Rum Runner's Revenge

19TH MAY 2023

Game Studio Project – Group E
Bournemouth University

Logo Name

Contents

Overview

This document outlines the technical design for the game studio project titled “Rum Runner’s Revenge. This game is an action adventure 2D platformer, developed for PC and console platforms. The game project will be developed using the Unity game engine and C# programming language.

Target Platform Considerations

As mentioned previously the game is targeted for PC and Console platforms. The minimum system requirements for PC platforms are:

- **Windows 10 operating system**
- **Intel Core i3 processor or equivalent**
- **4GB RAM**
- **DirectX 11 compatible graphics card**
- **1GB available storage space**

The game will be developed to run on Xbox 360 and PlayStation 3 consoles, with minimum requirements of:

- **XBox 360 or later / PlayStation 3 or late**
- **2GB RAM**
- **1GB available storage space**

Technology Choices

These are the following technologies that we decided to use in the development of the game:

Hardware:

- **Drawing Tablets:** artists would use drawing tablets when creating assets for its precision and comfortability in asset art creation

Software:

- **Unity Game Engine:** this was chosen for its easy cross platform support and ease of use, since most of the team already had experience with the engine
- **C# Programming Language:** this language was chosen for its simplicity, ease of use as well as its OOP programming features and finally it is the language that is used by the Unity Game Engine
- **Asperite:** Asperite was used to create the sprites for the players and enemies for its dedicated software for pixel art, which is the main theme of the game
- **Photoshop:** UI components and elements were created on photoshop for more advanced layering which Asperite could not uphold

Websites:

- **Git Version Control:** All team members were working from home; therefore, the best environment setup was to use Git to share a remote collaboration feature as well as ease of use in terms of project syncing
- **Trello Project Management:** Trello was used for its free collaboration features as well as the sufficient upload space to share assets as well as timeline management
- **Gant Chart:** will be used to monitor tasks assigned to each member as well as duration of task and planned execution date

System Architecture/Technical Description

The 2D platformer game will be developed using a component design approach, with a class hierarchy that reflects the game's enemies, player, and systems. The following classes will be used:

- **Player Base:** this is the base script of the player that holds all the information of the state machine, the states of the player, as well as all component references such as the Animator, the core and finally the player data
- **Core:** the core is a component system where each individual component has a specific set of functions that can be accessed by both the player and the enemies such as the Movement Component which is responsible for changing the velocity of the entity
- **Enemy Base:** this is like the player base script however this is responsible for all enemies to use as a template to override when creating enemy specifics and a change to the default behaviour
- **State Machine:** this is a simple class that has one main function which is to enable the enemy and player to change between states
- **States:** this class acts as a template for inheriting states where we declare the logic used to check whether we are entering a state or exiting a state as well as change the animator bool parameter to handle animations
- **Player Weapon:** this is the multi weapon component system where we can change the properties of a scriptable object weapon by adding or removing predefined components such as knockback, movement on hit, damage, and others
- **Entity FX:** this is a script that can handle sprite changes either for player or the enemy, it will be used to make the sprites flash white on hit to show visual confirmation of a successful attack
- **Player Manager:** a simpleton script that will give us access to the player from any script without having to declare the component
- **Player Input Handler:** this will handle user input through unity's new input system and translate it into game Booleans that will be translated into game actions
- **Animation To State Machine:** this is an intermediary where it will act as connection between the animation controller and the enemy or the player state machine so we can use animation trigger events to cause specific actions to happen during parts of animations
- **Damage, Knockback and Poise Interface:** these interfaces are used to call simple functions such as Damage() which takes the parameter of a float that is the damage amount, an interface is a place to define a method's function where if a class implements the interface, the class has access to the function and since C# cannot inherit from multiple classes this is the best option

- **Mini map: This is a simple mini map that allows the player to see what is coming up as well as what they have already past to help direct the player to secret paths and see upcoming enemies.**
- **Physics bridge: This is a simple Physics bridge that uses rigidbody as well as box colliders to simulate a simple realistic 2D bridge which uses 2D Hinge joints to achieve the desired simulation.**

Workflow/Processes

We have decided to follow the agile methodology for the development process, meaning there will be a weekly cram session and we would have a daily meeting once per week for a progress report. Both Trello and Gant Chart will be used to track progress and remaining tasks that have been assigned to each member. GitHub will be used to store backups of source code where each team member will be creating their own branch for testing and once the code review has been complete then the merging changes will be brought to the master branch.

Asset List:

Player:

- Player

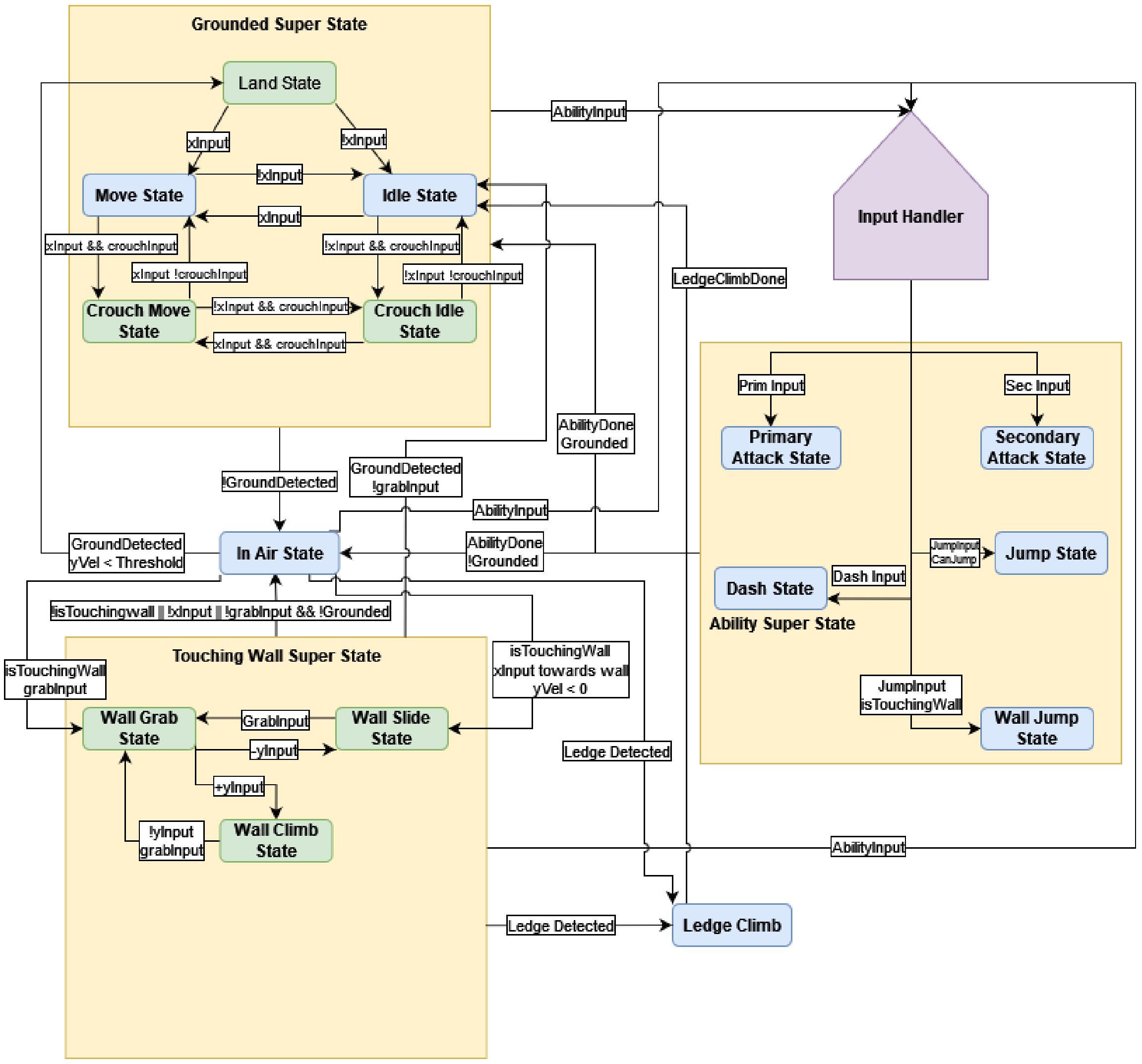
Enemies:

- Archer
- Crab
- Sentient Cannon

Props:

- HP Bottle
- Firefly
- Torches
- Signs
- Treasure Chest
- Gold Coin

Player Flow Chart



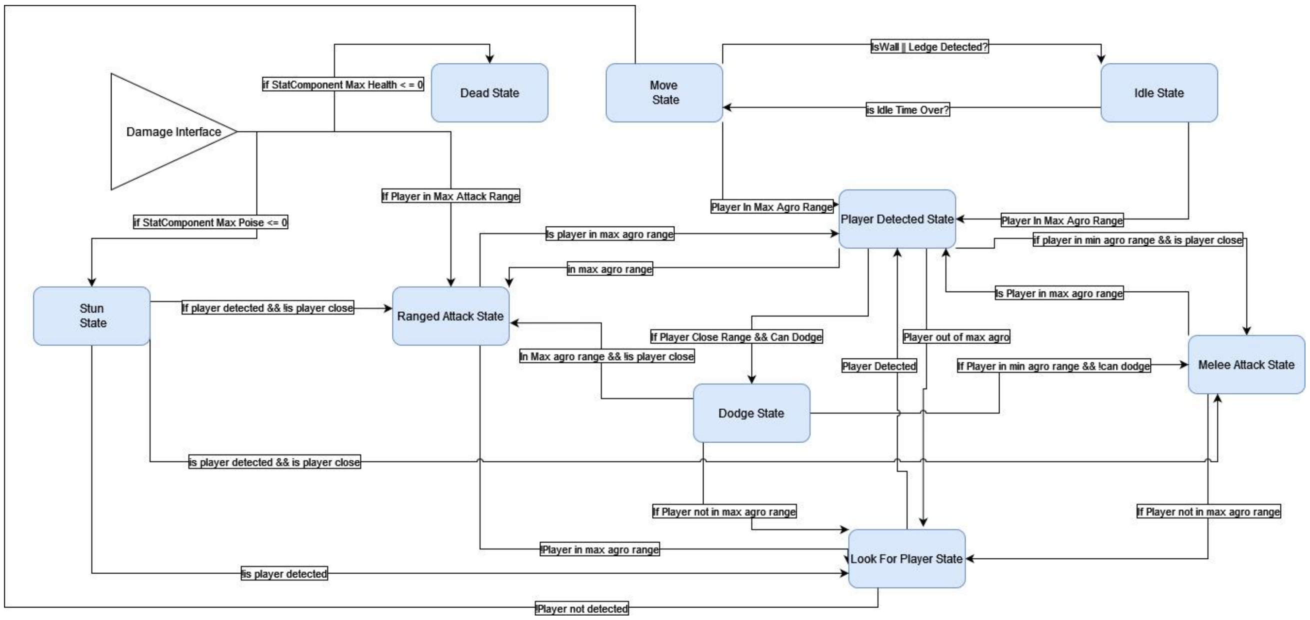
Player flow chart showing how the state machine transitions from one state to the other

The player state machine can only transition from one state to the other if only specific parameters are met, these parameters can be calculated automatically using the collision senses, whether the player is touching ground or not or with a button press instead.

