# 2D Samurai Game

Documentation by the developer, Ata Eren Arslan

## About the developer

I am a 23-year-old Mechatronics Engineering graduate who works as a software developer, as of the time of writing this documentation, working on a web application for a company. Had worked on modding an open-source Unity 3D game as a hobby before, so have some experience with the Unity Game Engine as well.

## Why did I make this?

I've been interested in the game industry for years but never really did anything about it until I started working on AoTTG2:EE, so I decided I want to make something that is going to actually teach me the ropes. Working on that project was very fun and challenging for me but after a while, I realized that modding a game is not the same as making one. I wanted to learn the ropes from scratch and also have something to show for it as "experience", so I decided to make small games. This project serves as the beginning of that journey.

## What to expect from this game?

The expectation should be to see someone who is in the process of learning game development on their own. This game by no means claims to be fun or anything remotely associated with the concept of fun. I know that this goes strictly against the concept of a "game" but please bear with me and let me explain what I mean by this: **This game is made as a way to learn the ropes, get more familiar with the core concepts of Unity 2D and game development; learning the hardships of making a game, encountering problems that I could not have seen before and *learning from this entire process*.**

With that out of the way, below you will find a detailed mechanical breakdown and visualization of the game.

# General Overview

The game is a physics based 2D action-platformer, made of a single actual level. The objective is to clear the area of enemies and reach the end of the level. The core mechanics could be listed as below:

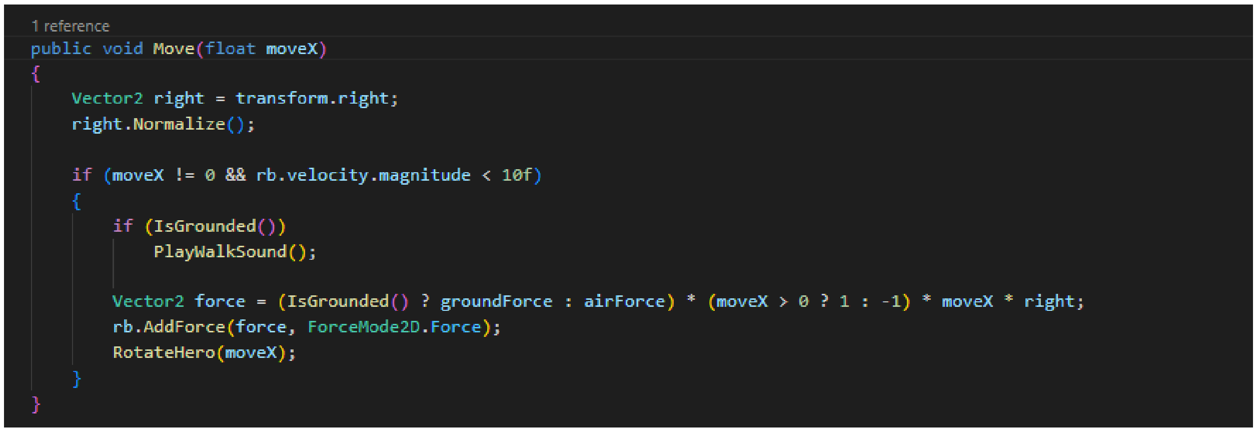
- **Running**
- **Jumping**
- **Wall Jumping**
- **Double Jumping**
- **Jumping off Enemies**
- **Attacking**

# Running

Running was the first mechanic developed when starting the project. As the game will be physics based, it's easily predictable that the movement will be controlled by the physics engine, which is mainly used to apply force on the x-axis for horizontal movement.

The inputs are taken in a custom Character Controller script. They are always supposed to be taken from the ***Update*** method, which ensures that there are no missed inputs by reading inputs every frame. Physics based movement calculations like adding force to a gameobject, however, should be made in ***FixedUpdate*** as these are completely frame rate independent.

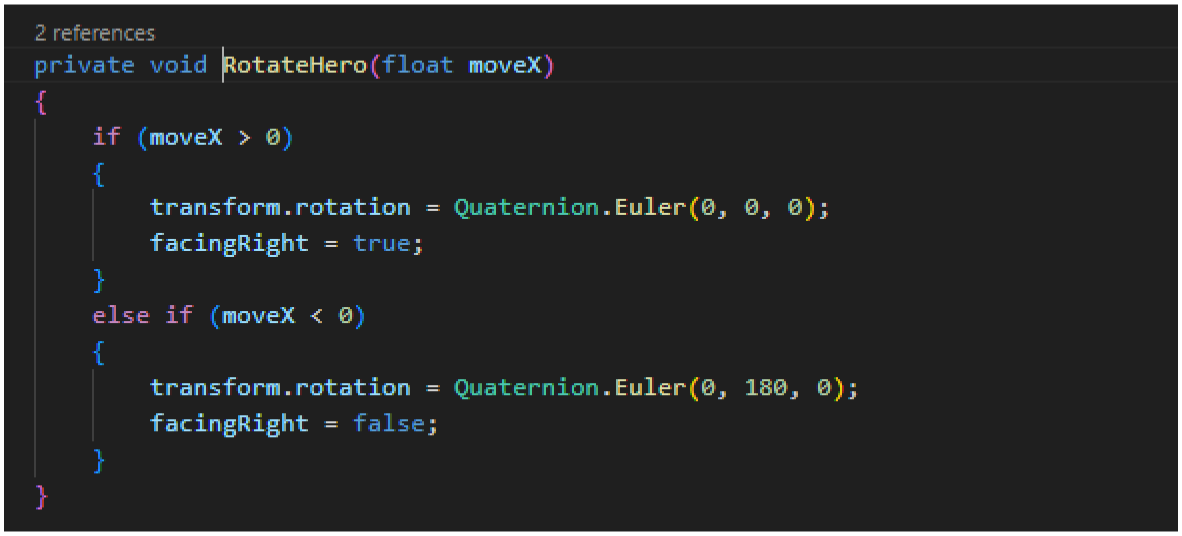
This was where the first problem occurred. When I took inputs from the ***Update*** method and tried to apply the effects of these inputs in the ***FixedUpdate*** method, the hero game object started behaving inconsistently. Therefore, I've created a buffer method that assigns data to another buffer variable in the ***Update*** method. The data from the buffered variable is read in the ***FixedUpdate*** method, which calls the "***Move()"*** method I use to move my character.



```
1 reference
public void Move(float moveX)
{
    Vector2 right = transform.right;
    right.Normalize();

    if (moveX != 0 && rb.velocity.magnitude < 10f)
    {
        if (IsGrounded())
            PlayWalkSound();

        Vector2 force = (IsGrounded() ? groundForce : airForce) * (moveX > 0 ? 1 : -1) * moveX * right;
        rb.AddForce(force, ForceMode2D.Force);
        RotateHero(moveX);
    }
}
```

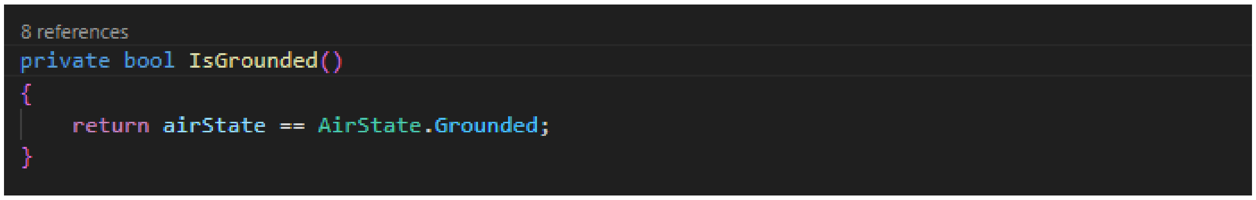
There are some other things going on in this code, which I will talk about later but the main part here is adding the normalized force to the ***Rigidbody2D*** component of the hero game object.

