



# TREBALL FINAL DE GRAU



ESCOLA  
POLITÈCNICA SUPERIOR  
UNIVERSITAT DE LLEIDA  
INSPIRING THE FUTURE

**Estudiant:** Kevin López Muñoz

**Titulació:** Grau en Disseny Digital i Tecnologies Creatives

**Títol de Treball Final de Grau:** **Cocodrilo Waton: Programación y montaje de un videojuego**

**Director/a:** Marc Balsà Diaz

Presentació

Mes: Setembre

Any: 2022

# Índice

<b>1. Introducción</b>	<b>5</b>
<b>2. Objetivos</b>	<b>6</b>
<b>3. Justificación de la elección</b>	<b>7</b>
<b>4. Softwares</b>	<b>8</b>
4.1. Unity . . . . .	8
4.2. Microsoft Visual Studio . . . . .	10
4.3. Photoshop . . . . .	11
4.4. Github . . . . .	12
<b>5. Lenguaje de programación</b>	<b>13</b>
5.1. C# . . . . .	13
<b>6. Personaje</b>	<b>15</b>
6.1. Controles básicos . . . . .	15
6.1.1. Movimiento lateral . . . . .	16
6.1.2. Salto . . . . .	18
6.1.3. Lanzallamas . . . . .	21
6.2. Power ups . . . . .	22
6.2.1. Doble salto . . . . .	22
6.2.2. Dash . . . . .	25
6.2.3. Bola de fuego . . . . .	27
6.2.4. Gancho . . . . .	30
6.3. Animaciones . . . . .	33
6.4. Vida . . . . .	36
6.5. Interacciones con los enemigos . . . . .	38
6.5.1. Recibir daño . . . . .	38
6.5.2. Hacer daño . . . . .	40
6.6. Muerte . . . . .	41

6.7.	Muslos de carne . . . . .	41
6.8.	Respawn . . . . .	43
6.9.	Seguimiento de la cámara . . . . .	45
<b>7.</b>	<b>Enemigos</b>	<b>47</b>
7.1.	Tortuga . . . . .	47
7.1.1.	Movimiento de patrulla . . . . .	47
7.1.2.	Daño al jugador . . . . .	50
7.1.3.	Esconderse . . . . .	50
7.1.4.	Recibir daño . . . . .	52
7.1.5.	Muerte . . . . .	55
7.2.	Murciélagos . . . . .	56
7.2.1.	Movimiento de patrulla . . . . .	56
7.2.2.	Atacar al jugador . . . . .	56
7.2.3.	Recibir daño . . . . .	58
7.2.4.	Muerte . . . . .	58
7.3.	Araña . . . . .	58
7.3.1.	Idle . . . . .	59
7.3.2.	Perseguir al jugador . . . . .	59
7.3.3.	Recibir daño . . . . .	61
7.3.4.	Muerte . . . . .	61
7.4.	Castor . . . . .	61
7.4.1.	Idle . . . . .	61
7.4.2.	Ataque al jugador . . . . .	61
7.4.3.	Recibir daño . . . . .	63
7.4.4.	Muerte . . . . .	64
<b>8.</b>	<b>Puntuación</b>	<b>65</b>
8.1.	Funcionamiento . . . . .	65
8.2.	Uso . . . . .	66
<b>9.</b>	<b>Checkpoints</b>	<b>68</b>
9.1.	Activación . . . . .	69
<b>10.</b>	<b>UI</b>	<b>71</b>
10.1.	Menú de inicio . . . . .	71
10.1.1.	Selector de nivel . . . . .	73
10.2.	Menú de opciones . . . . .	73
10.3.	UI in-game . . . . .	75
10.4.	Menú de pausa . . . . .	76
10.5.	Nivel completado . . . . .	79

10.6. Derrota . . . . .	80
<b>11. Partículas</b>	<b>81</b>
11.1. Fuego del lanzallamas . . . . .	81
11.2. Lluvia . . . . .	83
11.3. Polvo al caminar . . . . .	86
<b>12. Niveles</b>	<b>88</b>
12.1. Pantano . . . . .	88
12.1.1. Boceto inicial . . . . .	89
12.1.2. Boceto final . . . . .	90
12.2. Cueva . . . . .	90
12.2.1. Boceto inicial . . . . .	91
12.2.2. Boceto final . . . . .	92
12.3. Cueva oscura . . . . .	92
12.3.1. Boceto inicial . . . . .	93
12.3.2. Boceto final . . . . .	94
12.4. Pantano oscuro . . . . .	94
12.4.1. Boceto inicial . . . . .	95
12.4.2. Boceto final . . . . .	96
<b>13. Tilemap</b>	<b>97</b>
13.1. Spritesheets . . . . .	97
13.1.1. Separación de tiles . . . . .	97
13.2. Creación del nivel . . . . .	98
13.2.1. Palettes . . . . .	99
13.2.2. Layers . . . . .	100
13.2.3. Colliders . . . . .	100
13.3. Parallax . . . . .	101
<b>14. Conclusión</b>	<b>103</b>
<b>Bibliografía</b>	<b>105</b>
<b>A. Game Design Document (GDD)</b>	<b>106</b>
A.1. Log . . . . .	106
A.2. Overview . . . . .	106
A.3. Referencias del gameplay . . . . .	107
A.4. Gameplay . . . . .	107
A.5. Target . . . . .	107
A.6. Unique Selling Points (USP) . . . . .	108
A.7. Mecánicas . . . . .	108

A.8.	Dinámicas	109
A.9.	Aesthetics	109
A.10.	Finalidad	109
A.11.	Objetivos	109
A.12.	Blueprint	110
A.13.	Referencias de arte	112
A.14.	Personaje principal	112
A.15.	Enemigos	113
A.15.1.	Tortuga	113
A.15.2.	Murciélagos	113
A.15.3.	Araña	113
A.15.4.	Castor	114
A.16.	Power ups	114
A.16.1.	Doble salto	114
A.16.2.	Dash	114
A.16.3.	Bola de fuego	114
A.16.4.	Gancho	114
A.17.	Cutscenes	114
A.18.	Assets	115

# Capítulo 1

## Introducción

A través de la realización de este proyecto se ha creado un videojuego de plataformas 2D con cámara de desplazamiento lateral y un pequeño toque de Metroidvania.

El videojuego consta de un menú principal con un selector de niveles, un menú de opciones y cuatro pequeños niveles. En cada uno de estos se aprende una nueva mecánica de juego (a la vez que se pone en práctica) y aparece un nuevo enemigo que reacciona de forma diferente a los vistos con anterioridad. Para hacerlo, se ha puesto atención en programar la jugabilidad, la creación de algunas partículas, montar los escenarios y colocar todos los elementos de la UI (User Interface).

# **Capítulo 2**

## **Objetivos**

Este proyecto consta de varios objetivos:

1. El objetivo principal es llegar a crear un videojuego con sus respectivos menús y cuatro niveles distintos entre ellos.
2. Dar un toque distintivo a cada nivel.
3. Añadir una nueva mecánica y enemigo por nivel.
4. Cambiar la ambientación del escenario.

# Capítulo 3

## Justificación de la elección

Lo que motivó principalmente a elegir este proyecto fue una gran pasión hacia los videojuegos, ya que es algo accesible hoy en día desde que se es pequeño, ya sea jugándolos o viendo contenido relacionado en internet.

Las ganas de crear un videojuego aparecieron, al mismo tiempo que aumentaban considerablemente en poco tiempo, al hacer un ciclo superior sobre ellos. Después de tanto estar en contacto con algo, algunas veces es inevitable el querer añadir aunque sea una diminuta aportación, siendo en este caso un pequeño videojuego hecho con todo el cariño posible para que pueda llegar a agradar a quien lo pruebe.

Además de lo mencionado anteriormente, en la asignatura de programación se descubrió la pasión de programar, así que se quiso poner a prueba los conocimientos obtenidos a través de este proyecto.

Tras llegar a la conclusión de que se quería hacer un videojuego, se decidió que Arnau se encargaría de la parte artística del proyecto. Esto se debió a que el tiempo disponible para realizarlo no era suficiente, ya que la complejidad y tiempo que se requiere en la creación de un videojuego es bastante elevada.

# **Capítulo 4**

## **Softwares**

Antiguamente era muy complicado desarrollar un videojuego como los que conocemos hoy en día desde cero. Eran videojuegos simples y programados por una persona, como por ejemplo Pong, un juego donde hay dos líneas y una pelota, siendo el objetivo hacer que el oponente no logre acertar a la pelota.

La programación de antaño se aplicaba directamente en la máquina que iba a permitir su uso, lo que obligaba al programador a tener un gran conocimiento sobre esta. Por otra parte, hoy en día se puede empezar a programar teniendo unos conocimientos básicos, creando una gran diferencia entre ambas épocas.

En cambio, hoy en día está al alcance de todo el que quiera intentarlo, se dispone de muchos softwares gratuitos para la creación de videojuegos, capaces de poder crear videojuegos de todo tipo. Entre estos softwares se encuentra Unity, el cual será usado en este proyecto.

### **4.1. Unity**

Unity es un motor de videojuegos multiplataforma, que permite desarrollar videojuegos para móviles, ordenadores, consolas, realidad virtual, webs e incluso para Smart TV.

Es un software muy práctico gracias a su compatibilidad con los softwares 3D más usados en el mercado y a varias de sus características que facilitan bastante la realización de diferentes acciones, como por ejemplo, actualizar automáticamente los cambios realizados en un archivo sin necesidad de volver a importarlo.

Es el software principal para la realización de este proyecto, donde se junta todo lo realizado para darle vida al videojuego. Con él se han creado los escenarios, la UI y las diferentes partículas usadas, animado los personajes, enemigos y diferentes objetos del juego, programado toda la jugabilidad y añadido la música ambiental.

En Unity se hace todo a través de escenas, cada parte del juego es una diferente. Estas sirven para tenerlo todo separado y mejor ordenado, pudiendo así acceder a la parte del videojuego deseada sin tener que buscar donde se encuentra. También cargan en la memoria del videojuego solo lo que es necesario.

Finalmente, puede exportar todo lo creado en su interior en unos pocos archivos para poder ejecutar el videojuego sin necesidad de estar usando el software y poder compartirlo para que cualquiera pueda jugarlo.

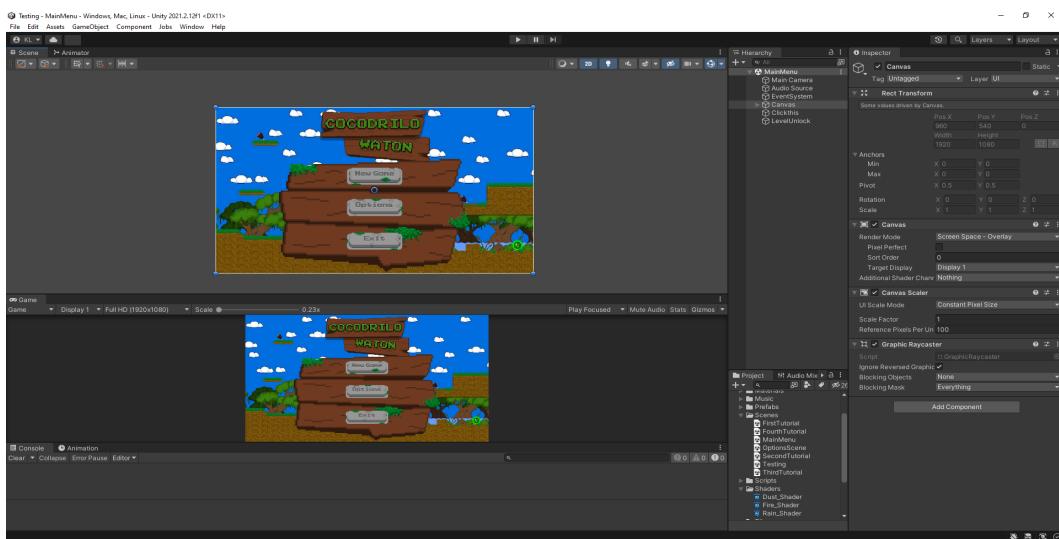
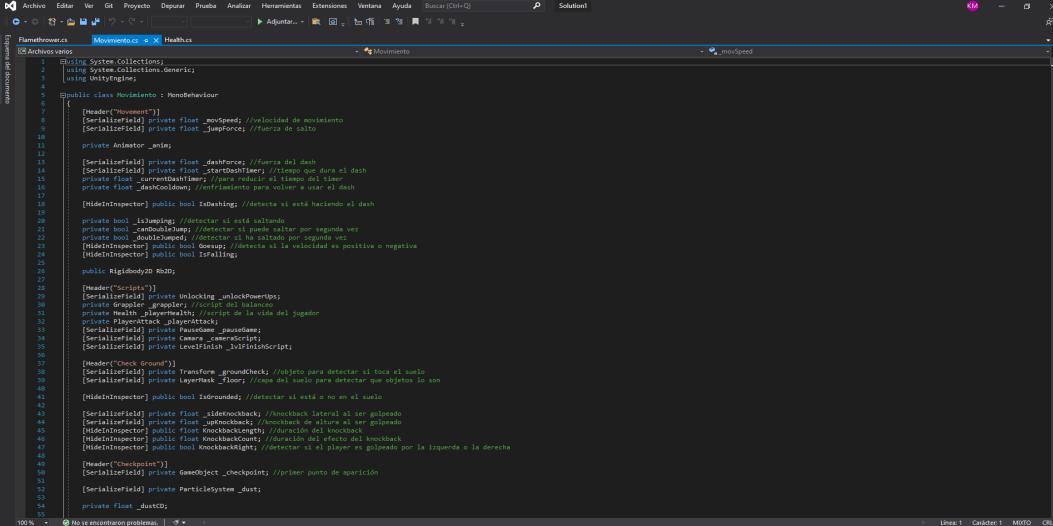


Figura 4.1: Espacio de trabajo de Unity

## 4.2. Microsoft Visual Studio

Microsoft Visual Studio es un entorno de desarrollo integrado para Windows y macOS. Es compatible con varios de los lenguajes de programación más activos en el mercado y permite editarlos, depurarlos y compilarlos, haciendo que los desarrolladores puedan crear aplicaciones y sitios web.

Es el segundo software más importante del proyecto, ya que a través de este se ha realizado toda la programación. Su uso se debe principalmente a que, con la instalación de Unity, este software viene recomendado. Aparte de eso, ha sido el software utilizado en clase, por lo que ya se tenía conocimiento previo sobre cómo usarlo.



The screenshot shows the Microsoft Visual Studio interface with the code editor open. The file being edited is 'Movement.cs'. The code is a C# script for a Unity MonoBehaviour named 'Movement'. It includes imports for System.Collections.Generic, UnityEngine, and UnitySerialization. The class contains fields for movement speed, jump force, and dash variables. It features methods for updating movement, performing a dash, and handling double jumps. The code also includes logic for detecting ground, walls, and checkpoints, as well as particle effects for dust and knockback. Numerous Unity serialization attributes like [Header], [SerializeField], and [HideInInspector] are used throughout the script.

```
1 //using UnityEngine;
2 //using System.Collections.Generic;
3 //using UnitySerialization;
4
5 [Header("Movement")]
6 [SerializeField] private float _moveSpeed; //velocidad de movimiento
7 [SerializeField] private float _jumpForce; //fuerza de salto
8
9 private Animator _animator;
10
11 [Serializable] private float _dashForce; //fuerza del dash
12 [Serializable] private float _startDashTime; //tiempo que dura el dash
13 private float _currentDashTime; //para reducir el tiempo del timer
14 private float _dashCooldown; //tiempo para volver a usar el dash
15
16 [HideInInspector] public bool IsDashing; //detecta si está haciendo el dash
17
18 private bool IsJumping; //detectar si está saltando
19 private bool IsDoubleJumped; //detectar si ha saltado por segunda vez
20 private bool DoubleJumped; //detectar si ha saltado por segunda vez
21 [HideInInspector] public bool IsUp; //detecta si la velocidad es positiva o negativa
22 [HideInInspector] public bool IsFalling;
23
24 public Rigidbody2D Rb2D;
25
26 [Header("Check Ground")]
27 [Serializable] private Transform _groundCheck; //objeto para detectar si toca el suelo
28 [Serializable] private LayerMask _floor; //máscara del suelo para detectar qué objetos lo son
29
30 [HideInInspector] public bool IsGrounded; //detectar si está o no en el suelo
31
32 [Serializable] private float _lateralKnockback; //knockback lateral al ser golpeado
33 [Serializable] private float _upKnockback; //knockback de altura al ser golpeado
34 [Header("Knockback")]
35 [Serializable] public float KnockbackLength; //distribución del knockback
36 [Serializable] private ParticleSystem _knockback; //partículas para el knockback
37 [HideInInspector] public bool KnockbackRight; //detectar si el player es golpeado por la izquierda o la derecha
38
39 [Header("Checkpoint")]
40 [Serializable] private GameObject _checkpoint; //primer punto de aparición
41
42 [Serializable] private float _dustD;
43
44 [Serializable] private float _lateralKnockbackL; //knockback lateral al ser golpeado
45 [Serializable] private float _upKnockbackL; //knockback de altura al ser golpeado
46 [HideInInspector] public float KnockbackLengthL; //distribución del knockback
47 [Serializable] private ParticleSystem _knockbackL; //partículas para el knockback
48 [HideInInspector] public bool KnockbackLeft; //detectar si el player es golpeado por la izquierda o la derecha
49
50 [Header("Checkpoint")]
51 [Serializable] private GameObject _checkpointP; //segundo punto de aparición
52
53 [Serializable] private ParticleSystem _dust;
54
55 private float _dustD;
```

Figura 4.2: Espacio de trabajo de Microsoft Visual Studio

## 4.3. Photoshop

Photoshop es un software creado principalmente para la edición de fotografías, pero también es utilizado para el diseño de arte, siendo así bastante utilizado en la industria de los videojuegos para la realización de los dibujos de los personajes, escenarios, etc.

Su uso en este proyecto ha sido principalmente para juntar los sprites que me pasaba la persona encargada del arte y así crear las “spritesheets”, una imagen (normalmente de gran tamaño) donde se juntan otras muchas imágenes, de un mismo estilo, para poder localizarlas más fácilmente y así tener un mejor orden del material usado, al mismo tiempo que mejoran mucho el rendimiento. Su otro uso ha sido mejorar los bocetos de los niveles para aumentar su calidad y que se entiendan mejor.

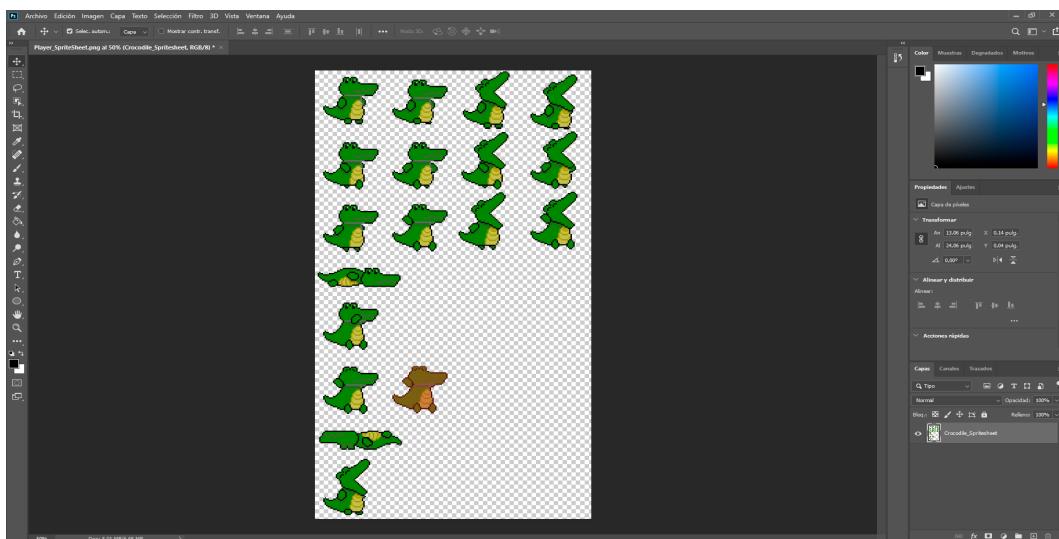


Figura 4.3: Espacio de trabajo de Photoshop

#### 4.4. Github

Github es una plataforma de desarrollo colaborativo, donde se pueden alojar proyectos utilizando su sistema de control de versiones, el cual permite acceder a versiones anteriores del mismo. De gran utilidad en caso de que haya sucedido algún problema y deba ser rectificado.

A través de esta web, se ha podido tener un mayor control sobre el proyecto. Al mismo tiempo que se tenía la seguridad de no perderlo, ya que puedes acceder a él desde cualquier dispositivo y a cualquier versión anterior. Gracias a ello, el tutor ha podido ir haciendo un seguimiento constante del trabajo al tener siempre acceso a los cambios que se realizaban en este.

Para el control de versiones he usado Github Desktop, una versión descargable de Github donde puedes tener un proyecto de la web en una carpeta dentro del ordenador. Al estar ambas cosas vinculadas, cada vez que yo aplicaba un cambio en la carpeta, el software lo detectaba automáticamente y lo añadía a la lista de cambios para el historial de versiones.

Figura 4.4: Espacio de trabajo de Github

# Capítulo 5

## Lenguaje de programación

Un lenguaje de programación es un tipo de lenguaje que le permite a una persona escribir una serie de instrucciones en forma de algoritmos. De esta manera puede controlar el comportamiento de un sistema informático para que ejecute unas órdenes determinadas.

Hay mucha variedad en cuanto a lenguajes de programación, pero los más usados hoy en día son Java, Python, C#, C/C++ y Javascript.

De los lenguajes anteriormente mencionados, en este proyecto se usará C# debido a que es el único lenguaje de programación disponible en Unity.

### 5.1. C#

C# es un lenguaje de programación multiparadigma diseñado para la infraestructura de lenguaje común. Su sintaxis básica deriva de C/C++, aunque incluye mejoras derivadas de otros lenguajes.

Todos los scripts de este proyecto están escritos en C#, principalmente porque es el único lenguaje de programación disponible en Unity, además de ser el que se ha utilizado durante estos meses en clase.

