ROYAL DEFENDERS

BY

NAME:	ID
ALI HABIBULLAH MERAJ	1945958
FERAS AL-HILABI	1945814
KHALID AL-GHAMDI	1936811

DEPARTMENT OF COMPUTER SCIENCE

KING ABDULAZIZ UNIVERSITY

[Winter 2023]

ROYAL DEFENDERS

BY

NAME:	ID
ALI HABIBULLAH MERAJ	1945958
FERAS AL-HILABI	1945814
KHALID AL-GHAMDI	1936811

THIS REPORT IS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF BACHELOR OF SCIENCE IN COMPUTER SCIENCE

SUPERVISED BY

DR MOHAMED DAHAB

DEPARTMENT OF COMPUTER SCIENCE

FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY

KING ABDULAZIZ UNIVERSITY

JEDDAH – SAUDI ARABIA

[FEBRUARY 2023]

Declaration of Originality by Students

We hereby declare that this project report is based on our original work except for citations and quotations, which had been duly acknowledged. We also declare that it has not been previously and concurrently submitted for any other degree or award at KAU or other institutions.

NAME:	ID	Date	Signature
ALI HABIBULLAH MERAJ	1945958		
FERAS AL-HILABI	1945814		
KHALID AL-GHAMDI	1936811		

Declaration of Originality by Supervisor

I, the undersigned hereby certify that I have read this project report and finally approve it with recommendation that this report may be submitted by the authors above to the final year project evaluation committee for final evaluation and presentation, in partial fulfillment of the requirements for the degree of BS Computer Science at the Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah.

Dr. Mohamad Dahab

.

Abstract

In video games, artificial intelligence (AI) is often used to enable non-player characters (NPCs) to find paths to their goals. There are many different pathfinding algorithms that can be used to accomplish this, each with its own set of advantages and disadvantages. For our project, we chose to focus on three specific algorithms: Breadth First Search (BFS), Uniform Cost Search (UCS), and A* Search. These algorithms were implemented in the context of a tower defense game (TDG) called Royal Defenders, which we developed using the Unity game engine. In this game, the player builds towers that serve as obstacles for the NPCs to navigate around as they search for a path to the goal. The player can choose to use any of the three pathfinding algorithms to control the behavior of the NPCs. As the player builds towers, the NPCs are notified and must find a new path in real time. Our project showed that different search algorithms can have a significant impact on the gameplay experience, and that game designers must carefully consider which algorithms to use to create a fun and engaging game. We found that A* search performed well when the heuristic function was implemented in an efficient and smart manner, outperforming the other two algorithms in terms of both cost and correctness.

نبذة مختصرة

رويال ديفنديرز: في العاب الفيديو يتم استخدام الذكاء الاصطناعي لإيجاد مسارات مختلفة للوصول إلى هدف معين. للعثور على مسار صحيح الى الهدف يجب على الوكيل الذكي التنقل داخل منطقة البحث باستخدام واحدة من عدة خوارزميات لتحديد طريقة الوصول إلى الهدف .هناك العديد من المجالات التي يمكن فيها استخدام خوارزميات تحديد المسار هذه؛ بالنسبة لمشروعنا، اخترنا بيئة العاب الفيديو. سوف يكون هذا المشروع عبارة عن لعبة دفاع ضد الأعداء باستخدام الأبراج (TDG) تسمى رويال ديفنديرز، وسوف يتمكن اللاعب من تجربة ثلاث خوارزميات لتحديد المسار وهي: خوارزمية بحث العرض أولا، وخوارزمية البحث بالتكلفة الموحدة، وخوارزمية بحث *A. يمكن استخدام محركات تطوير الالعاب لتطوير العاب الفيديو، وقد تم انشاء هذا المشروع باستخدام محرك تطوير العاب ثلاثي الأبعاد(3D) يسمى Unity. يعد بناء لعبة الفيديو مشكلة تتطلب المرونة والتكيف في مراحل مختلفة من التطوير، لذلك تم اتباع منهجية Sprint بدقة لبناء هذا المشروع. تسمح اللعبة للاعب ببناء ابراج مختلفة تكون بمثابة عقبات امام الوكيل الذكي للتنقل حولها. عندما يتم بناء برج، يتم إبلاغ الوكيل الذكي لكي يجد مسارا جديدا. أظهر هذا المشروع ان خوارزميات البحث المختلفة لها مزايا وعيوب يجب ان يوازنها مصممو الالعاب لجعل اللعبة ممتعة ومسلية بأكثر قدر ممكن .أظهر هذا المشروع أيضا انه عندما يتم تنفيذ الوظيفة الإرشادية (Heuristic Function) للوكيل الذكي بطريقة فعالة وذكية، فإن خوارزمية *A تتفوق على الخوارزميتين الأخريين.

Table of Contents

Declaration of Originality by Students ..... **ii**

Declaration of Originality by Supervisor ..... **iii**

Abstract ..... **iv**

نبذة مختصرة ..... **v**

List of Tables ..... **ix**

List of Figures ..... **x**

1. Introduction: **1**

1.1. Introduction 1

1.2. Problem statement: 1

1.3. Motivation: 2

1.4. Scope of deliverables 2

2. Background and Review of related work **4**

2.1. Background: 4

2.2. Literature Review: 4

2.2.1. A study on Bee Algorithm and A* Algorithm for Pathfinding in Games: 4

2.2.1.1. PROMPT Criteria: 7

2.2.2. Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding: 7

2.2.2.1. PROMPT Criteria: 9

2.2.3. Comparison of A* and Dynamic Pathfinding Algorithm with Dynamic Pathfinding Algorithm for NPC on Car Racing Game: 9

2.2.3.1. PROMPT Criteria: 11

2.2.4. Hybrid Pathfinding in StarCraft 11

2.2.4.1. PROMPT Criteria: 14

2.3. Searching the Internet: 14

2.4. Playing Games 15

2.5. Summary of the two methods: 17

3. ARCHITECTURE, DESIGN AND METHODOLOGY **19**

3.1. METHODOLOGY 19

3.2. ARCHITECTURE AND DESIGN 20

3.2.1. Functional Requirements: 20

3.2.2. Non-Functional Requirements: 21

3.3. Use Cases: 22

3.3.1. Use case, Pick tower type: 22

3.3.1.1. Actors: 22
3.3.2. Use Case, Build tower: 23
3.3.2.1. Actors: 23
3.3.3. Use Case, Enemy spawn from object pool: 24
3.3.3.1. Actors: 24
3.3.4. Component Diagram: 25
3.3.5. Pathfinidng component 27
3.3.6. User Interface component 28
3.4. Activity Diagrams: 29
3.4.1. Activity diagram, Build a tower: 29
3.4.2. Activity diagram, Path finding: 30
4. Implementation: 32
4.1. Languages: 32
4.1.1. C#: 32
4.2. Software: 32
4.2.1. Unity: 32
4.2.2. Blender: 33
4.2.3. Github 34
4.3. Backend: 34
4.3.1. AbilityBtnCode: 34
4.3.2. AudioManager 34
4.3.3. AudioOptionsManager 35
4.3.4. Bank 35
4.3.5. CoordinateLabeler 35
4.3.6. Enemy 35
4.3.7. EnemyHealth 36
4.3.8. EnemyMover 36
4.3.9. GridManager 36
4.3.10. mouseDrawCircle 36
4.3.11. NodeClass 36
4.3.12. ObjectPool 37
4.3.13. ParticleHandler 37
4.3.14. Pathfinding 37
4.3.15. PauseMenu 38
4.3.16. PlayerController 38

4.3.17. RetryMenu....39
4.3.18. ScoreUI....39
4.3.19. SettingsMenu....39
4.3.20. SpeedChanger....40
4.3.21. TagetLocator....40
4.3.22. Tile....42
4.3.23. Tower....43
4.3.24. TowerBtnCode....43
4.3.25. UpgradeUI....43
4.3.26. VictoryMenu....43
4.3.27. WaveManager....43
5. Experiments and results....45
5.1. Software results:....45
5.1.1. Main menu:....45
5.1.2. Settings menu....45
5.1.3. Level select menu....46
5.1.4. In game menu:....47
5.1.5. Pause menu.....48
5.1.6. Gameplay in progress:....49
5.2. Testing & Integration....50
5.2.1. Types of testing performed:....50
5.2.1.1. White box testing....50
5.2.1.2. Black box testing:....50
5.2.2. Conclusion of the testing phase:....50
6. Conclusion and Plans for the Future....52
6.1. Conclusion:....52
6.2. Future work....52
7. BIBLIOGRAPHY:....54

List of Tables

Table 2-1 Advantages and Disadvantages of different map representations in pathfinding. 6
Table 2-2 PROMPT Evaluation of A study on Bee Algorithm and A* Algorithm for Pathfinding in Games 7
Table 2-3 PROMPT Evaluation of Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding 9
Table 2-4 PROMPT Evaluation of Comparison of A* and Dynamic Pathfinding Algorithm with Dynamic Pathfinding Algorithm for NPC on Car Racing Game 11
Table 2-5 Comparison results between the algorithms in different maps 13
Table 2-6 PROMPT Evaluation of Hybrid Pathfinding in StarCraft 14
Table 2-7 Comparison of information gathering methods. 17

List of Figures

Figure 2-1 Pathfinding in a grid-based environment 5
Figure 2-2 A single synthesis trial 8
Figure 2-3 Grid representation of the map 10
Figure 2-4 Hybrid navigation system 12
Figure 2-5 Bloons enemy types 15
Figure 2-6 Total War Pathfinding. 16
Figure 3-1 Use case: pick tower type 22
Figure 3-2 Use case: build tower 23
Figure 3-3 Use case: spawn from object pool 24
Figure 3-4 Component Diagram Short 26
Figure 3-5 Pathfinding component details 27
Figure 3-6 User Interface Component 28
Figure 3-7 Activity diagram, Build a tower 29
Figure 3-8 Activity diagram, Path Finding 30
Figure 5-1 Main menu 45
Figure 5-2 Settings menu 46
Figure 5-3 Level Select menu. 47
Figure 5-4 In game interface 48
Figure 5-5 Pause menu. 48
Figure 5-6 Tower attacking. 49

CHAPTER 1
INTRODUCTION

1. Introduction:

This section will go over the history of the project, what it hopes to achieve, and what it will include once completed:

1.1. Introduction

Welcome to our project on the application of artificial intelligence in video games. Specifically, we focus on the use of pathfinding algorithms in a tower defense game called Royal Defenders. Using the Unity game engine, we implemented three different algorithms - Breadth First Search (BFS), Uniform Cost Search (UCS), and A* Search - for the non-player characters to navigate around the player-built towers and reach their goal. Through gameplay testing, we discovered that the choice of algorithm can greatly impact the player experience and the performance of the NPCs. Our results showed that A* search, when implemented with an efficient and smart heuristic function, performed the best in terms of cost and correctness. Join us as we delve deeper into the fascinating world of AI in video games.

1.2. Problem statement:

In many problems, searching algorithms are the backbone of AI. Some of us are familiar that search algorithms fall into one of 2 categories, uninformed search algorithms, and informed search algorithms.

Uninformed search algorithms are the weaker of the two; examples include breadth first search (BFS), which expands its search space horizontally without considering the cost of any search node. Another example is uniform cost, which works similarly to BFS but considers the cost of each node that is expanded, prioritizing nodes with low cost.

Informed search algorithms are the smarter of the two; they consider a heuristic value, which varies from implementation to implementation, but often, the heuristic value is only the Euclidean distance between two points in Euclidean space; the Euclidean distance between two points in Euclidean space is the length of a line segment between the two points.

A* search, which is one of the more popular search algorithms in video games (Rafiq et al., 2020), is an example of an informed search algorithm; it works similarly to Uniform Cost Search, but it also includes a heuristic value. Another example is the minimax search algorithm, which is the most used search algorithm when developing an AI that plays chess or other similar games (CS221, n.d.) and assumes that the opponent will always make the best possible move.

Search algorithms are one of the most important building blocks in video game AI (Sabri et al., 2018); the choice of a searching algorithm can have a significant impact on how much the player enjoys the game and how well the enemy simulates human intelligence.

This project's primary goal is to compare search algorithms in terms of cost and correctness. The strategic game "Royal Defenders" serves as a case study for the

comparison, and because our game's theme is fantasy with some cartoonish elements, it will target all players over the age of ten.

1.3. Motivation:

The motivations behind this project are multifaceted, but they all center around the importance of selecting the right searching algorithm in video game AI development:

1. To demonstrate the importance of choosing the right searching algorithm in video game AI development.
2. To evaluate the performance of different search algorithms in terms of cost and correctness.
3. To provide insights for game designers on how to implement pathfinding algorithms in their games to enhance the gameplay experience.
4. To contribute to the broader field of AI and game development by sharing the findings of the project.
5. To showcase the potential of AI in video games and its potential impact on the player's experience.
6. To experiment and compare the performance of different algorithms in a real-world game scenario.
7. To educate players and game developers on the importance of AI in video games.
8. To compare the performance of different algorithms in a real-world game scenario, and gain insights that can be used in other games or other fields.
9. To develop a fun and engaging game for the target audience using the latest AI techniques.

1.4. Scope of deliverables

The project will focus on developing a tower defense video game using the Unity game engine. The game will be designed to be played on a variety of popular desktop platforms, including Windows, Linux, and Mac operating systems. This will allow for a broad range of players to experience the game and provide valuable feedback on the performance of the different pathfinding algorithms implemented. The game will be developed using industry standard tools and techniques, ensuring that it is of high quality and performs well on a variety of hardware configurations. The game will be designed to be fun and engaging for players, while also providing an opportunity to experiment with different pathfinding algorithms in a real-world scenario.

