UNIVERSITÀ DEGLI STUDI FIRENZE

TDS Game: Bullet Catacomb

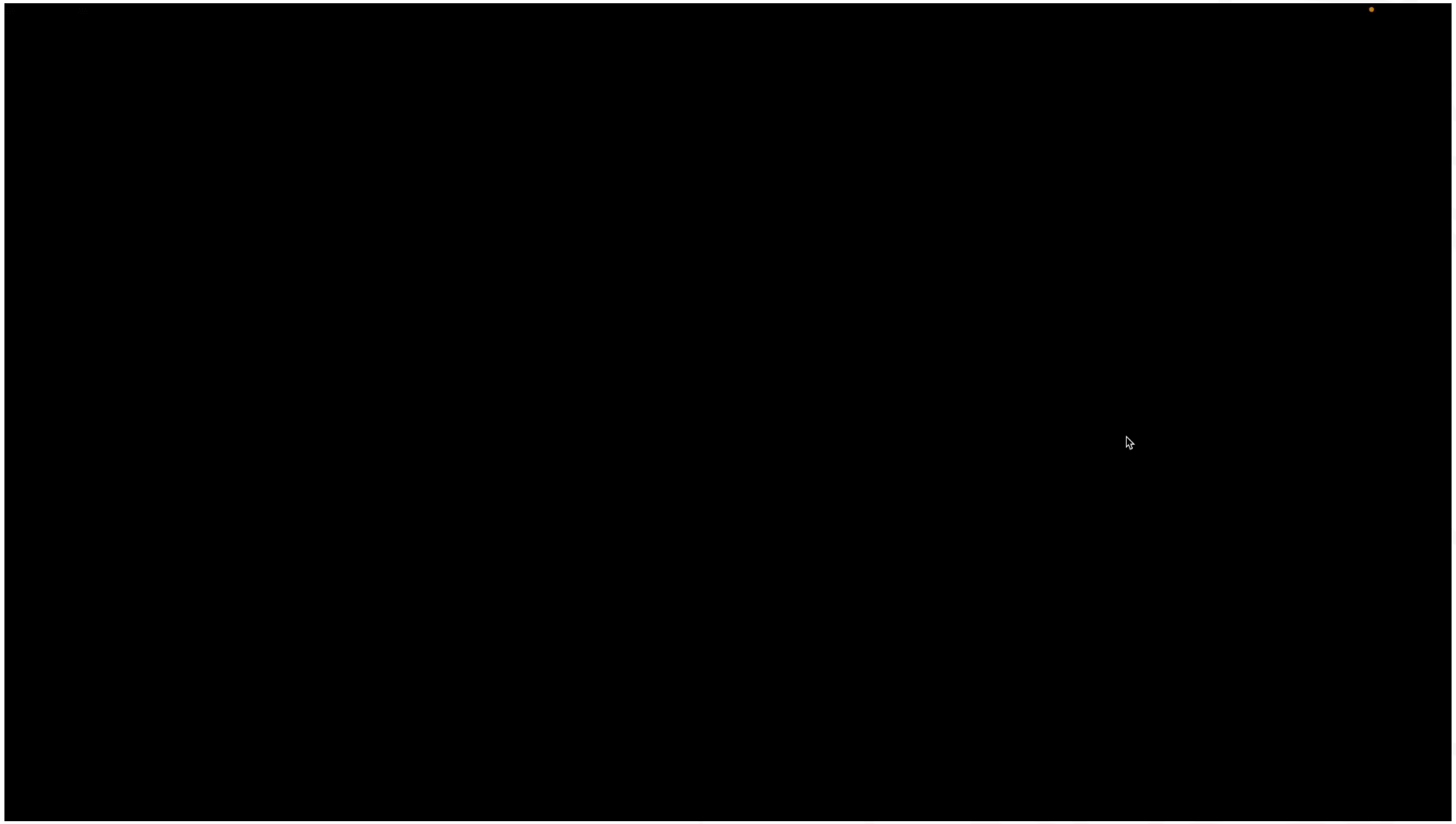
Mirko Bicchierai, Filippo Di Martino

Course: Game Development.

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduction

Showcase



Menu

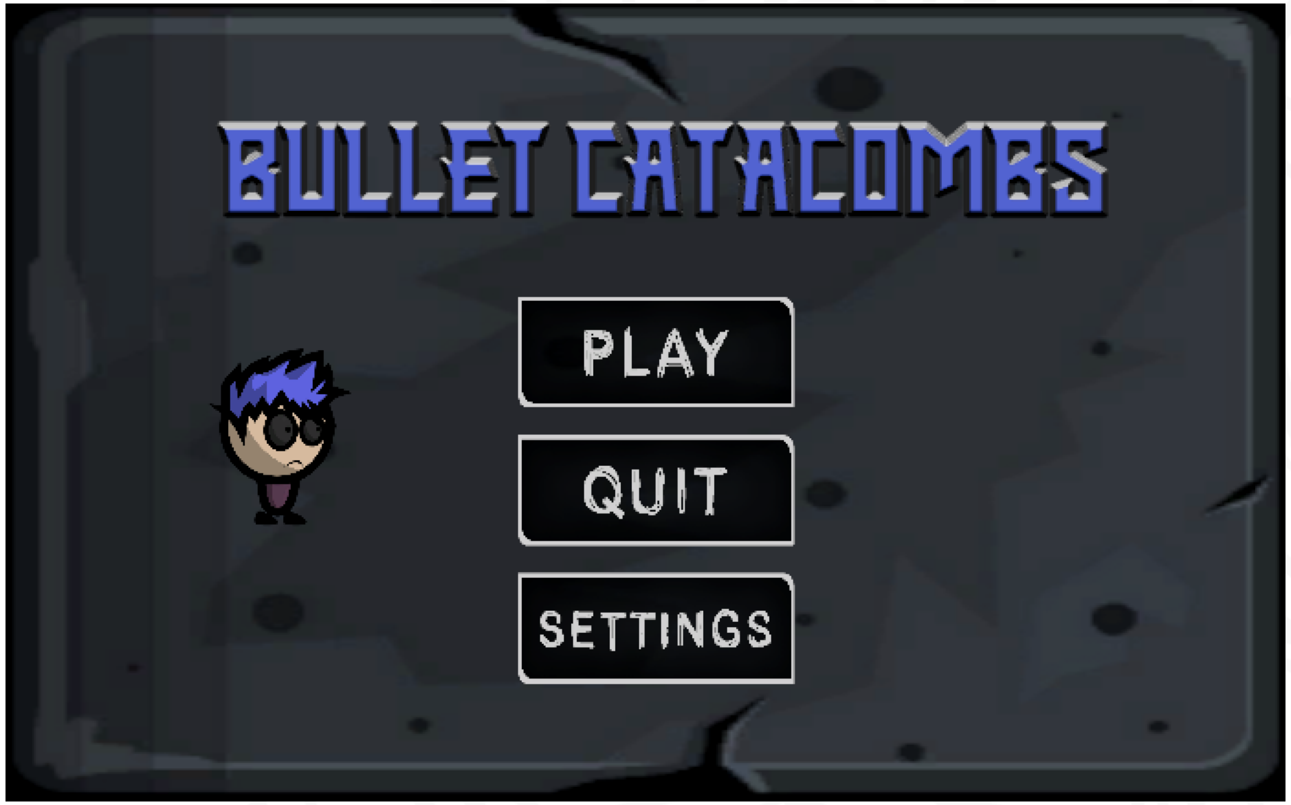
UNIVERSITÀ DEGLI STUDI FIRENZE

MainMenu Structure

The menu structure is quite simple.

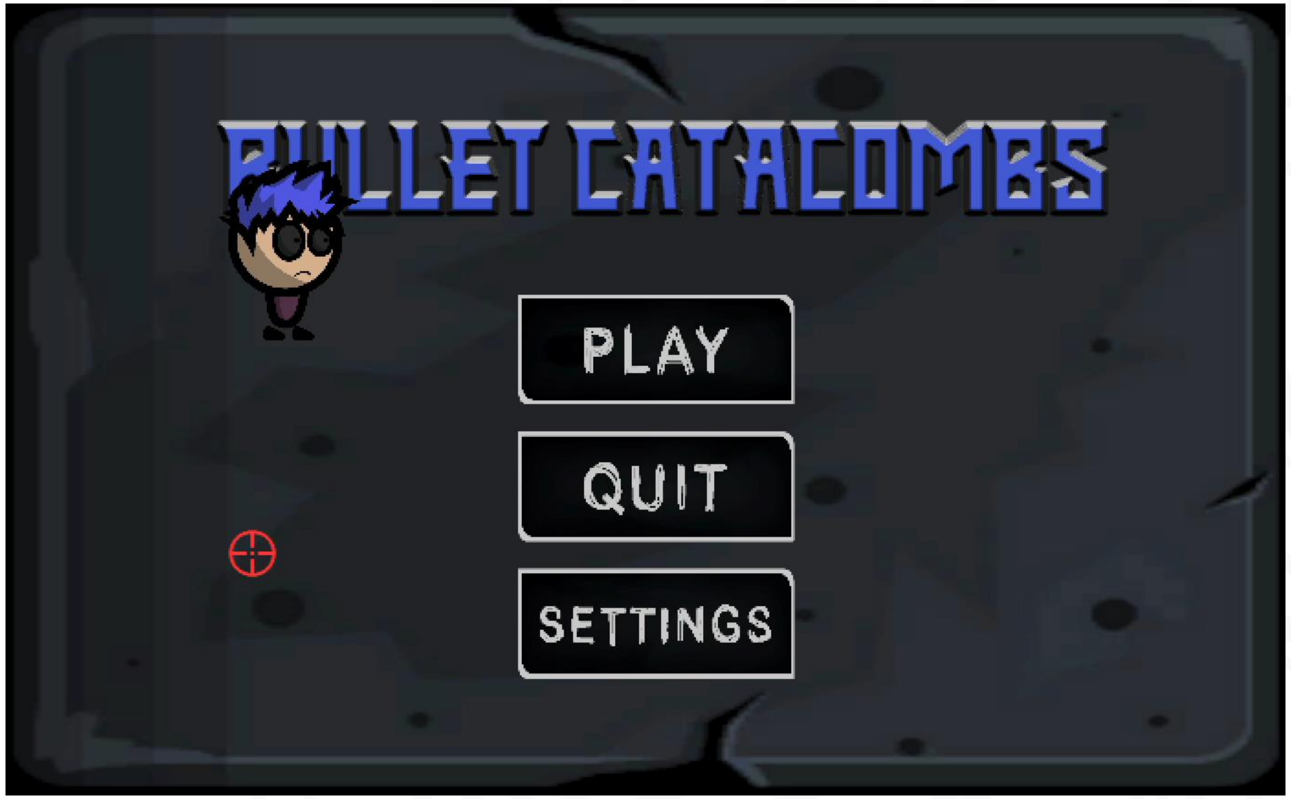
We have three classic buttons:

1. Play button
2. Quit button
3. Settings button

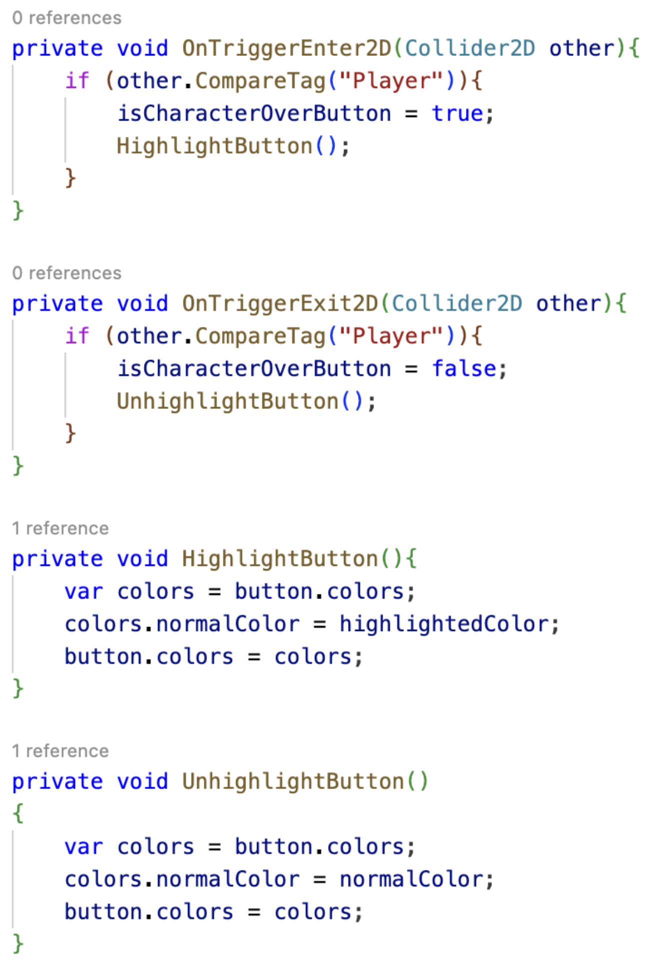


MainMenu Interactions

One unique feature is that the player can interact with the menu in two ways: either by using the mouse pointer or by moving the character over a button and pressing the spacebar to confirm the selection.



MainMenu Implementation

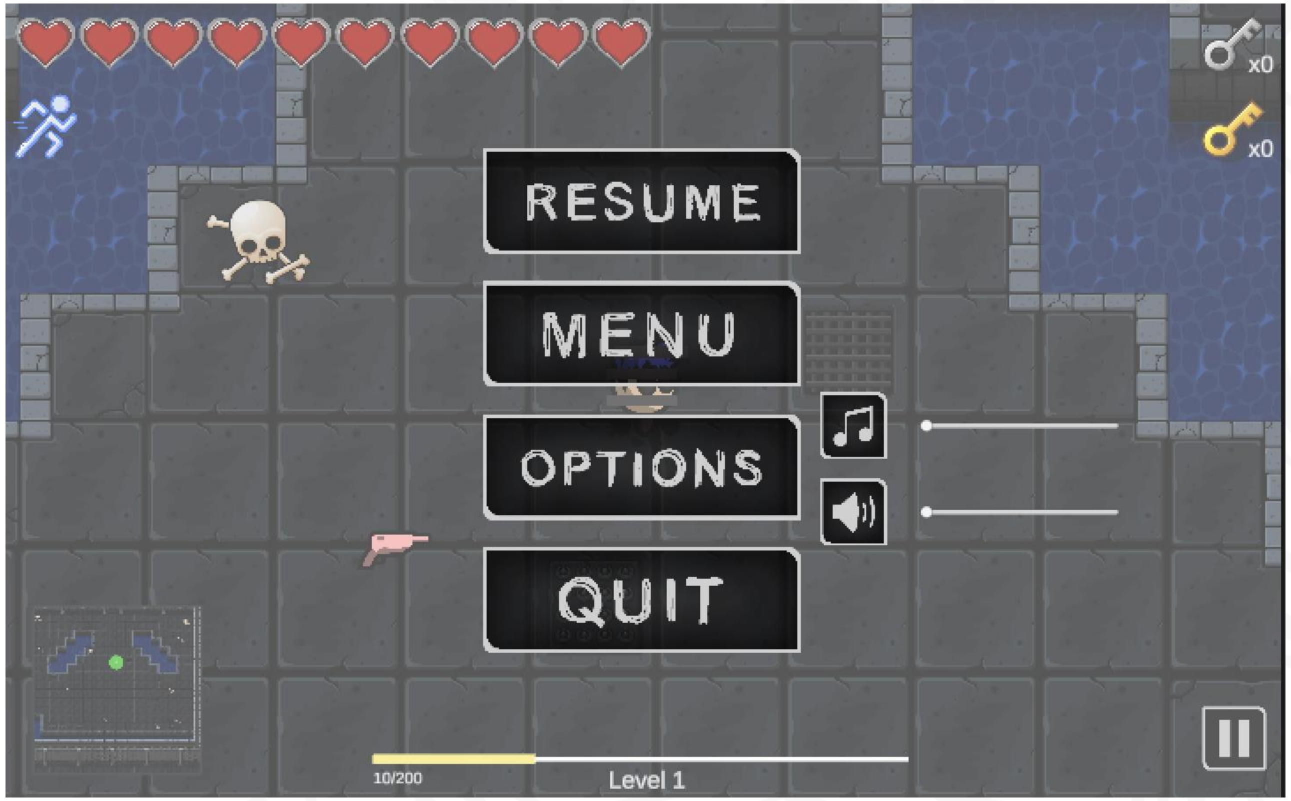


```
0 references
private void OnTriggerEnter2D(Collider2D other){
    if (other.CompareTag("Player")){
        isCharacterOverButton = true;
        HighlightButton();
    }
}

0 references
private void OnTriggerExit2D(Collider2D other){
    if (other.CompareTag("Player")){
        isCharacterOverButton = false;
        UnhighlightButton();
    }
}

1 reference
private void HighlightButton(){
    var colors = button.colors;
    colors.normalColor = highlightedColor;
    button.colors = colors;
}

1 reference
private void UnhighlightButton()
{
    var colors = button.colors;
    colors.normalColor = normalColor;
    button.colors = colors;
}
```

This feature is implemented by assigning a collider to each button, which triggers the highlighted color when the player's collider interacts with the button's collider.

PauseMenu Structure

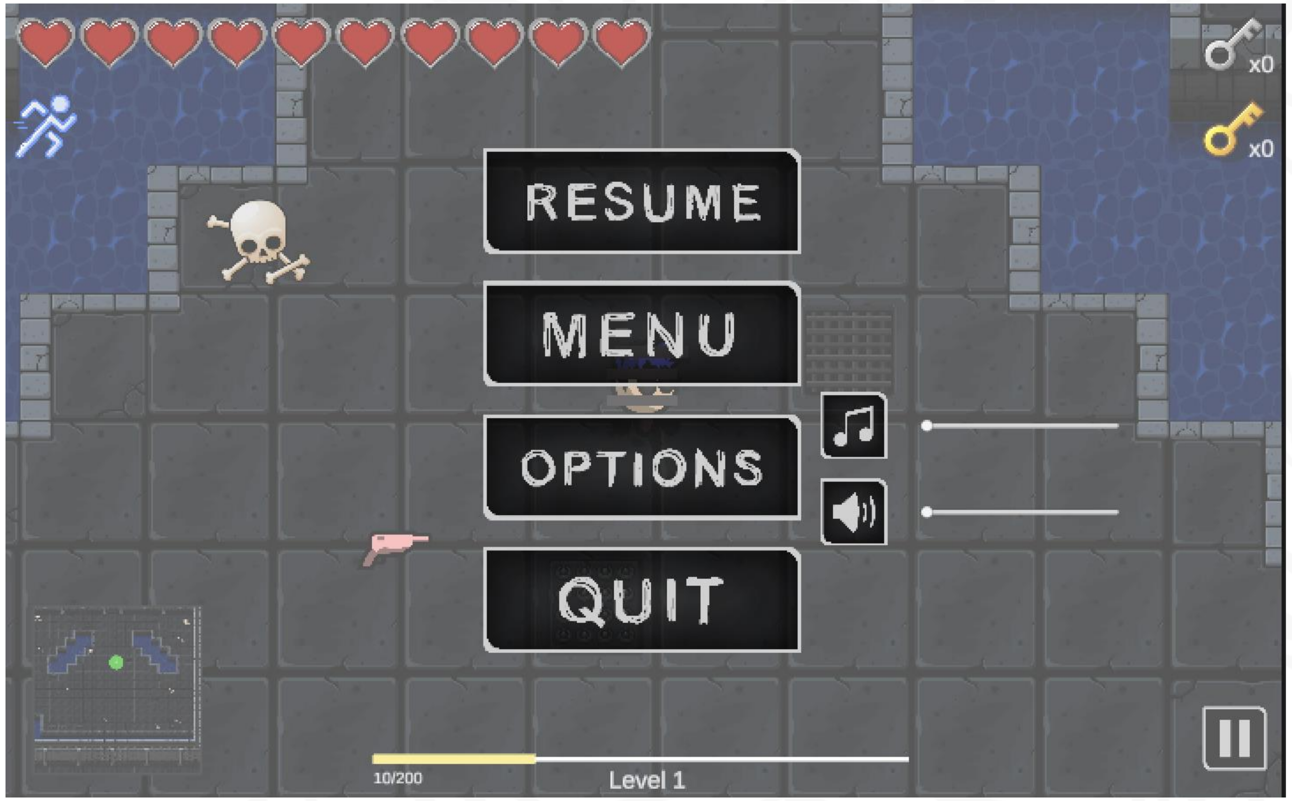
The structure is similar to the main menu, with the addition of a "Resume" button that allows the player to continue the game from where it was paused.



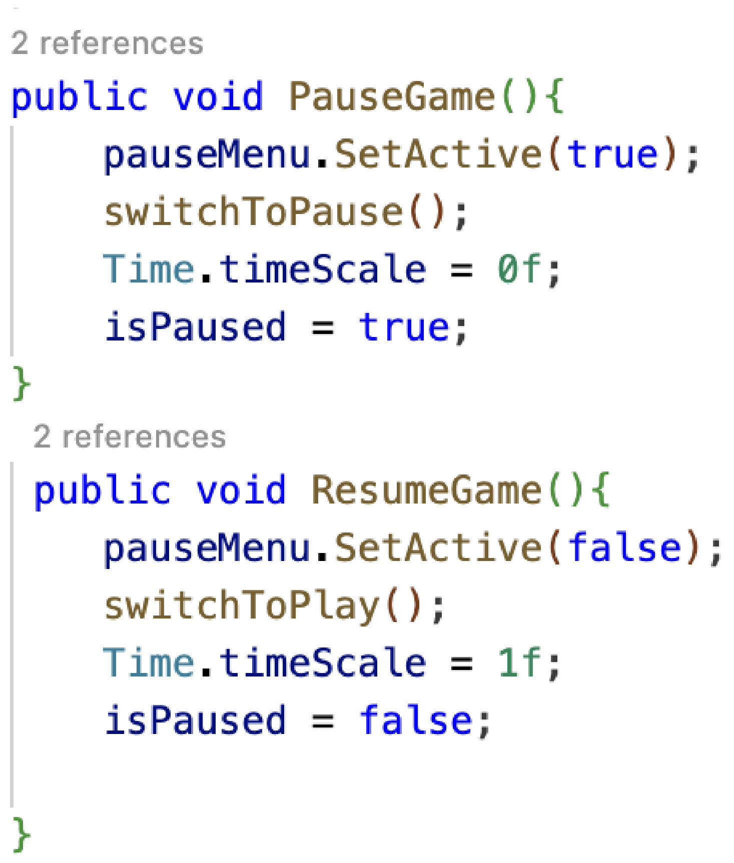
UNIVERSITÀ DEGLI STUDI FIRENZE

PauseMenu Interactions

There are two ways to access the pause menu: by clicking the pause button or by pressing the "Esc" key. Interaction with the pause menu is done using the cursor.



