HELLENIC MEDITERRANEAN UNIVERSITY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

COURSE TYPE - INFORMATICS ENGINEERING T.E.

THESIS

TWO DIMENSIONAL SURVIVAL GAME IN UNITY

ELENI KOUNDOURAKI AM 4056

ADVISOR

IOANNIS PACHOULAKIS

OCTOBER 2020

THANKING SECTION

I would like to thank my supervisor teacher Mr. Ioanni Pachoulaki for supporting and accepting my idea and ultimately giving me the chance to implement, this 2D video game. I would also like to thank all my family for supporting me and giving me courage through the journey of this major project.

Thank you.

ΠΕΡΙΛΗΨΗ

Αυτή η διατριβή αφορά τη δημιουργία ενός δισδιάστατου παιχνιδιού επιβίωσης για υπολογιστές το οποίο αναπτύχθηκε χρησιμοποιώντας το Unity Game Engine και σχεδιάστηκε χρησιμοποιώντας τα προγράμματα Aseprite για τα μοντέλα χαρακτήρων και Tiled για τη διαμόρφωση των χαρτών.

Σε αυτό το δισδιάστατο παιχνίδι επιβίωσης "Bow and Jelly", έχω εφαρμόσει 2D φόντα, χάρτες και χαρακτήρες με ξεχωριστά και ποικίλα σχέδια και animations. Το παιχνίδι είναι εμπλουτισμένο με αλγόριθμους εντοπισμού μονοπατιών για την προσομοίωση έξυπνων συμπεριφορών κυνηγιού και αλγορίθμων πιθανότητας για να παρέχει στον παίκτη χρήσιμα λάφυρα αντικειμένων κατά την πλοήγηση στις διάφορες περιοχές του χάρτη. Η διεπαφή χρήστη έχει έναν ελκυστικό σχεδιασμό που τροφοδοτείται από καθηλωτικούς ήχους κινούμενων σχεδίων και μουσική υπόκρουση, που επιτρέπει μια ευχάριστη εμπειρία.

Οι λεπτομέρειες της δημιουργίας σχεδιασμού και των συμπεριφορών εντός του παιχνιδιού εξηγούνται παρακάτω, συνοδευόμενες από γραφήματα με βάση το σενάριο και στιγμιότυπα οθόνης του παιχνιδιού.

ABSTRACT

This thesis is about the creation of a 2Dimensional Survival Game for computers that was developed using Unity Game Engine and designed using the programs Aseprite for the character models and Tiled for the forming of the maps.

In this 2D Survival Game “Bow and Jelly”, I have implemented 2D backgrounds, maps, and characters with distinct and varied designs and animations. The game AI uses pathfinding algorithms to simulate smart chasing behaviors and probability algorithms to provide the player with helpful item loot drops while navigating through the various map areas. The UI has an inviting design powered by immersive animation sounds and background music, which provides an enjoyable experience.

The details of the design creation and in-game behaviors are explained below, accompanied by scenario-based graphs and screenshots of game-play.

Table of Contents

Table of Figures

1.Introduction

1.1 Summary of the Game

The 2D Survival Game “Bow and Jelly” is a shoot ‘em up type of game that can be played on the computer. It has elements of exploration and horror with randomly generated enemies that keep the game engaging for numerous replays.

The main character is a nameless traveler who wields a bow and rids the peaceful town of various monsters, slimes, and zombies that roam around the various maps. It has a pixel art aesthetic and various map areas in which the main character can traverse through interconnected paths. The enemies in the game, are designed to drop items that both increase and decrease the health of the hero, thus the player needs to be careful to not pick up every item blindly. The items have intuitive designs that can be distinguished easily on the computer screen, for example, the red apples heal and grubs hurt. The game also has interactive enemy waves that keep the gameplay interesting along with being chaotic.

1.2 Incentive of Making this Project

The incentive of this project derives from my long appreciation of 2D game development. I was always intrigued by these kinds of games that offered me a more immersive experience, with the colorful graphical environments with games like “Star-dew Valley”. The fascination continued with the exceptional stories that rivaled big triple-A game companies, with games like “Undertale”. These games made me want to learn more about them and how they are made, hence the subject of this Thesis.

1.3 Objective and Purpose of the Project

The purpose of this project is the development of a two-dimensional video game primarily coded in C# and using the very reliable game engine Unity. The project is a complete game built from scratch with many areas to explore. This game should be familiar to those aware of shoot ‘em up games with modern examples like “Hotline Miami” and “Binding of Isaac” but more importantly like “Contra” and “Metal Slug”. The objective of the game is to struggle for a high score much like the classic game “Pacman” and the hit from a few years back “Flappy Bird”. The player goes through waves and waves of randomly spawning enemies and tries to survive for as long as they can with occasional heath drops by enemies. The game has simple controls to get used to but is a good challenge on purpose, as research suggests more difficult but fair games are highly addicting to individuals.

2. Technologies and Concepts that Contributed in the Project

2.1 What is Unity

Unity is a cross-platform game engine developed by Unity Technologies, that supports more than 25 platforms.[18] Unity provides many built-in tools that can help novice and experienced developers alike to make games. Some of those tools are Physics engines, rendering, collision detection light rendering, and more for both 3D and 2D environments. The engine can be used to develop all kinds of games virtual reality, augmented reality games, as well as simulations and educational apps. Unity also provides a vast variety of assets, like graphics, sound effects, and fonts, through its "Asset Store", that the unity community updates almost every day.

2.2 What is Game Development

Video games have come a long way from a few pixels on-screen in the first proper video game developed 'pong' to virtual reality simulations which are becoming harder to distinguish from the actual world with hundreds of hours of content. Gaming has become so mainstream that the collective revenue of digital games, dwarfs the revenue of the movie industry. More than entertainment, it has led to revolutions in technology which would have been a pipe dream if not for the great demand that video games have. One such example is the advancement in the GPU, which is not just limited to video game applications but also data mining and training deep neural networks.

Game development as the name suggests is the process of constructing a video game, that can have one or a team of developers working on it. With the growing open-source resources available and established video games companies making their game development tool-kits available; It has led to the development of countless games on every platform, especially with the indie game revolution which gives high-budget gaming companies something to be concerned about.

2.3 What is Graphic Development and Pixel Art

Graphic development is the process of creating all the visual content and effects that are gonna be used for the development of a video game and more. The graphic designs may vary depending on the game's needs, meaning that the designs can be as simple as creating a small box, to more complicated designs that depict reality that is so realistic that trick the users senses into thinking that they are in the real world.

Pixel art is the process of creating two-dimensional artwork. It is also used as the graphic development method for two-dimensional gaming environments and characters. Unlike three dimensional graphics, pixel art is created pixel by pixel placed methodically to create a bigger image. Despite its minimalism pixel art, graphical environments can be immersive and alluring, which can captivate the player's senses better than the three-dimensional environments.

2.4 What is Aseprite and Tiled

Aseprite is a pixel art tool, that lets you create 2D animations and art. It is widely used for video game graphic development. In this project, it was used to create the player and enemy character designs and

animations, as well as the loading screen that is displayed when the hero changes scenes, the game's cursor icon, and various other sprite sheets.

Tiled is a 2D level editor, like Aseprite it helps you develop the content of your game. Unlike character and effect creation, Tiled is used for bigger tasks like the creation of tile-maps and tile-map animations. For this project, Tiled was used for the creation of all the maps and menus as it is referred to in later chapters.

2.5 What is State Machine

State Machine are the different movement behaviours of the different characters in a game. At any point in time, the characters in the game are always engaged in an action that is referred to as a state. The state machine can consist either of blend trees that can be nested and provide a more complicated state machine along with their blending parameters and simple states with simple transitions that allow smooth animation movements.

2.6 What is Pathfinding

Pathfinding algorithms will be familiar to those who have studied subjects like graph theory in Math, Dijkstra, and Bellman-Ford algorithms. These algorithms try to find the shortest path to the target but within limits of certain domain logic, for example- the enemy should not just jump over obstacles to get to the player.

2.7 What is UI

UI design is concerned with the aesthetics of the application, as it is the first introduction to any platform, it should convey the emotions the designer is trying to invoke. This project's game UI is non-diegetic meaning that is rendered outside the game world and is only visible or audible. It consists of player statistic bars that can be viewed at all times, damage effects for both the player and the enemies, and dialogue bubbles that can be viewed when the player interacts with the non-playable characters as is referred to in later chapters.

3. Features and Workflow of the Project

3.1 Analysis Models of the Game Menus

3.1.1 Scenario Based Models

The game project is based on a scenario based model, using a use case system. The following diagrams depict how the user can interact with the interfaces and the sequences that take place as he navigates through the software.

