Development Document for:

Star Shooter

The Sci-Fi Top-Down Shooter

Written by Tracy Potter, Jacob Aguilar, Spencer Meren

Monday, March 24, 2025

Table of Contents

Sprint 1: The Concept

Project Overview

Star Shooter is a top-down sci-fi shooter game with pixel art that offers gamers round-based challenges. The player must fight through endless waves of enemies, upgrading weapons to achieve a new high score. A loadout of standard weapons can temporarily be enhanced by collecting a power-up for special weapons. Boss enemies will appear at breakpoint rounds as a sign that the difficulty is about to ramp up.

Key Areas of Computer Science

Algorithms and Data Structures:

- Efficiently manage enemy waves, pathfinding, and AI behavior using algorithms like A* or state machines
- Implement dynamic data structures (queues for managing enemy spawns, heaps for priority-based resource management, etc.)

Artificial Intelligence:

- Program enemy AI with pathfinding, decision-making algorithms, and behaviors (seeking, evading, or swarming)
- Create boss enemies with state-based AI and adaptive difficulty

Mathematics and Physics:

- Apply vector math for player movement and enemy tracking
- Use physics-based mechanics for weapon projectiles, hit detection, and collision resolution

Mock-up

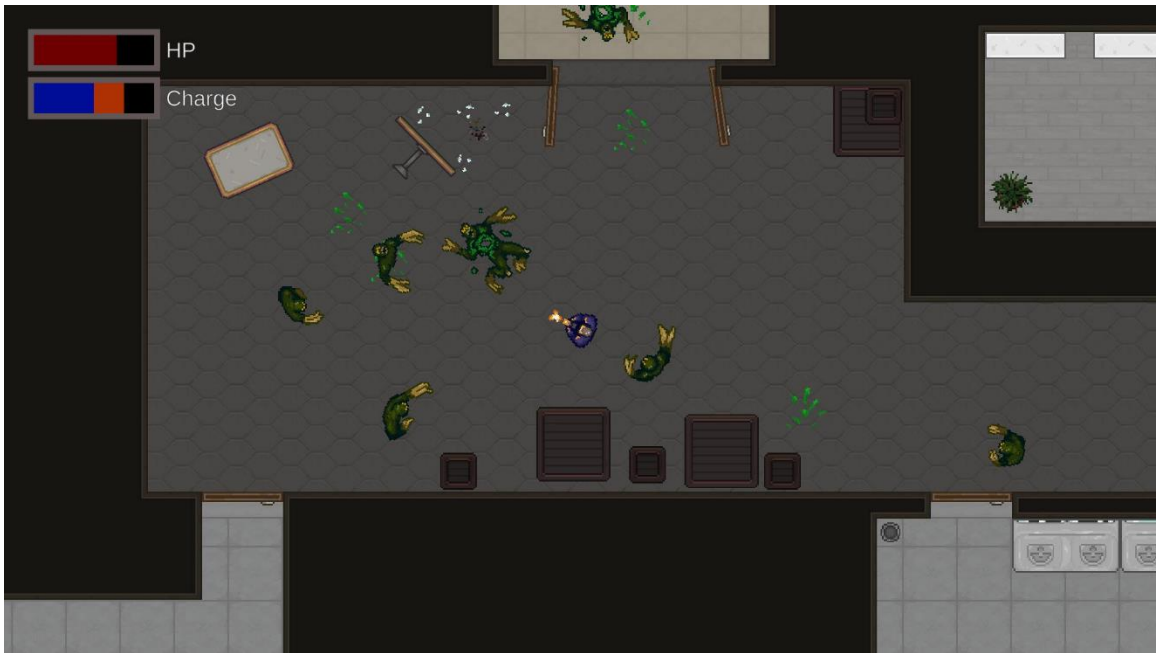


Figure 1: The image above is a simple mock-up of what the game would look like through editing predefined scenes that were included from paid assets obtained from the Unity Assets Store

Making the Game Design Document

For our capstone project, we are undertaking the development of a video game. To ensure the project's success and maintain a structured approach, we will adopt the industry-standard practice of creating a Game Design Document (GDD). This document will serve as a comprehensive blueprint, detailing the game's vision, mechanics, narrative, and technical specifications. It will act as a key reference point for all team members, ensuring alignment across the development cycle and facilitating seamless collaboration between developers, designers, and stakeholders. This approach will simulate the professional development environment found in the game industry.

Collaboration

Our team will convene during the weekend—Saturday, Sunday, or Monday afternoons—to engage in collaborative development efforts. The workflow will be highly coordinated, strategically dividing tasks between team members. For example, one member may focus on designing and developing individual game scenes, while another will concentrate on implementing underlying game logic and code architecture. This division of labor ensures parallel progress and efficient task execution, leveraging the strengths of each team member. Version control will be managed via Unity Version Control (Plastic SCM) integrated with GitHub to facilitate seamless collaboration and continuous integration across all development phases.

Impacts

Entertainment: This product is meant to be for enjoyment for those interested, especially in the sub-genre of top-down or sci-fi shooters

Environment: By producing the game digitally, it will minimize the environmental footprint in terms of manufacturing, packaging, and shipping of physical copies by reducing wastes such as plastic cases, discs, etc.). This also allows the reduction in carbon emissions associated with shipment transportation.

Economic: Has the potential to increase spending in the gaming market, such as buying new headsets, mouse, controllers, etc.

Local and Global: Could be an inspiration for others pursuing development to start their own journey in game development

Concerns

Legal:

- All game assets will be utilized in strict compliance with applicable licensing agreements, ensuring full adherence to intellectual property laws and preventing any infringement on copyrights.

Ethical:

- The portrayal of violence in video games remains a topic of concern for certain demographic groups. To address this, our game will feature stylized, non-graphic combat mechanics, intentionally designed to maintain a tone suitable for a broader, more diverse audience.
- We will also prioritize accessibility by integrating features such as customizable control remapping, ensuring the game remains inclusive and accessible to players with various needs.

Sprint 2: The Mechanics

Sketch of UI

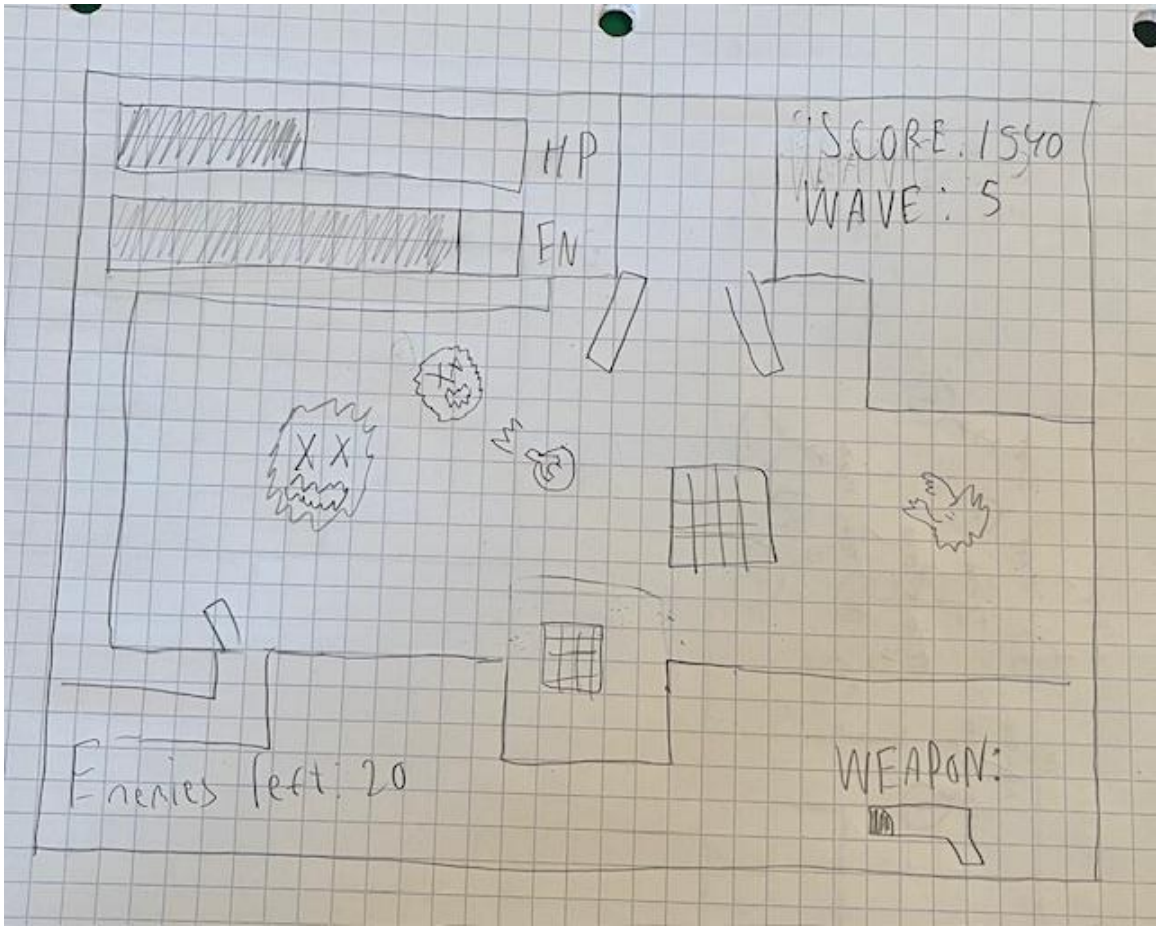


Figure 2: A simple hand-drawn sketch of what our UI would look like in the game

User Interface Elements:

- **Health Meter:** A dynamic visual representation of the player's current health status, updating in real-time based on in-game events and player actions.
- **Energy Meter:** A resource bar displaying the player's available energy, which depletes and regenerates based on specific in-game mechanics, influencing the use of special abilities or actions.
- **Score:** A numerical value reflecting the player's progress or performance, incremented based on specific in-game achievements, such as defeating enemies or completing objectives.
- **Round Indicator:** A display showing the current round or level the player is engaged in, updating as they progress through the game's stages or challenges.
- **Weapon Loadout:** A section detailing the player's available weapons or equipment, highlighting the current selection and allowing for quick access or changes during gameplay.
- **Remaining Enemies:** A counter indicating the number of enemies still present in the current round or stage, providing players with an objective to focus on.

Key Areas of Functionality

Player Movement:

- The player will have full control over movement, with support for both keyboard and mouse input as well as game controller configurations. Movement will be responsive, with smooth transitions across different control schemes for optimal player experience.