CHAPTER 2
BACKGROUND AND REVIEW OF RELATED WORK

2. Background and Review of related work

This section of the project will provide a comprehensive overview of the background and preparation that went into the development of the game. It will begin by discussing the current state of the art in the field of video game AI and pathfinding algorithms. This will include an examination of the most used algorithms, their strengths, and weaknesses, and how they have been applied in previous games. This background information will provide context for the project and show the need for further research in this area.

2.1. Background:

To get ready for our project to be accepted, we built and showed a demo to our supervisor Dr Mohammad Dahab. After getting his initial approval, we knew that the next step was to start researching the various search algorithms that we would use in our project. We knew that this would be a crucial aspect of our work, as the choice of algorithm would have a significant impact on the performance and effectiveness of our game.

2.2. Literature Review:

This section will conduct a comprehensive review of four research papers related to the topic at hand. The papers will be evaluated based on their methodology, results, and overall contribution to the field of study. The review will also provide a summary of the key findings and insights gained from each paper, as well as any potential areas for future research or further exploration.

2.2.1. A study on Bee Algorithm and A* Algorithm for Pathfinding in Games:

The authors begin the paper by providing a brief history of games and what they have evolved into. They also state that some games improve a person's ability to think logically and creatively.

They then go on to say that pathfinding in video games is almost a requirement to simulate human-like thinking, and they go on to list other applications for pathfinding, such as commercial games and robot navigation (Sabri et al., 2018).

The authors discuss what pathfinding is and what problems you might encounter when attempting to program a pathfinding algorithm in a video game environment in the second part of their paper, stating that some of the problems include memory allocation and time taken (Sabri et al., 2018).

They also mention that dynamic pathfinding in a video game environment has piqued the interest of researchers in recent years. They demonstrate in the figure below that in a grid-based environment, finding the shortest path can be accomplished in two ways.

The first path, shown in yellow, can move in all directions, including diagonally, whereas the second path, shown in green, can only move in two directions: horizontal and vertical.

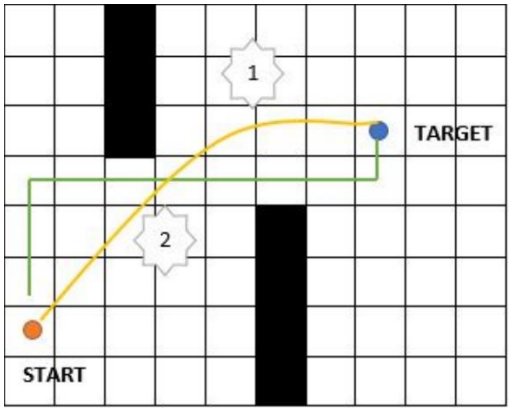


Figure 2-1 Pathfinding in a grid-based environment

Source: (Sabri et al., 2018).

In the following section, the authors defined what a heuristic is, gave an example of an algorithm that uses a heuristic function called A*, and explained what the limitations of A* are, primarily that A* becomes more expensive in terms of CPU time required in large map sizes.

After discussing the shortcomings of A*, they discussed how to overcome them, which included developing a metaheuristic (meta in this context refers to above or higher level) (Sabri et al., 2018). They go on to claim that current metaheuristic algorithms nearly invariably draw their inspiration from naturally occurring events (Sabri et al., 2018).

They then demonstrate the Bee Method, a metaheuristic algorithm proposed by S. Nakrani and C. Tovey in 2004(Nakrani & Tovey, 2004). This algorithm performs a neighborhood search combined with a random search; as an aside, this algorithm can be used to find a good solution to the traveling salesman problem.

The Bee algorithm is severely limited by the number of parameters that must be considered, including the number of scout bees, the number of sites chosen, the number of best sites, the number of bees recruited for best sites, the number of bees recruited for other selected sites, and the initial size of patches (Sabri et al., 2018).In the section that follows, the authors contrasted various pathfinding map representations and discussed their advantages and disadvantages:

Table 2-1 Advantages and Disadvantages of different map representations in pathfinding.

Map representation	Advantages	Shortcomings
Waypoint	• Suitable used in small search space and static map environment. • Traditional approach used to navigate path	• Not suitable used in dynamic map environment (obstacles exist) • Expensive time consuming when in large map environment
Navigation Mesh	• Better than waypoint graph technique. • Guarantee to find optimal path. • Reducing the number of nodes used in map environment. • Suitable in static and dynamic map environment	• Have limitation when in large map environment and dynamic environment. • Computational cost increases when creating and updating navigation mesh graph. • It may cost time and memory consuming when the location of obstacle changes.
Grid	• Simple and easy to understand in a game environment. • Cheap in updating graph. • Very suitable in static and dynamic environment • Do not required complicated node for rebuilding process	• Required a grid representation of each map. • Inaccuracy in minimum distance.

The authors then go on to discuss how and why a grid-based system is used in games, as well as how to build one using different types of tiles, such as squares, octiles, and hexagons. They go on to say that one advantage of using grid-based graphs in video games is that the agent can move freely along the x-y axis (Sabri et al., 2018).

The authors continue their papers by comparing how the A* algorithm and the Bee algorithm behave in the same environment with and without grid obstacles. According to their findings, the Bee algorithm is superior in optimizing memory allocation with and without obstacles, but in terms of time taken, the A* algorithm is faster without obstacles and the Bee algorithm is faster with obstacles (Sabri et al., 2018)

2.2.1.1. PROMPT Criteria:

Table 2-2 PROMPT Evaluation of A study on Bee Algorithm and A Algorithm for Pathfinding in Games*

PROMPT Criteria	Notes
Presentation	The presentation of the paper was excellent; the only flaw was the presence of some grammatical errors, which can be overlooked because not everyone is a native speaker.
Relevance	The information presented in this paper was relevant to my project; it informed me about what map presentation I should use and what algorithms to use in the project for pathfinding.
Objectivity	Because this paper is only a comparison of algorithms, the authors' assessments are objective. They mentioned the testing environment in both cases, as well as the number of times the test was completed to ensure objectivity.
Method	There was no data collection.
Provenance	All relevant papers were correctly referenced and provided at the end of the paper.
Timelines	The paper was published in 2018, but the results and testing dates are not mentioned, which is a shortcoming of this paper.

2.2.2. Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding:

Vadim Bulitko, Sergio Poo Hernandez and Levi H. S. Lelis consider the problem of synthesizing heuristic functions for the standard A*-based heuristic search (Bulitko et al., 2021).

They aim to create a synthesis method that requires little human intervention, is simple to implement, and yields human-readable heuristics with a small memory footprint (Bulitko et al., 2021).

The authors mention that memory-based heuristics have already been developed; they do work, but they have a significant memory footprint, lack portability (cannot be used as is on a different map), and can be difficult for humans to read(Bulitko et al., 2021).

The authors use a similar method and mention that their synthesized method can be used with subgoals (Bulitko et al., 2021). Heuristic functions can be improved by computing them with a closer subgoal instead of the original goal.

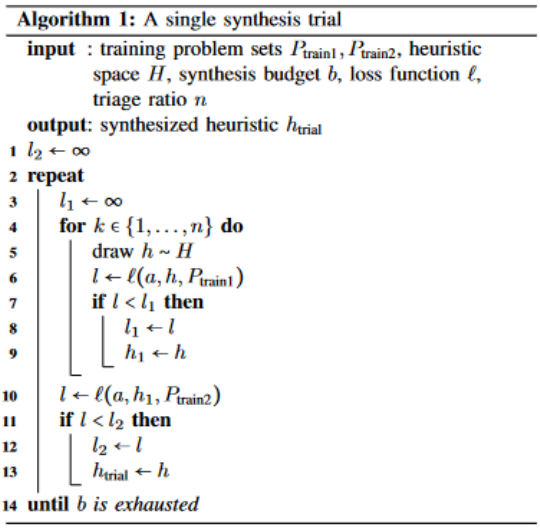
Another method they mention for computing high-performance heuristics is to embed the structure in the original search graph in a Euclidean space, then use the Euclidean distance between the embedded vertices as a heuristic; however, the embedding must be performed in each map (Bulitko et al., 2021).

The authors approach is letting synthesized heuristics take advantage of specifics of a given map by synthesizing heuristics on a per map basis (Bulitko et al., 2021).

This synthesis method is simple which makes it easier for game development and can be easily optimized to give better results using genetic algorithms.

In their method they pick a random heuristic (h) from a set of heuristics (H) and use a series of large training sets of search algorithms that progressively get larger to evaluate the random samples (h).

To not waste time on computing the algorithm will try to quickly filter out the low performing heuristics without wasting time on computing how bad their performance is, it does that by running the randomly selected heuristic (h) with (a) algorithm on a small training set (P_train1) this is done n times then the heuristic (h_1) with the lowest loss is selected, the selected heuristic (h_1) is then tested with a larger training set if they lose is lower than the formally best heuristic then (h_1) takes its place all this is repeated until (b) is exhausted and the heuristic with the least loss is returned.



Algorithm 1: A single synthesis trial

input : training problem sets $P_{\text{train1}}, P_{\text{train2}}$, heuristic space H, synthesis budget b, loss function ℓ, triage ratio n

output: synthesized heuristic h_{trial}

$l_2 \leftarrow \infty$
repeat
  $l_1 \leftarrow \infty$
  for $k \in \{1, \ldots, n\}$ **do**
    draw $h \sim H$
    $l \leftarrow \ell(a, h, P_{\text{train1}})$
    if $l < l_1$ **then**
      $l_1 \leftarrow l$
      $h_1 \leftarrow h$
  $l \leftarrow \ell(a, h_1, P_{\text{train2}})$
  if $l < l_2$ **then**
    $l_2 \leftarrow l$
    $h_{\text{trial}} \leftarrow h$
until *b is exhausted*

Figure 2-2 A single synthesis trial

Source: (Bulitko et al., 2021).

2.2.2.1. PROMPT Criteria:

Table 2-3 PROMPT Evaluation of Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding

PROMPT Criteria	Notes
Presentation	The paper was presented in a very informative manner; the algorithm and figures provided were useful, though some figures were a little too small to see. The chapters were excellent and contributed to the overall readability of the paper.
Relevance	The algorithm they use is designed to create maps, most likely for procedurally generated maps. We don't have procedurally generated maps, so the path finding algorithm is for the enemies to use to find the goal while avoiding obstacles.
Objectivity	The paper describes an easy and reliable method for creating maps and procedurally generated content; this is a useful tool that many game developers can use.
Provenance	All relevant papers were correctly referenced and provided at the end of the paper; it was published in IEEE.

2.2.3. Comparison of A* and Dynamic Pathfinding Algorithm with Dynamic Pathfinding Algorithm for NPC on Car Racing Game:

Yoppy Sazaki, Hadipurnawan Satria, and Muhammad Syahroyni compared the dynamic pathfinding algorithm to the A* algorithm in a racing game environment where the opponent is Non-Playable Characters (NPC).

The authors claim that the A* algorithm outperforms the dynamic pathfinding algorithm when it comes to finding the shortest path to the goal. However, in terms of avoiding dynamic obstacles on the path, which are common in racing games, the dynamic pathfinding algorithm outperforms the A* algorithm (Sazaki et al., 2017).

The paper then compares the combined capabilities of the dynamic pathfinding algorithm and the A* algorithm with the dynamic pathfinding algorithm used in the NPC. This was done to obtain a better pathfinding algorithm for the NPC.

The authors then discuss the limitations of this comparison (Sazaki et al., 2017) to only focus on pathfinding algorithms, including the following:

1. The NPC moves at a consistent rate.
2. The distance between the collision point with the car and the obstacle position on the track is the primary data used in the paper.
3. The game employs three-dimensional graphics.

The A* algorithm was used by the authors to find the shortest route on the racetrack. A grid of x and y coordinates covering the entire path is required for route search. Each grid cell's midpoint will be a node that will determine whether the node is in the walkable area or not. The A* algorithm will then find the shortest path from the start node to the finish node via the nodes within the walkable area (Sazaki et al., 2017).

They used the dynamic pathfinding algorithm to avoid obstacles in the path. DPA employs two collision points in front of the car object. Point collision detection is used to detect obstacles to the right and left of the car's front end. If both points detect barriers at the same time, it means that the obstacle is directly in front of the car object. Under these conditions, the collision points will move 45 degrees to the right and left of the car to detect obstacles on the right and left sides of the car object (Sazaki et al., 2017).

The paper then discusses the details for the combined A* and dynamic pathfinding algorithms, which are based on the benefits of each algorithm. When the game begins, the NPC will take the shortest route possible based on the results of the A* algorithm and will use a dynamic pathfinding algorithm to detect obstacles along the way. When the NPC detects an obstacle, it will use the dynamic pathfinding algorithm to avoid it. If the obstacle is not detected, the NPC will retrace its steps using the A* algorithm. This process will be repeated until the NPC completes the task (Sazaki et al., 2017).

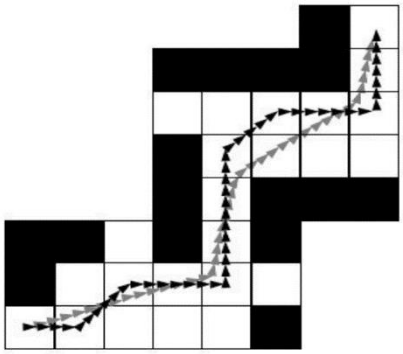


Figure 2-3 Grid representation of the map

Source: (Sazaki et al., 2017).