En este proyecto, se centrará principalmente en la explicación del uso de dos variables en concreto de las que dispone este lenguaje, los “float”, que contienen números enteros y decimales, y los “bool”, que tienen dos valores disponibles, verdadero o falso.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5  using UnityEngine.UI;
6
7  public class Health : MonoBehaviour
8  {
9      [SerializeField] private int _maxHealth; //vida máxima del jugador
10     [HideInInspector] public int CurrentHealth; //vida actual del jugador
11
12     [SerializeField] private Image _lifeUI; //array de imágenes
13     [SerializeField] private Sprite[] _lives; //sprite del corazón lleno
14
15     private Movimiento _movementScript; //script de movimiento
16     private PlayerAttack _attackScript; //script de ataque
17
18     [HideInInspector] public float DamageCD; //cooldown para volver a recibir daño
19
20     [HideInInspector] public bool PlayerIsDead; //detectar si el jugador ha muerto
21
22     [HideInInspector] public bool CanGetDamage; //detectar si el jugador puede recibir daño
23
24     private Animator _anim;
25
26     private Transform _enemyPos; //posición del enemigo para el knockback
27
28     [SerializeField] private GameObject _lose; //canvas de derrota
29
30     void Start()
31     {
32         _anim = GetComponent<Animator>();
33
34         _movementScript = GetComponent<Movimiento>();
35         _attackScript = GetComponent<PlayerAttack>();
36
37         CurrentHealth = _maxHealth; //la vida actual será igual a la vida máxima al comenzar
38     }
39
40     private void Update()
41     {
42         DamagedAgain();
43     }
44
45     public void PlayerDamaged(Collision2D collision)
46     {
47         if (CanGetDamage && !PlayerIsDead) //si el jugador puede recibir daño perderá vida
48         {
49             CanGetDamage = false;
50             TakeDamage();
51             DamageCD = 0.5f; //se activa el cooldown para volver a recibir daño
52
53             if (CurrentHealth >= 1) //si la vida es diferente a 0 recibirá el knockback
54             {

```

Figura 5.1: Ejemplo de código con C#

# Capítulo 6

## Personaje

El personaje con el que el jugador se abrirá paso por este videojuego es un cocodrilo llamado Cocodrilo Waton. Un animal conocido por su peligrosidad y que a la mayoría de gente le entrarían ganas de salir corriendo al encontrarse con uno. En este caso, es todo lo contrario, un cocodrilo pequeño y regordete con una sonrisa constante en su rostro que no infunde miedo en absoluto.

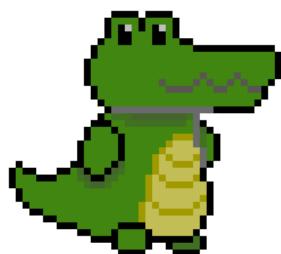


Figura 6.1: Cocodrilo Waton

### 6.1. Controles básicos

Como en todo videojuego, se empieza con unos controles que el jugador tendrá a su disposición desde el principio para lograr avanzar en su aventura. Al ser un plataformas 2D no pueden faltar el movimiento lateral y el salto, este último no es tan necesario como el anterior, pero le da algo más de movilidad al jugador. La última mecánica básica, pero no por ello menos importante, es un lanzallamas que el cocodrilo podrá usar para quemar a quién se interponga en su camino.

### 6.1.1. Movimiento lateral

El movimiento lateral es la mecánica básica en todo videojuego de desplazamiento lateral 2D, ya que solo puedes moverte hacia los lados al no existir profundidad como en un videojuego 3D.

Al ser la mecánica más básica, fue lo primero en ser programado para que así el personaje ya pudiera, como mínimo, desplazarse por el suelo provisional creado en una escena creada con el fin de ir probando las cosas que se iban programando y, una vez funcionasen correctamente, poder moverlas a su escena correspondiente sin riesgo de añadir fallos.

Para empezar a programar el desplazamiento lateral, lo inicial era que el personaje detectase cuando se pulsan las teclas correspondientes y así poder reaccionar con ello. Esta parte era sencilla ya que C# puede detectar automáticamente las teclas asignadas al movimiento lateral con una sola línea de código, poniendo que detecte los inputs horizontales, siendo estos las flechas laterales o, más usado en los videojuegos, las letras A y D. Esto no se puede lograr sin antes crear una variable “float”, la cual guarda la información que devuelve la función de detectar el movimiento, permitiendo saber qué tecla está siendo pulsada. De esta manera, tendrá un valor negativo al desplazarse hacia la izquierda y un valor positivo hacia la derecha.

```
24  private void Update()
25  {
26      Movement = Input.GetAxis("Horizontal"); //detectar el eje horizontal en el movimiento
27 }
```

Figura 6.2: Código de la detección de teclas

Para que el cocodrilo mire hacia la dirección adecuada, se cambia la rotación del eje Y en base al valor del movimiento.

```

339     |     private void FlipPlayer() //depende de la dirección en la que camine mirará hacia ella
340     |     {
341     |     |         if (PlayerManager.PManager.Movement < -0.1f)
342     |     |         {
343     |     |             transform.eulerAngles = new Vector3(0, -180, 0);
344     |     |         }
345     |     |
346     |     |         else if (PlayerManager.PManager.Movement > 0.1f)
347     |     |         {
348     |     |             transform.eulerAngles = new Vector3(0, 0, 0);
349     |     |         }
350     |     }

```

Figura 6.3: Código para el giro

Una vez se pueden obtener esos valores, lo siguiente es añadir velocidad para que el personaje pueda moverse. Se crea una nueva variable “float” que es la velocidad, se le asigna un valor manualmente y, multiplicándola por el valor del movimiento en un nuevo vector2, el personaje podrá moverse en la dirección que el jugador esté pulsando en ese momento.

```

181     |     |     //se aplica velocidad en el eje X para el movimiento
182     |     |     Rb2D.velocity = new Vector2(_movSpeed * PlayerManager.PManager.Movement, Rb2D.velocity.y);

```

Figura 6.4: Código para el movimiento

Una vez aplicado el código de arriba, el personaje ya era capaz de moverse, pero siempre de forma horizontal. Para evitarlo era necesario añadirle gravedad, así el cocodrilo eventualmente regresará al suelo, ya que no tendría mucho sentido que un cocodrilo pudiera volar. Aun así, el jugador podrá “moverse” en el aire ya que, una vez salta, la gravedad lo único que hace es devolverle al suelo.

Toda acción del cocodrilo dispone de su propia animación que se activa mediante código aparte de ponerla mediante el “Animator”, del cual se hablará más adelante. Por ello, si el jugador se mueve, hará la animación de caminar y si está estático hará el idle.

```

142     private void PlayerMovement()
143     {
144         if (!_playerAttack.IsAttacking)
145         {
146             //si el jugador no se mueve hará el idle
147             if (PlayerManager.PManager.Movement == 0 && IsGrounded && !IsDashing)
148             {
149                 _anim.SetBool("_isStatic", true);
150                 _anim.SetBool("_isFalling", false);
151             }
152         }
153         else
154         {
155             _anim.SetBool("_isStatic", false);
156         }
157         //si el jugador se mueve en el suelo hará la animación de caminar
158         if (PlayerManager.PManager.Movement != 0 && IsGrounded && !IsDashing)
159         {
160             _anim.SetBool("_isWalking", true);
161             _anim.SetBool("_isFalling", false);
162         }
163     }
164     else
165     {
166         _anim.SetBool("_isWalking", false);
167     }
168 }
169 }
170
171     //se aplica velocidad en el eje X para el movimiento
172     Rb2D.velocity = new Vector2(_movSpeed * PlayerManager.PManager.Movement, Rb2D.velocity.y);
173 }
```

Figura 6.5: Función completa del movimiento

### 6.1.2. Salto

Inicialmente el salto iba a ser constante, una vez saltase siempre sería la misma altura, pero más adelante se pensó en hacerlo progresivo, es decir, mientras más rato se pulsa la tecla más salta (hasta cierto límite). Había dos posibilidades por las cuales empezar: crear un detector de suelo para que el cocodrilo solo fuese capaz de saltar tocando el suelo o añadir fuerza en el eje vertical para que el personaje se elevase. Ambas opciones eran totalmente viables para comenzar, pero se decidió hacer primero el detector de suelo.

Para hacerlo era necesario crear una capa que actuase como suelo para poder añadirla a las tiles (explicadas en el punto 13) que serían puestas en un futuro e incorporar un detector de suelo al cocodrilo para que pueda detectar cuándo saltar y cuándo no. El detector de suelo es un objeto vacío del cual se coge la posición, se le añade un pequeño círculo invisible y se le dice qué capa debe reconocer cuando la toque. Para finalizar, faltaba crear una variable “bool”, el cual devolvería verdadero o falso en base al reconocimiento anterior. De esta manera, el cocodrilo solo podrá saltar cuando el círculo toque el objeto con la capa del suelo.

```
223     //añadir rango al objeto para detectar el suelo  
224     IsGrounded = Physics2D.OverlapCircle(_groundCheck.position, 0.15f, _floor);
```

Figura 6.6: Código detectar el suelo

El siguiente paso era ir a por la adición de fuerza al eje vertical para simular el salto. Se crea una variable “float” para el salto, asignándole un valor manualmente, y se utiliza para cambiarla por el valor del eje vertical en un nuevo vector2. Como de esta manera solo aplica fuerza sin ninguna condición, el personaje se elevará infinitamente. Para evitarlo, se le añade la condición de que solo lo haga si el personaje está en el suelo, pero ahora saltará automáticamente cada vez que toque el suelo. Se añade una nueva condición dentro de la anterior, la cual hará que el personaje solo salte si el jugador pulsa la tecla W o la flecha superior.

Para complementar con las animaciones y que funcionasen lo mejor posible, se creó otro “bool” para detectar si el cocodrilo está saltando que se activa al saltar y se desactiva cuando el jugador deja de pulsar la tecla correspondiente.

Al ser un salto progresivo, cuando el jugador dejase de pulsar la tecla, debía caer, por lo que se creaba un nuevo vector2 que volviera cero la velocidad vertical para que en ese momento empezase a caer. El salto tiene su propia animación, la cual irá de la mano con el “bool” para saber si está saltando, ya que si lo está haciendo, se activará la animación de salto.

```

218     private void Jump()
219     {
220         IsGrounded = Physics2D.OverlapCircle(_groundCheck.position, 0.15f, _floor); //añadir rango al objeto para detectar el suelo
221
222         if (IsGrounded)
223         {
224             //PlayerMovement(); //el jugador solo podrá moverse en el suelo
225             //FlipPlayer();
226
227             if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.UpArrow)) //si el jugador está en el suelo y pulsa "W" saltará
228             {
229                 _isJumping = true;
230                 Rb2D.velocity = new Vector2(Rb2D.velocity.x, _jumpForce); //se aplica velocidad al eje Y para saltar
231
232                 if (!_playerAttack.IsAttacking)
233                 {
234                     _anim.SetBool("_isJumping", true);
235                 }
236             }
237
238             if (_isJumping) //si el jugador está en el aire y deja de pulsar "W" no subirá más
239             {
240                 if (Input.GetKeyUp(KeyCode.W) || Input.GetKeyUp(KeyCode.UpArrow))
241                 {
242                     _isJumping = false;
243                     _anim.SetBool("_isJumping", false);
244                     _canDoubleJump = true;
245                 }
246             }
247         }
248     }

```

Figura 6.7: Función completa del salto

De esta manera, si el jugador dejaba de pulsar la tecla cuando el cocodrilo estaba cayendo, hacía una parada en seco y empezaba a caer de nuevo, quedando bastante mal visualmente. La forma de arreglar esto era crear un nuevo “bool” detectando si la velocidad vertical era positiva y añadirlo como condición para que la velocidad se volviera 0. Al hacer esto, si el cocodrilo cae, la velocidad es negativa, no entraría en la condición y simplemente indicaría que ha dejado de saltar.

```

243         //si el jugador está en el aire y deja de pulsar "W" no subirá más
244         if (_isJumping)
245         {
246             if (Input.GetKeyUp(KeyCode.W) || Input.GetKeyUp(KeyCode.UpArrow))
247             {
248                 _isJumping = false;
249                 _anim.SetBool("_isJumping", false);
250                 _canDoubleJump = true;
251
252                 if (Goesup)
253                 {
254                     //la velocidad del eje Y se vuelve 0 para caer
255                     Rb2D.velocity = new Vector2(Rb2D.velocity.x, 0);
256                 }
257             }
258         }

```

Figura 6.8: Código con adición del ”bool” Goesup

### 6.1.3. Lanzallamas

Al inicio del videojuego es el único ataque a distancia del que dispone el cocodrilo y es bastante útil para evitar a los enemigos. También daña varias veces por segundo, por lo que su daño es bastante menor en comparación a los otros ataques, así el jugador no abusa del ataque.

El lanzallamas es un sistema de partículas, por lo que no tiene forma de colisionar con los enemigos. Por ende, el cocodrilo tendrá un objeto que servirá como colisionador para el sistema de partículas, con un script en su interior que su única función es indicar el daño del lanzallamas, indicado por una variable “float”.

```
5     public class Flamethrower : MonoBehaviour
6     {
7         public float FlamethrowerDamage;
8     }
```

Figura 6.9: Script del lanzallamas

A parte del sistema de partículas, el cual fue realmente difícil obtener un resultado aceptable, la programación de este fue relativamente sencilla. Para que el cocodrilo expulsase las llamas lo único necesario era activarlas al pulsar la tecla, en este caso la E, y desactivarlas dejando de pulsar la misma tecla. Las partículas se activan al mismo tiempo que la animación del cocodrilo cambia a ataque, por ello había que activar la animación pertinente al pulsar la tecla.

Para futuras interacciones con los enemigos, elementos del mapa, animaciones o incluso para añadir más condiciones a otras acciones del personaje, también era necesario crear dos “bool” que detectasen si estaba atacando y si era con el lanzallamas, ya que más adelante dispondrá de otro ataque a distancia.

```
103     private void Flamethrower()
104     {
105         if (Input.GetKeyDown(KeyCode.E) && !_movementScript.IsDashing)
106         {
107             IsAttacking = true;
108             _flamethrower = true;
109             AttackAnimations();
110         }
111
112         if (Input.GetKeyUp(KeyCode.E))
113         {
114             IsAttacking = false;
115             _flamethrower = false;
116             AttackCancel();
117         }
118     }
```

Figura 6.10: Función del lanzallamas

## 6.2. Power ups

Los power ups, también conocidos como potenciadores o mejoras, existen en la mayoría de los videojuegos de plataformas. Este nombre se les da debido a que son mecánicas obtenidas mientras el jugador avanza por el juego y no están disponibles desde un inicio. En este caso, el jugador se encontrará con la necesidad de obtener la correspondiente habilidad de cada nivel si quiere seguir avanzando, ya que sin su obtención habrá zonas a las que no podrá acceder.

Para obtener el power up correspondiente, deberá buscar por el nivel una bola luminosa con movimiento en el aire que en su interior tiene una imagen del cocodrilo con el respectivo poder que desbloqueará.

Cada vez que una de las bolas sea obtenida, su “bool” creado en un script exclusivo para el desbloqueo de los power ups pasará a verdadero para que el jugador pueda usarlo.

```
5     public class Unlocking : MonoBehaviour
6     {
7         public bool AviableDoubleJump;
8
9         public bool AviableDash;
10
11        public bool AviableBreakWalls;
12
13        public bool AviableGrap;
14    }
```

Figura 6.11: Script de desbloqueo de habilidades

### 6.2.1. Doble salto

El primer power up con el que se topará el jugador es el doble salto, una pequeña mejora en cuanto a la potencia de salto del cocodrilo, lo que le permitirá alcanzar zonas más altas a las que es imposible de acceder con el primer salto. Este segundo salto se podrá hacer en el aire justo después de hacer el primero, teniendo un tercio de la potencia inicial.

La forma de saber si el personaje puede volver a saltar es crear una variable “bool” para detectarlo.

Para la programación del doble salto, las condiciones para que pueda hacerlo son que pueda volver a saltar y que no esté en el suelo.

```
263     //si el jugador no está en el suelo y ha saltado una vez, podrá saltar de nuevo pulsando "W" o la flecha hacia arriba
264     if (_canDoubleJump && !IsGrounded)
```

Figura 6.12: Condición para hacer el doble salto

Si ambas se cumplen y el jugador pulsa W o la flecha superior, la potencia de salto se dividirá en un tercio, se aplicará esa fuerza con un nuevo vector2 y se volverá a multiplicar la potencia de salto para que vuelva a su valor original y el salto principal funcione correctamente. Debido a que la acción es saltar, también se indicará que inicie la animación de salto.

Acto seguido se crea una nueva variable “bool” para saber si ha hecho ese segundo salto.

```
263     //si el jugador no está en el suelo y ha saltado una vez, podrá saltar de nuevo pulsando "W" o la flecha hacia arriba
264     if (_canDoubleJump && !IsGrounded)
265     {
266         if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.UpArrow))
267         {
268             _jumpForce /= 1.5f; //el segundo salto sera mas chiquito
269             Rb2D.velocity = new Vector2(Rb2D.velocity.x, _jumpForce);
270             _jumpForce *= 1.5f; //la fuerza de salto vuelve al valor original
271             _doubleJumped = true;
272
273         if (!_playerAttack.IsAttacking)
274         {
275             _anim.SetBool("_isJumping", true);
276         }
277     }
```

Figura 6.13: Código para el doble salto

Como el salto es progresivo se necesita que cuando el jugador deje de presionar la tecla el cocodrilo vuelva a bajar. Por ello, cuando la tecla deje de ser presionada ambas condiciones del doble salto pasarán a ser falsas, cancelará la animación de salto y si el personaje tiene una velocidad vertical positiva pasará a ser 0 para que baje en ese momento.

```

261     private void DoubleJump()
262     {
263         //si el jugador no está en el suelo y ha saltado una vez, podrá saltar de nuevo pulsando "W" o la flecha hacia arriba
264         if (_canDoubleJump && !IsGrounded)
265         {
266             if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.UpArrow))
267             {
268                 _jumpForce /= 1.5f; //el segundo salto sera mas chiquito
269                 Rb2D.velocity = new Vector2(Rb2D.velocity.x, _jumpForce);
270                 _jumpForce *= 1.5f; //la fuerza de salto vuelve al valor original
271                 _doubleJumped = true;
272
273                 if (!_playerAttack.IsAttacking)
274                 {
275                     _anim.SetBool("_isJumping", true);
276                 }
277             }
278         }
279
280         if (_doubleJumped)
281         {
282             if (Input.GetKeyUp(KeyCode.W) || Input.GetKeyUp(KeyCode.UpArrow))
283             {
284                 _canDoubleJump = false;
285                 _doubleJumped = false;
286                 _anim.SetBool("_isJumping", false);
287
288                 if (Goesup)
289                 {
290                     Rb2D.velocity = new Vector2(Rb2D.velocity.x, 0);
291                 }
292             }
293         }
294     }

```

Figura 6.14: Función del doble salto

Finalmente, para que no pueda ser usado desde un principio, falta añadir la condición de que haya obtenido la bola de habilidad del doble salto. Para ello se crea un script aparte que llevará el objeto del power up, donde detectará si el cocodrilo entra en contacto con dicho objeto y acto seguido permitirá al jugador usar el doble salto.

```

10     private void OnTriggerEnter2D(Collider2D other)
11     {
12         if (other.GetComponent<Movimiento>() != null)
13         {
14             _unlockPowerUps.AvailableDoubleJump = true;
15
16             _doubleJumpText.DespawnTime = 6.5f;
17
18             Destroy(this.gameObject);
19         }
20     }

```

Figura 6.15: Script para el desbloqueo del doble salto

```

99      if (_unlockPowerUps.AvailableDoubleJump == true)
100     {
101       DoubleJump();
102     }

```

Figura 6.16: Código para permitir usar el doble salto

### 6.2.2. Dash

El segundo power up que encontrará el jugador es el “dash”, un pequeño impulso de velocidad hacia la dirección en que el personaje esté mirando. Su utilidad no es solo ese pequeño impulso, también servirá para pasar por zonas estrechas por las cuales el cocodrilo no puede pasar de pie y evitan que siga avanzando.

Para empezar a programar se empieza creando las variables necesarias. En este caso se necesitarán tres “float” y un “bool”. Los “float” son para la fuerza del impulso, saber cuánto durará y el tiempo que lleva activo. Por otra parte, el “bool” nos indicará si el personaje está usando el impulso.

La fuerza del impulso y cuánto durará se aplican manualmente para ponerlo como más guste. Una vez decididos, lo primero es cambiar el movimiento por la fuerza del impulso, multiplicando esta por el lado derecho, lado hacia el que siempre mirará el cocodrilo debido a que al cambiar de dirección rota el eje.

```

196      Rb2D.velocity = transform.right * _dashForce;

```

Figura 6.17: Código para aplicar la fuerza del dash

Para poder aplicar esta fuerza, hace falta añadir la condición de pulsar una tecla, en este caso el shift izquierdo. Cuando la tecla sea pulsada, el “bool” para saber si está usando el impulso pasará a verdadero, se aplicará el valor al contador de tiempo, se cambiará la velocidad del cocodrilo a 0 y no se verá afectado por la gravedad.

```

186     if (Input.GetKeyDown(KeyCode.LeftShift)) //si pulsa shift empezará el dash
187     {
188         IsDashing = true;
189         _currentDashTimer = _startDashTimer;
190         Rb2D.velocity = Vector2.zero;
191         Rb2D.gravityScale = 0;
192     }

```

Figura 6.18: Código para usar el dash

A continuación entrará en acción la duración, restándole tiempo mientras el personaje esté usando el impulso hasta llegar a 0 y la animación cambiará a la adecuada. Cuando el contador alcance el valor anterior, dejará de usar el impulso, la gravedad volverá a ser la normal y la animación del impulso se desactivará.

Para evitar que el jugador lo use constantemente, se creará otra variable “float” para añadir un enfriamiento, dándole un valor al finalizar el impulso y restándole tiempo cuando no esté en uso.

```

187     private void Dash()
188     {
189         //si pulsa shift y el dash no está en cooldown empezará el dash
190         if (Input.GetKeyDown(KeyCode.LeftShift) && _dashCooldown <= 0)
191         {
192             IsDashing = true;
193             _currentDashTimer = _startDashTimer;
194             Rb2D.velocity = Vector2.zero;
195             Rb2D.gravityScale = 0;
196         }
197
198         //si ha pulsado el shift aplicará la fuerza del dash a la velocidad del jugador
199         if (IsDashing)
200         {
201             Rb2D.velocity = transform.right * _dashForce;
202             _currentDashTimer -= Time.deltaTime;
203             _anim.SetBool("_isDashing", true);
204
205             //si finaliza la duración del dash o el jugador es golpeado, finalizará
206             if (_currentDashTimer <= 0 || !_playerHealth.CanGetDamage)
207             {
208                 IsDashing = false;
209                 _dashCooldown = 2;
210                 _anim.SetBool("_isDashing", false);
211                 Rb2D.gravityScale = 5;
212             }
213         }
214
215         else if (!IsDashing)
216         {
217             if (_dashCooldown >= 0)
218             {
219                 _dashCooldown -= Time.deltaTime;
220             }
221         }
222     }

```

Figura 6.19: Función del dash

Finalmente se añadirá la condición de obtener la bola de habilidad del dash para poder acceder a su uso.