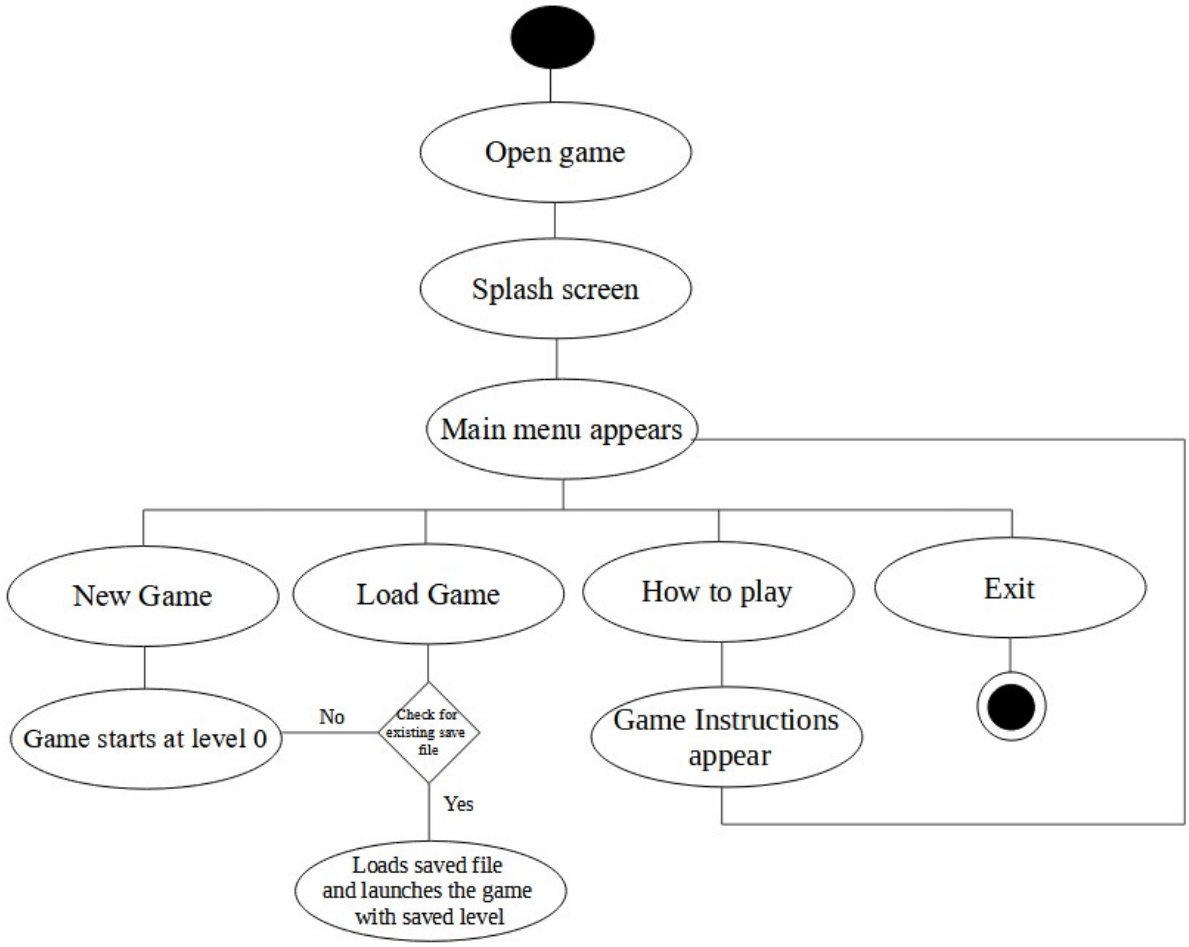

Figure 1: Start the Game - Sate Diagram

We can see the behavior of the software when the user first opens up the game application. When he presses the desktop icon, the game sequence starts by first showing the user the unity splash screen which is the unity watermark and loads up the game's main menu. When the menu is open the model goes into a waiting state for the user's input. The user has four choices to decide from.

The first choices that can be made are New Game or Load Game option which initialize with the following activity diagrams respectively.

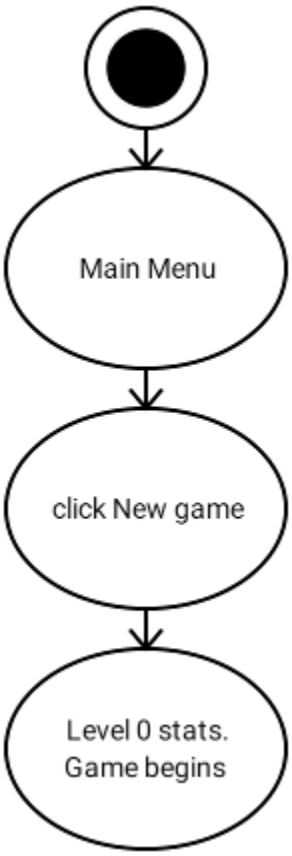

Figure 5: Activity Diagram for "New Game" feature

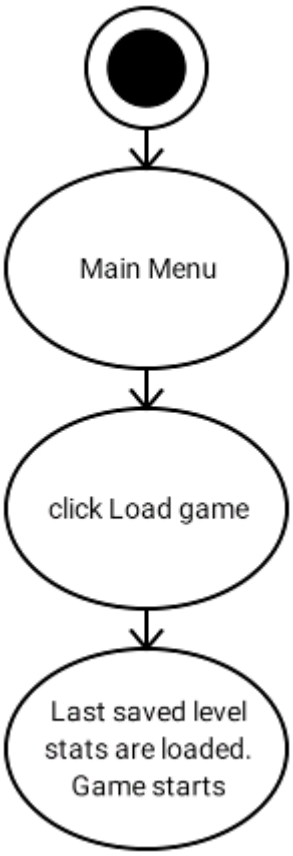

Figure 3: Activity Diagram for "Load Game" feature

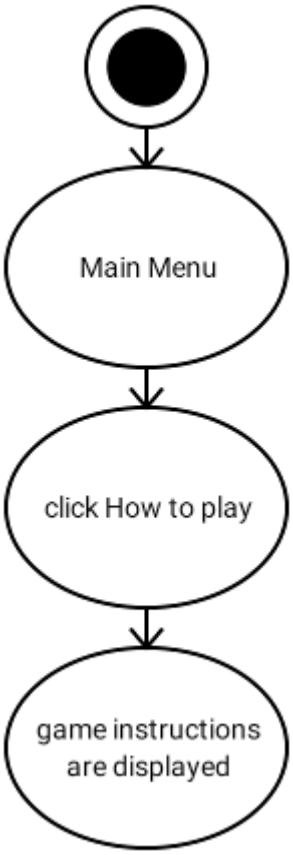

Figure 4: Activity Diagram for "How To Play" feature

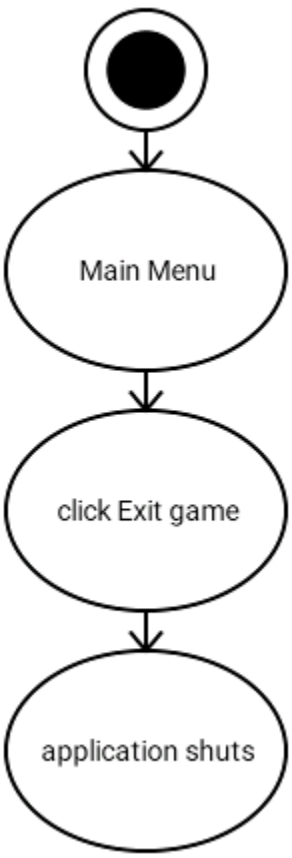

Figure 2: Activity Diagram for "Exit Game" feature

When the New Game button is clicked the system initializes the player stats at Level 0 and starts the game from the main scene. On the other hand, if the Load New Game option is selected then the system looks for any potential load files that have been made. If the files exist then the game loads with the characters saved level, else the game starts at level 0.

The last two choices the user has at the main menu are the How To Play button and the Exit Game button.

The How To Play option provides the user with the game controls so he will familiarize himself with how he can interact when the game starts and then he can return to the main menu as the main menu diagram depicts to select a different action. The Exit Game option as it suggests when selected shuts the game application.

An additional scenario-based model can be found during the gameplay of the “Bow and Jelly” game, and those are the pause and restart menus. Their activity diagrams are the following.

Starting with the Pause Menu Activity Diagrams:

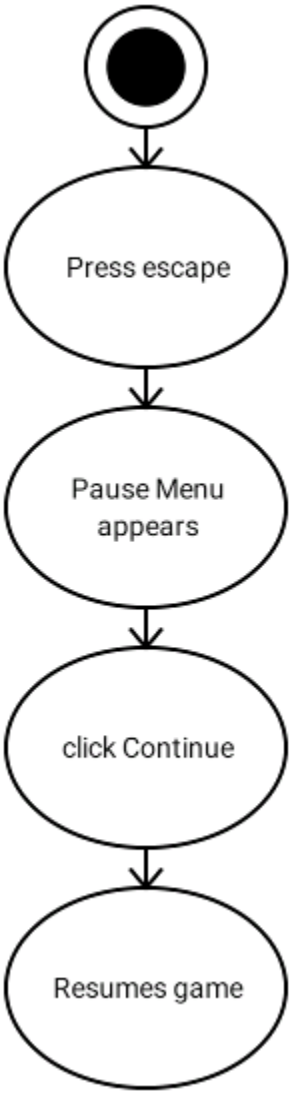

Figure 6: Activity Diagram of "Continue" module of Pause Menu

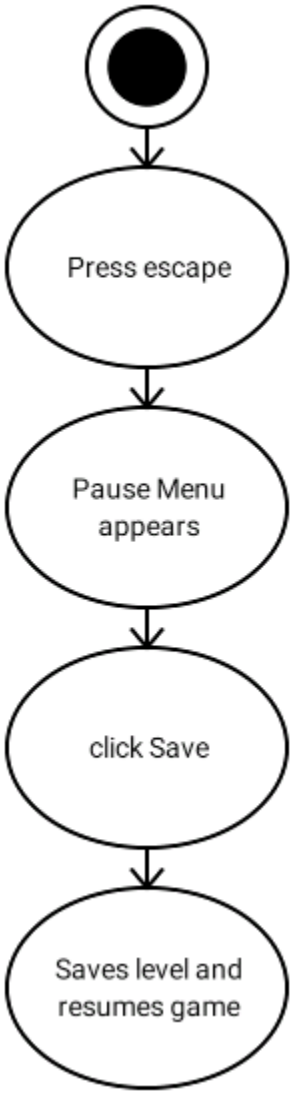

Figure 7: Activity Diagram of "Save" module of Pause Menu

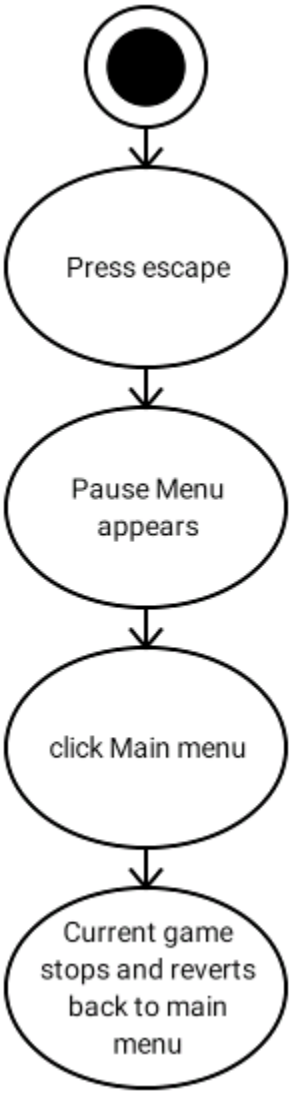

Figure 8: Activity Diagram of "Main Menu" module of Pause Menu

During the game-play when the escape key button is pressed the game stops momentarily and the Pause Menu appears offering three choices. The First choice is simply to Continue the game as shown in the first diagram. The second choice is to save the current level of the player so when it is pressed the game saves the level of the player in a binary file and resumes the game, as shown in the second diagram. Lastly, the third option of the pause menu is the Main Menu button, which redirects the user back to the main menu.