This player state machine is from a tutorial that was used as a guide for development of the 2D platformer game, however transitions in states and with the combination of the core component stats new variables of stamina as well as new states it has been adjusted to suit the needs of our game.[1]

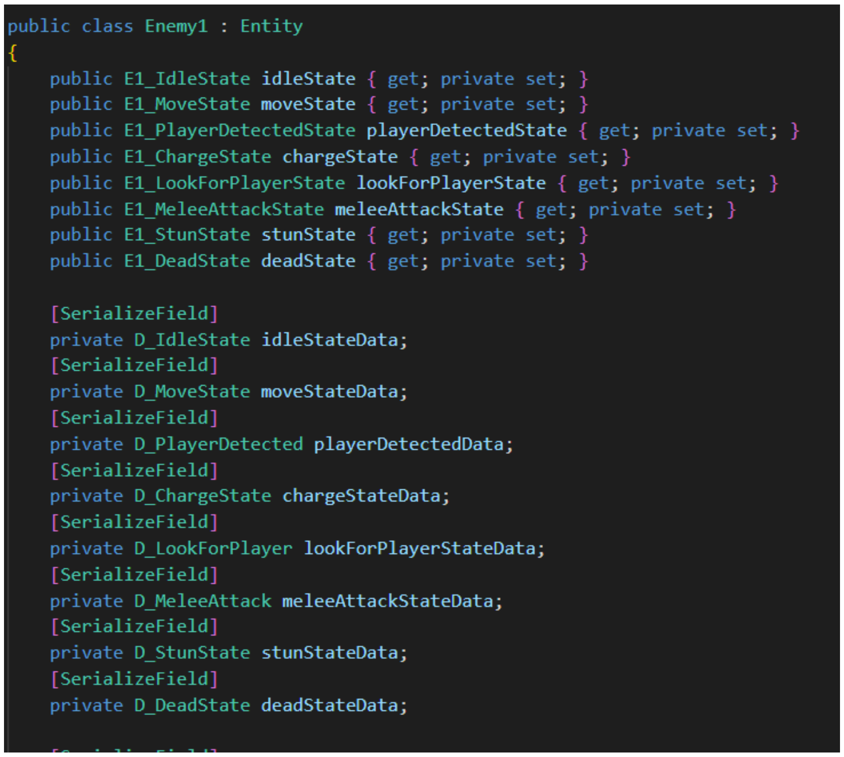
[1] Bardent (2019) 2D Platform Controller [Playlist], YouTube. Available at: https://www.youtube.com/watch?v=Pux1GlFwKPs&list=PLy78FINcVmjA0zDBhLuLNL1Jo6xNMMq-W (Accessed 16th February 2023)

Enemy Flow Chart


!isWall || Ledge Detected?
if StatComponent Max Health < = 0
Dead State
Move State
Idle State
is Idle Time Over?
Damage Interface
if Player in Max Attack Range
Player In Max Agro Range
Player Detected State
Player In Max Agro Range
if StatComponent Max Poise <= 0
!is player in max agro range
if player in min agro range && is player close
!in max agro range
Stun State
if player detected && !is player close
Ranged Attack State
Is Player in max agro range
If Player Close Range && Can Dodge
Player out of max agro
In Max agro range && !is player close
Player Detected
If Player in min agro range && !can dodge
Melee Attack State
Dodge State
is player detected && is player close
If Player not in max agro range
If Player not in max agro range
!Player in max agro range
Look For Player State
!is player detected
!Player not detected

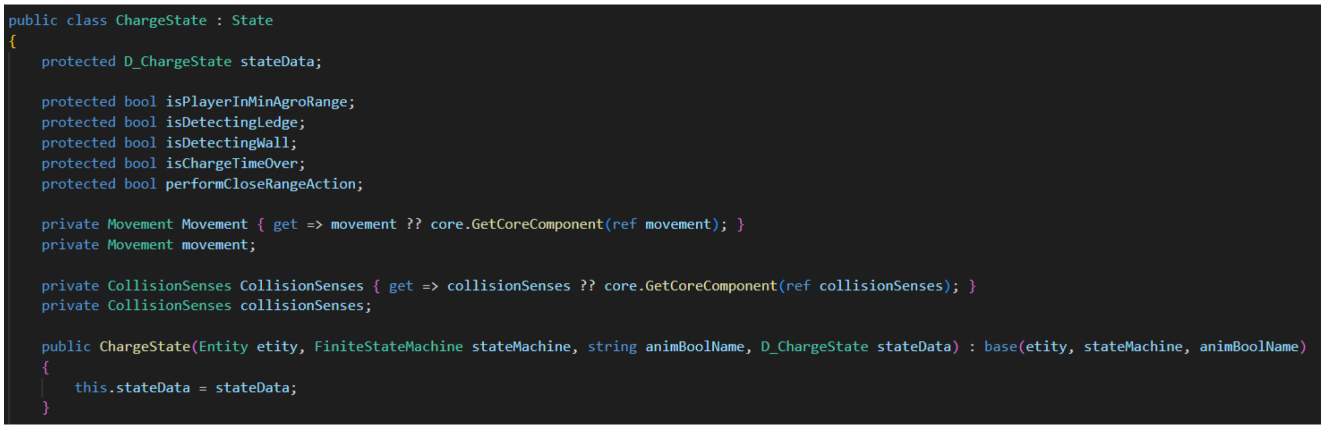
https://drive.google.com/file/d/1ixfqnd-ZYY2H1KhaUoKrRqOh3UrNWMCG/view?usp=sharing drive link incase image is too small to read, Enemy Flow Chart diagram

The enemy flow chart works similar to the player state machine where the transitions will happen when a specific condition is met, however control is not needed so this makes it an Artificial Intelligence. This initial enemy state machine is also from the tutorial mentioned above, however, it has been heavily refactored because the process of creating a new enemy with the original state machine seems tedious and unnecessary. For example, this is how an enemy specific script would look like with the old system:



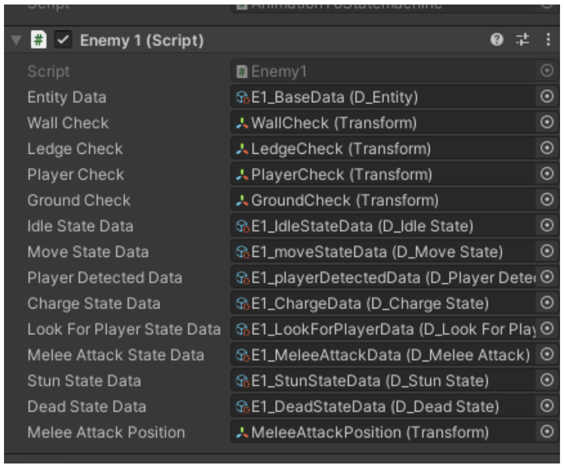
```
public class Enemy1 : Entity
{
    public E1_IdleState idleState { get; private set; }
    public E1_MoveState moveState { get; private set; }
    public E1_PlayerDetectedState playerDetectedState { get; private set; }
    public E1_ChargeState chargeState { get; private set; }
    public E1_LookForPlayerState lookForPlayerState { get; private set; }
    public E1_MeleeAttackState meleeAttackState { get; private set; }
    public E1_StunState stunState { get; private set; }
    public E1_DeadState deadState { get; private set; }

    [SerializeField]
    private D_IdleState idleStateData;
    [SerializeField]
    private D_MoveState moveStateData;
    [SerializeField]
    private D_PlayerDetected playerDetectedData;
    [SerializeField]
    private D_ChargeState chargeStateData;
    [SerializeField]
    private D_LookForPlayer lookForPlayerStateData;
    [SerializeField]
    private D_MeleeAttack meleeAttackStateData;
    [SerializeField]
    private D_StunState stunStateData;
    [SerializeField]
    private D_DeadState deadStateData;
```

Meaning that with the way the constructor was set in the original state machine, we would need a scriptable object that contains data about the specific state, and it would have to be declared for every new state. Like so:



```
public class ChargeState : State
{
    protected D_ChargeState stateData;

    protected bool isPlayerInMinAgroRange;
    protected bool isDetectingLedge;
    protected bool isDetectingWall;
    protected bool isChargeTimeOver;
    protected bool performCloseRangeAction;

    private Movement Movement { get => movement ?? core.GetCoreComponent(ref movement); }
    private Movement movement;

    private CollisionSenses CollisionSenses { get => collisionSenses ?? core.GetCoreComponent(ref collisionSenses); }
    private CollisionSenses collisionSenses;

    public ChargeState(Entity etity, FiniteStateMachine stateMachine, string animBoolName, D_ChargeState stateData) : base(etity, stateMachine, animBoolName)
    {
        this.stateData = stateData;
    }
```