```
11     private void OnTriggerEnter2D(Collider2D other)
12     {
13         if (other.GetComponent<Movimiento>() != null)
14         {
15             _unlockPowerUps.AviableDash = true;
16
17             _dashText.DespawnTime = 6.5f;
18
19             Destroy(this.gameObject);
20         }
21     }
```

Figura 6.20: Script para el desbloqueo del dash

```
104     if (_unlockPowerUps.AviableDash == true)
105     {
106         Dash();
107     }
```

Figura 6.21: Código para permitir usar el dash

### 6.2.3. Bola de fuego

La bola de fuego será el segundo y último ataque a distancia que pueda obtener el jugador, siendo este mucho más poderoso que el lanzallamas, ya que puede recorrer largas distancias siempre que no colisione y es capaz de matar a los enemigos de un solo golpe.

Debido a su gran potencia, tendrá un pequeño enfriamiento para que el jugador no pueda abusar de este ataque y refine su puntería para no fallar ante los enemigos y quedar desprotegido durante unos segundos.

Este ataque son dos sprites que se van alternando en un animación para añadir realismo al movimiento.

Lo primero, al crear la bola de fuego, como es un objeto que va aparte del cocodrilo, es crear un punto de disparo para tener algo que la vincule al personaje y añadir el objeto en el script para que el personaje pueda lanzarla.

Al ser un objeto aparte, necesita su propio script para interactuar con el entorno y tener su propio movimiento. En él se crearán dos variables “float” para la velocidad y el daño.

La velocidad se multiplicará por el tiempo y la dirección y se le sumará a la posición actual de la bola de fuego para que se mueva constantemente.

```
11     void Update()
12     {
13         transform.position += _fireballSpeed * Time.deltaTime * transform.right;
14     }
```

Figura 6.22: Código para el movimiento de la bola de fuego

Para que no se mueva permanentemente, necesita ser destruida en algún momento. Esto sucederá cuando colisione con las tiles del suelo o la pared, detectadas a través de su capa correspondiente.

```
16     private void OnTriggerEnter2D(Collider2D other)
17     {
18         if (other.gameObject.layer == LayerMask.NameToLayer("Wall"))
19         {
20             Destroy(gameObject);
21         }
22         if (other.gameObject.layer == LayerMask.NameToLayer("Floor"))
23         {
24             Destroy(gameObject);
25         }
26     }
27 }
```

Figura 6.23: Código para la destrucción de la bola de fuego

Cuando la bola de fuego ya tiene movimiento, hay que hacer que el cocodrilo la dispare. Para ello hay que asignar una tecla, como en todas las demás acciones del cocodrilo, siendo la Q en este caso. En cuanto el jugador pulse la tecla, se instanciará el objeto referenciado en la primera imagen, el cual es la bola de fuego, en la posición asignada y el “bool” creado para el lanzallamas que detecta si está atacando pasará a verdadero ya que esta habilidad también es un ataque.

```
93     if (Input.GetKeyDown(KeyCode.Q) && !_movementScript.IsDashing)
94     {
95         IsAttacking = true;
96         Instantiate(_fireball, _shootPoint.transform.position, transform.rotation);
97     }
```

Figura 6.24: Código para la creación de la bola de fuego

La animación del ataque debe ser corta, por eso se crea una nueva variable “float” para su duración y otra para la demora entre disparo y disparo como he mencionado anteriormente, ambas aplicadas cuando el jugador pulse la

tecla. Para evitar que la bola sea disparada simultáneamente con el lanzallamas, se añade una nueva condición que solo permite dispararla si el personaje no está atacando.

```

89     □    private void ShootFireball()
90     {           if (!IsAttacking)
91     □    {
92     □    if (Input.GetKeyDown(KeyCode.Q) && !_movementScript.IsDashing)
93     □    {
94         IsAttacking = true;
95         Instantiate(_fireball, _shootPoint.transform.position, transform.rotation);
96         _attackTime = 0.15f;
97         _timeBtwShots = 2f;
98     }
99   }
100  }
101 }
```

Figura 6.25: Función para disparar la bola de fuego

Lo siguiente es reducir el tiempo de enfriamiento para que el jugador pueda volver a disparar la bola de fuego, por lo tanto se le restará tiempo al valor mientras este sea mayor a cero. Cuando el valor llegue al mencionado, el jugador podrá volver a disparar.

```

120     □    private void TimeToShoot()
121     {           if (_timeBtwShots <= 0 && !_grapplerScript.IsGrappling)
122     □    {
123         ShootFireball();
124     }
125     □    else
126     {
127         _timeBtwShots -= Time.deltaTime;
128     }
129   }
130 }
```

Figura 6.26: Función para el enfriamiento entre disparos

Finalmente, falta que solo pueda ser disparada una vez obtenida la bola de habilidad de la bola de fuego.

```

10     □    private void OnTriggerEnter2D(Collider2D other)
11     {           if (other.GetComponent<Movimiento>() != null)
12     □    {
13         _unlockPowerUps.AvailableBreakWalls = true;
14         _breakWallsText.DespawnTime = 6.5f;
15         Destroy(this.gameObject);
16     }
17   }
18 }
```

Figura 6.27: Script para el desbloqueo de la bola de fuego

```

48     if (_unlockPowerUps.AviableBreakWalls)
49     {
50         TimeToShoot();
51     }

```

Figura 6.28: Código para permitir usar la bola de fuego

### 6.2.4. Gancho

El gancho es la última habilidad obtenible por el personaje, una habilidad que podrá usar para agarrarse a las raíces que encontrará por el último nivel y balancearse con ellas, pudiendo evitar caerse mientras avanza.

El objeto necesario para el balanceo es externo al personaje, por tanto habrá que crear un punto de lanzamiento y el objeto, referenciando este para que el personaje pueda acceder a él.

En cuanto al funcionamiento del gancho, se necesitará asignar una tecla para su uso, siendo en este caso el Espacio, y una variable “bool” para saber si lo está usando. Al pulsar la tecla, se creará un nuevo vector2 cogiendo la posición de las raíces y se asignará dicha posición al final de la línea que se creará para simular el balanceo. Se conecta la línea mencionada al cubo y se activa, volviendo verdadera la variable del balanceo y activando la animación correspondiente.

```

52     if (Input.GetKeyDown(KeyCode.Space)) //si el jugador pulsa el "espacio" se balanceará
53     {
54         Vector2 cubePosition = new Vector2(_cubito.transform.position.x, _cubito.transform.position.y);
55         _lineRenderer.SetPosition(0, cubePosition); //posición del vértice en el cubo
56         _distanceJoint.connectedAnchor = cubePosition; //la linea se conecta a la posición del cubo
57         _distanceJoint.enabled = true;
58         _lineRenderer.enabled = true;
59         IsGrappling = true; //cambia el estado a balanceo
60         _anim.SetBool("_isFalling", false);
61         _anim.SetBool("_isJumping", false);
62         _anim.SetBool("_isDashing", false);
63         _anim.SetBool("_isGrappling", true);
64     }

```

Figura 6.29: Código para usar el gancho

En caso de que la línea esté activa, se le dará la posición inicial del punto de lanzamiento creado anteriormente.

```
75     if (_distanceJoint.enabled)
76     {
77         //posición del vértice en el jugador
78         _lineRenderer.SetPosition(1, _grapPoint.transform.position);
79     }
```

Figura 6.30: Código para aplicar la posición del personaje

Cuando el jugador deje de pulsar la tecla, se desactivará todo lo activado con anterioridad, haciendo que el cocodrilo deje de balancearse.

```
66     //si el jugador deja de pulsar el "espacio" dejará de balancearse
67     else if (Input.GetKeyUp(KeyCode.Space))
68     {
69         _distanceJoint.enabled = false;
70         _lineRenderer.enabled = false;
71         IsGrappling = false; //vuelve al estado normal
72         _anim.SetBool("_isGrappling", false);
73     }
```

Figura 6.31: Código para desactivar el gancho

Para finalizar esta función, se añadirá que el personaje use la animación de balanceo. Una vez completa, el personaje se balanceará desde cualquier parte y siempre en la posición de la raíz original.

```

50    private void Grappling()
51    {
52        if (Input.GetKeyDown(KeyCode.Space)) //si el jugador pulsa el "espacio" se balanceará
53        {
54            Vector2 cubePosition = new Vector2(_cubito.transform.position.x, _cubito.transform.position.y);
55            _lineRenderer.SetPosition(0, cubePosition); //posición del vértice en el cubo
56            _distanceJoint.connectedAnchor = cubePosition; //la linea se conecta a la posición del cubo
57            _distanceJoint.enabled = true;
58            _lineRenderer.enabled = true;
59            IsGrappling = true; //cambia el estado a balanceo
60            _anim.SetBool("_isFalling", false);
61            _anim.SetBool("_isJumping", false);
62            _anim.SetBool("_isDashing", false);
63            _anim.SetBool("_isGrappling", true);
64        }
65
66        else if (Input.GetKeyUp(KeyCode.Space)) //si el jugador deja de pulsar el "espacio" dejará de balancearse
67        {
68            _distanceJoint.enabled = false;
69            _lineRenderer.enabled = false;
70            IsGrappling = false; //vuelve al estado normal
71            _anim.SetBool("_isGrappling", false);
72        }
73
74        if (_distanceJoint.enabled)
75        {
76            _lineRenderer.SetPosition(1, _grapPoint.transform.position); //posición del vértice en el jugador
77        }
78    }

```

Figura 6.32: Función del gancho

Para que no se balancee desde cualquier parte, es necesario detectar la raíz con la que el personaje entra en contacto. Unity tiene una funcionalidad que se llama “Trigger”, la cual sirve para saber si un objeto está en el rango de otro, función que se usará para saber dónde y cuándo tiene que balancearse. Cuando el cocodrilo entre dentro del rango, una nueva variable “bool” pasará a verdadera para que pueda balancearse.

De la misma manera, cuando salga del rango pasará a falso.

```

80     private void OnTriggerEnter2D(Collider2D other)
81     {
82         if (other.gameObject.name == "Grap") //si entra dentro del rango activará la detección
83         {
84             _triggerActive = true;
85             _cubito = other.gameObject;
86         }
87     }
88 }
89
90     private void OnTriggerExit2D(Collider2D other)
91     {
92         if (other.gameObject.name == "Grap") //si sale de rango desactiva la detección
93         {
94             _triggerActive = false;
95         }
96     }

```

Figura 6.33: Código para detectar el rango del gancho

Para finalizar este power up, necesitará la obtención de la bola de habilidad del gancho para su uso.

```

11     private void OnTriggerEnter2D(Collider2D other)
12     {
13         if (other.GetComponent<Movimiento>() != null)
14         {
15             _unlockPowerUps.AviableGrap = true;
16             _grapplerText.DespawnTime = 6.5f;
17             Destroy(this.gameObject);
18         }
19     }
20 }
21

```

Figura 6.34: Script para el desbloqueo del gancho

```

40     if (_triggerActive) //si está dentro de rango activa la función
41     {
42         if (_unlockPowerUps.AviableGrap == true) //debe coger el powerup para activarlo
43         {
44             Grappling();
45         }
46     }

```

Figura 6.35: Código para permitir usar el gancho

### 6.3. Animaciones

Las animaciones, al ser en 2D, son el cambio constante del sprite del objeto o que dicho objeto se mueva automáticamente sin la influencia externa de otro objeto.

Estas han sido hechas con el componente de Unity “Animator”. Este componente sirve como controlador para las animaciones, pudiendo relacionarlas

con todos los estados relacionados entre ellos con condicionales.

En cada una de las habilidades el personaje hará la animación correspondiente en base a su acción, pero en este apartado explicaré más detalladamente en qué consiste.

Primero, para hacer posible el cambio de un sprite a otro, es necesaria la creación de una variable “bool” dentro del Animator para cada una de los diferentes estados en los que podrá encontrarse el personaje. De esta manera, dependiendo de qué variable sea verdadera en cada momento, sabrá que sprites debe mostrar.

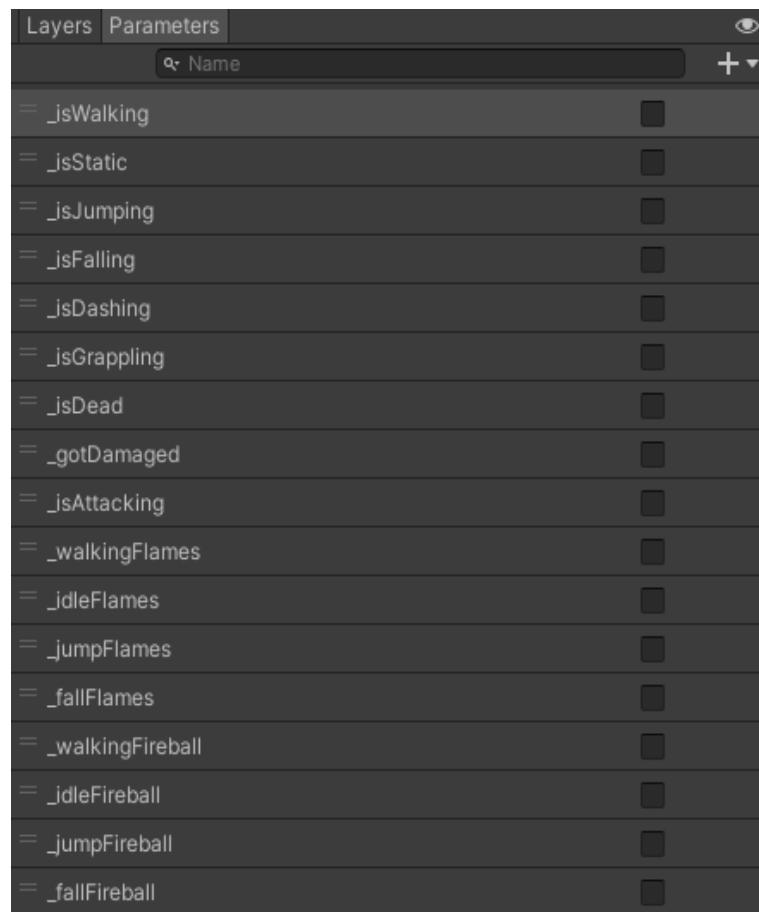


Figura 6.36: Todas las variables ”bool” del personaje

En este caso, como dispone de muchas acciones diferentes y los conocimientos en animación no son muy elevados, todas ellas se relacionan desde el “Any State”, un estado que sirve para acceder a otro desde cualquiera de los demás. Aún así, el caminar y el idle acceden a sus diferentes versiones, siendo estas

la normal y ambos ataques, para evitar errores visuales cuando cambiaban entre ellas ya que son las únicas con más de un sprite, los cuales se van alternando.

Para poder acceder a las diferentes animaciones, necesita tener una de ellas como entrada para poder iniciar, siendo en este caso el Idle, ya que el jugador empezará estático.

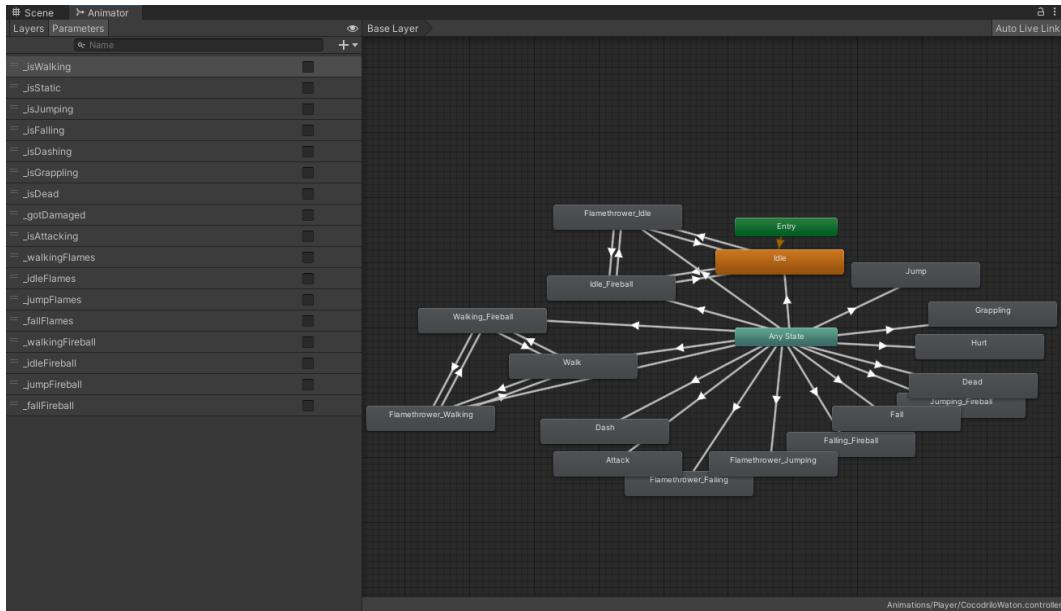


Figura 6.37: Espacio de trabajo del Animator

El personaje debe tener el componente “Animator” con las diferentes animaciones creadas y vinculadas al personaje para funcionar correctamente. En la imagen sobre este texto, se ve el Animator con las diferentes animaciones de las que dispone el personaje, mostrándose como rectángulos con su nombre. La mayoría de animaciones funcionan correctamente situándolas donde toca, pero en el caso de los ataques fue necesaria la creación de funciones enteras simplemente para que la animación funcionará, diferenciándolas entre lanzallamas y bola de fuego.

```

182     private void AttackFalling()
183     {
184         if (_movementScript.IsFalling)
185         {
186             _anim.SetBool("_gotDamaged", false);
187             _anim.SetBool("_isStatic", false);
188             _anim.SetBool("_isWalking", false);
189             _anim.SetBool("_isJumping", false);
190             _anim.SetBool("_isFalling", false);
191             _anim.SetBool("_isDashing", false);
192             _anim.SetBool("_isGrappling", false);
193
194         if (_flamethrower)
195         {
196             _anim.SetBool("_fallFlames", true);
197             _anim.SetBool("_walkingFlames", false);
198             _anim.SetBool("_idleFlames", false);
199             _anim.SetBool("_jumpFlames", false);
200         }
201
202         else
203         {
204             _anim.SetBool("_walkingFireball", false);
205             _anim.SetBool("_fallFireball", true);
206             _anim.SetBool("_idleFireball", false);
207             _anim.SetBool("_jumpFireball", false);
208         }
209     }
210 }
```

Figura 6.38: Código para las animaciones de ataque al caer

## 6.4. Vida

La vida en un personaje es un elemento base en todo videojuego, el hecho de poder morir lo hace más emocionante y tenso. En el caso del cocodrilo, dispone de cinco corazones de vida, perdiendo uno de ellos cada vez que sea golpeado, caiga al vacío o se de un cabezazo quedando encallado dentro de un muro.

Para añadir la vida al personaje, primero se deben crear dos variables “int”, las cuales contienen solo números enteros. Una de ellas con la vida máxima que tendrá y otra para la vida actual.

En cuanto al código, se empezará haciendo que el valor de la vida actual sea el mismo que la vida máxima, ya que cada nivel comienza con la vida al máximo.

```

37 | //la vida actual será igual a la vida máxima al comenzar
38 | CurrentHealth = _maxHealth;

```

Figura 6.39: Código para añadir la vida

Para que el cocodrilo pierda vida cada vez que suceda una de las anteriores situaciones, habrá que restarle uno al valor de la vida actual.

```

99  public void TakeDamage() //el jugador pierde vida y se actualiza en la UI
100 {
101     CurrentHealth -= 1;
102
103     UpdateSprites();
104 }

```

Figura 6.40: Función para perder vida

De esta manera el personaje perderá vida, pero el jugador es incapaz de saber cuánta vida tiene en todo momento. Por ello, hay que crear una barra de vida que se actualice cuando sea necesario.

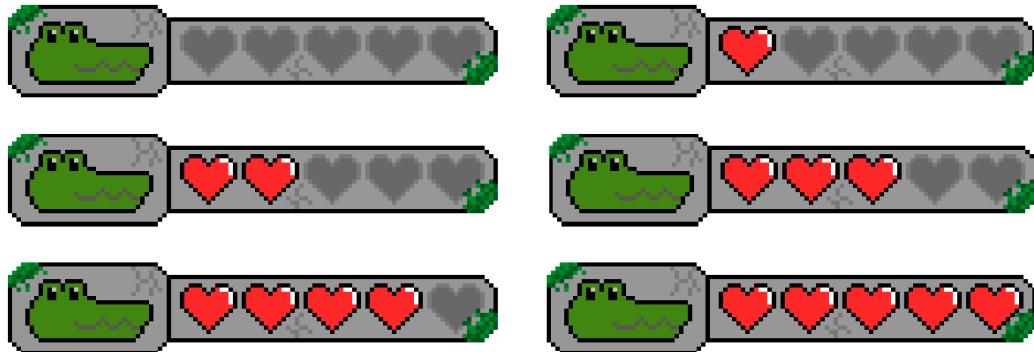


Figura 6.41: Spritesheet de la vida

Para que la imagen se actualice, hay que referenciarla al cocodrilo para que pueda detectar la vida y crear un array de sprites que contenga los diferentes valores de vida para ir cambiando.

Si queremos que el sprite cambie en base a la vida del personaje, hay que cambiar el valor del sprite de la imagen por el del número de vidas actual.

```
130     private void UpdateSprites()
131     {
132         _lifeUI.sprite = _lives[CurrentHealth];
133     }
```

Figura 6.42: Función para actualizar la UI de vida

## 6.5. Interacciones con los enemigos

En el videojuego hay enemigos que intentarán matarle, por lo que necesitará poder interaccionar con ellos, ya sea recibiendo daño o haciéndolo.

### 6.5.1. Recibir daño

Cada enemigo interactúa de una forma diferente con el personaje, pero todos tienen algo en común, le dañan cuando entran en contacto con él.

La colisión estará en un script de los enemigos para tener un acceso más fácil a ella. Una vez la colisión suceda, el personaje cogerá la posición del enemigo para saber desde qué dirección ha sido golpeado. También será necesario un “bool” para saber si el personaje puede recibir daño.

