



L-Università ta' Malta
Faculty of Information &
Communication Technology

Beating the Pandemic – Game Jam 2020

Karl Attard¹ 203501(L), Ethan Zammit 4802(L)¹,
Samira Noelle Cachia Spiteri² 485800(L), Elena Fomiceva² 1809766

¹B.Sc. (Hons) Artificial Intelligence

²B.Sc. (Hons) Software Development

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines” (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

We, the undersigned, declare that the assignment submitted is our work, except where acknowledged and referenced.

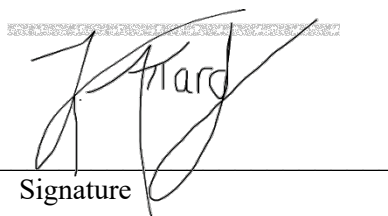
We understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Karl Attard

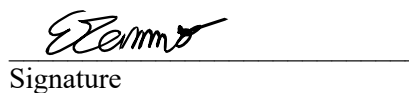
Student Name



Signature

Ethan Zammit

Student Name



Signature

Samira Noelle Cachia Spiteri

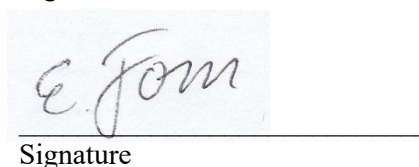
Student Name



Signature

Elena Fomicева

Student Name



Signature

ICS2211

Course Code

Beating the Pandemic – Game Jam 2020

Title of work submitted

21/12/2020

Date

Table of Contents

Teamwork Declaration	5
Attendance Log	5
Contributions Made by Each Team Member	5
Team Responsibilities.....	5
Group Dynamic	6
Project Build.....	6
Game Overview.....	7
How the Game Correlates with Theme	7
Flow of the Game	8
Outline of game mechanics	10
Player Movement.....	10
Player Combat.....	10
Enemies (Behaviour, Movement & Combat).....	10
Enemy Intelligence	10
Weapon systems	11
General Implementation.....	11
Dynamicity & Unexploited Potential.....	11
Animations.....	11
Boss Animator	11
Blend trees	12
Loot system.....	12
General Description	12
Implementation	12
Scene Transition.....	13
Game Audio	13
Game Modules and Components	14
Data Flow Diagrams (DFDs)	14
Level 0 DFD (Context Level)	14
Start Scene DFD.....	15
Initial and Final Cut Scene DFD	15
Main Menu Scene DFD.....	16
Options Menu DFD.....	16
Level Scenes DFD.....	17
A detailed description of AI Aspects	18
Pathfinding	18
Navigation Meshes.....	18
Pathfinding & Graph theory	18
Impact of pathfinding.....	18
Types of Pathfinding	18
Undirected	18

Directed.....	19
Why A* and not others for games	19
How A* specifically works	19
Why the 'A* pathfinding project' (Library of choice).....	20
Finite State Machines.....	21
How a Finite State Machines works	21
Why use Finite State Machines	21
Application to unity and our Game	21
<i>References.....</i>	22

Teamwork Declaration

Attendance Log

Due to the ongoing COVID-19 situation, the Game Jam for this year had to be held virtually and team members were not able to meet up. Hence, team meetings between our group were held frequently using Zoom. In these video calls, we began by first discussing extensively this year's Game Jam theme title and brainstormed many ideas. After coming up with an idea, we then discussed the tasks each team member will take. In addition, daily meetings were held to update each other on the tasks we were responsible of. Moreover, to facilitate the communication between us, we set up a WhatsApp chat where we regularly gave advice to each other.

Contributions Made by Each Team Member

During meeting calls, each team member discussed their progress on their task and used the "share screen" feature on Zoom to update everyone else on their task. With the help of this feature, team members were able to understand better how the game will eventually look like and nonetheless, we were all able to give our suggestions.

Moreover, to combine the whole project together, we exported all our packages and sent them on our WhatsApp chat. Then, during our video call, one team member would share their screen, and import all these packages.

Team Responsibilities

Before we started working on the tasks, we were responsible of, we first searched extensively for sprites and sounds that we were going to use throughout our game and shared them with each other.

The tasks for each member were split in the following manner:

1. Karl Attard:
 - i. User Interface – main menu, options menu, pause menu, controls menu, player/boss health bars, background music and loading bar.
 - ii. Game story's cutscenes.
 - iii. Time, potion's text counter and player's nickname.
 - iv. Start Scene of the game
2. Ethan Zammit:
 - i. Player, its movement, combat, animations & documentation
 - ii. Boss & Enemies, their Behaviour, Movement, Combat, Animations & documentation
 - iii. Pathfinding, FSMs & documentation
 - iv. Weapons, their behaviours, animations & documentation
3. Samira Noelle Cachia Spiteri:
 - i. Tile maps
 - ii. Scene Lighting

- iii. Flame and door animation
 - iv. Presentation Editing
4. Elena Fomicева:
- i. Loot System: item spawn upon enemy death, player picking up items – potion collection, virus damage, weapon equipment, end-of-game book.
 - ii. Scene transition with fade-in/fade-out animation.
 - iii. Audio and sound effects in the game.

Each member also discussed their own parts in the video and this report.

Group Dynamic

The experience was overall a positive one and there were no issues with our workflow, or team conflicts. Frequent video calls and chat messages aided us to work together in a highly efficient manner, almost as though we were in the same room.

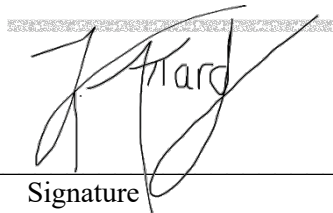
Project Build

This is the Google Drive link for the project build (includes video presentation):

https://drive.google.com/drive/folders/1abWwvwt_GB9CKKOslEoFHN_EhoAmLxa2?usp=sharing

Karl Attard

Student Name



Signature

Ethan Zammit

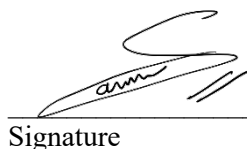
Student Name



Signature

Samira Cauchi Spiteri

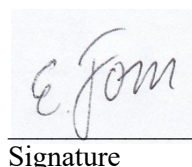
Student Name



Signature

Elena Fomicева

Student Name



Signature

Game Overview

How the Game Correlates with Theme

This year's Game Jam theme was "Beating the Pandemic". After brainstorming several distinct ideas with the group, we decided to move away from the current pandemic we are living in. Hence, we chose a deadly pandemic which struck the world during the middle ages, i.e. 'The Plague' which is also known as 'The Black Death'.