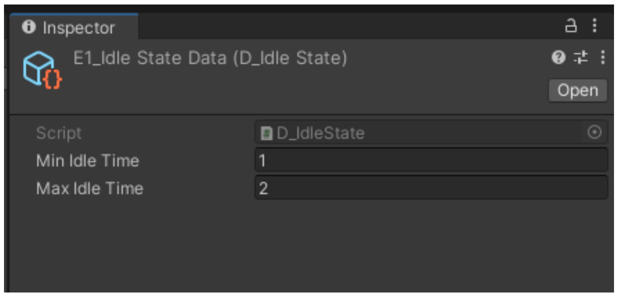
This truly became a tedious process as the enemy script in the end would look something like this:



Enemy 1 (Script)

Script	Enemy1
Entity Data	E1_BaseData (D_Entity)
Wall Check	WallCheck (Transform)
Ledge Check	LedgeCheck (Transform)
Player Check	PlayerCheck (Transform)
Ground Check	GroundCheck (Transform)
Idle State Data	E1_IdleStateData (D_Idle State)
Move State Data	E1_moveStateData (D_Move State)
Player Detected Data	E1_playerDetectedData (D_Player Dete
Charge State Data	E1_ChargeData (D_Charge State)
Look For Player State Data	E1_LookForPlayerData (D_Look For Play
Melee Attack State Data	E1_MeleeAttackData (D_Melee Attack)
Stun State Data	E1_StunStateData (D_Stun State)
Dead State Data	E1_DeadStateData (D_Dead State)
Melee Attack Position	MeleeAttackPosition (Transform)

Where one data scriptable object would often contain 1 – 2 variables:



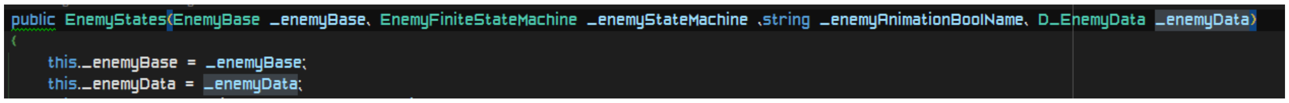
Inspector

E1_Idle State Data (D_Idle State)

Open

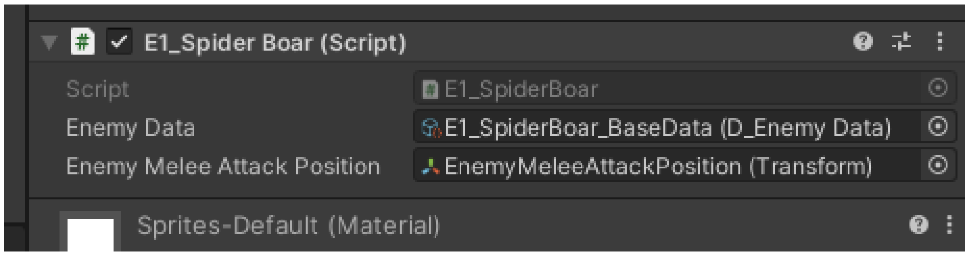
Script	D_IdleState
Min Idle Time	1
Max Idle Time	2

Therefore, by going into the enemy states and changing the constructor of the states to include the enemy data in the past variables:



```
public EnemyStates(EnemyBase _enemyBase, EnemyFiniteStateMachine _enemyStateMachine ,string _enemyAnimationBoolName, D_EnemyData _enemyData)
{
    this._enemyBase = _enemyBase;
    this._enemyData = _enemyData;
```

We can make the enemy script much faster to setup a new enemy as well as only have to create one new enemy data scriptable object

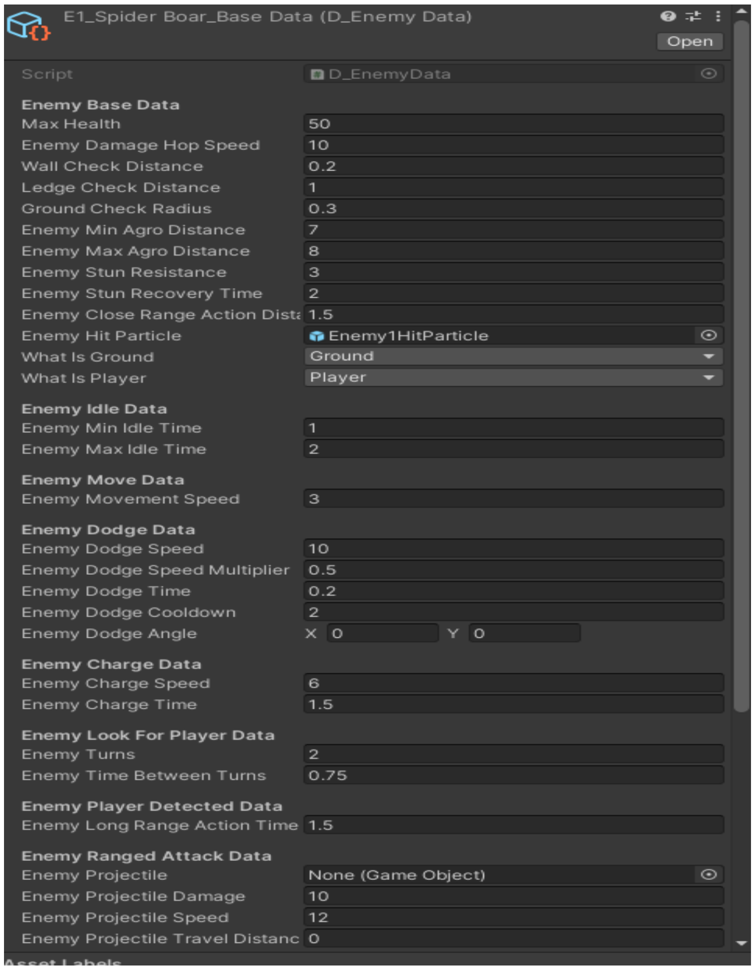


E1_Spider Boar (Script)

Script	E1_SpiderBoar
Enemy Data	E1_SpiderBoar_BaseData (D_Enemy Data)
Enemy Melee Attack Position	EnemyMeleeAttackPosition (Transform)

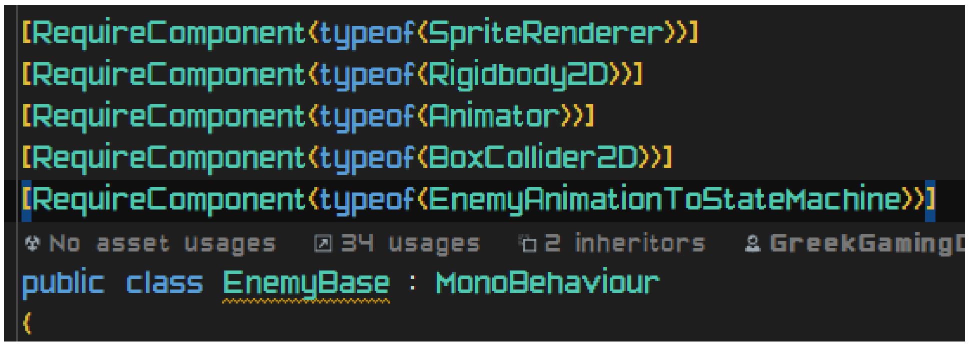
Sprites-Default (Material)

Transferring all the previous variables and being able to change them all at once into one object



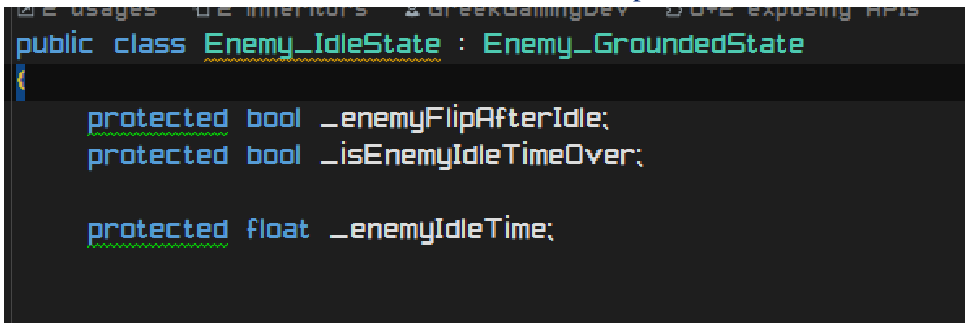
```
E1_Spider Boar_Base Data (D_Enemy Data)                    Open

Script                              D_EnemyData

Enemy Base Data
Max Health                          50
Enemy Damage Hop Speed              10
Wall Check Distance                 0.2
Ledge Check Distance                1
Ground Check Radius                 0.3
Enemy Min Agro Distance             7
Enemy Max Agro Distance             8
Enemy Stun Resistance               3
Enemy Stun Recovery Time            2
Enemy Close Range Action Dista      1.5
Enemy Hit Particle                  Enemy1HitParticle
What Is Ground                      Ground
What Is Player                      Player

Enemy Idle Data
Enemy Min Idle Time                 1
Enemy Max Idle Time                 2

Enemy Move Data
Enemy Movement Speed                3

Enemy Dodge Data
Enemy Dodge Speed                   10
Enemy Dodge Speed Multiplier        0.5
Enemy Dodge Time                    0.2
Enemy Dodge Cooldown                2
Enemy Dodge Angle                   X 0          Y 0

Enemy Charge Data
Enemy Charge Speed                  6
Enemy Charge Time                   1.5

Enemy Look For Player Data
Enemy Turns                         2
Enemy Time Between Turns            0.75

Enemy Player Detected Data
Enemy Long Range Action Time        1.5

Enemy Ranged Attack Data
Enemy Projectile                    None (Game Object)
Enemy Projectile Damage             10
Enemy Projectile Speed              12
Enemy Projectile Travel Distanc     0
```

Another overhaul to the state machine would be adding a few simple lines but they would speed up production of enemies overall by removing the tedious process of adding each component manually to the game object, therefore we can make the EnemyBase script add them automatically for us since all variant enemies inherit from EnemyBase.

