# *FROST*

Capstone Final Report

*Keegan White (WHTKEE004), Jesse Mark (MRKJES003), Luc Hayward (HYWLUC001)*

## Abstract

The planning and design phase of the Capstone project resulted in the creation of the idea of a role-playing, arena-combat game that can be played both in single-player and split-screen. Through the testing and implementation phase, a body of work emerged that combines the groups Computer Science and Games Development knowledge into one, producing an enjoyable and challenging gaming experience. The player battles through waves of enemies unlocking different areas of the map on their way with the ultimate goal of beating their high score.

## 1.1 Introduction

Frost is an arena-style game that allows the player or players - as there is the option to play in split-screen - to fight against an array of Mutants, Zombies and Vampires in a fantasy combat amphitheatre known as the Frostlands. When in split-screen mode, the players team up to try to survive as many waves of enemies as they can, with their score increasing with the number of enemies they kill. Scores are measured individually, adding a competitive aspect to the game so not only are you trying to survive together, but you are also trying to outscore each other. The scoring system is the same in the single-player version of the game.

The Frostlands play a pivotal role in the game. It is filled with enchanted items that players can use to aid their survival and have a direct effect on not only enemies but the players as well. There are environmental traps and healing zones that can be activated using Frost Essence. Frost essence is a key aspect of the game. It is dropped when enemies die and is used to unlock new areas of the map and enable players special abilities as well as the healing zones and traps.

The player will embark on their journey through the Frostlands as a wizard, named Nero, that has both melee and ranged attacks. The ranged attack shoots ice shards at enemies and the melee attack uses his staff as a weapon. Nero also possesses two special abilities that are powered by frost essence. He can stun enemies that have been hit by his ice shards and he can heal himself and his teammate given that he has enough Frost Essence to do so.

Throughout the project, an agile approach to software development was taken with working prototypes being the major milestones along the way. We focussed on upholding key aspects of agile development such as always having a functional product and working in an iterative manner. We used a project management tool called Glo which emulates the workflow found in other project management tools such as Trello and Slack. This allowed us to track tasks and subdivide them into ‘To-do’, ‘In Progress’, ‘Review’, ‘Polish’ and ‘Completed’ subcategories. We set weekly goals on the tasks that we needed to accomplish and at the end of each week we reviewed each other’s code, tested the prototype and set further deadlines for the following iteration, thus following a cyclical development process. We utilised Git throughout the project with Github as our remote git hosting. Our master branch was always occupied by a completely functional prototype meaning that at no point in the project did we have a non-functional product.

## 1.2 Requirements Captured

### 1.2.1 Functional Requirements

- The game must be developed in Unity and make use of C# scripts.
- The game must allow for user input in the form of movement and attacks, in other words, the player must be able to control their character and interact with the world they are immersed in.
- The player must have health and be able to give and take damage.
- Enemy agents must have health and be able to give and take damage.
- The game must have both single and multiplayer options.
- The character must be viewed from a third-person camera.
- The game must be played on Xbox controllers for single or multiplayer or alternatively on mouse and keyboard if there is only one player.
- The game must have infinite levels with an ever-increasing number of enemies and difficulty.

### 1.2.2 Non-functional Requirements

- The game must be playable on Linux, Windows and Mac OS computers.
- The game must run at a consistent 30fps on all platforms with minimal frame drops.

### 1.2.3 Usability Requirements

- Xbox controllers or a mouse and keyboard are required.

### 1.2.4 Core Use Cases

Start Game

Actor: Player

The player will select create a new game. The player then selects between single and multiplayer. Where multiplayer is selected the system changes the view to split-screen. The system then sets up a new game and the players are loaded into the world, ready to begin.

Gameplay Loop

Actor: Player

The players explore the world. When the players locate enemies, the players initiate combat with enemies. Where the players are seen by the enmy without initiating combat themselves, the enemies will initiate combat with players. Players use their attacks and abilities to defeat enemies. Players defeat enemies by reducing their health points to 0 and gain experience by doing so. When the players defeat all enemies in a wave, a new level will begin. Where the players’ health points are reduced to 0 the player dies and the end game screen is displayed. Where only one player dies in a multiplayer game, the wave does not restart. The player attacks the rest of the enemies on their own. The dead player respawns at the start of the next wave.