Shooting Mechanics:

- The game will feature multiple weapon types, each with distinct characteristics. These include variations in firing rate, spread patterns, reload mechanics, and ammunition usage. Weapons will be designed with unique functionalities to offer players diverse combat strategies.

Enemy AI:

- Enemies will be controlled by an advanced AI system utilizing A* pathfinding algorithms for navigation. Modifications to the basic pathfinding will be implemented to accommodate specific enemy behavior patterns, ensuring that different enemy types react and engage the player in unique ways.

Health System:

- The player's health will regenerate over time based on predefined conditions, with a maximum health threshold that varies depending on player upgrades or power-ups. Enemies will exhibit varying health points (HP), with different classes of enemies possessing distinct durability levels, impacting their survivability and combat behavior.

User Interface (UI):

- The UI will include various elements such as:
    - **Main Menu** and **Settings** for player customization and configuration.
    - **Health** bar to indicate the player's current vitality.
    - **Ammo count** for tracking ammunition status.
    - **Weapon display** for showing equipped items and their attributes.
    - **Power-ups** and their active duration or effects.
    - **Round indicator** to track the player's progression through game stages.

Gameplay:

- The game will operate on an endless, round-based system where each round presents increasing difficulty through enemy variety and complexity. Rounds will continue until a set game-over condition is met, promoting strategic play and long-term engagement.

Player Input and Nature of Outputs

Player Input:

- Player input can be provided through either a keyboard and mouse (KBM) or a game controller, with each control scheme offering tailored actions:
    - **Movement:** WASD keys for KBM; left thumb stick for controller.
    - **Aiming:** Mouse cursor for KBM; right thumb stick for controller.
    - **Shooting:** Left mouse button for KBM; right trigger for controller.
    - **Throw Grenade:** Right mouse button for KBM; left trigger for controller.
    - **Switch Weapon:** Assigned key (e.g., 1, 2, 3, etc.) for KBM; unbound for controller.
    - **Cycle Weapons:** Mouse scrolls up/down for KBM; left and right bumpers for controller.
    - **Melee Attack:** Spacebar for KBM; right stick button (R3) for controller.
    - Default control configurations are fully customizable from the in-game settings menu to accommodate player preferences.

```
using UnityEngine;

0 references
public class PlayerMovement : MonoBehaviour
{
    1 reference
    public float moveSpeed = 5f; // Speed of player movement
    2 references
    public Rigidbody2D rb; // Reference to Rigidbody2D component
    2 references
    public Animator animateLegs;

    7 references
    Vector2 movement; // Stores player movement input

    0 references
    void Update()
    {
        // Get movement input from player
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");
        movement.Normalize();

        if (movement != Vector2.zero) {
            animateLegs.Play("Player_Legs_Walk");
            float angle = Mathf.Atan2(movement.y, movement.x) * Mathf.Rad2Deg;
            transform.rotation = Quaternion.Euler(new Vector3(0, 0, angle));
        }
        else {
            animateLegs.Play("Player_Legs_Idle");
        }
    }

    0 references
    void FixedUpdate()
    {
        // Move player using Rigidbody2D
        rb.MovePosition(rb.position + movement * moveSpeed * Time.fixedDeltaTime);
    }
}
```

Figure 3A code snippet that demonstrates the player's movement

Visuals:

- Player and Enemy Representation: The player and enemies are visually represented using animated sprites. These sprites are segmented into two parts—upper body and lower body—enabling the simultaneous play of walking animations while performing other actions (e.g., shooting, switching weapons).
- Environment Representation: A tile map system is employed to create and represent the game environment, with levels consisting of tile-based grids for efficient rendering and collision detection.
- UI Elements: The user interface displays critical in-game information, including player health, energy, current wave number, and extra lives.
- Damage Indicators: Particle effects and lingering decal sprites are used to visually represent damage to the player and enemies, enhancing feedback during combat scenarios.

Audio:

- Sound Effects: Custom sound effects are implemented for various player and enemy actions, including walking, shooting, throwing grenades, and weapon switching, providing immersive auditory feedback for each action.
- Combat Sounds: Additional sound effects are triggered when successful hits are landed on enemies, providing auditory confirmation of effective attacks.

Normalize Diagonal Movement

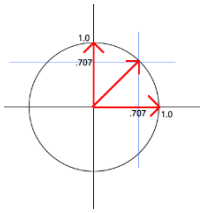


Figure 4: A depiction of how Vector2.Normalize() works

To ensure consistent player movement speed in all directions, diagonal movement is normalized using the Vector2.Normalize() function. This process scales the directional input vectors for both the X and Y axes so that the player's movement speed remains constant, regardless of the direction. Specifically, when the player moves diagonally (e.g., pressing both the "W" and "D" keys), the velocity vector is normalized to prevent faster movement along the diagonal compared to vertical or horizontal movement. This ensures a more predictable and uniform movement experience for the player, maintaining precise control over character movement.

APIs & Data Structures

NavMesh:

- The *A pathfinding** algorithm is utilized for enemy navigation, enabling enemies to intelligently follow the player across the environment. The pathfinding system leverages Unity's NavMesh system, with dynamic updates to ensure that enemies can navigate around obstacles and choose the most efficient route toward the player, enhancing the overall challenge and engagement during gameplay.

Unity Input System:

- The **Unity Input System** manages all player input, allowing for smooth and responsive player movement. It supports both keyboard and mouse (KBM) and game controller configurations, ensuring flexibility in control schemes. This system interprets player actions, translating them into real-time movement and interactions within the game world.

Physics:

- The **Physics Engine** governs the interaction between game objects, handling the collision detection and response between bullets, the player, and enemies. When bullets are fired, their trajectory and impact are calculated, detecting when they hit an enemy or the player. Appropriate reactions, such as damage or destruction, are triggered based on these collisions, ensuring realistic and engaging gameplay mechanics.

Cinemachine:

- Cinemachine is employed to manage the camera system, allowing it to follow the player dynamically during regular gameplay. The camera's movement is smooth and responsive, ensuring an optimal view of the action. Additionally, Cinemachine can be utilized for special in-game events such as cutscenes or scripted sequences, where precise camera control and cinematic effects are required to enhance storytelling and player immersion.

Developmental Tools

Unity 2022.3.46f1 - A game development platform. Allows users to create complex 2D and 3D scenes using highly flexible classes known as "Game Objects". Game Objects can be arranged in a tree-

esque hierarchy, and can be assigned various “components” to suit the specific needs of the various entities within a scene.

Visual Studio Code - An integrated development platform. Highly customizable due to its support of third-party extensions, and allows for integration with the Unity editor as a result. Used to modify the C# scripts required for game logic in Unity.

Unity Store - A marketplace that provides various user-created assets for use in projects. Used to obtain the current sprite and environment artwork used in our project.

Sprint 3: The Basics

Simple Enemy Pathfinding

Currently, enemies initiate pursuit when they are within proximity to the player, utilizing a basic pathfinding system. As development progresses, this system will be upgraded to incorporate *A pathfinding**, allowing for more advanced navigation. This enhancement will enable enemies to dynamically calculate optimal routes while avoiding obstacles, resulting in more intelligent and challenging AI behavior.

Assets > Scripts > EnemyPathfinding2D.cs > ...

```
using UnityEngine;

0 references
public class EnemyPathfinding2D : MonoBehaviour
{
    2 references
    public Transform player;         // Reference to the player's transform
    1 reference
    public float moveSpeed = 2f;     // Speed of the enemy
    1 reference
    public float chaseDistance = 5f; // Distance at which the enemy starts chasing the player

    3 references
    private Rigidbody2D rb;          // Reference to the enemy's Rigidbody2D
    3 references
    private Vector2 movement;        // Stores the direction of movement

    0 references
    void Start()
    {
        // Get the Rigidbody2D component attached to the enemy
        rb = GetComponent<Rigidbody2D>();
    }

    0 references
    void Update()
    {
        // Calculate the distance between the enemy and the player
        float distanceToPlayer = Vector2.Distance(player.position, transform.position);

        if (distanceToPlayer <= chaseDistance)
        {
            // Calculate the direction vector from the enemy to the player
            Vector2 direction = (player.position - transform.position).normalized;

            // Store the direction in the movement variable
            movement = direction;
        }
        else
        {
            // If the player is out of range, stop movement
            movement = Vector2.zero;
        }
    }

    0 references
    void FixedUpdate()
    {
        // Move the enemy using Rigidbody2D
        rb.MovePosition(rb.position + movement * moveSpeed * Time.fixedDeltaTime);
    }
}

```

Figure 5: A code snippet showing off a simple enemy pathfinding algorithm

Camera Movement

Since our game features moderately large-scale levels, we aim to maintain an optimal player experience by ensuring that the camera dynamically follows the player rather than displaying the entire map at once. Expanding the camera to encompass the full level would reduce the player's on-screen presence, making them appear too small and diminishing gameplay clarity. Instead, we will implement a player-following camera system to provide smooth tracking and maintain an ideal field of view. This

approach ensures that the player remains the focal point while preserving the visibility of nearby gameplay elements, enhancing both immersion and playability.