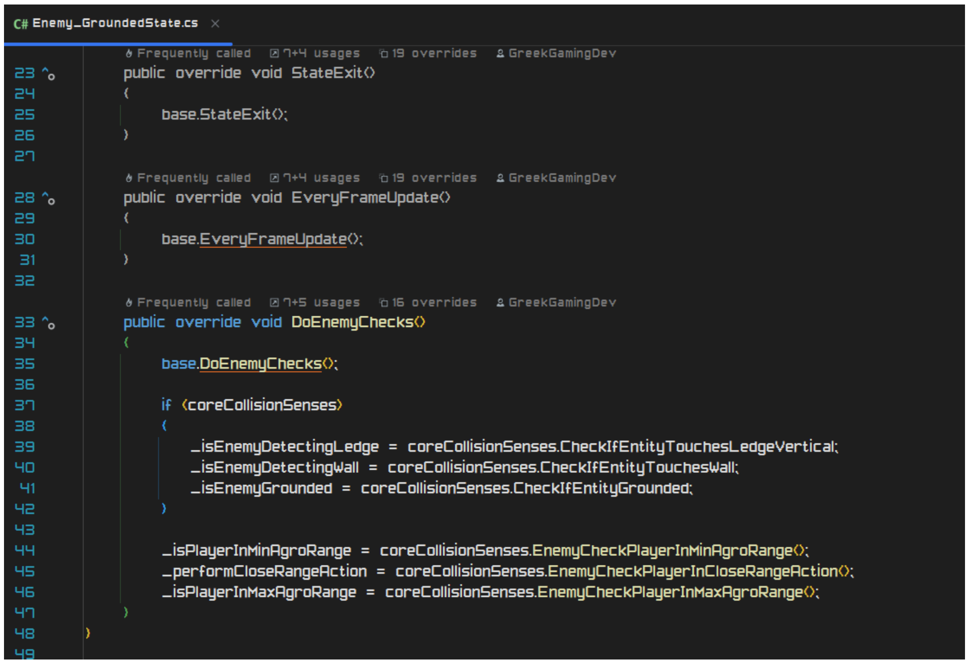


```csharp
[RequireComponent(typeof(SpriteRenderer))]
[RequireComponent(typeof(Rigidbody2D))]
[RequireComponent(typeof(Animator))]
[RequireComponent(typeof(BoxCollider2D))]
[RequireComponent(typeof(EnemyAnimationToStateMachine))]
 No asset usages   34 usages   2 inheritors   GreekGaming
public class EnemyBase : MonoBehaviour
{
```

The final overhaul of the old state machine was a much-needed change by creating the Enemy_GroundedState as many states such as Enemy_Idle or Enemy_Move would call the same collision checks in their own state therefore by making both of those states inherit from the grounded state we do not need to repeat code we have already written.



```
2 usages   2 inheritors   GreekGamingDev   0+2 exposing APIs
public class Enemy_IdleState : Enemy_GroundedState
{
    protected bool _enemyFlipAfterIdle;
    protected bool _isEnemyIdleTimeOver;

    protected float _enemyIdleTime;
```



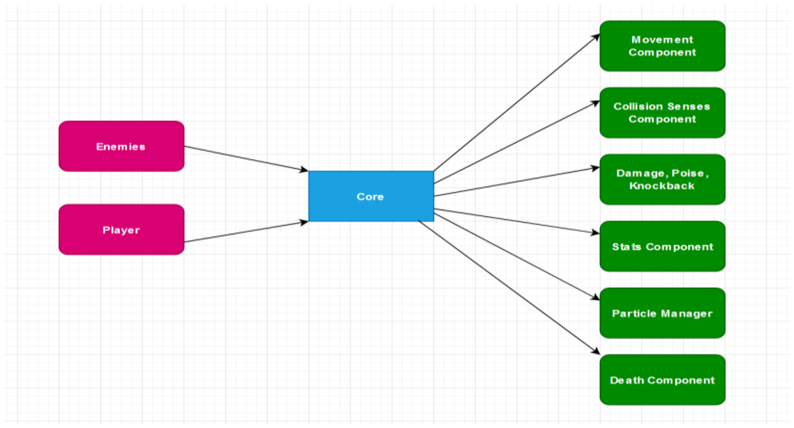
C# Enemy_GroundedState.cs

```
    Frequently called   7+4 usages   19 overrides   GreekGamingDev
    public override void StateExit()
    {
        base.StateExit();
    }

    Frequently called   7+4 usages   19 overrides   GreekGamingDev
    public override void EveryFrameUpdate()
    {
        base.EveryFrameUpdate();
    }

    Frequently called   7+5 usages   16 overrides   GreekGamingDev
    public override void DoEnemyChecks()
    {
        base.DoEnemyChecks();

        if (coreCollisionSenses)
        {
            _isEnemyDetectingLedge = coreCollisionSenses.CheckIfEntityTouchesLedgeVertical;
            _isEnemyDetectingWall = coreCollisionSenses.CheckIfEntityTouchesWall;
            _isEnemyGrounded = coreCollisionSenses.CheckIfEntityGrounded;
        }

        _isPlayerInMinAgroRange = coreCollisionSenses.EnemyCheckPlayerInMinAgroRange();
        _performCloseRangeAction = coreCollisionSenses.EnemyCheckPlayerInCloseRangeAction();
        _isPlayerInMaxAgroRange = coreCollisionSenses.EnemyCheckPlayerInMaxAgroRange();
    }
}
```

Core

The core consists of components that go both on the enemy and the player to reduce the amount of code that is needed to be written, this was also a part of the tutorial that is mentioned before and once again we needed an overhaul as we were reusing the same code every time. But first to list the core components:

- **Movement Component:** the movement component is responsible for controlling the rigid body of the entity that it is attached to with functions such as SetVelocityZero which will freeze the entity
- **Collision Senses:** This is the eyes and ears of the entity that this component is attached to as it is responsible for all ray cast checks that are responsible for the state machine to switch states based on these conditions
- **Core Damage, Core Knockback, Core Poise Receiver:** these components handle their respective name function using the interfaces that have been created and through these components we are able to interact between state machines to either damage, stun or knockback the player or enemy
- **Stats Component:** this component as the name suggests is responsible for the enemy stats as well as the player stats, such as the maximum health, poise, and stamina which is one of our overhauls
- **Particle Manager:** this is responsible for instantiating particles from the particle system with set functions to help us create particles at desired locations at specific times with specific rotations
- **Death Component:** This component is quite simple; it handles what happens when the enemy or player have their HP reduced to zero as well as instantiating prefabs of blood splatter which can also be changed to item drops
- **Core Stats System:** this serves as a more generic function that is linked with the stats component to initialize the variables as well as make sure that the current entity value is equal to its maximum value

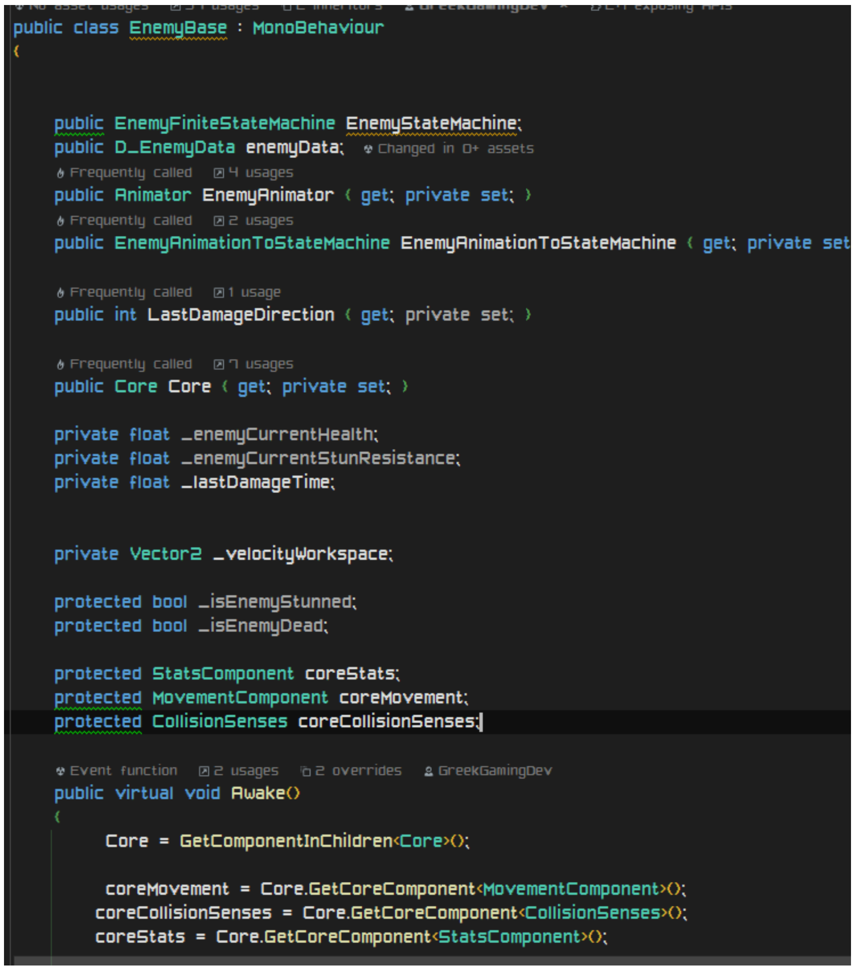


Flowchart of how the core communicates with the components

The overhaul that was needed for this system was in the way that we get reference to the core components, in the original core system we would need to use a generic function with custom getters and setters to get reference to one core component which quickly became a repetitive and time constraining task

```
private Movement Movement { get => movement ?? core.GetCoreComponent(ref movement); }
private Movement movement;
```

