializeField] private Transform bulletSpawn;
11
12     private GameObject bulletInst;
13     private bool allowShoot = true;
14     private float overheatTimer = 0;
15     private bool overheat = false;
16     private int maxOverheat = 4;
17
18     private void Start()
19     {
20         OverheatBar.Instance.SetMaxValue(4);
21         OverheatBar.Instance.SetValue(0);
22     }
```

```
23
24      private void Update()
25      {
26          if (Input.GetButtonDown("Fire1") && allowShoot && !overheat)
27          {
28              StartCoroutine(Shoot());
29          }
30
31          Overheat();
32      }
33
34      private IEnumerator Shoot()
35      {
36          allowShoot = false;
37          overheatTimer++;
38
39          if (CharMovement.Instance.isPlayerRight)
40          {
41              bulletInst = Instantiate(bullet, bulletSpawn.position, Quaternion.Euler(0, 0, 0));
42          }
43          else
44          {
45              bulletInst = Instantiate(bullet, bulletSpawn.position, Quaternion.Euler(0, 0, 180));
46          }
47
48          yield return new WaitForSeconds(1/(fireRate/60f));
49          allowShoot = true;
50      }
51
52      private void Overheat()
53      {
54          if (!overheat)
55          {
56              OverheatBar.Instance.SetValue(overheatTimer);
57              if (overheatTimer >= 0)
58              {
59                  //Heat reduces over time while shooting
60                  overheatTimer -= Time.deltaTime * (2 + 0.4f * overheatTimer) * (fireRate / 450f);
61
62                  //Disable shoot when overheat
63                  if (overheatTimer >= maxOverheat) overheat = true;
64              }
65          }
66          else
67          {
68              OverheatBar.Instance.SetValueRed(overheatTimer);
69              overheatTimer -= Time.deltaTime * 2;
70              //Enable shoot when heat is reduced
71              if (overheatTimer <= 0) overheat = false;
72          }
73      }
74  }
```

Bullet Behavior

```
1
2 using UnityEngine;
3
4 public class BulletBehavior : MonoBehaviour
5 {
6     [SerializeField] private Sprite[] sprites;
7     [SerializeField] private Material[] materials;
8     [SerializeField] private float normalBulletSpeed = 25f;
9     [SerializeField] private float physicsBulletSpeed = 30f;
10    [SerializeField] private float lifeTime = 3f;
11    [SerializeField] private LayerMask destroyLayers;
12
13    private Rigidbody2D rb;
14    private SpriteRenderer sr;
15
16    public enum BulletType
17    {
18        Normal,
19        Physics
20    }
21    public BulletType bulletType;
22
23    public enum BulletElement
24    {
25        Fire,
26        Water,
27        Ice,
28        Metal,
29        Corrosion
30    }
31    public BulletElement bulletElement;
32
33    private void Start()
34    {
35        rb = GetComponent<Rigidbody2D>();
36        sr = GetComponent<SpriteRenderer>();
37
38        SetSprite();
39        SetLifeTime();
40        SetSpeed();
41    }
42
43    private void FixedUpdate()
44    {
45        if (bulletType == BulletType.Physics)
46        {
47            transform.right = rb.velocity;
48        }
49    }
50
51    private void OnTriggerEnter2D(Collider2D collision)
52    {
```

```
53          //Is collision in the layers
54          if ((destroyLayers.value & (1 << collision.gameObject.layer))>0)
55          {
56              //Damage Enemy
57              if (collision.gameObject.layer == 8)
58              {
59                  collision.gameObject.GetComponent<Enemy>().Damage(1f);
60              }
61  
62              //Destroy bullet
63              Destroy(gameObject);
64          }
65      }
66  
67      private void SetSprite()
68      {
69          switch (bulletElement)
70          {
71              case BulletElement.Fire:
72                  sr.sprite = sprites[0];
73                  sr.material = materials[0];
74                  break;
75              case BulletElement.Water:
76                  sr.sprite = sprites[1];
77                  sr.material = materials[1];
78                  break;
79              case BulletElement.Ice:
80                  sr.sprite = sprites[2];
81                  sr.material = materials[2];
82                  break;
83              case BulletElement.Metal:
84                  sr.sprite = sprites[3];
85                  sr.material = materials[3];
86                  break;
87              case BulletElement.Corrosion:
88                  sr.sprite = sprites[4];
89                  sr.material = materials[4];
90                  break;
91              default:
92                  sr.sprite = sprites[0];
93                  sr.material = materials[0];
94                  break;
95          }
96      }
97  
98      private void SetLifeTime()
99      {
100         Destroy(gameObject, lifeTime);
101     }
102 
103     private void SetSpeed()
104     {
105         if (bulletType == BulletType.Normal)
106         {
```

```
107             rb.velocity = transform.right * normalBulletSpeed;
108             rb.gravityScale = 0f;
109         }
110         else if (bulletType == BulletType.Physics)
111         {
112             rb.velocity = transform.right * physicsBulletSpeed;
113             rb.gravityScale = 4f;
114         }
115 
116     }
117 }
```

Laser Behavior

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  
5  public class LaserBehavior : MonoBehaviour
6  {
7      [SerializeField] private LineRenderer lineRenderer;
8      [SerializeField] private Transform bulletSpawn;
9      [SerializeField] private LayerMask Player;
10     private float laserLength = 23f;
11 
12     void Update()
13     {
14         lineRenderer.SetPosition(0, bulletSpawn.position);
15         lineRenderer.SetPosition(1, bulletSpawn.position);
16 
17         if (Input.GetButton("Fire1"))
18         {
19             SetDestination();
20         }
21     }
22 
23     void SetDestination()
24     {
25         Vector2 destination = bulletSpawn.position;
26         if (CharMovement.Instance.isPlayerRight)
27         {
28             destination = (Vector2)transform.position + new Vector2(laserLength, 0);
29         }
30         else
31         {
32             destination = (Vector2)transform.position + new Vector2(-laserLength, 0);
33         }
34 
35         RaycastHit2D hitInfo = Physics2D.Raycast((Vector2)transform.position, transform.right, laserLength, ~Player);
36         if (hitInfo)
37         {
38             if (hitInfo.collider != null)
```

```
39             {
40                 if (hitInfo.collider.gameObject.layer == 6)
41                 {
42                     destination = hitInfo.point;
43                 }
44                 else if (hitInfo.collider.gameObject.layer == 8)
45                 {
46                     destination = hitInfo.point;
47                 }
48             }
49         }
50 
51         lineRenderer.SetPosition(1, destination);
52     }
53 }
```