PauseMenu implementation

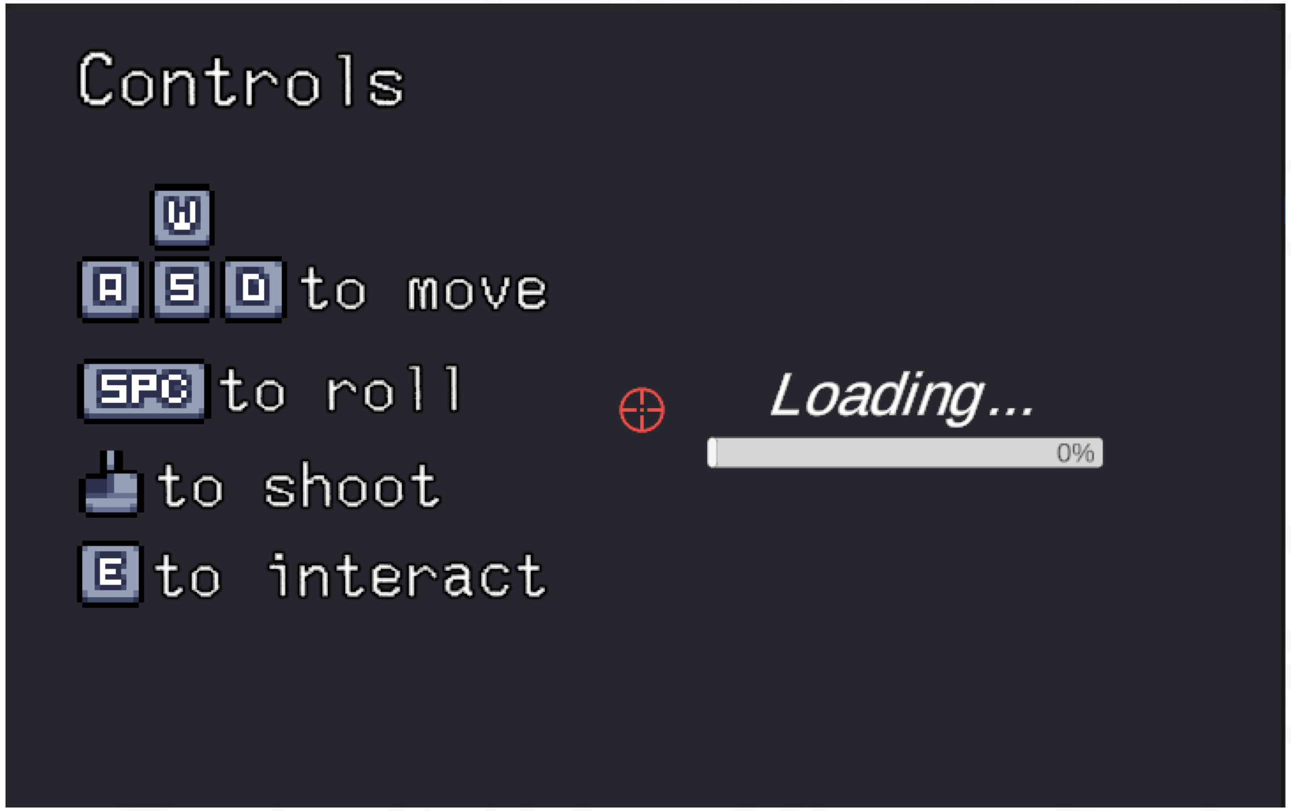


```
2 references
public void PauseGame(){
    pauseMenu.SetActive(true);
    switchToPause();
    Time.timeScale = 0f;
    isPaused = true;
}
 2 references
 public void ResumeGame(){
    pauseMenu.SetActive(false);
    switchToPlay();
    Time.timeScale = 1f;
    isPaused = false;

}
```

To achieve the "pause" effect, simply set `Time.timeScale` to zero, causing all game elements to freeze.

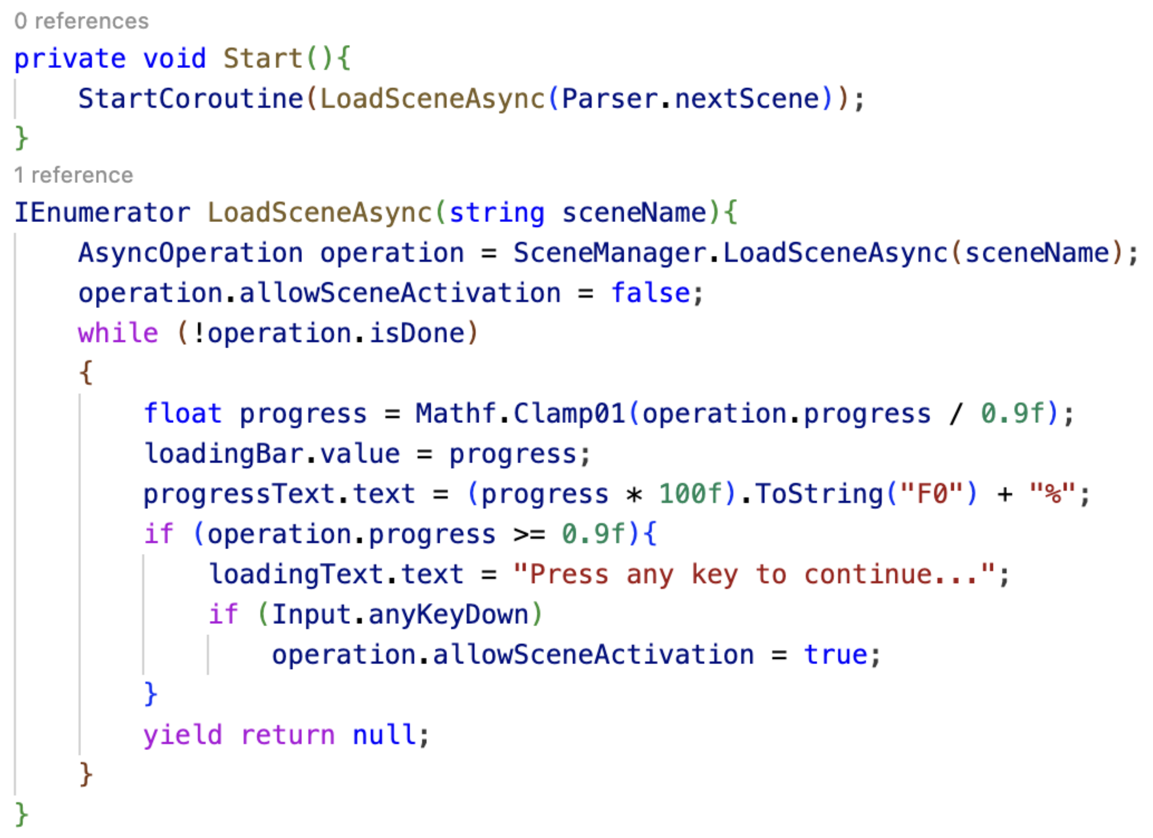
Loading Scene

The loading scene is triggered during every transition between scenes and is responsible for loading the next scene.



Loading Scene

In Unity, scene loading stops at 90% because most assets are loaded, but Unity waits for you to activate the scene. This allows time for loading screens or final tasks before transitioning to the new scene.



```
0 references
private void Start(){
    StartCoroutine(LoadSceneAsync(Parser.nextScene));
}
1 reference
IEnumerator LoadSceneAsync(string sceneName){
    AsyncOperation operation = SceneManager.LoadSceneAsync(sceneName);
    operation.allowSceneActivation = false;
    while (!operation.isDone)
    {
        float progress = Mathf.Clamp01(operation.progress / 0.9f);
        loadingBar.value = progress;
        progressText.text = (progress * 100f).ToString("F0") + "%";
        if (operation.progress >= 0.9f){
            loadingText.text = "Press any key to continue...";
            if (Input.anyKeyDown)
                operation.allowSceneActivation = true;
        }
        yield return null;
    }
}
```

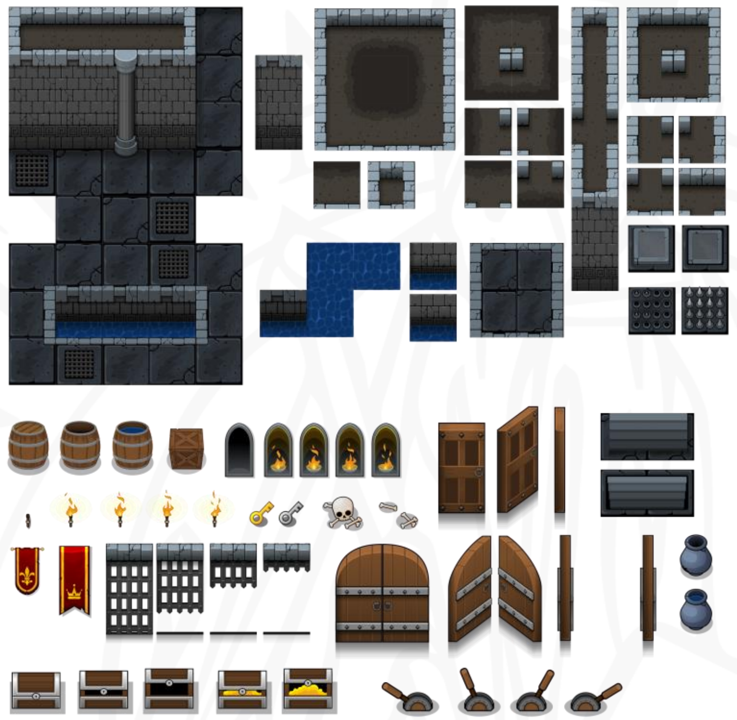
UNIVERSITÀ DEGLI STUDI FIRENZE

Graphycs

Tilemap

TileMap: Tile Palette

- The tiles used for the map are 256x256 pixels in size.
- The tiles used for decorations have variable sizes.



TileMap: Grid System

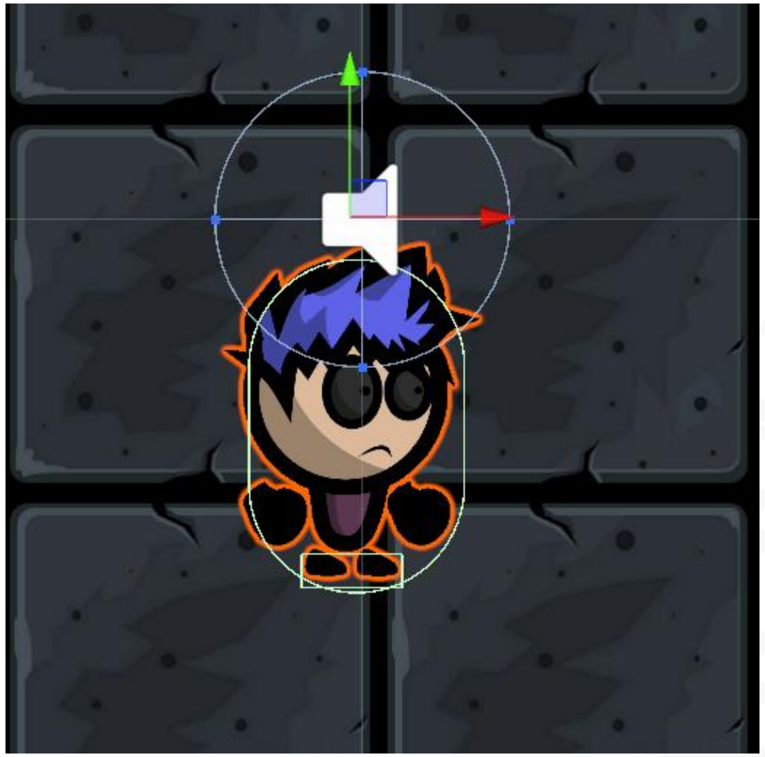
- A **four-grid system** has been implemented.
- Three of these grids are 256x256 pixels in size: one for the base **terrain**, one for the **water**, and one for the **walls** (layer order as specified, from 0 to 2).
- One of the grids is smaller in size and is dedicated to decorations, allowing for greater flexibility in positioning.
- The layers related to water and walls include additional components such as a static Rigidbody2D, a **Tilemap Collider**, and a Composite Collider. These components are used to register collisions with the player and enemies.

UNIVERSITÀ DEGLI STUDI FIRENZE

Graphycs

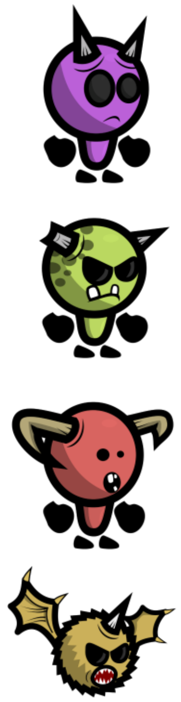
Sprites and Animation

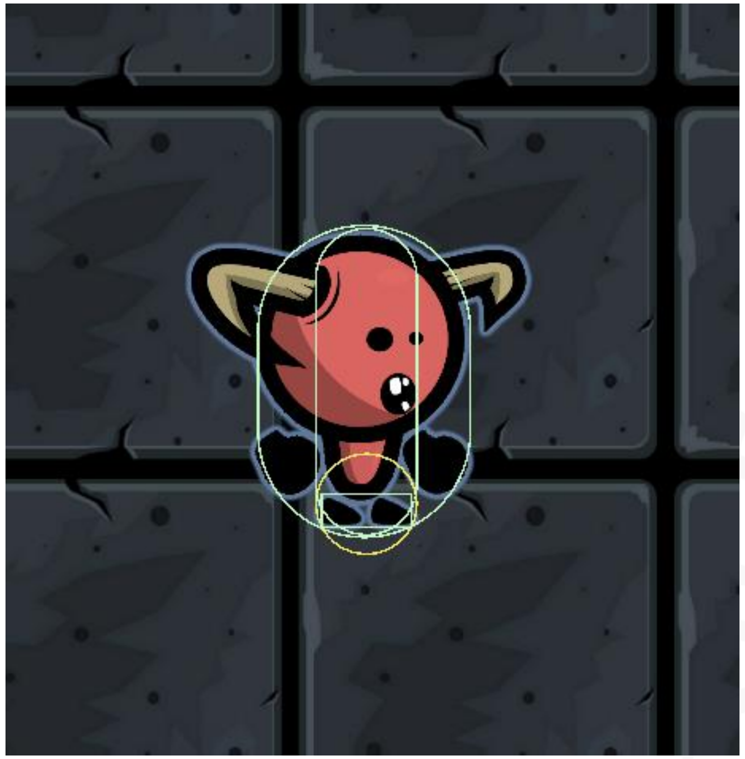
Player and Weapon



- The player is characterized by two colliders: a BoxCollider around the feet and a CapsuleCollider that covers the entire body.
- The foot collider is used to detect **collisions with non-walkable surfaces**, such as walls and water.
- The CapsuleCollider is used to register collisions with anything that can **damage** the player.

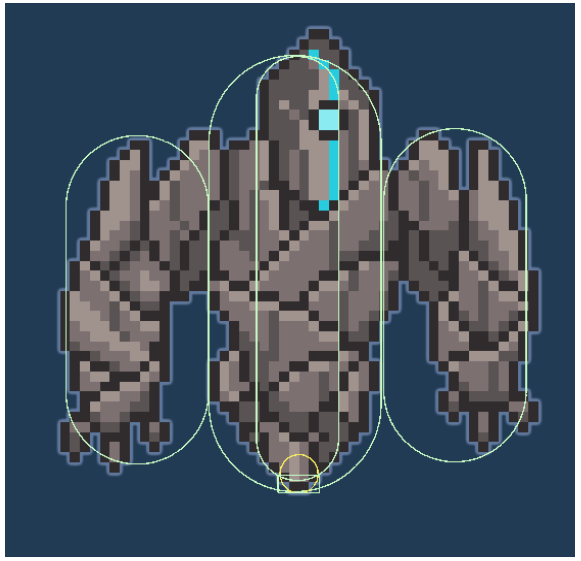
Standard Enemies

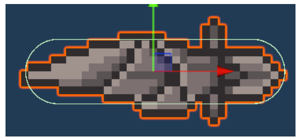


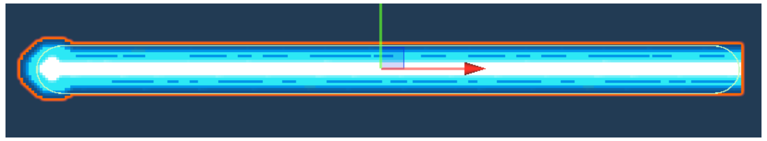


- All enemies are modeled in the same way.
- The foot collider for enemies functions the same way as it does for the player.
- The outer CapsuleCollider for enemies functions the same way as it does for the player.
- The inner CapsuleCollider is used for the AI system, which will be explained in more detail later.

Rock Boss

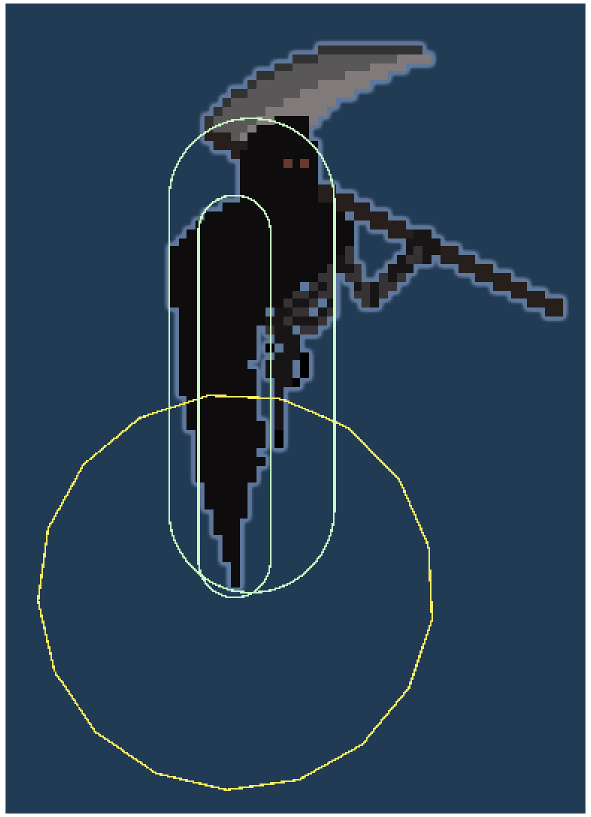


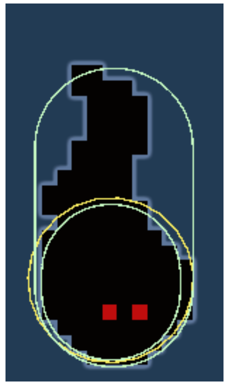




- This boss is modeled in the same way as the enemies in terms of colliders. However, instead of a single external collider for registering collisions with things that can damage it, the boss has three external colliders.
- The thrown knives and the laser each have a collider to register collisions with the player, allowing them to deal damage.
- From an animation standpoint, in addition to the Idle, Walk, and Death animations, the boss also has Attack, **Shoot**, and **Laser animations**.

UnDead Boss and ghost enemies





- This boss is modeled like all the other enemies in terms of colliders.
- In addition to the standard animations, the boss also has an animation for **summoning** the "Ghosts."
- The "Ghosts" are special enemies spawned by the boss that can pass over water.
- From a collider perspective, the Ghosts function the same way as all the other enemies.

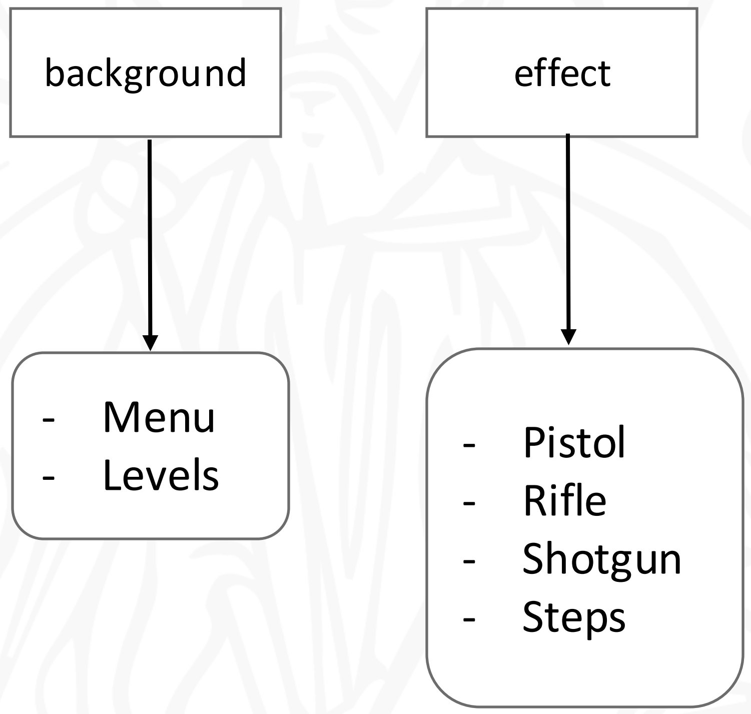
Audio system

Sound list

There are a total of six soundtracks, divided into two main groups: background music and sound effects.

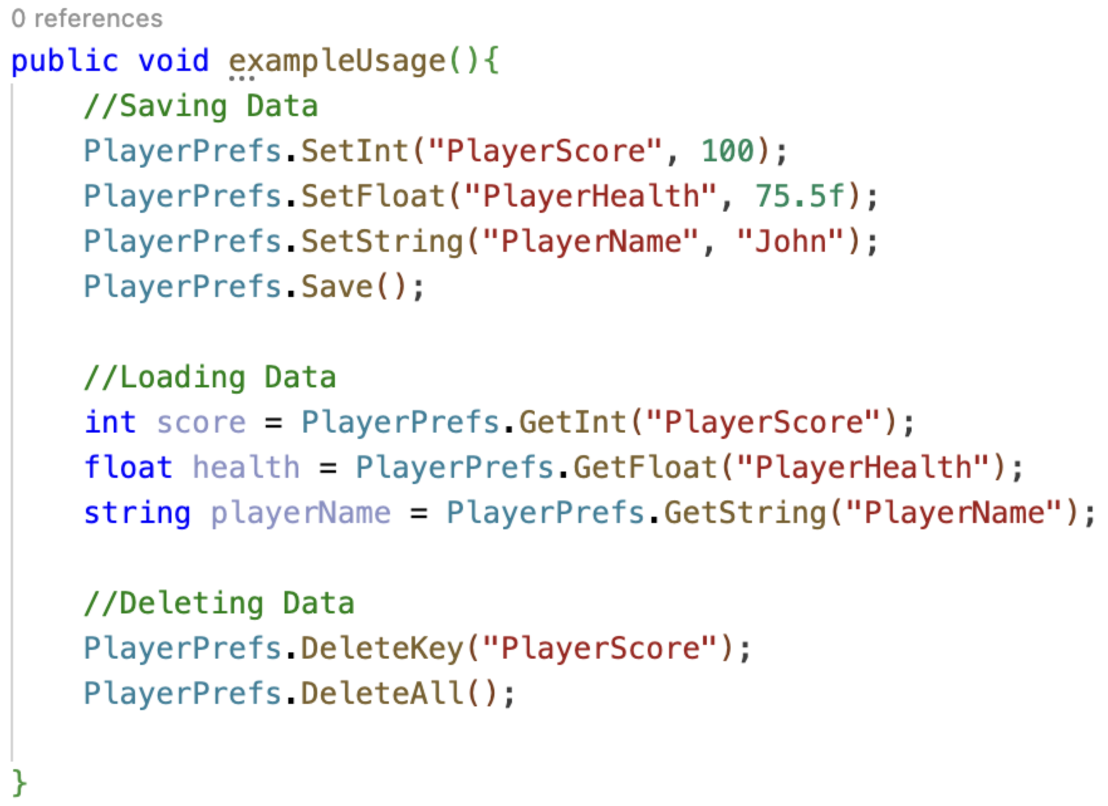
The background music includes two tracks—one for the menu scenes and one for the level scenes.

The sound effects consist of shooting sounds and footsteps.