And Lastly the Restart Menu Activity Diagrams:

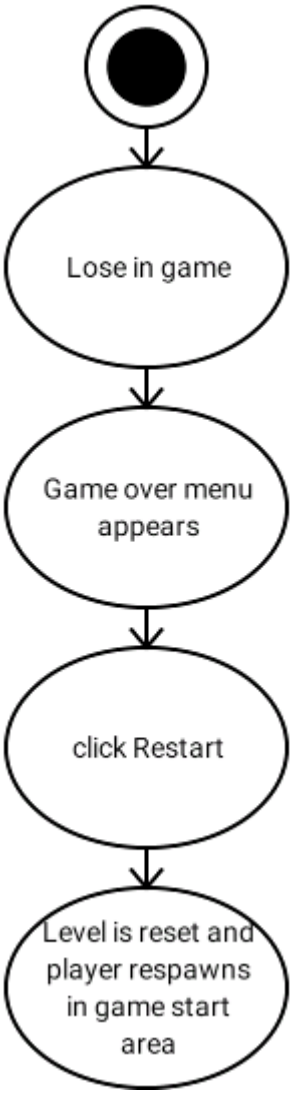

Figure 9: Activity Diagram of "Restart" module of Restart Menu

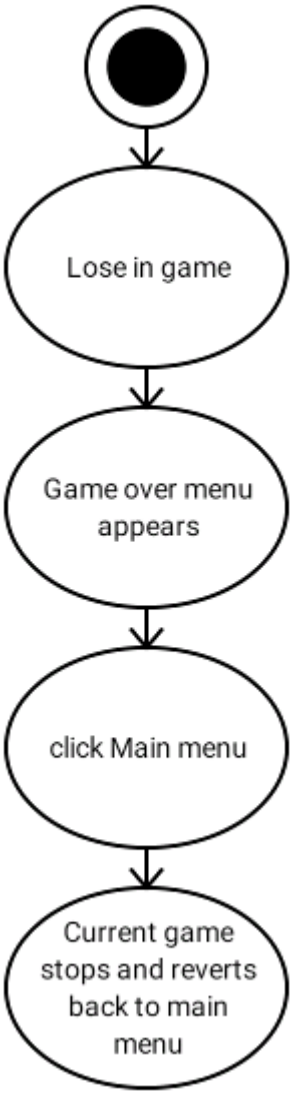

Figure 10: Activity Diagram of "Main Menu" module of Restart Menu

During game-play when the hero's health goes to zero the game stops and a Restart Menu appears on the screen giving the player two choices. One option as shown in the first diagram is to Restart the game, when pressed the hero's Level restarts, meaning it goes to zero, and the player re-spawns. The second option given is the Main Menu button which reverts the game to the main menu as well as making the player's level to zero.

3.1.2 Menu Designs

The menu designs were developed with the program Tiled from the same sprite sheets that the maps were created. The Pausing menu along with the Restart menu and the menu buttons were not difficult to design since they only required only one or two layers in the Tiled Editor to get created. The main menu, on the other hand, was the most complicated to create since it required a more complicated

design for the user to get a first feeling of the kind of graphics he is going to encounter when he first launches the game. So to achieve that, the main menu needed more layers, for the correct sorting of the items to prevent any overlaps.

The functionality of the menus was given with the Unity Editor. By creating three canvases that contained Image fields for the Tiled files to be inserted, text fields for all the titles for the menus and the button names, as well as button fields that allow the desired actions, of course, powered by code that is gonna get mentioned in the next chapter.

In the following figures, you will see the Canvas system in Unity and also the backgrounds that were created in Tiled that were later used for the menus in the game, as well as, how they appear in the game.