This pandemic was very brutal, and historians suggest that during that time, there was this well-known physician called 'The Plague Doctor' who used to treat victims. Therefore, as our main player, we decided to choose this character. The story behind this game is that this character is on a mission to find out the evil person who is known to have invented this pestilence as well as has the cure. Hence, the player must complete all the levels (by killing all enemies in each level) and then kill the boss (the suspected character). Once the player successfully kills him, this boss will drop a mysterious book, which the player then finds out that he has found the cure for the rest of the world (See figure 1 and 2 for more elaboration of story).

To continue accommodate this theme, we chose the majority of the enemies to be rats and bats which were known to be major carriers of the virus back then. These enemies can drop two kinds of things when they die: potions or viruses. The potions were implemented to resemble a medicine to help the player increase their health whilst the virus is deadly and harmful for the player. Moreover, since this plague happened during the Middle ages, all our User Interface were in a Medieval kind of setting. Nonetheless, the dungeon was created in a horror manner (with dark colours and light flickering) to highlight the depressed times of this brutal plague.

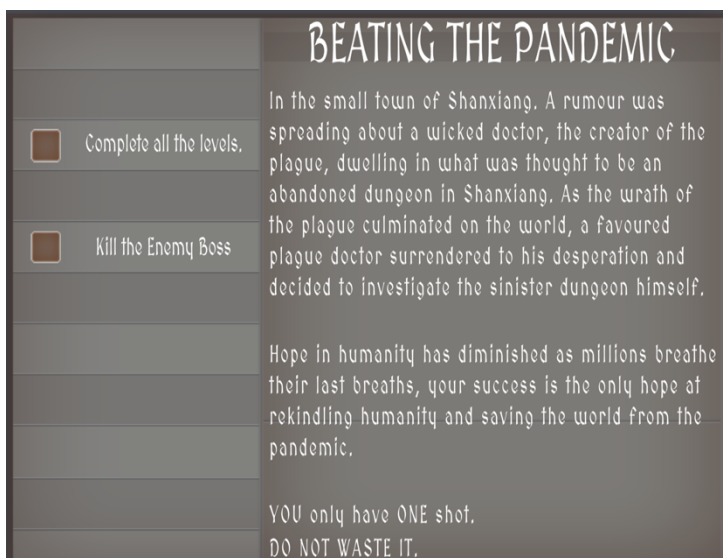


Figure 1: Initial Cut Scene – Beginning of the story

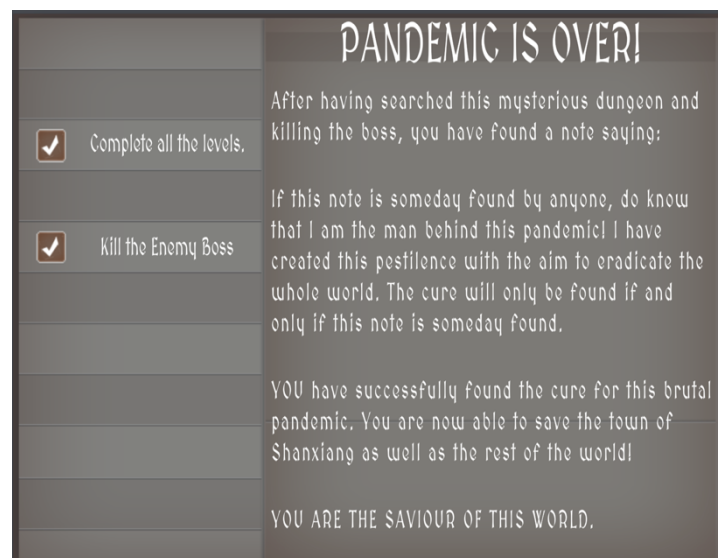


Figure 2: Final Cut Scene – End of story

Flow of the Game

In this section, a brief explanation of how the game is played will be outlined.

The first scene that the user sees when playing this game is the start scene, which essentially allows the user to pick their character nickname. Once the user submits their chosen nickname, the initial cut scene is loaded – the story behind this game begins from here. After a number of seconds (to allow the user to read the story text), the user will be redirected automatically to the main menu of the game. Within this menu, the player is allowed to either start the game, go to the options menu (to adjust sound, full screen and/or player controls) and they are also able to quit the game.

If the player decides to start the game, then the most basic level is loaded. This level will essentially show the user the controls of the game and then they can start playing. In each of the levels, the player must kill all the enemies which then would enable him/her to go to the next level. All levels are implemented in such a way that once all enemies are killed, then the door to the next level is activated. To make the game more intuitive, sounds were implemented, for instance, door unlock, rats squeaking and much more. It is important to note that as the levels increase, the difficulty also increases. Moreover, to make the game more interesting, we added a secret room which is found in level 2. Within this secret room, the player finds themselves in a room full of bacteria, hence, they are given a fumigator as a weapon to kill all the bacteria. In this room, the player has a high probability of landing a great weapon which he can then use throughout the whole game.

In addition, we added a loot system. This means that when enemies die, they can drop either potions or harmful viruses. The potions are great asset for the player as he can pick up as much as he desires, and these will increase their health. On the other hand, the viruses are harmful, and these will damage the player drastically. This loot system is based on a probabilistic feature meaning that the higher the level, the more probability of landing a good weapon and more potions!

In the last level, after the player has killed all the enemies, the boss is spawned. The boss is the most difficult enemy in the game to defeat and if the player is able to kill him, this boss will then drop a book on the floor. The player then must pick up this book which will in turn redirect them to the final cut scene – the end of the game story.

NB: SEE DIAGRAM ILLUSTRATED IN FIGURE 3 TO SEE FLOW OF GAME AS DIAGRAM.