Assets > Scripts > CameraFollow.cs > ...

```
using UnityEngine;

0 references
public class CameraFollow : MonoBehaviour
{
    3 references
    public Transform target;
    1 reference
    public float smoothSpeed = 0.125f;
    1 reference
    public Vector3 offset;

    0 references
    void LateUpdate()
    {
        if (target == null)
            return;

        Vector3 desiredPosition = new Vector3(target.position.x, target.position.y, -10) + offset;
        Vector3 smoothedPosition = Vector3.Lerp(transform.position, desiredPosition, smoothSpeed);
        transform.position = smoothedPosition;
    }
}
```

Figure 6: A code snippet of how the camera will follow the player by getting the position of the player

Mouse Movement

The player will be rotated in the direction of the camera to make it easier to control the player within the game.

```
mouse_pos = Input.mousePosition;
mouse_pos.z = -(transform.position.z - Camera.main.transform.position.z);
object_pos = Camera.main.WorldToScreenPoint(transform.position);
mouse_pos.x = mouse_pos.x - object_pos.x;
mouse_pos.y = mouse_pos.y - object_pos.y;
angle = Mathf.Atan2(mouse_pos.y, mouse_pos.x) * Mathf.Rad2Deg;
transform.rotation = Quaternion.Euler(new Vector3(0, 0, angle));
```

Figure 7: The calculation being performed to move the player in direction relating to the mouse

Shooting Ability

The player's shooting is done through the calculation of the player's direction angle using the coordinates of the mouse and reads input from the mouse if there was a left click to fire a bullet based on the angle.

```
mouse_pos = Input.mousePosition;
mouse_pos.z = -(transform.position.z - Camera.main.transform.position.z);
object_pos = Camera.main.WorldToScreenPoint(transform.position);
mouse_pos.x = mouse_pos.x - object_pos.x;
mouse_pos.y = mouse_pos.y - object_pos.y;
angle = Mathf.Atan2(mouse_pos.y, mouse_pos.x) * Mathf.Rad2Deg;
```

Figure 8: The calculation done to have the bullet aim in the direction the player is facing

```
if (Input.GetMouseButtonDown(0)) {
    bullet = Instantiate(bulletPrefab);
    bullet.transform.position = bulletSpawnPoint.position;
    bulletRB = bullet.GetComponent<Rigidbody2D>();

    // Calculate bullet direction using the player's angle
    Vector2 bulletDirection = new Vector2(Mathf.Cos(angle * Mathf.Deg2Rad), Mathf.Sin(angle *
    bulletRB.velocity = bulletDirection * bulletSpeed;
}
```

Figure 9: When the player left clicks on the mouse, this will create a bullet prefab which will travel in the direction that the player is shooting

USB Support

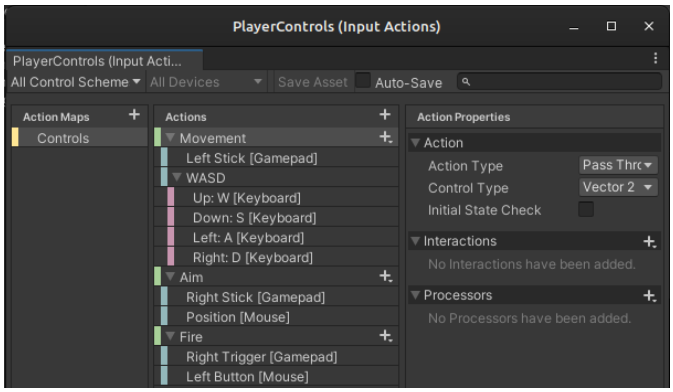


Figure 10: Unity's Input System with actions incorporated to allow the player to move, aim, and fire their weapon

Unity's Input System package allows developers to map many inputs across multiple devices to the various actions a player can take; the package includes support for switching control schemes mid-session. By using the "Controls Changed Event", a script can be notified of control scheme changes in order to handle largely different control schemes; in our project, this allows the player's rotation to be controlled by either the mouse cursor (character faces towards mouse) or a gamepad's right stick (character faces in direction of the stick).

```
public void OnDeviceChange()
{
    PlayerInput pin = GetComponent<PlayerInput>();
    isGamepad = pin.currentControlScheme.Equals("Gamepad") ? true : false;
}
```

Figure 11: This demonstrates getting the player input and whether a gamepad controller is connected

Sprint 4: Starting the Paper Prototype

GDD Complete

With the Game Design Document (GDD) complete, we can effectively showcase the creative process behind our game's design. The GDD serves as a structured framework that consolidates our ideas, mechanics, and concepts into a tangible format, allowing us to critically evaluate what works well and identify areas for improvement. By refining our vision before full-scale development begins, we can establish a clear direction, mitigate potential design challenges, and ensure a more cohesive and well-executed final product.

Player-Enemy Interactions

The player fires bullets to inflict damage on enemies, with hit detection managed through collision detection within the enemy script. When an enemy enters a predefined proximity to the player, it initiates an attack sequence, which is controlled by the attack() function in the enemy script. Both player and enemy death animations are triggered and processed within the Update() function, ensuring smooth transitions and proper visual feedback upon elimination.

```
0 references
void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("bullet")) {
        // Debug.Log("Enemy hit by bullet");
        health -= 5;
        Destroy(other.gameObject);
        if (health <= 0) {
            isDead = true;
        }

    }
}

1 reference
IEnumerator DeathTimer(float delay) {
    yield return new WaitForSeconds(delay);
    Destroy(gameObject);
}

1 reference
IEnumerator Attack() {
    animateBody.Play("Enemy_Attack");
    playerMovement.health -= 25;
    yield return new WaitForSeconds(3f);
    isAttacking = false;
}
```

Figure 12: Here it is checked if the enemy is hit by a bullet, and if so, depletes their health, as well as if they die to trigger an animation, along with an animation when attacking

```
if (isDead) {
// stop enemy movement
movement = Vector2.zero;
transform.position += new Vector3(0, 0, 0.2f);
// toggle active states of collider and GOs
transform.GetComponent<CircleCollider2D>().enabled = false;
transform.GetChild(0).gameObject.SetActive(false);
transform.GetChild(1).gameObject.SetActive(false);
transform.GetChild(2).gameObject.SetActive(true);
// animate death and start death timer
animateDeath.Play("Enemy_Death");
StartCoroutine(DeathTimer(5f));
}
```

Figure 13: If the enemy is dead, the movement will stop and their collider with be inactive and start the death animation

```
if (health <= 0) {
    // animate player death
    transform.GetChild(0).gameObject.SetActive(false);
    transform.GetChild(1).gameObject.SetActive(false);
    transform.GetChild(2).gameObject.SetActive(false);
    transform.GetChild(3).gameObject.SetActive(true);
    animateDeath.Play("Player_Death");
    return;
}
```

Figure 14: If the player has no health remaining, the death animation is played

Settings and UI Mock-up

Figure 15: Another UI mockup design to demonstrate gameplay

A settings menu has been implemented to allow players to adjust the game's volume and brightness. This feature ensures that players can tailor the audiovisual experience to their personal preferences, enhancing comfort and accessibility. Volume adjustments affect overall sound levels, including music, sound effects, and ambient noise, while brightness settings modify screen visibility to accommodate different lighting conditions and player needs.

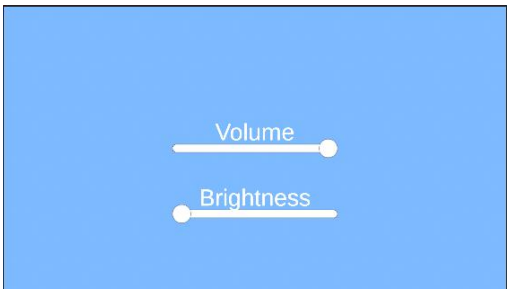


Figure 16: A basic settings screen to show functionality of the volume and brightness settings

Paper Prototype

https://docs.google.com/document/d/1O94K_yUUxkX-eL6OX1-4mKwiJgZI7L1W4AkPnBOv0Vs/edit?tab=t.0#heading=h.hfo0xthqp4iy

This paper prototype is designed to simulate the core mechanics of our game in a physical, board game-like format. By translating the digital experience into a tangible form, we can better analyze gameplay flow, mechanics, and interactions. This approach allows players to physically engage with the game's structure, making it easier to identify potential improvements, refine mechanics, and ensure clarity in design before full-scale digital development begins.

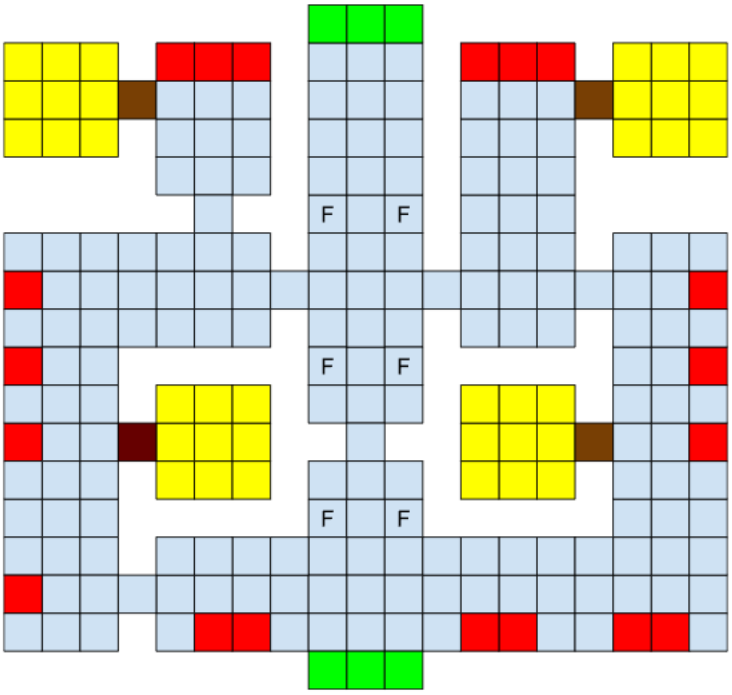


Figure 17: This is a digital representation of the physical board layout that will be used for the paper prototype

Sprint 5: Completion of the Paper Prototype

Finished Prototype & Updated Rule sheet

The document now includes an in-depth breakdown of how our game can be played physically as a board game. To enhance clarity and structure, we have updated our rule sheet to reflect the finalized mechanics and interactions. Additionally, we incorporated an additional round of playtest feedback, refining our gameplay based on player observations and insights.
To document the iterative development process, we have finalized the ruleset and included a changelog detailing modifications made after each playtest. Furthermore, a gameplay demonstration video has been added to showcase the physical prototype in action, providing a visual representation of how the mechanics translate into a tangible format.

https://drive.google.com/file/d/1DSjk60E8rcNn8QyLTHxxB107uqgnCnwI/view?usp=sharing

Pause Functionality

When the player presses the Escape key on the keyboard or the Start button on the controller, the game enters a fully paused state, freezing all gameplay elements, including player movement, enemy behavior, animations, and timers. This state remains active until the player presses the respective button again, at which point the game seamlessly resumes from where it left off. This pause functionality ensures that players can temporarily stop gameplay without any unintended progression or interactions occurring in the background.