You can also see a ***"RotateHero"*** method here, which has a self explanatory name.



```
2 references
private void RotateHero(float moveX)
{
    if (moveX > 0)
    {
        transform.rotation = Quaternion.Euler(0, 0, 0);
        facingRight = true;
    }
    else if (moveX < 0)
    {
        transform.rotation = Quaternion.Euler(0, 180, 0);
        facingRight = false;
    }
}
```

Another method, ***"IsGrounded"***, is as follows:

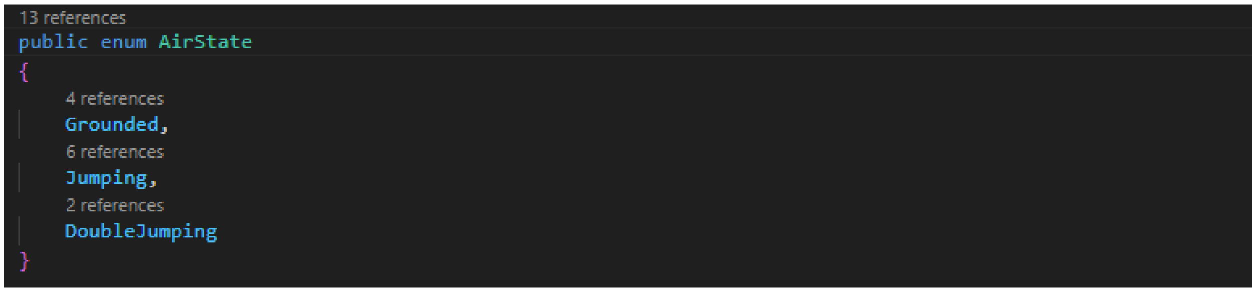


```
8 references
private bool IsGrounded()
{
    return airState == AirState.Grounded;
}
```

This returns an ***"airState"*** variable value check, which carries us to the breakdown of the next mechanic.

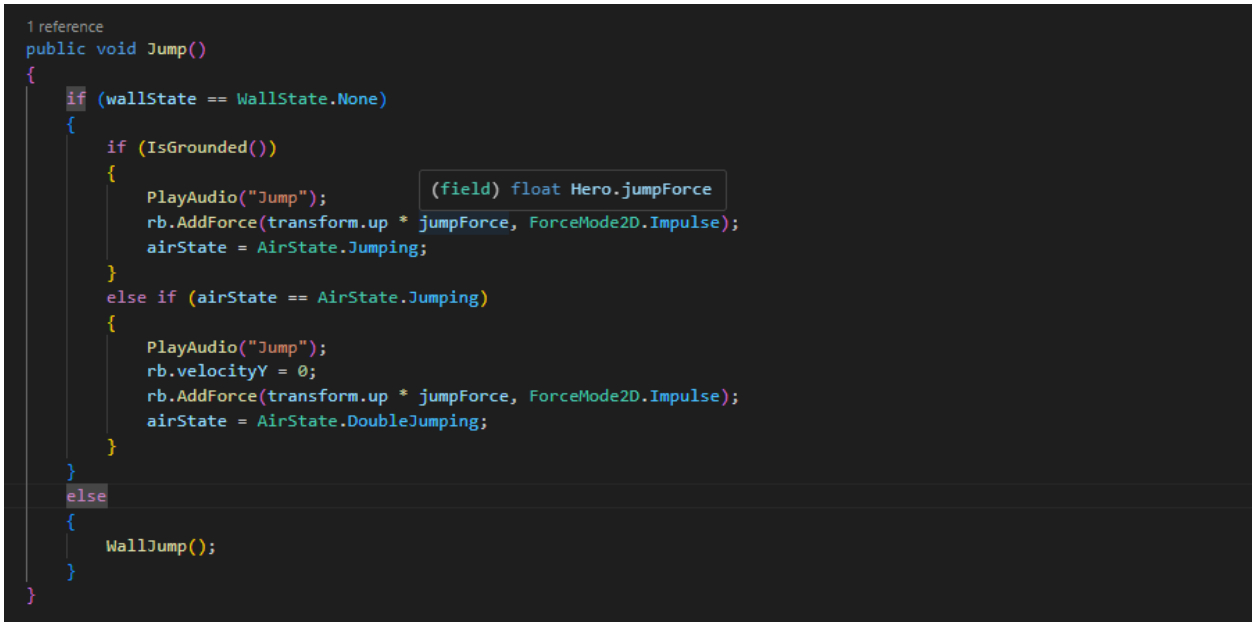
## Jumping

Jumping is mainly controlled by an ***"airState"*** variable, which is of value of an enum I created. If I wanted to make a basic, one-jump-only mechanic, I could just use a "grounded check", but I wanted to be able to make use of double jumps, so this seemed like a suitable way to achieve that.



```
13 references
public enum AirState
{
    4 references
    Grounded,
    6 references
    Jumping,
    2 references
    DoubleJumping
}
```