```
126     //detectar posición del enemigo
127     public void EnemyImpacted(Transform EnemyCollisioned)
128     {
129         _enemyPos = EnemyCollisioned;
130     }
```

Figura 6.43: Función para detectar la posición del enemigo

Esta posición servirá para añadir un pequeño retroceso al cocodrilo y que no sea golpeado constantemente por el enemigo. Para añadirlo se necesitan cinco nuevas variables, cuatro “float” y un “bool”. Los “float” serán para el retroceso lateral y superior, la duración del retroceso y el tiempo que el personaje estará inactivo. El “bool” dirá en qué dirección debe ser lanzado el cocodrilo en base a la posición obtenida anteriormente.

Si el cocodrilo puede recibir daño y la vida es superior o igual a un corazón, recibirá daño y hará la animación correspondiente, dará valor a la duración del retroceso y la inactividad y detectará la posición de la colisión.

En caso de que la vida sea inferior a uno, será explicado en el punto 6.6.

```

45     public void PlayerDamaged(Collision2D collision)
46     {
47         if (CanGetDamage && !PlayerIsDead) //si el jugador puede recibir daño perderá vida
48         {
49             CanGetDamage = false;
50             TakeDamage();
51             DamageCD = 0.5f; //se activa el cooldown para volver a recibir daño
52
53             if (CurrentHealth >= 1) //si la vida es diferente a 0 recibirá el knockback
54             {
55                 _anim.SetBool("_gotDamaged", true);
56                 _anim.SetBool("_isStatic", false);
57                 _anim.SetBool("_isWalking", false);
58                 _anim.SetBool("_isJumping", false);
59                 _anim.SetBool("_isFalling", false);
60                 _anim.SetBool("_isDashing", false);
61                 _anim.SetBool("_isGrappling", false);
62
63                 _movementScript.KnockbackCount = 0.5f;
64                 _movementScript.KnockbackLength = 0.2f;
65
66                 if (collision.transform.position.x < _enemyPos.transform.position.x)
67                 {
68                     _movementScript.KnockbackRight = true;
69                 }
70
71                 else
72                 {
73                     _movementScript.KnockbackRight = false;
74                 }
75             }
76
77             else //si la vida es 0 el jugador muere
78             {
79                 PlayerDeath();
80             }
81         }
82     }

```

Figura 6.44: Función de colisión con un enemigo

Una vez golpeado, aplicará los valores del retroceso en un nuevo vector2 según la dirección del impacto durante la duración correspondiente.

```

324     private void Knockback()
325     {
326         if (KnockbackRight) //si el enemigo está en la derecha, el knockback será hacia la izquierda
327         {
328             Rb2D.velocity = new Vector2(-_sideKnockback, _upKnockback);
329         }
330         else
331         {
332             Rb2D.velocity = new Vector2(_sideKnockback, _upKnockback);
333         }
334     }
335     KnockbackLength -= Time.deltaTime;
336 }
337 }
```

Figura 6.45: Función del retroceso

```

129     if (KnockbackLength > 0)
130     {
131         Knockback();
132     }
```

Figura 6.46: Código de la condición para hacer el retroceso

### 6.5.2. Hacer daño

Aparte de los dos ataques de los que dispone el cocodrilo, cada enemigo tendrá una pequeña zona en su parte superior en la que el personaje podrá caer encima y hacerles daño. Cuando esto suceda, el jugador tiene que saberlo de alguna manera, por lo tanto, el cocodrilo tendrá un pequeño retroceso hacia arriba con el que evitará seguir en contacto con el enemigo.

El retroceso es una nueva variable “float”, que servirá para añadirle un valor que sustituirá el eje vertical de la velocidad en un nuevo vector2 para hacer un pequeño salto.

```

84     public void JumpAttack() //knockback al saltar encima de un enemigo
85     {
86         _movementScript.Rb2D.velocity = new Vector2(_movementScript.Rb2D.velocity.x, _knockback);
87     }
```

Figura 6.47: Función del retroceso al saltar encima de un enemigo

La función será pública para poder acceder a ella desde los enemigos.

## 6.6. Muerte

En todo videojuego con vida, existe la muerte, la cual te lleva a la derrota.

Cuando la vida del cocodrilo llegue a cero, entrará en la función de muerte, como se puede observar en la figura 6.44.

La función consta de una nueva variable “bool” para detectar si el personaje ha muerto, la cual cambia a verdadera, cancela todas las animaciones y activa la animación de muerte y le enseña al jugador el panel de muerte.

```
84     private void PlayerDeath()
85     {
86         PlayerIsDead = true;
87         _anim.SetBool("_isDead", true);
88         _anim.SetBool("_isWalking", false);
89         _anim.SetBool("_isFalling", false);
90         _anim.SetBool("_isStatic", false);
91         _anim.SetBool("_isJumping", false);
92         _anim.SetBool("_isDashing", false);
93         _anim.SetBool("_isGrappling", false);
94         _attackScript.AttackCancel();
95
96         _lose.SetActive(PlayerIsDead);
97     }
```

Figura 6.48: Función de muerte

La variable para detectar si el personaje ha muerto también tiene uso para cancelar todas las posibles acciones del jugador con el personaje.

## 6.7. Muslos de carne

El cocodrilo tendrá que recoger objetos repartidos en cada nivel, siendo estos unos muslos de carne, haciendo referencia a que la principal fuente de alimento de los cocodrilos de pantano son animales.

Cada nivel tendrá diez de estos muslos repartidos por todo su entorno que el jugador deberá recoger si quiere alcanzar la máxima puntuación disponible en cada uno de ellos. Para añadir algo de dificultad a la búsqueda, hay unos más escondidos que otros.

Para la realización del contador de muslos, hace falta una variable “int” que cuente la cantidad obtenida y darle el valor de cero al iniciar el nivel para evitar que empiece con el valor del nivel anterior.

```

13     private void Start()
14     {
15         CurrentCoins = 0; //el valor inicial es 0
16     }

```

Figura 6.49: Código para darle valor al contador de muslos

El jugador debe ver en todo momento cual es la cantidad de muslos que ha conseguido, eso se logra a partir de una imagen y cambiando su sprite cada vez que el valor se actualice. Para ello, habrá que cambiar el valor de dicho sprite por el número de muslos obtenidos en ese momento.



Figura 6.50: Spritesheet contador de muslos

```

18     public void ShowCoins()
19     {
20         _ham.sprite = _hamCounter[CurrentCoins];
21     }

```

Figura 6.51: Código para actualizar el contador de muslos

El valor debe aumentar cada vez que el cocodrilo entre en contacto con un muslo, por tanto habrá que sumarle uno al valor cuando ambas cosas colisionen, actualizarlo en la imagen y destruir el objeto para que no pueda sumar permanentemente.

```

7     private void OnTriggerEnter2D(Collider2D other)
8     {
9         //si entra en contacto con el personaje sumará 1 y se destruirá
10        if (other.GetComponent<Movimiento>() != null)
11        {
12            TextManager.TManager.Counter.CurrentCoins += 1;
13            TextManager.TManager.Counter.ShowCoins();
14            Destroy(this.gameObject);
15        }
16    }

```

Figura 6.52: Código para la colisión con los muslos de carne

## 6.8. Respawn

El respawn es la reaparición del personaje en la escena. Esto puede ocurrir de dos maneras distintas durante los diferentes niveles, cayendo en zonas sin suelo o quedando atrapado dentro de pequeños huecos por los que se puede pasar con el dash, perdiendo una vida en ambas ocasiones.

Por una parte tenemos las caídas, las cuales en los niveles de pantano están representadas por una piscina de barro, así el jugador puede identificarlas más fácilmente. En cambio, en los niveles de cueva, el jugador verá una caída al vacío ya que en ambas esquinas de la caída el suelo hace una forma curvada para facilitar su visibilidad.

Estas son objetos vacíos con solo un colisionador para poder hacer contacto con el personaje, de esta manera, cuando el cocodrilo detecte la colisión recibirá daño y reaparecerá en caso de tener más de un corazón de vida. En caso contrario, morirá.

```

33     private void OnTriggerEnter2D(Collider2D other)
34     {
35         if (other.gameObject.name == "Caida")
36         {
37             //cuando el jugador caiga volverá al último checkpoint, perdiendo 1 de vida
38             if (_playerHealth.CurrentHealth > 1)
39             {
40                 _playerHealth.TakeDamage();
41                 this.gameObject.transform.position = _checkpoint.transform.position;
42                 _mov.Rb2D.velocity = new Vector2(0, 0);
43             }
44
45             else //si el jugador cae quedándose a 0 de vida, morirá
46             {
47                 PlayerDeath();
48             }
49         }

```

Figura 6.53: Código para la detección de la caída

Por otra parte, los huecos están representados con una doble pared la cual

llega casi al suelo dejando un pequeño espacio, de esta manera el jugador es consciente de que por ahí puede pasar de alguna manera.

En este caso, el jugador dispone de dos puntos de guardado invisibles en ambos laterales del hueco para reaparecer al lado en caso de quedar atrapado.

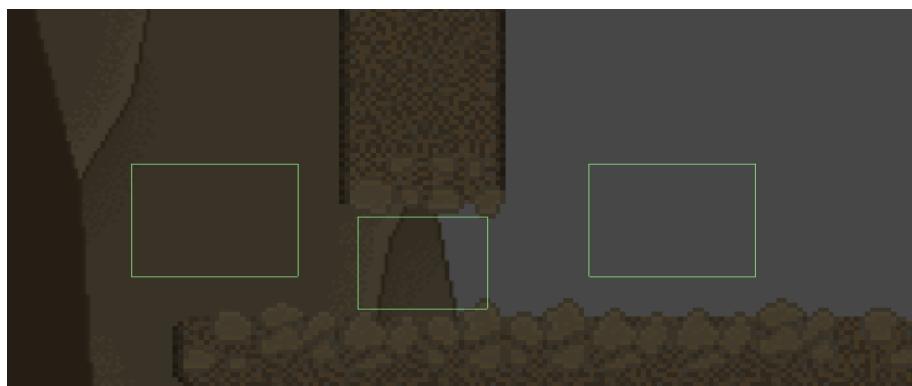


Figura 6.54: Zona de dash con los triggers

```
55     if (other.gameObject.name == "DashRespawn")
56     {
57         _dashCheckpoint = other.gameObject;
58     }
```

Figura 6.55: Código para cambiar el respawn

A diferencia de la caída, la reaparición solo sucederá si el personaje no está usando el impulso cuando quede dentro del rango de acción, ya que sin esta condición reaparecería con el simple hecho de pasar.

```
60     if (other.gameObject.name == "StuckRespawn")
61     {
62         if (!_mov.IsDashing && !PlayerManager.PManager.PlayerAttackScript.IsAttacking)
63         {
64             if (_playerHealth.CurrentHealth > 1) //el jugador perderá 1 de vida
65             {
66                 _playerHealth.TakeDamage();
67                 this.gameObject.transform.position = _dashCheckpoint.transform.position;
68                 _mov.Rb2D.velocity = new Vector2(0, 0);
69             }
70
71             else //si el jugador cae quedándose a 0 de vida, morirá
72             {
73                 PlayerDeath();
74             }
75         }
76     }
```

Figura 6.56: Código para el respawn en la zona de dash

## 6.9. Seguimiento de la cámara

La cámara debe seguir al personaje en todo momento para que el jugador pueda visualizar constantemente qué está haciendo y qué debe hacer a continuación.

Lo primero que debe obtener la cámara es la posición del personaje para poder acceder a ella y un vector3 para ajustar la posición de la cámara. Teniendo ambas cosas, se crea un nuevo vector3 sumándolas para obtener la posición deseada.

```
23 | //detectar la posición del jugador y modificarla con el offset  
24 | Vector3 playerPosition = _player.position + _offset;
```

Figura 6.57: Código para ajustar la posición de la cámara

Con esto el seguimiento ya era funcional, pero también quería añadir límites a la cámara para que no se saliera del mapa y un pequeño retraso en el seguimiento para darle más fluidez.

Para añadir los límites, lo primero era poner las posiciones de todos los ejes manualmente en un nuevo vector3. A través del vector anterior y el nuevo creado, se establece otro con un cálculo automático que compara la posición actual de los tres ejes con los límites, impidiendo que la cámara los traspase.

```
25 | Vector3 boundPosition = new Vector3(  
26 |     Mathf.Clamp(playerPosition.x, _minValues.x, _maxValues.x),  
27 |     Mathf.Clamp(playerPosition.y, _minValues.y, _maxValues.y),  
28 |     Mathf.Clamp(playerPosition.z, _minValues.z, _maxValues.z));
```

Figura 6.58: Código para añadir límites a la cámara

Finalmente, para obtener el pequeño retraso en el seguimiento de la cámara, se crea una variable “float” con el valor y se crea un último vector3 que coge la posición actual de la cámara y la siguiente posición que tendrá. Al tener ambas posiciones, utiliza el valor del retraso multiplicándolo por el tiempo para añadir una transición más fluida entre posición y posición.

Se le añade la posición modificada a la cámara y el seguimiento funciona a la perfección.

```
21  private void Follow()
22  {
23      //detectar la posición del jugador y modificarla con el offset
24      Vector3 playerPosition = _player.position + _offset;
25
26      Vector3 boundPosition = new Vector3(
27          Mathf.Clamp(playerPosition.x, _minValues.x, _maxValues.x),
28          Mathf.Clamp(playerPosition.y, _minValues.y, _maxValues.y),
29          Mathf.Clamp(playerPosition.z, _minValues.z, _maxValues.z));
30
31      //añadir el delay
32      Vector3 smoothPosition = Vector3.Lerp(transform.position, boundPosition, _smoothFactor*Time.fixedDeltaTime);
33      transform.position = smoothPosition; //cambiar la posición de la cámara por la modificada
34  }
```

Figura 6.59: Función para el seguimiento de la cámara

# **Capítulo 7**

## **Enemigos**

Aparte del cocodrilo, hay otros cuatro seres vivos que están distribuidos en los diferentes niveles, siendo estos los enemigos. Cada nivel tiene un enemigo diferente en base a su estética, en los pantanos hay animales que viven en pantanos y lo mismo para las cuevas. Estos son capaces de dañar al personaje de diferentes maneras.

### **7.1. Tortuga**

El primer enemigo que encontrará el jugador es una tortuga en honor a los míticos juegos de Super Mario. Este enemigo es apacible, su único objetivo es ir de un punto a otro sin que nadie le moleste. Si el cocodrilo se atreve a meterse con ella, esta no dudará en hacerle daño o esconderse si se le aproxima fuego con el personaje cerca.

#### **7.1.1. Movimiento de patrulla**

El movimiento base de la tortuga consiste en desplazarse lateralmente de un punto a otro constantemente, hasta que otra acción lo interrumpa.

La programación se inicia creando una variable “float” para la velocidad, un “bool” para saber si está patrullando, una lista de puntos que la tortuga usará para desplazarse, el valor de cada punto y un valor para cambiar el anterior.

Para que solo iniciar el nivel haga la patrulla, empezará con el “bool” en verdadero y la velocidad ya aplicada.

```
17     private void Start()
18     {
19         _speed = 2.5f;
20         Patrolling = true;
21     }
```

Figura 7.1: Código al iniciar la tortuga

El movimiento empezará creando un nuevo punto que será hacia el cual irá la tortuga, al cual le añadimos la lista de puntos creada y aplicaremos la animación de caminar.

```
40     private void MoveToNextPoint()
41     {
42         Transform goalPoint = _movementPoints[_nextID]; //obtener el transform del siguiente punto
43
44         _anim.SetBool("_isWalking", true);
```

Figura 7.2: Inicio de la función de patrulla

Dependiendo de donde esté situado dicho punto, la tortuga mirará hacia el lado correspondiente.

```
46     if (goalPoint.transform.position.x > transform.position.x) //flip para que mire a la dirección correcta
47     {
48         transform.localScale = new Vector3(-1, 1, 1);
49     }
50
51     else
52     {
53         transform.localScale = new Vector3(1, 1, 1);
54     }
```

Figura 7.3: Código para que la tortuga se gire

Para que la tortuga pueda desplazarse, su posición deberá ser actualizada constantemente con el uso de la velocidad y el punto objetivo. Para ello, se cambiará su posición por la de un nuevo vector2 que indicará que vaya de su posición actual hacia el punto objetivo con la velocidad asignada.

```
56 |     transform.position = Vector2.MoveTowards(transform.position, goalPoint.position, _speed * Time.deltaTime); //movimiento
```

Figura 7.4: Código para que la tortuga se mueva

De esta manera irá hacia un punto, pero no tiene forma de ir al siguiente. Por lo tanto, hay que actualizar el valor del punto al que deberá ir. Para lograrlo, si la posición de la tortuga llega al punto objetivo el valor de este cambiará en base a qué punto de la lista ha llegado.

```
58 |     //calcular la distancia entre la tortuga y el siguiente punto
59 |     if (Vector2.Distance(transform.position, goalPoint.position) < 0.5f)
60 |     {
61 |         //si está en el segundo punto el valor cambiará a negativo
62 |         if (_nextID == _movementPoints.Count - 1)
63 |         {
64 |             _idChangeValue = -1;
65 |         }
66 |
67 |         if (_nextID == 0) //en el primer punto será positivo
68 |         {
69 |             _idChangeValue = 1;
70 |         }
71 |
72 |         _nextID += _idChangeValue; //aplicar el cambio en el valor
73 |     }
74 | }
```

Figura 7.5: Código para cambiar de punto objetivo

Si la tortuga está patrullando, se le dará el valor a la velocidad y hará el movimiento de patrulla. En cambio, si la patrulla ha sido interrumpida, la velocidad pasará a ser cero y se cancelará la animación de caminar.

```
25 |     private void Update()
26 |     {
27 |         if (Patrolling)
28 |         {
29 |             _speed = 2.5f;
30 |             MoveToNextPoint();
31 |         }
32 |
33 |         else //si no está patrullando se parará
34 |         {
35 |             _speed = 0;
36 |             _anim.SetBool("_isWalking", false);
37 |         }
38 |     }
```

Figura 7.6: Código para que la tortuga sepa qué hacer

### 7.1.2. Daño al jugador

La tortuga solo dañará al cocodrilo por contacto físico en caso de que este se meta en su camino.

Para que la tortuga haga daño al personaje, deberá detectar la colisión en caso de que no esté muerta y, acto seguido, entrará en dos funciones del jugador, enviando su posición actual con una y haciendo daño al cocodrilo con otra.

```
16  private void OnCollisionEnter2D(Collision2D collision)
17  {
18      //si colisiona con el personaje sin tocar la hitbox de salto y no está muerta hará daño al jugador
19      if (collision.gameObject.name == "CocodriloWatson")
20      {
21          if (_jumpHitbox.DontHurtPlayer <= 0 && !_turtle.IsDead)
22          {
23              //detecta la posición de la tortuga para el knockback
24              PlayerManager.PManager.PlayerHealthScript.EnemyImpacted(this.transform);
25
26              PlayerManager.PManager.PlayerHealthScript.PlayerDamaged(collision);
27          }
28      }
29  }
```

Figura 7.7: Código para hacer daño al jugador

### 7.1.3. Esconderse

Esta tortuga sabe que el fuego es peligroso, si ve que se aproxima se esconderá para evitar ser quemada.

El cocodrilo dispone de dos habilidades que pueden quemar a la tortuga, debido a esto, hay dos maneras distintas de hacer que esta se esconda.

La primera y más sencilla es que, si la tortuga está mirando hacia el cocodrilo y este está atacando, pasará a verdadera una nueva variable “bool” para saber si debe esconderse o no. Esta variable se volverá falsa cuando el cocodrilo salga de su rango de visión.

```

102     private void OnTriggerEnter2D(Collider2D other)
103     {
104         if (other.GetComponent<Movimiento>() != null
105             && PlayerManager.PManager.PlayerAttackScript.IsAttacking)
106         {
107             _isHiding = true;
108         }
109     }
110
111     private void OnTriggerExit2D(Collider2D other)
112     {
113         //cuando el jugador salga del trigger volverá a salir del caparazón
114         if (other.GetComponent<Movimiento>() != null)
115         {
116             _isHiding = false;
117             _crocodileTrigger = false;
118         }
119     }

```

Figura 7.8: Código para detectar al cocodrilo atacando

En este fragmento de código hay otro “bool”, este sirve para saber si el cocodrilo en sí está dentro del rango. Este se activa cuando el personaje entra en el rango de visión, ya sea atacando o no.

```

89     private void OnTriggerEnter2D(Collider2D other)
90     {
91         if (other.GetComponent<Movimiento>() != null)
92         {
93             _crocodileTrigger = true;
94         }

```

Figura 7.9: Código para saber si el cocodrilo está dentro del rango

La otra forma de esconderse es que el “bool” del cocodrilo sea verdadero y la bola de fuego también esté presente.

```

96     //si la bola de fuego y el jugador están en el trigger se esconde
97     if (other.GetComponent<Fireball>() != null && _crocodileTrigger)
98     {
99         _isHiding = true;
100    }
101 }

```

Figura 7.10: Código para saber si esconderse ante la bola de fuego

Cuando la tortuga esté escondida, dejará de patrullar y hará una animación en la que se esconde en su caparazón. En el momento en que vuelva a mostrarse, reanudará la patrulla.

```

119     private void TurtleHide() //cuando la tortuga se esconde deja de patrullar
120     {
121         _patrol.Patrolling = false;
122         _anim.SetBool("_danger", true);
123     }
124
125
126     private void TurtleShow() //la tortuga vuelve a patrullar
127     {
128         _patrol.Patrolling = true;
129         _anim.SetBool("_danger", false);
130     }
131 }
```

Figura 7.11: Funciones de esconderse y aparecer

```

36     if (_isHiding)
37     {
38         TurtleHide();
39     }
40
41     else if (!_isHiding && EnemyCanGetDamage)
42     {
43         TurtleShow();
44     }
```

Figura 7.12: Código para saber si esconderse o patrullar

#### 7.1.4. Recibir daño

La principal forma de dañar a los enemigos es saltar encima suyo, aunque también pueden recibir daño siendo golpeados por el lanzallamas o la bola de fuego. Cuando dicha acción suceda, el movimiento de patrulla será interrumpido.

Para recibir daño cuando el cocodrilo salta encima, hay que crear un nuevo objeto vacío que servirá para separar la colisión que daña al personaje y la que daña a la tortuga. Cuando detecte que el cocodrilo impacta, accederá a la función que daña al enemigo y al retroceso que hace el personaje al dañar a un enemigo.