However, by using the GetCoreComponent function in the base scripts that are inherited by the player and enemies we only must declare them once and we have access to them from all inherited scripts



```
public class EnemyBase : MonoBehaviour
{


    public EnemyFiniteStateMachine EnemyStateMachine;
    public D_EnemyData enemyData;  Changed in 0+ assets
    Frequently called  4 usages
    public Animator EnemyAnimator { get; private set; }
    Frequently called  2 usages
    public EnemyAnimationToStateMachine EnemyAnimationToStateMachine { get; private set

    Frequently called  1 usage
    public int LastDamageDirection { get; private set; }

    Frequently called  7 usages
    public Core Core { get; private set; }

    private float _enemyCurrentHealth;
    private float _enemyCurrentStunResistance;
    private float _lastDamageTime;


    private Vector2 _velocityWorkspace;

    protected bool _isEnemyStunned;
    protected bool _isEnemyDead;

    protected StatsComponent coreStats;
    protected MovementComponent coreMovement;
    protected CollisionSenses coreCollisionSenses;

    Event function  2 usages  2 overrides  GreekGamingDev
    public virtual void Awake()
    {
        Core = GetComponentInChildren<Core>();

        coreMovement = Core.GetCoreComponent<MovementComponent>();
       coreCollisionSenses = Core.GetCoreComponent<CollisionSenses>();
       coreStats = Core.GetCoreComponent<StatsComponent>();
```

Multi Weapon System

The multi weapon system is from a tutorial that was used as a guide for development of the 2D platformer game.[2] this is a component-based system where we can create a weapon component as well as the data we want to pass into that component and have it all assigned in the custom editor that is part of the guide process. This makes creation new weapons very efficient and simple which is why we chose this system as we were focused on being able to create many weapons efficiently with minimum asset generation for the designers. We were also able to understand the component system well to create our own weapon components such as the life steal component which will heal the player for a specific amount with every hit. The rest of the components are:

- **Weapon Hitbox:** this is responsible for detecting all enemies within the specific collider
- **Weapon Damage:** this is responsible for dealing damage in accordance with the weapon hitbox through the damage interface
- **Weapon Knockback:** this also uses the hitbox component to knockback enemies using the knockback interface
- **Weapon Movement:** this calls a reference to the core movement component, where with our refactoring was amazingly easy to get access to, with every swing of the sword the player will move forward to give the sense of power behind every swing
- **Weapon Poise Damage:** the final component that uses the weapon hitbox and this deals damage to the enemy's poise stat through the damage component and if it reaches 0 the enemy will get stunned

[2] Bardent (2023) Multi – Weapon System [Playlist], Youtube. Available at: https://www.youtube.com/watch?v=DYx9OC39his&list=PLy78FINcVmjDeHVYh8SMjEP1B_MdSLCSz&index=1 (Accessed 16th February 2023)

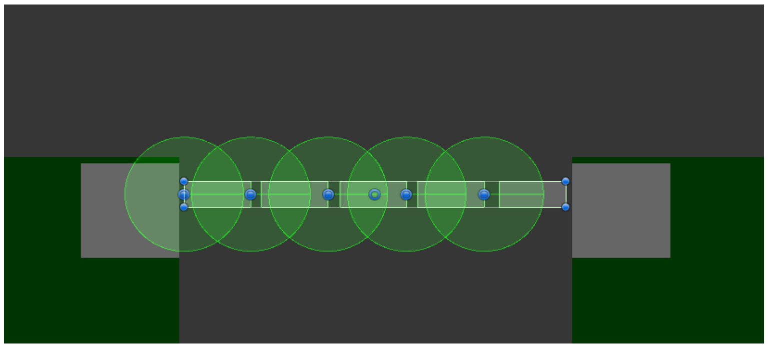
Physics Bridge

Building a simple 2D physics bridge in Unity can be an exciting process that adds depth and interactivity to a game environment. By leveraging the HingeJoint2D component, realistic physics-based connections can be created between bridge elements. The following steps were followed to implement such a bridge:

1. First, the bridge elements were designed and positioned in the Unity scene view.

2. Then, a Rigidbody2D component was attached to each bridge element that required physics interaction, specifically the planks and not the edges.

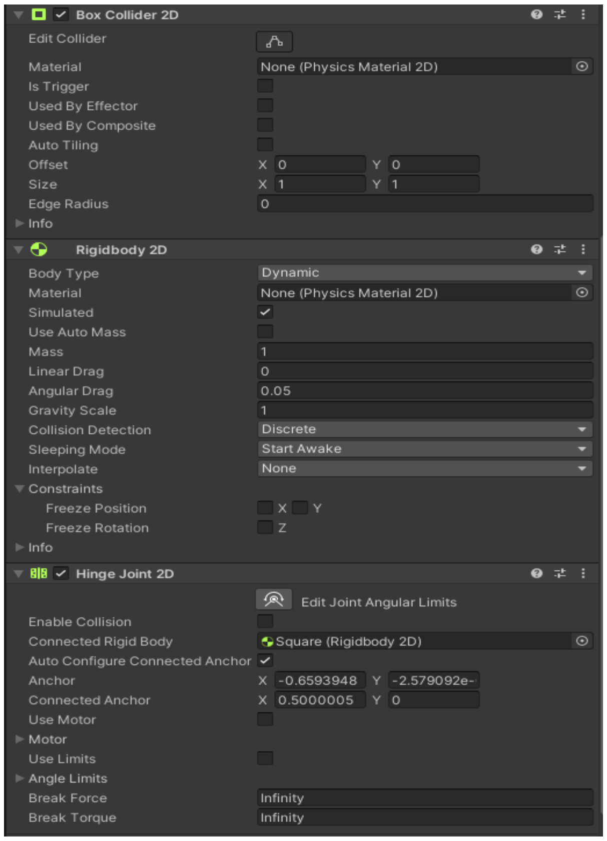
3. A HingeJoint2D component was added to connect adjacent bridge elements, with appropriate anchor points selected and joint properties configured, such as limits and motor.



4. To achieve the desired physics behaviour, the mass, drag, and other Rigidbody2D properties of the bridge elements were adjusted accordingly.

5. The bridge's stability was tested, and any necessary adjustments were made to the joint parameters, mass distribution, or anchor points.

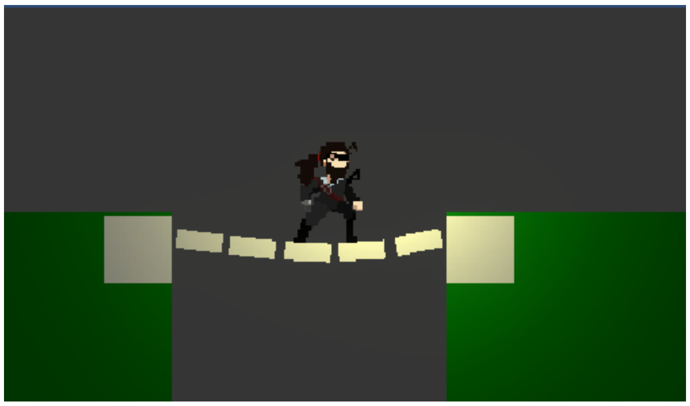
6. Colliders were added to the bridge elements for more accurate physics interactions with other objects in the scene.



```
Box Collider 2D
Edit Collider
Material                        None (Physics Material 2D)
Is Trigger                      ☐
Used By Effector                ☐
Used By Composite               ☐
Auto Tiling                     ☐
Offset                          X 0          Y 0
Size                            X 1          Y 1
Edge Radius                     0
Info

Rigidbody 2D
Body Type                       Dynamic
Material                        None (Physics Material 2D)
Simulated                       ✓
Use Auto Mass                   ☐
Mass                            1
Linear Drag                     0
Angular Drag                    0.05
Gravity Scale                   1
Collision Detection             Discrete
Sleeping Mode                   Start Awake
Interpolate                     None
Constraints
  Freeze Position               ☐ X ☐ Y
  Freeze Rotation               ☐ Z
Info

Hinge Joint 2D
                                Edit Joint Angular Limits
Enable Collision                ☐
Connected Rigid Body            Square (Rigidbody 2D)
Auto Configure Connected Anchor ✓
Anchor                          X -0.6593948 Y -2.579092e-
Connected Anchor                X 0.5000005  Y 0
Use Motor                       ☐
Motor
Use Limits                      ☐
Angle Limits
Break Force                     Infinity
Break Torque                    Infinity
```

7. **The bridge physics implementation underwent testing and iteration to ensure it provided the desired gameplay experience. The value of the Break force was tested to find a suitable value which will make the bridge break when the player walks on it.**

8. **Additionally, the bridge physics were optimized by adjusting physics settings, such as the physics timestep or solver iterations, to achieve smooth and stable simulation.**