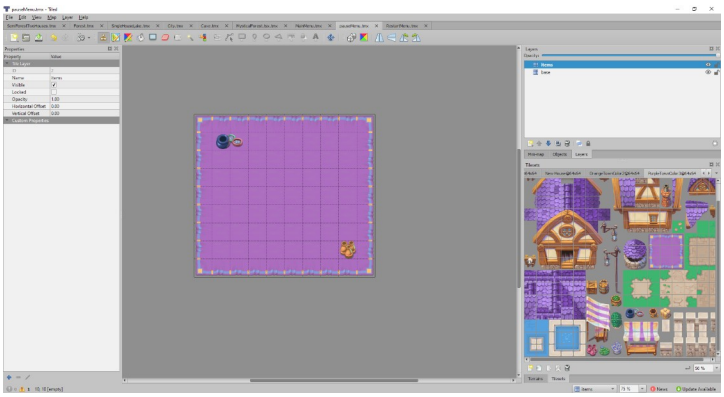

Figure 11: Background Image creation for Pause Menu in Tiled

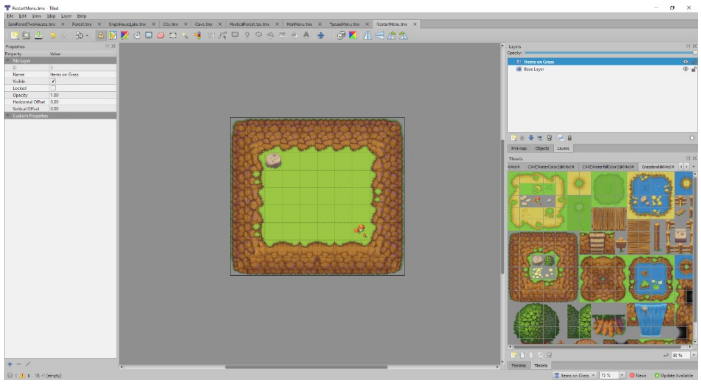

Figure 12: Background Image Creation for Restart Menu in Tiled

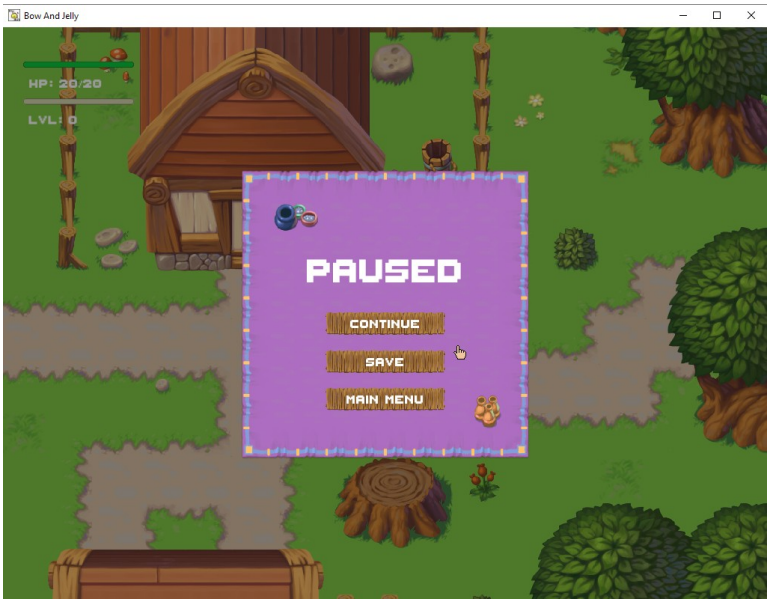

Figure 13: Pause Menu

Figure 14: Restart Menu

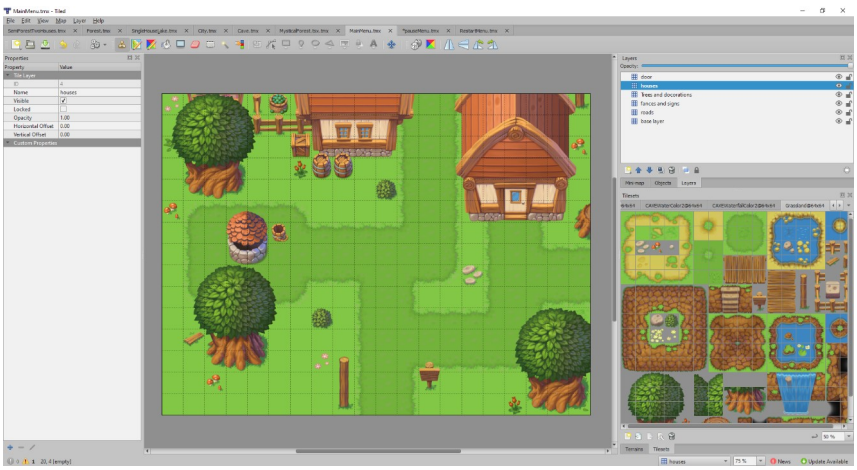

Figure 15: Background Image Creation for Main Menu in Tiled

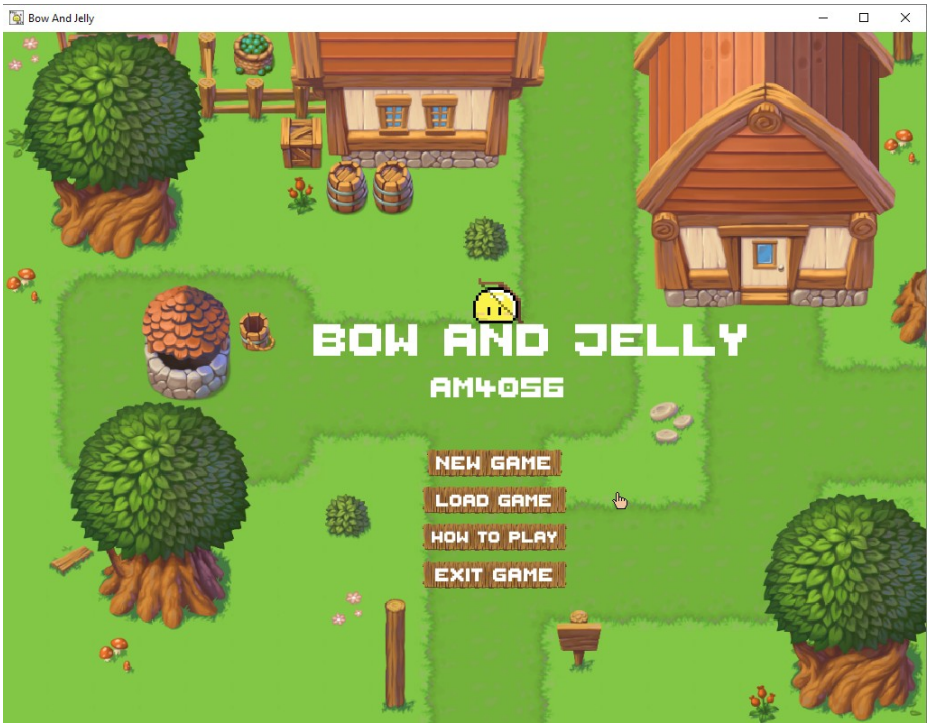

Figure 16: Main Menu

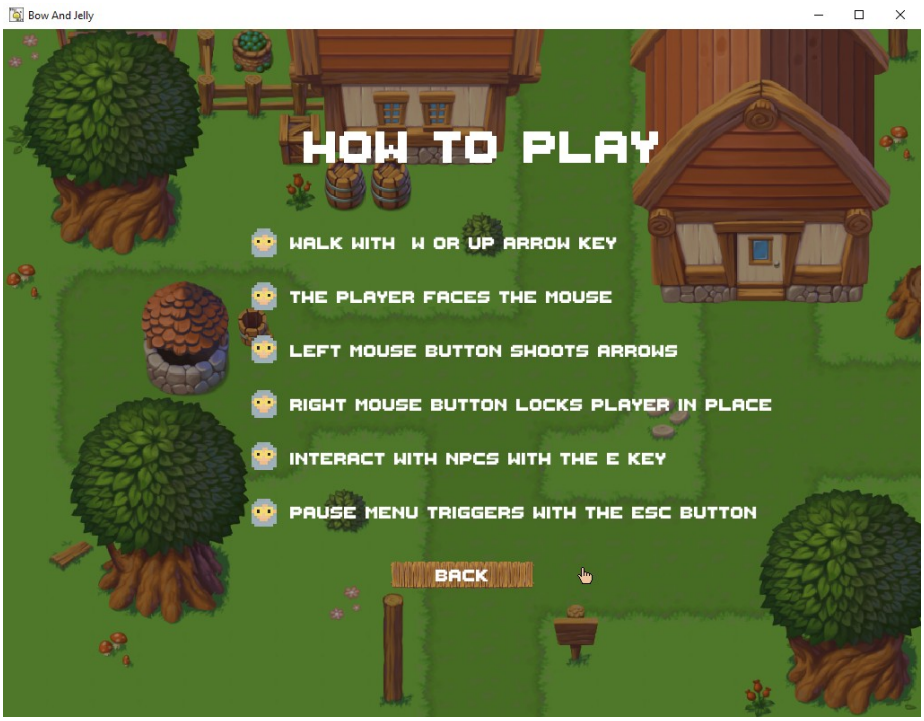

Figure 17: How To Play Menu

3.2 Map Designs

Unlike the character creations, I purchased the tile-set assets for the design of the maps for the game from Unity's Asset store. For the map creation similar to the menu designs the program Tiled was used. I used this program because of its flexibility and how easy it cooperates with Unity. If a change on any part of the map designs is desired, you can easily redo through the Tiled program and it will automatically refresh in Unity without having to import the new files. Also Tiled has its collision system that unity supports, so it makes the designing a step easier. For this project, six maps were created. They provide an immersive continuity so that the player won't be confused with where he is.

The main scene is where the player gets spawned when the game starts and where random Slime enemies appear. This map is connected on the right side with the Forest map and on the left side with the Town map. The Forest map provides the player the option to activate enemy waves along with a bigger variety of random Slime Enemies. This map is connected with three others. The main one that the player can go back to if he desires, a Cave map, that can be accessed through the mine door on the wall and the Lake map that can be accessed from the north path. The Cave is a mini-map compared to the others. It provides a high-level Slime enemy with valuable loot drop items and a gateway that the player can go back to the forest. The Lake map, like the forest one, gives the option to the player to activate waves of new enemies along with new random spawning ones. And has access back to the forest the Town and Mystic Forest maps. The mystic Forest like the previous maps has the previous and more enemies for the player both for the random spawner and the wave spawner The town gives the player a place to breathe as it has no enemies, and the stalls in the town provide a variety of health items that the hero can pick up and restore his health.

Figure 18: Cave Map

Figure 19: Main Map

Figure 20: Forest Map

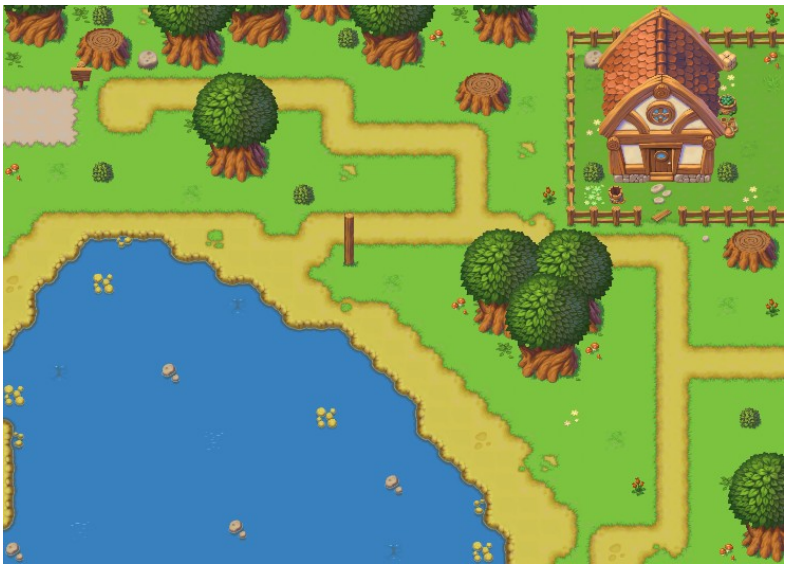

Figure 21: Lake Map

Figure 22: Mystic Forest Map

Figure 23: Town Map

3.3 Player

3.3.1 Player State Machine

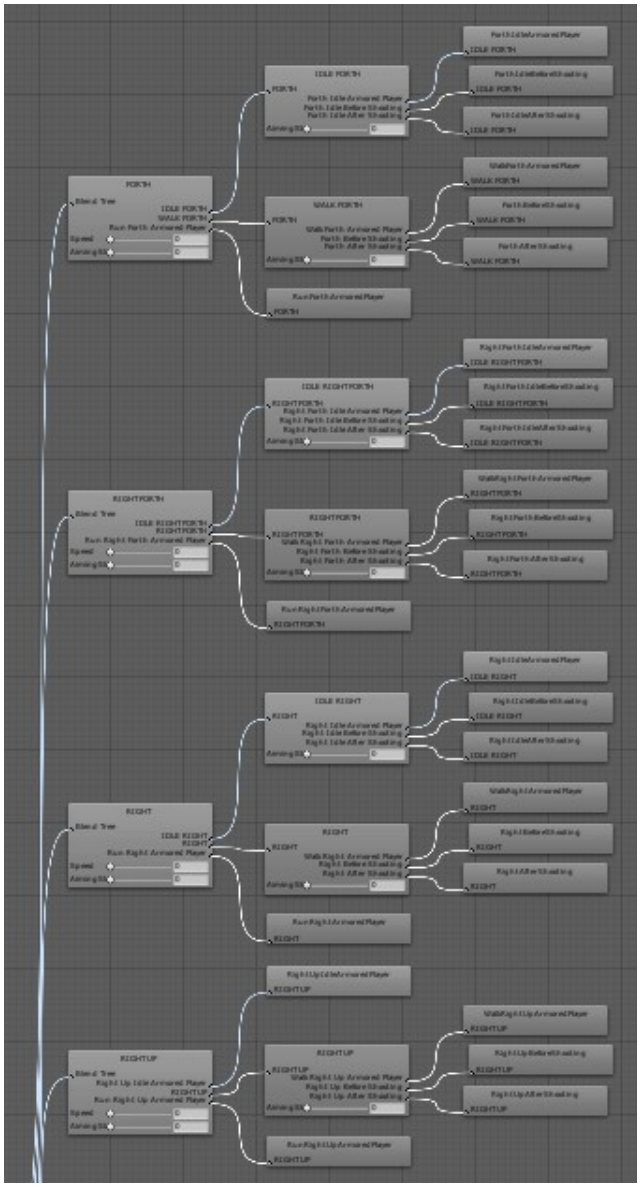

Figure 25: Player State Machine 1st part

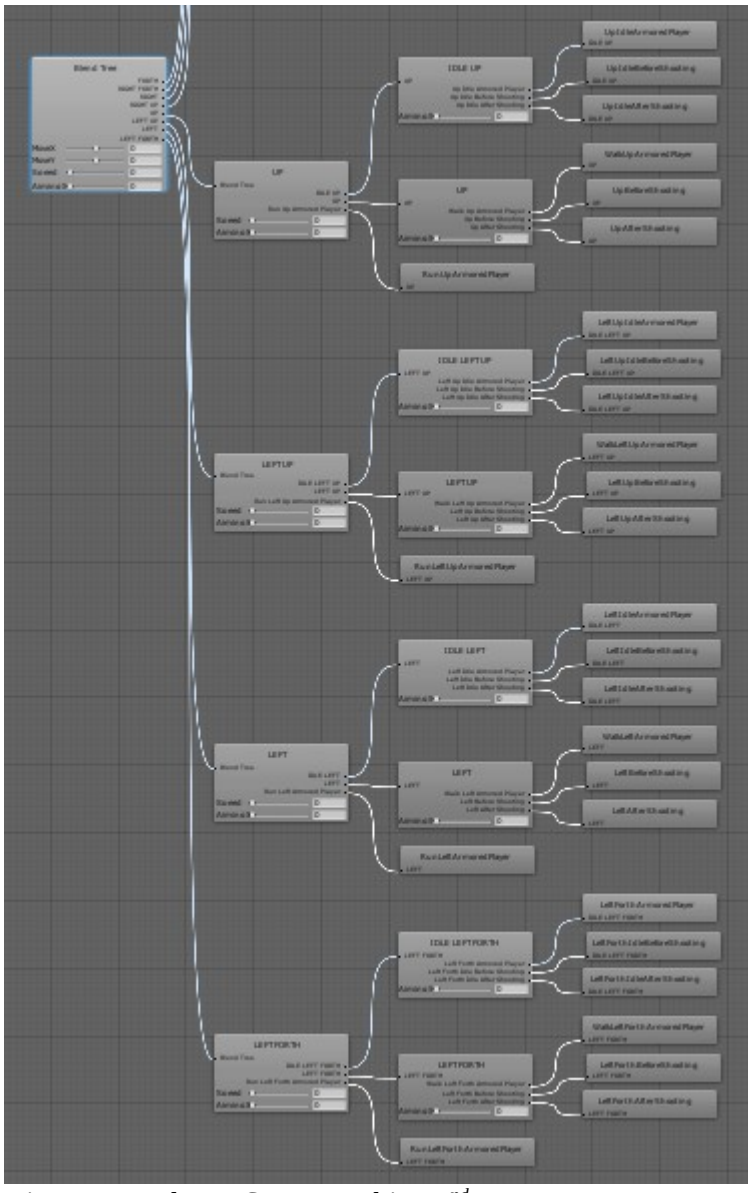

Figure 24: Player State Machine 2nd part

The state machine for the Player is complicated, as it consists of multiple nested blend trees so he would have smooth animation transitions. The player supports eight directions and five types of animation behaviours. Which are Idle, Walking, Running, Walking with his bow loaded, and his bow released. These animations are doable through the blending parameters that are set for the blend trees and coding.