Player Attack

Actor: Player

The player chooses whether to attack or use their abilities. The system checks if the attack or ability is on cooldown. The player can attac if their selected option is not on cooldown. Where the attack or ability is on cooldown, the system displays the number of seconds before the attack or ability is available.

Player Death

Actor: Player

The player's health points are reduced to 0 and the game will end, with the end game screen being displayed. Where only one player dies in a multiplayer game, the wave does not reset and the player does not respawn.. The player must then complete the wave. Once the wave is complete the dead player respawns. Where both players die the end game screen is shown.

Exit Game

Actor: Player

The player opens the pause menu. The player selects exit game. The system closes the game.

### 1.2.5 Analysis Artefacts

The main analysis artefacts produced were a class diagram and use case narratives. The use case narratives drove our development and steered us towards a goal. The class diagram was constructed in pieces, where each team member constructed the portion of the diagram that their code represented. This allowed the whole team to have a better understanding of how the different game systems worked as a whole and allowed the team to work interchangeably on different sections of the code.

## 1.3 Design Overview

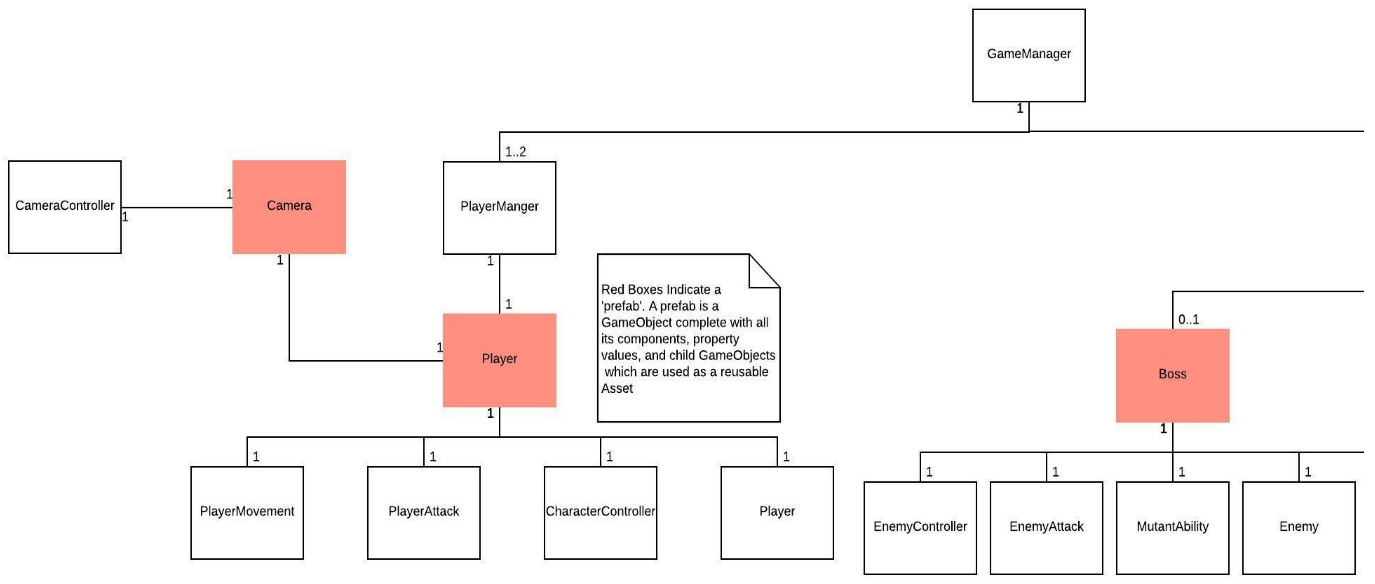

GameManager
1..2
CameraController
Camera
PlayerManger
Red Boxes Indicate a 'prefab'. A prefab is a GameObject complete with all its components, property values, and child GameObjects which are used as a reusable Asset
Player
0..1
Boss
PlayerMovement
PlayerAttack
CharacterController
Player
EnemyController
EnemyAttack
MutantAbility
Enemy