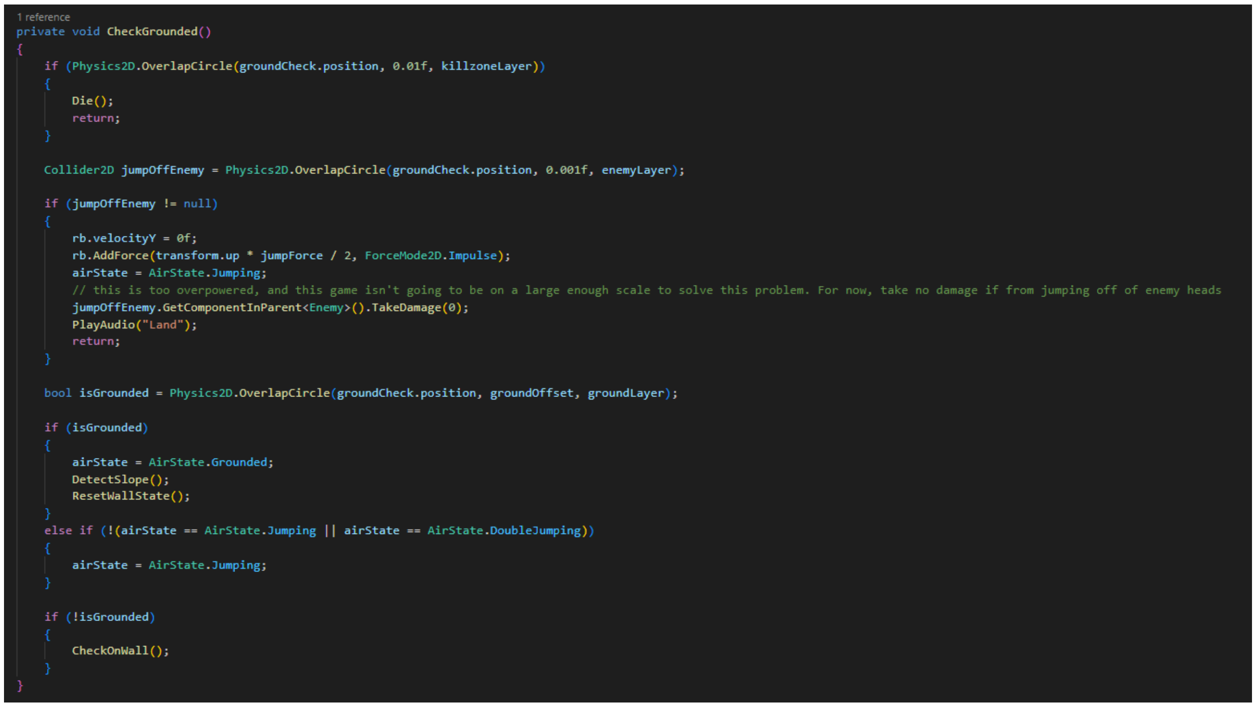
Above are the values that ***"airState"*** could take.



```
1 reference
public void Jump()
{
    if (wallState == WallState.None)
    {
        if (IsGrounded())
        {                           (field) float Hero.jumpForce
            PlayAudio("Jump");
            rb.AddForce(transform.up * jumpForce, ForceMode2D.Impulse);
            airState = AirState.Jumping;
        }
        else if (airState == AirState.Jumping)
        {
            PlayAudio("Jump");
            rb.velocityY = 0;
            rb.AddForce(transform.up * jumpForce, ForceMode2D.Impulse);
            airState = AirState.DoubleJumping;
        }
    }
    else
    {
        WallJump();
    }
}
```

The code above first checks for a ***"wallState"*** variable value check, which is a very similar enum I created for the wall jump functionality. I will dive into this right after the basic jumping mechanism.

The code is pretty simple: It checks if ***"airState"*** is equal to Grounded; if Grounded, it jumps for the first time. Then it checks if it's Jumping; if Jumping, it jumps for the second time. The one nuance here is setting the y-axis velocity to 0 before the second jump, as the double jump should first reset all the physics effects that have been applied by the first jump; or fall.

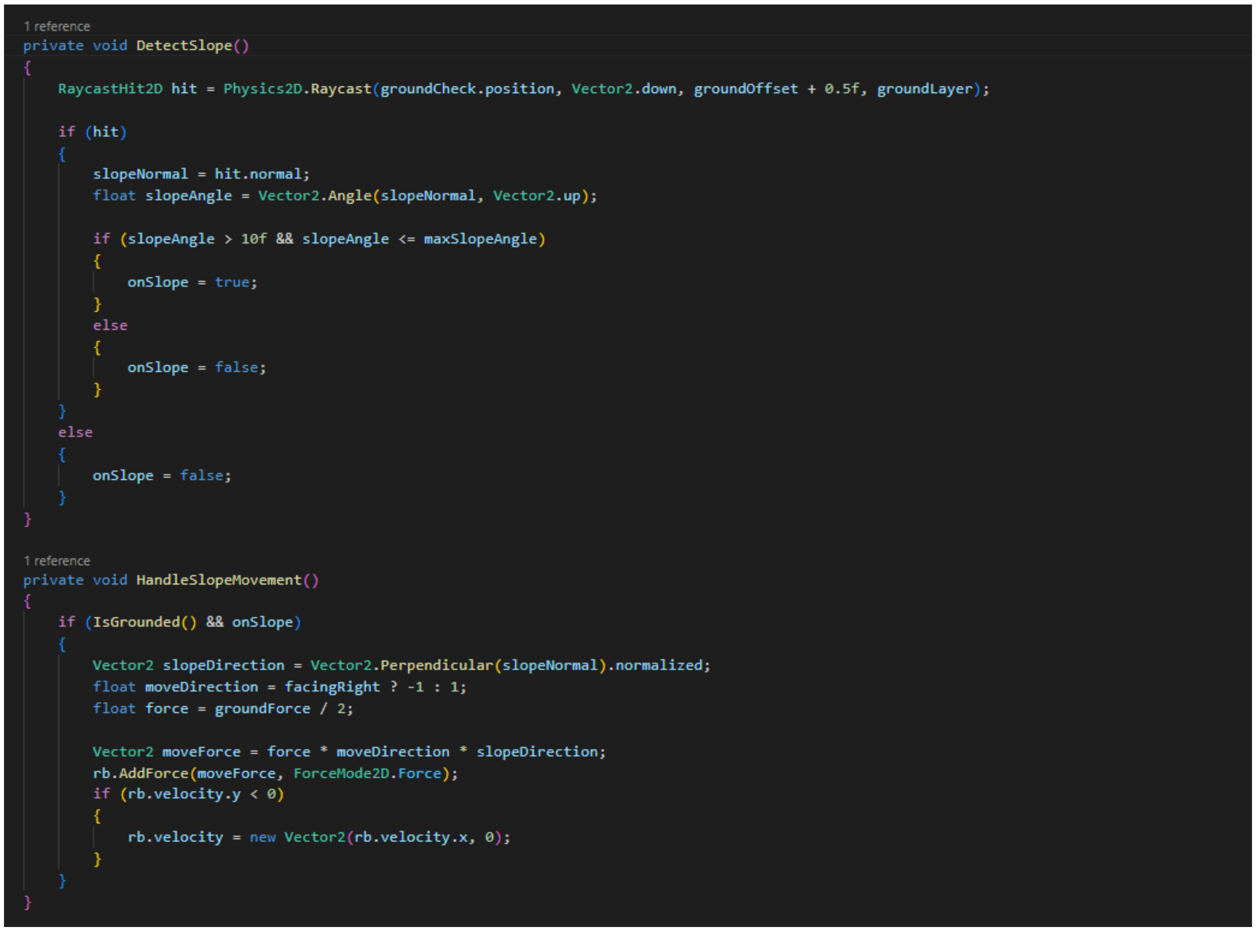


```
1 reference
private void CheckGrounded()
{
    if (Physics2D.OverlapCircle(groundCheck.position, 0.01f, killzoneLayer))
    {
        Die();
        return;
    }

    Collider2D jumpOffEnemy = Physics2D.OverlapCircle(groundCheck.position, 0.001f, enemyLayer);

    if (jumpOffEnemy != null)
    {
        rb.velocityY = 0f;
        rb.AddForce(transform.up * jumpForce / 2, ForceMode2D.Impulse);
        airState = AirState.Jumping;
        // this is too overpowered, and this game isn't going to be on a large enough scale to solve this problem. For now, take no damage if from jumping off of enemy heads
        jumpOffEnemy.GetComponentInParent<Enemy>().TakeDamage(0);
        PlayAudio("Land");
        return;
    }

    bool isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundOffset, groundLayer);

    if (isGrounded)
    {
        airState = AirState.Grounded;
        DetectSlope();
        ResetWallState();
    }
    else if (!(airState == AirState.Jumping || airState == AirState.DoubleJumping))
    {
        airState = AirState.Jumping;
    }

    if (!isGrounded)
    {
        CheckOnWall();
    }
}
```

The method above drives a lot of the movement functionality. It runs on ***FixedUpdate*** and determines the entire behavior. The first if check is pretty simple: If the player collides with a ***killZone,*** they die.