```

24     private void OnCollisionEnter2D(Collision2D collision)
25     {
26         //si el personaje toca el collider hará un pequeño bote y dañará la tortuga
27         if (collision.gameObject.name == "CocodriloWatson")
28         {
29             DontHurtPlayer = 0.3f;
30             _enemy.JumpDamage();
31             PlayerManager.PManager.PlayerAttackScript.JumpAttack();
32         }
33     }
```

Figura 7.13: Código para recibir daño

La condición para entrar en la función en la que la tortuga recibe daño, es que una nueva variable “bool” que detecta si puede recibir dicho daño sea verdadera. Al entrar en esta función, el movimiento de patrulla se cancelará, restará vida a la tortuga, volverá falsa la condición para recibir daño y, en caso de tener uno o más de vida, añadirá un nuevo “float” para tener un margen de tiempo en el que no pueda recibir daño y hará la animación correspondiente.

```

48     public void JumpDamage()
49     {
50         if (EnemyCanGetDamage) //si la tortuga puede recibir daño entrará en la función
51         {
52             _patrol.Patrolling = false; //dejará de patrullar al recibir daño
53
54             _enemyHP -= 1; //resta 1hp
55
56             EnemyCanGetDamage = false; //no puede volver a recibir daño
57
58             if (_enemyHP <= 0) //si la vida llega a 0 morirá
59             {
60                 _anim.SetBool("_normalDeath", true);
61                 IsDead = true;
62             }
63
64             else //si aún le queda vida hará la animación de recibir daño
65             {
66                 _damagedCD = 0.5f;
67                 _anim.SetBool("_getDamaged", true);
68             }
69         }
70     }

```

Figura 7.14: Función del daño del cocodrilo al saltar encima

Podrá volver a recibir daño cuando la variable sea menor o igual a cero, momento en que el movimiento de patrulla volverá a iniciar y se cancelará la animación de daño.

```

72     private void DamagedAgain()
73     {
74         //cuando el cooldown llegue a 0 volverá a patrullar y podrá recibir daño de nuevo
75         if (_damagedCD <= 0)
76         {
77             _anim.SetBool("_getDamaged", false);
78
79             _patrol.Patrolling = true;
80
81             EnemyCanGetDamage = true;
82         }
83
84         else
85         {
86             _damagedCD -= Time.deltaTime;
87         }
88     }

```

Figura 7.15: Función para volver a recibir daño

En cuanto al lanzallamas, la tortuga detectará si está colisionando con este y entrará en la función para recibir su daño.

```
39     □    private void OnTriggerEnter2D(Collider2D other)
40     {
41         □    if (other.GetComponent<Flamethrower>() != null)
42         {
43             □    _turtle.FlamethrowerDamaged();
44         }
45     }
```

Figura 7.16: Código para detectar la colisión con el lanzallamas

Si la tortuga no está escondida y puede recibir daño, se le restará vida en base al daño del lanzallamas y cancelará la patrulla. De la misma manera que con el salto, si su vida es superior o igual a uno, añadirá un pequeño enfriamiento y hará la animación correspondiente.

```
148     □    public void FlamethrowerDamaged()
149     {
150         □    if (!_isHiding && EnemyCanGetDamage)
151         {
152             _enemyHP -= PlayerManager.PManager.FlamethrowerScript.FlamethrowerDamage;
153
154             _patrol.Patrolling = false;
155
156             EnemyCanGetDamage = false;
157
158             □    if (_enemyHP > 0)
159             {
160                 _damagedCD = 0.2f;
161                 _anim.SetBool("_getDamaged", true);
162             }
163
164             □    else
165             {
166                 _anim.SetBool("_burnDeath", true);
167                 IsDead = true;
168             }
169         }
170     }
```

Figura 7.17: Función del daño del lanzallamas

Por último, la bola de fuego tiene la misma cantidad de daño que la tortuga, por lo tanto morirá directamente con la colisión.

```
31     □    private void OnTriggerEnter2D(Collider2D other)
32     {
33         □    //si colisiona con la bola de fuego morirá
34         □    if (other.GetComponent<Fireball>() != null)
35         {
36             □    Destroy(other.gameObject);
37
38             □    _turtle.FireDeath();
39         }
40     }
```

Figura 7.18: Código para detectar la colisión de la bola de fuego

### 7.1.5. Muerte

Cuando la vida de la tortuga llegue a cero, hará la animación de muerte y poco después desaparecerá. Tiene dos tipos de muertes disponibles, una para el salto, con la cual se queda boca abajo, y la otra para cuando es con una habilidad de fuego, quedando también boca abajo pero quemada.

En el caso de la muerte por salto, cuando la vida llegue a cero activará la animación de muerte y pasará a verdadero un “bool” para saber si ha muerto.

El código se puede observar en las líneas 58 a 62 de la figura 7.14.

Con el lanzallamas, activará la muerte por fuego y de la misma manera que la anterior, cambiará el “bool” de muerte a verdadero.

El código se puede observar en las líneas 164 a 168 de la figura 7.17.

Por último, como la bola de fuego la mata instantáneamente, cancelará el movimiento de patrulla y después activará la animación de muerte por fuego y el “bool” será verdadero.

```
133     public void FireDeath() //si no está escondida la bola de fuego le matará quemándola
134     {
135         if (!isHiding)
136         {
137             _enemyHP -= PlayerManager.PManager.FireballScript.FireballDamage;
138
139             if (_enemyHP <= 0)
140             {
141                 _patrol.Patrolling = false;
142                 _anim.SetBool("_burnDeath", true);
143             }
144         }
145     }
```

Figura 7.19: Función de muerte por la bola de fuego

Finalmente, la tortuga debe desaparecer una vez haya muerto, por lo que cuando el “bool” de muerte sea verdadero, destruirá el objeto una vez pase poco menos de un segundo.

```
14     private void DestroyObject()
15     {
16         if (_turtleScript.IsDead)
17         {
18             Destroy(this.gameObject, 0.7f);
19         }
20     }
```

Figura 7.20: Función para destruir el objeto

## 7.2. Murciélagos

El segundo enemigo encontrado es un murciélagos, un clásico de las cuevas. Vive tranquilamente hasta que el jugador se le acerca, momento en el que se enfada y no duda en ir a por él, aumentando su velocidad. Después del ataque volverá a volar tranquilamente.

### 7.2.1. Movimiento de patrulla

Al tener la misma funcionalidad que la tortuga, usará el mismo script que el 7.1.1.

### 7.2.2. Atacar al jugador

El murciélagos solamente hace daño por contacto físico cuando ataca al cocodrilo. Esto sucederá cuando el personaje se sitúe debajo del murciélagos, momento en que este dejará de patrullar e irá directo a la posición del jugador al mismo tiempo que aumenta su velocidad.

Para saber si el cocodrilo está debajo del murciélagos, un nuevo “bool” será verdadero en caso de que esto suceda. La patrulla será cancelada en el momento en que pase a verdadero y obtendrá la posición del personaje para saber dónde deberá ir.

```
57     private void OnTriggerEnter2D(Collider2D other)
58     {
59         if (!_crocTriggered)
60         {
61             if (other.GetComponent<Movimiento>() != null)
62             {
63                 _patrol.Patrolling = false;
64                 _crocTriggered = true;
65                 PlayerTriggered(other.transform.position);
66             }
67         }
68     }
```

Figura 7.21: Código para detectar al cocodrilo

```
96     //detectar posición del jugador
97     public void PlayerTriggered(Vector3 PlayerCollisioned)
98     {
99         _crocPosition = PlayerCollisioned;
100    }
```

Figura 7.22: Función para detectar la posición del jugador

En ese momento, iniciará la animación de ataque y aumentará considerablemente su velocidad. Dependiendo de la posición en la que esté el cocodrilo,

el murciélagos se girará para mirar hacia él.

Se moverá hacia el cocodrilo con un nuevo vector2 que lo desplaza de su posición actual hasta el cocodrilo con la velocidad asignada.

Cuando el murciélagos llegue a la posición del cocodrilo, le golpee o no, volverá al movimiento de patrulla.

```
70  private void AttackPlayer()
71  {
72      _anim.SetBool("_attack", true);
73
74      _patrol.Speed = 5f;
75
76      if (transform.position.x < _crocPosition.x)
77      {
78          transform.localScale = new Vector3(-0.65f, 0.65f, 0.65f);
79      }
80
81      else
82      {
83          transform.localScale = new Vector3(0.65f, 0.65f, 0.65f);
84      }
85
86      transform.position = Vector2.MoveTowards(transform.position, _crocPosition, _patrol.Speed * Time.deltaTime);
87
88      if (Mathf.Approximately(transform.position.x, _crocPosition.x))
89      {
90          _crocTriggered = false;
91          _patrol.Patrolling = true;
92          _anim.SetBool("_attack", false);
93      }
94 }
```

Figura 7.23: Función de ataque

Para que el murciélagos haga daño al personaje, deberá detectar la colisión en caso de que no esté muerto y, acto seguido, entrará en dos funciones del jugador, enviando su posición actual con una y haciendo daño al cocodrilo con otra.

```
16  private void OnCollisionEnter2D(Collision2D collision)
17  {
18      //si colisiona con el personaje sin tocar la hitbox de salto y no está muerta hará daño al jugador
19      if (collision.gameObject.name == "CocodriloWaton")
20      {
21          if (_jumpHitbox.DontHurtPlayer <= 0 && !_bat.IsDead)
22          {
23              //detecta la posición del murciélagos para el knockback
24              PlayerManager.PManager.PlayerHealthScript.EnemyImpacted(this.transform);
25
26              PlayerManager.PManager.PlayerHealthScript.PlayerDamaged(collision);
27
28          }
29      }
```

Figura 7.24: Código para dañar al personaje por contacto

### 7.2.3. Recibir daño

Como el murciélagos recibe daño de la misma forma que la tortuga, usará el mismo script que el 7.1.4, quitando la parte de esconderse.

### 7.2.4. Muerte

La muerte del murciélagos es igual a la tortuga añadiendo una parte, por lo que usará el script del apartado 7.1.5 y se le añadirá un poco más de código. Tiene dos animaciones al morir, en caso de morir en el aire, empezará a caer y hará una de ellas, cuando llegue al suelo hará otra.

Para saber cuál de ambas animaciones tiene que hacer, el murciélagos dispone de un detector de suelo, hecho de la misma manera que el código de la figura 6.6.

Cuando muera, obtendrá gravedad para caer, se activará que pueda caer en el eje vertical y, una vez esté en el suelo, de desactivará dicho eje y cambiará la animación de caer por la del suelo.

```
179     private void CheckGround() //detectar el suelo para cambiar de animación
180     {
181         _batRB.gravityScale = 2.5f;
182         _batRB.constraints = RigidbodyConstraints2D.FreezePositionX | RigidbodyConstraints2D.FreezeRotation;
183
184         //añadir rango al objeto para detectar el suelo
185         _isGrounded = Physics2D.OverlapCircle(_groundCheck.position, 0.2f, _floor);
186
187         if (_isGrounded)
188         {
189             _batRB.constraints = RigidbodyConstraints2D.FreezePositionY;
190
191             _anim.SetBool("_groundDeath", true);
192             _anim.SetBool("_groundBurn", true);
193
194             _anim.SetBool("_normalDeath", false);
195             _anim.SetBool("_burnDeath", false);
196         }
197     }
```

Figura 7.25: Función para caer y saber qué animación hacer

## 7.3. Araña

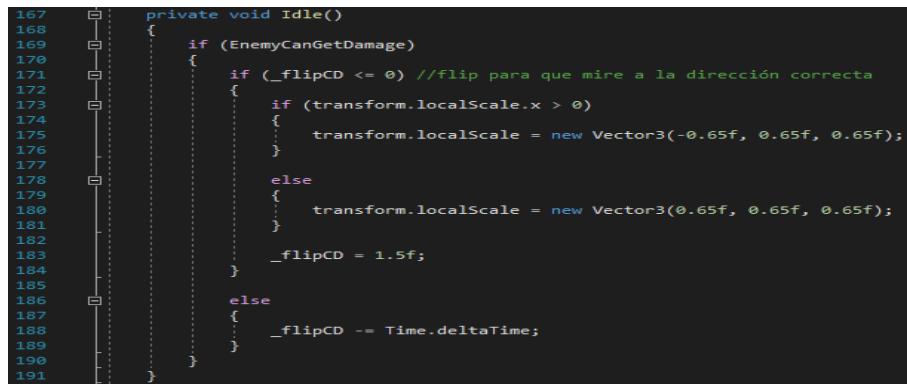
El tercer enemigo es una araña, otro ser vivo que es fácil pensar en él cuando se habla de cuevas. A diferencia de los dos anteriores, se mantendrá estática observando su alrededor hasta detectar a su presa. Si la araña se da cuenta de la presencia del cocodrilo, pasará a la acción en un parpadeo, corriendo hacia el personaje a una velocidad muy elevada.

### 7.3.1. Idle

La araña es un animal que espera a su presa en su tela, por ello la animación que hará constantemente mientras no aparezca el jugador es mirar de un lado a otro, esperando para atacar.

Como irá cambiando la dirección en la que mira cada cierto tiempo, se necesita una variable “float” para calcular ese tiempo. Cuando la araña pueda recibir daño y el tiempo para cambiar de dirección llegue a cero, se girará y volverá a ponerle un valor al tiempo de giro.

Mientras ese valor sea superior a cero, se le restará el tiempo.



```
167    private void Idle()
168    {
169        if (EnemyCanGetDamage)
170        {
171            if (_flipCD <= 0) //flip para que mire a la dirección correcta
172            {
173                if (transform.localScale.x > 0)
174                {
175                    transform.localScale = new Vector3(-0.65f, 0.65f, 0.65f);
176                }
177                else
178                {
179                    transform.localScale = new Vector3(0.65f, 0.65f, 0.65f);
180                }
181                _flipCD = 1.5f;
182            }
183            else
184            {
185                _flipCD -= Time.deltaTime;
186            }
187        }
188    }
189}
190}
191}
```

Figura 7.26: Función del idle

### 7.3.2. Perseguir al jugador

La araña solamente hace daño por contacto físico cuando ataca al cocodrilo. Esto sucederá cuando el personaje se sitúe enfrente de ella, momento en que irá directo a la posición del jugador a una velocidad alarmante.

Para saber si el cocodrilo está enfrente de la araña, un nuevo “bool” será verdadero en caso de que esto suceda y obtendrá la posición del personaje constantemente para poder perseguirle.

Para obtener la posición usará el mismo código de la figura 7.22.

```

62     □    private void OnTriggerEnter2D(Collider2D other)
63     □    {
64     □    □    if (other.GetComponent<Movimiento>() != null)
65     □    □    {
66     □    □    □    _crocTriggered = true;
67     □    □    □    PlayerTriggered(other.transform.position);
68     □    □}
69     □}

```

Figura 7.27: Código para detectar al jugador

En ese momento, iniciará la animación de ataque y su velocidad será muy elevada. Se moverá hacia el cocodrilo con un nuevo vector2 que lo desplaza de su posición actual hasta el cocodrilo con la velocidad asignada.

```

80     □    private void AttackPlayer()
81     □    {
82     □    □    _anim.SetBool("_attack", true);
83
84     □    □    Vector2 newPosition = new Vector2 (_crocPosition.x, transform.position.y);
85
86     □    □    transform.position = Vector2.MoveTowards(transform.position, newPosition, _speed * Time.deltaTime);
87     □}

```

Figura 7.28: Función para atacar al jugador

Si el personaje sale del rango de visión de la araña o esta se topa con un obstáculo, volverá a quedarse quieta.

El detector de suelo se hará de la misma manera que la figura 6.6.

```

71     □    private void OnTriggerExit2D(Collider2D other)
72     □    {
73     □    □    if (other.GetComponent<Movimiento>() != null)
74     □    □    {
75     □    □    □    _crocTriggered = false;
76     □    □    □    _anim.SetBool("_attack", false);
77     □    □}
78     □}

```

Figura 7.29: Código para dejar de detectar al jugador

```

47
48
49
50
51
52
53
54
55
56
57
58
    if (_EnemyCanGetDamage)
    {
        if (_crocodileTriggered && _isGrounded)
        {
            AttackPlayer();
        }
        else
        {
            Idle();
        }
    }

```

Figura 7.30: Código para saber si debe hacer el idle o atacar

Para que la araña haga daño al personaje, deberá detectar la colisión en caso de que no esté muerta y, acto seguido, entrará en dos funciones del jugador, enviando su posición actual con una y haciendo daño al cocodrilo con otra. El código se puede observar en la figura 7.24.

### 7.3.3. Recibir daño

Como la araña recibe daño de la misma forma que la tortuga, usará el mismo script que el 7.1.4, quitando la parte de esconderse.

### 7.3.4. Muerte

La muerte de la araña es igual a la tortuga, por lo que usará el script del apartado 7.1.5.

## 7.4. Castor

El último enemigo se trata de un castor, un ser muy territorial. Estará tranquilamente en su pequeña zona del nivel hasta que el cocodrilo la invada, momento en que el castor le tendrá en el punto de mira y le lanzará pequeños trozos de madera cada cierto tiempo, finalizando cuando el personaje huya de su territorio o con la muerte.

### 7.4.1. Idle

Como el idle del castor es igual al de la araña, usará el script del apartado 7.3.1.

### 7.4.2. Ataque al jugador

Los castores son conocidos por ser roedores de madera, por esto su principal forma de atacar al jugador es lanzarle un pequeño trozo de madera para in-

tentar que se aleje de su territorio. También es capaz de dañar al cocodrilo por contacto físico si este logra acercarse a él.

Lo primero que hará el castor en cuanto el cocodrilo entre en su territorio, es mirarle fijamente con cara de enfado y listo para lanzarle un trozo de madera. Para ello necesita obtener su posición para compararla con la suya y saber hacia dónde mirar.

Para detectar al jugador y su posición usará el código de las figuras 7.22 y 7.27.

```
173  private void LookToPlayer()
174  {
175      if (transform.position.x < _crocPosition.x)
176      {
177          transform.localScale = new Vector3(-0.65f, 0.65f, 0.65f);
178      }
179      else
180      {
181          transform.localScale = new Vector3(0.65f, 0.65f, 0.65f);
182      }
183  }
```

Figura 7.31: Función para mirar al cocodrilo

Acto seguido iniciará la preparación de su ataque, obteniendo tres puntos para calcular el lanzamiento del proyectil, siendo estos el punto de inicio, el punto objetivo y un último situado entre ambos para realizar el lanzamiento de forma curva.

```
15  public void SetUpAttack(Vector3 crocPosition)
16  {
17      _startPoint = _startPointTransform.position;
18
19      _endPoint = crocPosition;
20
21      _controlPoint = _startPoint + (_endPoint - _startPoint) / 2;
22
23      _controlPoint.y += 10;
24  }
```

Figura 7.32: Función para preparar el ataque

Una vez obtenidos, creará el proyectil, dándole la posición de esos tres puntos.

```
26  public void CreateBullet()
27  {
28      Instantiate(_projectile).GetComponent<BeaverProjectile>().SetUp(_startPoint, _controlPoint, _endPoint);
29  }
```

Figura 7.33: Función para crear el proyectil

Al ser creado, iniciará la función que le permitirá moverse, siendo esta un cálculo que hace una curva de Bézier usando tres puntos en vez de cuatro.

Para añadirle un poco más de realismo al lanzamiento, se le aplica una pequeña rotación constante que mejora visualmente el movimiento.

```

12  private void Update()
13  {
14      transform.Rotate(0, 0, 1);
15  }
16
17  public void SetUp(Vector3 start, Vector3 control, Vector3 end)
18  {
19      StartCoroutine(Shoot(start, control, end));
20  }
21
22  private IEnumerator Shoot(Vector3 start, Vector3 control, Vector3 end)
23  {
24      _bezierTime = 0;
25
26      while (_bezierTime < 1.5f)
27      {
28          _bezierTime += Time.deltaTime;
29
30          _curveX = (((1 - _bezierTime) * (1 - _bezierTime)) * start.x)
31              + (2 * _bezierTime * (1 - _bezierTime) * control.x)
32              + ((_bezierTime * _bezierTime) * end.x);
33          _curveY = (((1 - _bezierTime) * (1 - _bezierTime)) * start.y)
34              + (2 * _bezierTime * (1 - _bezierTime) * control.y)
35              + ((_bezierTime * _bezierTime) * end.y);
36
37          transform.position = new Vector3(_curveX, _curveY, 0);
38
39          yield return null;
40      }
41  }

```

Figura 7.34: Código para el movimiento del proyectil

Finalmente, para que el proyectil haga daño al cocodrilo, necesita detectar el contacto con este. Cuando lo haga, entrará en dos funciones del jugador, enviando su posición actual con una y haciendo daño al cocodrilo con otra y se destruirá el objeto.

En caso de que el proyectil no golpee al jugador, se destruirá al chocar con el entorno.

Por otra parte, para que el castor haga daño al personaje por contacto, deberá detectar la colisión en caso de que no esté muerto y, acto seguido, entrará en dos funciones del jugador, enviando su posición actual con una y haciendo daño al cocodrilo con otra.

El código se puede observar en la figura 7.24.

### 7.4.3. Recibir daño

Como el castor recibe daño de la misma forma que la tortuga, usará el mismo script que el 7.1.4, quitando la parte de esconderse.

#### **7.4.4. Muerte**

La muerte del castor es igual a la tortuga, por lo que usará el script del apartado 7.1.5.

# Capítulo 8

## Puntuación

La puntuación obtenible en este videojuego se cuenta con muslos de carne, de los cuales hay diez en cada uno de los niveles, que el jugador debe intentar obtener si quiere lograr la máxima puntuación posible.

### 8.1. Funcionamiento

El funcionamiento de la puntuación dentro de cada nivel ha sido explicado anteriormente en el punto 6.7, pero aparte de eso, en el selector de niveles el jugador podrá observar la máxima puntuación obtenida en cada uno de ellos.

Para que cada uno de los contadores detecte la puntuación correspondiente, disponen de un script que le dice en qué nivel se encuentra al finalizar para guardar la puntuación y, de la misma manera, el contador de ese nivel también usará dicho script para obtener esa puntuación.

Con la ayuda del script de la detección de nivel, también podrá desbloquear el siguiente nivel.

