Shape Fight: The Impact of Game Mechanics on User Experience

Anna Madzihon

N00226543

Supervisor: Joachim Pietsch
Second Reader: Michael McAndrew

Year 4 2025/2026
DL836 BSc [Hons] in Creative Computing

Declaration of Authorship

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

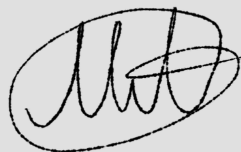
The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below.

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student: Anna Madzihon



Signed:

Failure to complete and submit this form may lead to an investigation into your work.

Abstract

This project explores the impact of game mechanics on the overall player experience with the final product, aiming to define how differently the players perceive the same game through using different tools to interact with it. To research this impact, a short game named Shape Fight was created using Godot 4.6.2, which gathered the user data while testing the project. The purpose of collecting this data is later to compare two different input systems within this game and how they affect the final product.

All the research, definitions, implementation and testing are discussed in this paper. The final result is a functional turn-based game which was tested by a group of users, providing visualised data at the end.

Acknowledgements

First, I would like to thank my friends and family back at home for supporting me for all the years I have spent abroad working towards this thesis. Despite the distance we stayed closer than ever, inspiring me to go forward no matter the circumstances.

I would like to thank my supervisor, Joachim Pietsch, for providing guidance when planning and developing this project, ensuring that it would become the best possible version of itself.

I would also like to thank my second reader, Michael McAndrew, for providing feedback during the initial stages of this project. These initial stages were crucial in deciding the aim of the entire project.

Lastly, I would like to thank all the friends I made during my time in IADT. You kept me motivated during the most difficult times over the past four years, slowly becoming my second family. I wouldn’t get this far without your constant support from the very beginning.

Table of Contents

Declaration of Authorship....2
Abstract....3
Acknowledgements....4
Table of Contents....5
1. Introduction....1
2. Research....2
  1.1. Background....2
  1.2. Research Questions....2
  1.3. Objectives....2
  1.4. Report Structure....2
3. The Definitions behind Games....3
  3.1. Introduction....3
  3.2. Game Design....3
  3.3. Game Mechanics....3
  3.4. Conclusion....4
4. Requirements....5
  4.1. Introduction....5
  4.2. Requirements Gathering....5
    4.2.1. Existing game mechanics studies....5
    4.2.2. Examples of game mechanics and interface....5
  4.3. Feasibility Study....6
    4.3.1. Game Concept....6
    4.3.2. Game design and mechanics....6
    4.3.3. Tools and Implementation....6
  4.4. Backlog of Features....7
  4.5. Conclusion....8
5. Design....10
  5.1. Introduction....10
  5.2. Game Design....10
    5.2.1. Genre and Gameplay....10
    5.2.2. Scene Management....10
    5.2.3. Game Logic and Characters....11
    5.2.4. Character management....13
    5.2.5. Camera Systems....14
    5.2.6. Enemy AI....14
  5.3. User Interface Design....15
    5.3.1. Visual Presentation....15
    5.3.2. Storyboard....15
    5.3.3. Mouse input system....16
    5.3.4. Keyboard input system....17
  5.4. Conclusion....17

6. Implementation....18
6.1. Introduction....18
6.2. Scrum Methodology....18
6.3. Sprints....18
6.3.1. Sprint 1 – Research and planning....18
6.3.2. Sprint 2 – Requirements Gathering and Design....19
6.3.3. Sprint 3 – Design and Implementation....19
6.3.4. Sprint 4 – User Interface and Functionality....21
6.3.5. Sprint 5 – Code optimisation and Bug Fixes....26
6.3.6. Sprint 6 – Final code implementations....28
6.3.7. Sprint 7 – Testing....35
6.4. Conclusion....35
7. Testing....37
7.1. Introduction....37
7.2. Functional Testing....37
7.3. User Testing....37
7.3.1. Mouse input testing....38
7.3.2. Keyboard input testing....38
7.3.3. Quantifiable data....38
7.4. Conclusion....40
8. Project Management....41
8.1. Introduction....41
8.2. Project Management tools....41
8.2.1. Joplin....41
8.2.2. Miro....41
8.3. Conclusion....42
9. Conclusion....43
9.1. Summary....43
9.2. Future Improvements....43
References....44

1. Introduction

This project aims to study how game mechanics affect the user's perception of a game by creating a game prototype and testing it.

The specific idea behind this project is to create a game prototype named Shape Fight with varying implementations of game mechanics, different ways of user interaction with the project, while keeping the same game design. This means creating essentially one game but providing different interfaces for the players to interact with, focusing on studying the differences of how these changes are perceived by the users.

The tools used for this project are all free and open-source, meaning that the program code is openly accessible to the users. This allows to modify the programs to the user's needs, creating plugins or modifying the entire program. The reason to choose these programs is the potential to develop the project further, producing tools and addons to make the later improvements easier.

Godot was used as the game engine, creating an integrated environment to write code and edit the in-game world. The engine also provides clean and easily accessible documentation, which has made the development process much easier.

Along with Godot, Blender was used as the 3D modelling software for creating the game assets. The game itself kept a simplistic design, focusing on its functionality instead. Regardless, the simple assets have made the project more approachable, providing the players with more action feedback using character animations.

Following the development process, the project was tested with various groups of participants using the two versions of the user input, providing the data on each of them.

The following sections go in-depth about the research, development process, testing and the final project result. The document describes each step, the encountered difficulties, changes from the initial concept and explanations of the concepts followed in these chapters.

The final product is a short, turn-based fighting game with settings which change the input and camera movement types.

2. Research

1.1. Background

The development of computer games is still a relatively new art form and science, which makes the related definitions vastly different from each other. Generally, game design is defined as a combination of pattern recognition and testing, with every game development studio providing their own ideas and philosophies behind this terminology. There is not a single defined way to design a game, which creates infinite variations of work flows in how the game is created, designed and tested.

1.2. Research Questions

At the end of this project, the following questions will be answered:

- How much do the game mechanics affect the way users interact with the game design?
- Will different inputs change the user experience?
- How can the testing and prototyping of games be improved further?

1.3. Objectives

Following the questions, the objectives of this research are to create an environment suitable for testing with different game mechanics and input methods. The testing results must be analysed and compared, finding the differences in user feedback and finding the ways to improve both testing and prototyping based on the experience with the development and testing process during this research.

1.4. Report Structure

This report will be structured into chapters, following each step of the project development. Each chapter contains sub-headers, which further explain each topic, providing examples and research. The following chapter will provide the project-related definitions and the theory behind game design and development.

3. The Definitions behind Games

3.1. Introduction

The history of video game development goes as far as the beginning of the 20th century, with many sources arguing which computer game was considered to be the first. According to the Guinness World Records website *(First Videogame, n.d.)* *Spacewar!* is often regarded as the first influential computer game, the first implementation of a computer game is also considered to be the *Christopher Strachey's Draughts* game which used the tube displays and was made in 1952.

This marks the beginning of game design, when the computer was first used as an opponent for the player.

3.2. Game Design

Game design and the following studies are often outlined by the developers themselves, creating varying definitions.
According to the definition by Koster (2014, pp. 20–25), games are made of patterns which follow specific rules. The human mind is built to recognise these patterns, which provide the brain with intellectual stimulation. When following the patterns and solving the provided problems humans experience fun.

Following this definition of game design, the job of a game designer is to create a recognisable pattern and provide the player with a challenge. However, according to *Schell (2019, pp. 24-36)* defining game design is the job of the game designer, since the definition itself relies on each individual developer.

For the purposes of this research, Kosher's definition will be used when referring to game design in the following chapters. Game design is the art of creating problems for the player to solve, providing a challenge with a specific set of rules.

3.3. Game Mechanics

Similarly to the definitions of game design, game mechanics are defined differently by individual developers. While some developers use game mechanics and game design terminologies interchangeably, *Brazie (2022)* defines game mechanics as the ways the player interacts with the game.

If game design defines the environment, challenges and rules, game mechanics provide the player with tools to solve these challenges: movement of the player character, weapons to attack the enemies, buttons to decide the next action in a story.

With this definition game mechanics become closely intertwined with the user interface, becoming the tools to form user experience.

3.4. Conclusion

Both game design and game mechanics are foundational concepts behind this research. Understanding how they interact and work together is important when designing and developing a game, creating an environment which provides the player with two different ways to interact with the game. Seeing how these mechanics affect the player’s response to the game design is the goal of this project.

4. Requirements

4.1. Introduction

This chapter analyses the existing studies of game mechanics and user interface. These existing examples will provide the idea and requirements for creating the game which allows for successful testing.
Feasibility study explains what goes into developing a game with the provided time limit and tools.

4.2. Requirements Gathering

As games quickly gained popularity as a medium, many studies provided their own definitions and researched the games which create a better environment and ways to interact with it than others.

4.2.1. Existing game mechanics studies

Many studies define game mechanics, with *Sicart (2011)* analysing the game mechanics and input of various games. This study focuses on understanding mechanics, the different skills developed with varying controller layouts such as the ones often found in fighting games. In many of those cases, the players are counted as skillful if they mastered the specific input layouts.

This theory is tested in this project as well, providing the players with different control layouts and comparing the skills developed using quantitative data. The developed prototype will track the time it takes for the players to beat each level, defeat the enemy and finish the game based on each of the layouts, testing the game mechanics in action.