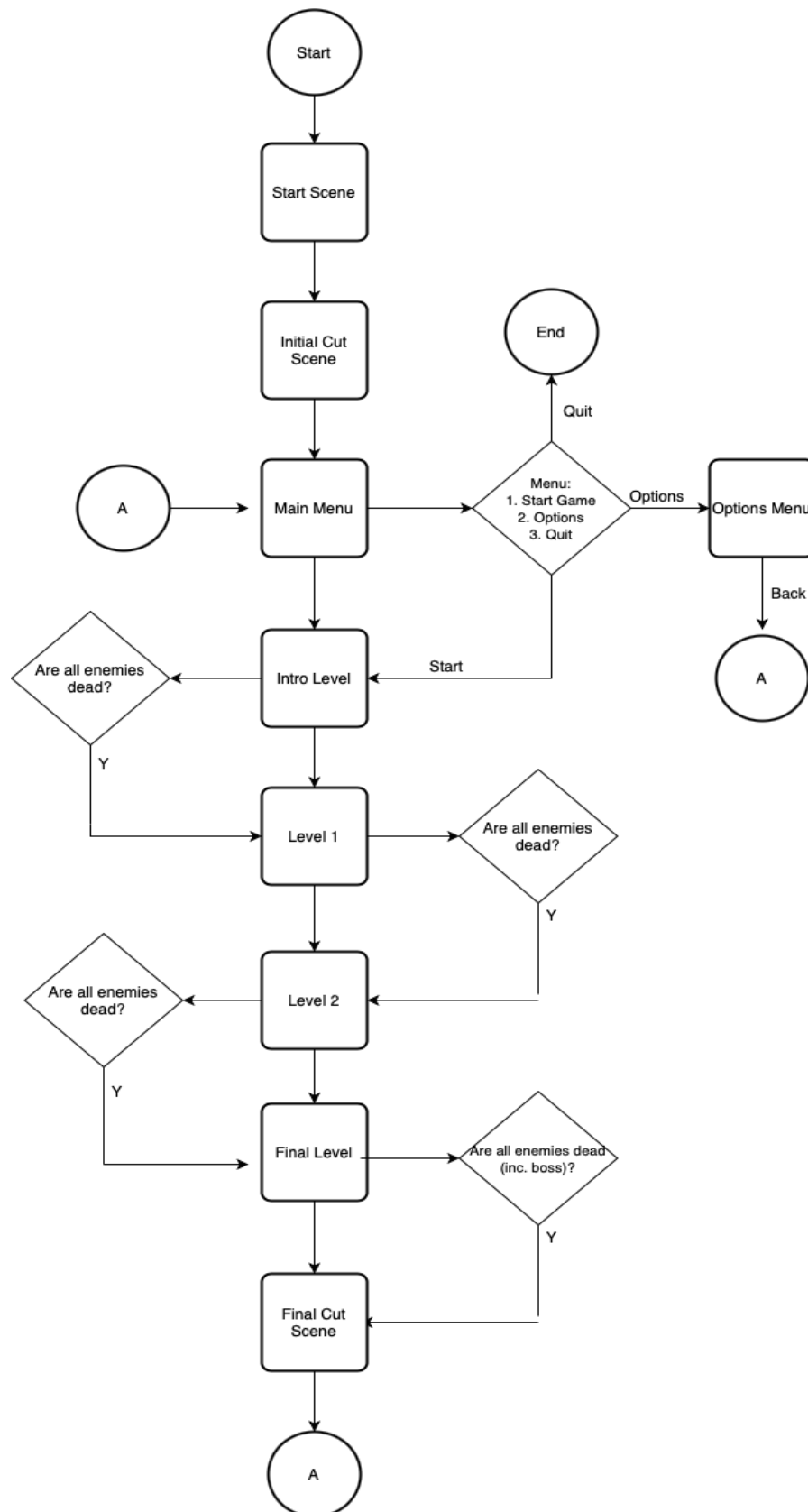


Figure 3: Gameplay Flowchart

Outline of game mechanics

Player Movement

The player movement implementation in this game was quite simple. The `Input.GetAxisRaw()` method was used (for vertical and horizontal axis) in the `Update` method. This allows for greater flexibility rather than using `'Input.GetKey'` for the inputs. Such as allowing to user to use 'w/a/s/d', the arrow keys, or even a gamepad! The 'Raw' version of this method was used to prevent unity from adding any smoothing effects to the movement, this allows the player controls to feel snappier.

The inputs were saved as a normalized vector which is then accessed by the `FixedUpdate()`. This method is opted for since it runs in sync with the physics engine and is usually preferred for movement.

Player Combat

The player combat was designed to allow for flexibility in the use of weapons. The player itself does very little work to attack, all it does is identifies which type of weapon it has (could have been avoided by using inheritance in weapons), and then call the `use()` function on their equipped weapon. This allows the player to change weapons on the fly without changing any functionality from itself.

Enemies (Behaviour, Movement & Combat)

The enemy implementation for this game revolves a lot around the use of the pathfinding library.

When an enemy is spawned, by default, it immediately starts pathing towards its pathfinding target (this can be changed). This was opted for since there were no instances where the enemy would just rest idle or need to patrol (since enemies were spawned by controlled spawners which only spawn enemies when needed). The pathfinding library also allowed me to know when the enemy reached (to a minimum distance) their target.

This functionality was exploited to make the enemy use their weapon when they arrive at their destination. Their weapon is used every 'attackSpeed' seconds.

Enemy Intelligence

The enemies were quite intelligent, as they only attack when they are close enough that the range of their weapon hits. They also always knew how to get to the player thanks to the pathfinding. Logical states were also sometimes implemented by using the animator a pure example of this was in an enemy which unfortunately **did not make it in the game** due to theme restrictions. A slime enemy was implemented which when close to the player, it used to start spinning and increased its movement speed. The slime used pathfinding to get to the player's location and attack him by spinning into him. The logical states were used to disable the collider of the slime so that could pass through player and enable a trigger instead so that it could still apply damage. The slime only started spinning when it was close enough to the player, having a chance to hit him, and its properties dynamically change on the fly according to the state that it is in.

Weapon systems

General Implementation

The weapon system was implemented to be flexible and scalable. The only thing which was needed to add a new weapon was adding the sprite and attaching the relevant weapon script (ranged or melee or enemyRanged or enemyMelee).

For melee weapons, one can just define a range of attack and add an animation with the “attack” trigger. When the weapon is used, the attack range is scanned, and the damage is applied to all hittable entities in the area (if any).