```
7   public bool Level1;  
8  
9   public bool Level2;  
10  
11  public bool Level3;  
12  
13  public bool Level4;
```

Figura 8.1: Script para detectar el nivel

```

26     if (_detectLevelScript.Level1)
27     {
28         if (!LevelUnlock.LevelManager.SecondLevelUnlock)
29         {
30             LevelUnlock.LevelManager.SecondLevelUnlock = true;
31             PlayerPrefs.SetInt("SecondLevel", 1);
32         }
33
34         if (PlayerPrefs.GetInt("Score1", 0) < TextManager.TManager.Counter.CurrentCoins)
35         {
36             PlayerPrefs.SetInt("Score1", TextManager.TManager.Counter.CurrentCoins);
37         }
38     }

```

Figura 8.2: Código para guardar la puntuación del primer nivel

```

22     if (_detectLvlScript.Level1)
23     {
24         if (PlayerPrefs.HasKey("Score1"))
25         {
26             _hamLevel1.sprite = _hamCounter[PlayerPrefs.GetInt("Score1", 0)];
27         }
28
29         else
30         {
31             _hamLevel1.sprite = _hamCounter[0];
32         }
33     }

```

Figura 8.3: Script para actualizar el contador según el nivel

## 8.2. Uso

El principal objetivo de la puntuación es que el jugador tenga otra motivación aparte de simplemente querer pasarse cada nivel, de esta manera, si no ha conseguido todos los muslos de carne, se verá tentado a volver a jugar el nivel para ver si esta vez logra obtenerlos todos.

En caso de haberse pasado todos los niveles con la máxima puntuación, puede reiniciar el juego y volver a empezar de cero, teniendo otra vez la motivación inicial.

Cuando el jugador pulse el botón de reinicio, se borrarán todas las puntuaciones y todos los niveles excepto el primero volverán a estar bloqueados

y no podrán jugarse hasta pasarse el anterior.

Para que las puntuaciones se reinic peace, el reset llama a una función que actualiza los sprites de las imágenes.

```
75  public void ResetGame()
76  {
77      PlayerPrefs.DeleteKey("Score1");
78      PlayerPrefs.DeleteKey("Score2");
79      PlayerPrefs.DeleteKey("Score3");
80
81      LevelUnlock.LevelManager.SecondLevelUnlock = false;
82      LevelUnlock.LevelManager.ThirdLevelUnlock = false;
83      LevelUnlock.LevelManager.FourthLevelUnlock = false;
84
85      PlayerPrefs.DeleteKey("SecondLevel");
86      PlayerPrefs.DeleteKey("ThirdLevel");
87      PlayerPrefs.DeleteKey("FourthLevel");
88
89      _lockLevel2.SetActive(true);
90      _lockLevel3.SetActive(true);
91      _lockLevel4.SetActive(true);
92
93      RefreshSprite();
94  }
95
96  private void RefreshSprite()
97  {
98      _hamLevel1.sprite = _hamCounter[0];
99      _hamLevel2.sprite = _hamCounter[0];
100     _hamLevel3.sprite = _hamCounter[0];
101     _hamLevel4.sprite = _hamCounter[0];
102 }
```

Figura 8.4: Código para reiniciar el juego

# Capítulo 9

## Checkpoints

Pasarse un nivel sin caerse ninguna vez puede ser todo un reto, sobre todo en las cuevas. Para evitar que el jugador tenga que volver a empezar cada vez que caiga, existen los puntos de guardado, pequeñas zonas en las que se actualizará el punto de reaparición y de esta manera no tener que recorrer todo el nivel de nuevo.

En este caso, los checkpoints son ranas subidas a una roca.

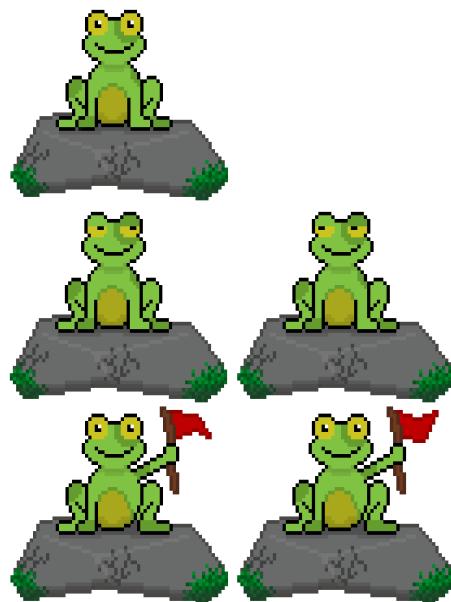


Figura 9.1: Spritesheet del checkpoint

## 9.1. Activación

Para que el checkpoint se active, el jugador deberá pasar por delante, momento en que la rana sacará una bandera para indicar que la posición ha sido guardada como punto de reaparición.

```
18     private void OnTriggerEnter2D(Collider2D other)
19     {
20         if (other.GetComponent<Movimiento>())
21         {
22             _anim.SetBool("_triggered", true);
23             _bc2D = this.GetComponent<BoxCollider2D>();
24             _bc2D.enabled = false;
25         }
26     }
27 }
```

Figura 9.2: Código para activar la animación del checkpoint

El jugador tendrá un punto de aparición ya asignado manualmente, pero los checkpoints deberá detectarlos de forma automática, por lo tanto se necesita que el cocodrilo pueda acceder al objeto del checkpoint.

Para asignar el punto de aparición, se le da su posición a la del checkpoint para que, en caso de caer sin haber pasado por ninguno, reaparezca en la zona inicial y no en un sitio aleatorio.

```
28     _checkpoint = _spawnPoint;
```

Figura 9.3: Código para asignar el punto de aparición

Cada vez que el personaje pase por delante de un checkpoint, la posición de este se actualizará.

```
33     private void OnTriggerEnter2D(Collider2D other)
34     {
35         //cuando el jugador entre en contacto con el checkpoint lo activará
36         if (other.gameObject.name == "Checkpoint")
37         {
38             _checkpoint = other.gameObject;
39         }
}
```

Figura 9.4: Código para cambiar el checkpoint

De esta manera, cuando el jugador caiga, siempre que su vida sea superior a uno, reaparecerá en el último checkpoint activado.

El código se puede observar en la figura 6.53.

# Capítulo 10

## UI

La UI, también conocida como Interfaz de Usuario, es el punto de interacción entre el jugador y el juego, un diseño intuitivo que facilitará la jugabilidad. Esta puede ser diegética o no diegética, espacial o meta.

- La diegética está incluida en el mundo del juego, el personaje puede verla y tocarla y forma parte de la narrativa del juego.
- La no diegética es lo opuesto a la anterior, es lo que está fuera del mundo del juego, solo es visible para el jugador.
- La espacial es una mezcla de las dos anteriores, son elementos integrados en el juego pero que el personaje no tiene constancia de ellos, como por ejemplo los elementos de un tutorial.
- La meta son elementos que están dentro de la historia de juego pero no en el espacio de este, como por ejemplo gotas de sangre en la pantalla. Esto hace que la experiencia de juego sea más inmersiva.

### 10.1. Menú de inicio

Este menú es lo primero que verá el jugador al abrir el juego. Es un menú sencillo con todo lo necesario: el título del juego, un fondo y tres botones dentro de una tabla de madera. Estos últimos sirven para empezar a jugar, ir al menú de opciones o salir del juego, hacen una pequeña animación de ser presionados cuando el jugador los pulsa.



Figura 10.1: Menú de inicio

El fondo es una imagen aleatoria entre las cuatro de los diferentes niveles cada vez que abra el menú, de esta manera el menú obtiene algo más de vida al ser simple.

Para que el fondo sea aleatorio, se necesita obtener la imagen a cambiar y añadirle las diferentes que podrá tener y un número para obtener un valor que dirá qué imagen saldrá.

Se le dará un valor aleatorio entre cero y cuatro al número, ya que disponemos de cuatro imágenes, y se le aplicará dicho valor a la imagen de fondo para que esta cambie cada vez que inicie el menú.

```
14  void Start()
15  {
16      _imageNumber = Random.Range(0, 4);
17
18      _menuImage.sprite = _menuImageSprite[_imageNumber];
19 }
```

Figura 10.2: Script para el cambio de imagen aleatorio

El botón de empezar abrirá el selector de niveles una vez pulsado.

El botón de opciones, cargará una nueva escena que contiene el menú de opciones.

Por último, el botón de salida hará que el jugador cierre el juego.

### 10.1.1. Selector de nivel

El selector de nivel está en la misma escena que el menú de inicio, por lo que cuando aparezca, lo hará justo encima de este, tapando sus botones.

Este menú consta en un panel de piedra con los cuatro niveles en su interior, tres de ellos bloqueados con un candado en caso de ser la primera vez que se juega, y dos botones encima de pequeñas tablas de madera, los cuales sirven para reiniciar el juego o cerrar el selector.

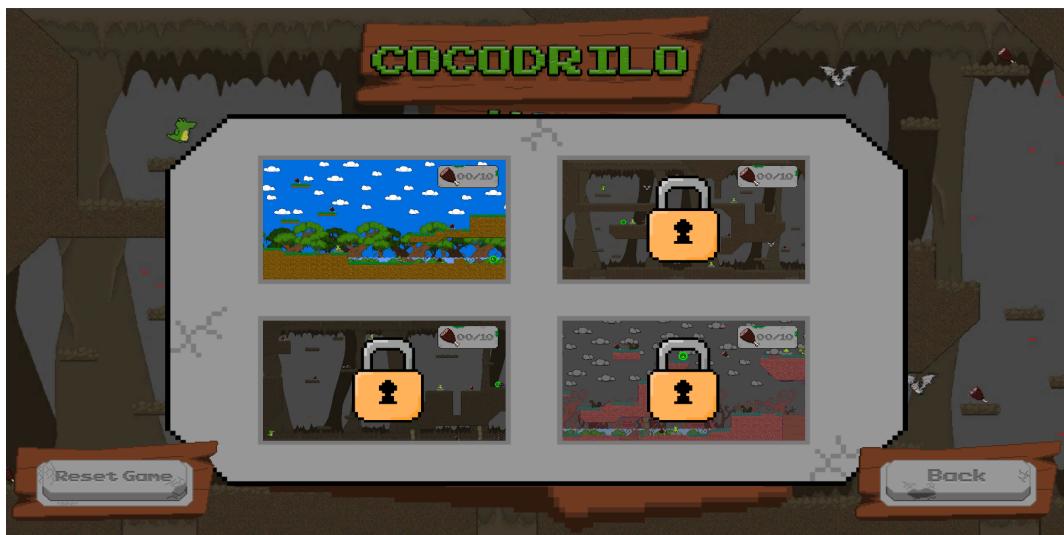


Figura 10.3: Menú de selección de nivel

Para que el candado aparezca o no, al iniciar el menú detectará si el nivel ha sido desbloqueado, caso en el que no aparecerá y el jugador podrá jugar ese nivel.

Las imágenes con los niveles disponen de un componente de botón, para que cuando sean pulsadas cargue el nivel correspondiente.

El botón de reiniciar el juego, accederá a la función mostrada en la figura 8.4 y actualizará los contadores para dejarlos como estaban inicialmente.

## 10.2. Menú de opciones

En este menú el jugador podrá cambiar los valores de sonido del juego. Sigue con la estética sencilla como el menú de inicio, donde está el título, una imagen de fondo y unas tablas de madera en el centro. La diferencia es

que este menú dispone de tres sliders para modificar valores, sus respectivos textos para saber qué hace cada uno y un botón para volver al menú de inicio.



Figura 10.4: Menú de opciones

El fondo sigue siendo una imagen aleatoria entre los niveles, la cual cambiará cada vez que se acceda a la escena.

En el caso de los sliders, están vinculados al mezclador de audio del juego para que su valor se modifique en base a dónde esté situado cada slider.

```

61     public void SetGeneralVolume(float volume)
62     {
63         //coge el anterior valor
64         PlayerPrefs.GetFloat("GeneralVolume", volume);
65         //valor del slider vinculado al audio
66         AudioMixer.SetFloat("volume", Mathf.Log10(volume) * 20);
67         //guarda el valor actual
68         PlayerPrefs.SetFloat("GeneralVolume", volume);
69     }
70
71     public void SetMusicVolume(float volume)
72     {
73         PlayerPrefs.GetFloat("MusicVolume", volume);
74         AudioMixer.SetFloat("musicvolume", Mathf.Log10(volume) * 20);
75         PlayerPrefs.SetFloat("MusicVolume", volume);
76     }
77
78     public void SetEffectsVolume(float volume)
79     {
80         PlayerPrefs.GetFloat("EffectsVolume", volume);
81         AudioMixer.SetFloat("effectsvolume", Mathf.Log10(volume) * 20);
82         PlayerPrefs.SetFloat("EffectsVolume", volume);
83     }

```

Figura 10.5: Código para actualizar el sonido con los sliders

Cuando la posición del slider varíe, también se mostrará un porcentaje para que el jugador sepa cuánto se escuchará el sonido que esté modificando.

Por último, el botón de volver atrás hará que el jugador regrese al menú de inicio.

### 10.3. UI in-game

Dentro del juego hay poca interfaz. Como es un videojuego sencillo y que sigue esa misma estética, añadir mucho contenido a la interfaz podría cargar demasiado la vista del jugador. En la interfaz se puede encontrar la vida del jugador, para que en todo momento sea consciente de cuánta le queda, la puntuación actual en el nivel y el texto de ayuda inicial o cuando obtiene una nueva habilidad.



Figura 10.6: UI in-game inicial

El texto inicial aparece en la esquina inferior izquierda ya que el jugador debe avanzar hacia la derecha, así molesta lo menos posible. Gracias a él, el jugador puede leer los controles básicos antes de empezar a jugar, por lo tanto solo aparecerá en el primer nivel. Una vez han pasado diez segundos, el texto desaparecerá para no estorbar la visión del jugador.

```

23     private void DestroyText()
24     {
25         if (_timeToDespawn <= 0) //si el tiempo llega a 0 el texto desaparece
26         {
27             Destroy(this.gameObject);
28         }
29     }

```

Figura 10.7: Función para que desaparezca el texto

Cuando el cocodrilo obtenga una nueva habilidad, aparecerá el texto correspondiente con la explicación de cómo usarla en la parte inferior central de la pantalla, para asegurar que el jugador lo vea. De la misma manera que el anterior, al cabo de un tiempo desaparecerá.



Figura 10.8: UI in-game al obtener una habilidad

## 10.4. Menú de pausa

El jugador podrá pausar el juego cuando quiera, deteniendo todas las funciones que usen el tiempo. El menú de pausa consta de unos tablones de madera en el centro de la pantalla junto a tres botones, de los cuales uno abre el menú de opciones, otro reinicia el nivel y el último permite regresar al menú de inicio.

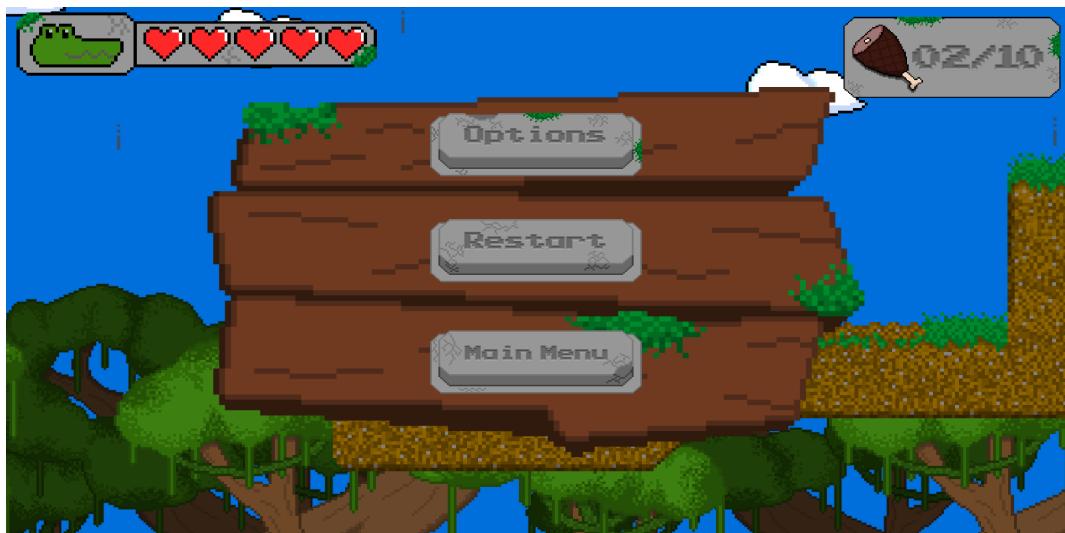


Figura 10.9: Menú de pausa

El menú de pausa se abre al pulsar la tecla Escape, momento en que el valor del tiempo en el juego pasará a ser cero para que no siga avanzando. Cuando el menú esté activo y la tecla vuelva a ser pulsada, reanudará el juego.

```
94     private void ShowMenu()
95     {
96         if (Input.GetKeyDown(KeyCode.Escape) && !IsPaused)
97         {
98             Pause();
99         }
100        else if (Input.GetKeyDown(KeyCode.Escape) && IsPaused)
101        {
102            Resume();
103        }
104    }
105 }
```

Figura 10.10: Función para mostrar o esconder el menú de pausa