2.2.3.1. PROMPT Criteria:

Table 2-4 PROMPT Evaluation of Comparison of A and Dynamic Pathfinding Algorithm with Dynamic Pathfinding Algorithm for NPC on Car Racing Game*

PROMPT Criteria	Notes
Presentation	The paper was presented with specific details that assist the reader in understanding the comparison that is being made, and the figures and tables that demonstrated the comparison were simple and easy to understand.
Relevance	The paper's comparison uses the A* algorithm to find the shortest path to the goal, which is the same algorithm we use in our game's hard difficulty. It also employs dynamic pathfinding to find the shortest path while avoiding all obstacles, which in our game are the towers that attack the enemies.
Objectivity	The study discusses the benefits of using A* or dynamic pathfinding algorithms, as well as the benefits of combining them to create a better path finding algorithm that reaches the goal in the shortest amount of time while avoiding all obstacles, which is a common problem in game development.
Provenance	All pertinent papers were properly referenced and included at the end of the paper. The paper was published in IEEE.
Timelines	Paper was published in 2017, all the results of the comparisons were mentioned.

2.2.4. Hybrid Pathfinding in StarCraft

In a Real Time, Strategy game environment, Johan Hagelback investigated how a hybrid pathfinding system compares to a non-hybrid solution for a micro-management system.

The author begins by defining Real Time Strategy (RTS) games. In a nutshell, they are games in which each player begins with a command center and a number of workers. Workers are used to gather one or more types of resources, which are then used to build buildings. Buildings can be used to grow the base, protect it, and create combat units. RTS games frequently feature technology trees where a player can invest resources in unit and/or building upgrades. Each player must make numerous decisions about how to allocate resources. Resources are limited and they are gathered over time (Hagelback, 2016).

The paper then describes the operation of a hybrid navigation system. It is divided into two parts. When there is no enemy unit or building in sight, agents use A* to navigate. A*, on the other hand, is not well suited for positioning units. Agents advance toward the goal without considering how to engage the enemy effectively in a combat situation. To address this, the navigation system switches to flocking with the boids algorithm as soon as an enemy unit or building comes

into view. To keep the squad (the collection of units to which an agent belongs) kept together, agents use a modified version of the boids algorithm to maintain a distance from the opponent that is near to the maximum shooting range of its own weapons. Additionally, agents should avoid running into their own agents and obstacles (Hagelback, 2016).

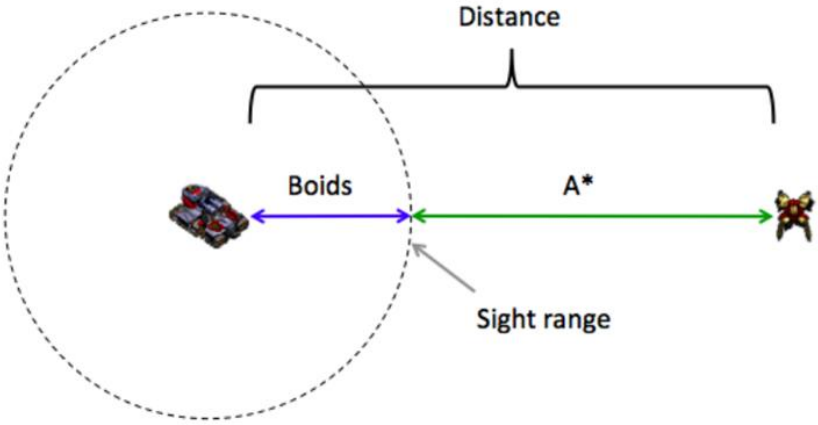


Figure 2-4 Hybrid navigation system

Source: (Hagelback, 2016).

The paper then discusses about boid algorithm implementation. Squads are formed from all agents. Any number of units from different unit kinds may make up a squad. Six rules make up the implementation, and all but the rules prohibiting contact with enemy units have a constant value in their pseudocode. Each rule's dominance is determined by this value. The rules are (Hagelback, 2016):

1. Cohesion:
   A squad's members must always stick together. Single agents that wander off are easy pickings for the adversary.
2. Alignment:
   Individual agents must go roughly in the same direction as each team as they work towards a common objective. As a result, each agent advances in the general direction of the other team members.
3. Goal:
   The commander of the bot chooses where to move each unit. One or more attacking squads are under the commander's command. The opponent is attacked once all these squads have been assembled (all squad members have been formed). The closest enemy building located during scouting is the target of the attack.
4. Separation - own agents:
   Agents shall avoid clashing with other own agents and shall also endeavor to preserve a short gap between each other.
5. Separation - enemy units:

The most crucial boids implementation rule is to keep your distance from opposing units. The rule's objective is to keep agents as close to hostile buildings and units as they may safely be.

6. Separation – terrain:
   Additionally, agents must avoid running into impenetrable terrain.

The study compares two hybrid pathfinding algorithms, one based on boids and the other based on potential fields, in its last section. Not surprisingly, the version without any hybrid pathfinding outperformed the hybrid pathfinder using either potential fields or boids. 175 wins (possible fields) against 178 wins, the difference between the two hybrid pathfinding techniques was barely noticeable (boids). Additionally, there is a noticeable distinction between straightforward two-player games and intricate four-player ones. For the two-player maps, the results for hybrid and non-hybrid pathfinding were nearly comparable, but for the four-player maps, the results were noticeably different. The comparison table is shown below.

Table 2-5 Comparison results between the algorithms in different maps

Map	Games	Wins(A*)	Wins(PF)	Wins(Boids)
Fading Realm	50	96%	98%	98%
Destination	50	98%	100%	100%
Fortress	50	50%	66%	68%
Empire o t Sun	50	72%	86%	88%
Total	200	79%	88%	89%

Source: (Hagelback, 2016).

2.2.4.1. PROMPT Criteria:

Table 2-6 PROMPT Evaluation of Hybrid Pathfinding in StarCraft

PROMPT Criteria	Notes
Presentation	Because the font size was inadequate and the paragraphs lacked organization, the document was badly presented. However, the information was precise, and the article did a nice job of outlining the issue and the work done.
Relevance	We are attempting to incorporate advanced path finding algorithms, therefore the comparison and method for creating a hybrid algorithm described in the study can be helpful in our game. If we had the time to develop it, it might also be helpful in the online mode (Player versus Player).
Objectivity	The comparison between a hybrid system and a non-hybrid system's outcomes are presented in the study. Additionally, it suggests what to do depending on the circumstances, such as whether the map is designed for two players or four players and is intricate.
Provenance	All pertinent papers were properly referenced and included at the end of the paper. The paper was published in IEEE.
Timelines	Paper was published in 2015, all the results of the comparisons were mentioned.

2.3. Searching the Internet:

Entails watching YouTube videos, Looking at academic papers, searching forums and looking at documentation all of these have their own positives and negatives, watching YouTube videos can be a helpful way to get information on how to implements and it being a video helps with visual learning and adds a layer of information which is visual information although sometimes it is hard to find a button and or a specific tab that we need but sometimes these helpful features can lead to harm, a video can be outdated or use a different system so tabs, buttons and other settings can change or be removed in later updates of the program this can lead to confusion and can waste a lot of time if you were already following a video closely and suddenly they use something that is not available anymore, another problem with YouTube videos is sometimes the needed information is not shown.

Looking at academic papers can be helpful when looking for algorithms and systems, but it is not very helpful when using specific programs since academic papers tend to be general and not really software focused so we don’t know if a program using a specific algorithm, or a system can handle it or if it is feasible to implement it onto the software.

Searching forums is a great tool that we mainly use for when there is an error or a problem with the software or code, they provide quick and simple solutions that we can use, but the problem with forums is that sometimes a problem can be too obscure and does not have a solution or sometimes the only solution for the problem is one from an old version of the software and does not work anymore.

Looking at documentation is probably the most useful tool for collecting information available, in it we learn how the software works and we can find new and useful methods that we never thought to use that would work perfectly in our program it makes our work much easier and we constantly learn new things on it, the problem with it would be that it is not very specific and a lot of the information in it can be useless to us so we have to look very carefully when searching so it can take a lot of time and effort to test new methods.

2.4. Playing Games

This means that we played games that are like our game in some way either from pathfinding, tower building, level design and or enemy behavior, the games we used are Bloons Tower Defense, the Total War Games, Minecraft, and Warcraft III.

Bloons is one of the biggest Tower Defense games so we had to look into it although the game lacks a path finding algorithms since enemies move in a pre-set path it makes up for it with the variety of towers and enemies the towers are “monkeys” and the enemies are “bloons” the monkeys need to shoot down the bloons before they reach the end there are many bloon types and many monkey types with their own interesting effects and abilities the game was also made using unity the same engine we’re using.

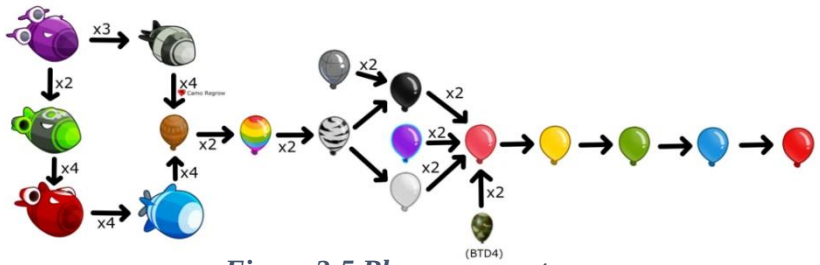


Figure 2-5 Bloons enemy types

Total War games is unique in the way that you can move a large number of characters together without breaking their movement and without them clipping over each other you can also move one character if you wish you can separate a large army and make them move in different directions they will go to the specified goal using the fastest way and while avoiding obstacles like mountains and large holes they can also move through forests they avoid trees and disperse between the trees until they exit Forrest then they will group together to their original formation.

Figure 2-6 Total War Pathfinding.

Minecraft enemies have their own path finding algorithms when the player is in range of an enemy the enemy will chase them down until they reach him, they will try to reach him using the fastest path and safest they will avoid deep holes and obstacles to reach the player.

Warcraft III uses the A* algorithm for the player to reach the selected goal in the fastest way possible. This is the same algorithm we are using on our enemies on the game's hardest difficulty.

2.5. Summary of the two methods:

Table 2-7 Comparison of information gathering methods.

	Strength	Weakness
Searching the Internet	Searching the internet is great in fixing small common problems and learning how to implement new systems into the game, it also provided many different resources all with their own Advantages and Disadvantages it's a great tool to communicate and ask help from professionals.	it is very difficult to find solutions for obscure and technical problems since people on the internet usually only have a surface level understanding of the software they are using; another problem is outdated information most of the information out there is outdated and not updated for the current software version.
Playing Games	Looking at older Games is important it's helps us understand what made these games fun and popular and it lets us experience systems firsthand without having to implement them it gives us great information that we can use to fuel our creativity.	Since game developers don't usually share their code, it is hard to understand how a system, or an algorithm works so we have to use context clues to understand how these systems work and how they interact with the environment most of the times our conclusions are wrong.

CHAPTER 3
ARCHITECTURE, DESIGN AND METHODOLOGY

3. ARCHITECTURE, DESIGN AND METHODOLOGY

3.1. METHODOLOGY

For our game, Royal Defenders, we followed an agile development approach, which allowed us to focus on delivering working software in small iterations, with a focus on flexibility and continuous improvement. To implement this approach, we formed a small, cross-functional team and broke the project down into several sprints.

In the first sprint, we focused on building a demo of the game, which we showed to our supervisor, Dr Mohammad Dahab. This demo helped us get initial approval to move forward with the project.

In the second sprint, we focused on researching and implementing various pathfinding algorithms, as we knew that this would be a crucial aspect of the game's AI. We read several research papers and experimented with different algorithms to determine which ones would work best for our game.

In the third sprint, we developed the tower types for the game, including the ballista (long range), laser (short range), and ice (slows down) towers. We spent a lot of time playtesting and fine-tuning the balance between these tower types to ensure that they were effective and fun to use.

In the fourth sprint, we focused on level design, creating a series of challenging and engaging levels for players to conquer. We also added various visual and audio effects to enhance the game's immersion and polish.

During the fifth sprint, we focused on the development of the tower upgrade system, which enables players to enhance the abilities of their towers as they advance through the game. This feature aims to provide an additional level of tactical complexity and depth to the overall gameplay experience.

During the sixth and final sprint, emphasis was placed on achieving a balance within the game to ensure an optimal level of challenge for players while maintaining fairness. This entailed adjusting the level of difficulty of certain stages, meticulously adjusting the equilibrium between various tower and enemy types and conducting extensive testing to identify and rectify any potential issues.

Overall, our implementation of an agile development methodology has facilitated significant progress on the game development project. By dividing the project into manageable sprints and consistently prioritizing improvement, we have been able to produce a game of a high standard that we anticipate will be well received by players.

3.2. ARCHITECTURE AND DESIGN

This section will discuss the project's design, including its functional and non-functional requirements, as well as some of the diagrams created to explain the project. Finally, it will briefly discuss the technology that will be used in the creation of this video game.

3.2.1. Functional Requirements:

Because our project is a unity game, its functional requirements are slightly different from the norm, and they are as follows:

- Building Towers:
  - The player must be able to build any of the five unlockable towers in any (x, y) coordinate of the grid, as long as the tile is free, the player has enough gold to spend, and building the tower there does not block all possible paths to the end goal.
- Tower Targeting System:
  - The tower will target the nearest enemy within its given range, and the target will take damage if the tower projectile hits their hitbox.
- Enemy Path finding:
  - The Enemy will find its way to the goal using one of three algorithms based on the difficulty level selected by the player.
- Main Menu:
  - When the player launches the game, they will be able to select which level and difficulty level they want to play at.
- Gold Management:
  - During the game, the player will have a gold reserve that they must manage. This gold is used to build towers, and more gold can be obtained by destroying enemies with towers.
- Score Tracking (Saving System):
  - All levels will keep track of the player's highest score in the level at the chosen difficulty.
- Scene Building:
  - Because our project is a game, we will need to create a few scenes, each of which will be distinct from the others, allowing for more player options.
- Others…

3.2.2. Non-Functional Requirements:

Our project's nonfunctional requirements are as follows:

- The software will have a fast response time to the player choices.
- The Software will be bug free.
- The game must be fun.
- The game provides a lot of options for the player.
- The game will run reliably on old machines.
- The game will run at a minimum of 30 frames per second on a standard computer.
- The user interface will be simple and engaging.