Figure 18: This displays which button is used for pausing

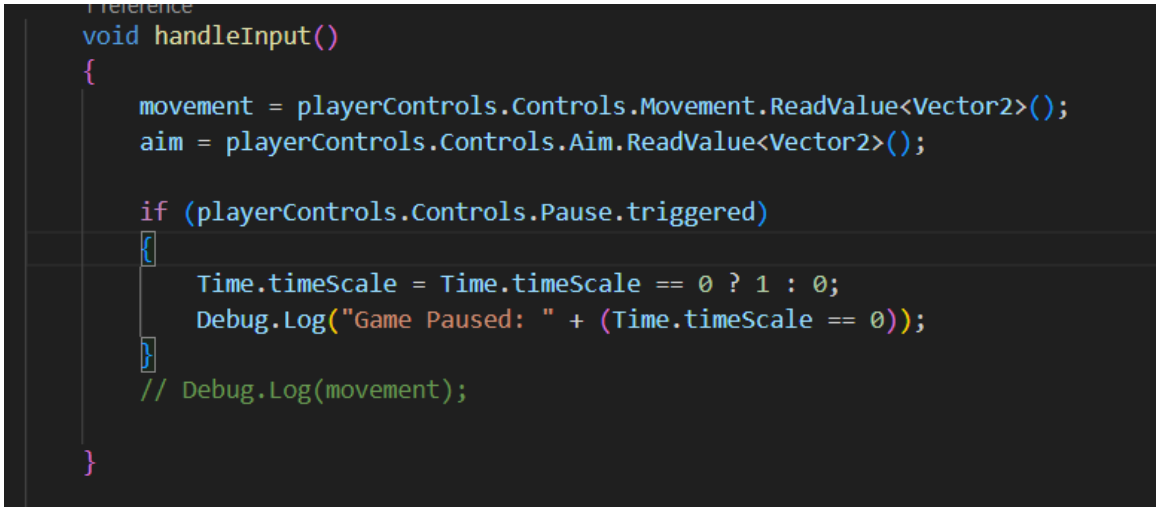


Figure 19: When the game is paused, everything in time scale is set to 0, freezing everything

Addition of Music & Sound Effects

Upon loading the game, the start menu will automatically begin playing background music, enhancing the immersive experience from the outset. The volume of this music, along with other audio elements, can be adjusted through the settings menu, allowing players to customize their sound preferences for a more comfortable and personalized experience. These settings ensure that audio levels remain consistent across different game sessions. When the player shoots, a corresponding sound effect is played to provide immediate auditory feedback. Additionally, a capped rate of fire is implemented to prevent unlimited rapid firing, ensuring balanced gameplay and encouraging strategic shooting mechanics.

```
// Music by Nicholas Panek from Pixabay

using UnityEngine;

1 reference
public class PersistentAudio : MonoBehaviour
{
    3 references
    private static PersistentAudio instance;

    0 references
    private void Awake()
    {
        if (instance != null && instance != this)
        {
            Destroy(gameObject); // Destroy duplicates
            return;
        }

        instance = this;
        DontDestroyOnLoad(gameObject); // Make persistent across scenes
    }
}
```

Figure 20: The music that is played throughout the game is loaded and persists throughout gameplay

```
if (Time.timeScale > 0 && Input.GetMouseButtonDown(0) && Time.time >= nextFireTime) {
    nextFireTime = Time.time + fireRate;

    bullet = Instantiate(bulletPrefab);
    bullet.transform.position = bulletSpawnPoint.position;
    bulletRB = bullet.GetComponent<Rigidbody2D>();

    // Calculate bullet direction using the player's angle
    Vector2 bulletDirection = new Vector2(Mathf.Cos(angle * Mathf.Deg2Rad), Mathf.Sin(angle * Mathf.Deg2Rad));
    bulletRB.velocity = bulletDirection * bulletSpeed;

    // pew
    gunAudio.Play();
}
```

Figure 21: Check to see if the bullet is shot and if so, cue the audio for the bullet

Start of HUD Development

A health bar has been implemented, successfully displaying the player's health in real time. However, its positioning is inconsistent across different screen resolutions, causing variations in its display location. To resolve this, further research into Unity's UI system, including Canvas scaling options and anchor settings, is required to ensure that the health bar maintains a fixed and responsive position regardless of resolution or aspect ratio.

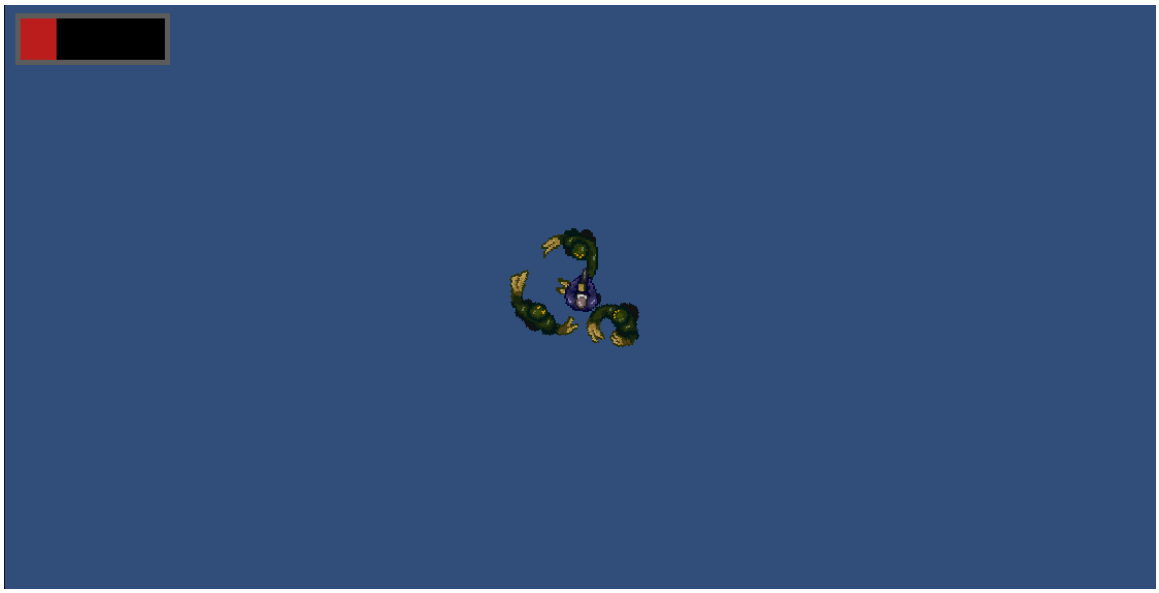


Figure 22: Here is the health bar in Unity's Free Aspect display

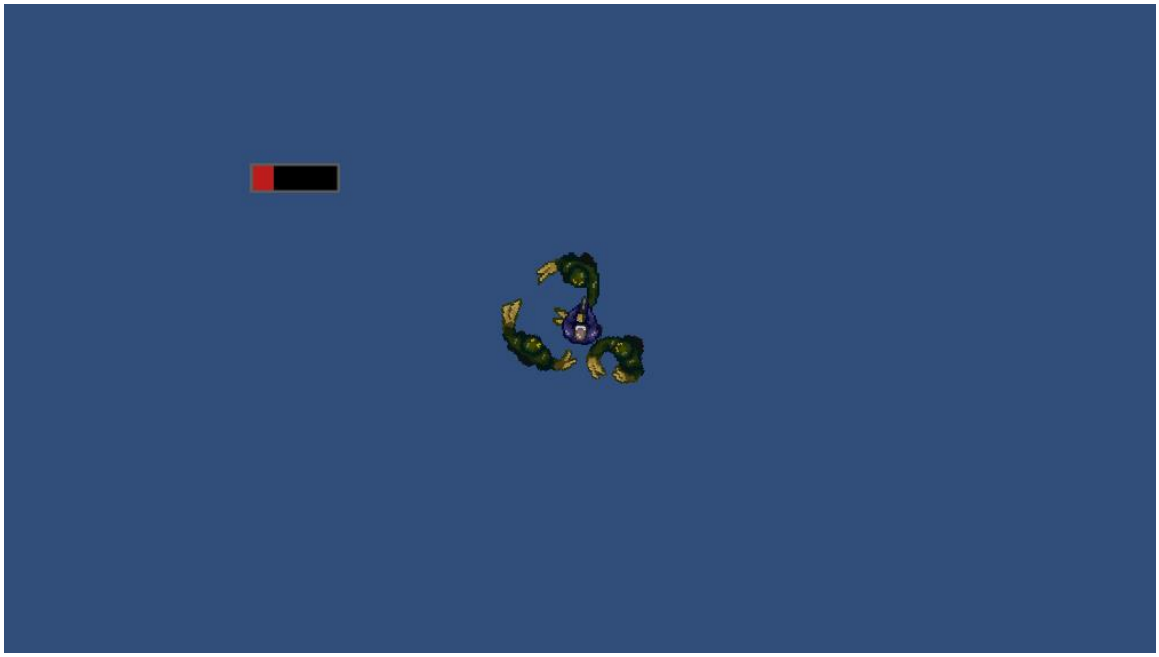


Figure 23: Here is the health bar in 1920 x 1080 display

Sprint 6: Advanced Mechanics

Additional Sound effects

Enemies now play a sound effect when attacking, providing immediate auditory feedback to enhance immersion and reinforce player awareness of incoming threats. This implementation improves the overall game feel by making enemy interactions more dynamic and engaging.

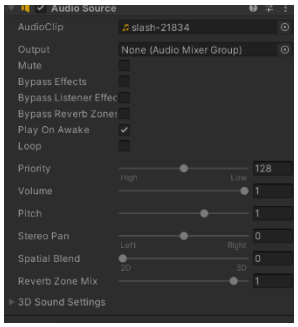


Figure 24: The sound effect for an enemy attack is played through an audio source

Collision

A new map has been created, featuring 2D collision tiles in Unity. These tiles define the playable area by restricting movement and interactions based on terrain layout. The collision system ensures that the player and enemies navigate the environment realistically, preventing unintended movement through obstacles while maintaining smooth pathfinding and gameplay flow.