3.3.2 Player UI

The User has two major elements to pay attention to in regards to their character. First, as with many games, we have the HP or Health points which at level 0 is 20 HP, and logarithmically grow according to each level gained by the player. The player HP gets reduced when enemy characters attack the player or you consume unhealthy foods like a rotten apple. Respectively the player heals or regains their HP when they consume healthy food items like roasted chicken.

The player level is another bar that appears below the player HP, which grows only when you defeat enemies and gain experience points, as the player progresses through the levels their HP and attack damage increases. The player starts at level 1 and grows till level 50 which is the maximum achievable level in-game.

3.4 Enemies

Enemies in “Bow and jelly” are mainly divided into two categories. The seemingly harmless and abundant slimes or jellies and the challenging zombies.

Slimes can be found out throughout the map and keep generating randomly, the production rate is kept slow for two main reasons; the player does not get overwhelmed with numbers as soon as they start the game or enter a new area, and second so that the scene won’t get overcrowded and slow down the game's performance.

Slimes come in different colors that indicate their different difficulties and can be noticed jumping around the map, they are slow and not interested in the player unless the player gets close to them. There is also a hidden variant of the slime which acts as a boss enemy and provides a tougher challenge for the player, the mega black slime, it takes a good amount of damage to go down. Logically then it also drops the best healing items in the game with a much higher rate.

Zombies are a tougher challenge and provide more experience in turn compared to the slimes. They are not found anywhere on the map by default and have to be activated by talking to certain Non-playable characters. This was a deliberate choice as they change the game into more combat focussed and reduce the desire to explore the maps. Thus a player can choose to call upon the zombies for a challenge at their own accord and also stop them from coming by talking to the Non-playable characters again. Zombies always come to attack in waves, after beating one wave with no zombies remaining can the next wave appear. These waves bring zombies of four different kinds, each varied in design and each with different health points, attack damage, and most importantly speed. After waves of each type of zombies, their waves repeat but with growing numbers.

Figure 26: Enemies

3.4.1 Enemies State Machine

Because there are different kinds of enemies in the game they required two different kinds of State Machines

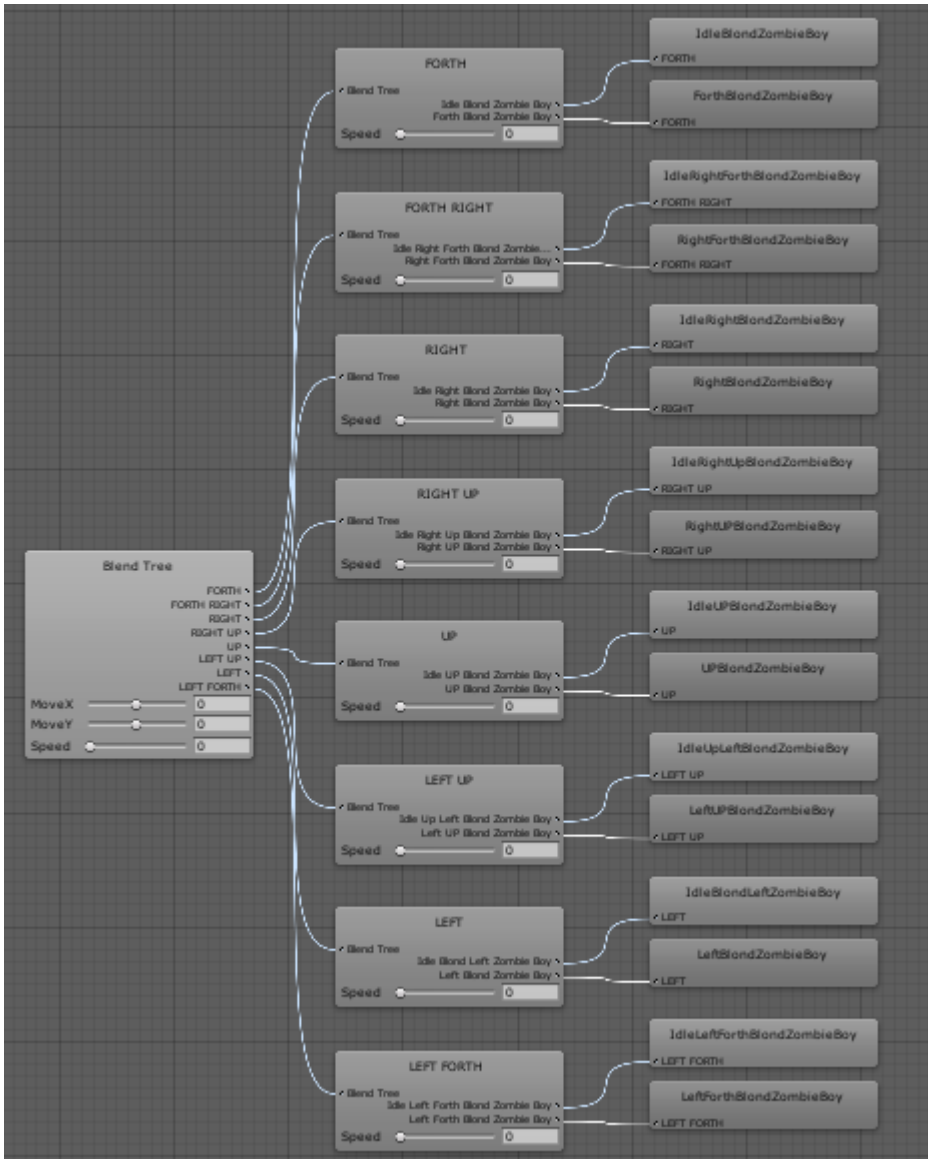

Figure 27: State Machine for Zombie Type of Enemies

The Zombie type of enemies required a state machine similar to the players with nested blending trees, also supporting eight directions, but not as complicated, as they only have two states Idle and Walking or Running depending on the zombie type.

The Slime Enemies on the other hand only needed a simple State Machine with no blend trees and just transitions; Since their animation and movement are so simple, only having two states Moving and not Moving

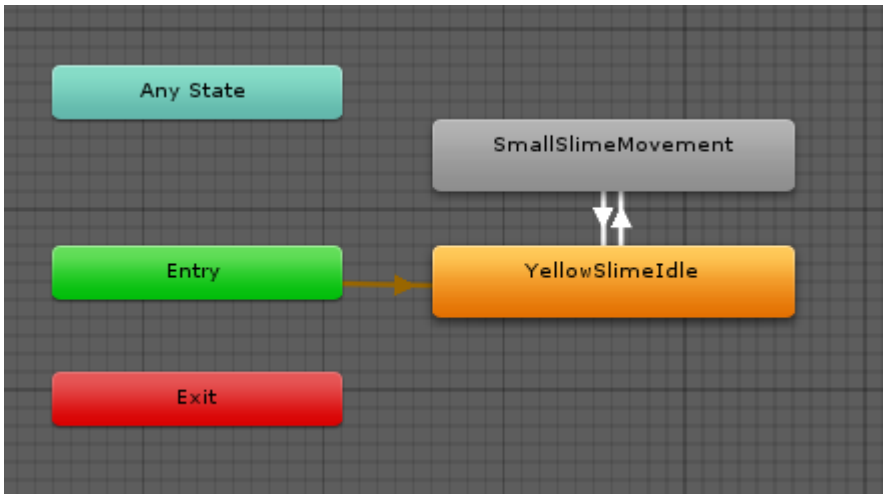

Figure 28: State Machine for Slime type Enemies

3.5 Non Playable Characters

There are four Non-playable characters(NPCs) in the game. Each found in a different location of the map.

Dr. Augustus is the lab coat wearing gentleman you can find in the town's square, he tells the player their statistics in the game, such as their level, attack damage, enemies killed, and experience needed to level up.

Mr. Dave is a lumberjack who lives deep in the woods, who wears the lumberjack dungaree. The player can interact with him and also activate the zombies in the woods area.

Hearsay the horse, is the horse on the far right corner on the map who lives on a cliff, players can interact with her and activate the zombies in the cliff-side area.

Ballon girl as the name suggests is a little girl holding a balloon, she can be found just outside the mysterious cave, the player can also interact with her and activate zombies in that area.

Figure 29: NPC Characters

4. Code Implementation

4.1 Camera Movement

In order for the camera to follow the player but still be inside the bounds of the maps we set the minimum and maximum bounds of the maps 2D collider.

```
1 reference
public void SetBounds()
{
    if(SceneManager.GetActiveScene().name== "Cave")
    {
        minBounds = caveBoundss.bounds.min;                    //we will get the lowest x and y of the collider
        maxBounds = caveBoundss.bounds.max;
    }
    else if (SceneManager.GetActiveScene().name == "MysticForest")
    {
        minBounds = mysticBounds.bounds.min;                   //we will get the lowest x and y of the collider
        maxBounds = mysticBounds.bounds.max;
    }
    else
    {
        minBounds = boundBox.bounds.min;                       //we will get the lowest x and y of the collider
        maxBounds = boundBox.bounds.max;
    }
}
```

Figure 30: Camera Movement 1st part

Then with the use of the default Unity functions; We take the position of the player and then set the camera to follow him. Then we calculate what position the camera should have based on the map's bounds and we update its position again. This process happens once at the before the first frame of the game starts, and then repeats continuously every frame for the camera to always follow the player.

```
void Start()
{
    SetBounds();

    halfHeight = theCamera.orthographicSize;
    halfWidth = theCamera.orthographicSize * Screen.width / Screen.height;

    targetPos = new Vector3(followTarget.transform.position.x, followTarget.transform.position.y, transform.position.z);
    transform.position = Vector3.Lerp(transform.position, targetPos, moveSpeed * Time.deltaTime);

    float clampedX = Mathf.Clamp(transform.position.x, minBounds.x + halfWidth, maxBounds.x - halfWidth);
    float clampedY = Mathf.Clamp(transform.position.y, minBounds.y + halfHeight, maxBounds.y - halfHeight);
    transform.position = new Vector3(clampedX, clampedY, transform.position.z);
}
```

Figure 31: Camera Movement 2nd part

4.2 Player

4.2.1 Player Movement

For the player's movement, we get the mouse coordinates on the screen and we set the player's movement direction to it. Next, we set the player to able to move only the positive side of the Y-axis to avoid errors like walking backward.

```
Vector2 mouseMovement = new Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse Y"));
movementDirection += mouseMovement;
movementDirection.Normalize();

if (Input.GetAxis("Vertical") > 0)
{
    movementSpeed = Input.GetAxis("Vertical");                    //Making the player move only when the y is positive
}
else
{
    movementSpeed = 0.0f;                                         //Else he wont move at all
}
endOfAiming = Input.GetButtonUp("Fire1");                         //It will become true when we release the button
isAiming = Input.GetButton("Fire1");                              //It will become true when the button is pressed ( LEFT CLICK )
lockPosition = Input.GetMouseButton(1);
```