Ranged weapons also worked flexibly. The weapon always pointed at the cursor of the player, this allows the user to simply aim with their cursor and shoot. When the weapon is used, a predefined prefab is instantiated and given force towards the cursor. Shots had their scripts which deal damage upon collision.

Dynamicity & Unexploited Potential

Weapons were implemented quite early in the game jam and the positive impact of inheritance was not known. What I had in mind was weapons being very dynamic and allowing both players and enemies to use weapons. This was done by separating most weapon logic from the player and putting this logic into the weapon scripts themselves. This allows the player and enemies behaviour to stay the same, while the different define their behaviour with a few accesses from the player or enemy. Although this mostly worked as intended, a lot of redundant work was done; for example, the difference between the enemy weapons and player weapons were very minimal and could have easily been a Boolean rather than whole new scripts. Yet I am still quite happy with the level of flexibility given the time constraints.

Animations

Animations were mostly triggered by using triggers in the animator. Since most animated entities only had a few states this was ideal. For example, most enemies only had two states, moving, and attacking, thus, to implement we just trigger the attack when we are using our weapon, and let the animator take care of the rest (we usually play the attack animation one time, then go back to moving state). This also allowed for the animations of entities to work hand in hand with what they were doing (very easy to play an attack animation when the player is attacking, or a move animation when the player desired velocity is not zero).

Boss Animator

However, the animator allows us to define different behaviours based on the different states the entity is in.

For the boss, this was exploited extensively. To use both the enemy behaviour and the use of Finite state machines, the bossBehaviour was defined as a subclass of the enemyBehaviour, and several scripts were implemented for the different states of the animator. The states were then used to communicate the bossBehaviour script and unlocked for a lot of potential, as this allowed the boss to behave and act differently based on the communication it receives from the states and allow it to change states to reflect its current behaviour. This cycle of communication allows for a lot of dynamic decisions. Let us take an example:

The boss starts as calm, and thus walks slowly and only does one attack. This is communicated with the bossBehaviour. When the boss has less than half of its health (managed by the bossBehaviour), this is communicated with the animator, and the boss changes state to enraging, and later angry. This makes the boss run faster and allows it to choose between 3 different attacks. The next attack is randomly chosen by the bossBehaviour, and this communicates with the animator to call the respective attack animation when it must.

Blend trees

Blend trees were used for the player animations, they were used to show the respective player animation based on the x and y values of the player movement. (walking left/right/up/down)

However, their full potential was not exploited. Blend trees allow the animator to choose (and blend) states based on several variables. For example, if the player is looking forward and moving left, an animation is played where the player faces forward but its leg movements go backwards.

Loot system

General Description

The loot system was designed to allow the player to get various items upon enemy death. The system is flexible, so that the items dropped, and probability of their appearance can be easily changed. The loot in the game is represented mostly by healing potions to help the player recuperate some of the health lost in combat. However, to make the gameplay more challenging and in order to emphasize the pandemic theme of the game the enemies also drop viruses that deal damage to the player. In this way the player should be extra careful when picking up items dropped by the enemies.

In the first three levels of the game the player has a chance to get a weapon as loot: a sword or a bow. This is done with the idea to make the game more fun and diverse, as well as to allow the player to get a feel of different types of weapons: melee and ranged. In the final scene the loot drop parameters are automatically changed to give the player more potions as this is the level with more powerful enemies and the final boss encounter. In addition, in the final level the player will not get any weapons as loot because the best way to beat the boss is with a bow.

The loot system is also used when the final boss is killed, and the end-of-the-game book is spawned. This book takes the player to the final cut scene.

Implementation

The loot system is implemented in LootSpawner script attached to the GameManager game object made persistent across all scenes with the help of DontDestroyOnLoad() function. In this way, the chance to get loot and the selection of lootable objects can be easily changed in the Unity Inspector. The automatic modification of the loot system in the final level is performed by ChangeChance() function that increases the overall chance of loot drop and turns off the spawn of weapons as loot.

In addition, each lootable object prefab has BoxCollider2D component and attached PickMeUp script that defines the player behaviour upon collision with the particular item in the game. Different objects trigger different behaviours: a potion is collected by the player and can be used any time by pressing Q key on the keyboard, a virus immediately deals 20 damage to the player, and a weapon is instantly equipped, but cannot be collected for future use.

The most challenging part was the implementation of the equipment of a new weapon. The WeaponBehaviour scripts attached to the weapon prefabs threw exception errors if a weapon prefab appeared in the game not attached to the player object. In this case the decision was taken to turn the scripts off on the prefabs and activate them only when the player picks the weapon. All this functionality is implemented by the EquipWeapon() function in playerBehaviour script.

Scene Transition

As all the levels are organized in separate scenes, there was a need to transport the player to the next level while storing his health, number of potions collected, and weapon equipped. This was done using the `DontDestroyOnLoad()` function.

In addition, to make transitions between scenes smoother, the fade-in/fade-out animation was used. This effect was implemented using Panel UI object with `SceneChanger` script and `BoxCollider2D` positioned on the exit door of each level. When the player collides with the door, the fade-in animation is triggered. The `SceneChange()` function is called at the end of the animation to load the next scene. When the next level is loaded, the fade-out animation is played. Furthermore, each scene has a `PlayerSpawnPoint` game object with `PlayerSpawner` script attached to it, which makes the player appear exactly near the level entrance door.

The `BoxCollider2D` component of the Panel is initially deactivated and gets activated by the script only when all enemies and enemy spawners in the level are destroyed. Thus, the player cannot directly proceed to the next level without engaging in combat with the enemies. To inform the player that he can move on to the next level, the collider activation is followed by the door unlocking animation and sound.

Game Audio

In order to create the special gloomy atmosphere for the game, different music and sound effects were used. The background music is persistent across all levels. In addition, there are enemy sounds – cries of rats and bats informing the player that there are enemies around.

Different player actions are accompanied by sounds: when the player hits an enemy, picks up an item, equips a weapon, drinks a potion, opens a door, etc. The sound effects were implemented in the `SoundManager` script attached to `GameManager` object along with the `AudioSource` component. The script determines which sound to play with the help of a switch statement.

Game Modules and Components

Data Flow Diagrams (DFDs)

DFDs are graphical representations of how data flows in a system. These describe the processes involved in a system to transfer data from the input to the file storage.