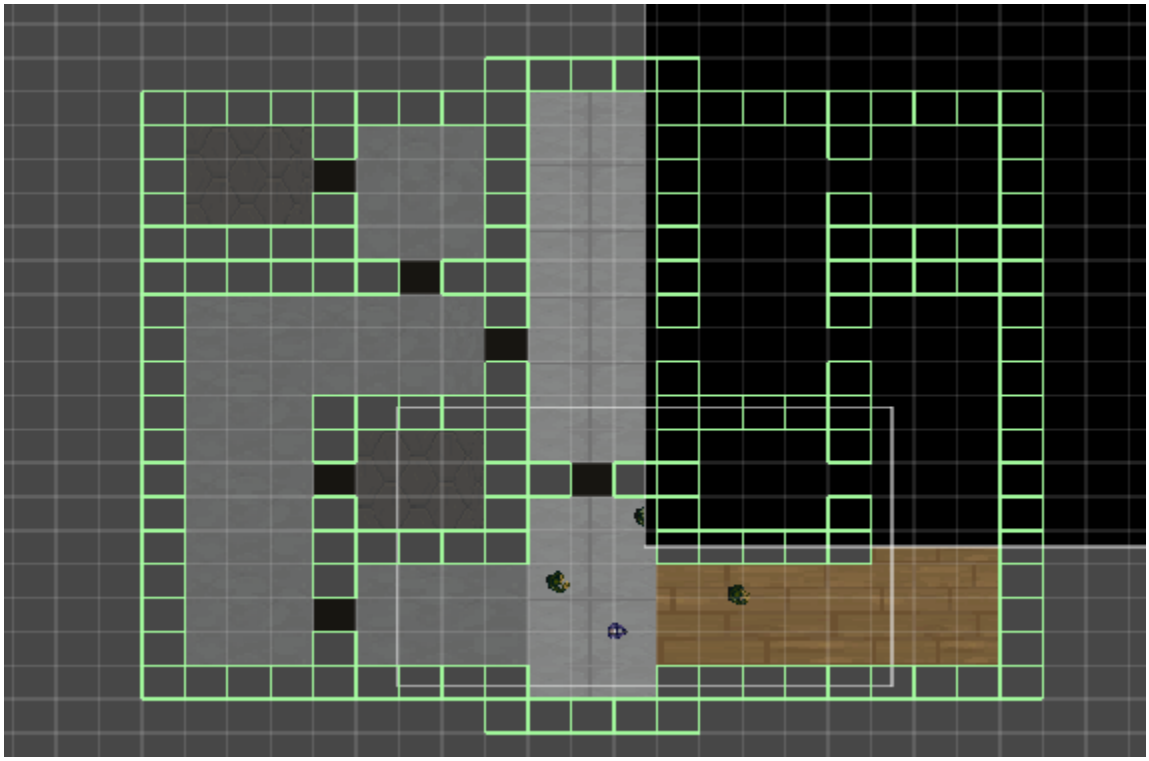


Figure 25: A map with the collision highlighted to display the boundaries of the map the player is held in

Improved Pausing

The pause menu is now integrated within the same scene as the game, rather than transitioning to a separate scene. This improvement allows for a seamless pause-and-resume experience, ensuring that the game state remains unchanged while paused. All gameplay elements, including movement, animations, and AI behavior, freeze upon activation, and resume smoothly when the menu is closed.

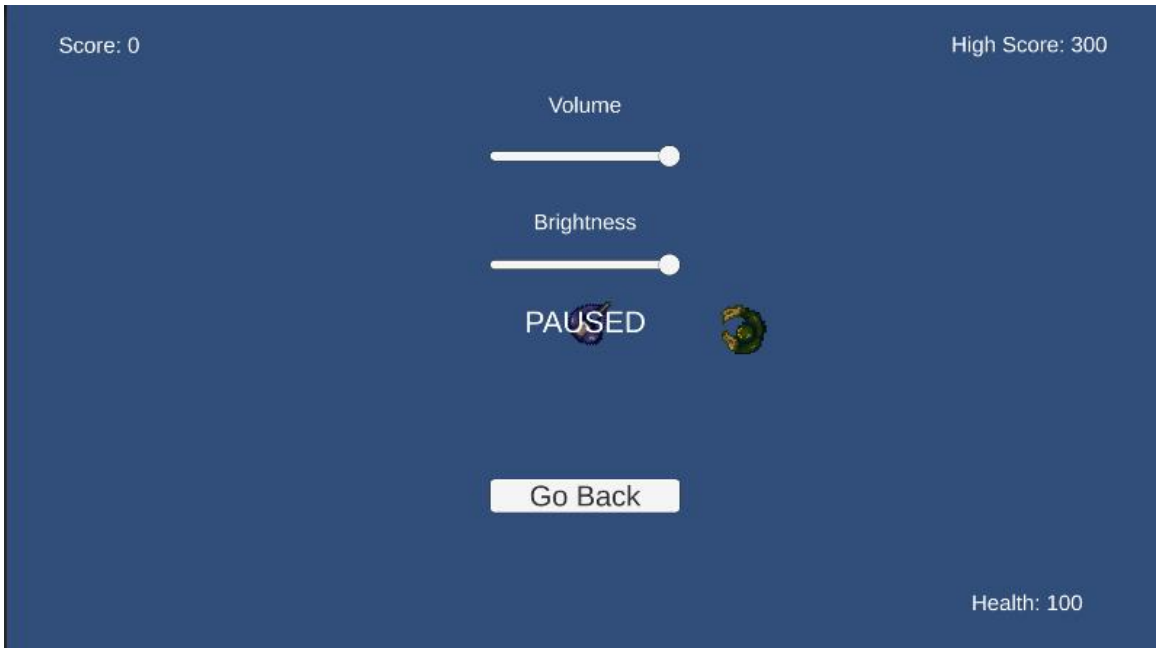


Figure 26: The pause menu now works without having to go to a separate scene

Weapon Switch System

The player can now cycle between multiple weapons, each featuring unique attributes that impact gameplay. Weapons may differ in fire rate, ammo capacity, and damage output, allowing for varied combat strategies. Additionally, different trigger types are implemented, including automatic and semi-automatic firing modes, influencing how each weapon is used in battle. Some weapons also feature specialized mechanics, such as multi-pellet shots or alternative firing behaviors, further diversifying the player's arsenal and tactical options.

Figure 27: Here is the player holding a pistol

Figure 28: Here is the player holding a rifle

In addition, a new weapon, the shotgun, was added. The shotgun fires a spread of five projectiles, each traveling in a slightly randomized direction within a defined spread angle. This mechanic simulates the natural dispersion of a shotgun blast, making it highly effective at close range while reducing accuracy at longer distances. The randomness in pellet trajectories ensures variability in each shot, adding depth to the weapon’s functionality and requiring players to consider positioning and range when using it.

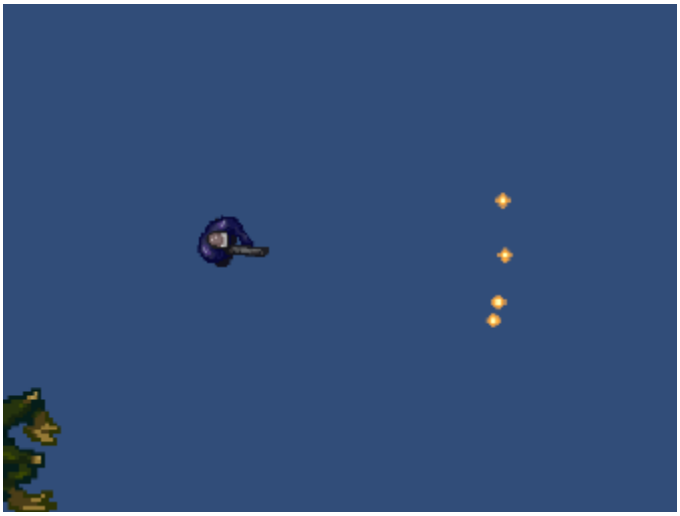


Figure 29: A demonstration of the shotgun being used

Enemy Spawn Logic

Enemy spawn points have been implemented, ensuring that enemies appear at designated locations throughout the map. To prevent unfair or immersion-breaking spawns, a proximity and visibility check has been added—enemies will not spawn if their designated spawn point is too close to the player or within the player's line of sight. This system encourages balanced gameplay by maintaining challenge while preventing enemies from suddenly appearing in unrealistic or exploitable locations.

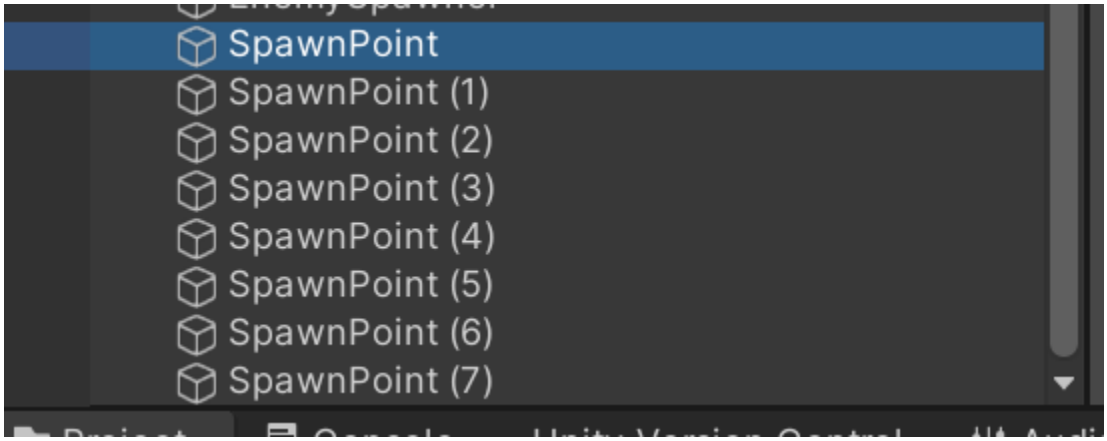


Figure 30: Various spawn points implemented in a single level

Melee Attacks Adjusted

- Implemented a melee attack hitbox that remains disabled by default to prevent unintended collisions outside attack animations.
- Utilized animation events to dynamically toggle the hitbox state, enabling it during active attack frames and disabling it immediately after to ensure precise hit detection.
- Resolved hit detection issues caused by OnTriggerEnter inconsistencies by repositioning the hitbox behind the player when inactive and moving it into the correct position only when activated, ensuring reliable enemy collisions.

```
// Called on the attacking frames of Enemy Attack animation
public void StartAttack() {
    // Toggle Enemy Attack Hitbox on
    EnemyHitbox.SetActive(true);

    // move hitbox in positon
    EnemyHitbox.transform.localPosition = new Vector3(1f, 0, 0);
}

// Called on the frame after attacking frames
public void EndAttack() {
    // Toggle Enemy Attack Hitbox off
    EnemyHitbox.SetActive(false);

    // move hitbox out of positon
    EnemyHitbox.transform.localPosition = new Vector3(-1f, 0, 0);
}
```

Figure 31: Melee attack code snippet to demonstrate how the enemy will attack

Item Pick-Up/Drop System

An enemy drop system has been implemented, allowing defeated enemies to spawn collectible items such as health packs and ammo packs upon death. The health pack functionality is fully integrated, enabling the player to restore health when collected. While ammo drops are generated correctly, the logic for replenishing the player's ammunition has not yet been implemented, but the system is designed to support this functionality in future updates.

```
if (other.CompareTag("HealthPack")) {
    Debug.Log("Player picked up Health Pack");
    HealDamage(25f);
    Destroy(other.gameObject);
}
if (other.CompareTag("AmmoPack")) {
    Debug.Log("Player picked up Ammo Pack");
    // logic to add ammo to reserves goes here
    Destroy(other.gameObject);
}
```

Figure 32: The health pack is functional, the ammo pack has yet to be fully implemented

New Enemies

Two additional melee enemy variants have been introduced, each with distinct attributes compared to the default enemy (enemy1). The Big & Slow enemy (enemy2) features a larger hitbox, increased health, reduced movement speed, and higher attack damage, making it a tank-like threat. The Small & Fast enemy (enemy3) has a smaller hitbox, lower health, increased movement speed, and an expanded detection radius, allowing it to engage the player more aggressively. Both variants currently use the same sprites and animations as the default enemy, though adjustments may be made in the future. Additionally, potential item drop rate modifications are being considered, with enemy2 favoring health drops and enemy3 favoring ammo drops.