3.3. Use Cases:

This section will go over the use cases, which describe how stakeholders can interact with the project.

3.3.1. Use case, Pick tower type:

A player can use one of many tower types, and they can update what tower they are going to build using the above use case, the player will first click one of many buttons shown on the right side of the screen, and based on that choice the player will then be shown a message stating that they don't have enough money, and can't build the new tower, or the button will change colors indicating that their choice has been updated correctly.

3.3.1.1. Actors:

- Player:
  - Is the person who is playing the game; they will be the main actor in the majority, if not all, of the use cases in the project.
- Mouse Position:
  - Is the actor in charge of all user inputs within the current level

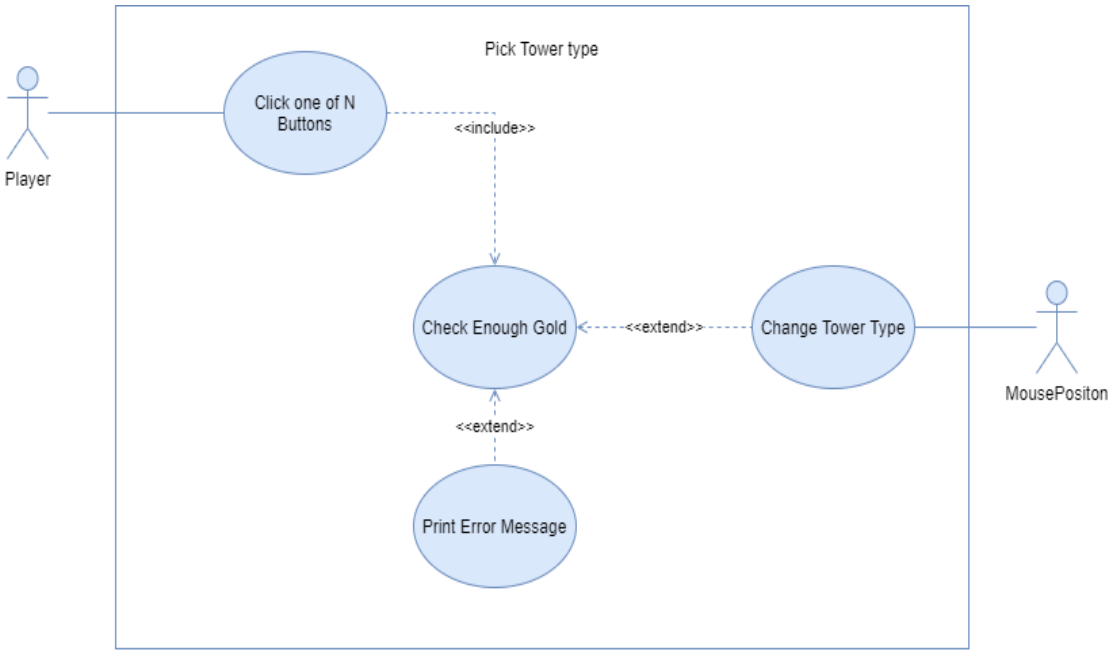


Figure 3-1 Use case: pick tower type

3.3.2. Use Case, Build tower:

When a player clicks on a tile, it will throw a ray cast, and depending on what that ray cast hits and whether or not the player has chosen a tower type, it will try to build the tower, and depending on whether or not the player has enough gold, a create tower method is called, and depending on whether or not creating a tower will close all possible paths to the end goal, the state of the tile as well as the path finding is updated.

3.3.2.1. Actors:

- Player:
  - The person who has clicked on the tile.
- Mouse Position:
  - The entity responsible for checking what tile has been clicked
- Tile:
  - The entity responsible for checking current gold and gold of the picked tower.
- Tower:
  - The Entity responsible for creating a tower based on the path.
- Grid Manager:
  - The Entity that keeps track of all Tiles in the scene
- Path Finding:
  - The entity that notifies all enemies to find a new path, it is also responsible to return a best path based on all difficulties.

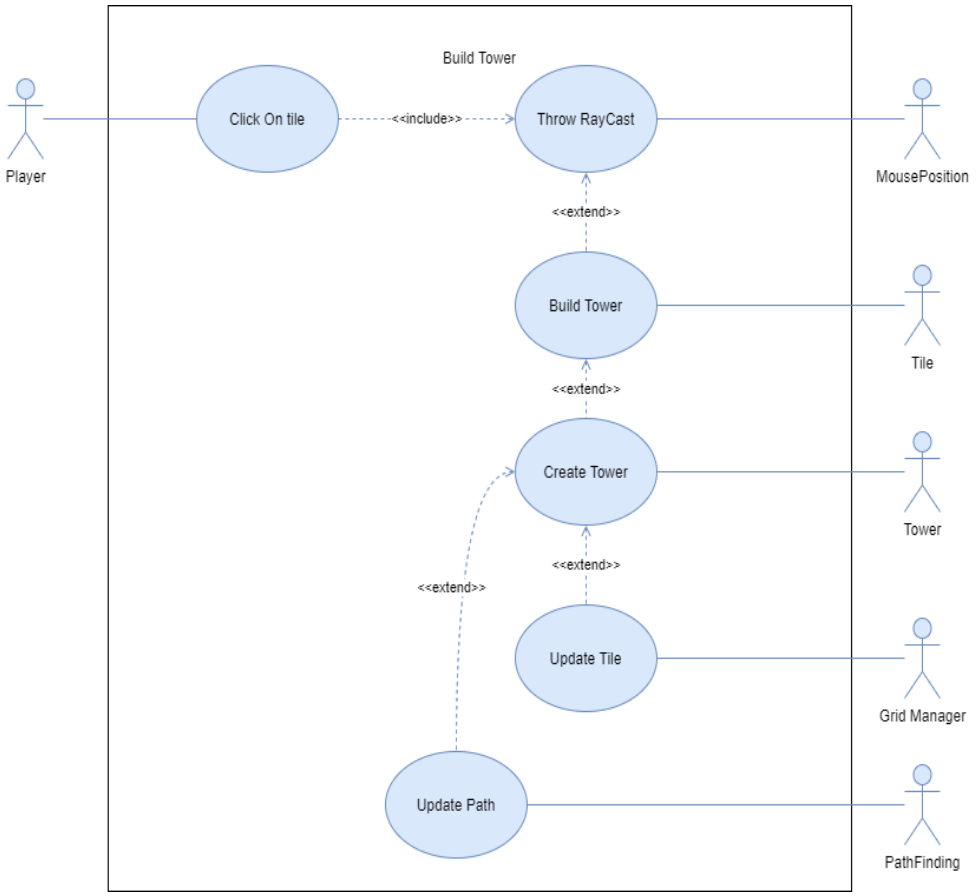


Figure 3-2 Use case: build tower

3.3.3. Use Case, Enemy spawn from object pool:

Every x seconds, a new enemy is spawned, it is returned to the starting coordinate, and then it is given a new path based on the current difficulty level, and it begins following that path node by node.

3.3.3.1. Actors:

- Object Pool:
  - The entity that holds all the enemies.
- Enemy:
  - The entity that acts as the artificial intelligence of the NPCs.
- Pathfinding:
  - The entity that finds the best possible path based on different parameters.

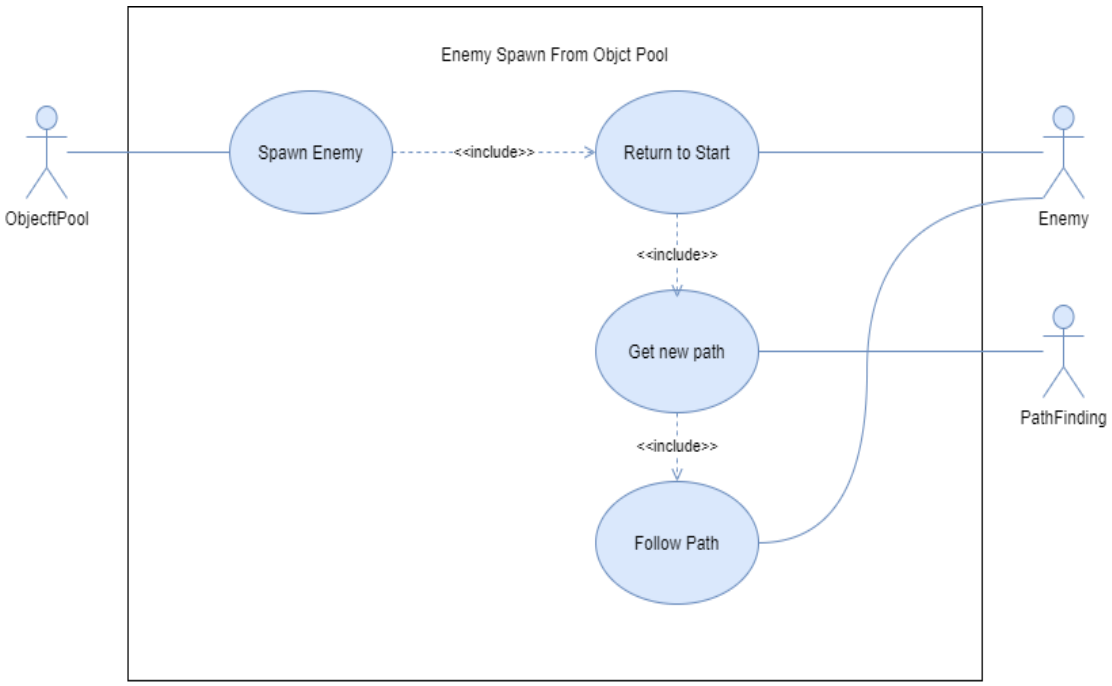


Figure 3-3 Use case: spawn from object pool

3.3.4. Component Diagram:

From the Component diagram (Figure 3-4), you can see that there are many entities that interact with each other, the following is a description of these interactions:

- User Interface Component:
  - Will update the money stored inside the bank.
  - Will interact with the player controller allowing it to build towers.
  - Towers are only built if they are picked from the user interface component.
- Player Controller Component:
  - Builds towers when a tile is clicked.
  - Updates the state of tiles when a tower is built.
- Bank Component:
  - Does not interact with others, but others interact with it.
  - Handles everything relating to the gold management.

- Audio Manager Component:
  - Does not interact with other, and others do not interact with it.
  - Handles everything relating to audio management
- Tower Component:
  - Interacts with the bank.
  - Attacks the enemy.
  - Has a preset cost and Damage output.
- Enemy Component:
  - Steals gold from the bank.
  - Interacts with the towers.
  - Navigates on the tiles.
  - Gets Path from Pathfinding.
- Tile Component:
  - The visual representation of the nodes.
  - Interacts with pathfinding.
  - Every tile has a speed based on its type.
- Pathfinding component:
  - The Artificial Intelligence of the game.
  - Gives a path to the Enemy.

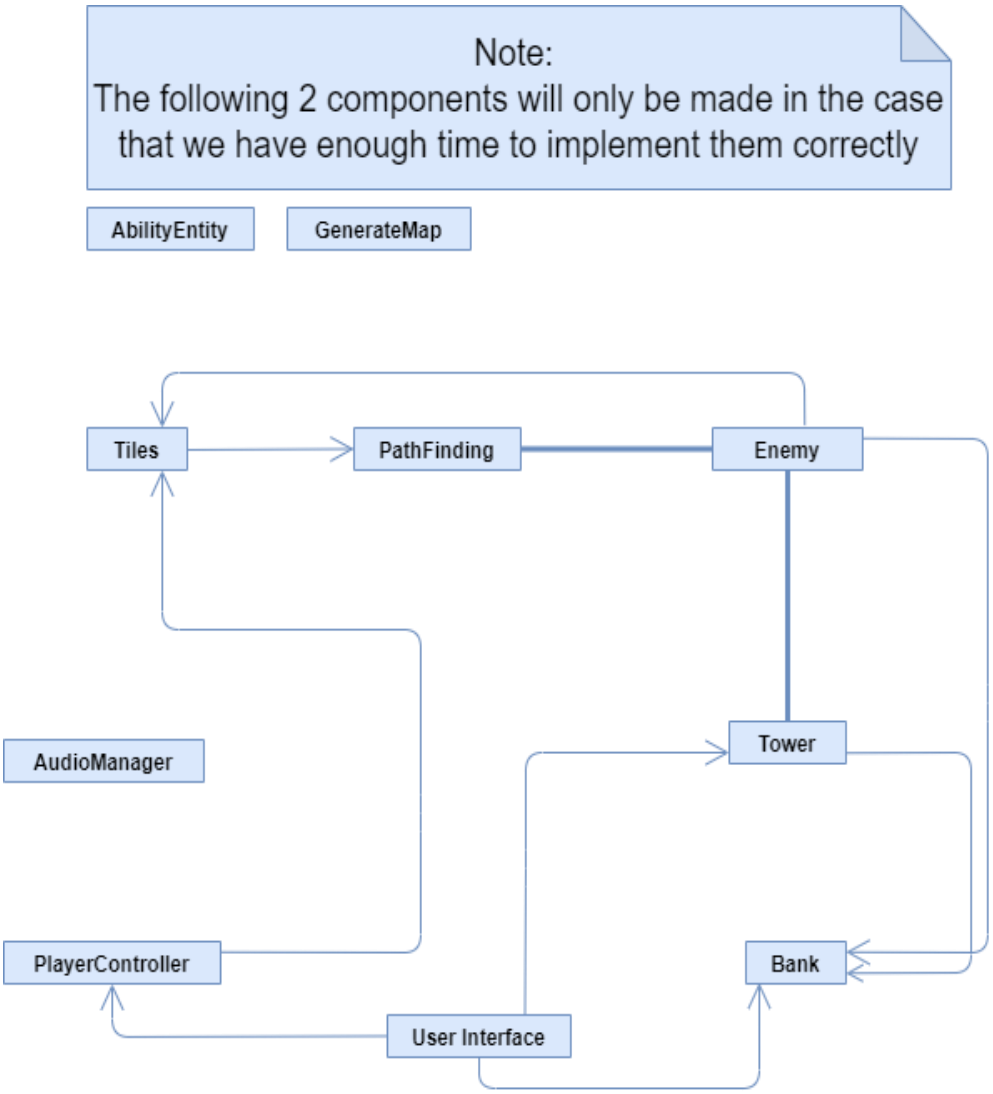


Figure 3-4 Component Diagram Short

3.3.5. Pathfinidng component

The pathfinding component can be seen in the above figure; it interacts with the enemy component by giving it a new path and being called by it whenever a tower is built, or an enemy requires a new path. Some of its functions are called by the tiles component in some cases; it also shows what objects and scripts the component is made up of; examples include:

- Node class, object:
  - The back end of the nodes contains various data members that are used inside the pathfinding algorithms, such as the cost, heuristic value, and flags that are changed when it is explored or reached.
- Grid Manager, script:
  - o Is the script that manages the nodes class, stores the nodes in a dictionary with the key being their coordinates, and can update the costs and heuristic values of each node.