4.2.2. Examples of game mechanics and interface

The game mechanics and the user interface are closely related. Taking examples provided by *Delfino (2025)*, it becomes easier to see how different games provide these interfaces through inventory management and menus. While many modern games tend to minimise the amount of space the menus take up on the screen, some games utilise more crowded interfaces.

This shows that game mechanics and user interfaces are heavily reliant on the game design: depending on how the player is meant to manage the resources provided by the game, the interfaces can either be minimal or very crowded.

4.3. Feasibility Study

4.3.1. Game Concept

Following the examples of the user interfaces, choosing which game type to develop becomes easier. Due to the time constraints the game has to be simple enough to implement during the provided development time while containing enough content variety to allow for the different implementations of Shape Fight's game mechanics and user interface.

After analysing the genres, the decision was made to create a turn-based role-playing game (RPG), inspired by Japanese role-playing games which provide the player with playable characters, all having a list of abilities they can use during their provided turn. These games provide a good balance of simplicity in the game design, while allowing for many ways to interact with the game and test the different input types.

4.3.2. Game design and mechanics

A turn-based game is required to have the following functionality:

- Allowing the player to control multiple characters on each of their turns;
- Allowing to select the character actions and the units to use these actions on;
- Handling the states of each character, including their health and energy;
- Handling the character deaths;
- Managing the winning or losing conditions, providing the player with the ability to try again;
- Smoothly managing the turn-based combat;
- Providing feedback to the player's actions.

4.3.3. Tools and Implementation

The engine of choice for the creation of this project is Godot. It is a free engine which utilises its own programming language called GDScript, based on the Python programming language. This engine provides clear documentation, allowing you to pick up on the programming concepts and functions easily. It was chosen instead of the Unity Engine, which was used to develop other game development projects before, due to its open-source nature, allowing for more flexibility when this project is developed further in the future. For this project, version 4.6.2-stable was used.

There is a single major difference to consider when comparing Godot to the Unity Engine: instead of using an object-based system, Godot uses nodes. The nodes act like easily configurable parts of the scene with many different types and presets. Each node acts like a scene: every element on the screen of a Godot game behaves like a separate scene, which allows to drag-and-drop any saved element into the world environment and it will function predictably.

To create assets for the game, Blender was used as the free, open-source environment for creating three-dimensional models. Thanks to the previous

experience of using this software it proved to be both simple and fast when developing the animated character models.

Joplin was used to organise the development and create easily accessible to-do lists. It is a note taking application providing the necessary formatting tools to create task lists and organise the notes in the order they must go in.

Sounds were created using a software named 1BITDRAGON. It is a lightweight digital audio wordstation which mainly uses retro instruments in a simple, easy to use environment. Most of the sounds in the project are short sequences of notes, signifying different states of the characters and providing extra input feedback.

Buttons for the user interface were created using Procreate, an art program which allows exporting most file formats, including short animated sequences.

With these tools being openly accessible to large communities it becomes easy to look for solutions to programming errors or issues with the assets.

4.4. Backlog of Features

Feature	Description	Priority
Round management system	The system which handles each round of the game, determining the functions which get called depending on the character and turn.	High
Character handling system	Handles all the characters on the screen, allowing them to use their unique actions. The system also sorts the characters based on their speed, making the characters with the highest speed play first.	High
Enemy AI	Decides the actions of the enemy unit, attacking using the best possible actions or healing other damaged enemies when possible.	High
Character attacks	Accessing the character's attack statistic, applying damage to the selected unit.	High
Character skills	Accessing the array of character's skills, defining whether they are meant to deal damage or heal units. Depending on the skill types the selection of units is different: damage skills must select	High

	enemy units, while healing skills must select the allied units. Apply the appropriate damage or healing.	
Character defense	Allowing the characters to defend themselves instead of attacking, lowering the incoming damage.	**High**
Death system	Creating the death state of units, not allowing them to play during their turn if their health reaches zero.	**High**
Mouse input system	Allowing the player to interact with the game and select the actions using a menu-based button system.	**High**
Keyboard input system	Allowing the player to interact with the game and select the actions using the keyboard bindings instead of the mouse, selecting the enemies or abilities.	**High**
Mouse enemy selection system	Allowing the player to click on any units on the scene, bringing up the action menu based on the selected unit. The idea is to create a minimalistic user interface.	**Medium**
Character assets	Creating three-dimensional animated assets for each of the characters.	**Medium**
Unit selection circle	Providing feedback to the player based on which unit is being selected by showing a circle above the units.	**Medium**
Character animations	Applying character animations when appropriate.	**Low**
Health bars above the enemy units	Displaying the unit health above their model.	**Low**
Pre-determined spawning locations for units	Pre-determining spawning locations for the units, handling their placement using the game script.	**Low**

4.5. Conclusion

Following the research into the existing studies and projects, the final project idea is to create a turn-based game which provides the users with different types of input, testing the responses to each of them.

The tools selected to create this project make it feasible to develop within the provided timeline, allowing access to documentation and community forums in the case of issues with any part of the implementation.

This chapter has outlined the key functionalities, providing a list of achievable goals for the project.

5. Design

5.1. Introduction

This chapter defines the core elements of Shape Fight building upon the specified requirements. The final design must allow for a reasonable amount of development while maintaining enough functionality for the future testing.

This chapter is divided into two major sections:

- **Game design:** the game design defines how the project works and behaves, involving the systems which can act outside of the player's control. The proper game design allows for a smooth system which makes the game run without any issues.
- **User Interface Design:** the UI design defines the ways the player interacts with the game and receives clear feedback after an action. This includes the input systems and a major part of the visual presentation of the game, not only defining the look but also the feel of the project.

Each of the sections includes sub-sections, which further define each aspect of the final design.

5.2. Game Design

5.2.1. Genre and Gameplay

As specified in the previous chapter, the selected game genre is a turn-based role-playing game. According to *Katarzyna Witkowska (2025)*, the genre is defined by the player taking an action sequentially within a limited-time turn, or, alternatively, after exhausting their action points. The main aspect of this genre is the decision making which comes with the limited action resources, forcing the player to plan their actions ahead of time to defeat their enemies.

This leaves the gameplay quite straightforward: the player is presented with a list of actions their character can take, along with the statistics the character possesses: health, energy, attack etc.

5.2.2. Scene Management

Due to the turn-based nature of the game, all the systems must be managed sequentially, making them work only after the player or the enemy have taken their action.
The best way to implement this system is to handle the logic through callable functions:

- Handling a new round — selecting the following function based on two parameters: is the next character an ally or an enemy? Is the next character alive or dead?

- Handling the player action — allowing the player to select their action based on their current played character.
- Handling the enemy action — allowing the enemy to act using a simple AI system, selecting whether to heal their ally or choosing a player's unit to attack at random.
- Handling the winning or losing condition — if all enemies are dead, the player wins the game. If all the player's characters are dead, the player loses.

The following Figure 1 shows a graph which explains the functions handling the game conditions.

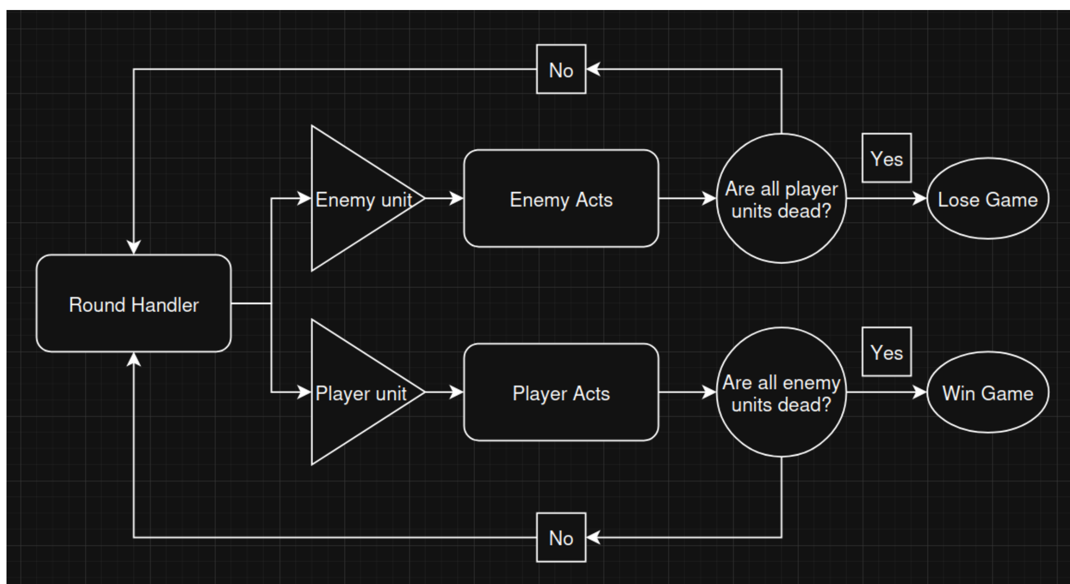


Figure 1: Function graph

5.2.3. Game Logic and Characters

Each of the characters must be able to act, which requires two things:

- **A list of actions:** the character must be able to attack, use a specific skill, defend or skip their turn. The attacks use the base character attack statistic to deal damage. The skills are a list of abilities which the character can use to deal extra damage or heal their allies, which allows for more complex gameplay. The defend ability will allow the character to block extra damage until the start of their next turn. The skip ability allows you to skip the character's turn.
- **A list of statistics:** depending on the action, an attack statistic will boost the character's attack, the defense will make them block more damage, speed will let them play earlier, health will let them take more hits and energy will allow them to use more expensive skills.