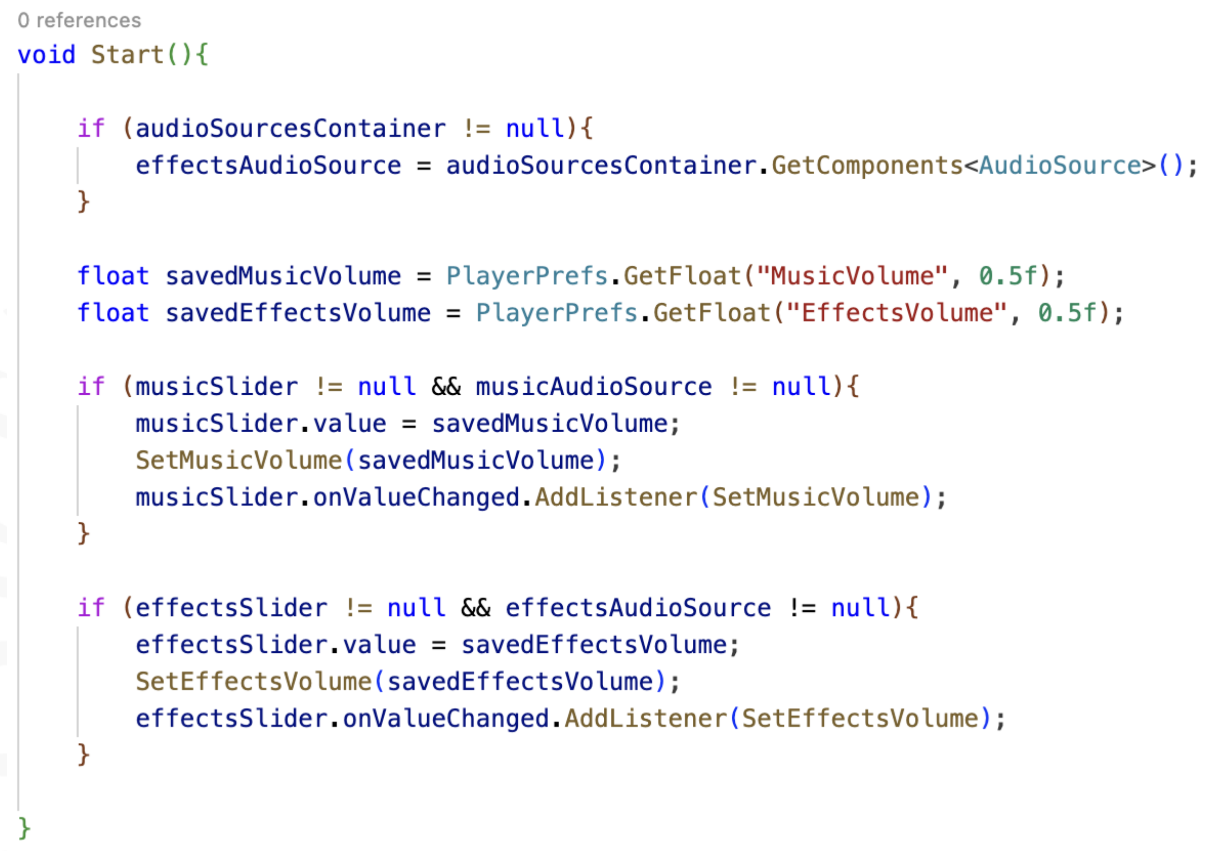
PlayerPrefs

`PlayerPrefs` in Unity is a class used to store and access small amounts of data between game sessions, typically for settings, preferences, or simple persistent values. It works like a key-value storage, where data is saved in the form of integers, floats, or strings and can be retrieved later.


```
0 references
public void exampleUsage(){
    //Saving Data
    PlayerPrefs.SetInt("PlayerScore", 100);
    PlayerPrefs.SetFloat("PlayerHealth", 75.5f);
    PlayerPrefs.SetString("PlayerName", "John");
    PlayerPrefs.Save();

    //Loading Data
    int score = PlayerPrefs.GetInt("PlayerScore");
    float health = PlayerPrefs.GetFloat("PlayerHealth");
    string playerName = PlayerPrefs.GetString("PlayerName");

    //Deleting Data
    PlayerPrefs.DeleteKey("PlayerScore");
    PlayerPrefs.DeleteAll();

}
```

Save audio settings: implementation

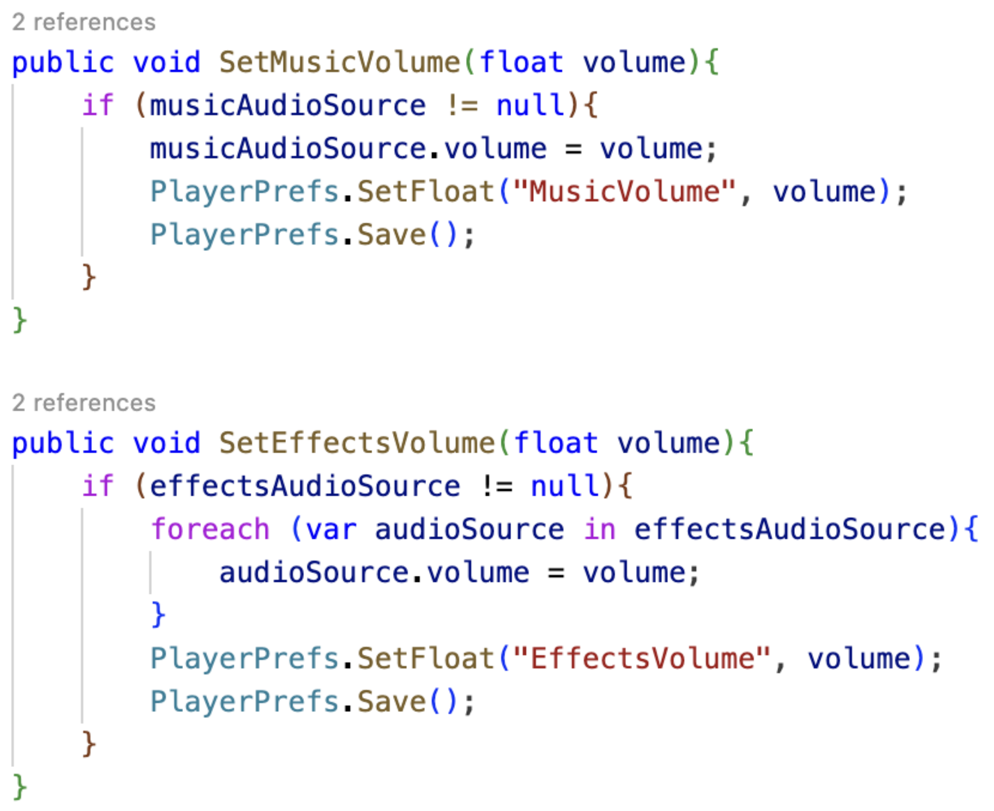
In the `Start` method, the last saved volume is retrieved and assigned to the current volume in the scene.



```
0 references
void Start(){

    if (audioSourcesContainer != null){
        effectsAudioSource = audioSourcesContainer.GetComponents<AudioSource>();
    }

    float savedMusicVolume = PlayerPrefs.GetFloat("MusicVolume", 0.5f);
    float savedEffectsVolume = PlayerPrefs.GetFloat("EffectsVolume", 0.5f);

    if (musicSlider != null && musicAudioSource != null){
        musicSlider.value = savedMusicVolume;
        SetMusicVolume(savedMusicVolume);
        musicSlider.onValueChanged.AddListener(SetMusicVolume);
    }

    if (effectsSlider != null && effectsAudioSource != null){
        effectsSlider.value = savedEffectsVolume;
        SetEffectsVolume(savedEffectsVolume);
        effectsSlider.onValueChanged.AddListener(SetEffectsVolume);
    }

}
```

Save audio settings: implementation

The `setVolumes` function is called every time a slider is moved, ensuring that the volume levels are always updated and saved in real time.

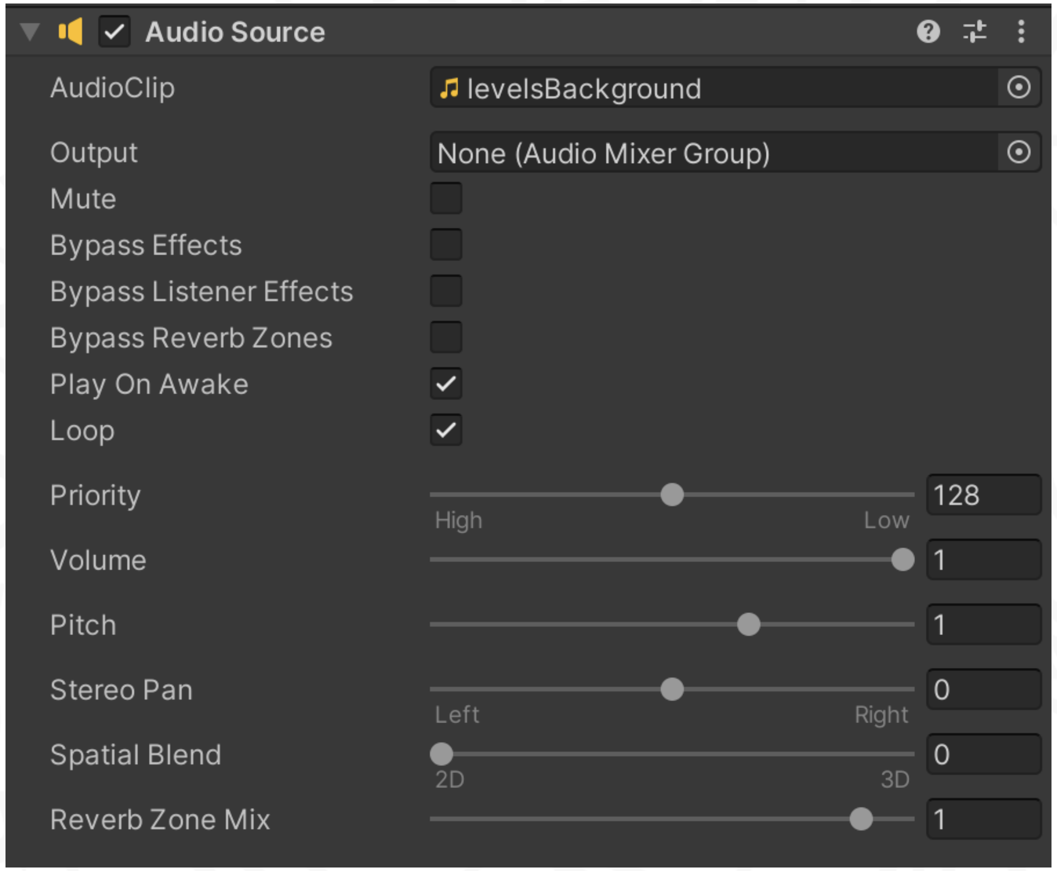


```
2 references
public void SetMusicVolume(float volume){
    if (musicAudioSource != null){
        musicAudioSource.volume = volume;
        PlayerPrefs.SetFloat("MusicVolume", volume);
        PlayerPrefs.Save();
    }
}

2 references
public void SetEffectsVolume(float volume){
    if (effectsAudioSource != null){
        foreach (var audioSource in effectsAudioSource){
            audioSource.volume = volume;
        }
        PlayerPrefs.SetFloat("EffectsVolume", volume);
        PlayerPrefs.Save();
    }
}
```

Playing Sounds

Three different methods are used to play the sounds:

1. Background music starts automatically when the scene is loaded and plays on a loop. This is achieved by setting the appropriate flags in Unity.

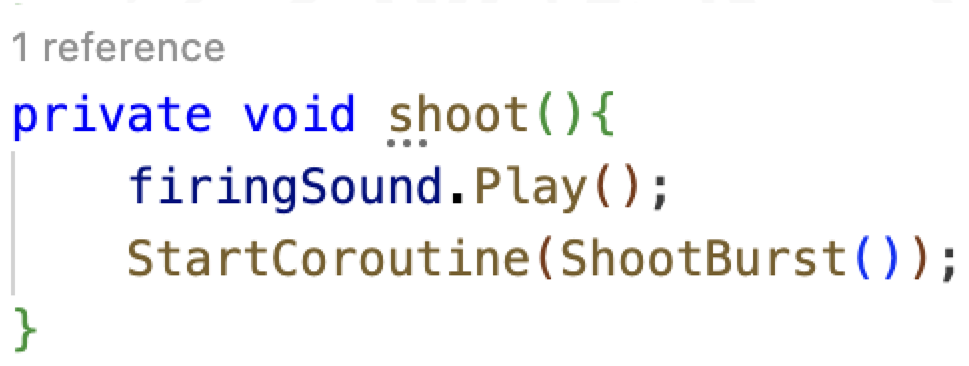


Audio Source	
AudioClip	levelsBackground
Output	None (Audio Mixer Group)
Mute	
Bypass Effects	
Bypass Listener Effects	
Bypass Reverb Zones	
Play On Awake	✓
Loop	✓
Priority (High – Low)	128
Volume	1
Pitch	1
Stereo Pan (Left – Right)	0
Spatial Blend (2D – 3D)	0
Reverb Zone Mix	1

Playing Sounds

Three different methods are used to play the sounds:

2. The firing sound is played whenever the shoot function is called.



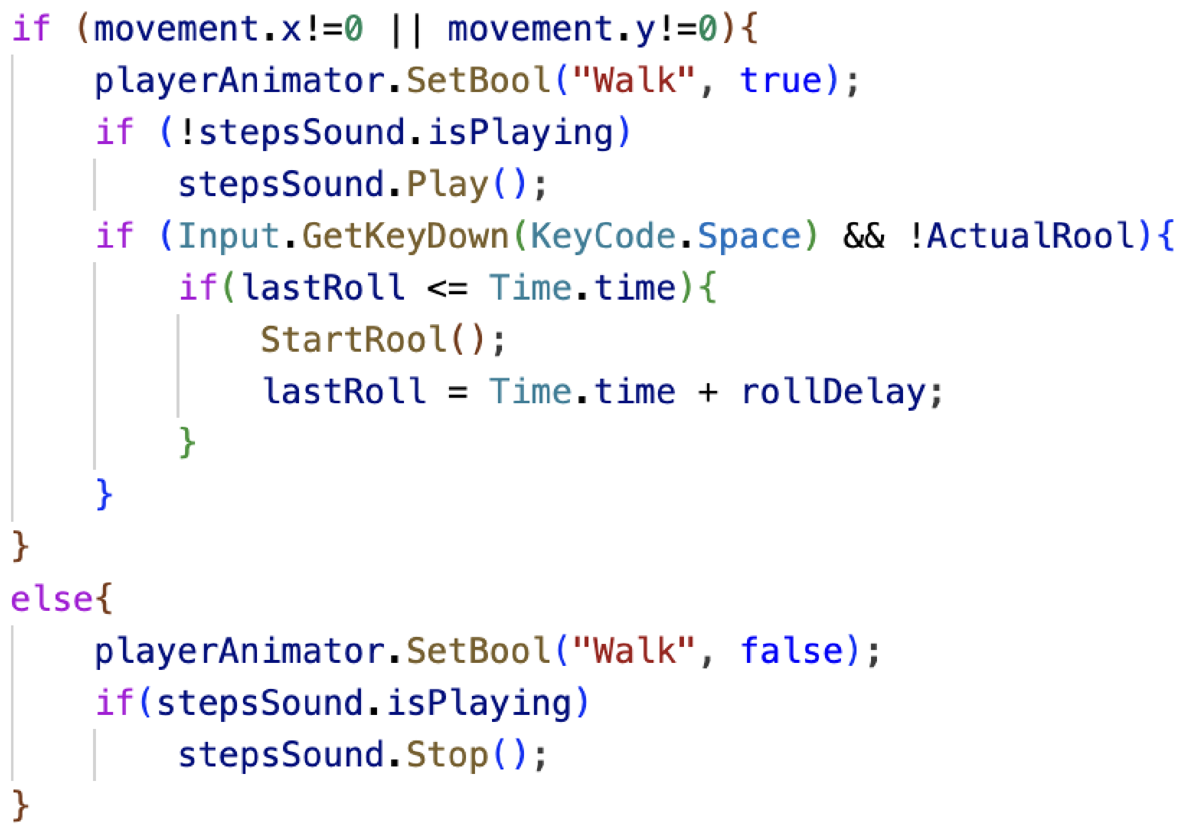
1 reference

```
private void shoot(){
    firingSound.Play();
    StartCoroutine(ShootBurst());
}
```

Playing Sounds

Three different methods are used to play the sounds:

3. For the footsteps sound, it is played only after the previous sound has finished to avoid overlapping or abrupt cuts. This ensures the sound doesn't restart before the current one ends.



```
if (movement.x!=0 || movement.y!=0){
    playerAnimator.SetBool("Walk", true);
    if (!stepsSound.isPlaying)
        stepsSound.Play();
    if (Input.GetKeyDown(KeyCode.Space) && !ActualRool){
        if(lastRoll <= Time.time){
            StartRool();
            lastRoll = Time.time + rollDelay;
        }
    }
}
else{
    playerAnimator.SetBool("Walk", false);
    if(stepsSound.isPlaying)
        stepsSound.Stop();
}
```

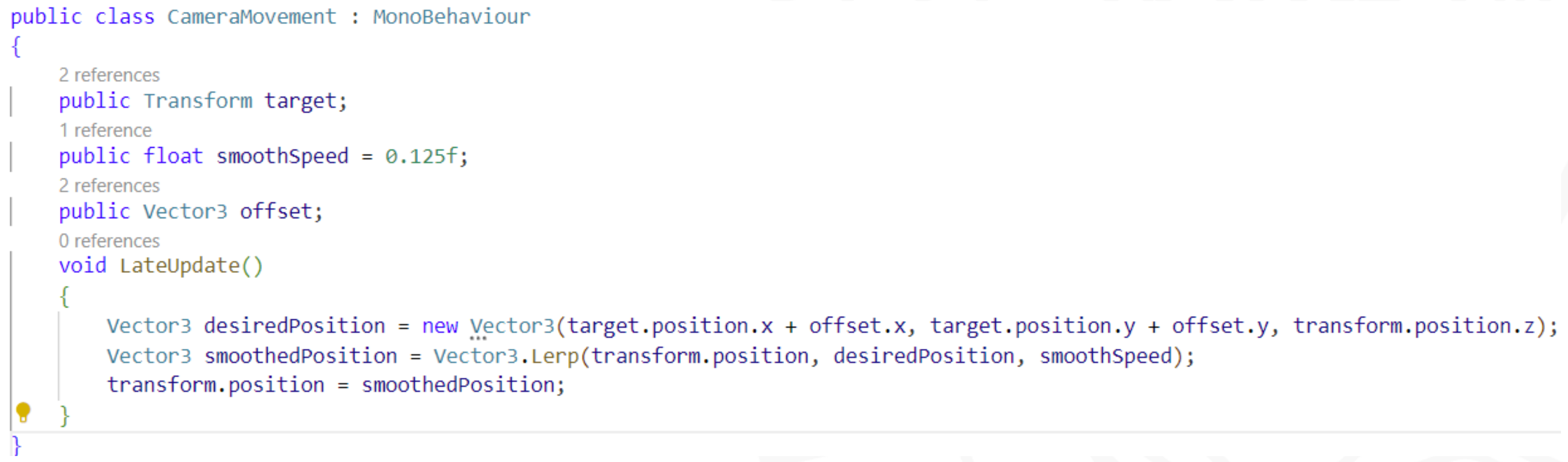
UNIVERSITÀ
DEGLI STUDI
FIRENZE

Camera System

Player Camera and Minimap

Camera System: Player Camera

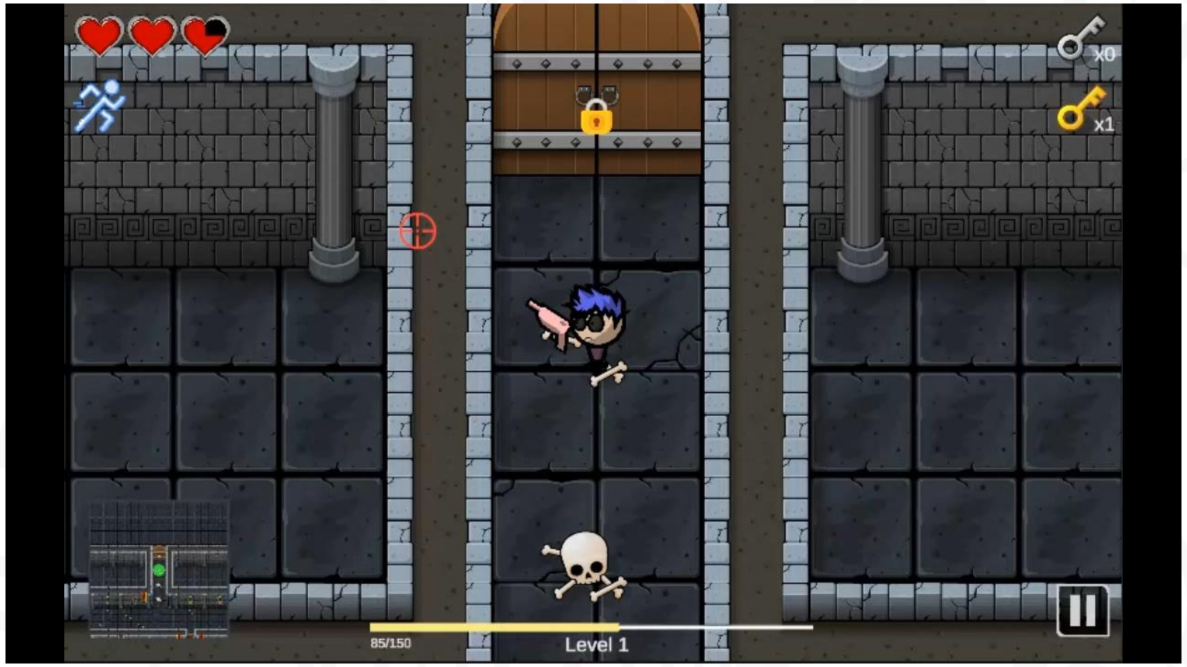
- For this type of game, a **top-down view** is used, where the camera follows the player from above as they move.
- We have reimplemented this classic functionality by moving the camera at runtime to follow the player, adding a smooth effect using the "Lerp" function.



```
public class CameraMovement : MonoBehaviour
{
    2 references
    public Transform target;
    1 reference
    public float smoothSpeed = 0.125f;
    2 references
    public Vector3 offset;
    0 references
    void LateUpdate()
    {
        Vector3 desiredPosition = new Vector3(target.position.x + offset.x, target.position.y + offset.y, transform.position.z);
        Vector3 smoothedPosition = Vector3.Lerp(transform.position, desiredPosition, smoothSpeed);
        transform.position = smoothedPosition;
    }
}
```

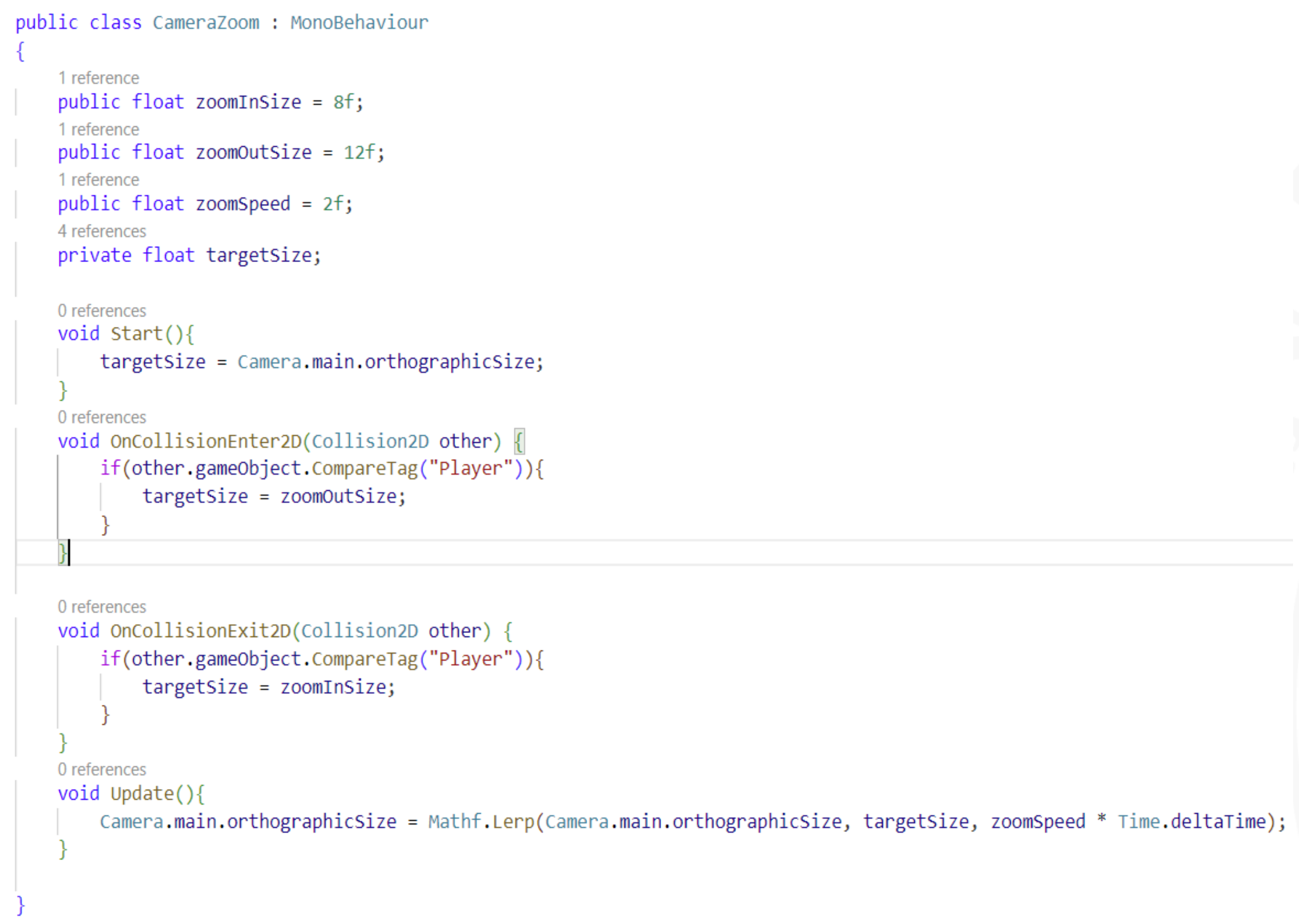
Player Camera: Zoom In/Out Effects

- For the rooms on the map where the end-level bosses are located, a BoxCollider has been added to detect collisions with the player.
- When the **player enters** the region, the camera zooms out, and when the **player exits**, the camera zooms back in to its original position.