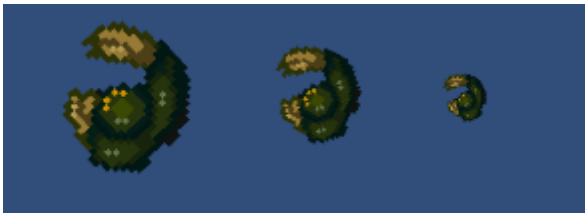


Figure 33: The enemies although identical have different properties to their stats

A* Pathfinding

Enemies now utilize *A Pathfinding** to effectively track and pursue the player. A NavMesh is defined to specify the walkable areas, ensuring that enemy AI can navigate around obstacles and dynamically adjust their path based on the player's position. As long as the player remains within the same continuous NavMesh region as an enemy, the AI will calculate and follow an optimal path to close the distance. This implementation enhances enemy behavior by allowing them to navigate complex environments efficiently. The NavMeshPlus library, available on GitHub (NavMeshPlus), is used to extend Unity's pathfinding capabilities, providing more flexibility for 2D navigation.

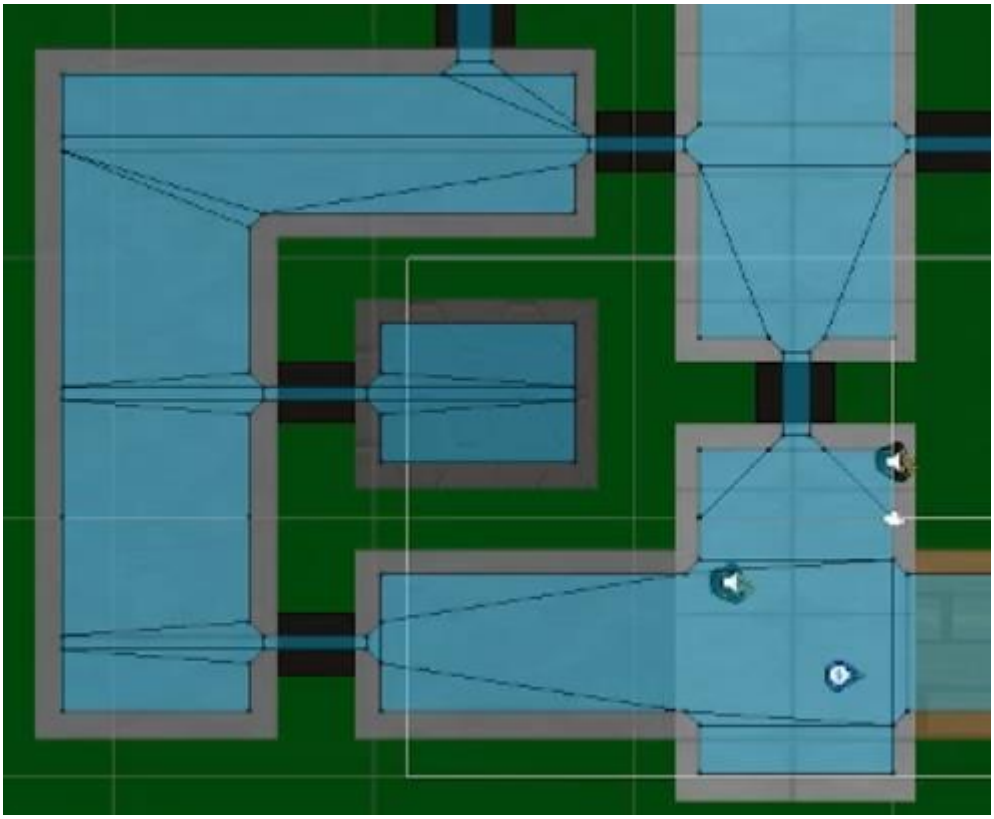


Figure 34: The light-blue region illustrates the walkable region for a NavMesh agent.

Sprint 7: Midpoint of Development

Creation of 2 New Levels / Looping

Complete **prototypes** for **Level 2 and Level 3** have been implemented, featuring fully functional **collision mechanics**, a defined **NavMesh** for enemy pathfinding, and strategically placed **enemy spawn**

points. Each level has been designed with unique layouts to challenge the player while ensuring smooth navigation for AI-controlled enemies.

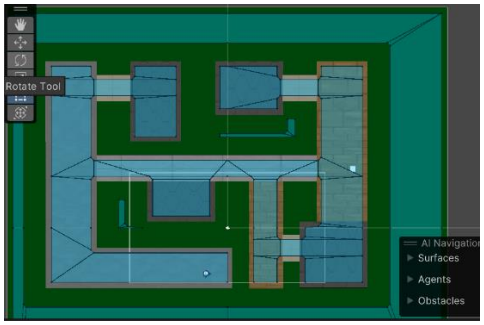


Figure 35: The design of level 2

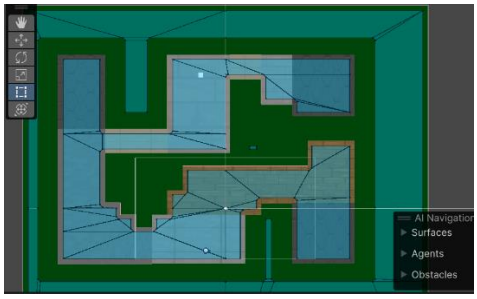
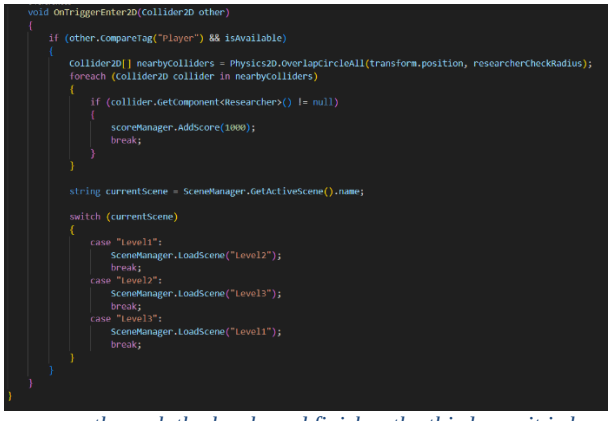


Figure 36: The design of level 3

Progression through the game follows a cyclical level structure. The player advances from Level 1 to Level 2, then to Level 3, and upon completion, the cycle restarts back at Level 1. To clear a level, the player must defeat all enemies, at which point the goal area becomes accessible, allowing them to proceed. This looped progression system maintains continuous gameplay while increasing the challenge as the player progresses through multiple cycles.



```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player") && isAvailable)
    {
        Collider2D[] nearbyColliders = Physics2D.OverlapCircleAll(transform.position, researcherCheckRadius);
        foreach (Collider2D collider in nearbyColliders)
        {
            if (collider.GetComponent<Researcher>() != null)
            {
                scoreManager.AddScore(1000);
                break;
            }
        }

        string currentScene = SceneManager.GetActiveScene().name;

        switch (currentScene)
        {
            case "Level1":
                SceneManager.LoadScene("Level2");
                break;
            case "Level2":
                SceneManager.LoadScene("Level3");
                break;
            case "Level3":
                SceneManager.LoadScene("Level1");
                break;
        }
    }
}
```

Figure 37: When the player goes through the levels and finishes the third one, it is looped back to the first level

Game Over & Leaderboard Screen

When the player dies, they are redirected to a Game Over Scene, where their performance is evaluated. The scene displays the top 5 highest scores of all time and updates the list dynamically if the player surpasses any of the current top scores during their playthrough. This system ensures that players are incentivized to improve their performance, and their progress is recorded across different sessions, providing a competitive aspect to the gameplay experience.

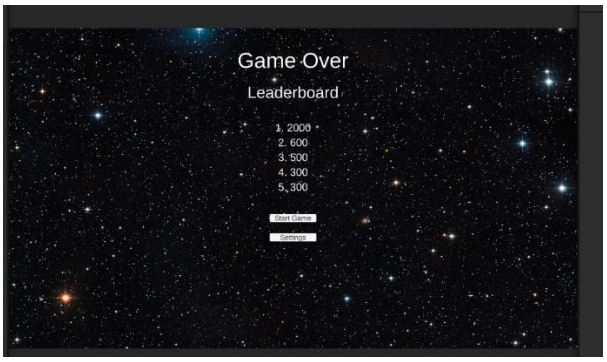


Figure 38: The game over screen displaying the leaderboard which consists of the top 5 scores

Basic Difficulty Scaling

Each level progressively increases in difficulty, ensuring a steady challenge curve. Level 1 serves as the easiest stage, introducing the player to combat mechanics with only two basic enemy types. As the player progresses to Level 2 and Level 3, additional enemy types are introduced, and the enemy count is increased, creating more complex encounters. This gradual scaling of difficulty enhances player engagement and encourages skill development while maintaining balanced gameplay.