The complex list of abilities allows for a flexible gameplay and the variety of characters. However, not only does the system need to make the characters

function, but they have to be easy to set up to function properly: this will allow for easier further development, creating new enemy types quickly.

The following list contains all the characters in the game:

- **Cube:** the first playable character. Has a single skill which deals extra damage to the enemies.

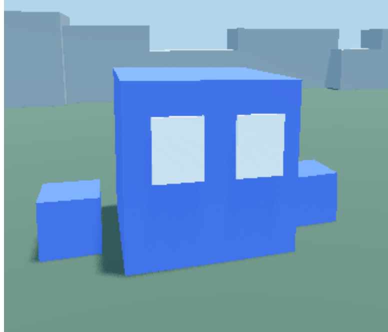
  

  Figure 2: The Cube Character
- **Pyramid:** the second playable character. Has one attack skill which deals extra damage and one healing skill to aid the allies.

  Figure 3: The Pyramid Character

- **Planet Cluster:** the first encountered enemy. Has a single skill dealing extra damage to the player characters.

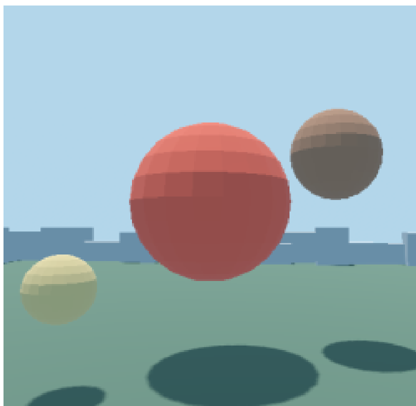
  

  Figure 4: Planet Cluster Character

- **The Spider:** the second encountered enemy. Has a single skill which heals the enemies.

Figure 5: The Spider Enemy

- **OrboRage:** the last encountered enemy. Has a very strong but expensive attack skill.

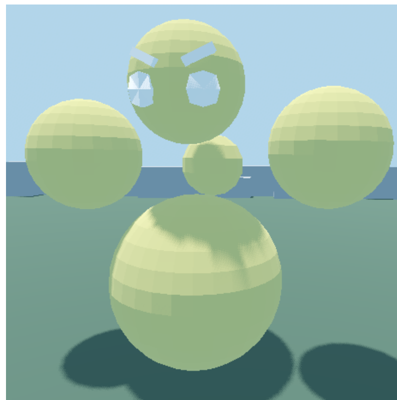


Figure 6: The OrboRage Enemy

5.2.4. Character management

There are many characters in the game, and managing them is a challenge unless there is a good system in place.
Instead of hardcoding the game to have a limited amount of units on the level, the code was written with the idea of placing as many units as required.
First, a parent node was created in Godot to store all the characters inside, as shown in Figure 7. This allows the script to later select all the child nodes of this parent, making them a part of the characters array.

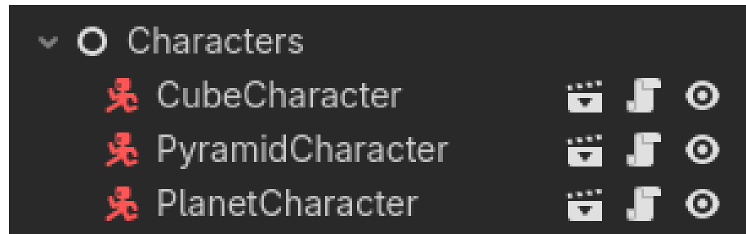


Figure 7: Characters Node

After structuring all the objects inside this node, the entire system was built around the idea of not knowing how many characters could be in the characters array. This streamlined the character management process into clear steps:

- Sorting the characters array by speed, allowing the fastest character to go first;
- Reading the first character in the array and checking their type: if the type is an “ally”, the player can control this character. If the type is “enemy”, the AI controls the character for the turn.
- Reading the statistics of each character: every character has a script attached which contains their statistics. The main game management script reads them and displays the character’s abilities.

With this system, the characters are easy to set up and work with. This allows you to put as many characters on the scene as required without worrying about changing any code values inside the main script.

5.2.5. Camera Systems

Due to the static nature of the game the camera must move from character to character, providing the player with the cinematic view of the scene while maintaining a good view allowing the player to understand what is happening during the turns.

This leads to a limitation in Godot’s base functionality: the camera would take a while to program manually to move smoothly between the units. Instead, a plugin called PhantomCamera can be used. It allows for a smooth transition between different units, providing the settings for the movement speed.

The camera must be re-programmed each turn, forcing it to follow the next unit. This can be done at the same time as the script functions handle the round, assigning the camera to the correct entity.

5.2.6. Enemy AI

The enemy AI is a complex function which must determine how the enemy acts during their turn.

The key to creating a good enemy AI is ensuring that they keep selecting the best course of action for their team while not making the enemy too smart or difficult for the player to be unable to beat.

There are two kinds of enemies in the game: attackers and healers. The attacking enemies only have the attack skills, while the healers possess the ability to heal their team.

The AI considers the following logic when selecting the enemy action:

- Does the character have a healing skill? If so, does the team need healing? If so, heal the team.
- Does the character have enough energy to use a strong attack skill? If so, choose whether to use it or not with a 50/50 chance.
- Which character to attack? Select all the alive units marked as ”player characters” and randomly select one to attack.

Following this logic, the enemy AI is capable of using all of its available abilities. The AI is random to give the player a higher chance of success, since it is not targeting specific player characters based on low health.

5.3. User Interface Design

5.3.1. Visual Presentation

All the user interface elements were created manually. The game has bright, pink menus which create a good contrast to the rest of the game, making the buttons easily readable.

The game mainly consists of menus, with the exception of the unit selection screen used with the keyboard input system. Figure 8 and Figure 9 showcase the examples of the user interface.

Figure 8: Button Layout

Figure 9: Health Bars

5.3.2. Storyboard

The following Figure 10 includes the storyboard for the game.

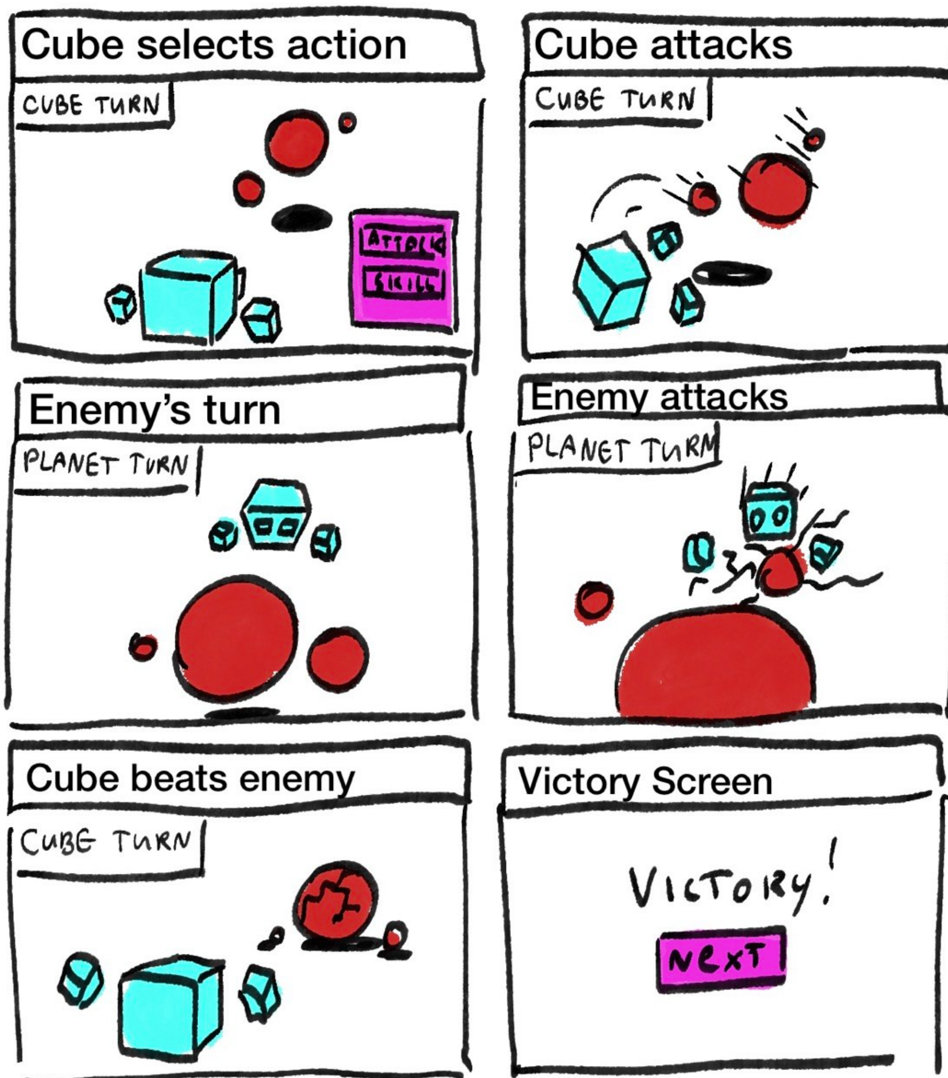


Figure 10: Storyboard

5.3.3. Mouse input system

The mouse input system is straightforward to implement. It consists of buttons provided by the Godot engine, which send the signal to the scripts once pressed or hovered over.