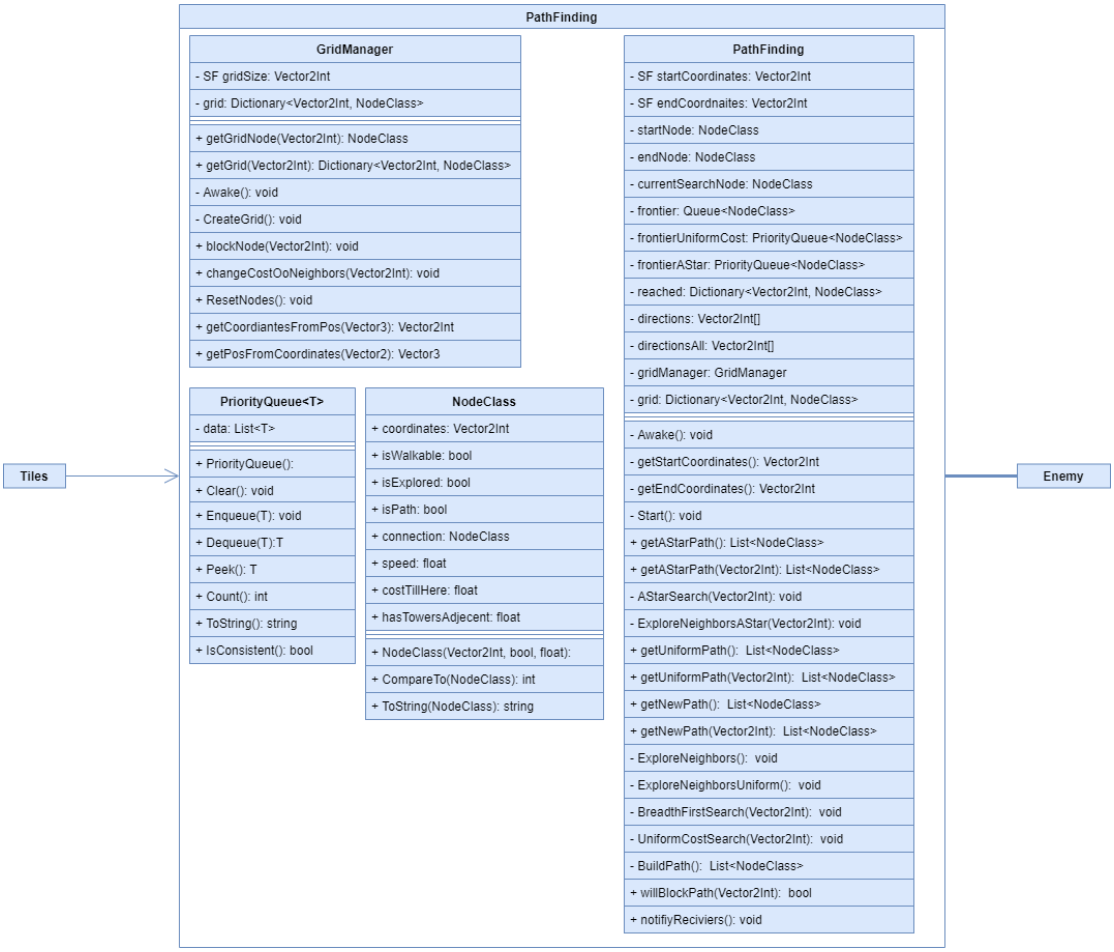


Figure 3-5 Pathfinding component details

3.3.6. User Interface component

The User Interface component is depicted above; it interacts with three other components: the Player Controller, the Tower, and the Bank. It only contains scripts, and these scripts are as follows:

- Settings menu, script:
  - Is in charge of updating the player preferences, such as changing the resolution, quality settings, difficulty, and volume.
- Pause menu, script:
  - Is in charge of managing game pausing and resuming by changing the Time global component.
- TowerBtnCode, script:
  - This UI appears only when the player is inside a level; it is in charge of selecting a tower, which is then sent to the tower component whenever the player clicks on a tile.
- Saving System, object:
  - In charge of saving and loading player progress from a file on the player's desktop.

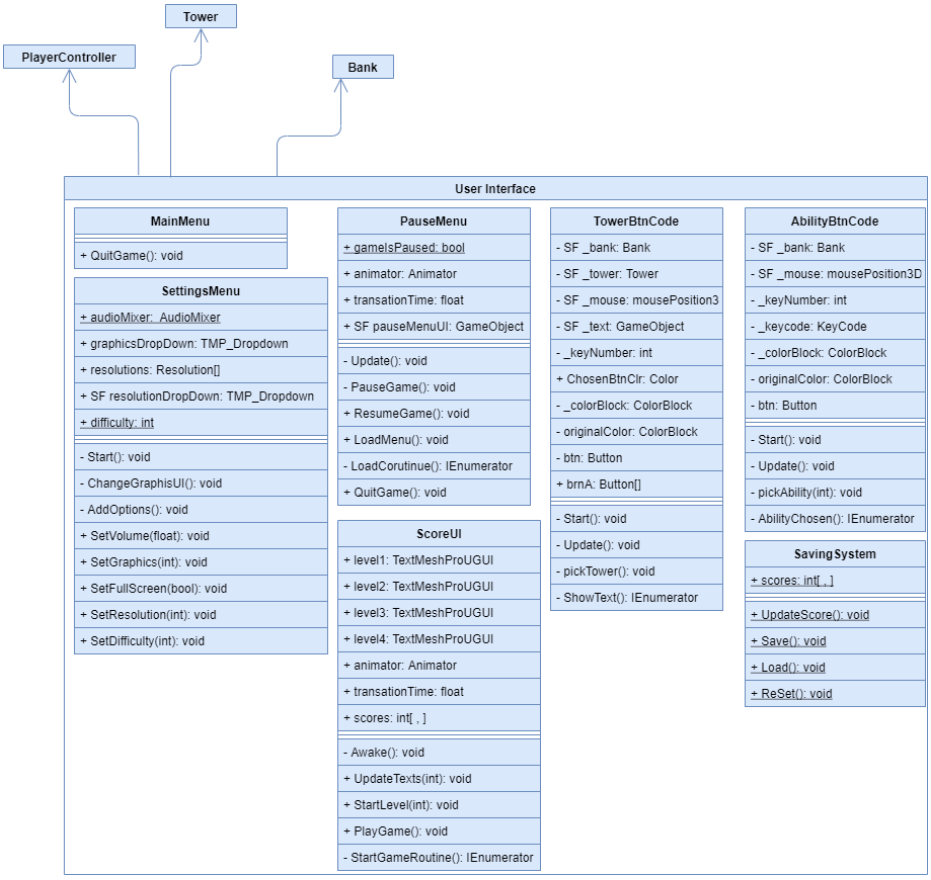


Figure 3-6 User Interface Component

3.4. Activity Diagrams:

This sub section will explain how some of the backend activities occur

3.4.1. Activity diagram, Build a tower:

When a player clicks on a tile, a ray cast will be thrown from the camera's position to the mouse's position, and it will then check if it hit anything or not; if it hit something, continue; otherwise, do nothing.

If it hits something, it will check its tag, and depending on its tag, it will either continue or do nothing. It will then check its tower type, and if no type is selected, it will do nothing. Finally, it will check to see if the mouse has been clicked; if the left mouse button has been clicked, continue; otherwise, do nothing.

It will then go to the tile object and based on if the tiler is walkable and it won't block the path, it will continue or do nothing.

It will then go to the Tower entity and based on if the tower can be built within the current gold reserves, it will either return false or return true after withdrawing gold from the reserves.

After returning a Boolean value, the tile status will be updated, and if the tower was built correctly, the cost of the neighbors will be updated, as well as all NPCs will be notified to find a new path.

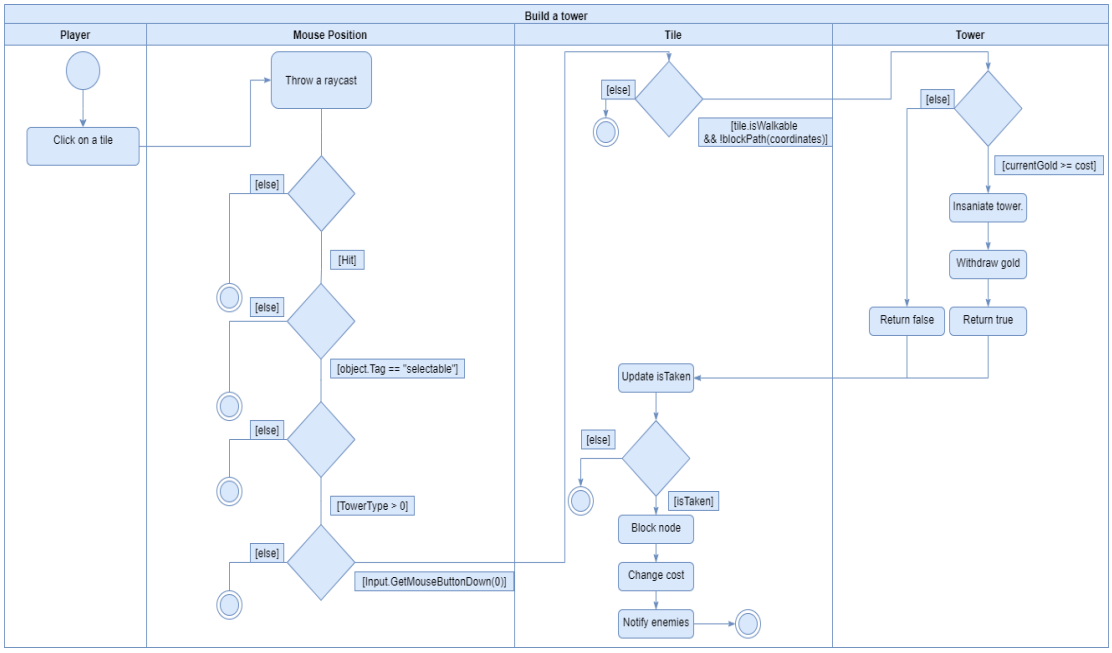


Figure 3-7 Activity diagram, Build a tower

3.4.2. Activity diagram, Path finding:

If the enemy is spawning, then get the start coordinates and stop all coroutines, else, get the current position coordinates of the enemy and stop all coroutines.

If the difficulty is set to easy, then use the Breadth First Algorithm to find the path, Else if the difficulty is set to normal, then use the Uniform Cost Algorithm to find the path, Else if the difficulty is set to hard, then use the A* Algorithm to find the path after that follow the used path to reach the goal and exit the path finding method.