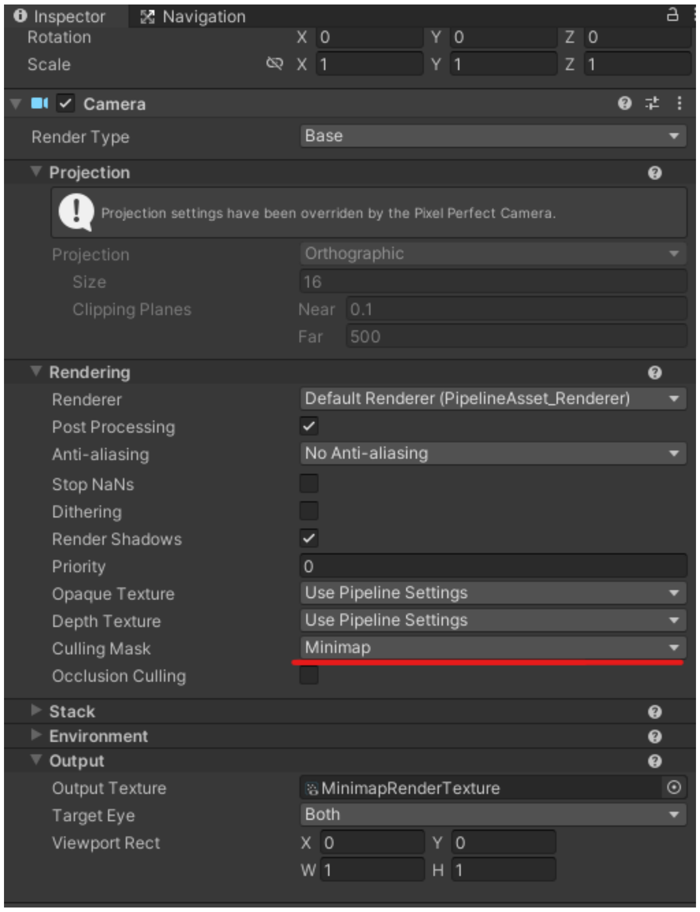
This serves as a basic overview of the process and steps taken to achieve the physics bridge presented in the final game. As you can see it is very simple and

Simple Mini-Map

The concept behind the development of a simplistic two-dimensional (2D) mini map involves the implementation of an additional camera that tracks the player's movements, rendering only specific game assets. These assets are then displayed within a user interface (UI) that is created for the mini map. The steps undertaken to achieve this functionality are outlined as follows:

A sprite, representing the player, is attached to the player's character. In this case, a circle sprite was utilized to depict the player on the mini map. To ensure that this sprite is exclusively rendered by the mini map camera and not the main camera, a new layer is created.

A second camera[3] is introduced, specifically configured to render assets associated with a designated layer. Only assets belonging to this layer will be displayed on the mini map.



Inspector | Navigation

Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Camera

Render Type: Base

Projection

Projection settings have been overriden by the Pixel Perfect Camera.

Projection: Orthographic
Size: 16
Clipping Planes: Near 0.1, Far 500

Rendering

Renderer: Default Renderer (PipelineAsset_Renderer)
Post Processing: ✓
Anti-aliasing: No Anti-aliasing
Stop NaNs:
Dithering:
Render Shadows: ✓
Priority: 0
Opaque Texture: Use Pipeline Settings
Depth Texture: Use Pipeline Settings
Culling Mask: Minimap
Occlusion Culling:

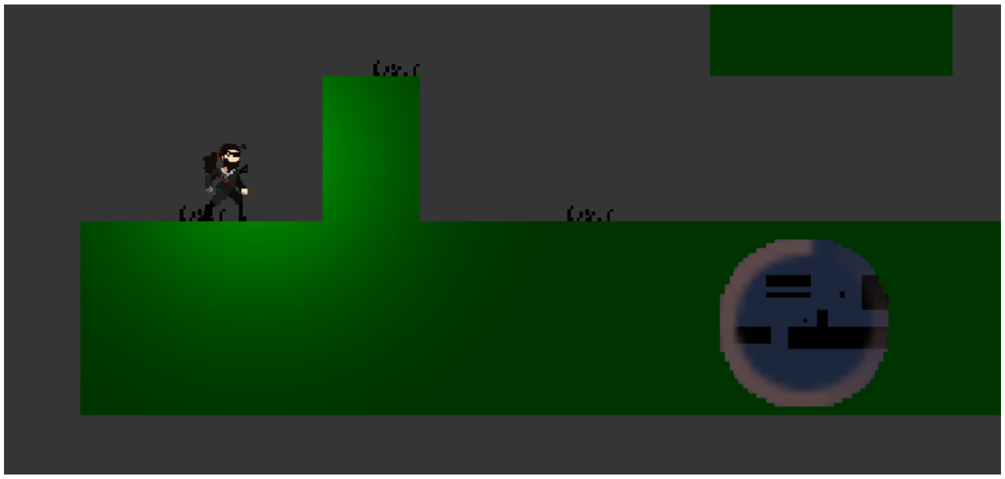
Stack

Environment

Output

Output Texture: MinimapRenderTexture
Target Eye: Both
Viewport Rect: X 0, Y 0, W 1, H 1

A render texture is generated and assigned to the mini map camera. Adjustments are made to the size of the render texture to ensure a smooth representation without pixelation.
A new canvas is created, and the render texture produced for the mini map is integrated into it.
To enhance the visual appeal of the map, an image and a mask component can be added to design the map's appearance.
Sprites and icons are then affixed to all desired elements that should appear on the mini map. The appropriate layer is assigned to these elements to ensure they are rendered by the designated camera.



By following these steps, a functional and visually appealing 2D mini map can be implemented within the game, displaying specific assets relevant to the player's movements. This can be further improved with specific sprites for the player, ground and enemies as well as any other game objects that are meant to be shown in the map.

[3]Camera (2023) How to make a Simple Minimap [Playlist], Youtube. Available at: https://www.youtube.com/watch?v=kWhOMJMihC0 (Accessed 11th May 2023)

Enemy Attacks

The provided code represents a class called "E3_Crab_MeleeAttack," which extends from the "Enemy_MeleeAttackState" class and implements the "DamageInterface" interface. This class defines a specific behaviour for an enemy crab character in a game.

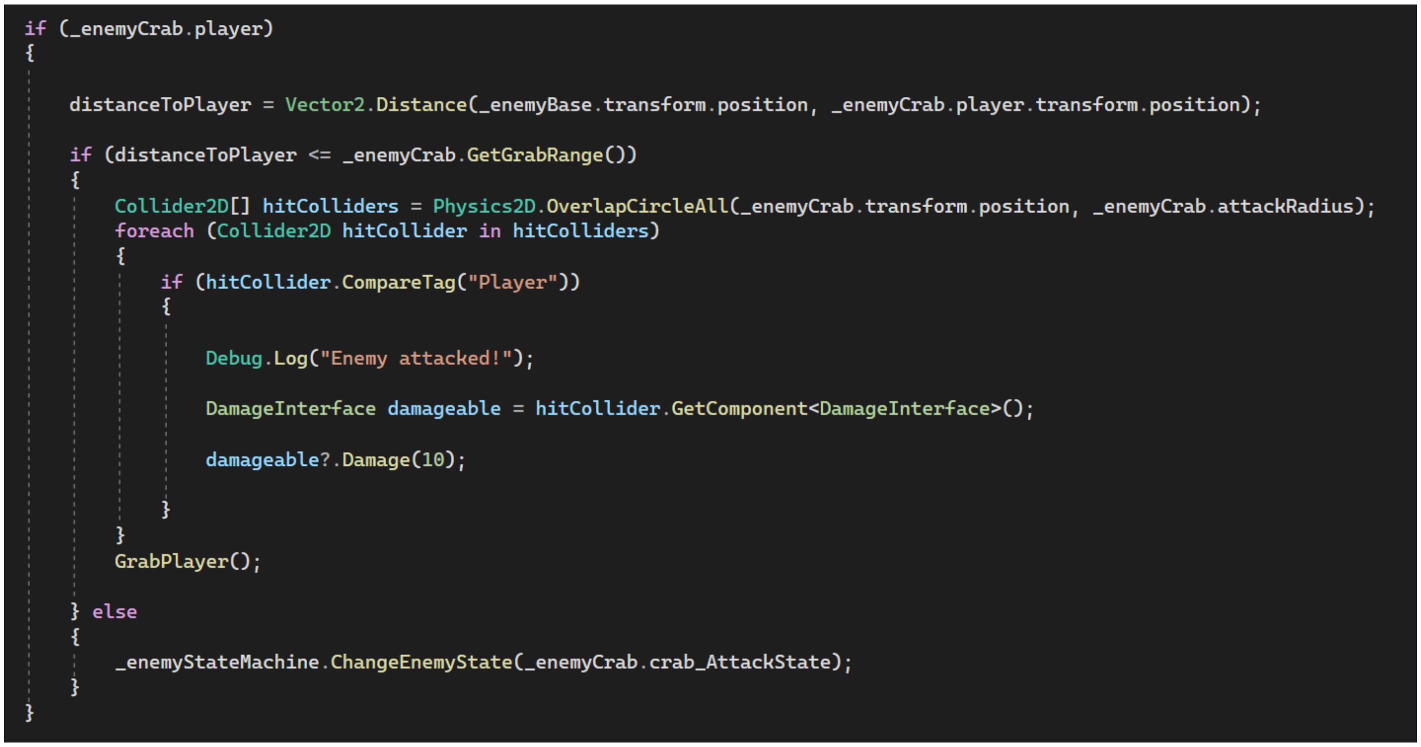
Within this class, there are various methods and variables utilized to handle the enemy's melee attack state. The "_enemyCrab" variable serves as a reference to the particular crab enemy object.

Upon entering the melee attack state, the "StateEnter" method is invoked, and a message of "MELLEATTACK" is logged to the console. Similarly, when exiting the state, the "StateExit" method is called.

During each frame update, the "EveryFrameUpdate" method is executed, while the "PhysicsUpdate" method is responsible for physics-related updates.