The second if check makes the player jump off an enemy if the ground check object collides with the head of an enemy. At first, this applied some minimal damage but on my play tests, I realized that this ability can very easily be abused. So I decided to make it only that the enemy is stunned briefly.

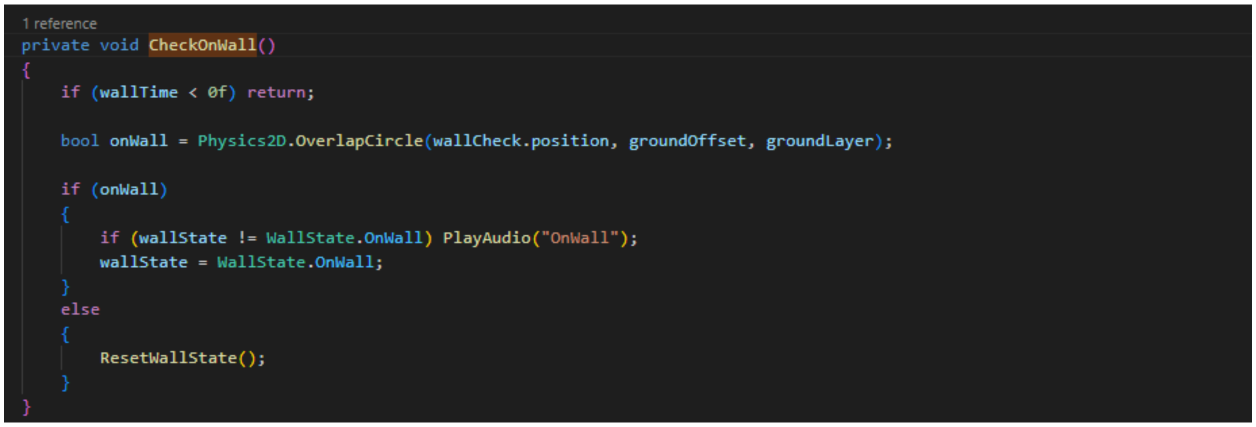
The last if checks drive the states of the ***"airState"*** and the ***"wallState"*** variables. The ***"DetectSlope()"*** method I snuck in there was a later addition which was a solution to the problem of moving on slopes with a simple physics based movement system.



```
1 reference
private void DetectSlope()
{
    RaycastHit2D hit = Physics2D.Raycast(groundCheck.position, Vector2.down, groundOffset + 0.5f, groundLayer);

    if (hit)
    {
        slopeNormal = hit.normal;
        float slopeAngle = Vector2.Angle(slopeNormal, Vector2.up);

        if (slopeAngle > 10f && slopeAngle <= maxSlopeAngle)
        {
            onSlope = true;
        }
        else
        {
            onSlope = false;
        }
    }
    else
    {
        onSlope = false;
    }
}

1 reference
private void HandleSlopeMovement()
{
    if (IsGrounded() && onSlope)
    {
        Vector2 slopeDirection = Vector2.Perpendicular(slopeNormal).normalized;
        float moveDirection = facingRight ? -1 : 1;
        float force = groundForce / 2;

        Vector2 moveForce = force * moveDirection * slopeDirection;
        rb.AddForce(moveForce, ForceMode2D.Force);
        if (rb.velocity.y < 0)
        {
            rb.velocity = new Vector2(rb.velocity.x, 0);
        }
    }
}
```

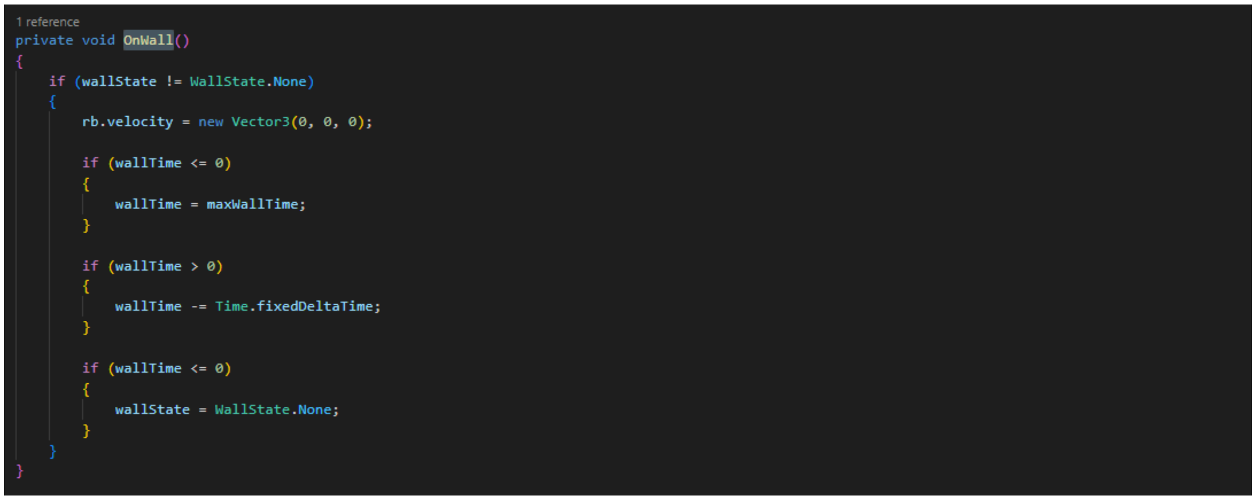
The method essentially raycasts right below the ground check position. If it hits a ground layer, it will calculate the normal angle of the slope, which if exceeds a 10 degree of threshold, the ***"onSlope"*** variable is set as **true**.

When that happens, the ***"HandleSlopeMovement"*** method will calculate the perpendicular angle to the slope normal, which is the angle of the plane of movement. The ***"facingRight"*** flag determines which direction of that plane the force will be applied to; and *voila*!