UNIVERSITÀ DEGLI STUDI FIRENZE

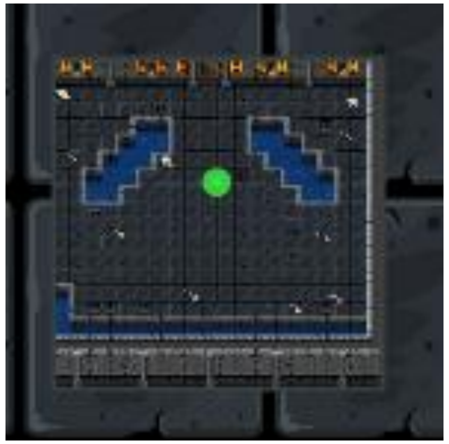
Player Camera: Zoom In/Out Effects implementations



```
public class CameraZoom : MonoBehaviour
{
    1 reference
    public float zoomInSize = 8f;
    1 reference
    public float zoomOutSize = 12f;
    1 reference
    public float zoomSpeed = 2f;
    4 references
    private float targetSize;

    0 references
    void Start(){
        targetSize = Camera.main.orthographicSize;
    }
    0 references
    void OnCollisionEnter2D(Collision2D other) {
        if(other.gameObject.CompareTag("Player")){
            targetSize = zoomOutSize;
        }
    }

    0 references
    void OnCollisionExit2D(Collision2D other) {
        if(other.gameObject.CompareTag("Player")){
            targetSize = zoomInSize;
        }
    }
    0 references
    void Update(){
        Camera.main.orthographicSize = Mathf.Lerp(Camera.main.orthographicSize, targetSize, zoomSpeed * Time.deltaTime);
    }

}
```

- Once again, to make the effect smoother and avoid jerky transitions, we used the "Lerp" function.
- The collider for this element is set to "**Trigger**" to ensure it doesn't apply any physical effects to other objects on the map but is used solely to register this event.

Camera System: Minimap camera

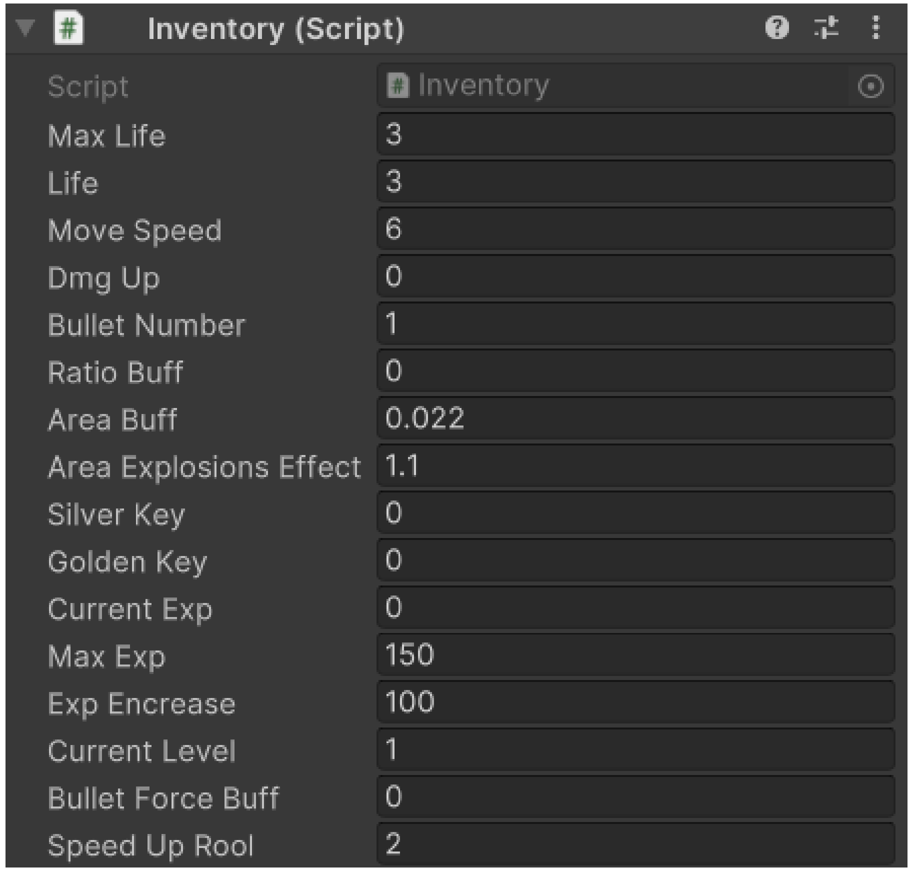


- To create the minimap, we implemented a second camera that follows the player just like the main camera but from a different height.
- This second camera **outputs to a render texture**, which is then displayed as the image on the minimap in the bottom left corner.
- This camera can only see specific layers of the game using the "**culling mask**" feature. As a result, enemies, keys, weapons, and the player are excluded from the minimap's view.
- The player has been replaced with a larger green dot that the main camera cannot see.

Inventory

Persistent Information

Inventory System

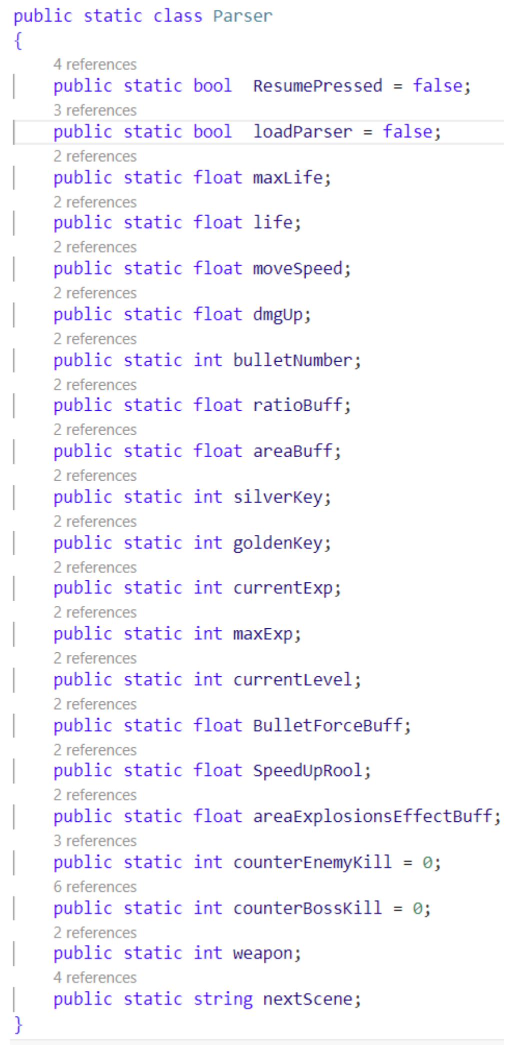


Inventory (Script)

Script	Inventory
Max Life	3
Life	3
Move Speed	6
Dmg Up	0
Bullet Number	1
Ratio Buff	0
Area Buff	0.022
Area Explosions Effect	1.1
Silver Key	0
Golden Key	0
Current Exp	0
Max Exp	150
Exp Encrease	100
Current Level	1
Bullet Force Buff	0
Speed Up Rool	2

- An Inventory script has been created to keep track of everything that belongs to the player during the game. As we will see later, it also tracks levels and power-ups acquired throughout the game.
- This class is a **singleton** with a static reference object, which prevents issues such as creating multiple inventories. Additionally, it allows for easy access from outside the class.

Inventory System Between Scenes: Parser Class

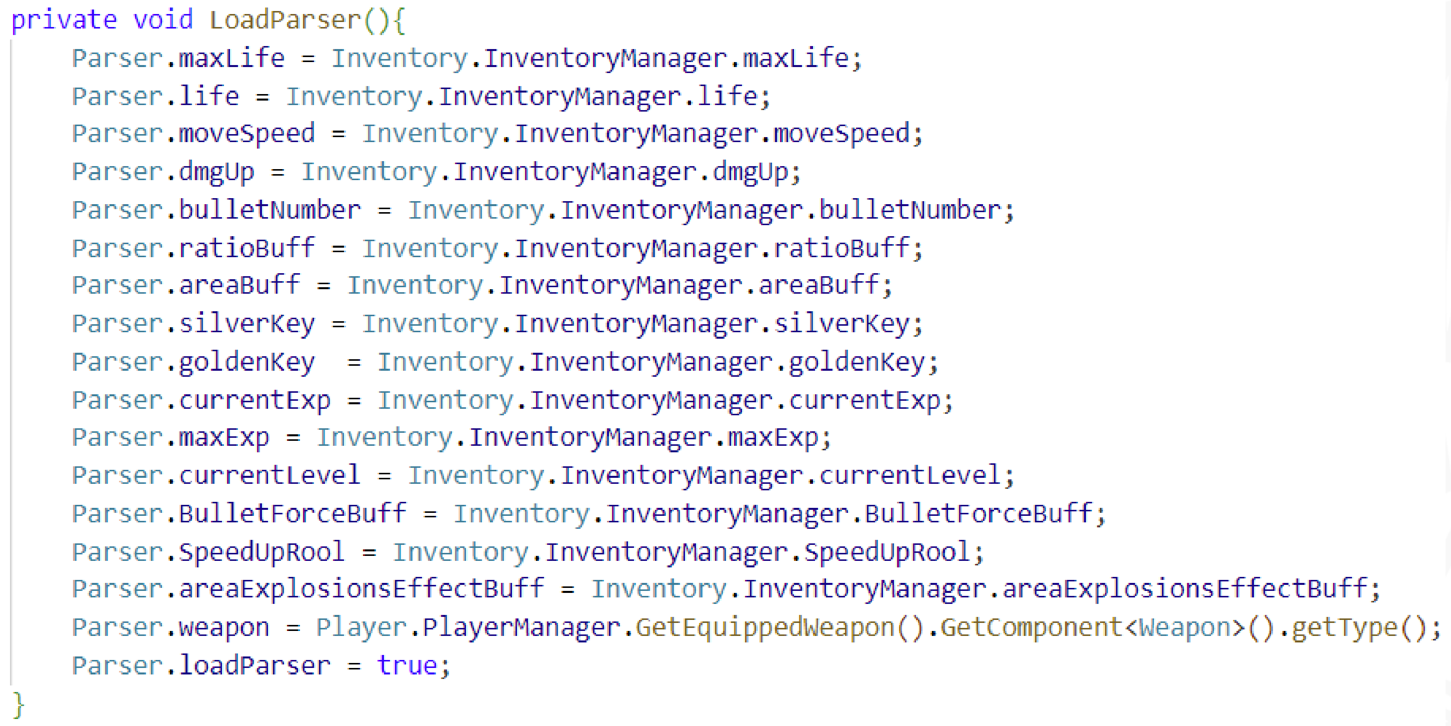


```
public static class Parser
{
    4 references
    public static bool  ResumePressed = false;
    3 references
    public static bool  loadParser = false;
    2 references
    public static float maxLife;
    2 references
    public static float life;
    2 references
    public static float moveSpeed;
    2 references
    public static float dmgUp;
    2 references
    public static int bulletNumber;
    2 references
    public static float ratioBuff;
    2 references
    public static float areaBuff;
    2 references
    public static int silverKey;
    2 references
    public static int goldenKey;
    2 references
    public static int currentExp;
    2 references
    public static int maxExp;
    2 references
    public static int currentLevel;
    2 references
    public static float BulletForceBuff;
    2 references
    public static float SpeedUpRool;
    2 references
    public static float areaExplosionsEffectBuff;
    3 references
    public static int counterEnemyKill = 0;
    6 references
    public static int counterBossKill = 0;
    2 references
    public static int weapon;
    4 references
    public static string nextScene;
}
```

- To keep track of the player's inventory values when transitioning from one level to another or changing scenes, we have implemented a **static** `Parser` class that holds all the values to be imported into the next scene.
- In addition to all the inventory values, this class also saves the player's current weapon at the end of the level, so the player can start the next level with the same weapon.

Level Finish, Store information

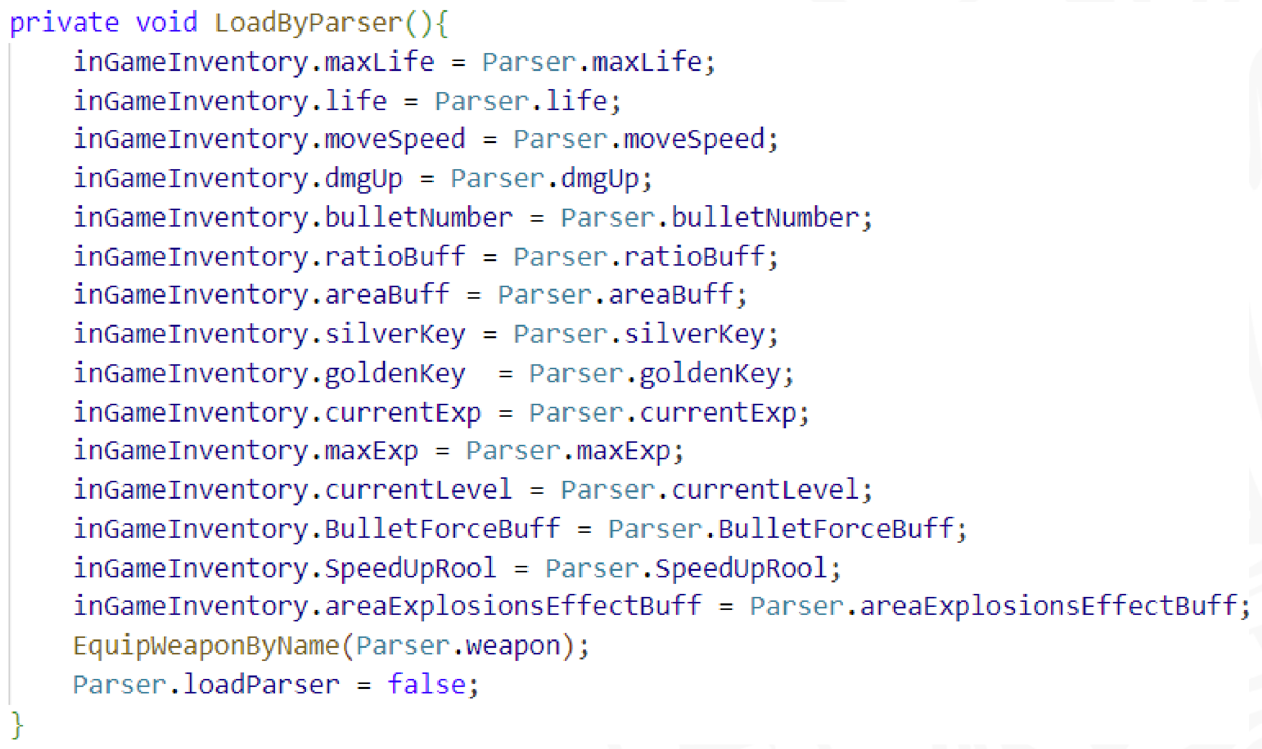
- This function is called when the player makes contact with the final stairs at the end of the level. It is used to store all relevant data into the `Parser` class.



```
private void LoadParser(){
    Parser.maxLife = Inventory.InventoryManager.maxLife;
    Parser.life = Inventory.InventoryManager.life;
    Parser.moveSpeed = Inventory.InventoryManager.moveSpeed;
    Parser.dmgUp = Inventory.InventoryManager.dmgUp;
    Parser.bulletNumber = Inventory.InventoryManager.bulletNumber;
    Parser.ratioBuff = Inventory.InventoryManager.ratioBuff;
    Parser.areaBuff = Inventory.InventoryManager.areaBuff;
    Parser.silverKey = Inventory.InventoryManager.silverKey;
    Parser.goldenKey  = Inventory.InventoryManager.goldenKey;
    Parser.currentExp = Inventory.InventoryManager.currentExp;
    Parser.maxExp = Inventory.InventoryManager.maxExp;
    Parser.currentLevel = Inventory.InventoryManager.currentLevel;
    Parser.BulletForceBuff = Inventory.InventoryManager.BulletForceBuff;
    Parser.SpeedUpRool = Inventory.InventoryManager.SpeedUpRool;
    Parser.areaExplosionsEffectBuff = Inventory.InventoryManager.areaExplosionsEffectBuff;
    Parser.weapon = Player.PlayerManager.GetEquippedWeapon().GetComponent<Weapon>().getType();
    Parser.loadParser = true;
}
```

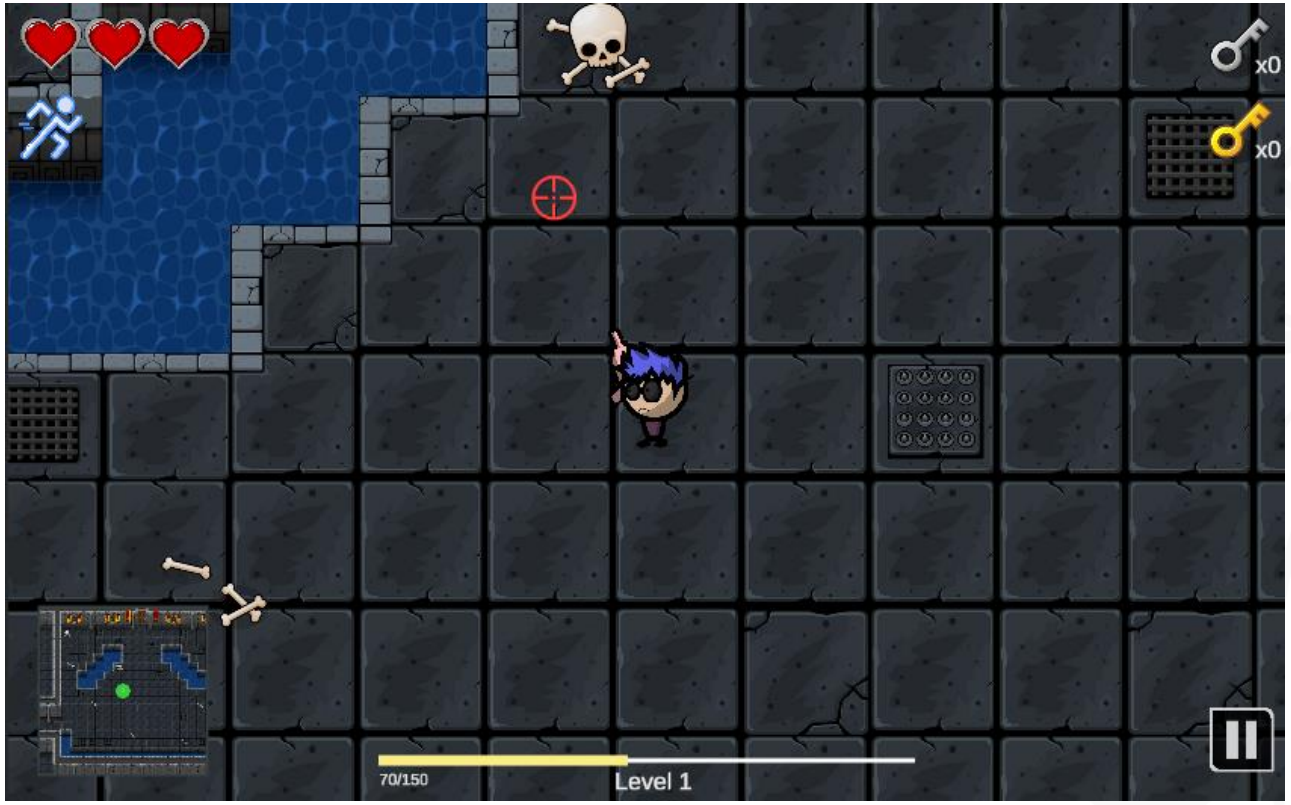
Level Start, Load information

- This function is complementary and is called after loading the next scene. It restores the inventory to its state from the previous level and equips the player with their previous weapon.



```
private void LoadByParser(){
    inGameInventory.maxLife = Parser.maxLife;
    inGameInventory.life = Parser.life;
    inGameInventory.moveSpeed = Parser.moveSpeed;
    inGameInventory.dmgUp = Parser.dmgUp;
    inGameInventory.bulletNumber = Parser.bulletNumber;
    inGameInventory.ratioBuff = Parser.ratioBuff;
    inGameInventory.areaBuff = Parser.areaBuff;
    inGameInventory.silverKey = Parser.silverKey;
    inGameInventory.goldenKey  = Parser.goldenKey;
    inGameInventory.currentExp = Parser.currentExp;
    inGameInventory.maxExp = Parser.maxExp;
    inGameInventory.currentLevel = Parser.currentLevel;
    inGameInventory.BulletForceBuff = Parser.BulletForceBuff;
    inGameInventory.SpeedUpRool = Parser.SpeedUpRool;
    inGameInventory.areaExplosionsEffectBuff = Parser.areaExplosionsEffectBuff;
    EquipWeaponByName(Parser.weapon);
    Parser.loadParser = false;
}
```

Inventory and HUD



- The HUD has its own script that **retrieves information** from the inventory at runtime and updates the graphics displayed to the user.
- Except for the "Roll" icon in the top left corner.

UNIVERSITÀ DEGLI STUDI FIRENZE

Power Up System

Scriptable Object

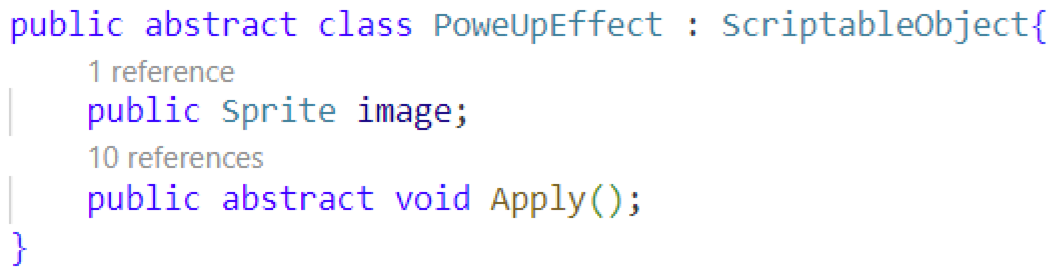
Power Up System

- For this game, a **progression system** based on experience and levels has been designed.
- Each monster killed **drops experience** points that can be absorbed by the player.
- At each level, the player is prompted to choose between three "random" upgrades.
- 9 different upgrades have been implemented, including increased damage, an extra projectile, enhanced movement speed, an additional HP, and more.