*Figure 1 Class Diagram*

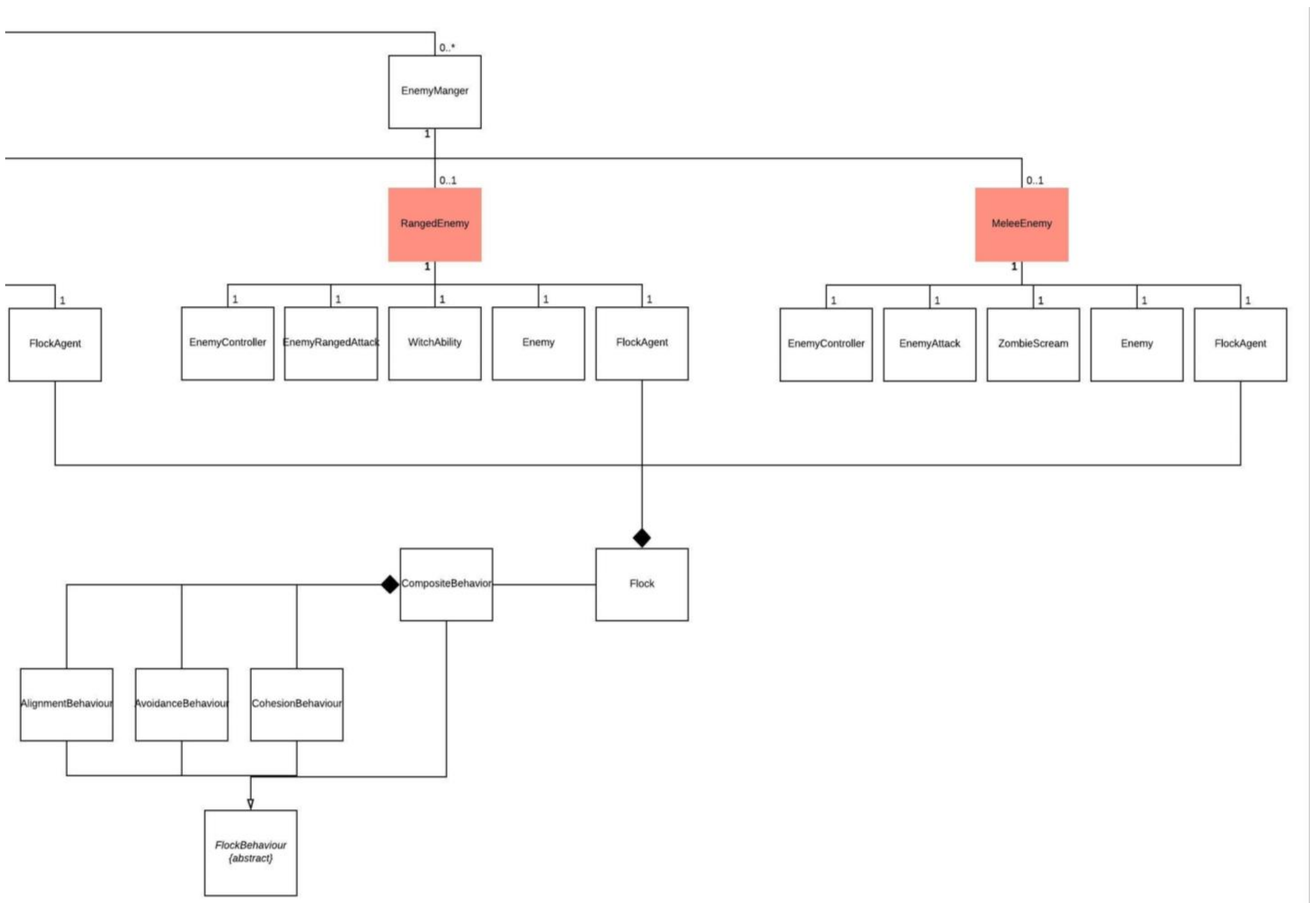


*Figure 2 Class Diagram*

### 1.3.1 System Architecture

The game state is managed by the Game Manager class that contains lists of Enemy Managers and Player Managers, which manage the enemies and players throughout their life spans, respectively.

Each Player Manager consists of a Player prefab, which is comprised of a Player Movement, Player Attack and Player script. These scripts manage the player’s ability to move, ability to attack the enemies and manage their base characteristics such as health while also managing key game events like the player’s death and the act of taking damage. There can be between zero and two player managers active at any point depending on whether the player has died, if it is a single-player game or if it is a multiplayer game. Each player is also assigned a camera that will follow them from a third-person angle.