The button system becomes tricky when certain enemies must be selected: the code function has to create an array of all the enemies which can be selected to be attacked, generating buttons for each of the enemies in the fight. The buttons must also be linked to the enemies, running the appropriate function to attack the next enemy.

Additionally, the player must understand which unit is being targeted on the scene. To aid with that, an additional selection circle appears above the selected units when hovering over the buttons attached to them, as shown in Figure 11.

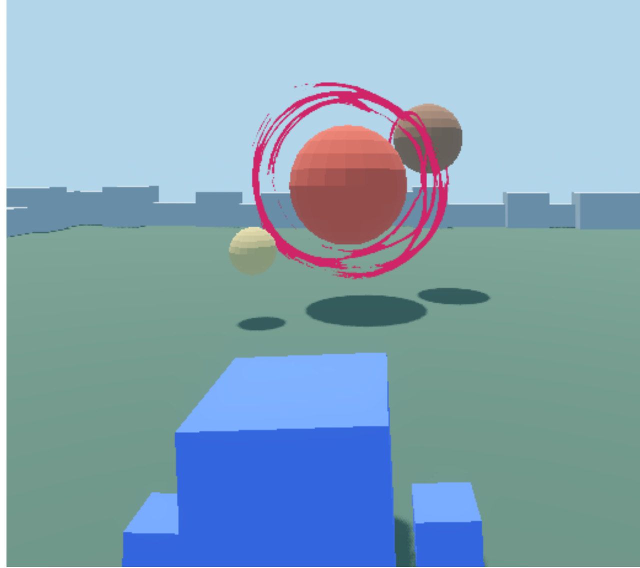


Figure 11: Enemy selection circle

5.3.4. Keyboard input system

The keyboard input system requires more effort than the mouse system: instead of handling all the input with mouse clicks, the user must be able to access the entire interface with the keyboard.
Godot includes some tools required to achieve this: the buttons on the scene can be focused, allowing the keyboard to select the correct button using the built-in input system.

The main complexity comes from the enemy selection screen. To achieve the same effect of using the keyboard to select the enemies, the code must be able to find all the selectable enemies and put them into a global array, accessible outside the functions. Additionally, a global function must store the selected enemy index, along with the boolean variable which stores the information about whether the enemy selection screen is active. If the screen is active, the input system registers the appropriate controls, allowing the user to select the enemy. Once the enemy is selected, an appropriate function is called to complete the action, and the enemy selection screen closes.

5.4. Conclusion

In conclusion, the game's turn-based aspect is carefully designed to work properly by calling the appropriate functions each turn. The inputs of the game require as much care and effort as the project itself, ensuring a smooth player experience when the game is developed and played.

The main challenge is the implementation of this system: this logic may look simple on the surface, but many careful calculations are required to ensure that the correct units deal or receive damage, that the system doesn't freeze or stop working entirely in the middle of the game.

6. Implementation

6.1. Introduction

The implementation of the project is the longest step of development, which means it requires the most care in both ensuring the proper functionality and managing the upcoming tasks. Due to the scale of the game it can become difficult to set the next goal without a proper plan.

The project management happens by using the Scrum methodology, which allows for the management of tasks to meet the final requirements of the project. In this chapter each of the sprints are described in detail, explaining each step of the project implementation.

6.2. Scrum Methodology

According to Scrum.org (2024), the Scrum methodology is a framework which allows to get the work done by separating every part of the project into smaller sprints. These sprints are the goals set in the beginning of each new sprint, which must be achieved by the end of the defined date. At the end of each sprint the goals are reviewed and are either finished or set as the goal of the next sprint along with the new goals.

This framework allows us to always have progress when developing the larger scale projects, knowing what the next goals are during every step of development.

6.3. Sprints

6.3.1. Sprint 1 – Research and planning

The first sprint was spent creating a plan and doing the research. The goals of the sprint are to do as much research as possible and decide on the game engine to use.

Multiple game design books were picked up for the research, allowing to deepen the understanding of the topic. This helped to narrow down the game-related definitions, providing the required context for later development and testing.

The main issue during the Sprint 1 was choosing the game engine for the implementation of the game. Initially, there was an idea to choose a more advanced engine, more specifically Unreal Engine 4. However, after looking into the development pipelines and having the initial difficulties when using the engine, this idea was scrapped in favour of using Godot.

Godot is still an engine that has to be learned from scratch, but the game prototyping process was immediately much easier and faster. Overall, it was a great decision made during the first sprint. The research goals were also achieved during this sprint, allowing to start work during the next sprint.

6.3.2. Sprint 2 – Requirements Gathering and Design

The goal of the second sprint was to gather the requirements for making the project and to start designing the turn-based fighting system.

Collecting the requirements was a fast process after understanding the concepts of game design and mechanics, however it was a slow process to get the turn-based system designed. It was mainly due to not understanding how Godot works, which means that most of the second sprint was spent working in the engine and testing its capabilities instead of actually designing the systems.

However, after understanding the syntax of GDScript and the functionality of Godot, it became easier to understand how the system would work. The approach was changed during the design stage: instead of simply designing the project and going ahead to develop it, the project was immediately being developed while learning how to use the engine. This slowed down the design stage but allowed to start developing the project.

Overall, at the end of this sprint the requirements were gathered but the design was not fully finished yet. The rest of the design work was instead done during the next sprint.

6.3.3. Sprint 3 – Design and Implementation

The goal of the third sprint was to finish the project design and continue implementing the basic project functionality.

This is the stage when the project development started becoming faster and more productive thanks to the better understanding of the engine. Along with that, the gameplay loop was fully designed, including all the statistics each character would have, as shown in Figure 12:

```
if characters[index].char_type == "ally":
	if characters[index].health <= 0:
		game_status.text = "Cannot play. " + characters[index].charName + " is dead."
		await get_tree().create_timer(2.0).timeout
		end_round()
	action_container.visible = true
	set_camera_pos()
else:
	action_container.visible = false
	set_camera_pos()
	if characters[index].health <= 0:
		game_status.text = "Cannot play. " + characters[index].charName + " is dead."
		await get_tree().create_timer(2.0).timeout
		end_round()
	else:
		enemy_attacking()
		await get_tree().create_timer(2.0).timeout
		end_round()
```

Figure 12: Round Handler

- First, the script determines if the current character is an ally or an enemy;
- If the character is an alive ally, the action container shows, allowing the user to interact with the game;
- If the character is an enemy, the script runs the enemy artificial intelligence;
- If the character is dead, the round is skipped.

During this sprint a normal attack was implemented for the allied characters, which allows them to use their base attack ability to deal damage.

```
# Handle damage to the enemy
func attack_enemy(enemy):
	var damage
	
	damage = calc_defense(characters[index].attack, enemy.defense, enemy.defending)

	if damage <= 0:
		damage = 1
	enemy.health -= damage
	totalDamageDealt += damage
	
	check_health(enemy)
		
	# Find correct health bar
	change_health_bar(enemy)
	
	game_status.text = "Dealt " + var_to_str(damage) + " damage to " + enemy.charName + "
```

Figure 13: attack_enemy() function

As shown in Figure 13, the logic behind the attack function is fairly simple: the attack is calculated by using a **calc_defense()** function.

```
func calc_defense(atk, def, defending):
	if defending:
		return atk - (def)
	else:
		return atk - (def / 3)
```

Figure 14: calc_defense function

```
func check_health(unit):
	if unit.health <= 0:
		unit.health = 0
		unit_die(unit)
```

Figure 15: check_health() function

This function shown in Figure 14 provides a simple logic by using the parameters to determine whether the attacked unit is defending or not. If the unit is not defending, the attack is reduced by its defense divided by three. If the unit is defending, the full defense statistic is used to reduce damage.

The rest of the function calculates the health and checks the health of the attacked unit using **check_health(enemy)** shown in Figure 15. The logic of this function determines whether the unit is alive by checking its health statistics.

In the figure is the provided example of the character statistics. Health and energy exist with the **maxHealth** and **maxEnergy** stats, determining the maximum values of these variables.

```
var charName = "Cube"
var maxHealth = 30
var health = maxHealth
var defense = 5
var attack = 20
var speed = 40
var char_type = "ally"
var charID = 0

var defending = false

var maxEnergy = 20
var energy = maxEnergy
var energyModifier = 0.5

var skillsNames = ["Cube Attack!"]
var skillsDescriptions = ["Does a POWERFUL cube attack!"]
var skillsDamage = [40]
var skillsCost = [20]
```

Figure 16: Character Statistics

The skills of the units are written in Figure 16, but they are made functional in the fifth sprint.

Overall, Sprint 3 was a success. During this stage of the project the game design was clearly defined and major parts of the implementation were completed.

6.3.4. Sprint 4 – User Interface and Functionality

The goal for the Sprint 4 was to improve User Interface by creating health bars, properly handling the player action menu and skill abilities.

One of the most important systems in the game is handling the character's skills, properly displaying their cost and applying their damage or healing to the units.

```
# After selecting a skill, select an enemy to attack
func _select_unit_skill(skill, skillCost):
	for child in selection_container.get_children():
		child.queue_free()
	if Settings.ui_mode == 0:
		selection_container.visible = true
		if skill > 0:
			var enemies = []
			for child in selection_container.get_children():
				child.queue_free()

			enemies = find_live_units_by_type("enemy")

			for enemy in enemies:
				add_skill_button(selection_container, skill, skillCost, enemy)
			add_back_button()

		elif skill < 0:
			var allies = []
			for child in selection_container.get_children():
				child.queue_free()

			allies = find_live_units_by_type("ally")

			for ally in allies:
				add_skill_button(selection_container, skill, skillCost, ally)
			add_back_button()
```

Figure 17: select_unit_skill() function

Figure 17 above shows the skill handling function. The function works by having an assigned container which has all the required buttons added to the container. When the function runs each time, the container is also cleared from all the previously instantiated buttons.