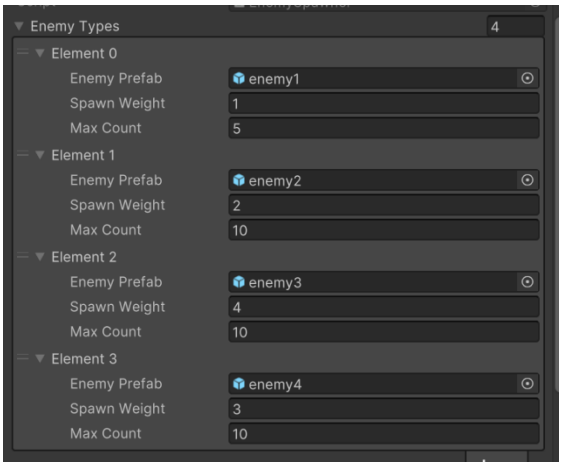


Figure 39: Multiple enemies are able to be spawned in and have a cap at their spawn and spawn intervals

Weapon Improvements

Weapons now feature limited ammo, requiring players to manage their resources carefully. Ammo is restored through pickups dropped by defeated enemies, adding an additional layer of strategy to gameplay. Additionally, bullet projectiles are now destroyed upon contacting walls, ensuring realistic interactions with the environment.

To better manage the player's arsenal, all weapons are stored in a dictionary. This system will facilitate the future implementation of JSON-based save files, where weapons will be referenced by a unique key, allowing the player's inventory to be restored from a previous session. This structure sets up seamless save/load functionality for weapon data in upcoming updates.

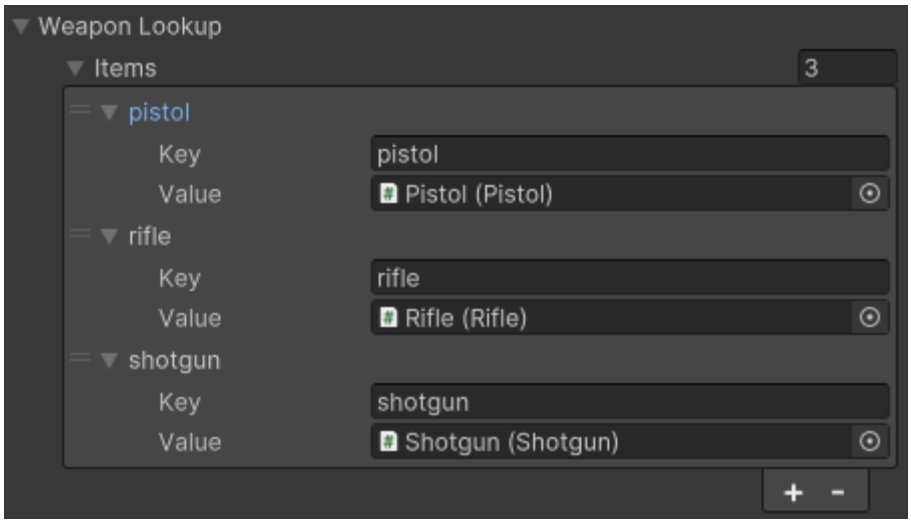


Figure 40: The weapons are able to be cycled and switched between easily in-game

Researcher added

The Researcher NPC uses NavMesh-based pathfinding for dynamic navigation, aligning with the NavMesh via agent.Warp(). It selects a random spawn point and follows the player when within followDistance, stopping at stopDistance. If near the goal (goalSwitchDistance), it transitions from player-following to goal-seeking behavior.

The NavMeshAgent updates destination vectors and rotation using Mathf.Atan2(), ensuring smooth navigation. Animation states are controlled dynamically, triggering walking or idle animations based on movement. Upon enemy impact, the NPC disables collision, enters a death state, and plays a death animation, with Destroy(gameObject, 5f) handling cleanup. This system efficiently integrates AI navigation, animation, and collision-based events.

Figure 41: Sprite of the researcher

Grenade functionality

The Grenade class manages a throwable explosive with a timed detonation system. It tracks its state (inFlight or landed) and uses Invoke("explode", fuseLength) to trigger an area-of-effect explosion after landing. Upon detonation, Physics2D.OverlapCircleNonAlloc() detects entities within the explosion radius, filtering them via HitLayer. Damage is applied only if the line-of-sight is clear, using Physics2D.Raycast() against CoverLayer to check for obstacles. Enemies (EnemyPathfinding2D) and players (PlayerMovement) take 80 damage if exposed. A visual explosion effect is instantiated, and the grenade self-destructs post-detonation (Destroy(gameObject)). The system ensures efficient physics-based collision checks and realistic explosion behavior.

```
    //Get all entities in range
    int numHits = Physics2D.OverlapCircleNonAlloc(origin, explosionRadius, Hits, HitLayer);

    //Check each entity in range to determine exposure to explosion; deal damage
    for (int i = 0; i < numHits; i++)
    {
        GameObject hitGO = Hits[i].gameObject;
        Debug.Log(hitGO);
        if (Hits[i].TryGetComponent<PlayerMovement>(out PlayerMovement player))
        {
            float distance = Vector3.Distance(origin, Hits[i].transform.position);

            if (!Physics2D.Raycast(origin, (Hits[i].transform.position - origin).normalized, explosionRadius, CoverLayer.value))
            {
                player.TakeDamage(80f);
            }
        }
        else if (Hits[i].TryGetComponent<EnemyPathfinding2D>(out EnemyPathfinding2D enemy))
        {
            float distance = Vector3.Distance(origin, Hits[i].transform.position);

            if (!Physics2D.Raycast(origin, (Hits[i].transform.position - origin).normalized, explosionRadius, CoverLayer.value))
            {
                enemy.TakeDamage(80f);
            }
        }
    }

    //Spawn Explosion Particle
    Instantiate<GameObject>(explosionParticle, transform.position, transform.rotation); //No need to store gameobject in variable, it's just a particle

    //Grenade has exploded -- destroy it
    Destroy(this.gameObject);
}
```

Figure 42: Code snippet of how the grenade works

Sprint 8: Making Adjustments

Tutorial

To ensure players understand the controls before starting, we implemented a clear and intuitive layout for both keyboard and controller inputs. This layout provides a visual reference, allowing players to quickly grasp movement, attacks, and interactions based on their chosen input device. By integrating this directly into the game, we enhance accessibility and reduce the learning curve, ensuring a seamless onboarding experience.

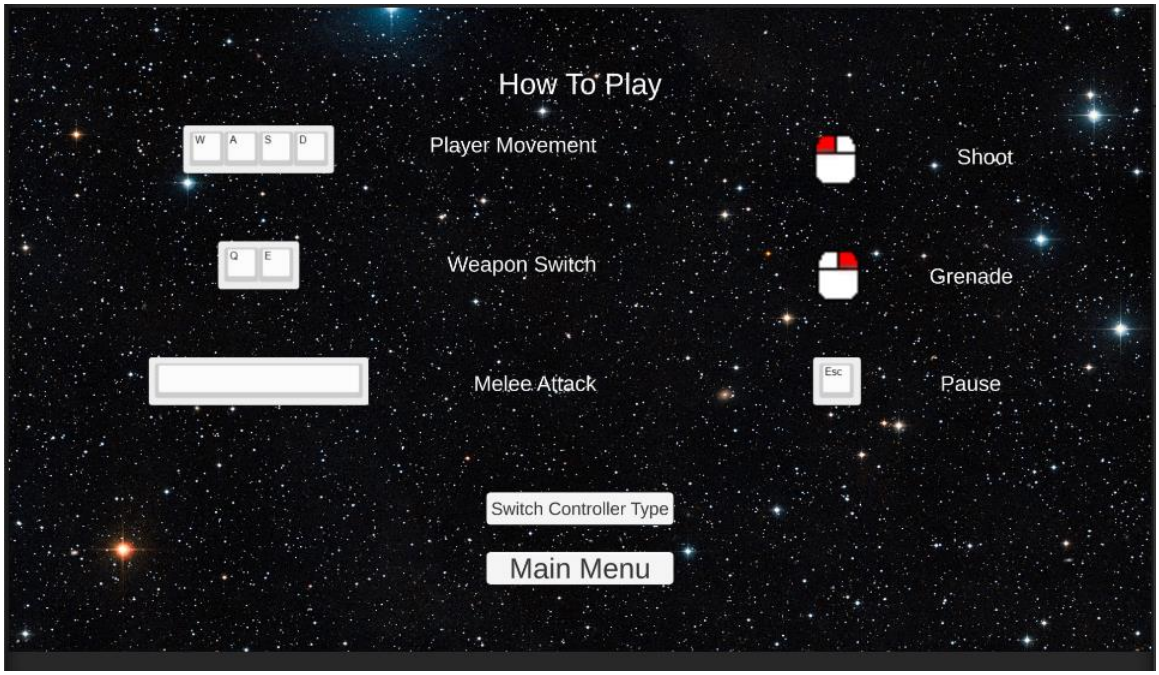


Figure 43: Keyboard layout of the controls

Figure 44: Gamepad layout (Xbox) of the controls

Fixing Enemy Bullets

Ranged enemy projectiles could previously pass through walls due to a lack of proper collision handling. This was resolved by attaching a Rigidbody2D (with zero gravity scale) to the fireball prefab, ensuring it interacts correctly with physics. Additionally, collision detection was implemented with the wall layer, preventing projectiles from bypassing solid obstacles and improving gameplay consistency.

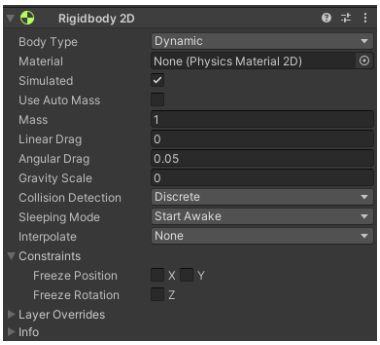


Figure 45: 2D Rigidbody component of the bullet prefab

Persisting Data

The player and HUD are now persistent across scene transitions. Instead of relying on direct scene-specific object references, singleton patterns are used to manage references (e.g., displaying the score or setting pathfinding targets). Additionally, the score manager is now a child of the player prefab, ensuring that the player's score persists when transitioning between levels.

```
public class PlayerManager : MonoBehaviour
{
    //Singleton for player character (PC). Ensures that only one PC exists, and that
    //Camera, Researchers, etc. only ever reference the existing PC.
    //(Should be automatically set to null if stored singleton is destroyed)
    public static PlayerManager playerInstance;

    // Start is called before the first frame update
    void Start()
    {
        if (playerInstance == null)
        {
            playerInstance = this;
            DontDestroyOnLoad(this);
        }
        else if (playerInstance != this)
        {
            //Move existing PC to this position, then delete this duplicate
            playerInstance.transform.position = this.transform.position;
            Destroy(this.gameObject);
        }
    }

    // OnDestroy, clear the singleton
    void OnDestroy()
    {
        if (playerInstance == this)
        {
            playerInstance = null;
        }
    }
```

Figure 46: Code snippet of the player manager only having one instance

Weapon Damage & Power-Ups

Weapons now have assigned damage values, which are transferred to their bullets upon instantiation. When a bullet collides with an enemy, it reduces the enemy's health based on its assigned damage value rather than a fixed, hard-coded amount.