Figure 32: Player Movement 1st part

As the direction and speed are initialized we set the player to walk by calling their RigidBody2D and then we animate him according to the blending parameters we have set in the player's State Machine, so the player will know when to walk, run, aim and release his bow.

```
//-----------------------------------------------------MOVE SETTINGS-----------------------------------------------------//

1 reference
private void Move()
{
    rb.velocity = movementDirection * movementSpeed * MovementBaseSpeed;
}

//---------------------------------------------------ANIMATOR SETTINGS---------------------------------------------------//

//Sending info to the animator
1 reference
private void Animate()
{
    if (movementDirection != Vector2.zero)                        //Only sets the parameters when the movementDirection !=0 so he'll face where he stopped
    {
        anim.SetFloat("MoveX", movementDirection.x);
        anim.SetFloat("MoveY", movementDirection.y);
    }
    anim.SetFloat("Speed", movementSpeed);

    if (isAiming && firedArrow == false)
    {
        anim.SetFloat("AimingState", 0.5f);
    }
    else if (shootingStart > 0.0f && firedArrow == false)
    {
        anim.SetFloat("AimingState", 1.0f);
    }
    else
    {
        anim.SetFloat("AimingState", 0.0f);
    }
}
```

Figure 33: Player Movement 2nd part

Lastly, the conditions are being set so the player can aim and shoot his bow. So to make the user understand that the player is following the mouse a target is being set in front of the player t a certain distance that is relative to the player's so it won't jump in front of him. And the parameters for shooting

the arrows are being set by getting the position of the player so the arrow will shoot in the right direction and if the right conditions are true, for instance, if a certain amount of seconds has passed since last fired arrow we take the arrows prefab to make a copy of it and shoot it by setting the correct rotation. Additionally, we play the released arrow sound and destroy the fired arrow after 2 seconds so it won't stay permanently in the scene and drop the games frame rate.

```
//-------------------------------------------------------TARGET SETTINGS-------------------------------------------------------------------//

1 reference
private void Aim()
{
    if (movementDirection != Vector2.zero)                                          //To keep the target from jumbing on the player
    {
        target.transform.localPosition = movementDirection * TargetDistance;        //Changing the position of the target to be relative to the player ( infront )
    }
}

//-------------------------------------------------------ARROWS SETTINGS-------------------------------------------------------------------//

1 reference
private void Shoot()
{
    Vector2 shootingDirection = target.transform.localPosition;                     // Direction of the target
    shootingDirection.Normalize();                                                  //So it wont change speed

    if (endOfAiming)
    {
        if (firedArrow == false)
        {
            arrow = Instantiate(arrowPrefab, transform.position, Quaternion.identity);          // Making Copies of the arrowPrefab(we change its position and rotation )
            arrow.GetComponent<Rigidbody2D>().velocity = shootingDirection * ArrowBaseSpeed;    //Fixing The Arrows Velocity
            arrow.transform.SetParent(this.transform);
            //Rotatig the arrows based on shooting direction ,Converting from radius to degree
            arrow.transform.Rotate(0, 0, Mathf.Atan2(shootingDirection.y, shootingDirection.x) * Mathf.Rad2Deg);
            audioManager.PlaySound("Arrow Release");
            Destroy(arrow, 2.0f);                                                       //Destroying the copies after 2 seconds
            firedArrow = true;
        }
    }
}
```

Figure 34: Player Movement 3rd part

4.3 Enemy Movement

The enemy movement has two categories for the different types of enemies in the game.

4.3.1 PathFinding

```
//-------------------------------------------------------UPDATE FUNCTION-------------------------------------------------------------------//
Unity Message | 0 references
void Update()
{
    if (state == SpawnState.WAITING)
    {
        if (!EnemyIsAlive())                                    // check if enemies are still alive
        {
            WaveCompleted();                                    //Begin a new round of wave
        }
        else
        {
            return;                                             //if enemies are still alive we return to avoid the below counters
        }
    }

    if (waveCountdown <= 0)
    {
        if (state != SpawnState.SPAWNING)                       //Start spawning wave
        {
            StartCoroutine(SpawnWave(waves[nextWave]));         //startCoroutine is required by the ienumerator its like a simple function
        }
    }
    else
    {
        waveCountdown -= Time.deltaTime;                        //making sure that we go down the timer for each frame,
    }
}
```

Figure 35:Pathfinding and Enemy Waves 1st part

Pathfinding is used for the zombie type of enemies that are being spawned in waves by the Player. This script contains IEnumerators that allow us to wait a certain amount of time in a method before continuing in the code file. Firstly the script checks if there are any alive enemies from the activated wave if not it initializes the next wave.

When the wave of enemies starts we will hear the according to enemy sound that has been called and spawns the corresponding amount of enemies. Inside the scene, there have been set spawn points, which we get their positions inside the function SpawnEnemy and we instantiate them according to them.

```
//-------------------------------------------------ENUMERATOR SPAWN WAVE-------------------------------------------------------------//
1 reference
IEnumerator SpawnWave(Wave _wave)                                   //we can wait a certain amount of seconds inside of a method
{
    Debug.Log("Spawning Wave: " + _wave.name);
    state = SpawnState.SPAWNING;                                    //we know that we are spwaning

    if (_wave.name == "brunette")...                                //Enemy Sounds
    if (_wave.name == "blond")...
    if (_wave.name == "rot")...
    if (_wave.name == "half")...

    for (int i = 0; i < _wave.count; i++)
    {
        SpawnEnemy(_wave.enemy);

        yield return new WaitForSeconds(1f / _wave.rate);           //wait a certain amount of seconds
    }

    state = SpawnState.WAITING;                                     //when we are waiting for player to kill the enemies

    yield break;                                                    //returns nothing is required by the ienumerator
}


//------------------------------------------------- ENEMY SPAWNER ---------------------------------------------------------------------//
1 reference
void SpawnEnemy(Transform _enemy)
{
    Debug.Log("Spawning Enemy: " + _enemy.name);

    Transform _sp = spawnPoints[UnityEngine.Random.Range(0, spawnPoints.Length)];
    Instantiate(_enemy, _sp.position, _sp.rotation).transform.SetParent(this.transform);

}
```

Figure 36: Pathfinding and Enemy Waves 2nd part

When the wave is completed a counter start going down for the next wave and all the waves in the wave array are also completed then, the difficulty of the wave raises, meaning the number of enemies that are being spawned gets logarithmically risen and we start the waves again from the start.

```
//------------------------------------------------- WAVE COMLETED ---------------------------------------------------------------------//
1 reference
void WaveCompleted()
{
    Debug.Log("Wave Completed!");

    state = SpawnState.COUNTING;                                    //countdown for the next wave
    waveCountdown = timeBetweenWaves;

    if (nextWave + 1 > waves.Length - 1)                            //if the wave is out of our array range
    {
        nextWave = 0;                                               //then start the array from the begging
        RaiseWaveDifficulty(waves[nextWave]);
        Debug.Log("ALL WAVES COMPLETE! Looping...");
    }
    else
    {
        nextWave++;
    }
}
```

Figure 37: Pathfinding and Enemy Waves 3rd part

The enemies spawned by the wave, always search for the player on the scene, and whenever the player is being found then the path of the pathfinding algorithm gets updated.

```
//-----------------------------------------------------ENUMERATOR SEARCH PLAYER-----------------------------------------------------//

4 references
IEnumerator SearchForPlayer()
{
    GameObject searchResult = GameObject.FindGameObjectWithTag("Player");

    if (searchResult == null)                           //If the search result is null
    {
        yield return new WaitForSeconds(1f);            //the wait for 0.5 seconds
        StartCoroutine(SearchForPlayer());              //and search again
    }
    else
    {
        target = searchResult;
        searchingForPlayer = false;

        StartCoroutine(UpdatePath());                   //Start updating the path again

        yield return false;
    }
}
```

Figure 38: Pathfinding and Enemy Waves 4th part

The UpdatePath method calls the path-finding seeker that is provided in the Pathfinding package, that update the enemy path so he will always follow the player

```
//-----------------------------------------------------ENUMERATOR UPDATE PATH-----------------------------------------------------//

3 references
IEnumerator UpdatePath()
{
    if (target == null)
    {
        if (!searchingForPlayer)
        {
            searchingForPlayer = true;
            StartCoroutine(SearchForPlayer());
        }

        yield return false;
    }

    //Start a new path to the target position and return the result to the OnPathComplete method
    seeker.StartPath(transform.position, target.transform.position, OnPathComplete);


    yield return new WaitForSeconds(1f / updateRate);       //wait a certain amount of seconds
    StartCoroutine(UpdatePath());                           //and call the function again
}
```

Figure 39: Pathfinding and Enemy Waves 5th part

4.3.2 Chase and Retreat

For the Slime type of enemies, a simple system of attack and retreat is being implemented. When the player enters the enemy's attack radius, the enemy starts following the player till it inflicts damage. When the player gets out of the enemy's range radius the chasing stops.

```
//-----------------------------------------------CHECK DISTANCE FUNCTION-----------------------------------------------//

2 references
public void CheckDistance()
{
    if (Vector3.Distance(targetPlayer.position, transform.position) <= chaseRadius)
    {
        chasingPlayer = true;
        anim.SetBool("Moving", true);
        transform.position = Vector3.MoveTowards(transform.position, targetPlayer.position, (reduceAttackSpeed * theSlimeController.moveSpeed) * Time.deltaTime);
    }
    else if (Vector3.Distance(targetPlayer.position, transform.position) > chaseRadius)
    {
        chasingPlayer = false;
    }
}
```

Figure 40: Chasing Attack

When the slime enemies collide with the player they inflict damage and a force is being used to pull them back and retreat from the player. For the player to have time to react and respond and enumerator is being used do the enemy will have to wait before attacking again.

```
Unity Message | 0 references
public void OnCollisionEnter2D(Collision2D other)
{
    if (other.gameObject.CompareTag("SlimeEnemy"))
    {
        Rigidbody2D slime = other.gameObject.GetComponent<Rigidbody2D>();

        if (slime != null)
        {
            Vector2 difference = slime.transform.position - transform.position;
            difference = difference.normalized * retreatDistance;

            slime.AddForce(difference, ForceMode2D.Impulse);
            StartCoroutine(KnockBackCoordinator(slime));
        }
    }
}

1 reference
private IEnumerator KnockBackCoordinator(Rigidbody2D slime)
{
    if (slime != null)
    {
        yield return new WaitForSeconds(knockBackTime);

        slime.velocity = Vector2.zero;
    }
}
```