There are three important helper functions used in this:

- **find_live_units_by_type()** – looks through the character's array, selecting live units with a certain type. The function, as shown in Figure 18, also contains a failsafe in case the game had an issue handling the allied user's deaths,

forcing the game to end if no allies are alive.

```
func find_live_units_by_type(type):
	var unit_array = []
	for character in characters:
		if character.char_type == type && character.health > 0:
			unit_array.push_front(character)
		
		## Failsafe in case ally array is empty
		if unit_array.size() == 0 && type == "ally":
			lose_fight()
	
	return unit_array
```

Figure 18: find_live_units_by_type() function

- **add_skill_button()** – instantiates a button inside the provided container with the provided information, as shown in Figure 19. The button is bound to pre-written functions, which handle the skills' behaviour.

```
func add_skill_button(container, skill, skillCost, unit):
	var newButton = BUTTON_EXAMPLE.instantiate()
	newButton.pressed.connect(handle_skill.bind(skill, skillCost, unit))
	newButton.mouse_entered.connect(_skill_button_mouse_entered.bind(unit))
	newButton.mouse_exited.connect(_skill_button_mouse_exited.bind(unit))
	if Settings.ui_mode == 1:
		newButton.focus_mode = Control.FOCUS_ALL
	newButton.text = var_to_str(unit.charName)
	container.add_child(newButton)
```

Figure 19: add_skill_button() function

- **add_back_button()** creates a button which hides the selection screen, going back to the previous menu. Its implementation is shown in Figure 20.

```
func add_back_button():
	# Add button to go back
	var backButton = BUTTON_EXAMPLE.instantiate()
	backButton.pressed.connect(_back_to_action_button_pressed)
	backButton.text = "Cancel"
	selection_container.add_child(backButton)
	if Settings.ui_mode == 1:
		backButton.focus_mode = Control.FOCUS_ALL
		backButton.call_deferred("grab_focus")
```

Figure 20: add_back_button() function

The function which handles the skills themselves is fairly complex. It contains multiple checks, determining the function behaviour depending on the skill type.

```
371 func handle_skill(skill, skillCost, unit):
372     if characters[index].energy < skillCost:
373         play_low_energy(characters[index])
374         status_text.text = "Not enough energy to use skill!"
375     else:
376         characters[index].energy -= skillCost
377         if characters[index].energy < 0:
378             characters[index].energy = 0
379 
380         # Find correct energy bar
381         change_energy_bar(characters[index])
382 
383         if skill > 0:
384             var damage = 0
385 
386             damage = calc_defense(skill, unit.defense, unit.defending)
387 
388             if damage <= 0:
389                 damage = 1
```

Figure 21: First half of handle_skill()

```
394             unit.health -= damage
395             totalDamageDealt += damage
396 
397             selection_container.hide()
398             # Handle Animations
399             characters[index].attack_animation()
400             characters[index].queue_idle_animation()
401 
402             if !is_dead(unit):
403                 await get_tree().create_timer(1.0).timeout
404                 sfx_attack.play()
405                 unit.attacked_animation()
406                 unit.queue_idle_animation()
407 
408             check_health(unit)
409             # Find correct health bar
410             change_health_bar(unit)
411 
412             game_status.text = "Dealt " + var_to_str(damage) + " damage to " +
```

Figure 22: Second half of handle_skill()

Figure 21 and Figure 22 above showcase the attack logic of the function. The skill becomes an attack if its attack value is higher than zero, handling the following skill behaviour similarly to the regular attack function.

```
410  elif skill < 0:
411      sfx_heal.play()
412      var heal = -skill
413      unit.health += heal
414      if unit.health > unit.maxHealth:
415          unit.health = unit.maxHealth
416  
417      # Find correct health bar
418      change_health_bar(unit)
419  
420      game_status.text = "Healed " + unit.charName + " by " + var_to_str(heal) + ",\nCurrently
421  
422  action_container.visible = false
423  selection_container.visible = false
424  end_round()
```

Figure 23: Healing skill

Following a different logic, Figure 23 shows how it handles the skill if its attack value is lower than zero. Negative attack value implies the healing skill type, which makes the function behave differently: instead of the enemy units, allied units become selectable, allowing to heal them when the skill is used.

```
629  func find_health_bar(unit):
630      var health_bar
631      for child in health_bars.get_children():
632          if child.charID == unit.charID:
633              health_bar = child
634      return health_bar
```

Figure 24: find_health_bar() function

```
608  ## Find correct health bar and assign health
609  func change_health_bar(unit):
610      var health_bar = find_health_bar(unit)
611      health_bar.set_health(unit.health, unit.char_type)
```

Figure 25: change_health_bar() function

As seen in Figure 24 and Figure 25 above, health bar related functions are also called. The figure shows how these functions work, assigning the appropriate health value to the health bars shown in Figure 26 below.

Figure 26: Changed Health Bars

Similarly to health, the unit's energy is changed when skills are used. Figure 27 and Figure 28 showcase the functions used to handle energy.

```
625  func change_energy_bar(unit):
626      var energy_bar = find_health_bar(unit)
627      energy_bar.set_energy(unit.energy, unit.char_type)
```

Figure 27: change_energy_bar() function

```
func regenerate_energy():
    # regenerate energy
    characters[index].energy += int(characters[index].maxEnergy * characters[index].energyModifier)
    if characters[index].energy > characters[index].maxEnergy:
        characters[index].energy = characters[index].maxEnergy
```

Figure 28: regenerate_energy() function

The energy bars work slightly differently: at the start of the round handler each character regenerates some energy based on their energy modifier. Then a different function applies the visual changes to the energy bar.

Overall, this sprint has been productive and met all the goals defined at the start.

6.3.5. Sprint 5 – Code optimisation and Bug Fixes

After implementing many functional requirements in the last sprint, many bugs and issues appeared.

To start with, the statistics of the Planet enemy had to be modified due to the very low damage it could deal to the allied units. This was due to using a very different formula for defense in the beginning, making regular defense fully block damage.

Then, the initial function which controlled the health bars was using the unit's name to find the appropriate health bar: this becomes an issue when two units of the same type are on the scene, changing both of their health bars incorrectly.

```
92      # Putting children of Characters node into array
93      var localIndex = 0
94      for x in characters_node.get_children():
95          characters[localIndex] = x
96          characters[localIndex].charID = localIndex
97          localIndex += 1
```

Figure 29: Assigning ID to the units

The solution to this issue is shown in Figure 29 above, which assigns each of the units with their own character ID number. This number is unique to each of the characters, letting the health bar function handle each object as unique and tied to each of the units.

Additionally, after placing multiple enemy units on the scene during testing it was revealed that they keep attacking the allied units even after they die. This issue was not difficult to fix, since it required simply adding an additional check during the round handler function. It starts treating the enemy units the same as it treats allies, not allowing the enemy's artificial intelligence to act when the unit is dead.

During this sprint the entire game management code was reviewed, encountering many cases of repetitive code. This code was later structured into smaller helper functions, being called during the appropriate times. This code includes the **calc_defense()** function, **add_skill_button()** and many others.

After fixing the code, there was time before the end of the sprint to add more functionality to the project. This time was spent researching the ways to make the in-game camera smoother, since the built-in camera kept snapping into the specified position instead of providing a smooth transition.

After the research, an add-on named Phantom Camera was found. This was exactly what this project needed, providing more flexibility to the camera movement. The phantom camera stores the variables which determine the unit it follows, which unit it looks at and how smoothly the camera moves. This allowed the creation of the appropriate function, making the camera move smoothly around the scene.

```
162 func set_camera_pos():
163     if Settings.camera_mode == 0:
164         var cam_pos = characters[index].cameraPosition()
165         camera_3d.look_at_damping = 0
166         if characters[index].char_type == "ally":
167             camera_3d.follow_damping = 0.5
168             camera_3d.look_at_target = enemies_center
169         else:
170             camera_3d.follow_damping = 0
171             camera_3d.look_at_target = allies_center
172         camera_3d.follow_target = cam_pos
173     elif Settings.camera_mode == 1:
174         var cam_pos = characters[index].cameraPosition()
175         camera_3d.look_at_damping = 0
176         if characters[index].char_type == "ally":
177             camera_3d.follow_damping = 0.5
178             camera_3d.look_at_target = enemies_center
179         else:
180             camera_3d.follow_damping = 0.2
181             camera_3d.look_at_target = allies_center
182         camera_3d.follow_target = cam_pos
```

Figure 30: set_camera_pos() function

Figure 30 above shows the two ways the camera is handled in the settings, with the main difference being the camera damping. Camera damping defines how smoothly the camera moves, with the number zero being instant movement and higher numbers providing smoother, slower movement.

Each time the camera position is set, the camera gets assigned a specific character. Each character contains a custom marker, which is a position behind the character. The position is set manually due to the different unit sizes and models. Then, the camera chooses to follow the selected unit while looking at the specified point on the scene to provide the cleanest view for the player.

Overall, this sprint has proven efficient, achieving an extra goal and fixing the already existing code.