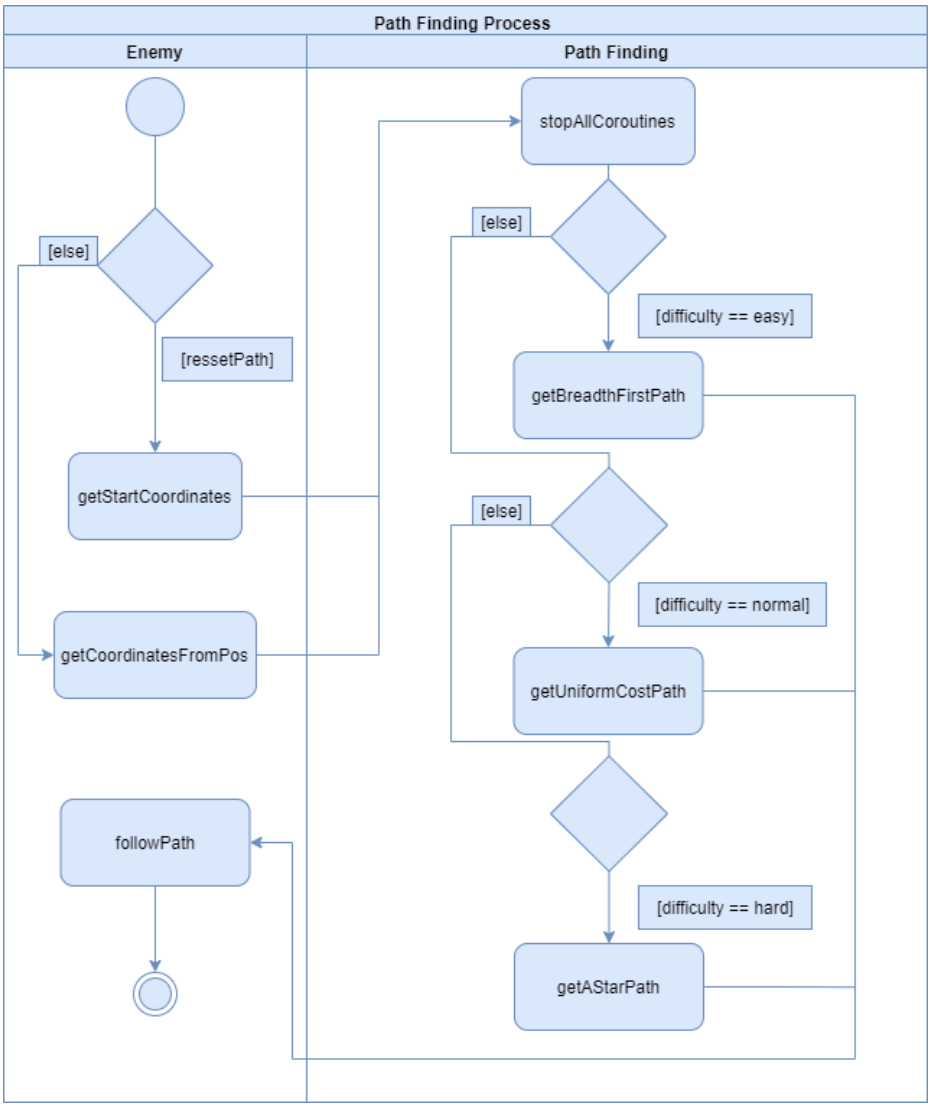


Figure 3-8 Activity diagram, Path Finding

CHAPTER 4
IMPLEMENTATION DETAILS

4. Implementation:

In this chapter, we will delve into the specific tools and technologies that we utilized during the development of our project. We will provide a detailed overview of the various systems, models, levels, menus, and other components that we created, as well as the technologies and tools that we used to bring these elements to life. Additionally, we will discuss the reasons behind our choices and how these tools and technologies helped us to achieve our objectives. By exploring the different technologies and tools that we used, we hope to provide a comprehensive understanding of the development process and the challenges and opportunities that we encountered along the way.

4.1. Languages:

There was only one language which was needed for this project:

4.1.1. C#:

C# is a programming language developed by Microsoft in 2000. It is a modern, object-oriented language that is designed to be easy to learn and use, while still providing powerful features for software development. C# is used primarily for developing desktop and web applications, as well as games. It is often used in conjunction with the Unity game engine to create cross-platform games. One of the key features of C# is its strong support for object-oriented programming, which allows developers to create reusable code and modular software. Additionally, C# has a large and active community of developers, which provides a wealth of resources and support for those learning the language. The language is also supported by Microsoft's Visual Studio development environment, which provides a wide range of tools to aid in the development process. C# is also widely used in the industry, making it a valuable skill for developers to learn.

4.2. Software:

In this section, we will discuss the tools that we will use for our project. These tools are Unity, Blender, and GitHub. Unity is a powerful game engine that is widely used for the development of 2D and 3D games, as well as other interactive content. Blender is a free and open-source 3D modeling and animation software that is widely used in the film, television, and video game industries. GitHub is a web-based platform that provides version control and collaboration tools for software development projects. Together, these tools will provide us with a comprehensive set of tools and features for designing and developing our game, including the ability to create 3D models and animations, manage code and track changes, and collaborate with others on the project.

4.2.1. Unity:

Unity is a powerful and popular game engine that is widely used for the development of 2D and 3D games, as well as other interactive content. One of the main reasons why developers choose to use Unity is its versatility and flexibility. Unity supports a wide range of platforms, including Windows, Mac, Linux, iOS, Android, and many others, which makes it an ideal choice for developers who want to reach a broad audience.

Another advantage of Unity is its strong feature set. The engine includes a wide range of tools and features for designing and developing games, including a visual scripting system, a physics engine, a built-in animation system, and support for shaders and post-processing effects. This makes it easier for developers to create complex and visually impressive games without having to rely on external tools or frameworks.

Unity also has a large and active community of developers and users, which provides a wealth of resources, support, and inspiration. The engine has a comprehensive documentation and tutorial library, as well as a vast selection of assets and plugins available on the Unity Asset Store. This makes it easier for developers to find solutions to problems, learn new techniques, and get feedback and support from other members of the community.

Overall, Unity is a powerful and popular game engine that offers a wide range of tools and features for game development, as well as strong support and a vibrant community. These factors make it an appealing choice for many developers, especially those who are looking for a versatile and feature-rich platform for creating games and other interactive content.

4.2.2. Blender:

Blender is a free and open-source 3D modeling and animation software that is widely used in the film, television, and video game industries. One of the main reasons why developers choose to use Blender is its comprehensive feature set and wide range of tools for creating 3D models, animations, and other assets.

Blender includes a variety of tools and features for modeling, texturing, sculpting, rigging, animating, and rendering 3D content. It also supports a wide range of file formats and can be used in conjunction with other software, such as game engines and compositing software.

Another advantage of Blender is its user-friendly interface and intuitive workflow. The software has a well-organized layout and includes a variety of tools and features that are easy to learn and use, even for beginners. This makes it an appealing choice for developers who want to create 3D models and animations without a steep learning curve.

Blender also has a large and active community of users and developers, who provide a wealth of resources, support, and inspiration. The software has a comprehensive documentation library and a vast selection of tutorials, as well as a wide range of third-party plugins and add-ons that extend its functionality. This makes it easy for developers to find solutions to problems, learn new techniques, and get feedback and support from other members of the community.

Overall, Blender is a powerful and user-friendly 3D modeling and animation software that offers a wide range of tools and features for creating 3D models and animations. Its comprehensive feature set, intuitive interface, and strong community support make it an appealing choice for many developers.

4.2.3. Github

GitHub is a web-based platform that provides version control and collaboration tools for software development projects. It is widely used by developers and organizations around the world to manage and share code, track changes, and collaborate with others on projects.

One of the main reasons why developers choose to use GitHub is its powerful version control capabilities. GitHub uses the Git version control system, which allows developers to track changes to their code, revert to previous versions, and collaborate with others on projects. This makes it easier for developers to work on projects as a team, without the risk of losing work or overwriting each other's changes.

Another advantage of GitHub is its collaboration features. The platform includes tools for sharing code, reviewing changes, and tracking issues and bugs. This makes it easier for developers to communicate with each other, coordinate their work, and stay organized.

GitHub also has a large and active community of developers and users, who contribute to a wide range of open-source projects and share their knowledge and experience with others. The platform has a comprehensive documentation library and a vast selection of resources, tutorials, and templates that can help developers learn new techniques and best practices.

Overall, GitHub is a powerful and popular platform that provides version control and collaboration tools for software development projects. Its powerful version control features, collaboration tools, and strong community support make it an appealing choice for many developers.

4.3. Backend:

4.3.1. AbilityBtnCode:

This code defines the behavior of a UI button that represents a player ability in a game. It is responsible for checking if the player has enough in-game currency to use the ability, changing the color of the button based on its state (locked or unlocked), and displaying a tooltip when the user hovers over the button. It also handles events related to picking and unpicking abilities, as well as turning the button off or on depending on whether a tower is selected or not.

4.3.2. AudioManager

This is a script for an Audio Manager in the game, which sets up and plays various sound effects and music in the game. The script creates an instance of the Audio Manager class, which is a singleton, ensuring that there is only one instance of the Audio Manager throughout the game. The AudioManager has an array of Sound objects, each containing information about the AudioClip to be played and the type of audio (sound effect or music). In the Awake() function, an AudioSource component is added to each Sound object, and the AudioMixerGroup for the sound effect or music type is assigned to the AudioSource. The Play() function finds the Sound object with the given name and plays its AudioClip. The Start() function sets the volumes for the music and sound effects, and the setMusic() and setSE() functions are called when the

user changes the volume in the Audio Options menu. Finally, the DontDestroyOnLoad(gameObject) line ensures that the Audio Manager object is not destroyed when a new scene is loaded.

4.3.3. AudioOptionsManager

This code is for managing audio options in the game Royal Defenders. It defines a class called AudioOptionsManager, which has two static properties, musicVolume and soundEffectsVolume, representing the current volume levels for the game's background music and sound effects. The class also has two methods, OnMusicSliderValueChange and OnSoundEffectsSliderValueChange, which are called when the player adjusts the music and sound effects volume sliders, respectively. These methods update the corresponding volume properties and call the setMusic and setSE methods in the AudioManager class to set the audio mixer groups' volumes accordingly.

4.3.4. Bank

The code defines a Bank class for a game that keeps track of the player's gold and health. It has methods to deposit and withdraw gold and health, and updates the display for both values accordingly. If the player's health goes below or equal to zero, it sets a boolean flag indicating that the player has lost, and shows a retry menu. The starting values for gold and health are set in the inspector, and the display for both values is also set in the inspector using the TextMeshProUGUI component.

4.3.5. CoordinateLabeler

The CoordinateLabeler script adds a label to a game object that displays its corresponding grid coordinates in the GridManager component. It uses the ExecuteAlways attribute to execute the Update() method even in edit mode. It also requires a TextMeshPro component and uses it to show the coordinate label. The label color changes depending on whether the grid node is walkable, explored, or part of a path. The ToogleLable() method toggles the visibility of the label, which can be triggered by pressing the 'C' key.

4.3.6. Enemy

The "Enemy" class is a MonoBehaviour script that is attached to all enemies in the game Royal Defenders. It has two public methods, "giveGoldOnDeath" and "yoinkHealthOnExit," which are called by the game when an enemy is killed or exits the player's base, respectively. The "giveGoldOnDeath" method deposits a certain amount of gold to the player's bank, and the "yoinkHealthOnExit" method withdraws a certain amount of health from the player's bank. These amounts are set by the "worth" and "yoink" variables, which are serialized fields that can be set in the Unity Inspector. The script also has a reference to the "Bank" script, which is obtained by calling "FindObjectOfType<Bank>()" in the Start method.

4.3.7. EnemyHealth

This code defines the behavior for the enemy's health in the game Royal Defenders. The script creates a reference to an associated enemy script and victory menu script, sets a maximum health value for the enemy, and initializes the current health value to the maximum. The script then listens for particle collision events and decreases the enemy's current health based on the damage of the particle system. If the enemy's health falls to zero, the victory menu's number of enemies destroyed is incremented, gold is awarded to the player, an explosion effect is instantiated, and the enemy game object is destroyed. If the particle system slows down the enemy, the enemy's slow-down modifier is updated.

4.3.8. EnemyMover

This code defines the behavior of enemy movement in the game Royal Defenders. The enemy moves along a pre-determined path from its starting position to the player's base. The path is calculated using the A* algorithm or a variant of it, depending on the game's difficulty level. The enemy's movement speed is influenced by the tiles it moves on, and the enemy can slow down if it is hit by a particular particle. When the enemy reaches the end of its path, it causes damage to the player's base and is destroyed.

4.3.9. GridManager

This is a script for managing the grid in the game Royal Defenders. It creates a grid of nodes based on a specified size, where each node contains information about its position, whether it is walkable or blocked, and the speed of the tile it is located on. The script also provides methods for blocking or unblocking nodes, changing the cost of neighboring nodes, and resetting all the nodes. Additionally, there are methods for converting between world positions and grid coordinates.

4.3.10. mouseDrawCircle

This is a script in C# for a game called Royal Defenders. The script creates a circle using a LineRenderer component, which can be used as a visual aid for an ability in the game. The size and number of segments of the circle can be adjusted through public variables. The circle is drawn in local space, and the script includes methods to remove and draw the circle. The script is attached to a game object, which is expected to have a LineRenderer component.

4.3.11. NodeClass

This is a class definition for a node in a game using the Unity engine. The class includes variables for the node's coordinates, whether or not it can be walked on, whether it has been explored, and whether it is on a path. It also includes variables for the node's connection to another node, its speed, and its cost to reach from the starting node. There are also methods for comparing nodes based on their cost to reach and for returning a string representation of the node's coordinates.

4.3.12. ObjectPool

This is a script that manages an object pool for a specific enemy GameObject. It creates and fills the pool with a certain number of enemies specified by the poolSize variable. The spawnTimer controls how frequently the enemies are spawned from the pool. The burstTimer and burstSize variables are used to create bursts of enemies in addition to the regular spawning. The SpawnWave coroutine is used to spawn enemies from the pool, with a delay between each spawn determined by the spawnTimer. Every burstSize spawns, there is a longer delay determined by the burstTimer. The script also checks if all enemies have been removed from the pool, and if so, it deactivates the object containing this script.

4.3.13. ParticleHandler

This is a script for handling particles in a game. The script has fields for setting the particle's damage and slow down modifier. It has methods for getting the damage and slow down modifier, as well as setting them. There are also methods for increasing the damage and slow down modifier by a specified amount. The script also has a boolean field for checking if the particle slows down and a method for getting its value.

4.3.14. Pathfinding

This is a script that implements three pathfinding algorithms: Breadth-First Search (BFS), Uniform Cost Search (UCS), and A* search.

The script takes two inputs, startCoordinates and endCoordinates, which specify the starting and ending points of the search. There are three pathfinding algorithms that can be used: getNewPath() uses BFS, getUniformPath() uses UCS, and getAStarPath() uses A*.

The algorithms use a priority queue to store the frontier, where the priority is based on the cost of the path. The search is done by expanding the frontier in a breadth-first manner, considering all the neighbors of the current node.

The algorithm also stores the nodes that have already been reached in a dictionary called reached. When a node is reached for the first time, it is added to the dictionary and the frontier. If the node has been reached before, the algorithm checks if the current path to that node is shorter than the previous path, and updates the cost and the connection to the current node if so.