Level 0 DFD (Context Level)

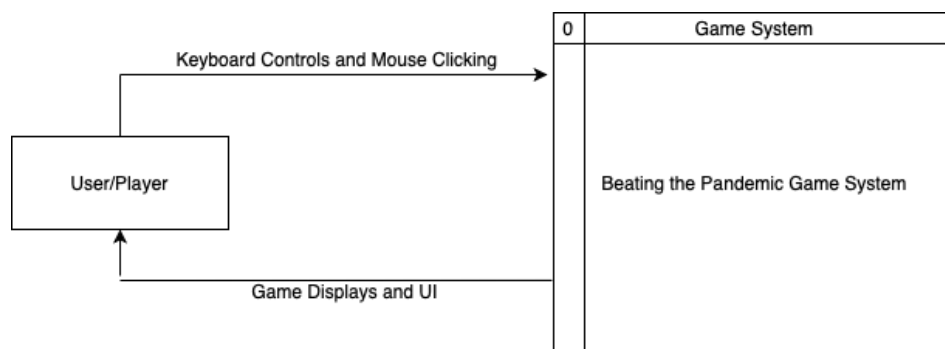


Figure 4: Level 0 DFD

Start Scene DFD

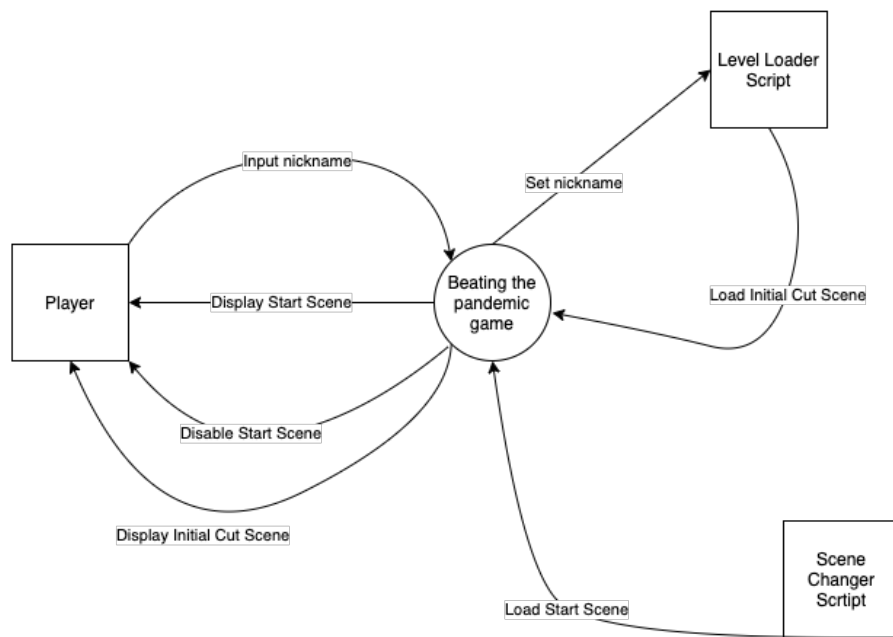


Figure 5: Start Scene DFD

The above diagram illustrates the first scene of the game, where the start scene is loaded and displayed to the player. When the player inputs a nickname and presses the submit button, the nickname is stored, and the initial cut scene is then loaded and displayed.

Initial and Final Cut Scene DFD

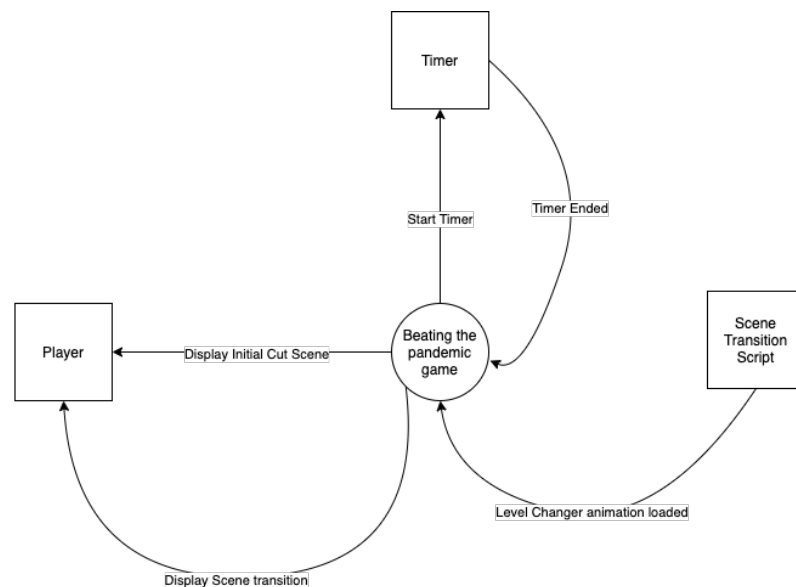


Figure 6: Initial and Final Cut Scenes

This DFD shows that once the initial cut scene is displayed, a 30 second timer is started in order to give the player time to read the prologue. At the end of the timer the scene transition animation is run in order to transition to the main menu scene. The final cut scene's DFD is similar to the above diagram with the only difference being that the user returns to main menu scene upon clicking on the button.