6.3.6. Sprint 6 – Final code implementations

This sprint focuses on implementing all the remaining project requirements before the final testing. It is split into three major parts:

- **Functional requirements:** implementing the remaining features, such as allowing the enemy units to heal each other, implementing the main menu and the settings menu, recording the player's data into the JSON format during the project testing.

- **Visual improvements and program feedback:** improving the existing user interface by adding sound effects, animations, new enemy types, displaying more information about the skills.
- **User input:** finalising and improving the two user input systems: the mouse input system with the functional buttons and the keyboard input system with the ability to play the entire game without a mouse.

Following the functional requirements, the enemy artificial intelligence was expanded to make use of the healing skills. As shown in the following Figure 31, the enemy units prioritise the healing skills over the attack skills when they are required. This works by filtering the enemies whose health is lower than their maximum, assigning the variable which shows if the enemy is injured. The unit then uses a very similar logic to the skill handling function, where the healing ability adds extra health to the selected unit.

```
465	# See available actions
466	var playable_skills = []
467	for i in range(attacker.skillsCost.size()):
468		if attacker.skillsCost[i] <= attacker.energy:
469			playable_skills.append(i)
470
471	# See if heal skills exist
472	var heal_skills = []
473	var has_heal_skill = false
474
475	for i in playable_skills:
476		if attacker.skillsDamage[i] < 0:
477			has_heal_skill = true
478			heal_skills.append(i)
479
480	var injured_enemy_exists = false
481	for e in enemies:
482		if e.health < e.maxHealth:
483			injured_enemy_exists = true
484			break
485
486	var use_skill = false
487
488	# Priorities heal skills
489	if has_heal_skill and injured_enemy_exists:
490		use_skill = true
491	elif playable_skills.size() > 0:
492		use_skill = randi() % 2 == 1
```

Figure 31: Enemy AI

This functionality has been the last requirement to make the enemy AI smarter, providing the player with a challenge.

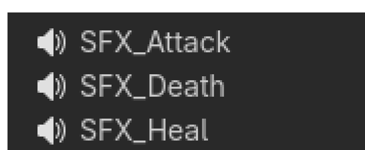
Following the visual improvements, the health bars used to be the same for every unit no matter which class they are. Now a tint is applied onto the bar upon instantiating, which not only paints the enemies' health bars a different colour, but adds a condition to change the tint if the unit is dead. The following Figure 32 provides the logic behind this change, which checks two conditions: health and unit type.

```
func set_health(health, type):
	if float(health) < 0:
		health_bar.set_value(0)
	else:
		health_bar.set_value(float(health))
	health_remaining.text = var_to_str(health)

	## Colour Logic
	if health <= 0:
		if type == "ally":
			set_colour_bar(ally_dead_text, ally_dead_var_counter, ally_dead_health,  ally_dead_energy)
		if type == "enemy":
			set_colour_bar(enemy_dead_text, enemy_dead_var_counter, enemy_dead_health,  enemy_dead_energy)
	else:
		if type == "ally":
			set_colour_bar(ally_text, ally_var_counter, ally_health,  ally_energy)
		if type == "enemy":
			set_colour_bar(enemy_text, enemy_var_counter, enemy_health,  enemy_energy)
```

Figure 32: set_health() function

Another notable change is the sound effects system. It was added by introducing three different sound effect nodes to the scene, activating the required node in the game management script when appropriate. Figure 33 and Figure 34 show the nodes and the example of calling the sound effects during the game.




```
elif skill < 0:
	sfx_heal.play()
	var heal = -skill
```

Figure 33: Sound Nodes *Figure 34: Calling Sound*

Lastly, to improve the testing results, data is gathered during the gameplay into a JSON file. This data will be later used to define how the different user input systems differ from each other.

The important part is defining which data is being collected: the most quantifiable data possible to collect is time, the current level and the input mode. After understanding which data may be useful, it can be written into a file.

First, a timer is created as soon as the game level starts. This timer is active for the duration of the level, making entries into the array of data.

The following Figure 35 shows an example of data entry, which runs when a unit dies. The data about the unit's death is saved into an entry, which is then appended into the JSON entry.

```
func unit_die(unit):
	sfx_death.play()
	if unit.char_type == "enemy":
		var entry = {
			"name": unit.charName,
			"time_since_start": Time.get_ticks_msec() - run_start_time,
			"turn_index": index,
			"killed_by": characters[index].charName
		}
		enemy_death_logs.append(entry)
	var allDead = true
```

Figure 35: unit_die() function

The following Figure 36 shows a snippet of the function which runs at the end of each level, writing the data into the file.

```
758	var data = {
759		"level": {
760			"id": current_level,
761			"ui_mode": Settings.ui_mode,
762			"camera_mode": Settings.camera_mode
763		},
764		"run": {
765			"time_ms": Time.get_ticks_msec() - run_start_time,
766			"actions": action_logs,
767			"enemy_deaths": enemy_death_logs
768		}
769	}
770
771	all_data.append(data)
772
773	var file = FileAccess.open(file_path, FileAccess.WRITE)
774	file.store_string(JSON.stringify(all_data, "\t"))
775	file.close()
```

Figure 36: Writing JSON

Following the user input, most work was put into the new input system, which uses the keyboard bindings to control the game. In Godot, all buttons can be made selectable using the keyboard when using the right focus mode and selecting the initial focused button to provide a starting point. This is not done by the default in the

engine, which is why the focus mode of each button in the container is changed at the start of the code.

```
if Settings.ui_mode == 1:
	for button in action_container.get_children():
		if button is Button:
			button.focus_mode = Control.FOCUS_ALL
	action_container.get_child(0).grab_focus()
```

Figure 37: Setting button focus

Figure 37 above gets every button in the specified container and sets the focus mode to the one used for the keyboard keybinds, along with getting the first button in the container and making it focused. The focused button later provides the user with the ability to navigate by pressing the buttons.

The main complexity of this input system comes from the enemy selection screen. In the regular enemy selection screen, buttons instantiate for each enemy on the screen. This functionality was written in the beginning of the project, but is not applicable to this input layout: it would be possible to simply use the existing buttons for enemy selection, but it wouldn't be different enough from the first input system.

Which is why a new system was written, with the variables as shown on the following Figure 38:

```
#### UI 1
var selected_unit_index = 0
var selectable_units = []
var selecting_unit = false
```

Figure 38: Selection Variables

These are global variables in the game manager code, which means they can get accessed by any function in the code. This is ideal for this input system: the function handling the enemy selection would also need to handle the input, which doesn't work with the existing functions written to select the units. Instead, the following function in Figure 39 is used when handling the input.

```
207 func _unhandled_input(event):
208     if !selecting_unit:
209         return
210 
211     if event.is_action_pressed("ui_right"):
212         selected_unit_index += 1
213         if selected_unit_index >= selectable_units.size():
214             selected_unit_index = 0
215         update_unit_selection()
216 
217     elif event.is_action_pressed("ui_left"):
218         selected_unit_index -= 1
219         if selected_unit_index < 0:
220             selected_unit_index = selectable_units.size() - 1
221         update_unit_selection()
222 
223     elif event.is_action_pressed("ui_accept"):
224         var target = selectable_units[selected_unit_index]
225         selecting_unit = false
226         clear_unit_selection()
```

Figure 39: unhandled_input() function

The **unhandled_input()** function is very similar to the existing **input()** function in godot, with the main exception: the function is only handling input if none of the graphic user interface elements are selected. Meaning that this function is never called when the player is handling the regular button menus.

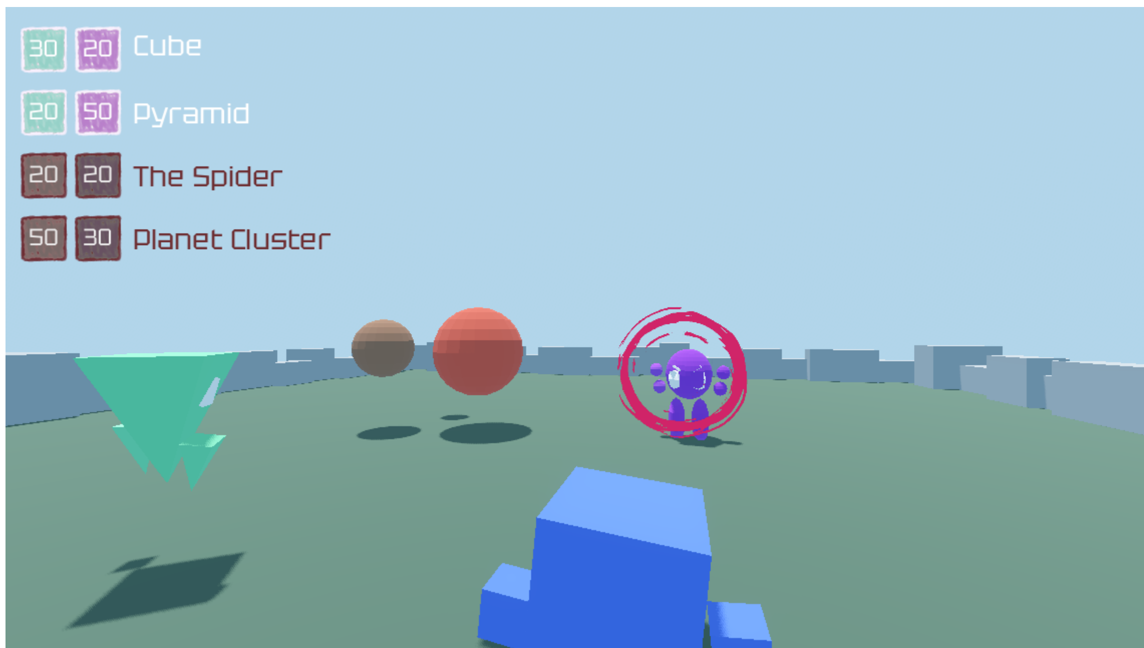


Figure 40: Enemy Selection using Keyboard

Figure 40 above shows the example of the keyboard-controlled enemy selection screen: instead of showing menus, the screen hides all of them in favour of selecting the units on the scene. When a unit is selected, a circle appears above it signifying that it is being currently selected.