Figure 41: Retreat

4.4 Game's UI

4.4.1 Player and Enemy Stats

The player's stats are being initialized arithmetically and they are being set for the player before the game begins.

```
1 reference
public void InitializingLevels()
{
    for (int i = 1; i < toLevelUp.Length; i++)
    {
        toLevelUp[i] = (int)(toLevelUp[i - 1] + Math.Log(i + 1) * 100);
    }

    toLevelUp[1] = 10;

    for (int i = 1; i < HPLevels.Length; i++)
    {
        HPLevels[i] = (int)(HPLevels[i - 1] + Math.Log(i + 1) * 50);
    }

    HPLevels[0] = 20;

    for (int i = 1; i < attackLevels.Length; i++)
    {
        attackLevels[i] = (int)(attackLevels[i - 1] + Math.Log(i + 1) * 8);
    }

    attackLevels[0] = 2;
}
```

Figure 42: Level Initialization

The players stats update every time the player levels up.

```
1 reference
public void LevelUp()
{
    nextLevelEXP = toLevelUp[currentLevel + 1];

    if (currentEXP >= nextLevelEXP)
    {
        sparedEXP = currentEXP - nextLevelEXP;
        currentLevel++;
        currentEXP = 0;

        audioManager.PlaySound("Level Up");

        currentHealth = HPLevels[currentLevel];
        currentAttack = attackLevels[currentLevel];

        thePlayerHealth.playerMaxHealth = currentHealth;
        thePlayerHealth.playerCurrentHealth = currentHealth;
    }
}
```

Figure 43: Leveling Up

The enemy stats works similarly to the players. Raising their attack, health, and experience point that the player will gain from them logarithmically respectively. The difference is that these are dependent on the player's level. So every time the player levels up do the enemies as well.

```
1 reference
public void InitializingStats()
{
    currentHealth = HPLevels[thePlayerStatistics.currentLevel];
    currentAttack = attackLevels[thePlayerStatistics.currentLevel];
    currentEXP = expLevels[thePlayerStatistics.currentLevel];

    theEnemyHealthManager.MaxHealth = currentHealth;
    theEnemyHealthManager.CurrentHealth = currentHealth;
    theEnemyHealthManager.expToGive = currentEXP;
    hurtPlayerScript.damageToGive = currentAttack;
}
```

Figure 44: Enemy Stats

4.4.2 Damage Numbers and Effects

The damage numbers that appear for all characters and the splash effect that appears when the enemies are hit are being instantiated when the characters take damage, meaning when an object that inflicts damage like the player's arrow or the enemies touch the player, these effects are being called and appear on the screen.

```
Unity Message | 0 references
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Enemy" || other.gameObject.tag == "SlimeEnemy")
    {
        other.gameObject.GetComponent<EnemyHealthManager>().HurtEnemy(damageToGive);

        Instantiate(damageBurst, transform.position, transform.rotation);
        var clone = (GameObject)Instantiate(damageNumber, transform.position, Quaternion.Euler(Vector3.zero));
        clone.GetComponent<FloatingNumbers>().damageNumber = damageToGive;

        Destroy(thePlayerController.arrow);
    }
}
```

Figure 45: Enemy Damage Effects

```
if (flashActive)
{
    if (flashCounter > flashLength * .66f)
    {
        playerSprite.color = new Color(playerSprite.color.r, playerSprite.color.g, playerSprite.color.b, 0f);
    }
    else if (flashCounter > flashLength * .33f)
    {
        playerSprite.color = new Color(playerSprite.color.r, playerSprite.color.g, playerSprite.color.b, 1f);
    }
    else if (flashCounter > 0)
    {
        playerSprite.color = new Color(playerSprite.color.r, playerSprite.color.g, playerSprite.color.b, 0f);
    }
    else
    {
        playerSprite.color = new Color(playerSprite.color.r, playerSprite.color.g, playerSprite.color.b, 1f);
        flashActive = false;
    }

    flashCounter -= Time.deltaTime;
}
```

Figure 46: Player Damage Effects

The player apart from the damage numbers also "flashes" momentarily to show that he has taken damage. So to achieve that, the player's opacity is being manipulated for a certain amount of time.

4.4.3 Dialogue Bubbles

The dialogue bubbles were created through Unity's UI features. By creating a panel with simple text on it. The dialogue is being activated and deactivated according to what the players choose by actually activating and deactivating the game-object and similarly, the text is being manipulated by affecting the text that exists on the dialogue panel.

```
0 references
public void ShowBox(string dialogue)
{
    dialogueBox.SetActive(true);
    dialogueActive = true;
    dialogueText.text = dialogue;
}

1 reference
public void DialogueSetUp()
{
    if (dialogueActive && Input.GetKeyUp(KeyCode.Space))
    {
        currentLine++;
    }

    if (currentLine >= dialogueLines.Length)
    {
        dialogueBox.SetActive(false);
        dialogueActive = false;

        currentLine = 0;
        thePlayer.canMove = true;
    }

    dialogueText.text = dialogueLines[currentLine];
}

2 references
public void ShowDialogue()
{
    dialogueActive = true;
    dialogueBox.SetActive(true);
    thePlayer.canMove = false;
}
```

Figure 47: Dialogue Manager

To use the dialogue bubble dynamically in the game we needed a separate script that handled the trigger for the bubbles. So whenever the player enters the collider area of a game-object and presses the key button E then the dialogue gets triggered and begins. If we want to trigger different kinds of dialogue for the character talking the MoreDialogueQuests function gets called.

```
public void MoreDialogueQuests(string[] questDialogue)
{
        theDialogueManager.dialogueLines = questDialogue;
        theDialogueManager.currentLine = 0;
        theDialogueManager.ShowDialogue();
}

Unity Message | 0 references
private void OnTriggerStay2D(Collider2D other)
{
    if (other.gameObject.name == "Player")
    {
        if (Input.GetKeyUp(KeyCode.E))
        {
            //theDialogueManager.ShowBox(dialogue);

            if (!theDialogueManager.dialogueActive)
            {
                theDialogueManager.dialogueLines = dialogueLines;
                theDialogueManager.currentLine = 0;
                theDialogueManager.ShowDialogue();
            }
        }
    }
}
```

Figure 48: Dialogue Holder

4.5 Enemy Waves Trigger

At the section 4.2.1, was explained the method of spawning waves powered with the path-finding algorithm. This section is gonna be discussed, how the waves are triggered. The waves can be triggered through the various NPC characters that wait on the maps. Through the dialogue trigger, the player

will “talk” with the NPC characters and he will be asked if he wants to trigger the enemy wave through creative dialogue. If the player wants to activate the wave he has to press the A key button. If he wants to deny the activation of the enemy wave then he has to press the D key button.

```
1 reference
public void Beginning()
{
    if (!active)
    {
        quest.dialogueLines = startText;
    }
    else
    {
        quest.dialogueLines = duringQuestText;
    }

    if (Input.GetKeyDown(KeyCode.A))
    {
        QuestStarter();
    }
    if (Input.GetKeyDown(KeyCode.S))
    {
        EndQuest();
    }
    if (Input.GetKeyDown(KeyCode.D))
    {
        quest.dialogueLines = endText;
        return;
    }
}
```

Figure 49: Wave Trigger 1st part

If the player is in the middle of the wave and he wants to stop it since the waves are infinite, he simply has to go to the NPC that activated the wave and after interacting with him he will be asked to stop the wave with the S key button or cancel by pressing D key button and continue fighting.

```
1 reference
public void QuestStarter()
{
    quest.MoreDialogueQuests(beforeQuestText);
    audioManager.StopSound("Main Music");
    waveSpawner.SetActive(true);
    audioManager.PlaySound("Wave Attack");
    active = true;
}

1 reference
public void EndQuest()
{
    audioManager.StopSound("Wave Attack");
    quest.MoreDialogueQuests(endText);
    waveSpawner.SetActive(false);
    audioManager.PlaySound("Main Music");
    active = false;
}
```

Figure 50: Wave Trigger 2nd part

4.6 Loot Table

The enemies every time they get eliminated they have a chance of dropping a loot drop item. Those chances are being calculated inside a Loot Calculator using probabilities. The loot item has a drop chance from 0 to 100. When the calculation begins we store all the percentage values inside a temporary variable to get one value.

```
2 references
public void CalculateLoot()
{
    int calc_dropChance = Random.Range(0, 101);                //to initialise the calculator

    if(calc_dropChance > dropChance)
    {
        Debug.Log("No Loot");
        return;
    }

    if (calc_dropChance <= dropChance)
    {
        itemWeight = 0;                                        //variable to store items values

        for(int i=0; i< lootTable.Length; i++)
        {
            itemWeight += lootTable[i].dropRarity;             //Add all the percentage values of all the item array
        }
```

Figure 51: Probabilities 1st part

Then we get a random value between 0 and the maximum value we got from the items the enemy might be carrying. If the random value we got matches or is close to a value of the enemies loot item then that item gets instantiated in the scene where the player was eliminated.

```
randomValue = Random.Range(0, itemWeight);              //Find a random value

for(int j = 0; j < lootTable.Length; j++)
{
    if (randomValue <= lootTable[j].dropRarity)         //If the value is less or equal to one of the elements
    {
        if(gameObject.tag == "Stalls")
        {
            Instantiate(lootTable[j].item, stallPosition, Quaternion.identity).transform.SetParent(this.transform);

        }
        else
        {
            Instantiate(lootTable[j].item, transform.position, Quaternion.identity);
            audioManager.PlaySound("Item Drop");        //Item Pick Up sound
        }
        return;
    }
```

Figure 52: Probabilities 2nd part

Apart from the enemy drops, the probability algorithm is being used for the items that the player can find in the town area. The difference between the two cases is that the items in the Town map have a 100% chance that something will appear on the stalls when in the enemies there is only a 50% chance of getting a loot item. That is the reason why the algorithm will run only two times for the enemy loot but keep running till a stall item appears.

```
randomValue -= lootTable[j].dropRarity;             //if no item was spawned then drop the rarity value

if(gameObject.tag != "Stalls")
{
    counter += 1;                                   //and try again

    if (counter >= 2)
    {
        counter = 0;
        return;
    }
}
```

Figure 53: Probabilities 3rd part

4.7 Gateways

The player can transfer between the different maps through gateways that hold a collider so they know when the player wants to change the scene as well as a name to distinguish to which area the player wants to move. When the player crosses a gateway, the new area loads through the Unity Scene Manager the music for the next area is being chosen and a loading panel appears on screen to indicate that the next is being loaded.