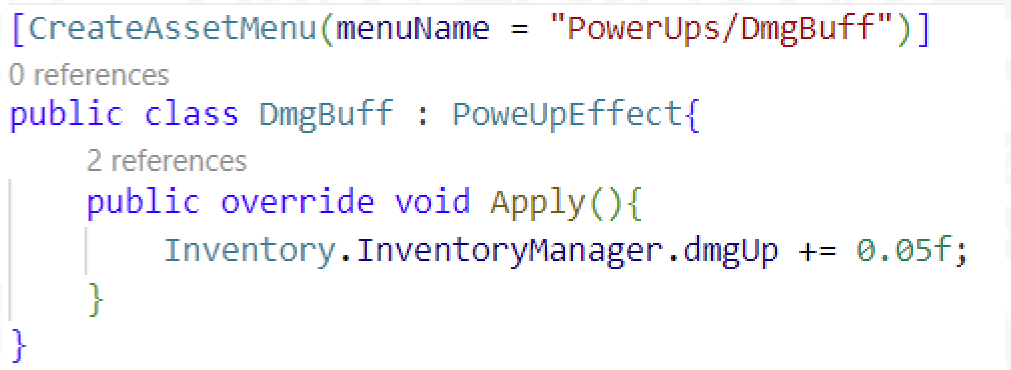
UNIVERSITÀ DEGLI STUDI FIRENZE

Power Up System: Implementation

- An abstract class, PowerUpEffects, was created, defining an **abstract method Apply()**, which is then overridden for each specific power-up in its corresponding class.
- PowerUpEffect is a class that extends **ScriptableObject**, allowing the power-ups to be created as assets and **easily managed** within the Unity editor.



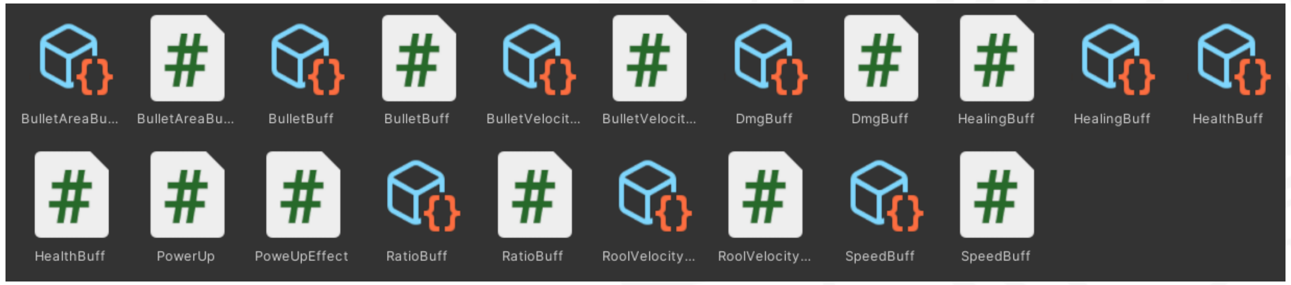
```
public abstract class PoweUpEffect : ScriptableObject{
    1 reference
    public Sprite image;
    10 references
    public abstract void Apply();
}
```



```
[CreateAssetMenu(menuName = "PowerUps/DmgBuff")]
0 references
public class DmgBuff : PoweUpEffect{
    2 references
    public override void Apply(){
        Inventory.InventoryManager.dmgUp += 0.05f;
    }
}
```

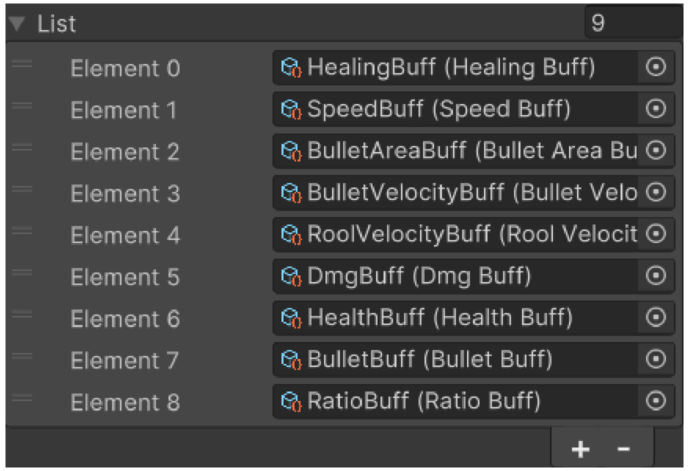
Power Up System: Scriptable Object

- **Separation of Data and Logic**: With ScriptableObjects, power-up data (like damage boosts or health increases) is stored separately from game logic. This makes your code cleaner and easier to manage, as the power-up logic can be handled by specific scripts, while the data is defined independently in the ScriptableObject.
- **Reusability**: Since ScriptableObjects can be referenced across different scenes or game objects, they allow you to reuse the same power-ups in different contexts. This avoids the need to recreate or duplicate power-up logic and data for each instance or scene.

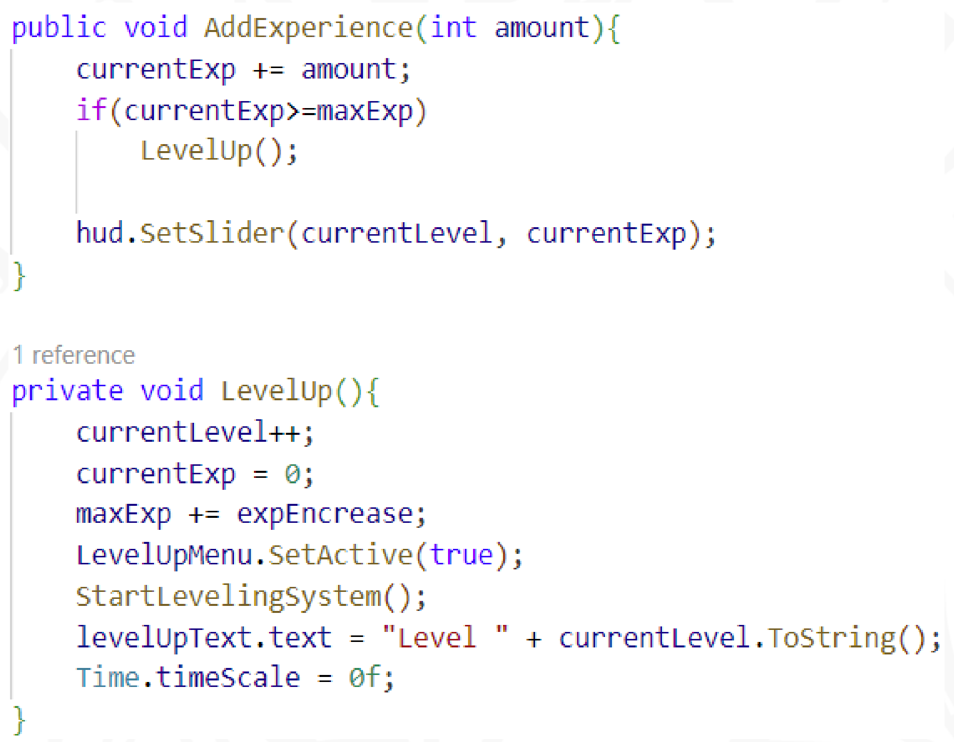


Power Up System: Level Up system

- The inventory contains a list of PowerUpEffects, including all 9 power-ups, each represented as a ScriptableObject. This allows the game to easily manage and apply the various power-up effects through the inventory system.

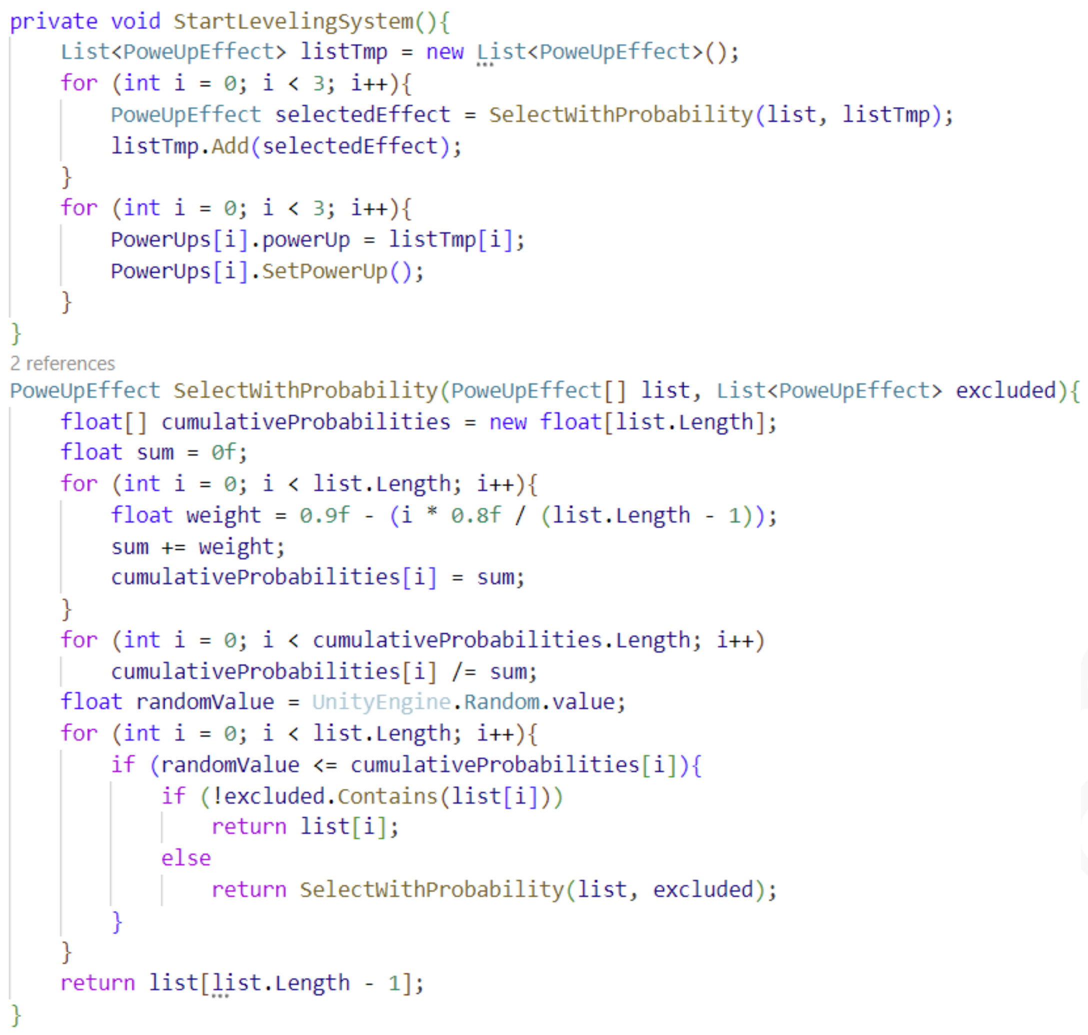


List	9
Element 0	HealingBuff (Healing Buff)
Element 1	SpeedBuff (Speed Buff)
Element 2	BulletAreaBuff (Bullet Area Bu
Element 3	BulletVelocityBuff (Bullet Velo
Element 4	RoolVelocityBuff (Rool Velocit
Element 5	DmgBuff (Dmg Buff)
Element 6	HealthBuff (Health Buff)
Element 7	BulletBuff (Bullet Buff)
Element 8	RatioBuff (Ratio Buff)



```
public void AddExperience(int amount){
    currentExp += amount;
    if(currentExp>=maxExp)
        LevelUp();

    hud.SetSlider(currentLevel, currentExp);
}

1 reference
private void LevelUp(){
    currentLevel++;
    currentExp = 0;
    maxExp += expEncrease;
    LevelUpMenu.SetActive(true);
    StartLevelingSystem();
    levelUpText.text = "Level " + currentLevel.ToString();
    Time.timeScale = 0f;
}
```

Power Up System: Level Up system



```
private void StartLevelingSystem(){
    List<PoweUpEffect> listTmp = new List<PoweUpEffect>();
    for (int i = 0; i < 3; i++){
        PoweUpEffect selectedEffect = SelectWithProbability(list, listTmp);
        listTmp.Add(selectedEffect);
    }
    for (int i = 0; i < 3; i++){
        PowerUps[i].powerUp = listTmp[i];
        PowerUps[i].SetPowerUp();
    }
}
2 references
PoweUpEffect SelectWithProbability(PoweUpEffect[] list, List<PoweUpEffect> excluded){
    float[] cumulativeProbabilities = new float[list.Length];
    float sum = 0f;
    for (int i = 0; i < list.Length; i++){
        float weight = 0.9f - (i * 0.8f / (list.Length - 1));
        sum += weight;
        cumulativeProbabilities[i] = sum;
    }
    for (int i = 0; i < cumulativeProbabilities.Length; i++)
        cumulativeProbabilities[i] /= sum;
    float randomValue = UnityEngine.Random.value;
    for (int i = 0; i < list.Length; i++){
        if (randomValue <= cumulativeProbabilities[i]){
            if (!excluded.Contains(list[i]))
                return list[i];
            else
                return SelectWithProbability(list, excluded);
        }
    }
    return list[list.Length - 1];
}
```

- The upgrades offered to the player are not entirely random. If they were, it could lead to significant **imbalance in the game**, as some upgrades are much stronger than others.
- To maintain balance, a **probability system** has been implemented, where certain power-ups have a higher or lower chance of being selected compared to others. This ensures that stronger upgrades appear less frequently, preventing overpowered combinations while still providing variety.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

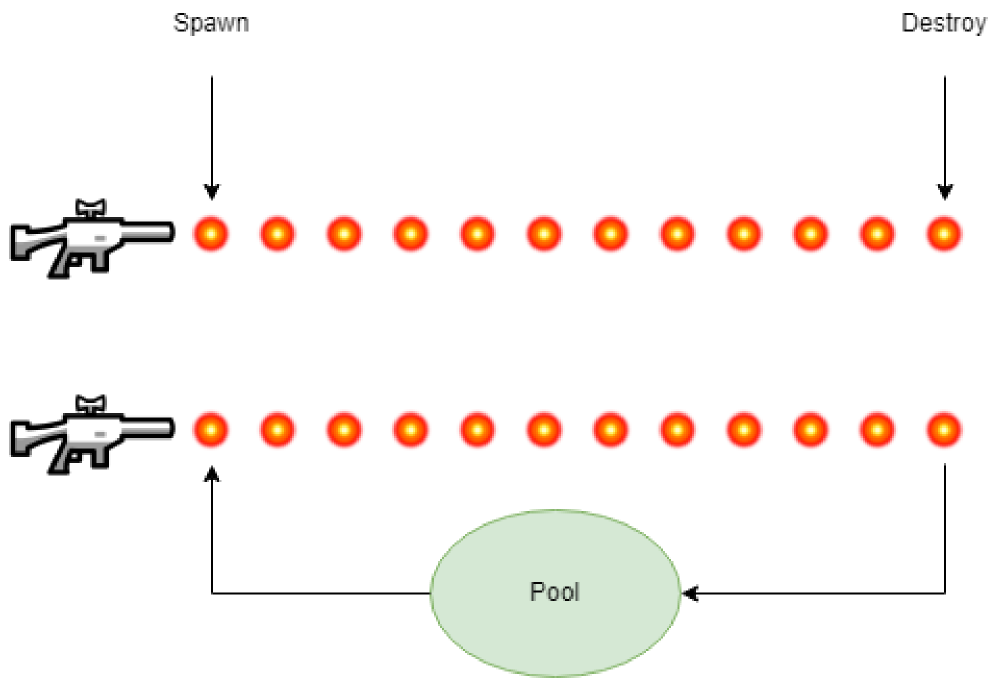
Game Patterns

Object Pooling

Object Pooling

- Object Pooling is an excellent technique for **optimizing** projects and reducing the **CPU load** caused by the frequent creation and destruction of GameObjects.
- It is a good practice and design pattern to keep in mind to help relieve the processing power of the CPU to handle more important tasks and not become inundated by repetitive create and destroy calls.
- This is particularly useful when dealing with **bullets** in a top-down shooter game.
- Object Pooling is a **creational design pattern** that pre-instantiates all the objects you'll need at any specific moment before gameplay. This removes the need to create new objects or destroy old ones while the game is running.

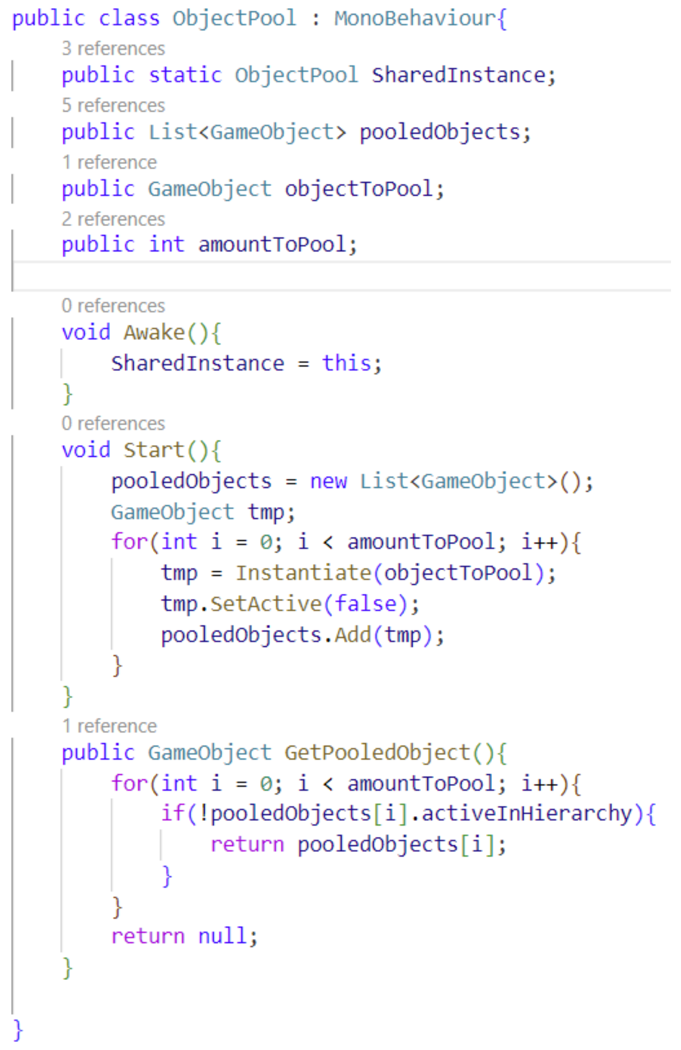
Object Pooling: How it Works



Instead of constantly spawning and destroying countless objects during the game, a fixed number of, for example, 1000 objects are spawned at the beginning of the game and **reused** throughout its duration.

With this implementation, a significant amount of workload is saved for the machine.

Object Pooling: Pool Implementation

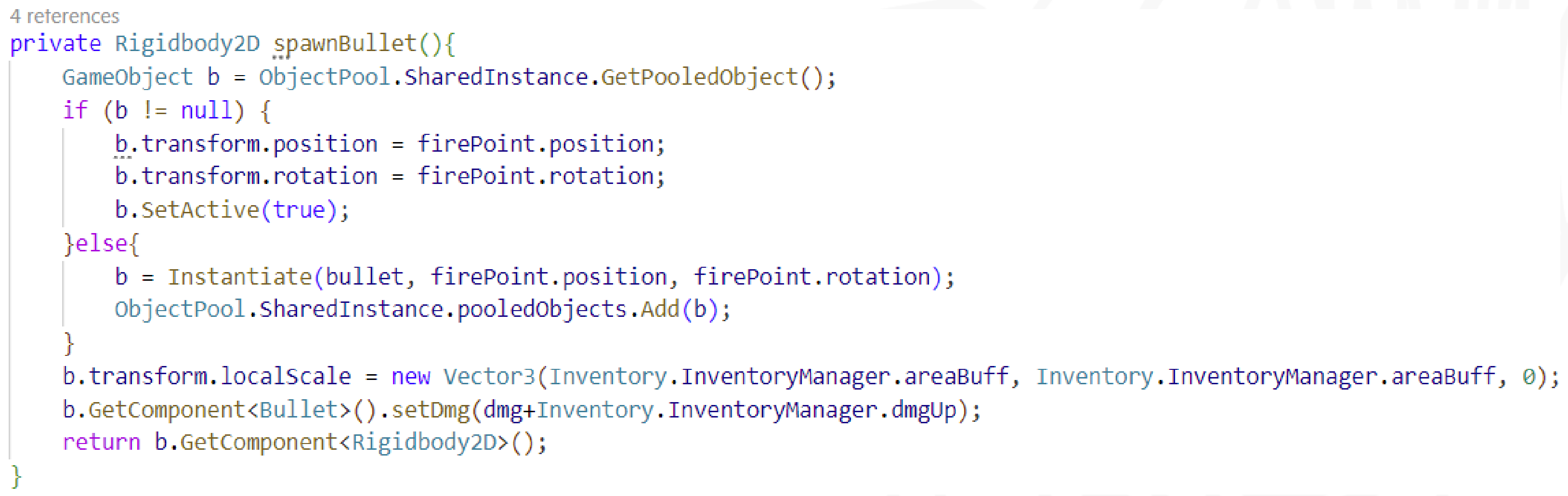


```
public class ObjectPool : MonoBehaviour{
    3 references
    public static ObjectPool SharedInstance;
    5 references
    public List<GameObject> pooledObjects;
    1 reference
    public GameObject objectToPool;
    2 references
    public int amountToPool;

    0 references
    void Awake(){
        SharedInstance = this;
    }
    0 references
    void Start(){
        pooledObjects = new List<GameObject>();
        GameObject tmp;
        for(int i = 0; i < amountToPool; i++){
            tmp = Instantiate(objectToPool);
            tmp.SetActive(false);
            pooledObjects.Add(tmp);
        }
    }
    1 reference
    public GameObject GetPooledObject(){
        for(int i = 0; i < amountToPool; i++){
            if(!pooledObjects[i].activeInHierarchy){
                return pooledObjects[i];
            }
        }
        return null;
    }

}
```

- During the initialization of this object, *n* objects, in this case *n* bullets, are instantiated and set to inactive.
- When one of these objects is needed, instead of instantiating a new one, the first inactive object in the pool can be found and used.