Different units can be selected using the arrow keys, which are the default input layout in Godot. Any enemy can be selected using the enter key, while the selection can be cancelled altogether by using the escape key.

The following Figure 41 shows the code snippet which handles this input in the **select_unit_skill()** function. All the required variables are established in this function if the keyboard user input is being used, updating the selected entity.

```
elif Settings.ui_mode == 1:
	pending_skill = skill
	pending_skill_cost = skillCost
	selecting_skill_target = true

	selectable_units.clear()
	selected_unit_index = 0

	if skill > 0:
		selectable_units = find_live_units_by_type("enemy")
	elif skill < 0:
		selectable_units = find_live_units_by_type("ally")

	if selectable_units.is_empty():
		return

	selecting_unit = true
	update_unit_selection()
```

Figure 41: Input Handling

Overall, this sprint has concluded the main development of the project, allowing to proceed with the testing of the game.

6.3.7. Sprint 7 – Testing

The last goal of the project is running the user testing, collecting data and organising it into an easily readable format.

The testing stage started by finding the group of volunteers looking to test the project. After the search, seven people agreed to do the user testing.

The testing consisted of two major aspects:

- Viewing the user's response to the project, analysing the behaviour and seeing the reaction to the game.
- Getting the quantifiable data and analysing the project further.

Overall, the testing was a success, with many findings which are explained in the next chapter. Unfortunately, the testing group was relatively small, not providing as much data as could be possible if there was more time available for testing.

6.4. Conclusion

The implementation of the project took a considerable amount of time, which was not expected in the beginning of the project. The previous stages where the functional requirements were established was exceptionally helpful while setting the sprint goals of this project, however even with the clear goals it took longer than was expected.

One of the main difficulties with this project was using the Godot engine for the first time. It is an engine which provides clear documentation, but even with that the

development environment is unfamiliar enough to provide a challenge, making the progress slower for the first few sprints.

Overall, the project achieved most of the requirements in its implementation despite a few minor flaws: the user interface management system is tricky to use in Godot, not allowing for easily scalable on-screen elements. This caused multiple issues after creating the first builds of the project, when the skill container was shifted slightly despite it not being the case beforehand. Despite the encountered difficulties, the project reached the best state it could in the given amount of time.

7. Testing

7.1. Introduction

The testing is a key aspect of this research, aiming to answer the questions asked in the very beginning: how do the user mechanics and inputs affect the perception of game design?

To answer this question, the testing involves a mix of approaches:

- **Personal:** asking the user testers for the opinions on the project, writing down the responses to each element of the game.
- **Quantifiable:** recording the data using the script inside the game itself, which gathers information on how fast the user goes from point A to point B, which methods are being used and how this data compares to the users who use a different set of tools provided.

In the end of this chapter, the main question of the project must be answered: do the input systems and game mechanics matter enough to strongly affect the user's experience with the game?

7.2. Functional Testing

The initial testing was done during the early development of Shape Fight. It was done by a tester outside of development, providing the clearest possible results for the further improvements:

- The tested build had no character animations, which made it confusing to play without the proper action feedback;
- The tested build lacked the skill descriptions, which caused the player to become confused about the purpose of the skills;
- The tested build had issues with the camera, providing a very smooth but unpleasant camera movement.

These issues were handled before the final testing, and the testing build had minimal functional issues.

7.3. User Testing

During the user testing, the users were separated into two groups:

- Mouse input testers: the project was tested using the mouse, navigating the user interface using clicks and hovering over buttons to receive more information.
- Keyboard input testers: the project was tested by the group using the keyboard to navigate the menus, using the arrow keys to select the actions.

7.3.1. Mouse input testing

During the testing, two major issues were found by the players using the mouse controls:

- The “skip” and “back” buttons were not selectable, which is not intended.
- The user interface elements failed to scale properly, causing confusion when selecting the skills.

Despite these issues, the users enjoyed navigating the menus using the mouse. It was clear where the users were looking when using the cursor, navigating the interface by hovering over buttons and looking through menus.

The obvious upside of the mouse control input was the ease of use: when using the mouse, it was obvious how to interact with the menus, causing the players to quickly adapt to the flow of the game. The user testers expressed very little confusion when interacting with the game this way.

The downside of the mouse control input is the speed of getting through the menus: unlike the keyboard layout, this input requires the players to manually drag the mouse around the screen. This is not a major downside, but the difference becomes more apparent when compared to the keyboard input testing.

7.3.2. Keyboard input testing

During the testing, three major issues were found when using the keyboard controls:

- The lack of skill description when focusing on the skill buttons.
- If the buttons are unfocused at any stage of the game, they do not gain focus again until a focused button gets instantiated.
- The win screen menu button was not focused, forcing the users to use the mouse to progress.

These issues likely appeared due to the lack of time to functionally test the keyboard input system. There is only one difference between this interface and the mouse interface: the “back” and “skip” buttons were selectable, unlike with the previous interface.

The obvious upside of the keyboard input system is its speed compared to the mouse: instead of manually dragging the mouse to select an action, a single button input can be used.

The downside of the keyboard input system is its initial confusion: unlike the mouse control input, it is initially less obvious how to navigate the menus, causing some confusion. This is quickly resolved when the players spend a little longer using the input system, making it faster and more efficient than the mouse selection.

7.3.3. Quantifiable data

To further provide evidence of the impact of game controls, data was collected and analysed from the JSON files. The following Figure 42 shows a snippet of gathered information, providing more context as to what data will be analysed.

level	camera_mode	ui_moc	killed_l	name	time (ms	turn_ind
1	0	0	Cube	Planet Cluster	13007	0
2	0	0	Cube	The Spider	58230	0
2	0	0	Cube	Planet Cluster	89727	0
3	0	0	Cube	The Spider	29266	0
3	0	0	Cube	The Spider	66923	0
3	0	0	Pyramid	OrboRage	113894	1
1	0	0	Cube	Planet Cluster	21658	0
2	0	0	Pyramid	The Spider	24373	1
2	0	0	Pyramid	Planet Cluster	42100	1
3	0	0	Pyramid	The Spider	20639	1
3	0	0	Cube	The Spider	36857	0
3	0	0	Pyramid	OrboRage	59051	1
1	1	1	Cube	Planet Cluster	15658	0
1	0	0	Cube	Planet Cluster	21658	0
2	0	0	Pyramid	The Spider	24373	1
2	0	0	Pyramid	Planet Cluster	42100	1
3	0	0	Pyramid	The Spider	20639	1

Figure 42: Data Snippet

The most notable testing data is the time taken to defeat each of the enemies. This data is of course mainly affected by the amount of health each enemy has and each level design, which is exactly why the data will be compared using the same levels.

The following Figure 43 shows the average time it takes to defeat each type of enemy on their levels.

level	ui_mode	name	avg.time
1	0	Planet Cluster	19411.17
2	0	Planet Cluster	54456.83
2	0	The Spider	40479.17
3	0	The Spider	36998.25
3	0	OrboRage	53798.83
1	1	Planet Cluster	14561.25
2	1	Planet Cluster	27698.33
2	1	The Spider	5551.333
3	1	The Spider	18801
3	1	OrboRage	68991.5

Figure 43: Average Time Data

This data showcases a slight difference in the time it takes to defeat the enemies. It becomes more obvious when this data is used in a graph in the figure.

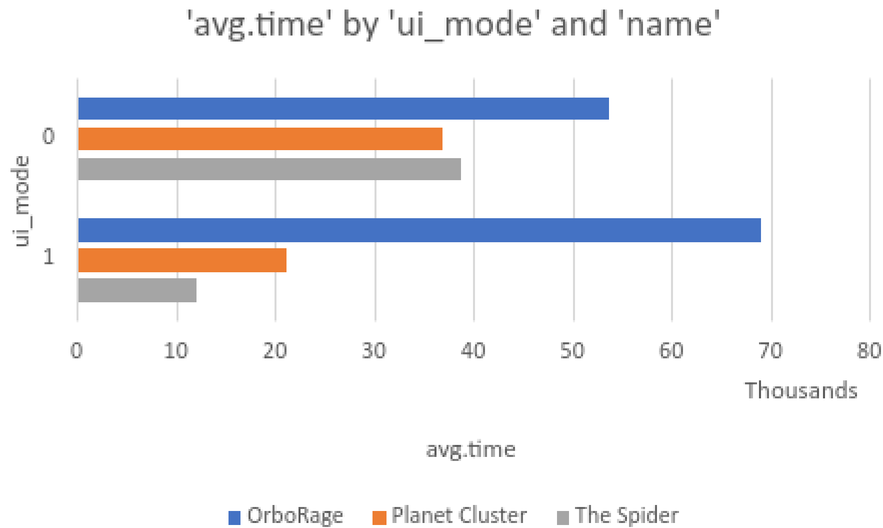


Figure 44: Visualised Data

When comparing the data on Figure 44, something becomes obvious: the enemies take much faster to defeat using the keyboard controls, with The Spider enemy type showcasing a dramatic difference.