```

107     private void TimeScale()
108     {
109         if (IsPaused)
110         {
111             Time.timeScale = 0;
112         }
113
114         else
115         {
116             Time.timeScale = 1;
117         }
118     }

```

Figura 10.11: Función para parar o reanudar el tiempo

El botón de opciones sobreescribe los botones actuales por los sliders de volumen al ser pulsado y añade un nuevo botón para volver al menú de pausa.

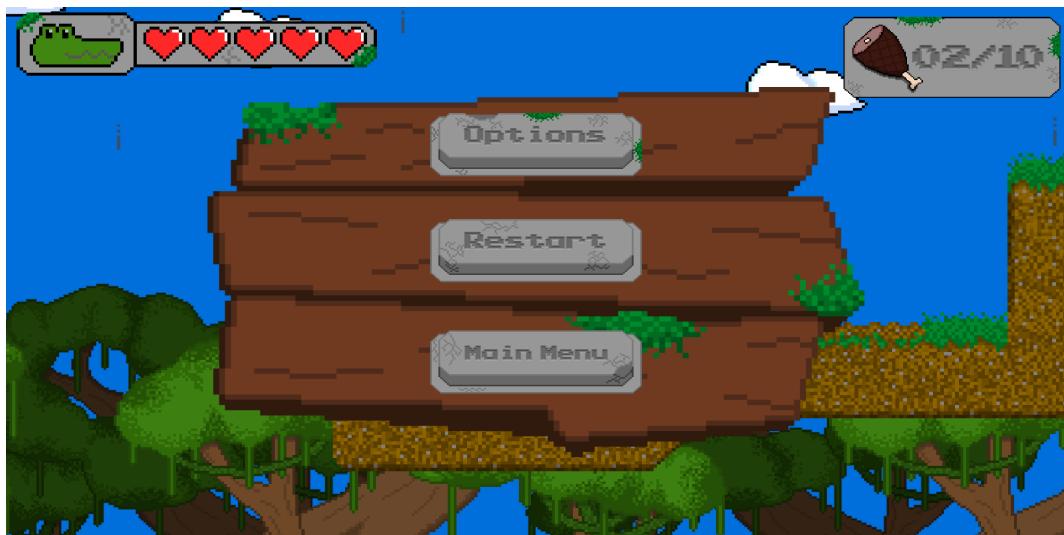


Figura 10.12: Menú de opciones in-game

El botón de reiniciar nivel accede a una función que vuelve a cargar la escena en la que está, cancelando también la acción de pausa para que el juego vuelva a funcionar.

Por último, el botón de regresar al menú, como su nombre indica, hará que el jugador vaya al menú inicial al ser pulsado.

## 10.5. Nivel completado

Al llegar al final de cada nivel, aparecerá la pantalla de victoria, oscureciendo levemente el fondo, anunciando en grande la victoria y dándole dos posibles opciones al jugador, ir al siguiente nivel o al menú inicial.



Figura 10.13: Menú de victoria

El botón de siguiente nivel llevará al jugador al nivel que vaya después del que ha jugado.

El botón de ir al menú de inicio llevará al jugador a dicho menú.

En el caso del último nivel, el botón de ir al siguiente es eliminado ya que actualmente no hay más, dejando solo el botón de volver al menú más centrado en la pantalla.



Figura 10.14: Menú de victoria en el último nivel

## 10.6. Derrota

En caso de que la vida del cocodrilo llegue a cero y muera, aparecerá la pantalla de derrota, muy similar a la de victoria pero cambiando el cartel y el botón de siguiente nivel por el de reiniciar el nivel actual.

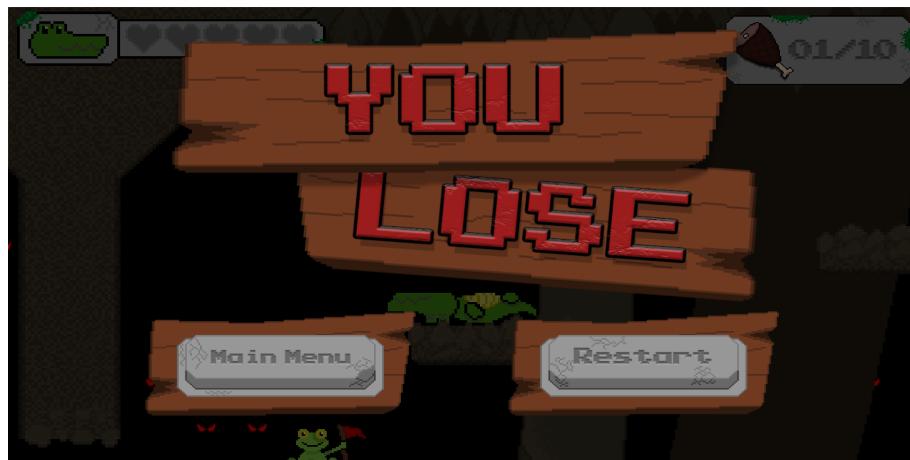


Figura 10.15: Menú de derrota

El botón de reiniciar nivel, como su nombre indica, hará que el nivel actual vuelva a comenzar.

El botón de volver al menú funciona de la misma manera que el de la pantalla de victoria.

# **Capítulo 11**

## **Partículas**

Las partículas son efectos visuales que se le añaden a los videojuegos para simular de forma más realista acciones que de otra forma no quedarían tan bien, como por ejemplo el fuego y explosiones.

Hay que tener mucho cuidado con no abusar de ellas, ya que pueden llegar a consumir muchos recursos del juego y hacer que este se ralentice.

Para añadir la textura a las partículas, se ha hecho mediante el uso de shaders, el cual permite modificar los parámetros de la textura, en este caso usado para pixelizarla y que encaje mejor en la estética del juego.

### **11.1. Fuego del lanzallamas**

El ataque del cocodrilo fue el primero en hacerse ya que era indispensable tenerlo. Sin este sistema de partículas, el personaje haría daño simplemente abriendo la boca, quedando muy raro.

Está hecho con dos sistemas de partículas diferentes que se juntan para crear un único elemento con más detalle. El primero simula el fuego, mientras que el segundo simula humo para añadir más realismo al lanzamiento del primero.



Figura 11.1: Lanzallamas activo

El fuego debía ser de corto alcance, por lo que la duración de las partículas es bastante reducida y la velocidad elevada, para que salga disparado desde el primer momento. Como el personaje es pequeño, el tamaño de las partículas debía serlo aún más para simular que salen del interior de su boca.

Al seguir el videojuego una estética de píxel art, no puede haber demasiadas partículas en pantalla porque molestaría a la vista del jugador, por lo que la emisión también es bastante reducida, siendo en este caso de veinticinco partículas por segundo.

La forma de salida es la de un cono con el ángulo muy pequeño y el radio reducido al mínimo para que queden comprimidas.

Las partículas también heredan una parte de la velocidad del cocodrilo para que cuando este se mueva, se adapten a ese movimiento.

El color irá cambiando en base al tiempo que llevan activas, siendo amarillas al principio y rojas al final, pasando también por el naranja.

Al salir de la boca del cocodrilo deben ser pequeñas y con el tiempo hacerse más grandes, por lo que cuanto más tiempo estén en pantalla más grandes se harán.

Para añadirle algo más de vida al movimiento, se le aplica una rotación aleatoria para que no salgan siempre iguales.

Por último, falta añadir la textura para darle forma a las partículas, la cuál

ha sido hecha por la persona encargada del arte. Añadiéndola al shader hecho para el fuego, se obtienen llamas con estética píxel art.

Como la imagen de la textura dispone de cuatro elementos utilizables, hay que modificar la carga de la textura en el sistema de partículas, poniendo que consta de una cuadrícula de dos por dos.

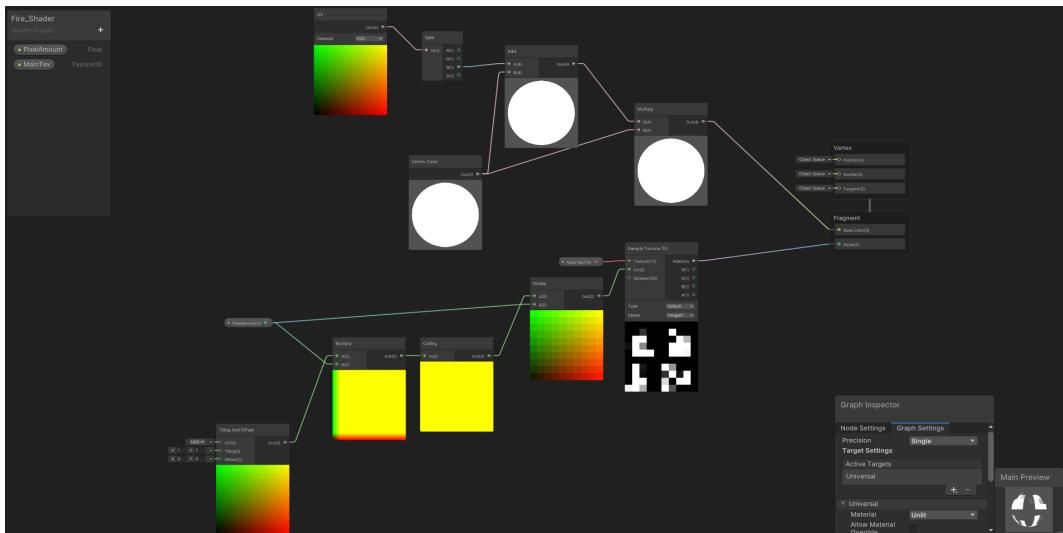


Figura 11.2: Shader del lanzallamas

De esta forma, la parte principal que es el fuego ya queda hecha.

Para el humo, la mayoría de parámetros se mantienen igual, pero algunos cambian como el tamaño y la emisión, siendo menores para que no destaque más que el fuego. El color también varía, ya que es humo es de tonalidades grises.

Una vez realizados los dos, se juntan, quedando como se ve en la figura 11.1.

## 11.2. Lluvia

La lluvia se trata de un sistema de partículas decorativo para los niveles de pantano, añadiendo algo más de similitud a un pantano real.

Para que cuando el jugador inicie el nivel la lluvia ya esté cayendo, la opción de precalentamiento estará activa, así será como si el nivel llevase un tiempo activo. Al estar el sistema de partículas a una altura bastante elevada, la

duración de las partículas tiene que ser suficiente como para llegar al suelo. El tamaño debe ser pequeño para más similitud a la lluvia y se le añadirá gravedad, de esta manera no hace falta que tengan velocidad.

Como la lluvia alcanza todas las zonas del nivel, la emisión debe ser algo elevada para que la lluvia se pueda llegar a apreciar, siendo en este caso setenta y cinco partículas por segundo.

La forma de salida es una caja con un valor horizontal elevado para que caiga por todo el nivel.

El color será azul constantemente, ya que la lluvia no cambia de color al caer.

Para añadir un pequeño detalle, cuando la lluvia colisione con cualquier objeto del nivel, desaparecerá.

Por último, hay que añadir la textura con el shader para que obtenga la forma.

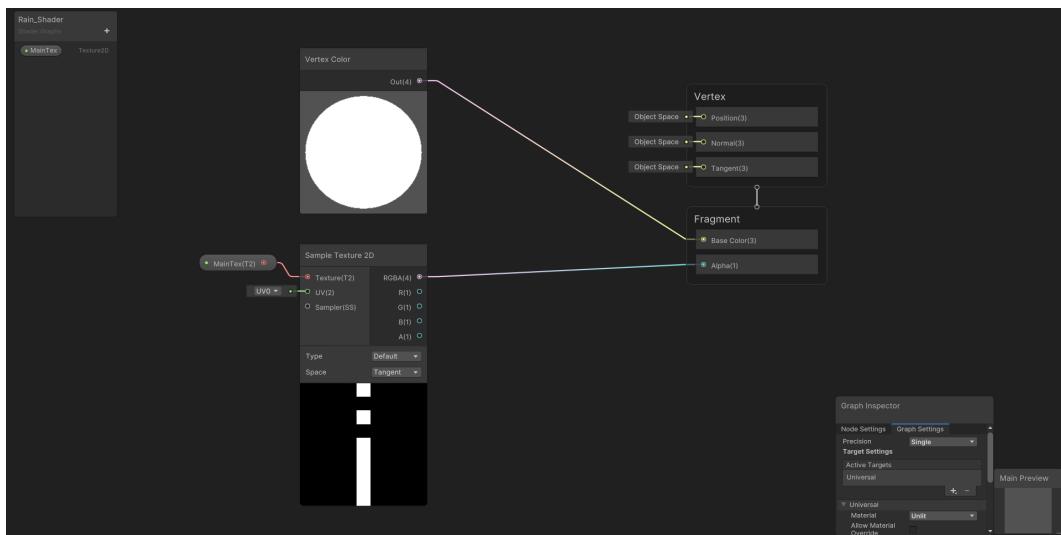


Figura 11.3: Shader de la lluvia



Figura 11.4: Lluvia en el primer nivel

Como el segundo pantano es más oscuro, la cantidad de lluvia es mayor, por lo que la emisión aumenta a trescientas partículas por segundo y se le añade fuerza al eje horizontal para que caiga en diagonal, simulando que el viento es fuerte en ese nivel.



Figura 11.5: Lluvia en el cuarto nivel

### **11.3. Polvo al caminar**

Para que el caminar del cocodrilo no se vea tan simple, cada cierto tiempo saldrán partículas de polvo de sus pies para simular que levanta tierra al caminar.

Como son pequeñas partículas de polvo, su duración será bastante reducida de la misma manera que el tamaño. Se le añadirá un poco de gravedad para que cuando se eleven un poco con la velocidad inicial, vuelvan a bajar. Las partículas saldrán todas de golpe ya que sería cuando el cocodrilo golpea el suelo, siendo en este caso cinco partículas, suficientes para añadir el detalle deseado.

La forma será de un cono apuntando hacia arriba con un poco de ángulo para que vayan hacia atrás, reduciendo el radio para que no salgan demasiado separadas entre ellas. El color será marrón oscuro cuando aparezcan, volviéndose más claro con el tiempo. Para que no sean estáticas, con el tiempo irán rotando para simular que han salido disparadas.

Por último, falta añadir la textura para darle forma a las partículas. Añadiéndola al shader hecho para el polvo, cambia la estética a píxeles para que encaje mejor con el estilo del juego.

Como la imagen de la textura dispone de cuatro elementos utilizables, hay que modificar la carga de la textura en el sistema de partículas, poniendo que consta de una cuadrícula de dos por dos.

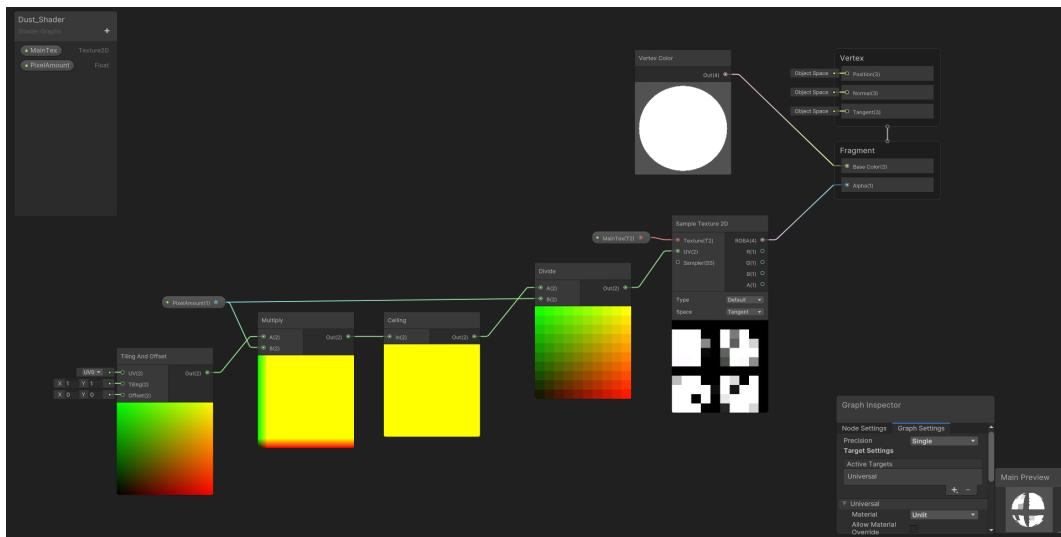


Figura 11.6: Shader del polvo



Figura 11.7: Partículas de polvo al caminar

# Capítulo 12

## Niveles

Este videojuego consta de cuatro niveles jugables, cada uno más oscuro que el anterior. Estos sirven como pequeños tutoriales para que el jugador aprenda a utilizar las diferentes mecánicas obtenibles en cada uno de ellos.

Para su creación, primero se hizo un boceto rápido a mano, lo que hace que su calidad no sea la más óptima para pasarlo a Unity. Este servía para tener una idea de cómo sería y ver que no fuera demasiado grande ni demasiado pequeño.

Cada nivel tiene una duración aproximada de cinco minutos, depende de como juegue cada persona.

Este boceto se centra principalmente en el recorrido del nivel, marcando los sitios importantes como dónde usar el doble salto y el dash. También se marca dónde estarán los muslos de carne, el power up, el punto de inicio y las caídas.

Después de este primer boceto, se hizo otro con Photoshop, mejorando la calidad, añadiendo más elementos al mapa y una leyenda para que fuera más entendible.

Los nuevos elementos añadidos fueron los enemigos, los checkpoints y el final del nivel. Aparte de esto, algunos de los elementos ya añadidos en el boceto inicial fueron actualizados con sus respectivos sprites.

### 12.1. Pantano

El primer nivel consiste en un pantano con un estilo alegre que hace que el jugador se sienta agusto. En este nivel, el jugador se encontrará con tortugas

como enemigos y aprenderá a usar la habilidad de doble salto.

Las tortugas se deben a que, aparte de ser un pantano, son el enemigo más simple. De esta manera, el jugador puede acostumbrarse a la interacción con los enemigos sin sufrir demasiado.

Para la práctica con el doble salto, se crearon zonas en las que si querías acceder tienes que aprender a usarlo, ya que es una habilidad que será de utilidad en los próximos niveles.



Figura 12.1: Nivel del pantano

### 12.1.1. Boceto inicial

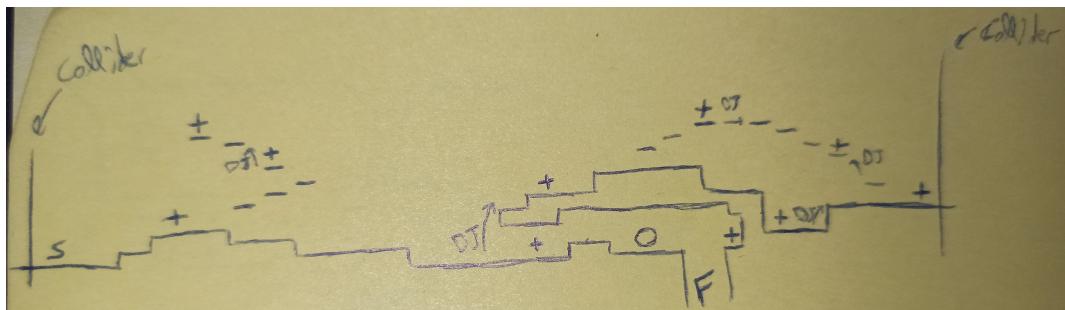


Figura 12.2: Boceto a mano

### 12.1.2. Boceto final

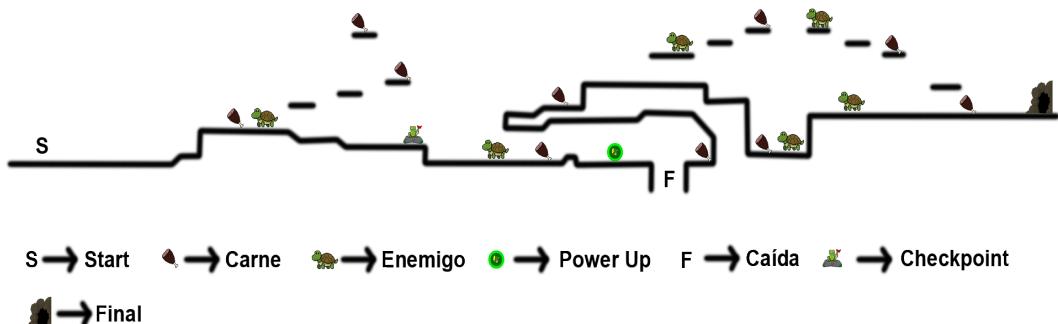


Figura 12.3: Boceto con Photoshop

## 12.2. Cueva

El segundo nivel se sitúa en el interior de una cueva, tomando un estilo un poco más oscuro que el anterior. Esta vez, los enemigos serán murciélagos que no dudarán en atacar al jugador si se acerca. El cocodrilo aprenderá a usar el dash.

En este nivel hay murciélagos como enemigos debido a que son el primer animal que se viene a la mente cuando se piensa en una cueva. Además, tienen una funcionalidad muy parecida a la tortuga, pero también vuelan y atacan al jugador. Esto supone un pequeño aumento en la dificultad al jugar.

El jugador se encontrará en una zona al principio de la que no podrá salir si no obtiene el dash, de esta manera aprenderá que puede pasar por pequeños huecos con su ayuda.



Figura 12.4: Nivel de la cueva

### 12.2.1. Boceto inicial

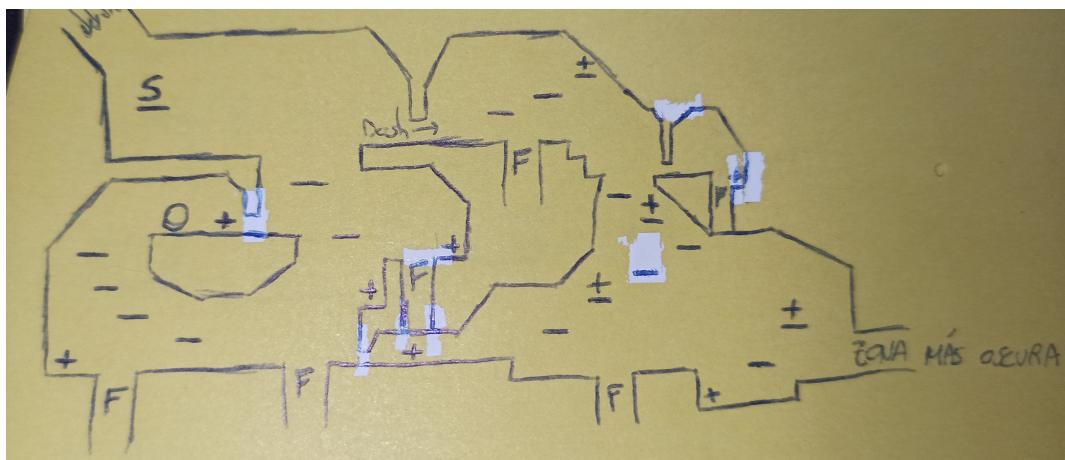


Figura 12.5: Boceto a mano

### 12.2.2. Boceto final

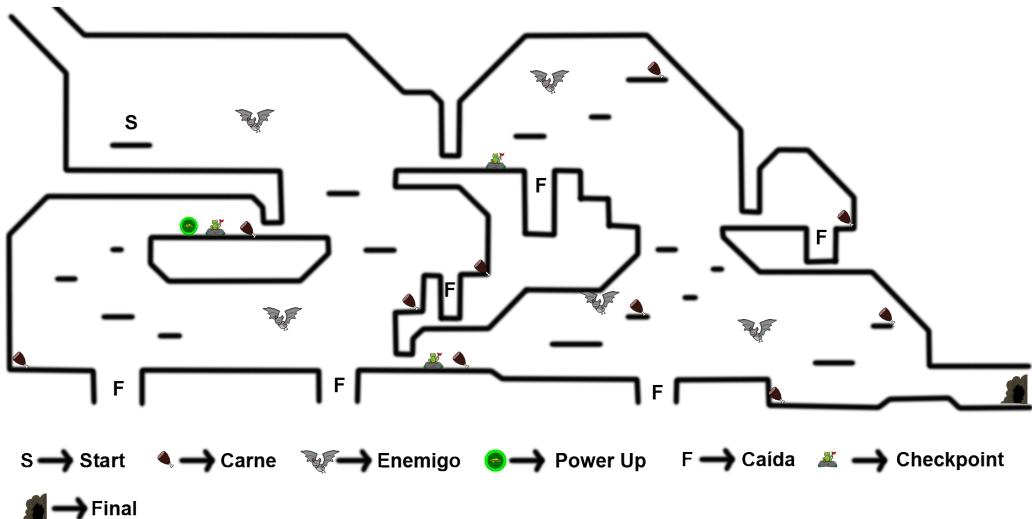


Figura 12.6: Boceto con Photoshop

### 12.3. Cueva oscura

El tercer nivel vuelve a ser en una cueva, pero esta vez con una paleta de colores más oscura. Los enemigos que esperan al jugador son arañas que parecen inofensivas a simple vista, pero aprenderá que no debe infravalorarlas. En este nivel obtendrá la bola de fuego, la cuál puede usar para abrirse paso o atacar a los enemigos.

Las arañas son otro ser vivo conocido por vivir en cuevas, aunque también puede hacerlo fuera de estas. La primera vez que el jugador se tope con una se sorprenderá.

Este nivel puede ser completado sin la necesidad de usar la bola de fuego, pero si el jugador quiere conseguir todos los muslos de carne deberá romper las paredes que le impiden el paso.



Figura 12.7: Nivel de la cueva oscura

### 12.3.1. Boceto inicial

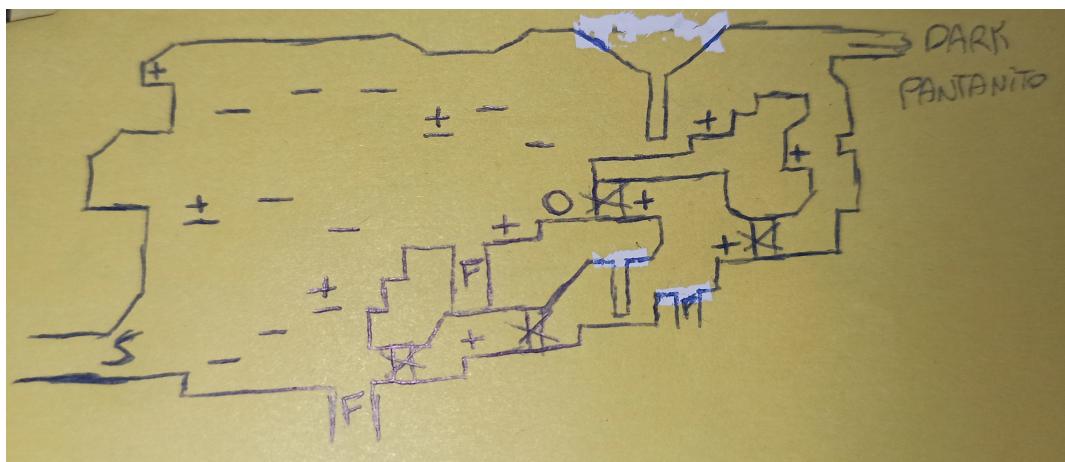


Figura 12.8: Boceto a mano

### 12.3.2. Boceto final

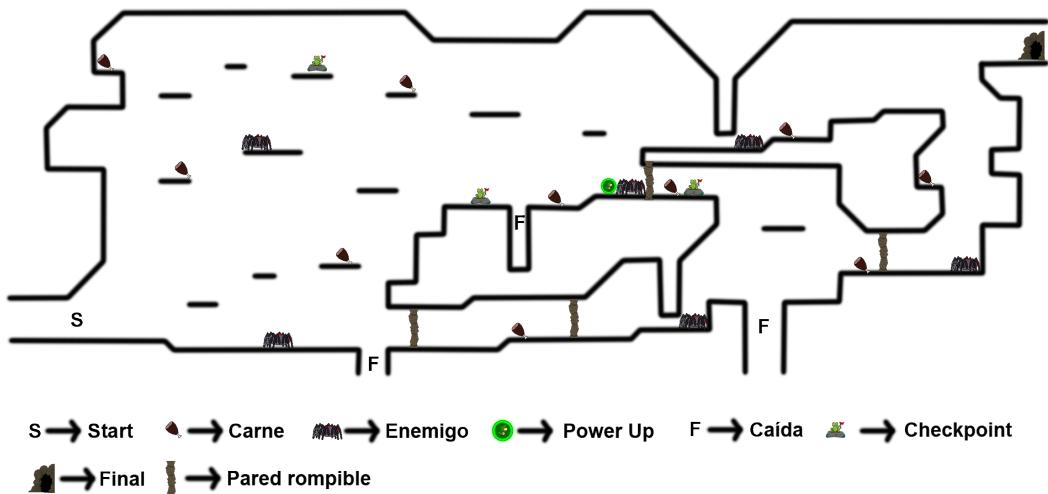


Figura 12.9: Boceto con Photoshop

## 12.4. Pantano oscuro

En el cuarto, y último nivel, el jugador regresará a un pantano, pero esta vez más tenebroso, cambiando la paleta de colores y aumentando la cantidad de lluvia. En este caso los enemigos serán castores, los cuales atacan a distancia, haciendo que sean los más difíciles que el jugador haya enfrentado hasta el momento. Para finalizar con las habilidades obtenibles, el jugador conseguirá el gancho.

Este nivel puede ser completado sin el uso del gancho, aunque es más complicado hacerlo que con el anterior.

Al ser el último nivel, contiene un poco de todos los anteriores: doble salto, dash y el uso de la bola de fuego para romper paredes.

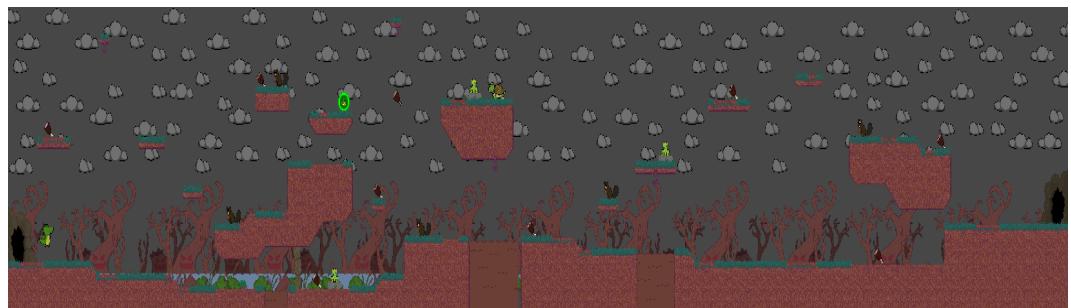


Figura 12.10: Nivel del pantano oscuro

#### 12.4.1. Boceto inicial

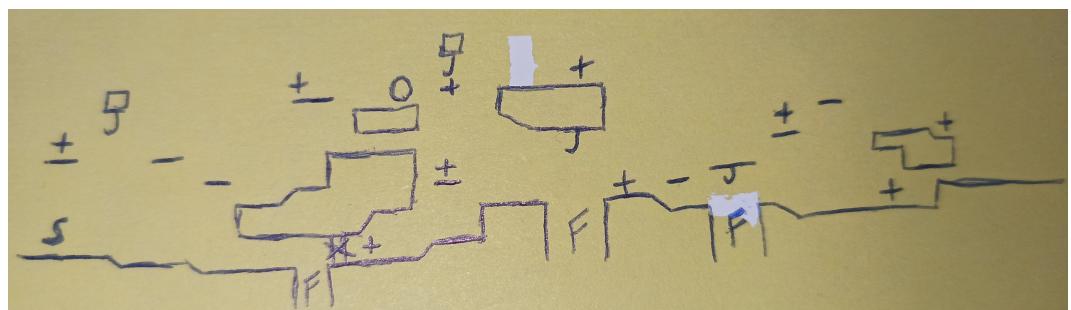


Figura 12.11: Boceto a mano

### 12.4.2. Boceto final

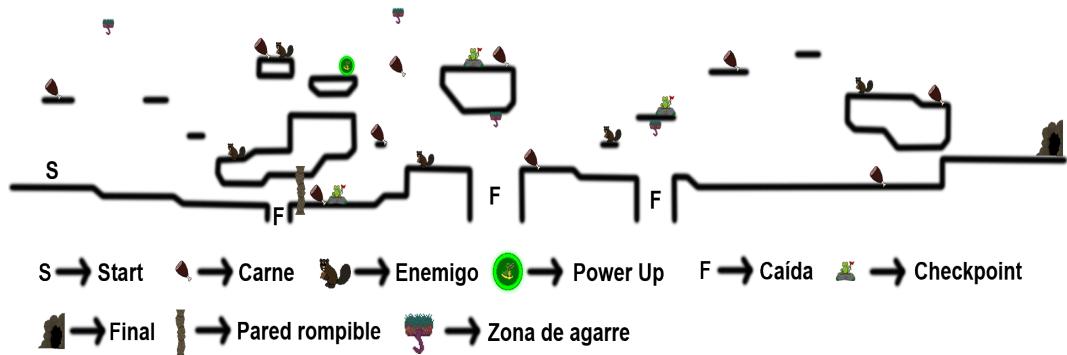


Figura 12.12: Boceto con Photoshop

# Capítulo 13

## Tilemap

El tilemap es un componente de Unity que tiene un sistema de almacenamiento para tiles, pudiendo manejarlas para crear niveles 2D. Las tiles son sprites adaptados para su uso con este componente, pudiendo ser colocados en una cuadrícula para que quede más lineal y ordenado.

Con este componente, las tiles se pintan en la cuadrícula, pudiendo también borrarlas, seleccionarlas para moverlas y seleccionar un grupo de tiles para pintarlas al mismo tiempo.

### 13.1. Spritesheets

Las spritesheets son imágenes (normalmente de gran tamaño) donde se juntan otras muchas imágenes, de un mismo estilo, para poder localizarlas más fácilmente y así tener un mejor orden del material usado, al mismo tiempo que mejoran mucho el rendimiento.

En el caso de las tiles, debían ser todas del mismo tamaño para que después fuera más fácil identificar cada una de ellas dentro de la spritesheet y separarlas.

#### 13.1.1. Separación de tiles

Las tiles se hacen de 16x16 píxeles, pero como el tamaño era muy pequeño, se exportaron en uno mayor, siendo este 80x80, para que quedasen mejor en escala con los otros elementos.

Para separarlas, Unity tiene un componente llamado “Sprite Editor”, el cual

permite editar las imágenes. Con ello se puede cortar la imagen indicando el tamaño de las celdas, dividiéndola en diferentes sprites que serán las tiles. En este caso se indica que la división sea de 80x80 píxeles que es el tamaño de las tiles.

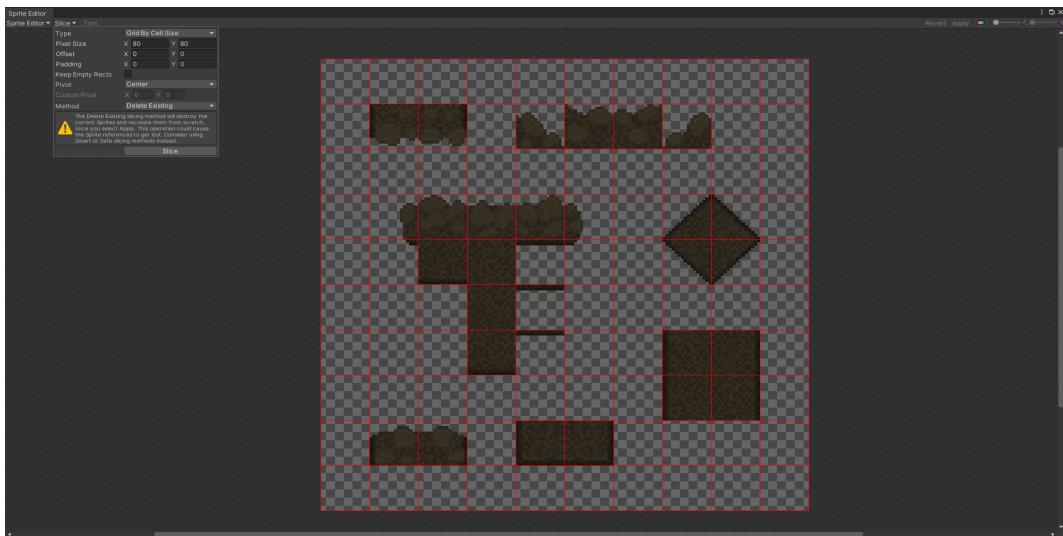


Figura 13.1: División de las tiles en el Sprite Editor

## 13.2. Creación del nivel

Con las tiles ya separadas, hay que crear la zona para colocarlas. Creando un componente “Tilemap”, se crea automáticamente una cuadrícula para pintar las tiles, a la cual se le pueden cambiar los valores para adaptarla mejor a lo necesario. Con la cuadrícula creada se pueden crear nuevos componentes “Tilemap” dentro de esta para las futuras capas que serán pintadas.

Para poder pintar las tiles en su respectivo “Tilemap”, primero deben ser importadas en una paleta.



Figura 13.2: Cuadrícula para colocar las tiles

### 13.2.1. Palettes

Las paletas son una opción que dispone el componente de “Tilemap”, sirviendo para poder añadir las tiles que deban ser pintadas y ordenarlas como más le guste al usuario.

En el caso de este proyecto, se ha creado una paleta para cada capa de los diferentes niveles. De esta manera, todas las tiles estaban perfectamente ordenadas para acceder a las necesarias en cada momento.

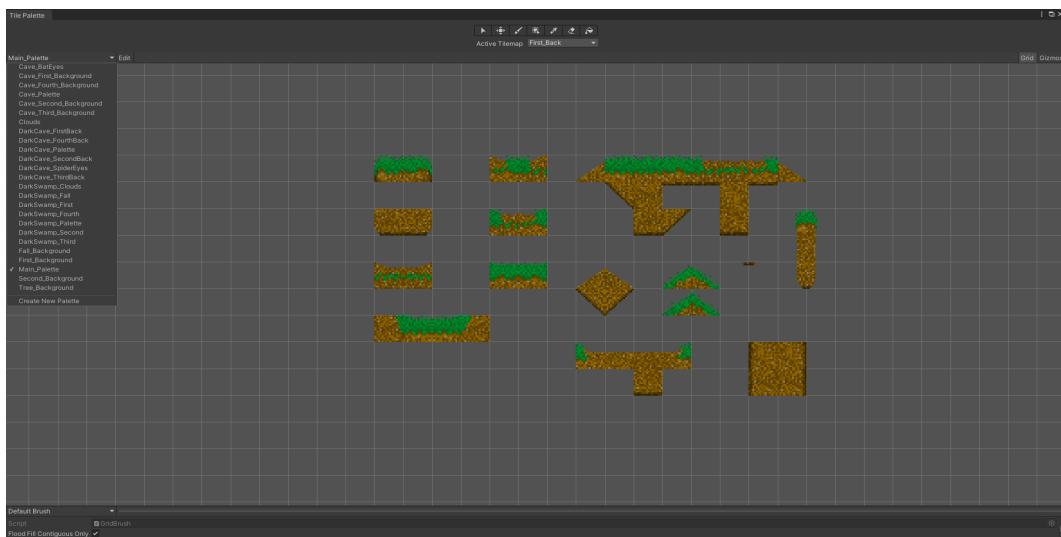


Figura 13.3: Espacio de trabajo del Tile Palette

### 13.2.2. Layers

Las tiles se distribuyen en diferentes “Tilemap”, pudiendo así modificar cuáles se verán sobre otras. Mientras más en el fondo debe verse, menor valor tendrá en el orden de capas.

Todos los niveles disponen de seis capas diferentes, siendo cuatro de estas para decorar el fondo y dos para el terreno que tocará el personaje.

Las dos capas que tocará el personaje son el suelo y la pared, por lo que su valor en el orden de capas será el más alto entre ellas para que siempre se vea.

Las cuatro capas de fondo estarán ordenadas de mayor a menor valor en base a cuál está más cerca del personaje. Poniendo de ejemplo el primer nivel, los arbustos tendrán el valor más alto y los árboles oscuros el más bajo.



Figura 13.4: Capas del tilemap

### 13.2.3. Colliders

Los únicos “Tilemap” que podrán colisionar con el personaje son el suelo y la pared, ya que son el terreno que el cocodrilo tocará. Para que estos pudieran y los otros no, era necesaria la adición del componente “Tilemap Collider”, el cual le añade la zona de colisión a las tiles.

Como la zona de colisión la hacía de forma automática, quedaba de forma errónea y tuvo que ser modificada para que quedase como se había planeado.

Con el tiempo se detectó un error con la colisión del suelo, ya que cada tile tenía su propia zona, quedando un pequeño espacio entre y una otra.

Para arreglarlo se añadió el componente “Composite Collider 2D” para que juntase todos los colliders y quedase una zona plana bajo el cocodrilo.

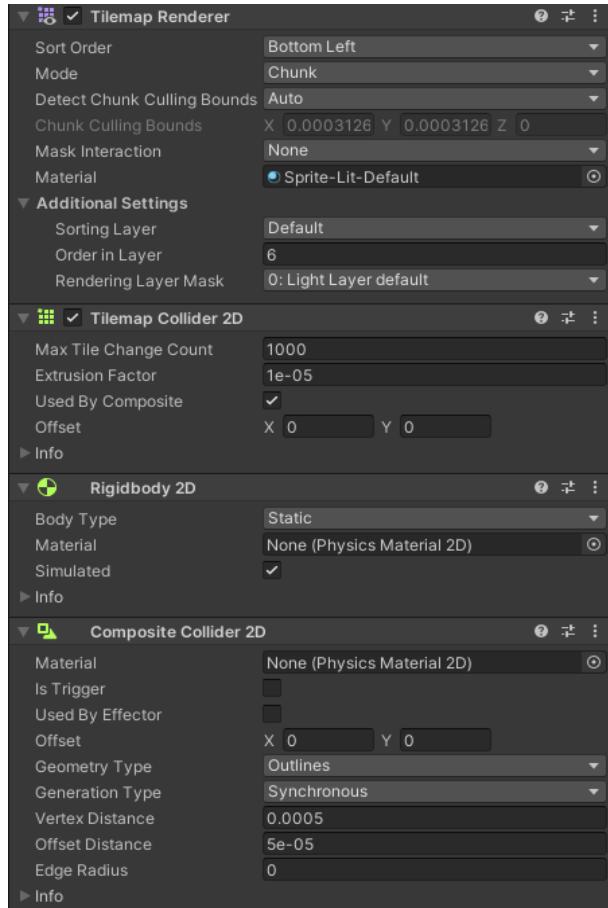


Figura 13.5: Componentes de la capa del suelo

### 13.3. Parallax

La última cosa que se añadió a la cámara y que no pudo ser añadida hasta tener el fondo fue el efecto parallax. Este hace que el fondo pueda desplazarse más lentamente que el jugador. Añadiendo un valor diferente a cada capa del fondo hace que uno se desplace más rápido que otro, dando así mayor sensación de movimiento al jugador.

Para programar el efecto parallax, era necesaria la creación de dos variables “float” para la modificación del valor en ambos ejes. También era necesario

obtener la posición de cámara para poder aplicar estos valores y modificar cómo se mueven las diferentes capas.

Con el fin de obtener la modificación deseada, se le aplica un nuevo vector2 a la posición de la capa, el cual multiplica la posición de la cámara por la variable creada en cada eje.