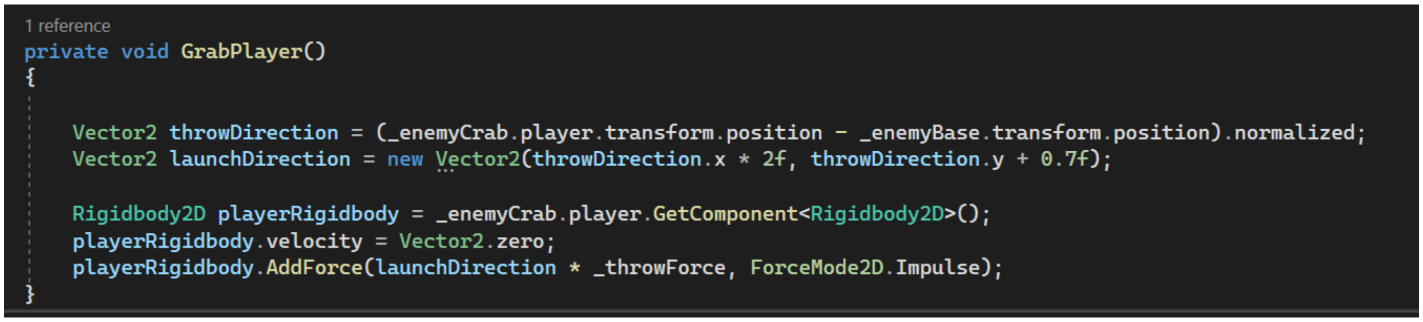
The "DoEnemyChecks" method and the "EnemyTriggerAttack" method are employed to perform necessary checks and trigger the enemy's attack, respectively.

The "EnemyFinishAttack" method is invoked when the enemy concludes its attack. Within this method, it verifies if the player is within a specified range. If the player is within range, the enemy executes an attack by inflicting damage upon the player. Subsequently, the enemy proceeds to grab the player using the "GrabPlayer" method.



```
if (_enemyCrab.player)
{

    distanceToPlayer = Vector2.Distance(_enemyBase.transform.position, _enemyCrab.player.transform.position);

    if (distanceToPlayer <= _enemyCrab.GetGrabRange())
    {
        Collider2D[] hitColliders = Physics2D.OverlapCircleAll(_enemyCrab.transform.position, _enemyCrab.attackRadius);
        foreach (Collider2D hitCollider in hitColliders)
        {
            if (hitCollider.CompareTag("Player"))
            {

                Debug.Log("Enemy attacked!");

                DamageInterface damageable = hitCollider.GetComponent<DamageInterface>();

                damageable?.Damage(10);

            }
        }
        GrabPlayer();

    } else
    {
        _enemyStateMachine.ChangeEnemyState(_enemyCrab.crab_AttackState);
    }
}
```

The "GrabPlayer" method calculates the appropriate direction for throwing the player and applies a force to the player's rigidbody, thereby launching them away.



```
1 reference
private void GrabPlayer()
{

    Vector2 throwDirection = (_enemyCrab.player.transform.position - _enemyBase.transform.position).normalized;
    Vector2 launchDirection = new Vector2(throwDirection.x * 2f, throwDirection.y + 0.7f);

    Rigidbody2D playerRigidbody = _enemyCrab.player.GetComponent<Rigidbody2D>();
    playerRigidbody.velocity = Vector2.zero;
    playerRigidbody.AddForce(launchDirection * _throwForce, ForceMode2D.Impulse);
}
```

In essence, this code delineates the behaviour of an enemy crab when executing a melee attack. It assesses the player's proximity, inflicts damage, and proceeds to grab and throw the player.

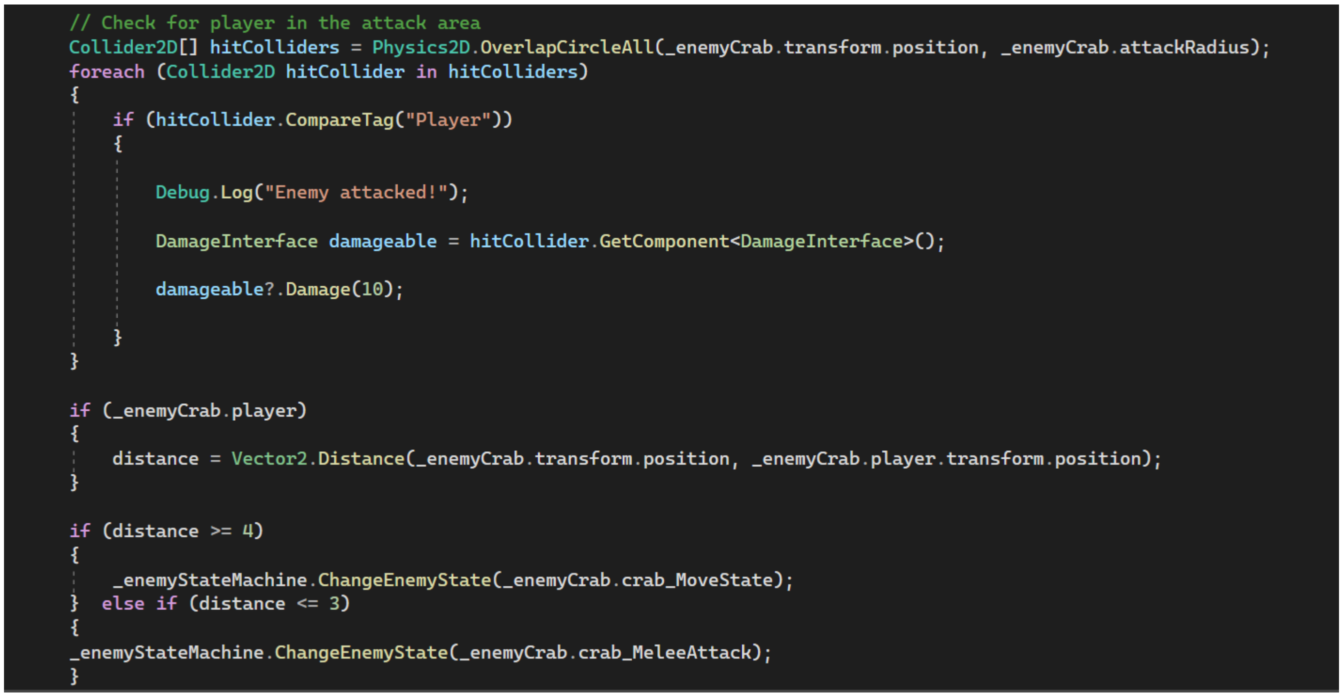
The provided code depicts a class called "E3_Crab_AttackState," which extends from the "Enemy_AttackState" class and implements the "DamageInterface" interface. This class defines the behaviour of an enemy crab during its attack state.

Within this class, various methods and variables are utilized to handle the enemy crab's attack state. The "_enemyCrab" variable serves as a reference to the specific crab enemy object.

Upon entering the attack state, the "StateEnter" method is invoked. In this method, the "statsComponent" variable is assigned to the "StatsComponent" component found in the game object tagged as "Player." A message stating "HELLO FROM ATTACK STATE" is logged to the console.

**During each frame update, the "EveryFrameUpdate" method is executed, while the "PhysicsUpdate" method is responsible for handling physics-related updates.
The "DoEnemyChecks" method and the "EnemyTriggerAttack" method are used for conducting necessary checks and triggering the enemy's attack, respectively.**

When the enemy finishes its attack, the "EnemyFinishAttack" method is called. Inside this method, the code checks for the presence of the player within the attack area by performing a circular overlap check. If the player is detected, the enemy inflicts damage on the player through the implementation of the "DamageInterface" interface.



```
// Check for player in the attack area
Collider2D[] hitColliders = Physics2D.OverlapCircleAll(_enemyCrab.transform.position, _enemyCrab.attackRadius);
foreach (Collider2D hitCollider in hitColliders)
{
    if (hitCollider.CompareTag("Player"))
    {

        Debug.Log("Enemy attacked!");

        DamageInterface damageable = hitCollider.GetComponent<DamageInterface>();

        damageable?.Damage(10);

    }
}

if (_enemyCrab.player)
{
    distance = Vector2.Distance(_enemyCrab.transform.position, _enemyCrab.player.transform.position);
}

if (distance >= 4)
{
    _enemyStateMachine.ChangeEnemyState(_enemyCrab.crab_MoveState);
}  else if (distance <= 3)
{
_enemyStateMachine.ChangeEnemyState(_enemyCrab.crab_MeleeAttack);
}
```

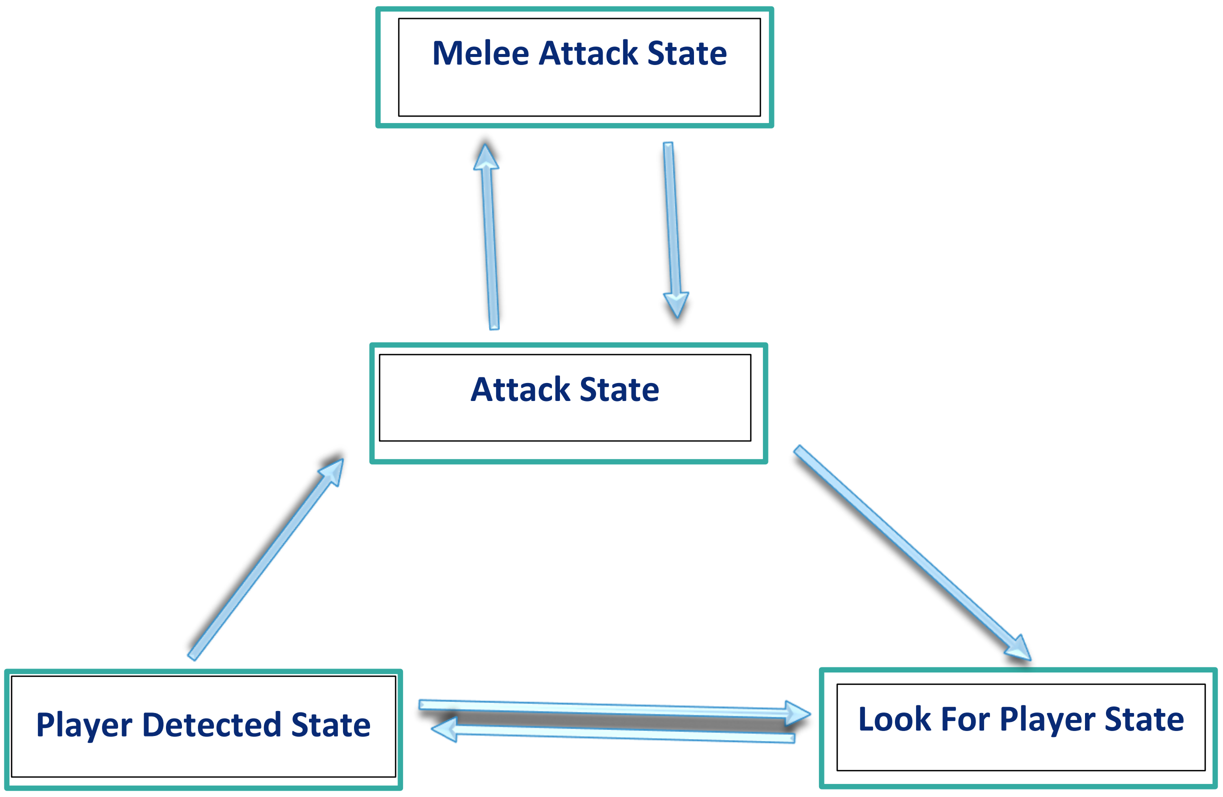
Additionally, the code calculates the distance between the enemy crab and the player. If the distance is equal to or greater than 4 units, the enemy transitions to the crab's move state. Conversely, if the distance is equal to or less than 3 units, the enemy switches to the crab's melee attack state.