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Player")
    {
        SceneManager.LoadScene(levelToLoad);
        thePlayer.startPoint = exitPoint;

        if (audioManager.currentlyPlaying != "Town" && levelToLoad == "City")...else if (audioManager.currentlyPlaying != "Cave" && levelToLoad == "Cave")
        {
            audioManager.StopSound(audioManager.currentlyPlaying);
            audioManager.PlaySound("Cave");
        }else if (audioManager.currentlyPlaying != "Main Music")
        {
            audioManager.StopSound(audioManager.currentlyPlaying);
            audioManager.PlaySound("Main Music");
        }

        if (fadeInPanel != null)
        {
            GameObject panel = Instantiate(fadeInPanel, Vector3.zero, Quaternion.identity) as GameObject;   //creating and saving the panel as a game object
            Destroy(panel, 1f);                                                                              // destroy after 1 sec so it wont fill the hierarchy
        }
    }
}
```

Figure 54: Gateway Trigger

In a different script, the matching of the area names occurs and the player along with the camera gets “transported” in the new area.

```
if (thePlayer.startPoint == pointName)
{
    thePlayer.transform.position = transform.position;  // making the position of the player = to the StartPoint
    thePlayer.movementDirection = startDirection;

    theCamera = FindObjectOfType<CameraControl>();
    theCamera.transform.position = new Vector3(transform.position.x, transform.position.y, theCamera.transform.position.z);
}
```

Figure 55: Gateway Transporter

4.8 Menus

The basic code of the menus is similar for all. The player will get deactivated and the menu panel activated. For the whole game to stop we set the time scale of the game to 0 and to see the cursor of the game we need to unlock it and activate it since the player has t locked to move around. When the menus get inactive then the opposite procedure happens.

```
player.SetActive(false);

Cursor.lockState = CursorLockMode.None;
Cursor.visible = true;

mainMenuPanel.SetActive(true);
pausePanel.SetActive(false);
Time.timeScale = 0f;
```

Figure 56: Basic Menu behaviour

4.9 Saving the Game

For the process of saving the game, a binary file is being created for the player data to get saved in it.

```
1 reference
public static void SavePlayer(PlayerStats player)
{
    BinaryFormatter formatter = new BinaryFormatter();
    string path = Application.persistentDataPath + "./player.savefile";
    FileStream stream = new FileStream(path, FileMode.Create);

    PlayerData data = new PlayerData(player);

    formatter.Serialize(stream, data);
    stream.Close();
}
```

Figure 57: Saving the game

In a different script file the level of the player is stored so it can be saved in the binary file. Only the level of the player is being stored since with it the according health and level will appear on the screen. The player has to have in mind that exp will not be stored when he saves the game only his level.

```
public class PlayerData
{
    public int level;

    1 reference
    public PlayerData(PlayerStats player)
    {
        level = player.currentLevel;
    }
}
```

Figure 58: Storing the players level

The load player function will go to the created binary file and return anything that has been stored in the binary path and bring it back to the player. This will replace the player's stats with the saved ones.

```
public static PlayerData LoadPlayer()
{
    string path = Application.persistentDataPath + "./player.savefile";

    if (File.Exists(path))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream = new FileStream(path, FileMode.Open);

        PlayerData data = formatter.Deserialize(stream) as PlayerData;
        stream.Close();

        return data;
    }
    else
    {
        return null;
    }
}
```

Figure 59: Loading the game

Lastly the function Delete Data will get activated whenever the player chooses a new game or the restart option in the Restart menu. This function simply finds the existing binary file and deletes it, achieving the desired result of resetting the player's level.

```
2 references
public static void DeleteData()
{
    string path = Application.persistentDataPath + "./player.savefile";

    File.Delete(path);
}
```

Figure 60: Resetting the game

4.10 Audio Manager

The audio manager as the name suggests manages all the audio files in the game. By making a custom class that stores the information of each audio clip along with the sound clips volume, pitch, and randomness to create the illusion of depth for the player.

```
[System.Serializable]                           //so it can be viewed by the inspector
1 reference
public class Sound                              //custom class
{
    [Space]
    [Header("Name & Sound Clip:")]
    public string name;                         //name of the sound
    public AudioClip clip;                      // the audio file we wanna use

    [Space]
    [Header("Volume & Pitch:")]
    [Range(0f, 1f)]                             //Creates a slider and Restricts the variable volume to be between 0 and 1
    public float volume = 0.7f;                 // the volume of the sound
    [Range(0.5f, 1.5f)]
    public float pitch = 1f;                    //how high or low the sound will be

    [Space]
    [Header("Randomness for Volume & Pitch:")]
    [Range(0f, 0.5f)]
    public float randomVolume = 0.1f;           //random volume
    [Range(0f, 0.5f)]
    public float radnomPitch = 0.1f;            //random pitch

    public bool loop = false;

    private AudioSource source;                 //refence for the audio source

    1 reference
    public void SetSource(AudioSource _source)  //method to set our audio reference
    {
        source = _source;
        source.clip = clip;                     //setting the clip to our source
        source.loop = loop;
    }
```

Figure 61: Audio Manager 1st part

With the creation of the function Play, we can indicate when the audio will start with randomized volume and pitch and the function Stop to stop the audio clips.

```
1 reference
public void Play()
{
    source.volume = volume * (1 + Random.Range(-randomVolume / 2f, randomVolume / 2f));
    source.pitch = pitch * (1 + Random.Range(-randomVolume / 2f, randomVolume / 2f));

    source.Play();
}

1 reference
public void Stop()
{
    source.Stop();
}
```

Figure 62: Audio Manager 2nd part

As already demonstrated in the code the audio manager gets called with the appropriate function names PlaySound and StopSound. To be able to change the music played in the current scene a variable of the currently playing music is being set every time a new type of music clip occurs.

```
33 references
public void PlaySound(string _name)
{
    for (int i = 0; i < sounds.Length; i++)
    {
        if (sounds[i].name == _name)
        {
            sounds[i].Play();
            if(_name == "Main Music" || _name == "Cave" || _name== "Town" || _name == "Wave Attack")
            {
                currentlyPlaying = _name;
                Debug.LogError("CURRENTLY PLAYING " + currentlyPlaying);
            }
            return;
        }
    }

    //no sound with the name we want
    Debug.LogWarning("Audio Manager: Sound not Found: " + _name);
}

7 references
public void StopSound(string _name)
{
    for (int i = 0; i < sounds.Length; i++)
    {
        if (sounds[i].name == _name)
        {
            sounds[i].Stop();
            return;
        }
    }

    //no sound with the name we want
    Debug.LogWarning("Audio Manager: Sound not Found: " + _name);
}
```

Figure 63: Audio Manager 3rd part

5. Epilogue

5.1 Conclusions

In the game Bow and Jelly, I have implemented a two dimensional game with backgrounds and maps of pixel art style with various object animations. The individual characters all have distinct and varied designs and unique animations, also enriched by pathfinding algorithms to simulate smart chasing behaviour. Item drops are based on the probability matrix which makes each item have unique drop rates and healing power. The UI of the game is made with a fun and inviting design that appeals to younger audiences. The game also has character-specific music animation sounds and ambient background music, which allows for a more immersive and enjoyable experience. The key to randomly generated shoot ‘em up is to keep the gaming loop engaging and challenging enough so that the gamers keep coming back for more challenges.

5.2 Difficulties During Implementation

The first obstacle that I encountered during the development of “Bow and Jelly” was not having prior experience with graphic design and game design. Tutorials and websites like Stack-Overflow helped me figure out the minor issues I ran into during development. Enemy spawn overlap was another issue which took up a good chunk of development time, enemies were being generated at the same position. Another enemy problem was the pathfinding algorithms create many situations where the enemy is stuck following the start point of the player instead of the actual player; Also the enemies often run around in circles and cannot follow the player. The recent and most annoying issue was the main menu scene which kept bringing up errors and could not reset the player as it was supposed to, which in the end I found an alternative solution of making the main menu into a panel instead of a different scene.

5.3 Future work and Extensions

This game would be a stepping stone to many other games with a much larger scope in mind on other more complicated game engines such as Unreal Engine, as well as experimenting with much more advanced and state of the art techniques like ray tracing. The current game also has a pretty generic AI which has deductible routines that can be outsmarted by paying close attention which can be greatly improved by adopting more rounded techniques like surrounding awareness and progressive learning. The games could be further developed for multi-platform ranging from small-scale mobile games to computer games of retail quality.

Bibliography

1. https://docs.unity3d.com/Manual/
2. https://assetstore.unity.com/packages/2d/environments/2d-hand-painted-town-tileset-67346
3. https://assetstore.unity.com/packages/2d/environments/2d-hand-painted-grassland-tileset-47763
4. https://www.youtube.com/playlist?list=PLPV2KyIb3jR42oVBU6K2DIL6Y22Ry9J1c
5. https://www.youtube.com/playlist?list=PL4vbr3u7UKWp0iM1WIfRjCDTI03u43Zfu
6. https://www.youtube.com/playlist?list=PLM83Z6G5iM3k48356VU6e-oXWl_uwwq4F
7. https://www.youtube.com/playlist?list=PLiyfvmtjWC_X6e0EYLPczO9tNCkm2dzkm
8. https://www.youtube.com/watch?v=W4SE0_cfAqc&list=PLTDA0MzBeKMeZHyoKuW20yVJHs7ZDgGO3&index=10&t=178s
9. https://www.youtube.com/watch?v=cWKhytYUGTg&list=PLTDA0MzBeKMeZHyoKuW20yVJHs7ZDgGO3&index=34
10. https://www.youtube.com/watch?v=N4Z4MdZ1KWY&list=PLTDA0MzBeKMeZHyoKuW20yVJHs7ZDgGO3&index=44
11. https://www.youtube.com/watch?v=J8MH-k0Fa6Y&list=PLTDA0MzBeKMeZHyoKuW20yVJHs7ZDgGO3&index=39
12. https://www.youtube.com/watch?v=Md6W79jtLJM&list=PLTDA0MzBeKMeZHyoKuW20yVJHs7ZDgGO3&index=48
13. https://www.zapsplat.com/
14. https://www.bensound.com/royalty-free-music/3
15. https://www.coursera.org/lecture/more-programming-unity/collision-free-spawning-7c02S
16. https://arongranberg.com/astar/
17. https://learn.unity.com/tutorial
18. https://en.wikipedia.org/wiki/Unity_(game_engine)