The enemy manager class is associated with one of three different types of enemy prefabs, the Ranged enemy, the Boss enemy and the Melee attack. These enemy prefabs are all comprised of their attack script, ability script, an Enemy Controller, the Enemy base class and a Flock Agent script. The attack and ability scripts control the agent's attacks that they will carry out on the player. The Enemy Controller acts as a decision-maker with respect to deciding whether the agent should move directly to the enemies position or act in a flock-like behaviour. The enemy base class manages the base elements of the enemy such as its animations, the energy stacks that it has absorbed from the player’s attack and controls the enemies health, as well as whether it is taking damage or not.

In addition to this, every enemy agent has a Flock Agent attached to it. The Flock Agents comprise the Flock. The flock has a composite behaviour that is made up of cohesion, avoidance and alignment behaviours that control how agents and their neighbours move when they cannot see the player.

### 1.3.2 Algorithm Analysis

The most noteworthy algorithm used in Frost is the flocking algorithm that controls enemy movement when they are not moving directly towards the player. This is an implementation of the Boids artificial life program, which mimics the flocking behaviour of birds.

Each enemy has three vectors calculated every update, namely an avoidance, alignment and cohesion vectors that are then combined to create a vector that the enemy will move along. The avoidance vector is responsible for steering the agents away from obstacles and nearby agents. Obstacles and agents are weighted differently in order for the agent to place greater emphasis on avoiding obstaclesrelative to the need to avoid neighbouring agents. It is necessary to avoid neighbour agents to avoid collisions within the flock. The alignment vector is responsible for steering each agent towards the average heading of its flock mates. In other words, it is responsible for the flock agents moving in the same direction and facing the same direction. Lastly, the cohesion vector steers the agent towards the centre mass of the flock so that the flock tends to stay together. Once these three vectors have been calculated they create a composite behaviour that the agent will follow.

## 1.4 Implementation

### 1.4.1 Flocking/BOIDS

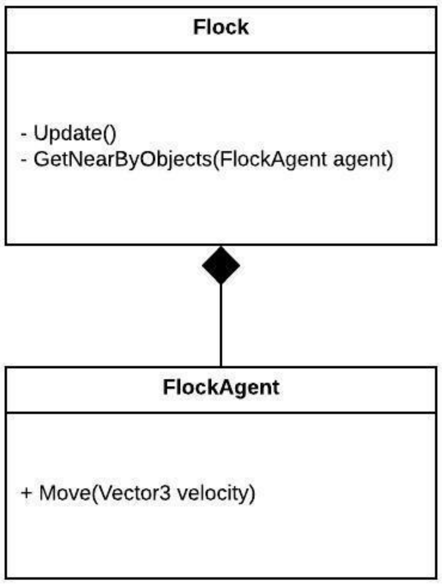


*Figure 3 Flock and FlockAgent Classes*

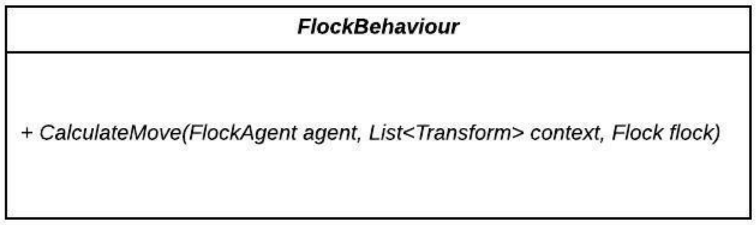

FlockBehaviour
+ CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)

*Figure 4 FlockBehaviour class*

The Flock, FlockBehaviour and FlockAgent classes are central to the flocking algorithm. A flock is made up of one or more enemies that all have a FlockAgent attached to them. The Flock class uses the GetNearbyObjects method in its Update method (Update is a built in unity method which is called every frame by the game engine) to get all agents around a particular agent. Once it has this list it uses the CalculateMove method from CompositeBehaviour which is a concrete implementation of the FlockBehaviour class to get the agents composite behaviour as a vector. This is then passed to the Move method called from the FlockAgent class. This moves the agent along the velocity vector.

### 1.4.2 PlayerMovement

The PlayerMovement class is responsible for handling all player movement input and calculations as well as updating the player's animation controller with the current velocity of the player. In order to maximise player agency and enjoyment, a Character Controller component is used to control the player character. All calculations are processed in the Update method to ensure accuracy on every frame. This means that the player is controlled directly by updating its transform rather than via the physics system using forces. This provides more fine-grained control allowing the player to move accurately, turning, accelerating and decelerating instantly.