Object Pooling: Usage

- When a bullet needs to be spawned, the system first attempts to retrieve one from the pre-created bullet pool. If available, the bullet is moved to the firing point and activated. If none are available, a new one is created, added to the pool, and used for future needs.



```
4 references
private Rigidbody2D spawnBullet(){
    GameObject b = ObjectPool.SharedInstance.GetPooledObject();
    if (b != null) {
        b.transform.position = firePoint.position;
        b.transform.rotation = firePoint.rotation;
        b.SetActive(true);
    }else{
        b = Instantiate(bullet, firePoint.position, firePoint.rotation);
        ObjectPool.SharedInstance.pooledObjects.Add(b);
    }
    b.transform.localScale = new Vector3(Inventory.InventoryManager.areaBuff, Inventory.InventoryManager.areaBuff, 0);
    b.GetComponent<Bullet>().setDmg(dmg+Inventory.InventoryManager.dmgUp);
    return b.GetComponent<Rigidbody2D>();
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

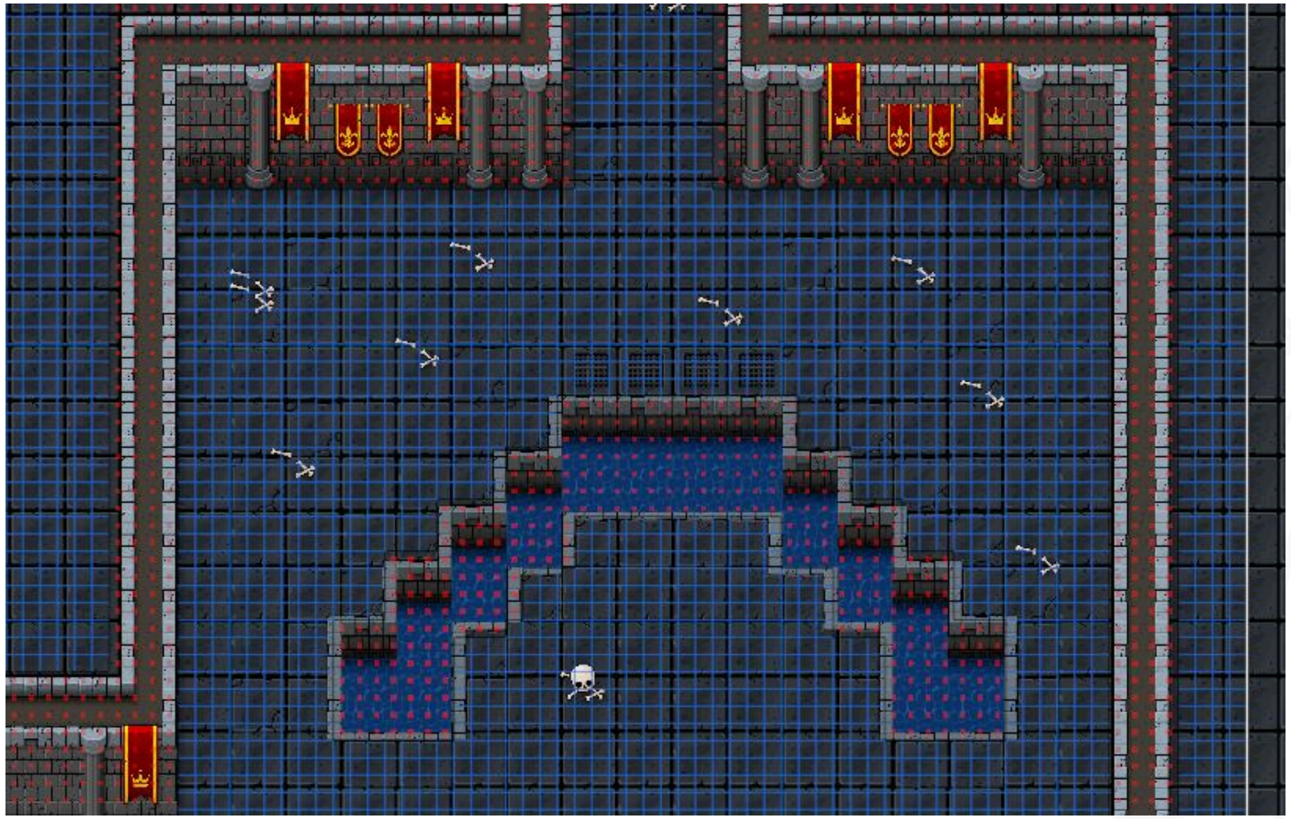
AI: A*

Path Finder and Grid System

Why is a path-finding algorithm necessary?

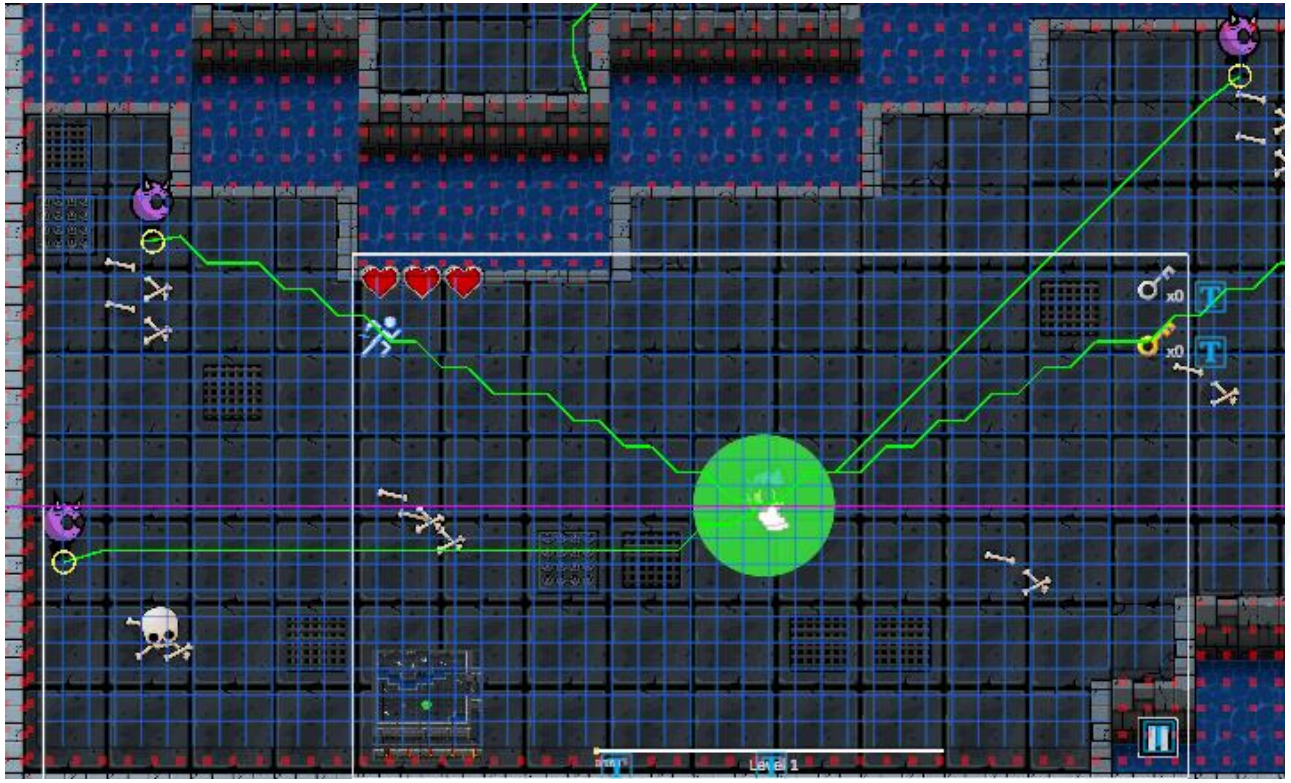
- The first solution would be to have the enemies move directly toward the player. However, the map is designed with various **obstacles**, such as water, walls, doors, etc., that hinder a straightforward path.
- To effectively and quickly navigate around these obstacles, we need a **pathfinding algorithm based on heuristics**.
- The **A*** algorithm with a Euclidean heuristic has been chosen.

Grid System



- It was therefore necessary to construct a **grid graph based** on the level maps, specifying the layers through which enemies cannot pass, such as "Walls," "Water," and the red markers.
- The **node size** was set significantly smaller than the map's tilesets to allow for smoother and more accurate movement.

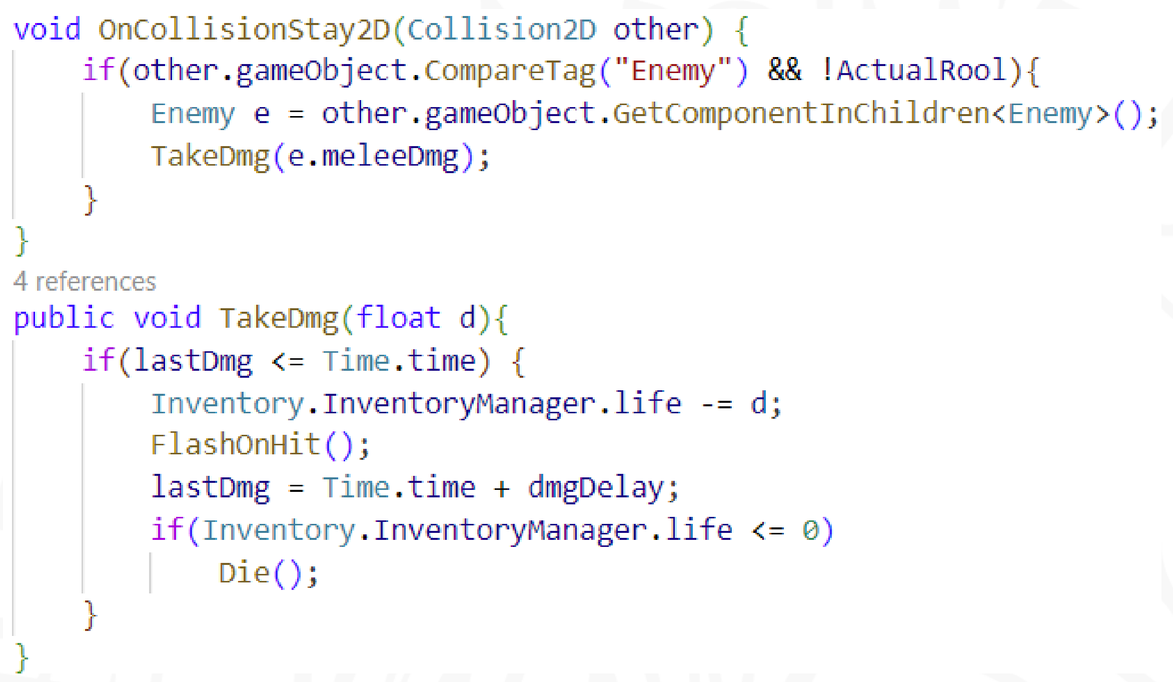
Enemy Search Best Path



- The green lines indicate the paths calculated by the various enemies to reach the player.
- Additionally, each enemy has a different **aggro distance**. This way, if the player moves far enough away, the AI for that enemy is deactivated.

Enemy Settings

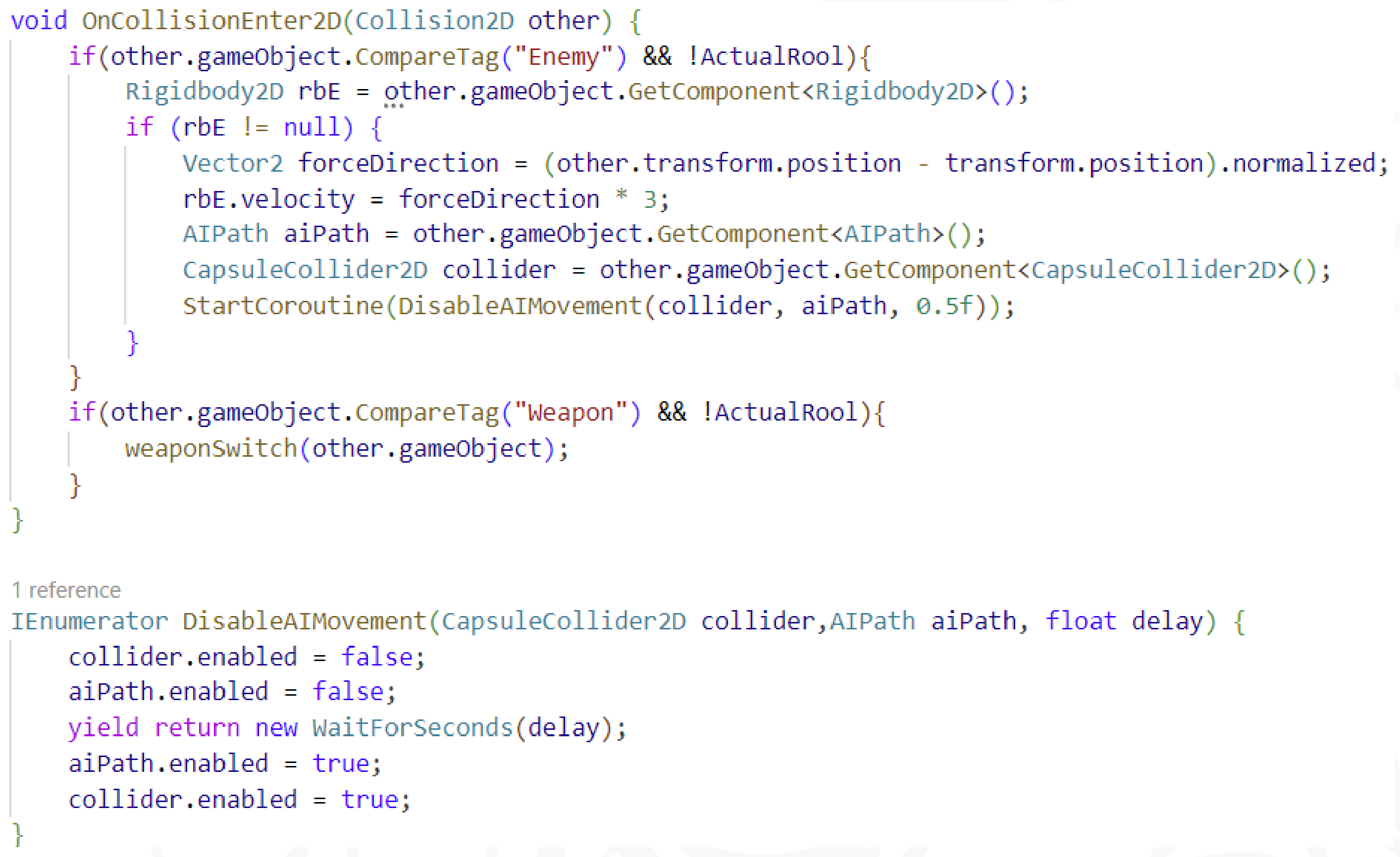
- Each enemy is assigned a grid (the graph) on which it can move and a specific interval at which it must recalculate the best path to the player.
- Additionally, each enemy is given the target they need to follow.
- Once the enemies reach the player, they can then trigger collisions and deal damage.



```
void OnCollisionStay2D(Collision2D other) {
    if(other.gameObject.CompareTag("Enemy") && !ActualRool){
        Enemy e = other.gameObject.GetComponentInChildren<Enemy>();
        TakeDmg(e.meleeDmg);
    }
}
4 references
public void TakeDmg(float d){
    if(lastDmg <= Time.time) {
        Inventory.InventoryManager.life -= d;
        FlashOnHit();
        lastDmg = Time.time + dmgDelay;
        if(Inventory.InventoryManager.life <= 0)
            Die();
    }
}
```

Enemy Attack Problem (2)

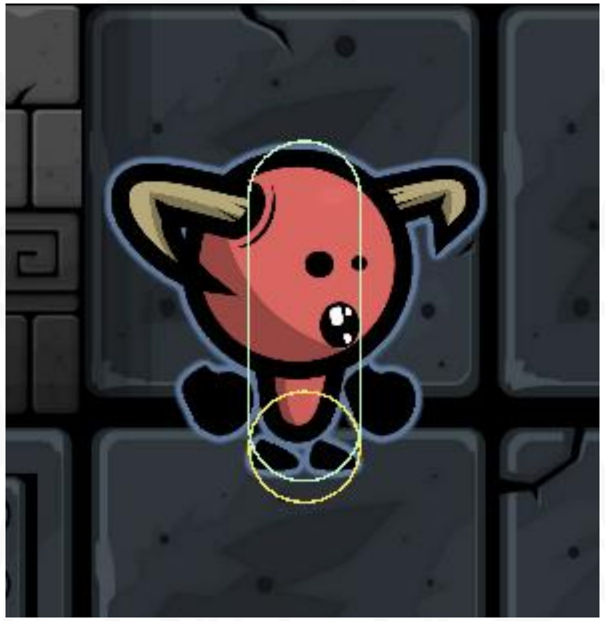
- In this way, the enemies tend to **cluster** around the player, often resulting in the player being **instantly killed**.
- **Solution:** After taking damage, the player cannot receive additional damage for a brief **delay**. Additionally, any enemy that hits the player will experience a **repulsion effect**.



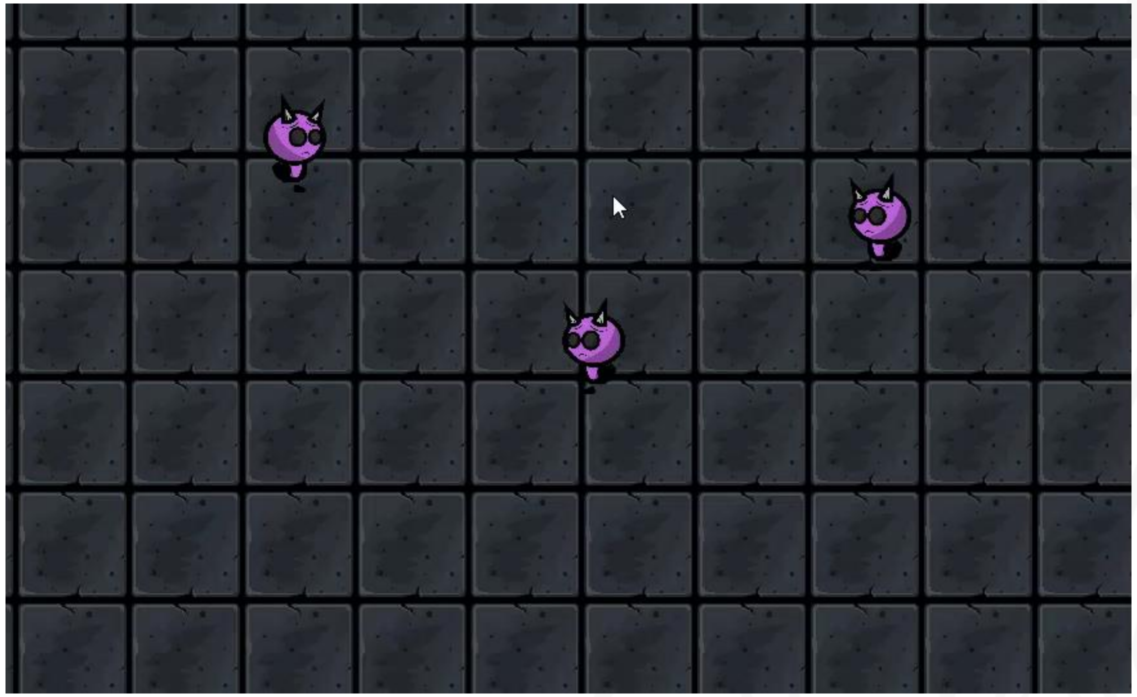
```
void OnCollisionEnter2D(Collision2D other) {
    if(other.gameObject.CompareTag("Enemy") && !ActualRool){
        Rigidbody2D rbE = other.gameObject.GetComponent<Rigidbody2D>();
        if (rbE != null) {
            Vector2 forceDirection = (other.transform.position - transform.position).normalized;
            rbE.velocity = forceDirection * 3;
            AIPath aiPath = other.gameObject.GetComponent<AIPath>();
            CapsuleCollider2D collider = other.gameObject.GetComponent<CapsuleCollider2D>();
            StartCoroutine(DisableAIMovement(collider, aiPath, 0.5f));
        }
    }
    if(other.gameObject.CompareTag("Weapon") && !ActualRool){
        weaponSwitch(other.gameObject);
    }
}

1 reference
IEnumerator DisableAIMovement(CapsuleCollider2D collider,AIPath aiPath, float delay) {
    collider.enabled = false;
    aiPath.enabled = false;
    yield return new WaitForSeconds(delay);
    aiPath.enabled = true;
    collider.enabled = true;
}
```

Enemy Follow Player problem

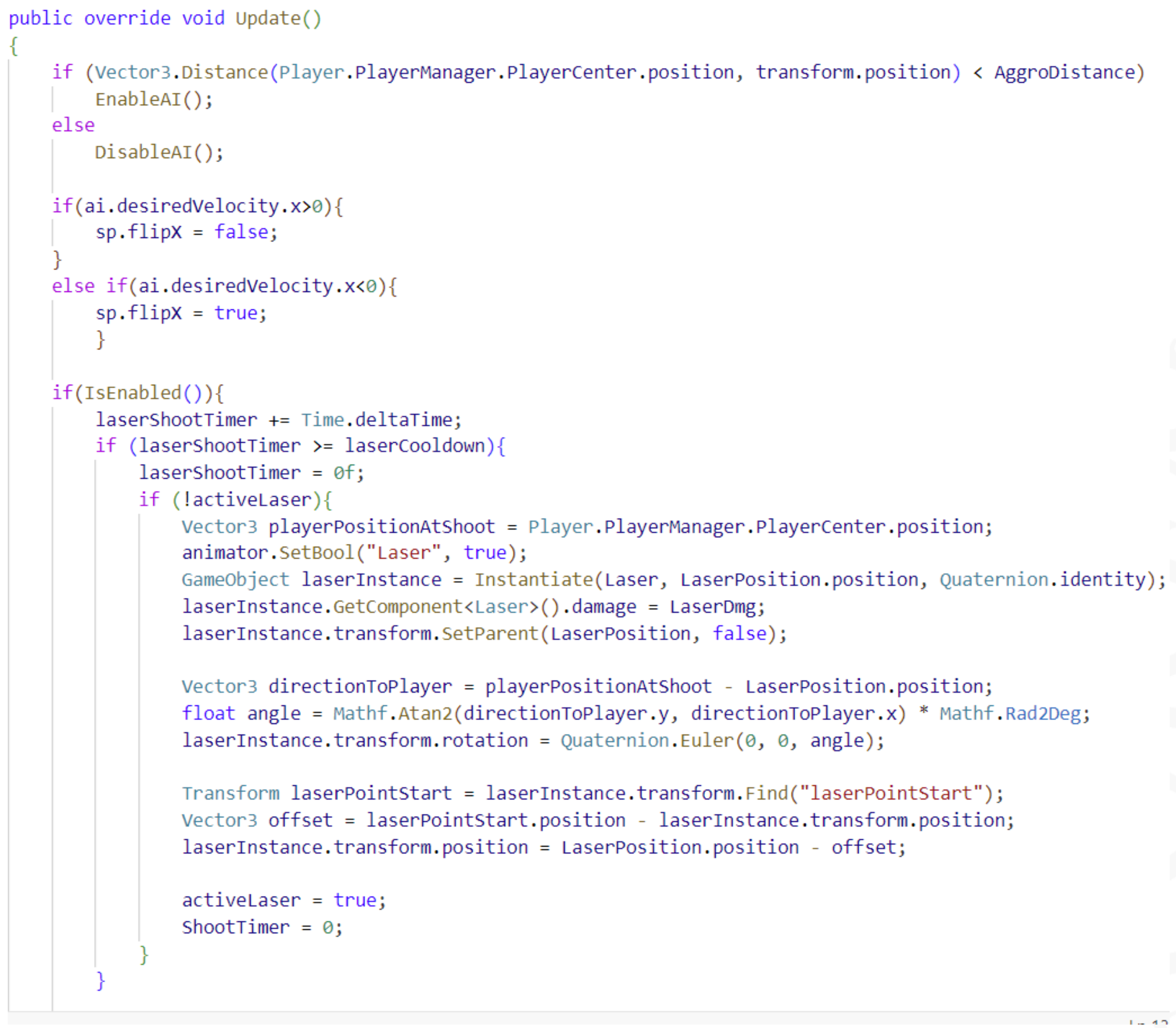
- Another issue encountered is that, over time, if the player moves and the enemies follow, the enemies start to follow the **exact same path** and begin to **overlap**.
- **Solution:** A 2D capsule collider was added to the enemies, which triggers collisions only with other enemies. This has resulted in much smoother, more pleasant, and realistic movement between them.



Final Result with all this improvements



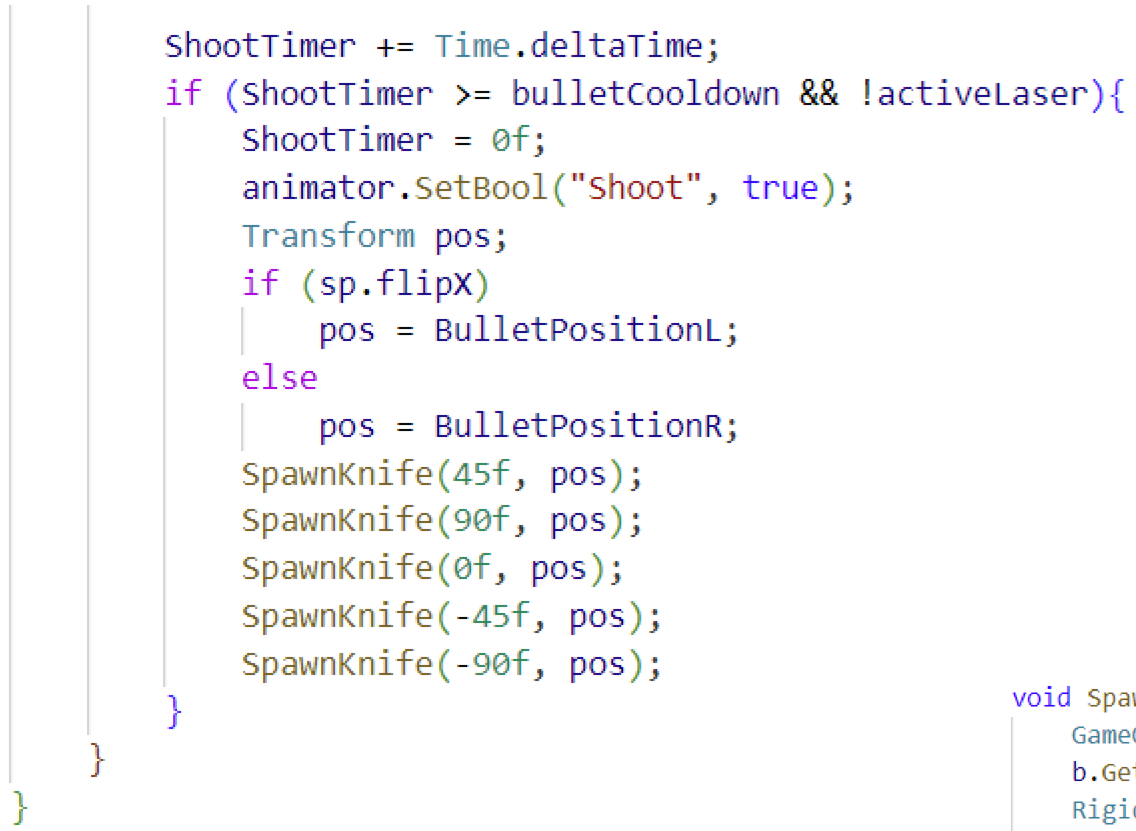
Rock-Boss (1/2)