Main Menu Scene DFD

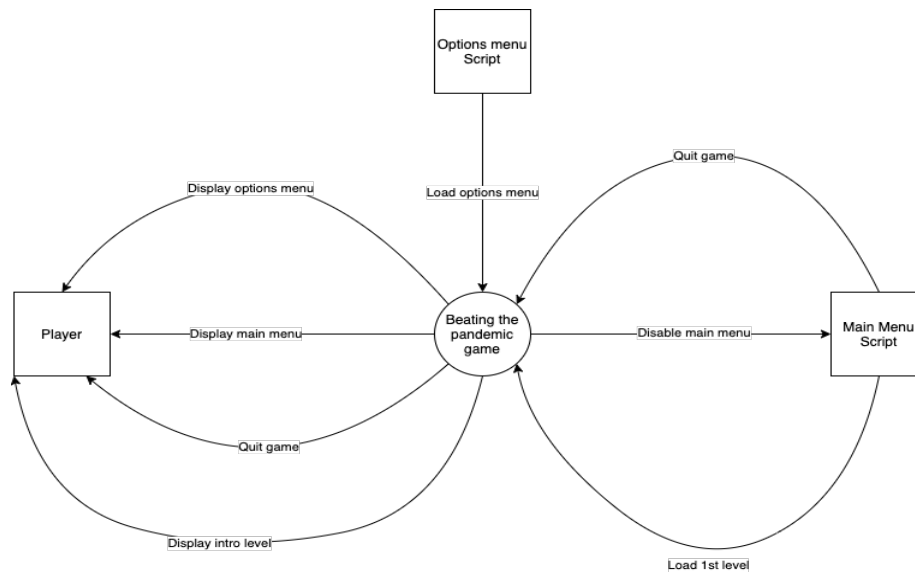


Figure 7: Main Menu DFD

The above DFD shows the main menu scene being displayed to the user. The main menu has 3 options: Start game, Options and Quit. When “Start game” is selected, the main menu scene is disabled, then the intro level is loaded from the main menu script and outputted to the player. When “Options” is selected, the options menu is shown (main menu canvas is disabled and options menu canvas is enabled). Finally, if “Quit” is selected, the game stops.

Options Menu DFD

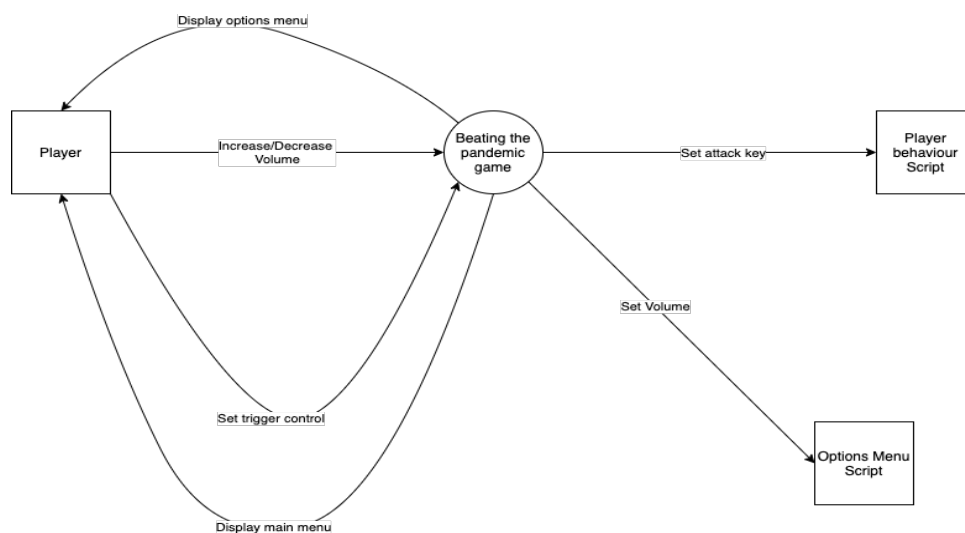


Figure 8: Options Menu DFD

In the options menu, the player is able to adjust the background music and set the attack key which by default is a mouse left button click. These values are then stored in their respective scripts. Finally, as soon as the player presses the back button, the main menu is re-displayed.

Level Scenes DFD

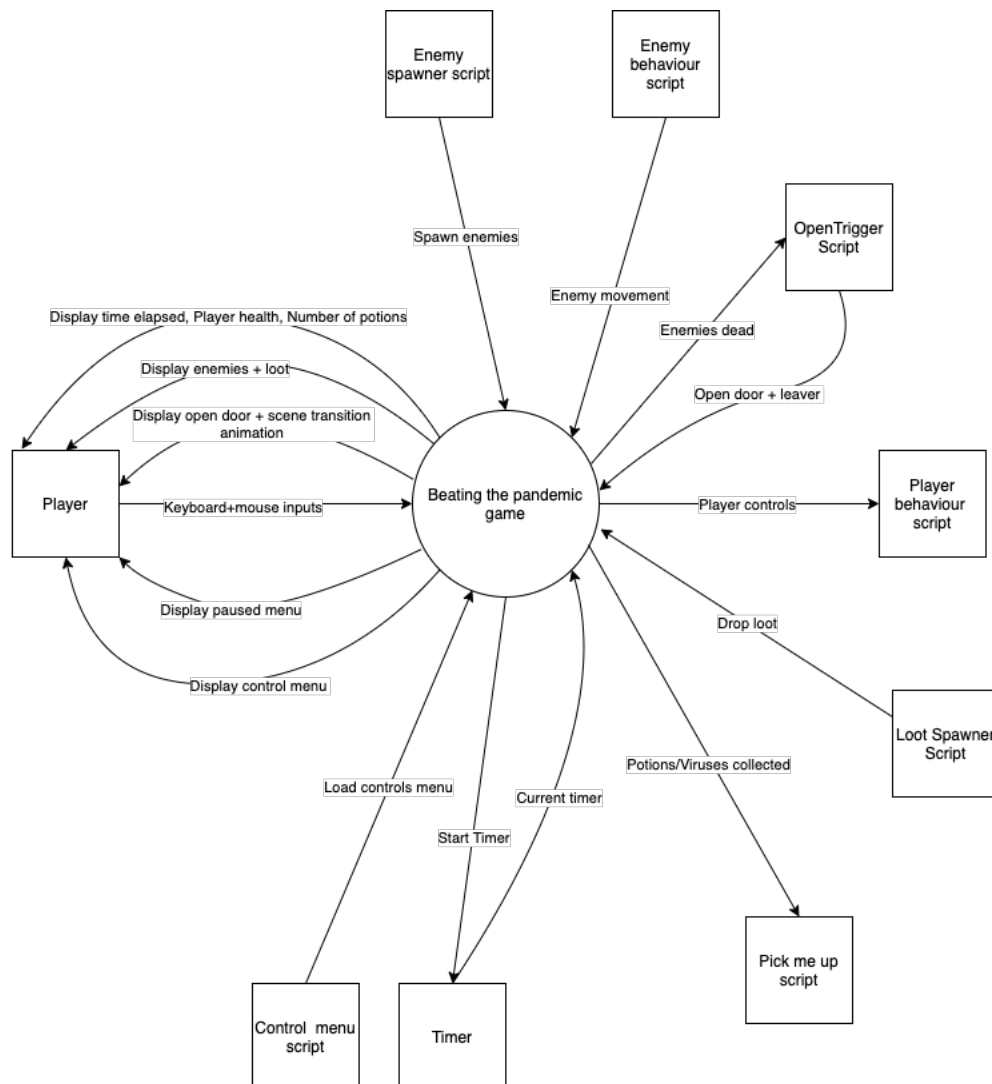


Figure 9: Level Scenes DFD

When the intro scene is loaded, the control menu with the list of controls is shown to the user. When the player closes this menu, the player's health bar and number of potions collected, and timer is displayed. The user then input keyboard keys to control the player in the game. When the player passes certain areas, the enemy spawners are triggered, and enemies start appearing. When enemies are killed, some may drop potions, weapons or viruses which can be collected by the player. The probability of enemies dropping loot is controlled by the loot spawner script and the loot collected by the player is stored and processed by the Pick me up script. The player health bar and number of potions keep on updating until all the enemies are killed which in turn will trigger the open trigger script which will animate the leaver and door on the tile map which the user will then have to walk through in order to be able to move to the next level.