In order to achieve this, the input is applied relative to the direction the camera is facing using the inverse transform direction and the player model is rotated smoothly using linear interpolation to face the direction of movement.

### 1.4.3 PlayerAttack

The PlayerAttack script is responsible for handling the input related to combat including keeping the animator state updated and ensuring different attacks are not able to be performed over each other or off cooldown unintentionally. Input is all detected in the update method to ensure no interactions are lost. The Shoot method is responsible for all processing related to the main attack ability. Additionally, this script is responsible for the creation of the main attack projectiles. This involves using raycasts to map from the screen view to a point in the world so that the projectiles always fire directly at whatever point in the world the player aims towards. In order to prevent the player from firing backwards into themselves, the relative orientation of the camera to the player is found by calculating the dot product of the camera and the players normalised horizontal forward vectors.

### 1.4.3 CameraController

The CameraController script is responsible for handling the input related to camera movement as well as ensuring the camera dynamically adjusts to the geometry of the level in the players' vicinity whilst still providing the player the ability to control the zoom amount or distance of the camera from the player. All calculations are processed in the Update method to ensure accuracy on every frame. The camera continuously checks to ensure that it is not clipping inside the level geometry nor that any geometry obscures the player from view. This is ensured by dynamically bringing the camera in closer to the player when needed and returning to the original position when level geometry allows.

In order to achieve this, the camera casts a ray between its near clipping plane and the orbit target and, if this ray intersects with the level geometry, the camera moves itself in closer to be just in front of the occluding object. Finally if the camera is not at the desired distance from the player, a raycast is performed between the camera and a point a short distance directly behind it. Once again this ray checks for any intersecting geometry. In the case that the raycast intersects an object, the camera will again move itself to ensure it does not clip inside the level. However if no intersect is found with the "backwards ray", the camera will smoothly interpolate back to the desired distance from the player.

### 1.4.4 Special Class Relationships

As seen in *Figure* 2 FlockBehaviour is an abstract class that is implemented by AlignmentBehaviour, AvoidanceBehaviour, CohesionBehaviour and CompositeBehaviour as they all need to implement the same CalculateMove method in order to produce a vector that will eventually comprise the final movement vector.

## 1.5 Program Validation and Verification

### 1.5.1 Testing

The majority of tests carried out were user central, in other words, they were “play tests” as almost all the tests required user input of some sort. All tests throughout the production phase were carried out by the programmers and once the game was complete, we gave people unfamiliar to the game the chance to play it in order to see if the experience was intuitive and easily carried out. All tests were monitored by logging output for each action that was carried out during the test case.

*Table 1: Summary of tests carried out*

| Test Case | Input | Expected Output |
|---|---|---|
| Player Damages Enemy | Player presses attack key | Enemy Health Reduced by the correct amount when the player is within the correct range |
| Enemy Death | Player attacks enemy | If the enemy’s health is less than or equal to zero, then a death animation should be played |
| Player Damaged | Player moves towards the enemy | Enemy attacks the player if they are in the correct range and the player health is reduced by the correct amount |
| Player Death | Player moves towards the enemy | Enemy attacks the player if they are in the correct range and once the player’s health is zero or below the player is sent to the end game scene |
| Player Death (multiplayer) | Players move towards the enemy | Enemy attacks the player if they are in the correct range and once the players’ health is zero or below the player is sent to the end game scene |
| Player Respawn (multiplayer) | Player dies | Player respawns once the next level starts. |
| Enemy Moves to Player When the Player is in Line of Sight | Move player into view of the enemy | The enemy should move towards the player |
| Enemies Display Flocking Behaviour | Make sure the player is not within view of the enemy | The enemies display flocking behaviour and move randomly in flocks |
| Player Unlocks Unlockable Area | Player attacks enemies and gains enough frost essence to open an area | Player opens area and their frost essence amount decreases by the correct factor |
| Player Can Use Abilities by Using Frost Essence | Player attacks enemies and gains enough frost essence to use abilities | Player can use their special ability and their frost essence is reduced by the correct factor |
| User Test* | User Play Test | The user will be able to navigate and play the game intuitively without support |

| | | |
|---|---|---|
| | | from one of the group members. |

*Notes on user testing:

- The play tests were run with three separate people.
- They consisted of letting the user play the game without guidance and provide feedback as they did so.
- The most important aspect of the feedback was that there was not enough information about the game given to the user. This lead to us adding a how to play section to the main menu and adding more details to our HUD.
- In general the game play loop was well received and players found it an intuitive experience.