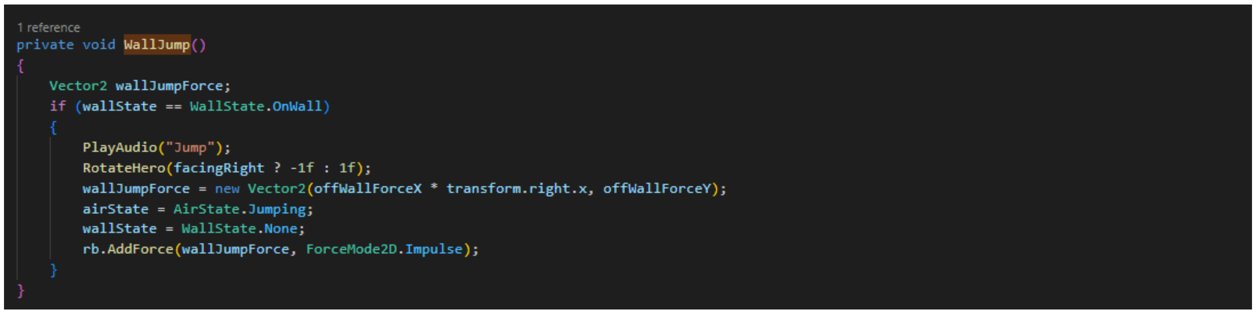
```
1 reference
private void CheckOnWall()
{
    if (wallTime < 0f) return;

    bool onWall = Physics2D.OverlapCircle(wallCheck.position, groundOffset, groundLayer);

    if (onWall)
    {
        if (wallState != WallState.OnWall) PlayAudio("OnWall");
        wallState = WallState.OnWall;
    }
    else
    {
        ResetWallState();
    }
}
```

The ***"CheckOnWall"*** method from the ***"CheckGrounded"*** method's final if check, drives the value of the ***"wallState"*** variable. The ***"OnWall"*** method below controls the timer and the effects of the timer. Player can stay on the wall until the timer runs out. During this time, a slight force downwards is applied to create a sliding effect.



```
1 reference
private void OnWall()
{
    if (wallState != WallState.None)
    {
        rb.velocity = new Vector3(0, 0, 0);

        if (wallTime <= 0)
        {
            wallTime = maxWallTime;
        }

        if (wallTime > 0)
        {
            wallTime -= Time.fixedDeltaTime;
        }

        if (wallTime <= 0)
        {
            wallState = WallState.None;
        }
    }
}
```

If the player's ***"wallState"*** variable is set as OnWall, the player can wall jump, which sets the ***"airState"*** to Jumping as well.



```
1 reference
private void WallJump()
{
    Vector2 wallJumpForce;
    if (wallState == WallState.OnWall)
    {
        PlayAudio("Jump");
        RotateHero(facingRight ? -1f : 1f);
        wallJumpForce = new Vector2(offWallForceX * transform.right.x, offWallForceY);
        airState = AirState.Jumping;
        wallState = WallState.None;
        rb.AddForce(wallJumpForce, ForceMode2D.Impulse);
    }
}
```

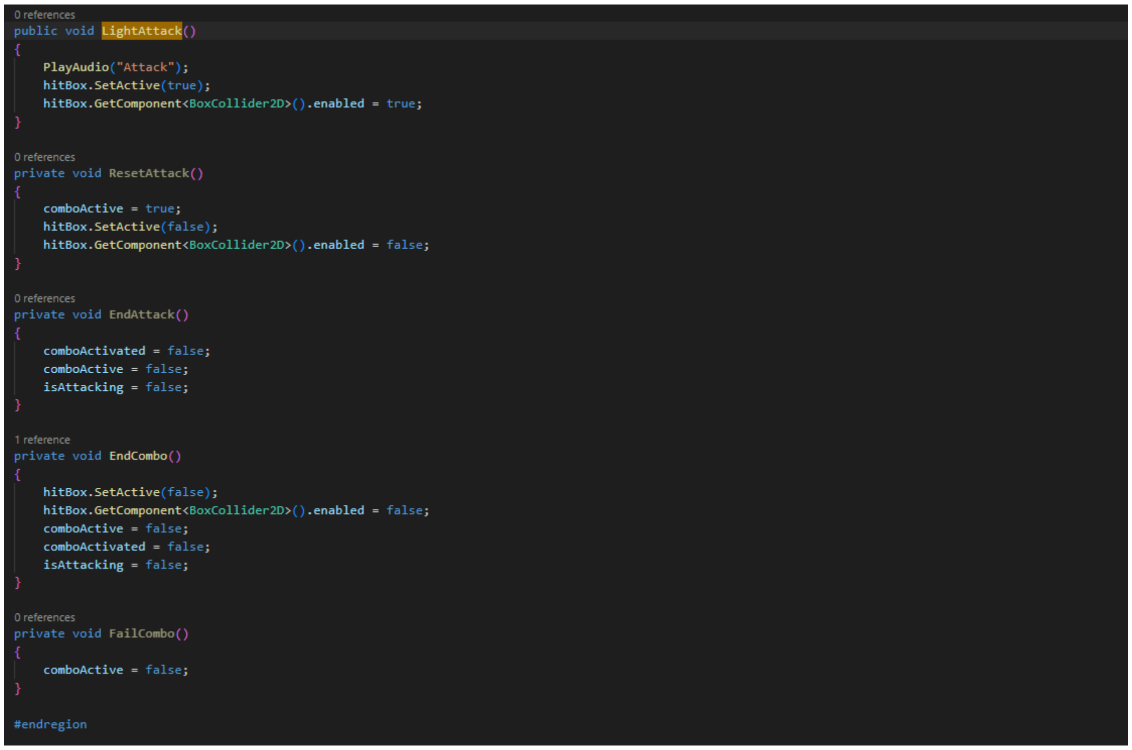
And with that, we can wrap up this section and move on to combat.

# Combat

Though the combat is very shallow, I tried structuring the codebase as extensively as I could by making use of inheritance. For example, the **Sword** class that is instantiated as the weapon of the hero inherits from the parent **Weapon** class, which holds all the main methods. I wanted to make more weapons and more mechanics, like weapons breaking but it would require me to make assets in extensively, which is not my main focus in game development.

Combat in the is controlled by animations at its core. Events at the key frames of the animation trigger certain methods which drives the combat. The one detailed thing I wanted to make was a very simple combo system, which I achieved through the animation event callbacks.

To explain it verbally, the sword flashes with a light blue color after the first attack finishes. There is a very brief period of time during this flash where the player can click again to trigger a follow up attack. This attack has no bonus damage as it's not very difficult to do but it is way quicker than waiting for the resetting of the first attack and then attacking again, so speed is the of attack is the reward for achieving this.