```
public override void Update()
{
    if (Vector3.Distance(Player.PlayerManager.PlayerCenter.position, transform.position) < AggroDistance)
        EnableAI();
    else
        DisableAI();

    if(ai.desiredVelocity.x>0){
        sp.flipX = false;
    }
    else if(ai.desiredVelocity.x<0){
        sp.flipX = true;
        }

    if(IsEnabled()){
        laserShootTimer += Time.deltaTime;
        if (laserShootTimer >= laserCooldown){
            laserShootTimer = 0f;
            if (!activeLaser){
                Vector3 playerPositionAtShoot = Player.PlayerManager.PlayerCenter.position;
                animator.SetBool("Laser", true);
                GameObject laserInstance = Instantiate(Laser, LaserPosition.position, Quaternion.identity);
                laserInstance.GetComponent<Laser>().damage = LaserDmg;
                laserInstance.transform.SetParent(LaserPosition, false);

                Vector3 directionToPlayer = playerPositionAtShoot - LaserPosition.position;
                float angle = Mathf.Atan2(directionToPlayer.y, directionToPlayer.x) * Mathf.Rad2Deg;
                laserInstance.transform.rotation = Quaternion.Euler(0, 0, angle);

                Transform laserPointStart = laserInstance.transform.Find("laserPointStart");
                Vector3 offset = laserPointStart.position - laserInstance.transform.position;
                laserInstance.transform.position = LaserPosition.position - offset;

                activeLaser = true;
                ShootTimer = 0;
            }
        }
```

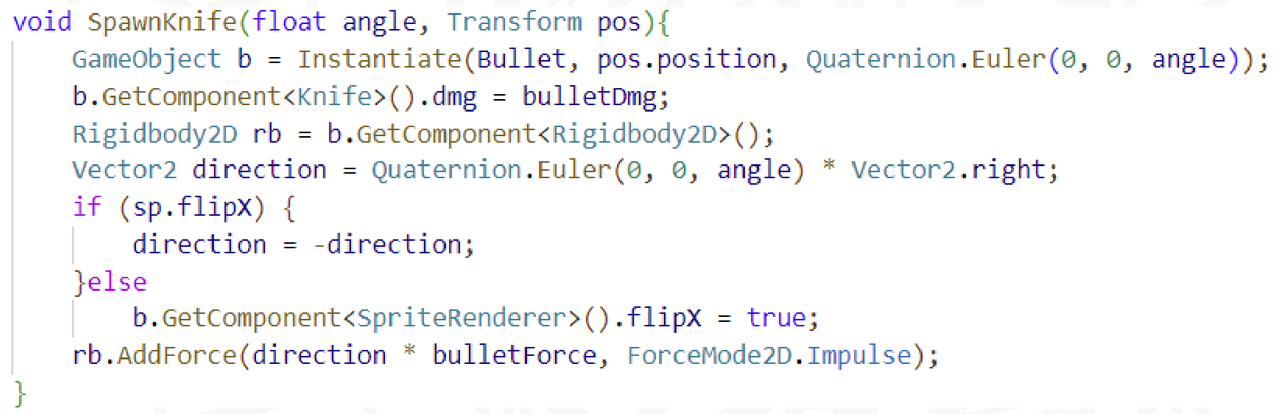
- This boss has two additional special attacks beyond the basic melee attack.
- The first is a **laser beam** that targets the player and fires every 6 seconds.

Rock-Boss (2/2)



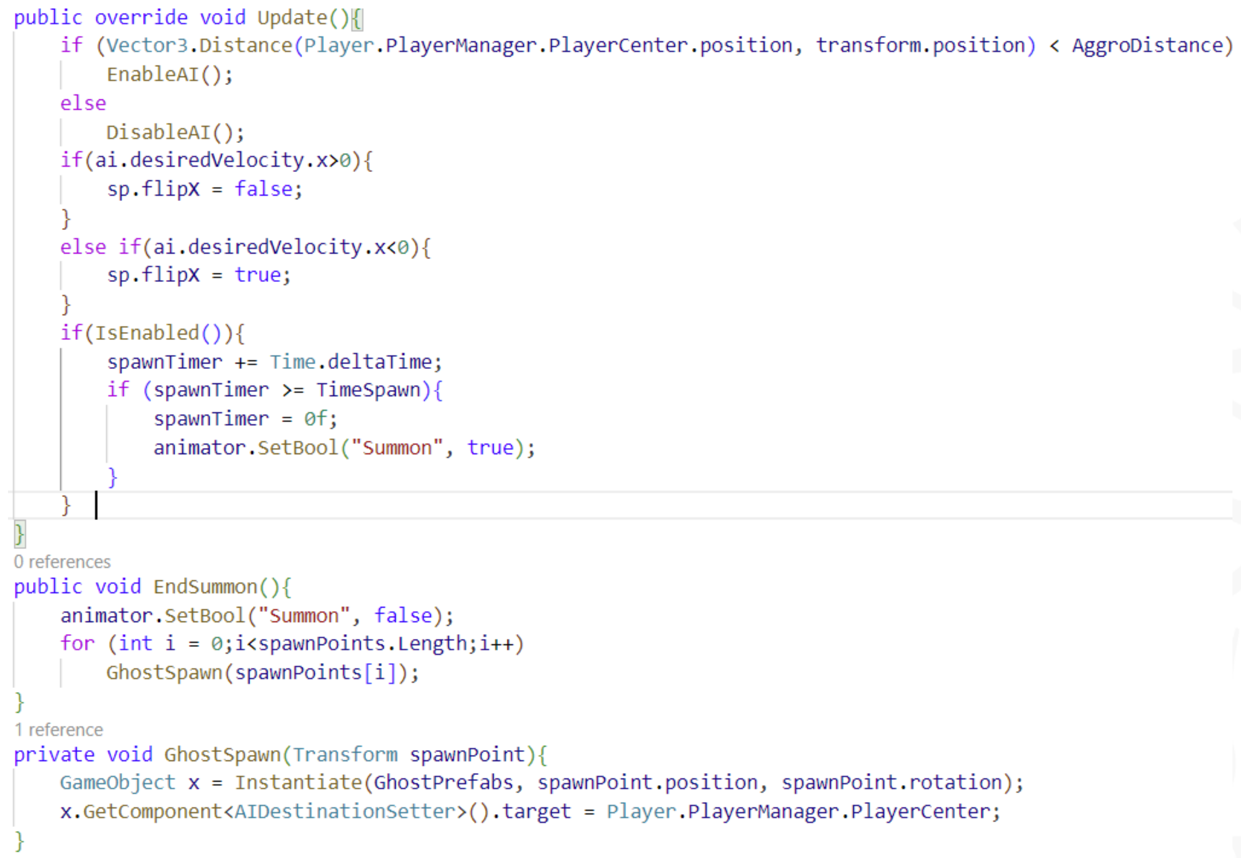
```
        ShootTimer += Time.deltaTime;
        if (ShootTimer >= bulletCooldown && !activeLaser){
            ShootTimer = 0f;
            animator.SetBool("Shoot", true);
            Transform pos;
            if (sp.flipX)
                pos = BulletPositionL;
            else
                pos = BulletPositionR;
            SpawnKnife(45f, pos);
            SpawnKnife(90f, pos);
            SpawnKnife(0f, pos);
            SpawnKnife(-45f, pos);
            SpawnKnife(-90f, pos);
        }
    }
}
```

- The second special attack involves **launching 5 knives** either to the right or to the left, each at a different angle.



```
void SpawnKnife(float angle, Transform pos){
    GameObject b = Instantiate(Bullet, pos.position, Quaternion.Euler(0, 0, angle));
    b.GetComponent<Knife>().dmg = bulletDmg;
    Rigidbody2D rb = b.GetComponent<Rigidbody2D>();
    Vector2 direction = Quaternion.Euler(0, 0, angle) * Vector2.right;
    if (sp.flipX) {
        direction = -direction;
    }else
        b.GetComponent<SpriteRenderer>().flipX = true;
    rb.AddForce(direction * bulletForce, ForceMode2D.Impulse);
}
```

UnDead Boss



```
public override void Update(){
    if (Vector3.Distance(Player.PlayerManager.PlayerCenter.position, transform.position) < AggroDistance)
        EnableAI();
    else
        DisableAI();
    if(ai.desiredVelocity.x>0){
        sp.flipX = false;
    }
    else if(ai.desiredVelocity.x<0){
        sp.flipX = true;
    }
    if(IsEnabled()){
        spawnTimer += Time.deltaTime;
        if (spawnTimer >= TimeSpawn){
            spawnTimer = 0f;
            animator.SetBool("Summon", true);
        }
    }
}
0 references
public void EndSummon(){
    animator.SetBool("Summon", false);
    for (int i = 0;i<spawnPoints.Length;i++)
        GhostSpawn(spawnPoints[i]);
}
1 reference
private void GhostSpawn(Transform spawnPoint){
    GameObject x = Instantiate(GhostPrefabs, spawnPoint.position, spawnPoint.rotation);
    x.GetComponent<AIDestinationSetter>().target = Player.PlayerManager.PlayerCenter;
}
```

- In addition to the melee attack, this boss **summons** 4 enemies (Ghosts) every 3 seconds to attack the player.
- The Ghosts are similar to the standard enemies but move on a separate A* grid that allows them to **pass over water**.

Additional Components for Enemies:

- All enemy types extend the abstract class `Enemy` and override the `Update()` and `Die()` methods. The `Update()` method contains the movement logic for the enemy, while the `Die()` method is customized for each enemy type to handle specific behaviors, such as dropping a key.
- All enemies have a chance to **drop** a special item, which is a healing potion that fully restores the player's health.
- For balancing purposes, Ghosts are enemies that yield very little experience when defeated.
- All enemies have different aggro distances.

Game saving

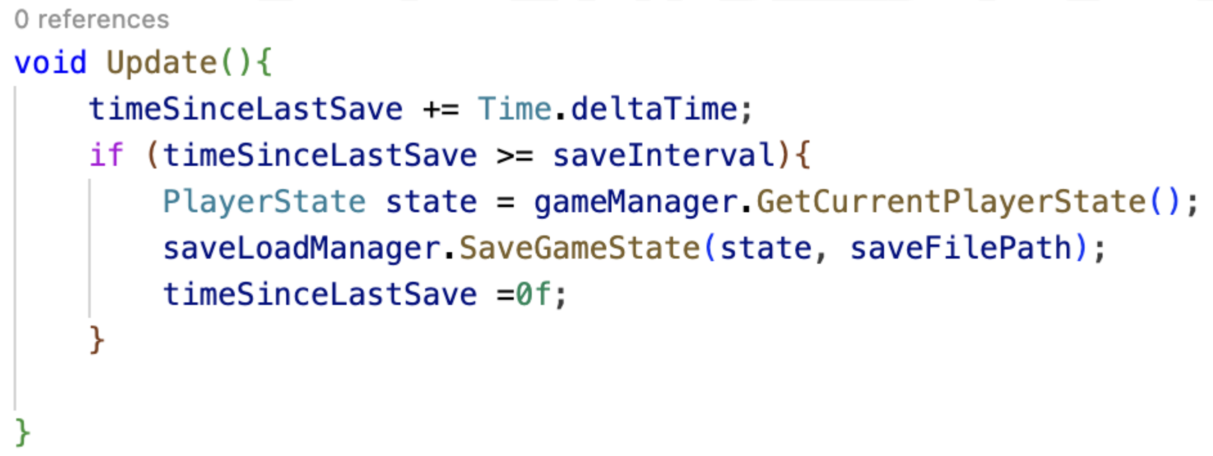
What is saved

The data saved in the game can be divided into two main groups: character statistics (such as HP, strength, position, power-ups, etc.) and scene elements (like door states, lever states, killed enemies, etc.).

All the data that are saved are part of a [System.Serializable] class called GameState

When is saved

The autosave features is a function that saves progress every 2 seconds. This design choice was made to prevent the use of checkpoints or manual saves as a "cheat", where players could easily restart from the last save if something went wrong while experimenting.



```
0 references
void Update(){
    timeSinceLastSave += Time.deltaTime;
    if (timeSinceLastSave >= saveInterval){
        PlayerState state = gameManager.GetCurrentPlayerState();
        saveLoadManager.SaveGameState(state, saveFilePath);
        timeSinceLastSave =0f;
    }

}
```

Where is saved

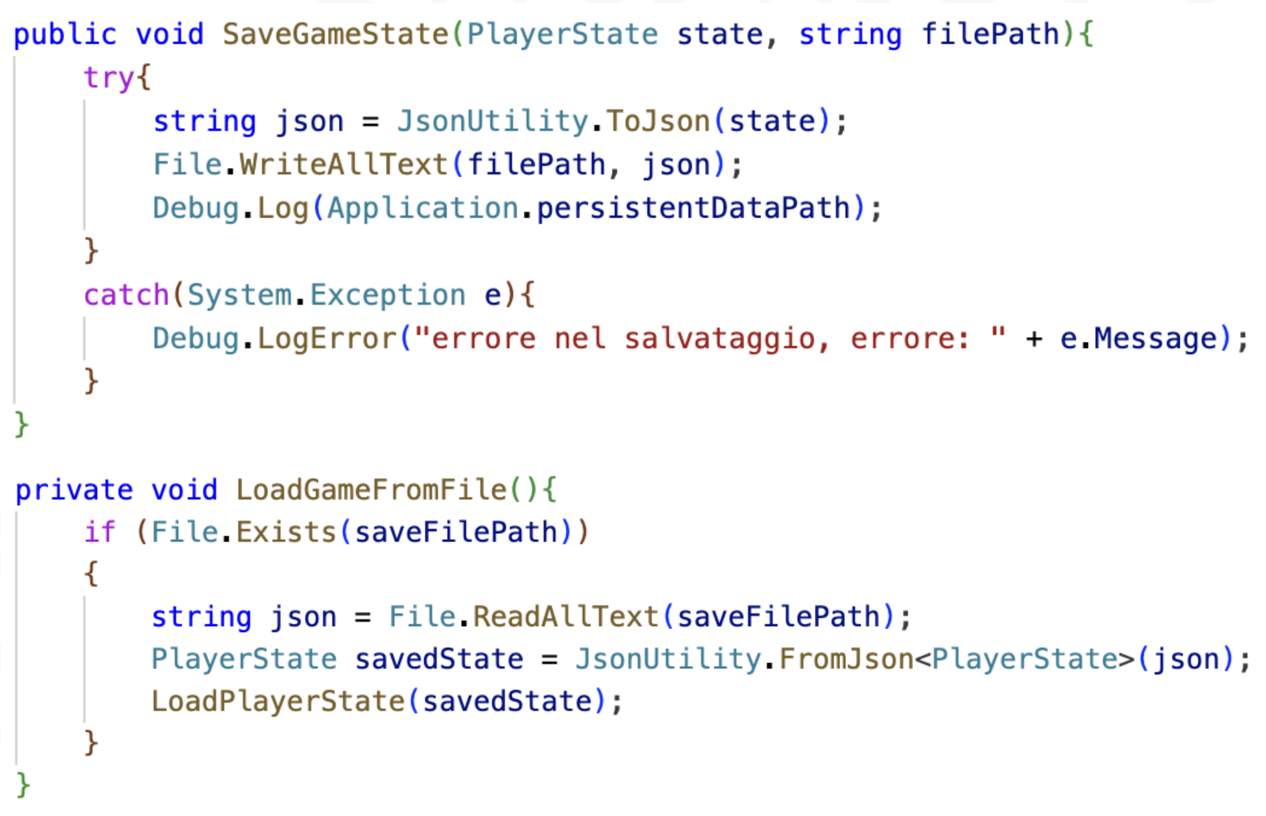
The features are saved in `Application.persistentDataPath`.

`Application.persistentDataPath` is a property in Unity that provides a path to a directory where you can store data that you want to persist between sessions. This directory is intended for saving files that need to be preserved across game runs, such as save files, configuration files, or user-generated content.

UNIVERSITÀ DEGLI STUDI FIRENZE

How is saved

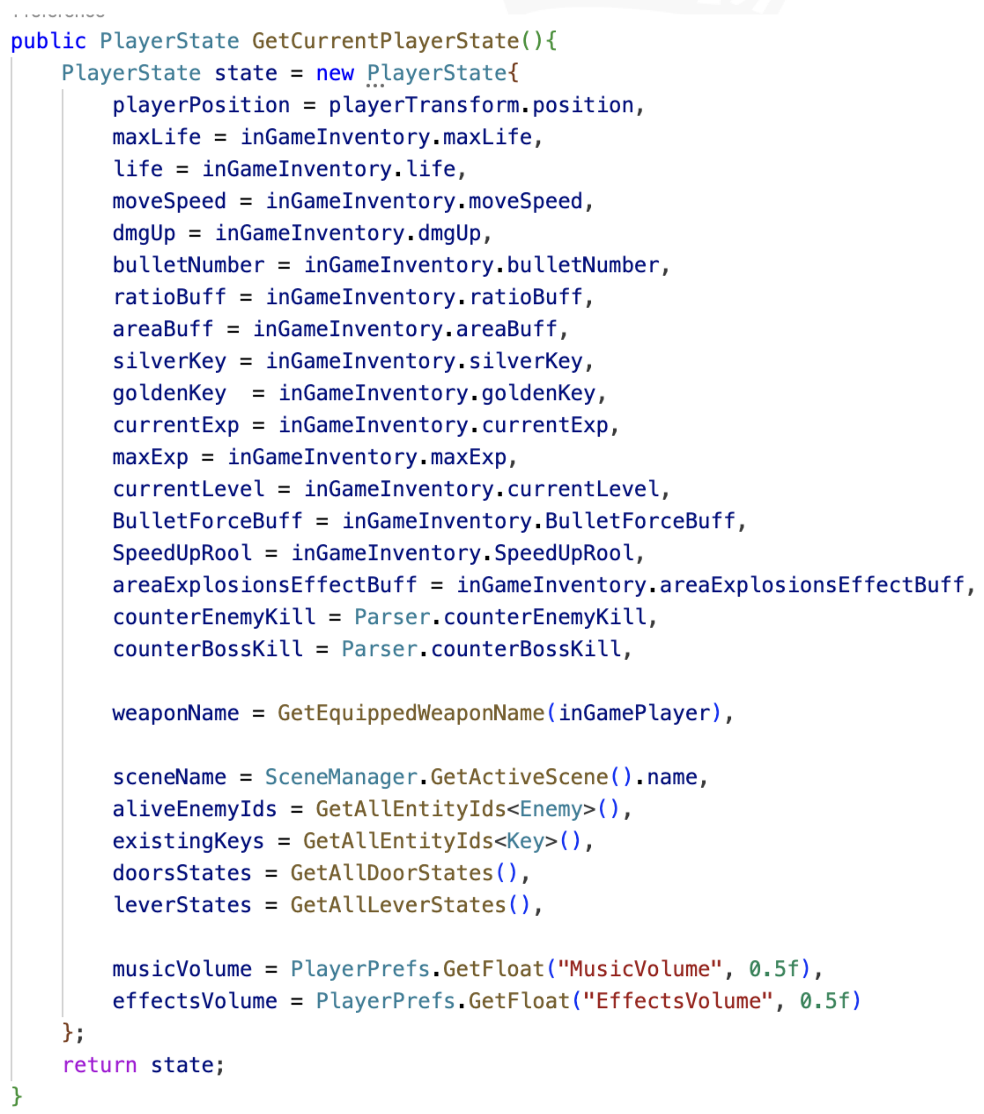
The `SaveState` class is converted into a JSON file and stored on disk, allowing it to be easily read and used to restore the game's last saved state.