In addition, user feedback from last semester's digital playtest showed we needed a way to make the player move faster. The newest power-up is the speed pack, which allows the player to move faster for a short duration.

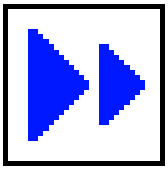


Figure 47: Sprite of the speed power up

Sprint 9: Custom UI & Procedural Generation

Additional Sprites & Logo

Additional sprites were designed to enhance the game's artistic style, including visuals for health, ammo, and trophies representing the top 3 scores. These graphical elements replace plain text, making the UI more visually appealing and immersive.

Figure 48: Three trophies are now displayed next to the top 3 scores

Figure 49: The HUD that has been implemented in game

A logo was created for our game to enhance its visual identity, replacing the plain text title with a more engaging and professional design.

Figure 50: The logo appearing at the start of the game

Fixing the Pause & Settings Menu

The **pause functionality**, which previously did not work, has been fixed. Additionally, the **settings menu** on the start screen now correctly returns to the main menu when exited.

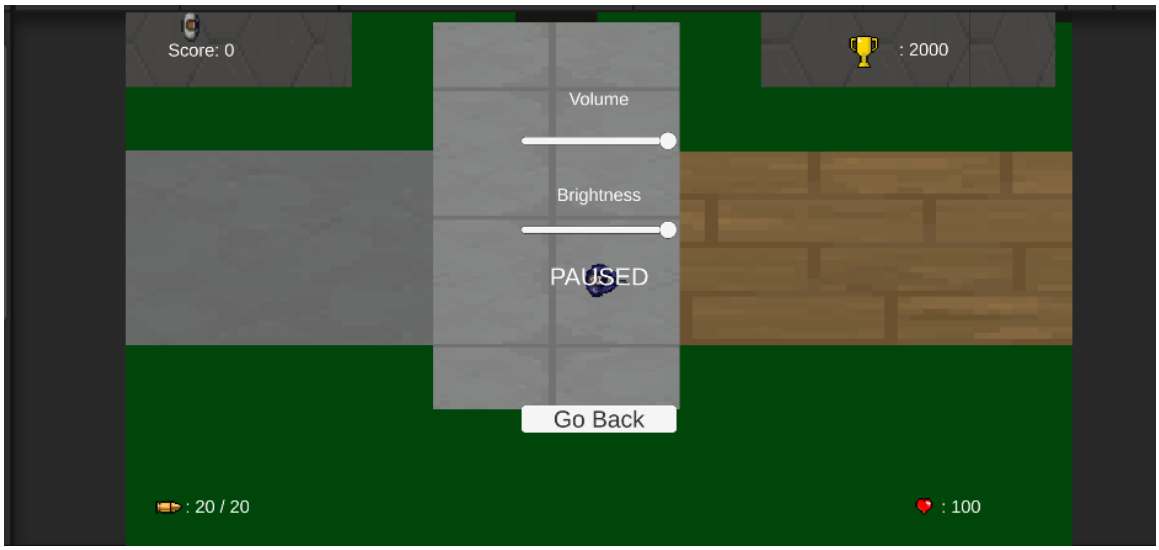


Figure 51: Pause menu working within the game scene

Save File System

Work has begun on a file system to allow players to restore data from a previous session. While the save feature has not been implemented yet, the LoadPlayer script can already read data from a JSON file. This allows the player to start with a desired health value, level, and a customizable inventory of weapons.

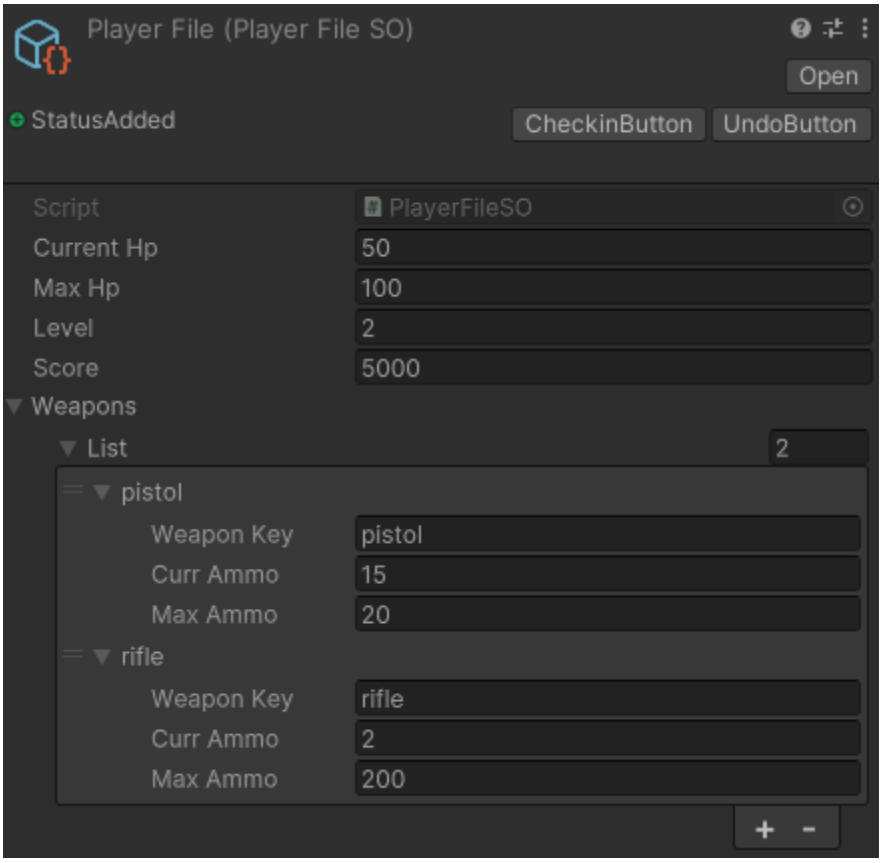


Figure 52: Implementation of saving the values of the player

Procedural Generation Research & BSP Early Implementation

After doing some research on different algorithms to procedurally generate endless levels, we found that the Binary Space Partitioning (BSP) would suit our needs best. This algorithm splits defined space into many sub-spaces that each can be used for a room, ensuring that there will be no overlap between rooms.

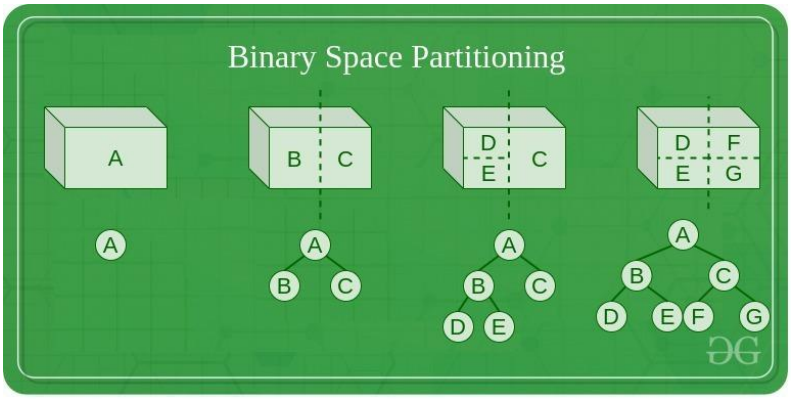


Figure 53: Visual of how Binary Space Partitioning works

A script has been created implementing the BSP (Binary Space Partitioning) algorithm to generate and connect rooms. Gizmos are being used to temporarily define the rooms, allowing for visualization. The process of integrating the generated rooms with the game assets is still being worked out.

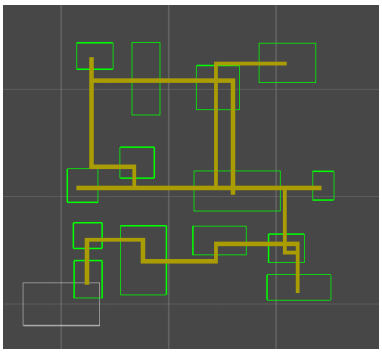


Figure 54: Early implementation of how BSP works

Sprint 10: Continued Development of UI and BSP

Custom Buttons & Text

Custom UI buttons and text have been created to replace some of the default Unity UI elements. This is done to enhance the visual style and consistency across the game's interface.

Figure 55: New custom buttons and background are used for the start screen

Key bind System

All player inputs are now managed by a dedicated "KeybindManager" script, allowing for easier customization. The script handles inputs for WASD movement, shooting, throwing grenades, punching,

and pausing. Additionally, keybindings can be re-assigned through a new tab in the settings scene, which is accessible from the title screen, providing players with flexibility in their controls.

Figure 56: Key binding feature to have customizable controls

Progress on BSP Level Generation

The HUD and other core systems have been added and are now present at every level. Additionally, the room spawners have been updated to populate rooms dynamically. The system works as follows:

- First room: Contains the player.
- Last room: Contains the goal.
- Random rooms: Contain the researcher.
- Every other room: Contains enemies.

This ensures that each level is procedurally generated with distinct elements, creating a varied gameplay experience.

```
1 reference
void PlaceSpawnPoints()
{
    if (rooms.Count == 0) return;

    Vector2 playerPos = rooms[0].center;
    Instantiate(playerSpawnPrefab, new Vector3(playerPos.x, playerPos.y, 0), Quaternion.identity);

    Vector2 goalPos = rooms[rooms.Count - 1].center;
    Instantiate(goalSpawnPrefab, new Vector3(goalPos.x, goalPos.y, 0), Quaternion.identity);

    int researcherRoomIndex = Random.Range(1, rooms.Count - 1);
    Vector2 researcherPos = rooms[researcherRoomIndex].center;
    Instantiate(researcherSpawnPrefab, new Vector3(researcherPos.x, researcherPos.y, 0), Quaternion.identity);

    for (int i = 1; i < rooms.Count - 1; i++)
    {
        if (i == researcherRoomIndex) continue; // Avoid placing an enemy in the researcher room

        Vector2 enemyPos = rooms[i].center;
        Instantiate(enemySpawnPrefab, new Vector3(enemyPos.x, enemyPos.y, 0), Quaternion.identity);
    }
}
```

Figure 57: Code snippet of spawn points of key elements of a level being generated