## 1.6 Conclusion

The main aim of our project was creating a game that you can jump into and play without any guidance and without the hinderance of any complex story line. It is evident that this has been done as our user testing proved the gameplay came intuitively to players. Additionally it was important to stick to our core use cases and meet the scope we outlined in our Game Design Document. The majority of the scope was achievable. The ability to upgrade your player, save your game and have multiple different characters were all features we could not add in, however, the core features of the game are still present, with all our test cases passing.

## 1.7 User Manual

See Appendix A.

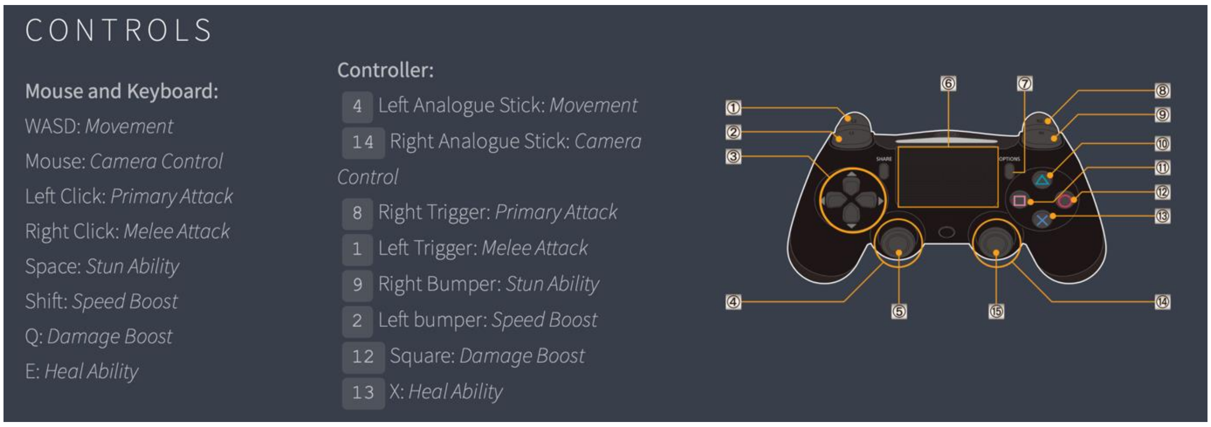
## Appendix A — User Manual

How to play:

- Survive as many waves of ruthless enemies as you can while trying to achieve your highest score!
- As you progress through the rounds different areas of the map will become unlockable, which will be indicated on your HUD.
- Explore these areas and find mythical healing stones to help you survive, but watch out for the traps!
- As you kill enemies you will gain Frost Essence, a magical substance that will allow you to defy your perceptions of reality and grant you unbelievable powers. The red Frost Essence allows you to deal more damage, the green Frost Essence allows you to heal yourself and the blue Frost Essence allows you to boost your speed. Be weary of how you use it because it runs out quickly!
- Feel like sharing the experience with a friend? Why not jump into a two-player game and try outscore your friend? It's as simple as pressing the multiplayer option. Make sure at least one of you has a controller for the optimal experience.
- Enjoy your adventures through the Frostlands!

Starting the game and initiaiting the game play:

- Double click on the executable and the game will start.
- Click the "How to Play" button to inform yourself of the game rules and general idea of the game.
- Once you are aquainted with the game return to the main menu and click play. From then onwards the enemies will spawn and the main game loop will have began.
- To exit the game or pause, prese 'escape' key and click 'exit' in order to return to the main menu.



CONTROLS

**Mouse and Keyboard:**

WASD: *Movement*
Mouse: *Camera Control*
Left Click: *Primary Attack*
Right Click: *Melee Attack*
Space: *Stun Ability*
Shift: *Speed Boost*
Q: *Damage Boost*
E: *Heal Ability*

**Controller:**

4 Left Analogue Stick: *Movement*
14 Right Analogue Stick: *Camera Control*
8 Right Trigger: *Primary Attack*
1 Left Trigger: *Melee Attack*
9 Right Bumper: *Stun Ability*
2 Left bumper: *Speed Boost*
12 Square: *Damage Boost*
13 X: *Heal Ability*