```
public void SaveGameState(PlayerState state, string filePath){
    try{
        string json = JsonUtility.ToJson(state);
        File.WriteAllText(filePath, json);
        Debug.Log(Application.persistentDataPath);
    }
    catch(System.Exception e){
        Debug.LogError("errore nel salvataggio, errore: " + e.Message);
    }
}

private void LoadGameFromFile(){
    if (File.Exists(saveFilePath))
    {
        string json = File.ReadAllText(saveFilePath);
        PlayerState savedState = JsonUtility.FromJson<PlayerState>(json);
        LoadPlayerState(savedState);
    }
}
```

How is saved

The first function saves both the player's inventory state and the scene's state by invoking specific functions that handle the scene elements.

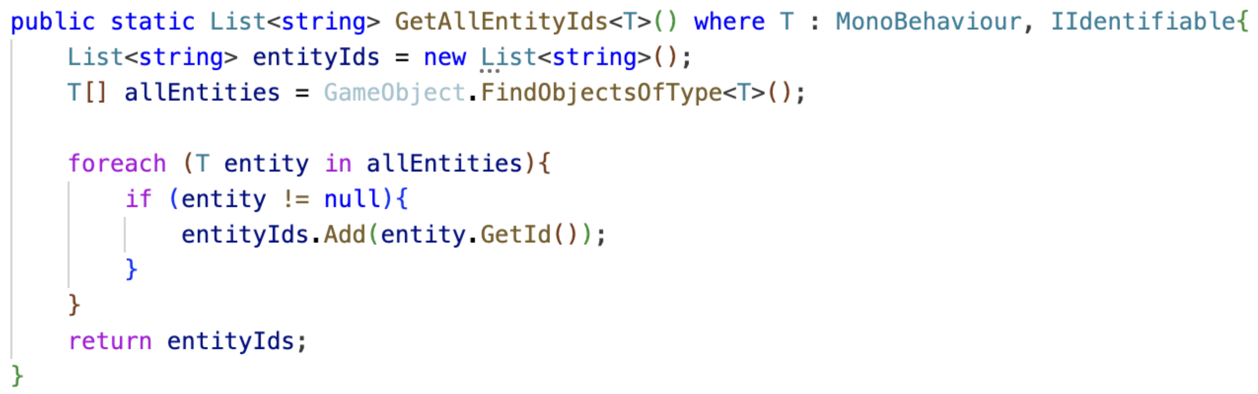


```
public PlayerState GetCurrentPlayerState(){
    PlayerState state = new PlayerState{
        playerPosition = playerTransform.position,
        maxLife = inGameInventory.maxLife,
        life = inGameInventory.life,
        moveSpeed = inGameInventory.moveSpeed,
        dmgUp = inGameInventory.dmgUp,
        bulletNumber = inGameInventory.bulletNumber,
        ratioBuff = inGameInventory.ratioBuff,
        areaBuff = inGameInventory.areaBuff,
        silverKey = inGameInventory.silverKey,
        goldenKey  = inGameInventory.goldenKey,
        currentExp = inGameInventory.currentExp,
        maxExp = inGameInventory.maxExp,
        currentLevel = inGameInventory.currentLevel,
        BulletForceBuff = inGameInventory.BulletForceBuff,
        SpeedUpRool = inGameInventory.SpeedUpRool,
        areaExplosionsEffectBuff = inGameInventory.areaExplosionsEffectBuff,
        counterEnemyKill = Parser.counterEnemyKill,
        counterBossKill = Parser.counterBossKill,

        weaponName = GetEquippedWeaponName(inGamePlayer),

        sceneName = SceneManager.GetActiveScene().name,
        aliveEnemyIds = GetAllEntityIds<Enemy>(),
        existingKeys = GetAllEntityIds<Key>(),
        doorsStates = GetAllDoorStates(),
        leverStates = GetAllLeverStates(),

        musicVolume = PlayerPrefs.GetFloat("MusicVolume", 0.5f),
        effectsVolume = PlayerPrefs.GetFloat("EffectsVolume", 0.5f)
    };
    return state;
}
```

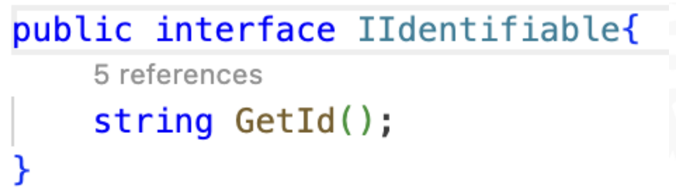
How is saved: GetAllEntityIds<T>()

This function returns a list of strings (IDs) for a specific GameObject passed as a parameter.

It can handle any GameObject that implements the `IIdentifiable` interface, allowing it to call the `GetId()` method.



```
public static List<string> GetAllEntityIds<T>() where T : MonoBehaviour, IIdentifiable{
    List<string> entityIds = new List<string>();
    T[] allEntities = GameObject.FindObjectsOfType<T>();

    foreach (T entity in allEntities){
        if (entity != null){
            entityIds.Add(entity.GetId());
        }
    }
    return entityIds;
}
```

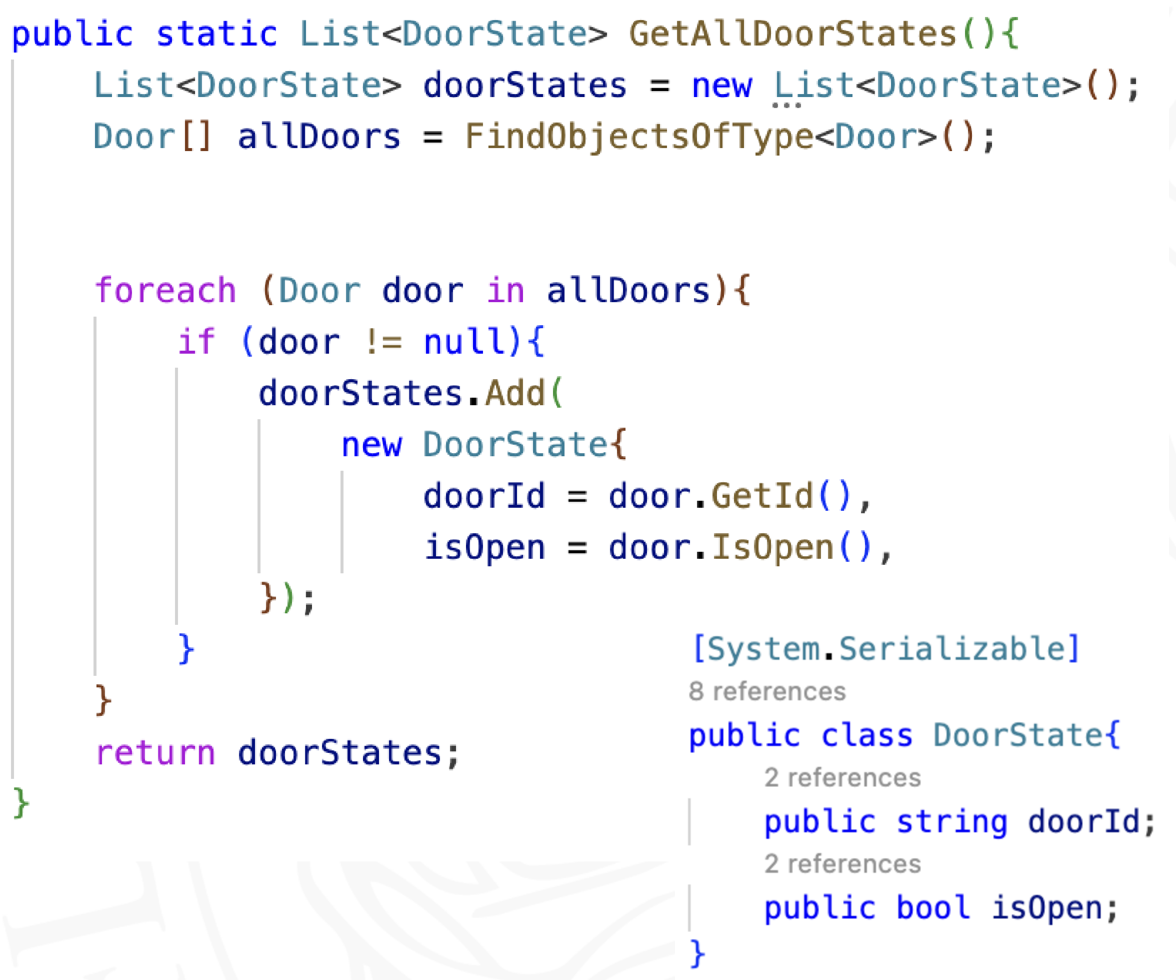


```
public interface IIdentifiable{
    5 references
    string GetId();
}
```

How is saved: GetAllDoor/LeverStates()

Similar to the previous function, this one returns a list of `DoorState` objects. Each `DoorState` contains a string `doorId` and a boolean `isOpen`, allowing the function to save the state of doors in the scene.

The same happens for Lever states.

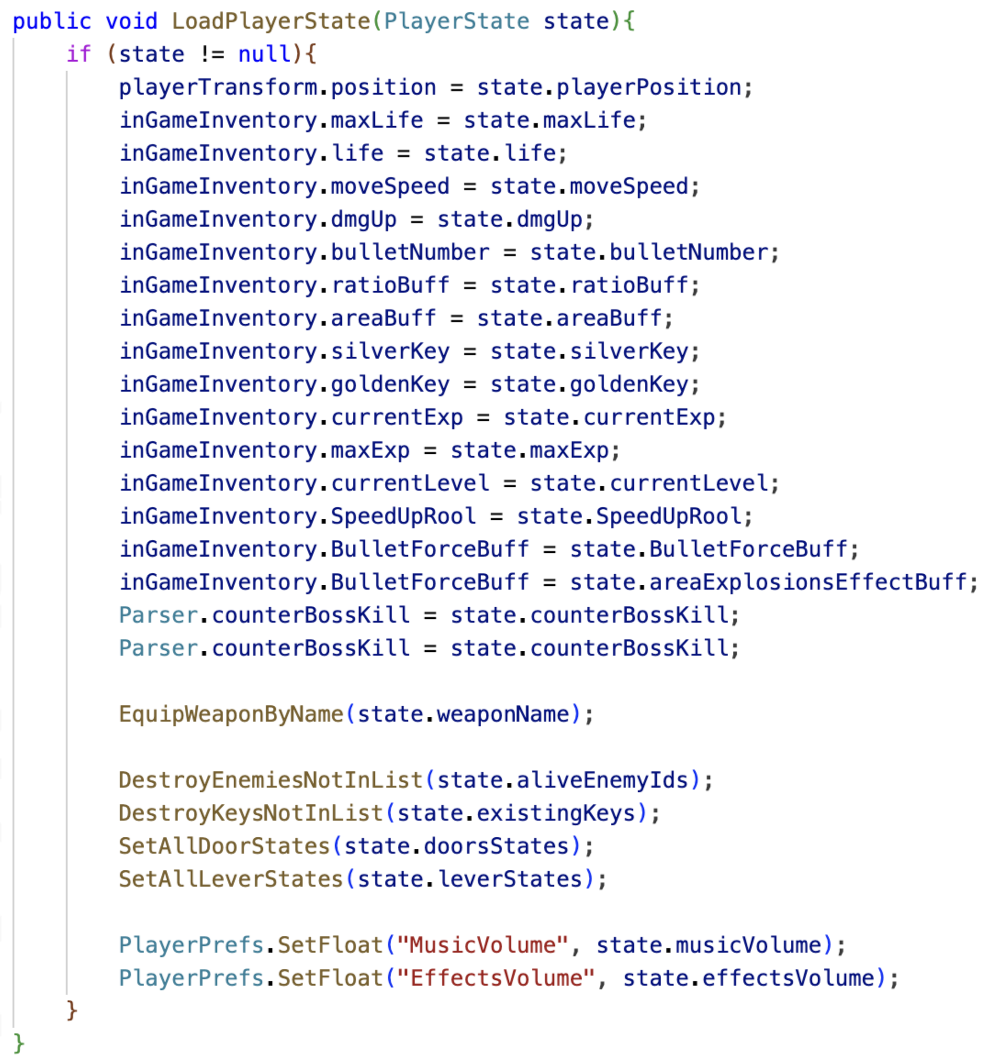


```
public static List<DoorState> GetAllDoorStates(){
    List<DoorState> doorStates = new List<DoorState>();
    Door[] allDoors = FindObjectsOfType<Door>();


    foreach (Door door in allDoors){
        if (door != null){
            doorStates.Add(
                new DoorState{
                    doorId = door.GetId(),
                    isOpen = door.IsOpen(),
                });
        }
    }
    return doorStates;
}
```

```
[System.Serializable]
8 references
public class DoorState{
    2 references
    public string doorId;
    2 references
    public bool isOpen;
}
```

How is loaded

Similar to the saving process, loading is divided into two parts: one that restores the player's inventory and another that calls specific functions to handle the loading of the scene elements. This ensures that both the player's state and the scene's state are properly restored.

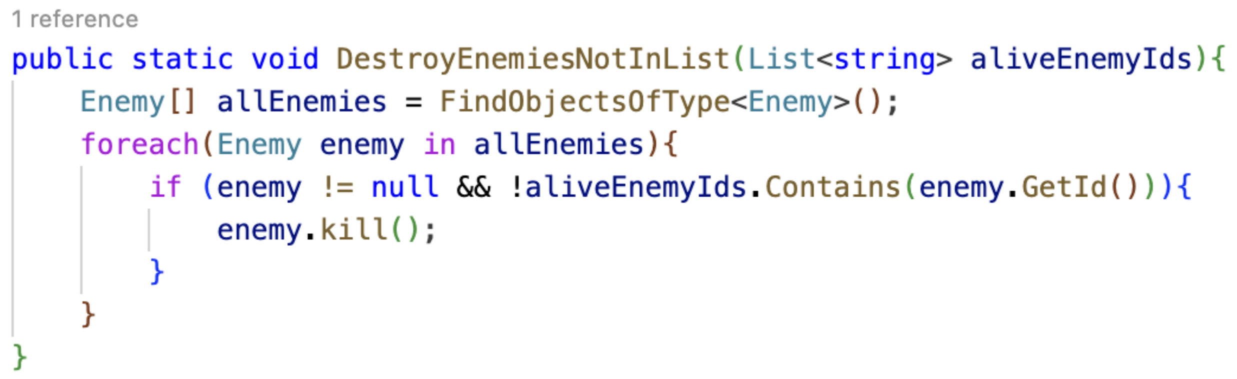


```
public void LoadPlayerState(PlayerState state){
    if (state != null){
        playerTransform.position = state.playerPosition;
        inGameInventory.maxLife = state.maxLife;
        inGameInventory.life = state.life;
        inGameInventory.moveSpeed = state.moveSpeed;
        inGameInventory.dmgUp = state.dmgUp;
        inGameInventory.bulletNumber = state.bulletNumber;
        inGameInventory.ratioBuff = state.ratioBuff;
        inGameInventory.areaBuff = state.areaBuff;
        inGameInventory.silverKey = state.silverKey;
        inGameInventory.goldenKey = state.goldenKey;
        inGameInventory.currentExp = state.currentExp;
        inGameInventory.maxExp = state.maxExp;
        inGameInventory.currentLevel = state.currentLevel;
        inGameInventory.SpeedUpRool = state.SpeedUpRool;
        inGameInventory.BulletForceBuff = state.BulletForceBuff;
        inGameInventory.BulletForceBuff = state.areaExplosionsEffectBuff;
        Parser.counterBossKill = state.counterBossKill;
        Parser.counterBossKill = state.counterBossKill;

        EquipWeaponByName(state.weaponName);

        DestroyEnemiesNotInList(state.aliveEnemyIds);
        DestroyKeysNotInList(state.existingKeys);
        SetAllDoorStates(state.doorsStates);
        SetAllLeverStates(state.leverStates);

        PlayerPrefs.SetFloat("MusicVolume", state.musicVolume);
        PlayerPrefs.SetFloat("EffectsVolume", state.effectsVolume);
    }
}
```

How is loaded: DestroyEnemiesNotInList()

This function loops through all the enemies in the scene and eliminates any enemy that was killed since the last save by comparing their IDs with the list of enemies that were still alive. This ensures the game correctly reflects the state of defeated enemies.

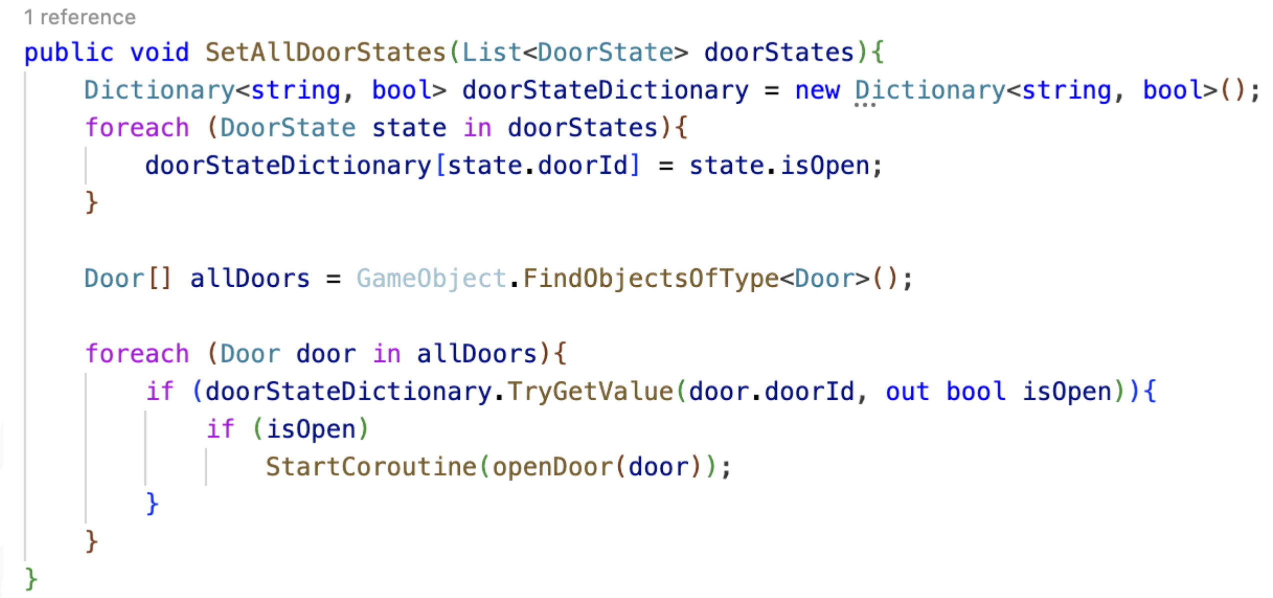
Same happens for keys.


1 reference
```
public static void DestroyEnemiesNotInList(List<string> aliveEnemyIds){
    Enemy[] allEnemies = FindObjectsOfType<Enemy>();
    foreach(Enemy enemy in allEnemies){
        if (enemy != null && !aliveEnemyIds.Contains(enemy.GetId())){
            enemy.kill();
        }
    }
}
```

How is loaded: SetAllDoorStates()

This function follows the same logic as the previous one but also triggers the animations to open doors that were opened in the last save. This ensures that door states are accurately restored along with their animations.

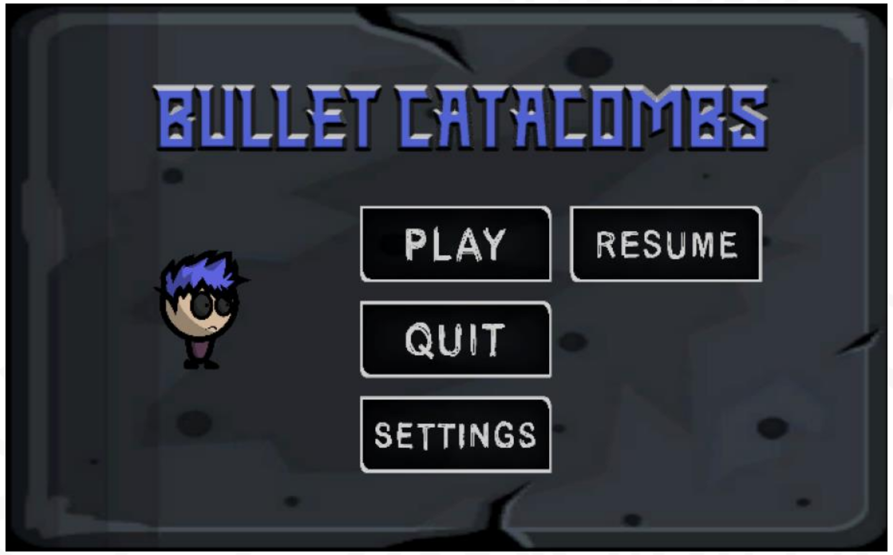
Same happens for Lever and Gates.

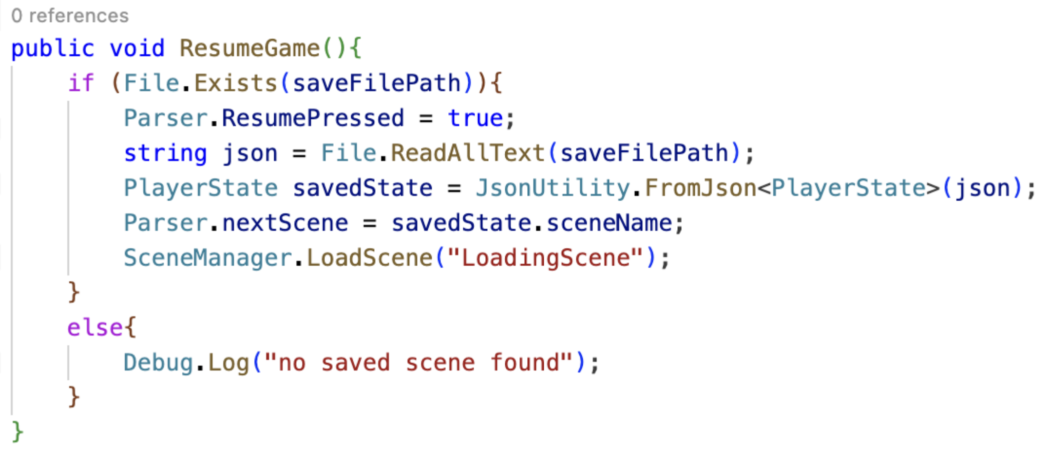


```
1 reference
public void SetAllDoorStates(List<DoorState> doorStates){
    Dictionary<string, bool> doorStateDictionary = new Dictionary<string, bool>();
    foreach (DoorState state in doorStates){
        doorStateDictionary[state.doorId] = state.isOpen;
    }

    Door[] allDoors = GameObject.FindObjectsOfType<Door>();

    foreach (Door door in allDoors){
        if (doorStateDictionary.TryGetValue(door.doorId, out bool isOpen)){
            if (isOpen)
                StartCoroutine(openDoor(door));
        }
    }
}
```

MainMenu updating

A final detail worth mentioning is that the main menu features a dynamic "Resume" button, which is only displayed if a save from a previous game session exists.





```
0 references
public void ResumeGame(){
    if (File.Exists(saveFilePath)){
        Parser.ResumePressed = true;
        string json = File.ReadAllText(saveFilePath);
        PlayerState savedState = JsonUtility.FromJson<PlayerState>(json);
        Parser.nextScene = savedState.sceneName;
        SceneManager.LoadScene("LoadingScene");
    }
    else{
        Debug.Log("no saved scene found");
    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

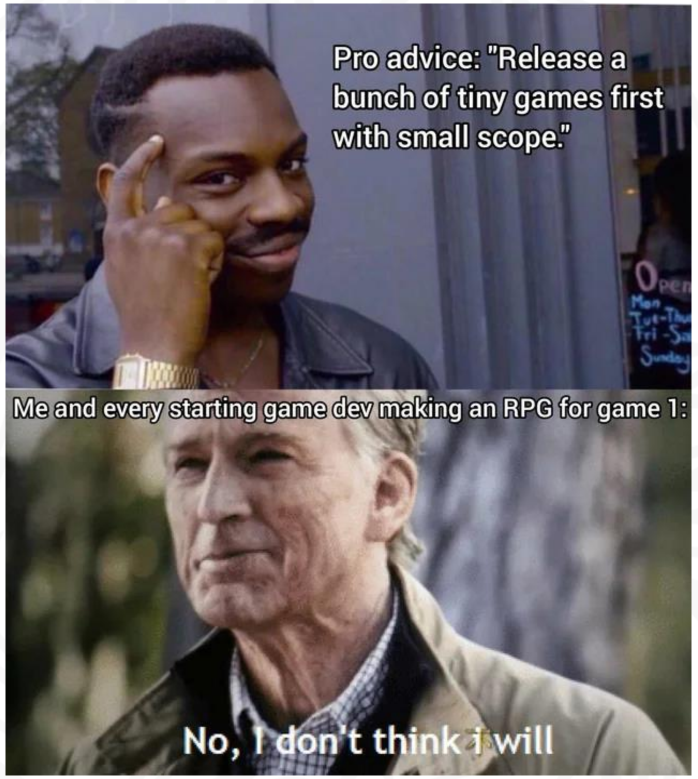
Conclusions

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusions

- In conclusion, we aimed to create a game that is as comprehensive as possible, covering all aspects and incorporating the standard features expected in modern games.
- Additionally, we have worked to balance the game as much as possible in terms of enemies, damage, and buffs, in order to enhance the player's overall experience.

WHEN YOU HAVE TO FIND OUT WHY YOUR CODE HAS AN ERROR
This little maneuver is gonna cost us 51 years
imgflip.com


Pro advice: "Release a bunch of tiny games first with small scope."
Me and every starting game dev making an RPG for game 1:
No, I don't think i will

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Thank you for your attention!

TDS Game: Bullet Catacomb

Mirko Bicchierai and Filippo Di Martino

Course: Game Development.