In summary, this code governs the behaviour of an enemy crab during its attack state. It examines the player's presence within the attack area, deals damage, and determines whether to move or initiate a melee attack based on the distance between the enemy crab and the player.

To make the state machine complete other scripts were added such as

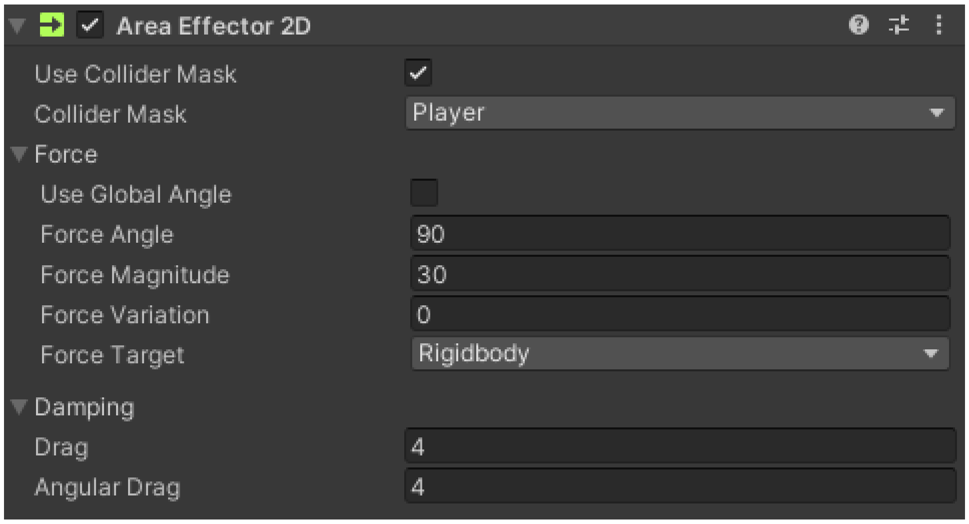
- E3_Crab_AttackState
- E3_Crab_DeadState
- E3_Crab_LookForPlayerState
- E3_Crab_MeleeAttack
- E3_Crab_PlayerDetectedState

How do these states work together: this shows how these specific states work together and how the enemy transitions from one state to another.



Underwater Movement

To implement slowed movement and increased jump height while underwater, we used a combination of two different approaches. The method used to slow the player’s movement consisted of adding a 2D Tilemap Collider as well as a 2D Area Effector to the water tiles and then adjusting the force and damping values.



Area Effector 2D

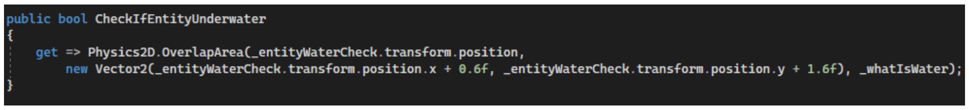
Use Collider Mask	✓
Collider Mask	Player
Force	
Use Global Angle	
Force Angle	90
Force Magnitude	30
Force Variation	0
Force Target	Rigidbody
Damping	
Drag	4
Angular Drag	4

Using a force angle of 90 and force magnitude of 30 means that whenever the player is inside a water tile, they will constantly have a force acting on them pushing upwards. However, since this force is not as strong as the gravity pulling the player downwards it essentially acts as a decrease in gravity, causing the player to fall slower than normal if they are underwater. The drag value being set to 4 also contributes to this effect in addition to slowing the players movement along the horizontal axis.

After adding the area effector, we decided that the lower gravity and slowed movement felt satisfactory, although we thought the underwater movement could be improved by allowing the player to jump even higher. To achieve this, we first needed to add a water layer mask variable and an entity water check transform variable to the Core’s “CollisionSenses” class.

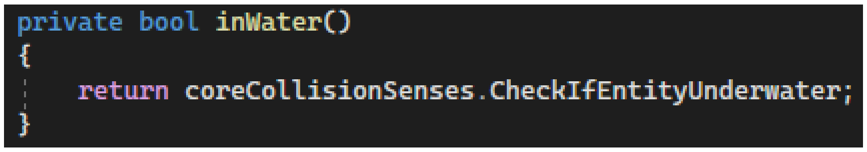
```
private LayerMask _whatIsWater; private Transform _entityWaterCheck;
```

Using these variables as well as float values representing the player character’s height and width, we created a property that can be used to return a Boolean value that indicates whether or not the player is currently touching a water tile.



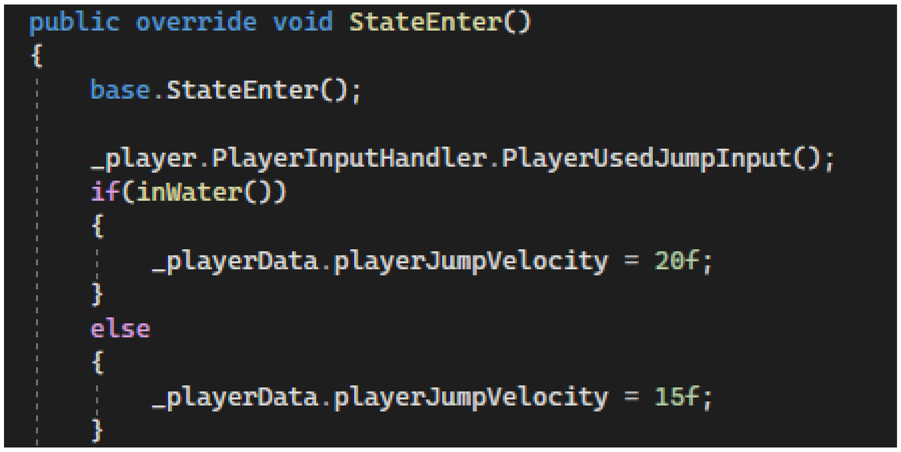
```
public bool CheckIfEntityUnderwater
{
    get => Physics2D.OverlapArea(_entityWaterCheck.transform.position,
        new Vector2(_entityWaterCheck.transform.position.x + 0.6f, _entityWaterCheck.transform.position.y + 1.6f), _whatIsWater);
}
```

This is used inside the “PlayerJumpState” script to create the “inWater” function.



```
private bool inWater()
{
    return coreCollisionSenses.CheckIfEntityUnderwater;
}
```

Subsequently, the “inWater” function is called inside the jump state’s “StateEnter” function, and if it returns ‘true’, the player’s jump velocity variable is increased to 20, otherwise the value is set to the default, 15. This results in a higher jump provided that the player is touching a water tile.



```
public override void StateEnter()
{
    base.StateEnter();

    _player.PlayerInputHandler.PlayerUsedJumpInput();
    if(inWater())
    {
        _playerData.playerJumpVelocity = 20f;
    }
    else
    {
        _playerData.playerJumpVelocity = 15f;
    }
```

Playtesting

To ensure our game is robust and to eliminate as many bugs as possible, we playtest each level we develop thoroughly from start to finish. During a playtest, we do our best to procedurally check that every part of the level is working as expected by using all of the player's movement mechanics, interacting with all of the level's pickups and distinctive features as well as allowing all of the level's enemies to transition between each of their different states.

Playtesting allows us to minimise the presence of unintended features such as gaps in the game level that let the player move out of bounds, as well as bugs that cause certain aspects of the game to behave incorrectly.

Through playtesting we also improve the gameplay experience by fine-tuning each level's layout to complement the movement mechanics, making sure the player must make use of their ability to dash, grapple and wall jump. In addition, we can determine whether we feel the level is too easy or too hard to beat and then make changes to the difficulty accordingly. These changes could be to the environment layout, to the enemies' data values (for example increasing an enemy's health or damage it deals) or they could simply be adjusting the number of enemies present in the level.

Bug Fixing

When a bug is first discovered during a playtest, it is recorded in a comprehensive list that includes all the bugs currently identified within the game and their respective priorities for fixing. This list serves as a reference for us to later address and resolve every bug until none remain unfixed. Since resolving unintended game behaviours is so crucial during development, we decided to employ a well-defined and efficient bug fixing process.

Our step-by-step bug fixing method is as follows:

- The bug with the highest priority for fixing is chosen to be worked on.
- We attempt to reproduce the bug to understand what it is being caused by.
- We analyse and debug the code to locate the root cause.
- We carry out the necessary code changes to resolve the bug.
- The implemented fix is tested to ensure the bug is resolved without introducing any new issues.
- The bug list is updated to display the chosen bug’s new fixed status.

By employing this method, we can maintain a clear understanding of the game's current issues, allowing us to address each bug accordingly based on their assigned priority and avoid being overwhelmed.