The time complexity of each algorithm is O(b^d), where b is the branching factor (i.e., the number of neighbors each node has), and d is the depth of the shortest path to the goal. The space complexity is also O(b^d), as the algorithm needs to store the entire frontier and the reached nodes. However, in practice, the space complexity is usually lower due to the use of heuristics (in the case of A* search) and pruning strategies.

4.3.15. PauseMenu

This script is a Pause Menu in a Unity game. It begins with defining some public variables, such as animator and transationTime, and some serialized private variables for the Pause Menu, Retry Menu, and Victory Menu UI. The Update() method is used to check if the retryMenuUI or victoryMenuUI are active, and if not, it listens for key inputs of 'Escape' or 'P'. When either key is pressed, the game checks if it is already paused, and either pauses or resumes the game accordingly.

When the game is paused, the PauseGame() method is called which sets the pauseMenuUI to be active, sets the game's timeScale to 0, and sets gameIsPaused to be true. The ResumeGame() method is used when the player wants to resume the game, which sets the pauseMenuUI to be inactive, sets the game's timeScale to the current speed of the game, and sets gameIsPaused to be false.

The LoadMenu() method sets the pauseMenuUI to be inactive, sets the game's timeScale to 1, sets gameIsPaused to be false, and starts a coroutine called LoadCoroutine(). The LoadCoroutine() method sets the animator's trigger to 'Start', waits for the transition time, and then loads the main menu scene. Lastly, the QuitGame() method simply prints a log message and quits the application.

4.3.16. PlayerController

This is a C# script for a player controller in our tower defense game. It allows the player to construct towers and use special abilities.

The script starts by importing necessary libraries, including Unity and System. Then, several public and private variables are declared, including a camera, layer mask, an array of towers, a victory menu, two hotbar game objects, an asteroid game object, a tower type integer, an ability type integer, an audio source, an audio clip, a static integer for penalty handling, and a private variable for the previous tile.

The Start() method initializes the penaltyHandler to zero.

The Update() method checks if the game is paused and if the player is not currently clicking on a UI element. If the player has selected an ability or tower, it calls the ability() or ConstructingTowers() methods, respectively. It also shows and removes the tower menu.

The ability() method allows the player to use a special ability. It casts a ray from the camera to the point clicked by the player, checks if the ray hits a tile (layerMask), changes the position of the player to that point, draws a line to indicate the ability's range, and then instantiates an asteroid object when the player clicks the left mouse button. The method then resets the ability type integer and sends messages to the hotbar game objects to turn them on and off, respectively.

The RemoveCurrentMenu() method is called when the player clicks the right mouse button. If the player had previously selected a tile with a tower, the method hides the canvas for the tower's user interface and removes the circle indicating the tower's range.

The ShowTowerMenu() method casts a ray from the camera to the point clicked by the player, checks if the ray hits a tile (layerMask), changes the position of the player to that point, and if the tile has a tower, shows the canvas for the tower's user interface and displays the circle indicating the tower's range. If the player clicks the middle mouse button while a tile with a tower is selected, the method hides the canvas and removes the circle.

The ConstructingTowers() method allows the player to build a tower. It casts a ray from the camera to the point clicked by the player, checks if the ray hits a tile (layerMask), changes the position of the player to that point, and if the tile is selectable, lights it up. If the player has selected a tower type, it checks if the tile is walkable, not taken by another tower, and doesn't block the path (for hard difficulty only). If the player clicks the left mouse button and all the requirements are met, it calls the BuildTower() method.

The BuildTower() method creates a tower object, places it on the selected tile, sets its attributes, and deducts the cost of the tower from the player's bank account.

4.3.17. RetryMenu

This is a C# script in Unity that handles a retry menu. It contains methods to show the menu, reload the game, load the menu, reload the level, and quit the game. The script uses an animator to trigger a transition effect when loading or reloading the game. It also sets the game's timescale to 0 when showing the retry menu, and sets it to the current speed when reloading the game. Finally, it includes a public variable for the retry menu UI, which can be assigned in the editor.

4.3.18. ScoreUI

The code controls the updating and display of the player's scores for each level of a game. The script has a public array of TextMeshProUGUI elements called "levels" that is used to display the best score for each level on the screen. Upon initialization, the script retrieves the player's best scores for each level from PlayerPrefs and updates the TextMeshProUGUI elements to display them. The script also has a function called "StartLevel" that initiates the start of a level when its corresponding button is clicked, using a coroutine to transition to the level scene with an animation. Finally, there is a function called "UpdateTexts" which is used to update the displayed scores for a specific difficulty level.

4.3.19. SettingsMenu

This script is used for managing various game settings such as graphics, resolution, audio volumes, and difficulty. It contains public variables for an AudioMixer, a dropdown for graphics settings, an array of available resolutions, a dropdown for selecting resolutions, and two sliders for adjusting music and sound effects volumes. The script initializes the UI elements and retrieves the user's preferred volume settings from player preferences on start. It also includes methods for setting the music volume, sound effects volume, graphics quality, fullscreen mode, and resolution. Finally, it has a static variable for difficulty, which can be set from other scripts.

4.3.20. SpeedChanger

This script is called "SpeedChanger" and is used for changing the speed of an object in a game. It has a public static integer variable called "currentSpeed" that is initialized to 1, and a method called "ChangeSpeed" that is called when a button is pressed. The method gets a reference to the TextMeshProUGUI component attached to the same game object and changes the currentSpeed value and Time.timeScale to either 1 or 2 depending on the current value of currentSpeed. Finally, it updates the text in the TextMeshProUGUI component to show the current speed value followed by "X".

4.3.21. TargetLocator

The script is a class named "TargetLocator" derived from MonoBehaviour, which provides built-in Unity functionality. It contains several serialized fields for public use:

- weapon, upgradedWeapon, and shooter are transforms of game objects that will be used for aiming and shooting.
- bullet is a ParticleSystem for controlling the bullet effect.
- shootingRange is the maximum distance the TargetLocator can attack.
- shootAoe is a boolean that toggles AOE shooting, which is used to attack multiple enemies.
- sound is a ParticleSystem that will be played when a bullet is fired.
- upgradePenelty is an integer representing the cost to upgrade the weapon.
- maxUpgradePrice is an integer that defines the maximum amount a player can spend on upgrading the weapon.
- upgradeOne and upgradeTwo are GameObjects representing upgrade options.
- bullet2, bullet3, bullet4, and bullet5 are additional ParticleSystem objects that are used when the weapon is fully upgraded.
- maximumDmage, maximumRange, and maximumRate are integers that define the maximum damage, maximum range, and maximum firing rate of the weapon.
- victoryMenu is a script that will be used for managing a victory menu.
- tower is a script that will be used for managing a tower.
- bank is a script that will be used for managing player bank account.

In the Start method, the script gets references to other components in the scene by using GetComponent and FindObjectOfType methods. Specifically, it gets a reference to the VictoryMenu component and the Tower component attached to the same game object as the script, and a reference to the Bank component in the scene. If the shootAoe boolean is true, the script instantiates a copy of the sound ParticleSystem at the location of the script.

The Update method is called once per frame and is used to update the state of the game. If shootAoe is false, the FindEnemies method is called to find the closest and second closest enemy game objects within range of the tower. The AimWeapon method is then called to rotate the weapon to face the closest enemy, and the

AttackToggle method is called to enable or disable the bullet ParticleSystem emission based on whether an enemy is in range.

If timer is greater than 40 and bulletSoundChecker is true, a copy of the sound ParticleSystem is instantiated at the location of the script. If shootAoe is true, timerSnow is incremented, and if it is greater than snowSound, a copy of the sound ParticleSystem is instantiated at the location of the script.

The FixedUpdate method is called at a fixed rate and is used to update the physics state of the game. It increments timer and timerSnow if bulletSoundChecker is true.

The DrawCircle method draws a circle using the LineRenderer component attached to the same game object as the script. The UnDrawCircle method removes the circle by setting the positionCount of the LineRenderer to zero.

The FindEnemies method finds all game objects with the "enemy" tag and stores the two closest enemies within range of the tower in the target and farTarget variables.

The AimWeapon method rotates the weapon to face the closest enemy, and if the tower is fully upgraded and of type 0 or 1 with a farTarget, it rotates the weapon to face the farTarget or target, respectively. It also rotates the shooter object to face the target. It then determines whether the target is within range of the tower and enables or disables the bullet ParticleSystem emission accordingly using the AttackToggle method.

The AttackToggle method enables or disables the bullet ParticleSystem emission and bulletSoundChecker boolean based on the isActive parameter. If the tower is fully upgraded, it also enables or disables the bullet2 and bullet3 ParticleSystems.

The OnDrawGizmos method is called by the editor to draw gizmos in the scene view. In this case, it draws a yellow wire sphere at the location of the script with a radius of shootingRange.

The IncreaseDamage method is called when the "upgrade" button is clicked in the UI. It checks if the upgradeTwo button is interactable and if the player has enough gold to purchase the upgrade. If these conditions are met, it subtracts the upgradePrice from the player's gold, sets maximumDmage to the maximum of its current value and increase, and updates the text of the "upgrade" button. If maximumDmage is greater than or equal to the maximum upgrade value, it sets fullyUpgraded to true and disables the "upgrade" button.

The IncreaseRate method is a function that increases the rate of fire of a tower. The method checks if the button associated with the upgrade is currently interactable, and if the player has enough gold to purchase the upgrade. If both of these conditions are met, the method increases the rate of fire of the tower and withdraws the cost of the upgrade from the player's gold. The method also updates the upgrade price to be slightly higher for future upgrades, and disables the button if the maximum rate of fire

is reached. The victoryMenu.numberOfTowerUpgrades is also incremented to keep track of the number of tower upgrades.

The IncreaseRange method increases the range of a tower by a certain amount. The method checks if the button associated with the upgrade is currently interactable, and if the player has enough gold to purchase the upgrade. If both of these conditions are met, the method increases the range of the tower, withdraws the cost of the upgrade from the player's gold, updates the upgrade price to be slightly higher for future upgrades, and disables the button if the maximum range is reached. The method also calls UnDrawCircle() to remove the old range circle, and DrawCircle() to draw a new range circle around the tower.

The DecreaseDuration method reduces the duration of the snow effect particle system by a certain amount. The method checks if the button associated with the upgrade is currently interactable, and if the player has enough gold to purchase the upgrade. If both of these conditions are met, the method decreases the duration of the particle system, withdraws the cost of the upgrade from the player's gold, updates the upgrade price to be slightly higher for future upgrades, and disables the button if the minimum duration is reached. The method also stops the snow particle effect, resets the timer and sound variables, and starts the particle effect again with the new duration.

The IncreaseSlowDown method increases the slowdown modifier of the snow effect particle system by a certain amount. The method checks if the button associated with the upgrade is currently interactable, and if the player has enough gold to purchase the upgrade. If both of these conditions are met, the method increases the slowdown modifier of the particle system, withdraws the cost of the upgrade from the player's gold, updates the upgrade price to be slightly higher for future upgrades, and disables the button if the maximum slowdown modifier is reached.

The CheckFullyUpgraded method is a function that checks whether both upgrade buttons are disabled. If both buttons are disabled, it calls the FullyUpgraded() function of the tower object to fully upgrade the tower, and sets the fullyUpgraded boolean flag to true. The method is called after each upgrade to check if the tower is fully upgraded.

The OnDestroy method is called when the object is destroyed, such as when the game is over. The method deposits the upgrade price into the player's gold, so that the player is not penalized for leaving an upgrade unfinished.

4.3.22. Tile

The above script is a Tile class that defines the properties and methods of a tile in a game level. It has public bool fields to indicate whether the tile is taken or if it has a tower, and a public float field to set the speed of the tile. It has a reference to the GridManager and PathFinding classes to access information about the game level and its nodes. The Awake method initializes the GridManager and PathFinding objects. The Start method gets the tile's coordinates and blocks the node on the grid if it's taken. The GetIsTaken method returns the isTaken field.

4.3.23. Tower

The class contains variables for the cost of building the tower, the delay time before building, the objects that make up the tower, objects to hold upgraded versions of the tower's body, weapon, and bullets, and a reference to the player's bank. The class also includes methods for creating a tower, building the tower with a coroutine, getting the tower's price, showing and destroying the tower, and upgrading the tower's appearance. Additionally, the class has a reference to a Tile object that the tower is built on.

4.3.24. TowerBtnCode

It allows the player to select a tower from a set of buttons, and highlights the selected button while disabling the others. It also displays a tooltip with the cost of the tower when the player hovers over the button. Additionally, it disables the button if the player cannot afford the tower, or if they are in a special ability select mode.

4.3.25. UpgradeUI

It implements the IPointerEnterHandler and IPointerExitHandler interfaces to handle pointer events. The script updates the text of a UI element to display information about the selected tower upgrade. The upgrade can be related to damage, rate of fire, range, slow effect, or destruction. The information displayed includes the upgrade's current value, the upgrade cost, and the refund value for the "destroy" upgrade. The script accesses components of other game objects such as TargetLocator and ParticleHandler to get and set relevant information.

4.3.26. VictoryMenu

It contains methods for showing the victory menu, reloading the game, loading the main menu, reloading the level, and quitting the game. It also calculates the score and saves it as the high score for the level and difficulty, if applicable. The victory menu UI is activated after a short delay, and the game is paused. The script uses an animator to transition to the main menu or the game level when a corresponding button is clicked.

4.3.27. WaveManager

This is a script for managing waves of enemies, it contains a WaveManager class that tracks the current wave and determines when all enemies in the wave have been defeated, and a StartWaveButton method that triggers the spawning of the next wave of enemies. It also has a reference to a VictoryMenu script that is used to display a victory menu when all waves have been completed. The ChildCountActive method is a helper function that counts the number of active child game objects in each transform.