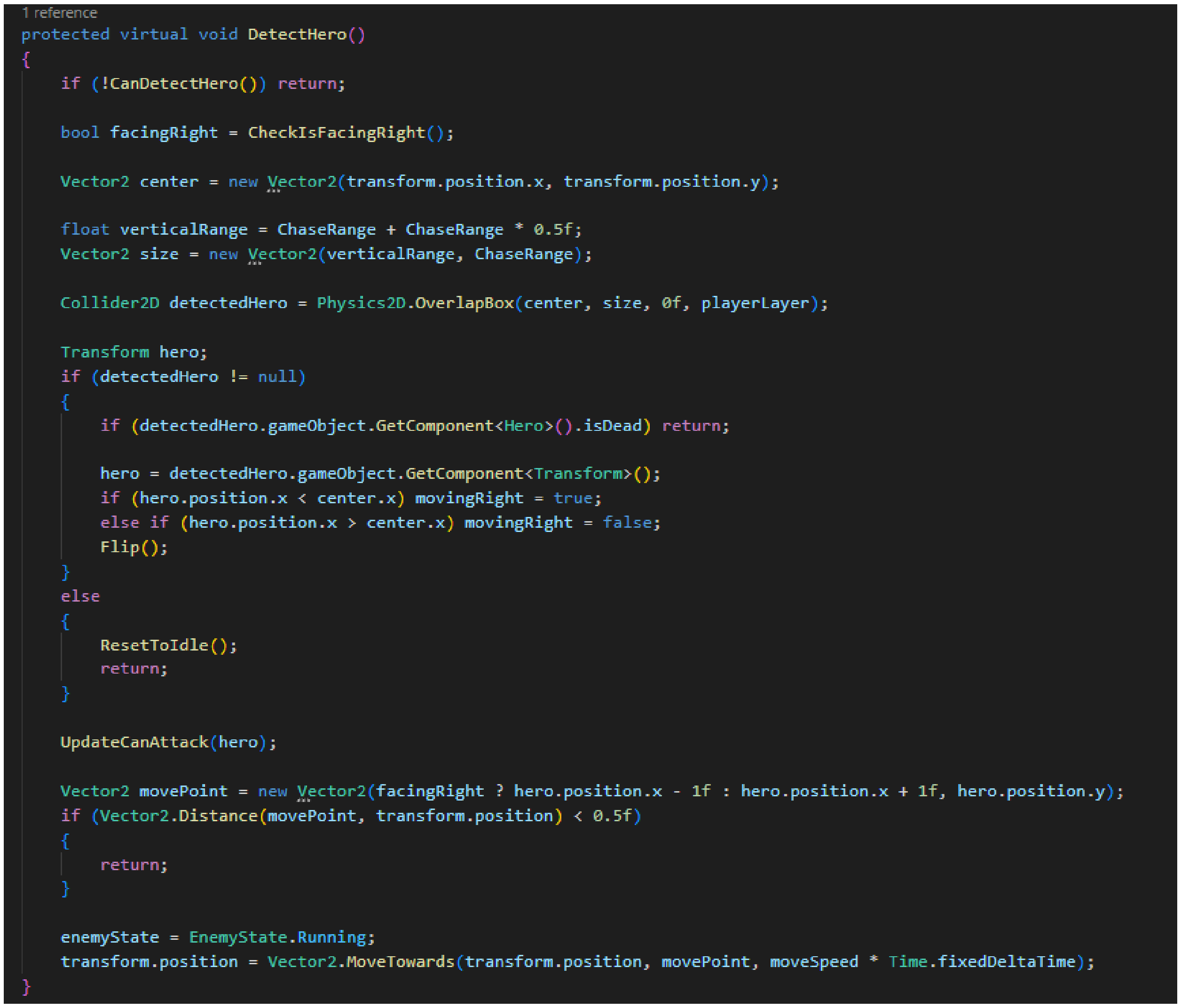
```
0 references
public void LightAttack()
{
    PlayAudio("Attack");
    hitBox.SetActive(true);
    hitBox.GetComponent<BoxCollider2D>().enabled = true;
}

0 references
private void ResetAttack()
{
    comboActive = true;
    hitBox.SetActive(false);
    hitBox.GetComponent<BoxCollider2D>().enabled = false;
}

0 references
private void EndAttack()
{
    comboActivated = false;
    comboActive = false;
    isAttacking = false;
}

1 reference
private void EndCombo()
{
    hitBox.SetActive(false);
    hitBox.GetComponent<BoxCollider2D>().enabled = false;
    comboActive = false;
    comboActivated = false;
    isAttacking = false;
}

0 references
private void FailCombo()
{
    comboActive = false;
}

#endregion
```

Above you can see the callback functions used in the animation events.

There are other helper methods like ***"PlaySound"*** in the code you see in the screenshots, but I will not dive into these as these are just wrapper methods to for easier use of the Unity API.

# Enemies

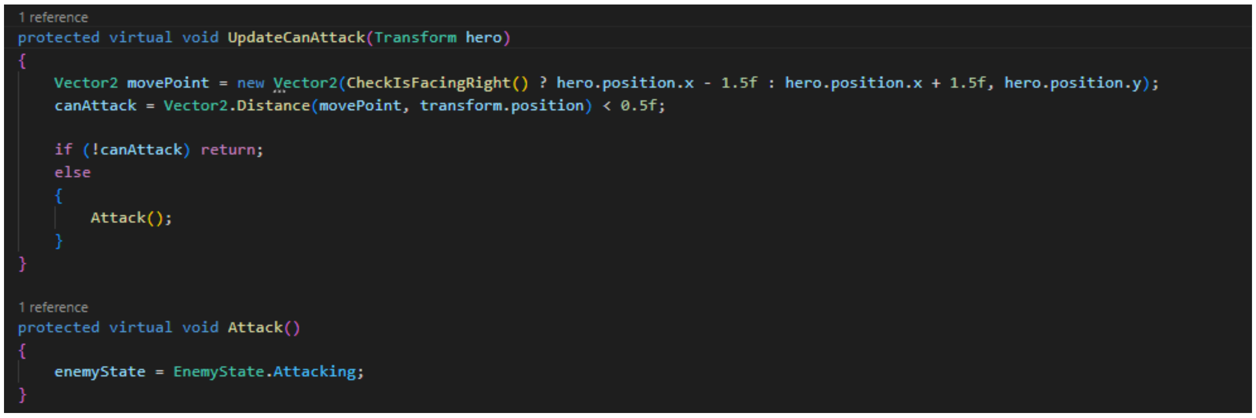
There is only a single type of enemy as I did not want to focus on diversity rather than mechanics and core concepts. But again, the codebase is structured in an extensive manner using inheritance.



```
1 reference
protected virtual void DetectHero()
{
    if (!CanDetectHero()) return;

    bool facingRight = CheckIsFacingRight();

    Vector2 center = new Vector2(transform.position.x, transform.position.y);

    float verticalRange = ChaseRange + ChaseRange * 0.5f;
    Vector2 size = new Vector2(verticalRange, ChaseRange);

    Collider2D detectedHero = Physics2D.OverlapBox(center, size, 0f, playerLayer);

    Transform hero;
    if (detectedHero != null)
    {
        if (detectedHero.gameObject.GetComponent<Hero>().isDead) return;

        hero = detectedHero.gameObject.GetComponent<Transform>();
        if (hero.position.x < center.x) movingRight = true;
        else if (hero.position.x > center.x) movingRight = false;
        Flip();
    }
    else
    {
        ResetToIdle();
        return;
    }

    UpdateCanAttack(hero);

    Vector2 movePoint = new Vector2(facingRight ? hero.position.x - 1f : hero.position.x + 1f, hero.position.y);
    if (Vector2.Distance(movePoint, transform.position) < 0.5f)
    {
        return;
    }

    enemyState = EnemyState.Running;
    transform.position = Vector2.MoveTowards(transform.position, movePoint, moveSpeed * Time.fixedDeltaTime);
}
```

This is the main method of the parent class **Enemy**. The method checks if the hero is in range by checking around an empty game object attached to it and if hero is detected, it starts moving that direction. If not, it resets to an idle state.