The above dfd can be applied to all the levels of the game with the only exception being the final level, where an additional script called bossbehaviour script is used to control the behaviour of the boss.

A detailed description of AI Aspects

Pathfinding

Navigation Meshes

As described by [1], Navigation meshes are very popular when it comes to pathfinding in games, put simply, this is a predefined walkable area or surface of an environment (allows to define obstacles such as walls, holes, tables, etc.) when in complex environments the mesh is built using polygons, to better match the walkable areas. Each tile must also lead directly to another polygon so that all polygons are reachable. Each walkable polygon can be used as a node for the algorithm to calculate paths on.

When the two locations are set (the entity and the target), a navigation path is calculated, this is a set of walkable polygons, which will lead the entity to the target.

Pathfinding & Graph theory

According to [1], Pathfinding algorithms can be used once the map is encoded as a graph, which enables for powerful tools gained from graph theory. Not only does this allow for more ways to tackle pathfinding, but since graph theory is heavily researched and optimized, most optimizations also apply for pathfinding optimization.

Impact of pathfinding

Pathfinding allows entities to make more intelligent decisions, allows them to navigate around obstacles on the fly, and not have to follow predefined paths. The algorithm has the responsibility of defining the walkable space, understanding and using opportunities for paths within it.

Types of Pathfinding

There are two major types of pathfinding algorithms, on one hand, we have directed and on the other we have undirected. These two main types were compared in [1], in the following paragraphs, some differences were extracted and applied to the scope of this documentation.

Undirected

This approach to pathfinding assumes quite a naïve entity and is resembles a rat in a maze ‘brute forcing’ its way out of it by walking randomly. There is no preplanning involved here and all the energy is spent on blindly moving around. There is also a major flaw here in which the rat may never find a way out, and just repeatedly go in between two dead ends. However, there is still an application for such an algorithm; due to the lack of planning involved, the entity can immediately start moving, whilst in the background, it is buying time for a more complex algorithm to complete.

There are two main approaches which drastically improve the efficiency of such an approach, being the breadth-first search and the depth-first search. The BFS works by encoding the world as a large, interconnected graph of nodes. It checks all nodes which are adjacent to the current node, after which it checks each node connected to these nodes as well, going on recursively. This also means that if a path exists, the BFS will find it. (Given enough time). The DFS is the opposite of the BFS, in which it recursively looks at all children of each node before looking at the rest, thus generating a path to the target. If it finds a dead-end it backtracks and tries to search in shallower nodes.

The difference between the two approaches is also clearly highlighted in Figure 10.

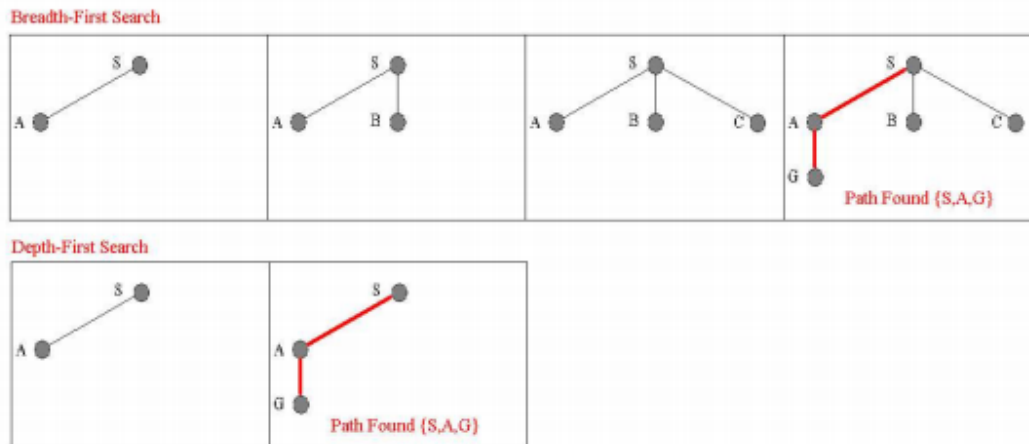


Figure 10 : BFS vs DFS

In Figure 10 above we can see how the two methods search differently, one searched all children of a node, whilst the other searched the whole first level of the current node before proceeding.

Directed

This method of pathfinding is usually better suited for games as we have a lot of power in knowing the locations of entities and targets. This method can be summarized as having an idea of the progress being made, it is aware of the progress done (if it is towards the target or further away).

There are two main approaches here as well, being Uniform cost search and Heuristic search.

UCS approaches the problem by always choosing the lowest cost node, this means that the approach is complete (finds path if exists) and minimised (shortest path), yet in certain applications, it is very inefficient. An example of such an algorithm is Dijkstra's algorithm.

Heuristic search estimates the costs from the subsequent nodes to the goal, which allows for much faster search time, yet it is never guaranteed to find the correct path, nor is it minimised.

Why A* and not others for games

As highlighted in [2] the A* algorithm is very applicable in games, and performance improvements over other methods have been observed. This is likely because A* algorithm is a **directed** algorithm which combines both UCS and Heuristic search, which minimises the total path cost (planning and movement). The A* returns an optimal path and in most cases/applications are more efficient than Dijkstra's. This explains why the A* is the driving pathfinding algorithm behind almost all modern computer games.

How A* specifically works

Now that we have seen the different possibilities and how each one is used; we can delve a bit more about how the algorithm of choice works. [1] and [2] give a very detailed and intuitive explanation of how the actual algorithm works.