The main downside of this study is the lack of participants: there were very few participants who played the game, causing the dataset to be very small. Despite that, this shows an obvious change which could further be researched when studying the different types of user input.

7.4. Conclusion

In conclusion, this chapter proves that there is a difference in the way users play the game when faced with different tools to interact with it.

Due to the lack of participants, this testing would require a larger group of people before creating a more concrete conclusion. However, these results still show an interesting difference which can be studied further in the future.

Overall, this testing was a success, providing various kinds of data on the topic of game inputs.

8. Project Management

8.1. Introduction

The project was managed using two kinds of tools: a writing tool and a visual tool. These tools in combination with the Scrum frameworks aided in organising the project development, providing different ways to gather and structure information, to-do tasks and notes.

8.2. Project Management tools

8.2.1. Joplin

Joplin is a note-taking application with the focus on structuring notes and formatting the text.
The software provides the user with the ability to create and name notes, organising them by name or date. This tool also has the formatting which allows to create a to-do list. These lists have proven to be a major help especially when each of the sprint goals was decided.
The following Figure 45 showcases Joplin’s interface.

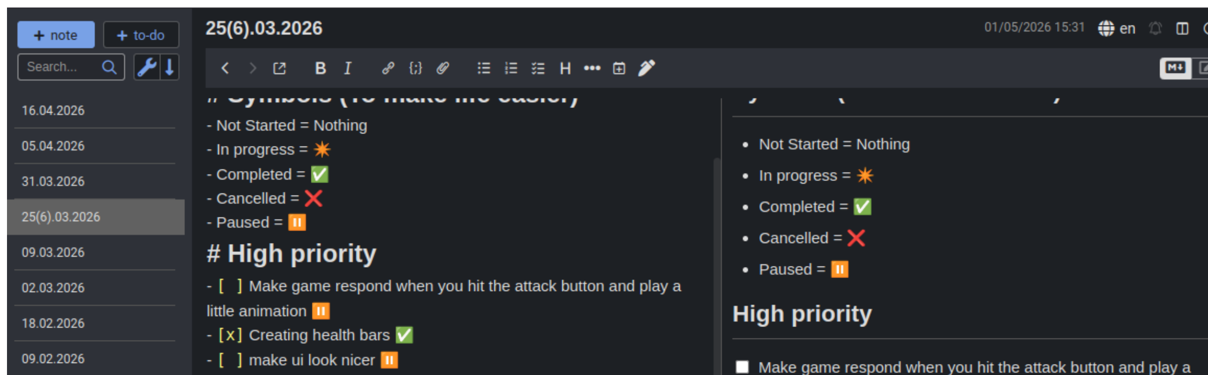


Figure 45: Joplin

8.2.2. Miro

Miro is a tool which works similarly to a whiteboard, allowing to gather and organise visual references, documents and useful links. Unlike Joplin, its main appeal is the lack of set formatting, allowing the user to organise the board however is seen fit.
The following Figure 46 showcases the Miro board used to gather resources used to develop the project later.

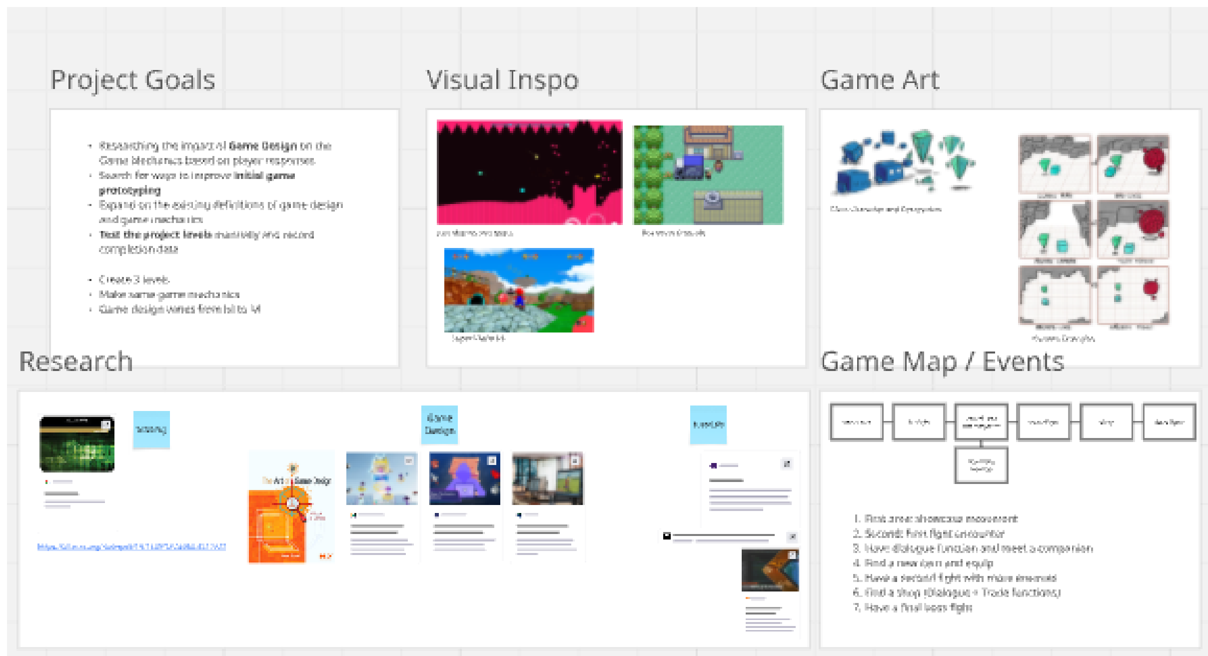


Figure 46: Miro

8.3. Conclusion

In combination with the Scrum methodology and different organisational tools, the project management has proven to be simple and straightforward.

The appeal of using different tools is the ability to gather different types of information in one place, which quickly becomes a necessity when developing a project of a larger scale. When the tools don’t allow for structured organisation, there is a risk to lose the previously gathered information.

9. Conclusion

9.1. Summary

To reflect on this project, it has been a very interesting experience. For me it quickly became a game of balancing each aspect of development, otherwise I would either spend too much time researching the project over implementing it, or implementing it for so long there has been little time to test.

This project is something I am very proud of, despite the constant battle of balance. In the beginning, there was a lot of uncertainty in how this project would achieve its goals, which is why I appreciated working with a supervisor. In many cases the initial project idea was too ambitious: it either aimed to create a much larger project or aimed to achieve something which I lack the skill to do.

The initial idea of this project was to create an entire game for the testing purposes using the Unreal Engine. Of course the idea was quickly changed in favour of a simpler engine and a smaller project: otherwise I wouldn't have the time nor knowledge to implement all the planned aspects of the game.

In the end, this project has taught me to perfect the smaller games instead of trying to build something too big to handle. This is an important lesson for a developer to learn, since great ambitions can both guide us to growth and make us become stuck in one place for far too long without a way to get out.

9.2. Future Improvements

In the future, I would love to be able to polish this project and use the written systems in other games. This project shows a lot of potential without the perfect implementation, although it has reached the best state I could get it to within this time period.

Along with the further development, it would be interesting to test this concept in a larger group of people, collecting more measurable data on how different inputs affect the player experience. It would be interesting to see how different these inputs can become from the initial idea, possibly creating something completely original and seeing how people would react to it.

Overall, it has been a pleasure to work on this project and I am looking forward to improving it even more.

References

Brazie, A. (2022, April 2). *Video Game Mechanics: A Beginner's Guide (with Examples)*. Game Design Skills. https://gamedesignskills.com/game-design/video-game-mechanics/

Delfino, B. (2025, January 20). *UX and UI in game design: exploring HUD, inventory, and menus*. Medium. https://medium.com/@brdelfino.work/ux-and-ui-in-game-design-exploring-hud-inventory-and-menus-5d8c189deb65

First Videogame. (n.d.). Guinness World Records. https://www.guinnessworldrecords.com/world-records/first-computer-game

Game Mechanics 101: What Every Aspiring Game Designer Should Know. (2025, September 5). Champlain.edu. https://online.champlain.edu/blog/game-mechanics-101

Katarzyna Witkowska. (2025, July 11). *What Is a Turn-Based Game?* G2A News; G2A.com. https://www.g2a.com/news/glossary/what-is-a-turn-based-game/

Koster, R. (2014). *A theory of fun for game design* (pp. 20–25). O'reilly Media Inc.

Schardon, L. (2021, September 15). *What is Game Design: The Creative Side of Development*. GameDev Academy. https://gamedevacademy.org/what-is-game-design/

Schell, J. (2019). *The Art of Game Design: A Book of Lenses* (pp. 24–36). Taylor & Francis, a Crc Title, Part of the Taylor & Francis Imprint, a Member of the Taylor & Francis Group, the Academic Division of T&F Informa, Plc. https://www.inventoridigiochi.it/wp-content/uploads/2020/07/art-of-game-design.pdf

Scrum.org. (2024). *What is Scrum?* Scrum.org. https://www.scrum.org/learning-series/what-is-scrum/

Sicart, M. (2011). Defining Game Mechanics. *Game Studies*, *11*(3). https://gamestudies.org/1103/articles/sicart