The ***"UpdateCanAttack"*** method checks if the hero is within a certain range, which corresponds to the hitbox collider range. If the hero is indeed within range, the state of the enemy turns to Attacking, which triggers the Attack animation to play. From there on, the same approach was used as in hero, which is that the animations have events that trigger callbacks.



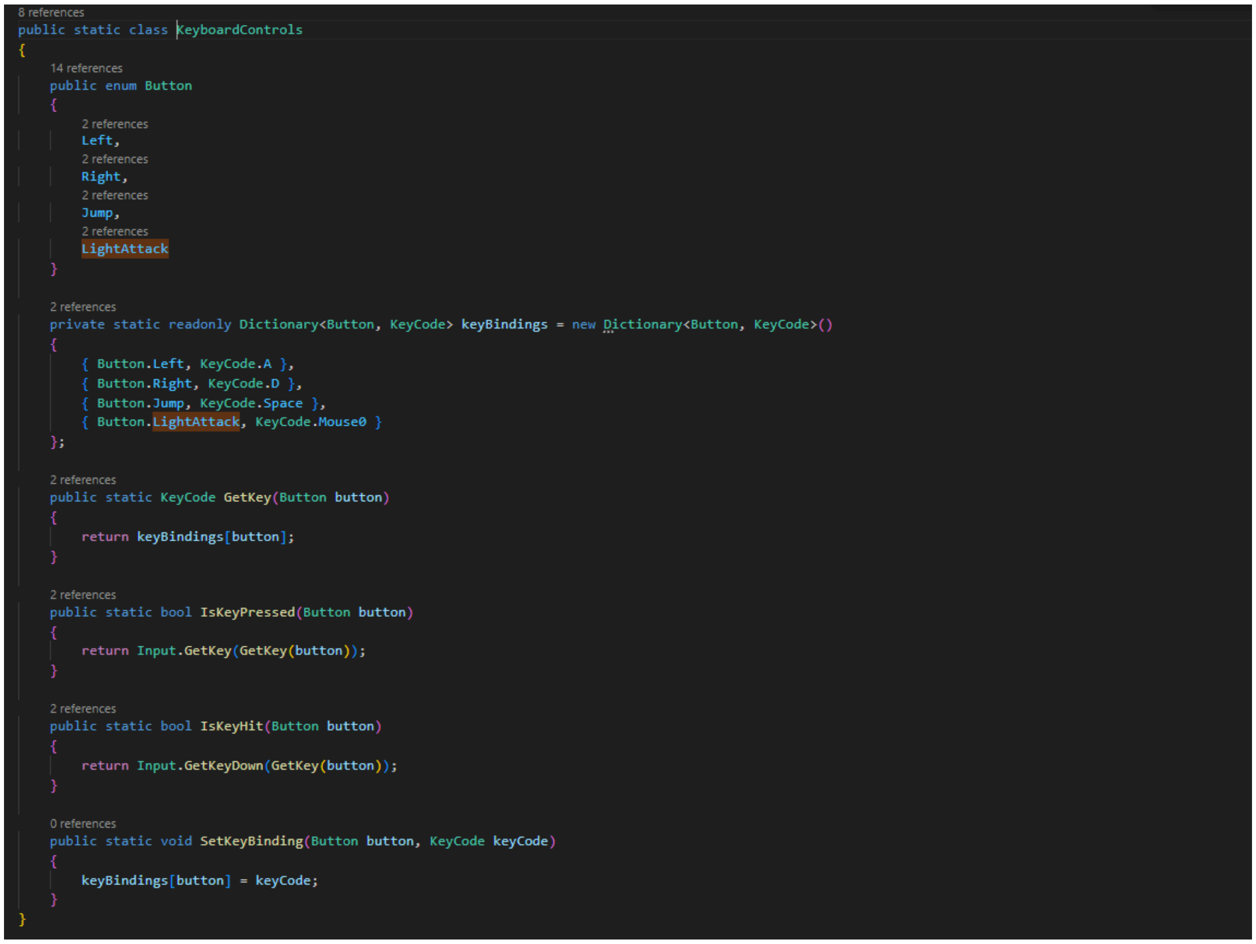
```
1 reference
protected virtual void UpdateCanAttack(Transform hero)
{
    Vector2 movePoint = new Vector2(CheckIsFacingRight() ? hero.position.x - 1.5f : hero.position.x + 1.5f, hero.position.y);
    canAttack = Vector2.Distance(movePoint, transform.position) < 0.5f;

    if (!canAttack) return;
    else
    {
        Attack();
    }
}

1 reference
protected virtual void Attack()
{
    enemyState = EnemyState.Attacking;
}
```

This is as deep as I feel it's worth to dive into this. There are of course other nuances like how the attacks connect, how the damage is applied, what happens when the enemy dies but these are very simple concepts.

And before I wrap up the mechanics, I'd like to talk about the character controller.

# Character Controller

The character controller takes inputs from a static ***KeyboardControls*** class that I created. I essentially wanted to implement gamepad support as well but didn't see this project as one I wanted to spend time on that as this will not be shipped as an actual game.



```
8 references
public static class KeyboardControls
{
    14 references
    public enum Button
    {
        2 references
        Left,
        2 references
        Right,
        2 references
        Jump,
        2 references
        LightAttack
    }

    2 references
    private static readonly Dictionary<Button, KeyCode> keyBindings = new Dictionary<Button, KeyCode>()
    {
        { Button.Left, KeyCode.A },
        { Button.Right, KeyCode.D },
        { Button.Jump, KeyCode.Space },
        { Button.LightAttack, KeyCode.Mouse0 }
    };

    2 references
    public static KeyCode GetKey(Button button)
    {
        return keyBindings[button];
    }

    2 references
    public static bool IsKeyPressed(Button button)
    {
        return Input.GetKey(GetKey(button));
    }

    2 references
    public static bool IsKeyHit(Button button)
    {
        return Input.GetKeyDown(GetKey(button));
    }

    0 references
    public static void SetKeyBinding(Button button, KeyCode keyCode)
    {
        keyBindings[button] = keyCode;
    }
}
```

This script is a very simple way to create the basic input schema and methods for it. The difference between ***"IsKeyPressed"*** and ***"IsKeyHit"*** was essential for distinguishing in taking inputs from continuous inputs like movement and impulsive inputs like jump or attack.