CHAPTER 5
EXPERIMENTS AND RESULTS.

5. Experiments and results

5.1. Software results:

In this section, we will discuss the interfaces that we currently have for our game. These interfaces include the main menu, settings menu, level select menu, in-game interface, and pause menu. The main menu is the first screen that players see when they launch the game and provides access to the other menus and options. The settings menu allows players to adjust various settings, such as the volume and graphic settings. The level select menu allows players to choose which level they want to play and the difficulty level. The in-game interface provides information and controls for the player during gameplay, such as the gold and amount of health. The pause menu allows players to pause the game to take a break at any time. These interfaces are all essential for providing players with a smooth and intuitive experience as they play the game.

5.1.1. Main menu:

The main menu is the first screen that players encounter when they launch our game. It serves as the gateway to the rest of the game and provides access to the various menus and options. The main menu features three buttons: "Play," "Settings," and "Quit." The "Play" button takes players to the level select menu, where they can choose which level they want to play. The "Settings" button brings up the settings menu, where players can adjust various settings such as the volume and difficulty level. The "Quit" button allows players to exit the game. These buttons provide a simple and intuitive interface for navigating the game, making it easy for players to get started and jump into the action.

Figure 5-1 Main menu

5.1.2. Settings menu

The settings menu is a feature of our game that allows players to customize various options to their liking. It provides a range of options that players can adjust, such as the resolution of the game, the graphics settings, and the volume of the music and sound effects. These options allow players to tailor the game to their preferences and optimize

their experience. For example, players who are playing on a lower-end computer might want to adjust the graphics settings to get a smoother frame rate, while players who prefer a more immersive audio experience might want to increase the volume of the music and sound effects. Overall, the settings menu provides a convenient way for players to customize their experience and get the most out of our game.

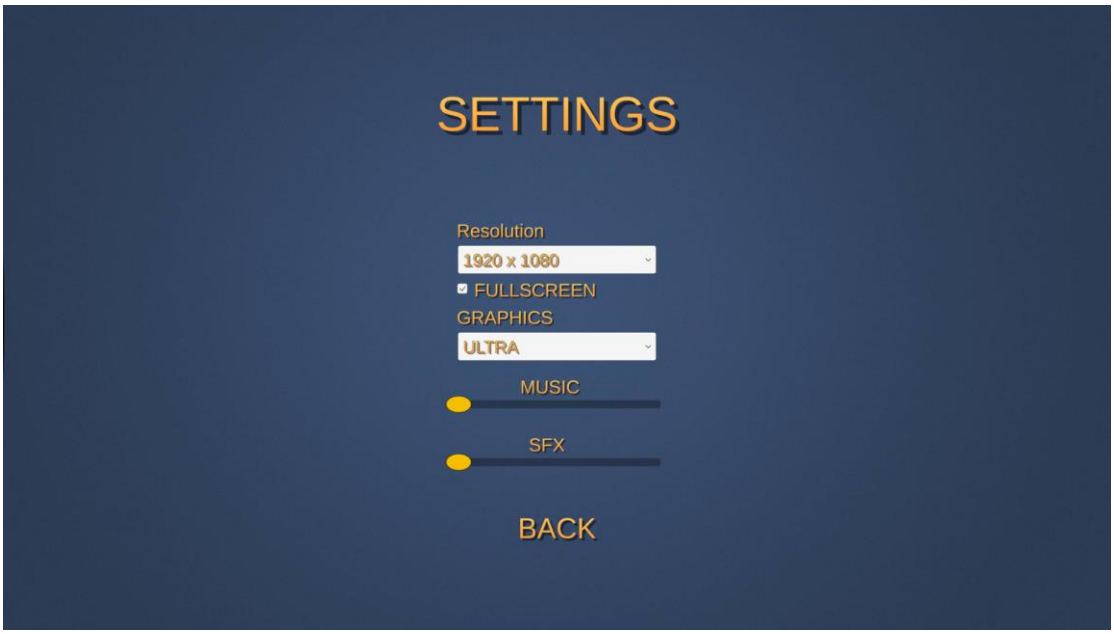


Figure 5-2 Settings menu

5.1.3. Level select menu

The level select menu is a feature of our game that allows players to choose which level they want to play and adjust the difficulty level. It displays a list of the available levels, along with the scores that have been achieved on each level. Players can use this menu to select a level and start playing by clicking the corresponding button. The menu also allows players to change the difficulty level, which adjusts the difficulty of the game accordingly. When the difficulty level is changed, the scores displayed in the level select menu are updated to reflect the scores that have been achieved on the new difficulty level. This allows players to track their progress and compete with others on different difficulty levels. Overall, the level select menu provides a convenient and intuitive interface for choosing levels and adjusting the difficulty of the game.

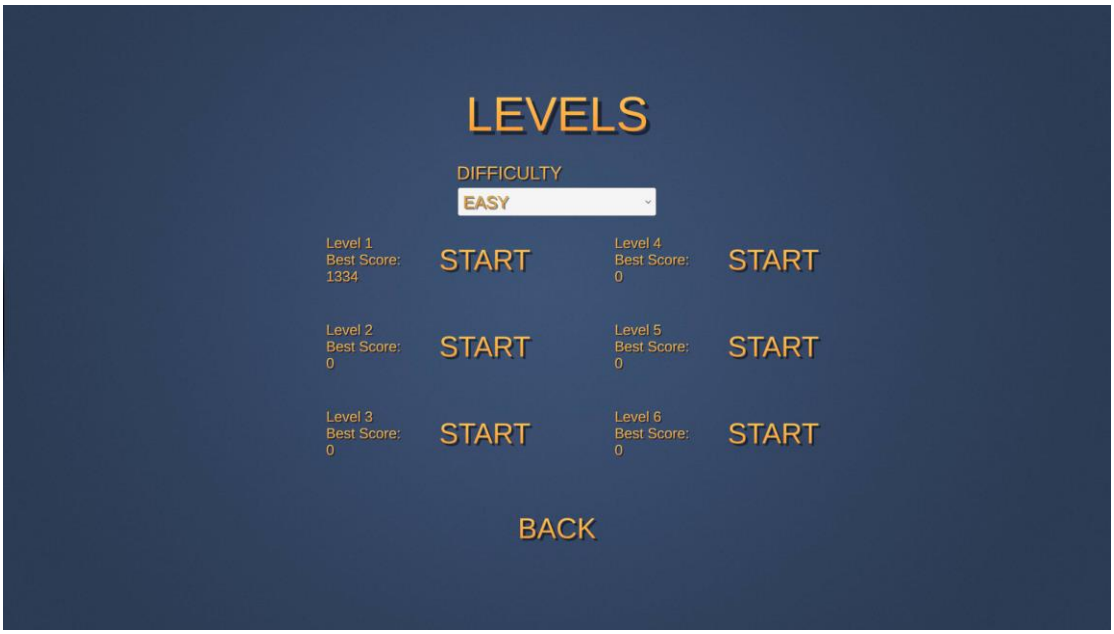


Figure 5-3 Level Select menu.

5.1.4. In game menu:

The in-game menu is a feature of our game that provides important information to the player as they play. It displays the player's current gold and health, as well as the controls for the game. It also includes buttons that allow the player to build towers, start the next wave, and speed up the game. Additionally, there is a button for skills, although it is currently not functional. The in-game menu serves as a convenient hub for the player to access various gameplay elements and keep track of their progress. It is an integral part of the game, providing the player with the information and tools they need to succeed.

To build a tower in our game, follow these steps:

1. Click on the tower button in the game interface.
2. If you have enough gold to purchase the tower, it will be selected. If you do not have enough gold, a message will appear to inform you of this.
3. Once a tower is selected, click on an empty tile in the game world to place the tower. If the tower blocks all possible paths to the goal, it will not be placed, and a message will appear to inform you of this.

Figure 5-4 In game interface

5.1.5. Pause menu.

The pause menu can be accessed at any time during gameplay by pressing the "p" or "esc" keys. This will pause the game and stop the movement of all objects in the game. From the pause menu, the player has three options: they can resume gameplay, return to the main menu, or quit the game. We hope the player chooses to resume gameplay or return to the main menu, rather than quitting the game. The pause menu is a useful feature that allows the player to take a break or make changes to their gameplay without losing their progress.

Figure 5-5 Pause menu.

5.1.6. Gameplay in progress:

The two images that are presented demonstrate the interaction between the player-built towers and the enemies that are traversing the designated path. The towers, which have been constructed by the player, can be seen actively attacking the enemies as they move along the path. This interaction is a crucial aspect of the game, as it forms the core gameplay mechanic of defending against waves of incoming enemies. The ability for the towers to effectively target and eliminate the enemies as they travel the path is a critical aspect of the game, and it is evident from the images that this system is functioning as intended. Overall, the images effectively convey the dynamic nature of the game and the player's ability to strategically utilize the towers to defend against the enemy forces.

Figure 5-6 Tower attacking.

5.2. Testing & Integration

This subsection will delve into the various forms of testing that have been conducted to guarantee the proper functionality of the video game. These tests aim to verify that the game operates according to its intended design and that any potential issues or bugs have been identified and addressed. This can include unit testing, functional testing, performance testing, and user acceptance testing, among others. These tests are crucial in ensuring that the game is of high quality and can provide a seamless and enjoyable experience for the players.

5.2.1. Types of testing performed:

5.2.1.1. White box testing

As a team of three game developers, we held weekly meetings to review the progress and functionality of our work from the previous sprints. This provided us with valuable feedback and insight, as it allowed us to discuss and identify any issues or discrepancies that may have arisen during our individual development efforts. Additionally, these meetings served as an opportunity for us to communicate and coordinate our efforts, ensuring that the game was progressing smoothly and effectively towards completion. Overall, these meetings were an essential aspect of our development process, as they helped us to stay on track and ensure that our game was functioning as intended.

5.2.1.2. Black box testing:

The game was shared with colleagues from diverse gaming backgrounds to evaluate the accuracy of our previous testing and to gauge the level of enjoyment from the game's balance. We were fortunate to have the assistance of approximately 20 early access testers, and we are grateful for their contributions in helping to improve the game. Their feedback provided valuable insights that allowed us to make necessary adjustments and improvements before the official release. This method of testing allowed us to ensure that the game was not only functional but also enjoyable for players of all backgrounds and skill levels.

5.2.2. Conclusion of the testing phase:

After conducting both the internal testing and external testing methods, we discovered a significant number of bugs in the game. We dedicated a considerable amount of time and effort to addressing these issues, which included correcting imbalances in game mechanics and fixing any errors in the game's programming. Specifically, we identified that certain elements of the game were too powerful and needed to be adjusted, as well as addressing issues where certain terrain tiles were mistakenly designated as impassable when they were in fact walkable. Through this rigorous testing process, we were able to ensure that the game functions as intended and provides a fair and enjoyable experience for players.

CHAPTER 6
CONCLUSION AND PLANS FOR THE FUTURE

6. Conclusion and Plans for the Future

This section will conclude the current state of the project and outline any future work that needs to be implemented to improve and advance the project. It will provide an overview of the progress that has been made thus far and identify any areas that require further attention or development. Additionally, it will consider potential opportunities for expansion or new features to be added to the project in the future.

6.1. Conclusion:

As we come to the end of our five-year journey at FCIT, we have successfully developed a fully functional video game using Unity. Throughout this project, we were able to apply a wide range of computer science concepts and theories to create a dynamic and engaging gaming experience. One of the key components of our project was the implementation of various algorithms for pathfinding, including A*, Uniform, and BFS. Through extensive testing and analysis, we were able to determine the relative performance of these algorithms in our game. We found that the A* algorithm performed the best, followed by Uniform, and finally BFS. This project has been a challenging but rewarding experience, and we are confident that the skills and knowledge we have gained will serve us well in our future endeavors. Additionally, this project could be considered as a steppingstone for future projects and research for us as computer science graduates, and we look forward to continuing to develop and improve our skills in this field.

6.2. Future work

As we conclude the development of our video game, we have several plans for future work to improve and expand the player experience. First and foremost, we aim to release the game on mobile devices to increase accessibility and allow players to enjoy the game on-the-go. Additionally, we plan to implement a monetization system to generate revenue and continue to support the development of the game. This may include the creation of downloadable content (DLC) which will provide players with new content and experiences. Furthermore, we are considering the development of a randomly generated map mode, which will provide players with a unique and unpredictable experience every time they play. Lastly, we are also exploring the possibility of implementing a player versus player (PVP) mode, which will allow players to compete against each other in real-time. All these future works will help us to create a better experience for the players and make the game more engaging and fun.

CHAPTER 7
BIBLIOGRAPHY

7. BIBLIOGRAPHY:

Bulitko, V., Hernandez, S. P., & Lelis, L. H. (2021, August 17). Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding. *2021 IEEE Conference on Games (CoG)*. https://doi.org/10.1109/cog52621.2021.9618995

Hagelback, J. (2016, December). Hybrid Pathfinding in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, *8*(4), 319–324. https://doi.org/10.1109/tciaig.2015.2414447

Nakrani, S., & Tovey, C. (2004, December). On Honey Bees and Dynamic Server Allocation in Internet Hosting Centers. *Adaptive Behavior*, *12*(3–4), 223–240. https://doi.org/10.1177/105971230401200308

Sabri, A. N., Radzi, N. H. M., & Samah, A. A. (2018, April). A study on Bee algorithm and A algorithm for pathfinding in games. *2018 IEEE Symposium on Computer Applications &Amp; Industrial Electronics (ISCAIE)*. https://doi.org/10.1109/iscaie.2018.8405474

Sazaki, Y., Satria, H., & Syahroyni, M. (2017, October). Comparison of A and dynamic pathfinding algorithm with dynamic pathfinding algorithm for NPC on car racing game. *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*. https://doi.org/10.1109/tssa.2017.8272918