First, the map is pre-processed and broken into points (polygons or squares), walkable points are encoded as nodes, these nodes will be used as a measure of progress to the search destination. Each node has three properties, being fitness, goal, and heuristic. The goal being the cost of getting from start node to the current node (usually sum of all nodes between the start and current nodes). The heuristic is the estimated goal from the current node to the goal (different heuristics can also be chosen such as Euclidean, Manhattan & diagonal). Finally, fitness is the sum between the goal and the heuristic, which usually denotes that the lower the fitness the better. Nodes are also able to be part of two lists, the open

or the close list. The open list contains all nodes which have not been fully explored yet, whilst the closed list holds the opposite (A node is fully explored when all linked nodes have been explored).

The usual search put as pseudo-code goes as follows:

1. Pre-process the searchable space
2. Select start node
3. Check nodes which surround the node
4. Selected best node and let it be X
5. If best node is goal -> quit, the path found
6. If the open list is empty -> quit, a path cannot be found
7. Is a valid node connected to X?
 - i. If so check if the found path has a lower cost in which case update path
 - ii. Else add the node to the open list
8. Repeat step 7 for all valid nodes
9. Repeat from step 4

Why the 'A* pathfinding project' (Library of choice)

The 'A* Pathfinding project' was chosen as the pathfinding library for this game. This library was chosen as it allows for a lot of flexibility. Most features can be customized according to your needs, and it also allows the use of 2D environments. It is also quite commonly used and thus a whole community is behind it, and a lot of help can be found. It is so flexible that it was used for the enemy combat as well, as we can start attacking when we are in a certain range of the destination/player. More information regarding the library can be found here: <https://arongranberg.com/astar/>.

Finite State Machines

As accurately defined in [3], Finite state machines are considered as one of the simpler AI models, contributing to their popular use in games. A finite state machine as defined consists of a set of connected states, which are connected in a graph by transitions (think triggers) between them. An entity starts at a predefined initial state and waits for triggers/events which will transition it to another state, where it can have different animations, behaviours, rules, and triggers. An entity can only be at one state at any given time.

How a Finite State Machines works

As mentioned in [3], A Simple FSM implementation consists of states, transitions, rules, and events.

States is a set of finite states which the entity can choose between.

Transitions are the relations which connect states and define which states an entity can go from its current state.

Rules are sort of listeners which decide whether to switch states or not (such as enemy is close to the player, health below a threshold, player line of sight etc.)

Events are triggered to enforce rules (such as distance from player, health, visible area etc.)

Why use Finite State Machines

Finite state machines are a very common AI pattern in games since they are quite easy to implement, especially in unity due to the visibility of such states in the animator. They are also very intuitive as one can easily visualize the states and the transitions, making them very easy to understand.

Application to unity and our Game

The unity animator enables for quick and easy implementation for such states. It allows us to define a script for each state and access the game object and a lot of features through it. Thus, it was exploited throughout the whole game, ranging from applications to weapons, the player, enemies and most importantly the boss. They were crucial when applied to the boss as through the communication between states and the bossBehaviour script, several features were merged to unlock a lot of flexibility and stage changes which are felt and visualized.

The figure below illustrates the final state machine (FSM) for the boss:

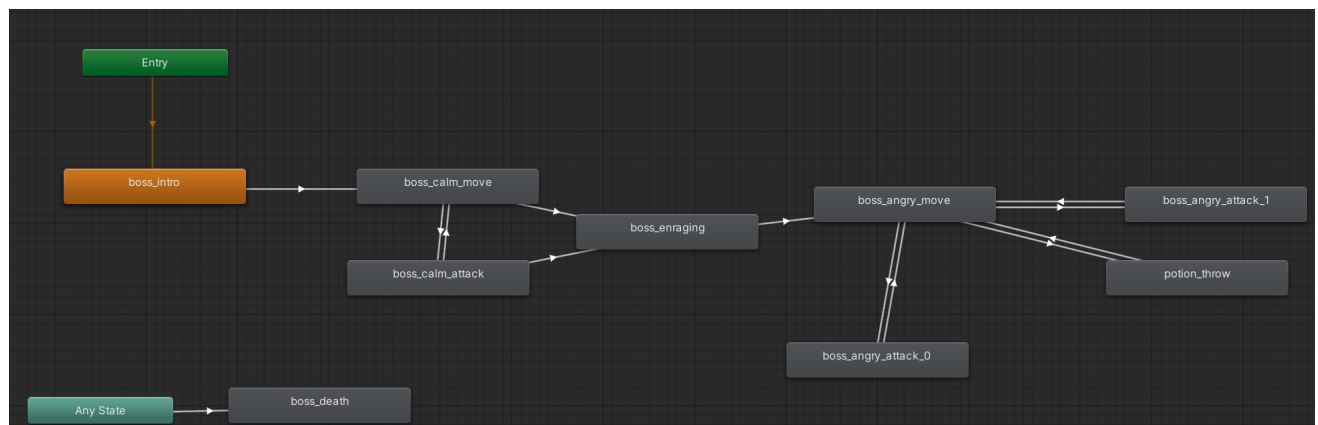


Figure 11: Boss FSM implementation

From the above figure, we can see how the boss starts with an idle animation during which he does not move, the movement is controlled by accessing the pathfinding script from the script associated with the state. After the animation is over, the boss transitions to the moving state, which he follows the player by utilizing the pathfinding, and when he is close enough, he goes into the attack state, which plays the respective animation, and his melee weapon is used. After the boss health is under a certain threshold, the enraging state is triggered, in which he stops moving and plays a relevant animation. After which he starts running and has 3 different attacks which are randomly chosen at runtime. As we can see, this allows us to give a sense of progression to the behaviour of the boss, where he starts as easy and calm, and then he starts running, kicking, and throwing potions.

References

- [1] R. Graham, S. Sheridan and H. McCabe, “Pathfinding in computer games,”, vol. 4, 2003, p. 6.
- [2] X. Cui, S. Hao and H. Shi, “A*-based pathfinding in modern computer games,”, vol. 11, 2011, pp. 125-130.
- [3] R. Barrera, C. Peters, A. S. Kyaw and T. N. Swe, Unity AI Game Programming, Packt Publishing Ltd, 2015.