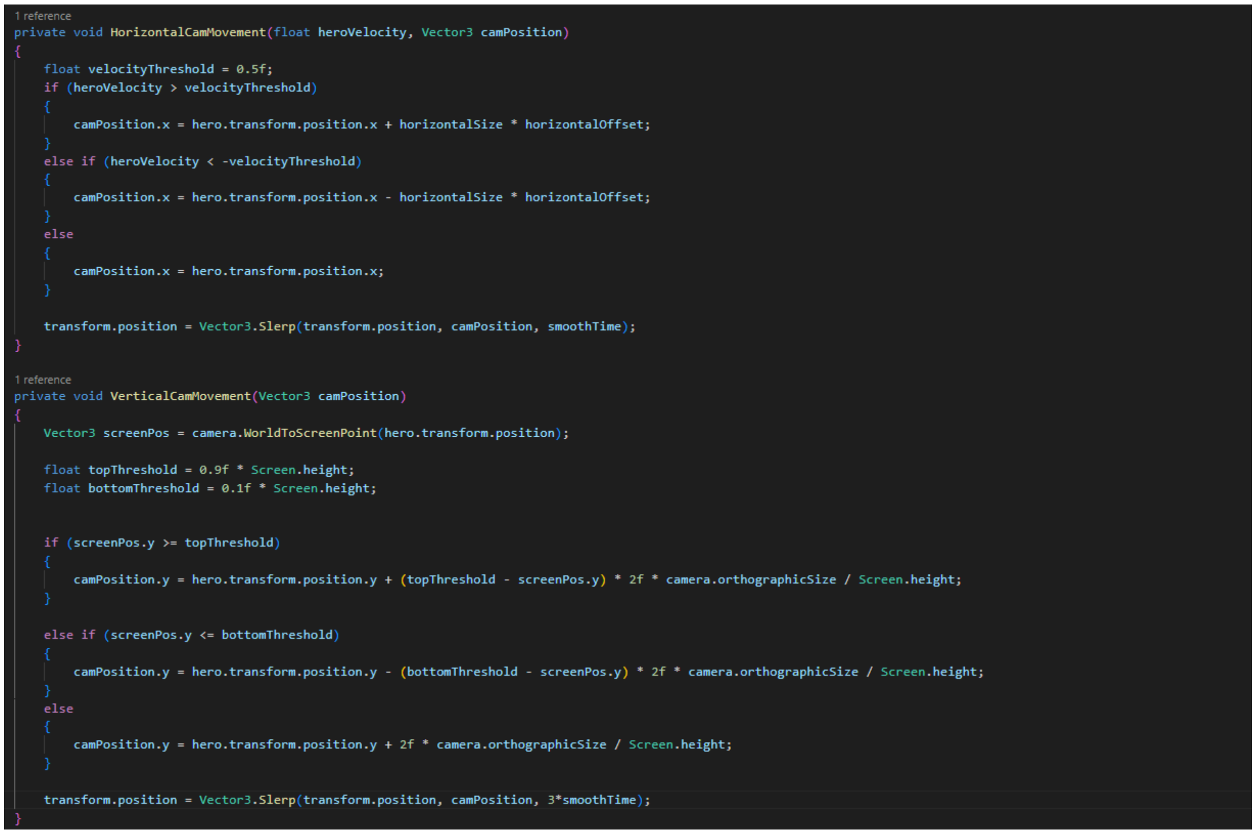
And with that, the mechanics are all wrapped up. In the next section, I will talk about how I handled the camera.

# Camera

I wanted to make my own camera script without using any libraries; external or Unity integrated. I'm not a fan of re-inventing the wheel but I wanted to learn how it works from scratch.

It's actually very simple. I had a threshold of view I wanted to keep my character in. At first I tried to do it by defining the thresholds on both the y-axis and the x-axis but the threshold on the x-axis did not work for me as its consistency was affected by the characters movement speed. If the character was too fast, it would go out of the camera's view. I could potentially fix that by increasing the sway speed of the camera but that made the camera so fast that playing the game for a long time could give motion sickness.

After all that, I opted to use threshold points for vertical camera movement and movement speed thresholds for horizontal camera movement.



```
1 reference
private void HorizontalCamMovement(float heroVelocity, Vector3 camPosition)
{
    float velocityThreshold = 0.5f;
    if (heroVelocity > velocityThreshold)
    {
        camPosition.x = hero.transform.position.x + horizontalSize * horizontalOffset;
    }
    else if (heroVelocity < -velocityThreshold)
    {
        camPosition.x = hero.transform.position.x - horizontalSize * horizontalOffset;
    }
    else
    {
        camPosition.x = hero.transform.position.x;
    }

    transform.position = Vector3.Slerp(transform.position, camPosition, smoothTime);
}

1 reference
private void VerticalCamMovement(Vector3 camPosition)
{
    Vector3 screenPos = camera.WorldToScreenPoint(hero.transform.position);

    float topThreshold = 0.9f * Screen.height;
    float bottomThreshold = 0.1f * Screen.height;


    if (screenPos.y >= topThreshold)
    {
        camPosition.y = hero.transform.position.y + (topThreshold - screenPos.y) * 2f * camera.orthographicSize / Screen.height;
    }

    else if (screenPos.y <= bottomThreshold)
    {
        camPosition.y = hero.transform.position.y - (bottomThreshold - screenPos.y) * 2f * camera.orthographicSize / Screen.height;
    }
    else
    {
        camPosition.y = hero.transform.position.y + 2f * camera.orthographicSize / Screen.height;
    }

    transform.position = Vector3.Slerp(transform.position, camPosition, 3*smoothTime);
}
```

This was pretty much what I had to do to get a decent enough camera movement and I think it worked pretty well for my game.

# Other Concepts

This section covers the other miscellaneous things I want to talk about before I end this documentation.

## Assets

At first I wanted to make my original assets. I spent a few days trying to learn how to make simple pixel art using Gimp. It was fun at first but I quickly realized that even the simplest of artwork I wanted to make would take me ages. If I kept on trying to create original assets, I would surely not last. As a programmer I can easily say that digital art is not the direction I was very willing to go.

Nearly all of the assets in the game are free assets I've found on the Unity asset store. I had to fix / edit some of them as they had minor issues; like the hero animation sprites having different heights off the ground for each other. I also added the blue flashing frame of the sword for my combo mechanic. I also made the animation that plays when the hero is on a wall by editing the jump animation sprites.

## Sound

The SFX are also taken from a Unity Asset Store free package. The music is made by my brother, who is credited at the end of the game. I can frankly say that the music is the best part of the game. Here also is his SoundCloud profile.

## Interface

The game nearly has no interface because it simply does not need to. There are health bars on top of each enemy, a health bar on the top left of the screen for the player, a credits UI at the end of the game and that's it.

# Final Thoughts

This project took me about 2.5-3 weeks to complete, by my definition of completion which I clarified at the start of this documentation. When I started making it, I had way more ambitious but "realistic" expectations of it. I accepted midway through it that if I were to realize these expectations, it'd take me way too much time and would stop being a learning curve experience and rather become an actual game project.

It was fun nonetheless and my next project will be a simple 2D platformer with Mario 2D style combat.

Thank you for taking the time to read this documentation.

**Ata Eren Arslan**
**ataerena599@gmail.com**