```
16  |  private void ParallaxMovement()
17  |  {
18  |  |  transform.position = new Vector2(_camera.position.x * _relativeMoveX, _camera.position.y * _relativeMoveY);
19  |  }
```

Figura 13.6: Función para el efecto parallax

# Capítulo 14

## Conclusión

En conclusión, este proyecto ha supuesto una aplicación práctica de todos los conocimientos obtenidos tras estos años de formación, siendo los más importantes los últimos tres meses, en los que el aumento del nivel de programación ha influido notablemente en la realización de este proyecto. También se han adquirido nuevos conocimientos, permitiendo superar los obstáculos que aparecían por el camino.

En este videojuego controlas un pequeño cocodrilo, cuyo objetivo es recoger los muslos de carne repartidos por los niveles, evitando a los enemigos y abriéndose paso entre los múltiples obstáculos. Para esquivar dichos obstáculos, conseguirá nuevas habilidades conforme avance en su aventura.

Este videojuego consta de cuatro niveles, cada uno más oscuro que el anterior, pasando de colores simpáticos y coloridos a oscuros y lúgubres. Cada nivel tiene una duración aproximada de cinco minutos.

Los objetivos de este proyecto han sido cumplidos:

- El principal se puede observar al jugar el videojuego, dispone de sus respectivos menús y de cuatro niveles diferentes.
- Dar un toque distintivo a cada nivel han faltado pequeños detalles que hubieran ayudado a cumplir en mayor medida dicho objetivo, pero aún así ha sido cumplido.
- Añadir una nueva mecánica y enemigo por nivel es igual al primer objetivo, jugando se puede ver que los cuatro niveles disponen de nuevas mecánicas y diferentes enemigos.
- Por último, el cambiar la ambientación del escenario se nota en tres de los cuatro niveles. En la cueva oscura no se nota tanta diferencia con la

cueva normal, solo se vuelve la paleta de colores un poco más oscura, pero se puede observar el cambio aunque sea ligeramente.

Al inicio se planteó seriamente hacer el juego en 2.5D, mezclando objetos de dos y tres dimensiones, pero al ver que no era viable se cambió al formato 2D.

Desarrollar este proyecto ha sido todo un reto, ha tenido sus buenos y sus malos momentos. En la parte de programación han surgido problemas casi constantemente, desde pequeños detalles como olvidarse cambiar una variable y no encontrar donde, hasta problemas más grandes como no saber programar algo y no encontrar cómo hacerlo en ninguna parte.

Los pequeños problemas tenían fácil solución, se dejaba descansar la vista un tiempo y al volver a mirar aparecía. En cambio con los más grandes, la mayoría de veces fue necesaria la ayuda del tutor del proyecto para encarar como debía programarse.

Hay bastantes cosas mejorables en este proyecto, sobre todo errores que suceden sin saber por qué, entre ellos que a veces el cocodrilo no hace el retroceso al saltar encima de un enemigo. Aún así, el resultado ha sido bastante más satisfactorio de lo esperado.

Finalizado el proyecto, hay elementos que hubiera sido ideal poder añadir, ya sea para dar más detalle o para que el jugador tuviera más contenido jugable, pero no ha sido posible por falta de tiempo.

Hubiera sido ideal poder añadir un jefe en el último nivel, para que el jugador pudiera poner realmente a prueba sus habilidades y darle algo más de tensión.

También se quería añadir iluminación, principalmente en las cuevas, para darle más realismo y que la inmersión y la tensión fueran mayores.

Con este proyecto se espera que la gente que lo lea, pueda comprender un poco mejor el trabajo que hay detrás del desarrollo de un videojuego y, en caso de probarlo, pueda notar el esfuerzo que hay detrás de cada pequeño detalle.

# Bibliografía

- [1] Antarsoft. Unity 2d platformer tutorial 19 - unity tilemaps + tile collider. <https://www.youtube.com/watch?v=iGQqVgNYxGM>. Accedido el 11-05-2022.
- [2] Brackeys. Settings menu in unity. <https://www.youtube.com/watch?v=Y0aYQrN1oYQ>. Accedido el 03-07-2022.
- [3] PriniBR. [solved] how do i fix 2d character bumping on tile collider2d edges?? <https://answers.unity.com/questions/1783640/how-do-i-fix-2d-character-bumping-on-tile-collider.html>. Accedido el 20-05-2022.
- [4] Tokio School. El diseño de interfaz de un videojuego. <https://www.tokioschool.com/noticias/diseno-interfaz-videojuego/>. Accedido el 23-08-2022.
- [5] Universia. Los lenguajes de programación más usados en la actualidad. <https://www.universia.net/es/actualidad/empleo/lenguajes-programacion-mas-usados-actualidad-1136443.html>. Accedido el 27-08-2022.
- [6] VIU. Programación de videojuegos: los lenguajes de ayer y hoy. <https://www.universidadviu.com/es/actualidad/nuestros-expertos/programacion-de-videojuegos-los-lenguajes-de-ayer-y-hoy>. Accedido el 21-08-2022.
- [7] Wikipedia. Lenguaje de programación. [https://es.wikipedia.org/wiki/Lenguaje\\_de\\_programaci%C3%B3n](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n). Accedido el 20-08-2022.
- [8] Wikipedia. Unity (motor de videojuego). [https://es.wikipedia.org/wiki/Unity\\_\(motor\\_de\\_videojuego\)](https://es.wikipedia.org/wiki/Unity_(motor_de_videojuego)). Accedido el 24-08-2022.

# **Apéndice A**

## **Game Design Document (GDD)**

### **A.1. Log**

Idea inicial de videojuego plataformas 2.5D.

Cambio a solo 2D a causa del conflicto entre objetos 2D y 3D en Unity.

### **A.2. Overview**

Este es un juego de plataformas con algunos toques y referencias al género metroidvania, en el que controlas un pequeño cocodrilo, cuyo objetivo es recoger los jamoncitos repartidos por los diferentes niveles, evitando a los enemigos y abriéndose paso entre los múltiples obstáculos. A lo largo de su aventura conseguirá nuevas habilidades que le permitirán seguir avanzando por zonas en las que antes no podía llegar o le eran imposible de atravesar. El videojuego consta de 4 niveles y cuanto más avanza a través de estos, más oscuros y retorcidos se tornan estos, una especie de descenso a la locura, en el que al inicio parece un juego para niños, simpático y colorido para terminar en una pesadilla oscura y lúgubre.

Cada uno de estos niveles anteriormente mencionados tienen una duración media de entre cuatro y cinco minutos, puesto que si fuesen más cortos apenas daría tiempo a considerarse un juego o algún tipo de reto, mientras que por el contrario, si durasen más tiempo podrían resultar tediosos e incluso aburridos.

### **A.3. Referencias del gameplay**

Como referencias para crear el gameplay de Cocodrilo Waton, nos hemos basado en los dos géneros más conocidos de los videojuegos, siendo estos el género de plataformas y el metroidvania, siendo este último un derivado del primero aunque manteniéndose igualmente como dos géneros distintos.

El género plataformas es uno de los más conocidos, con referentes como los clásicos Super Mario o Donkey Kong, en los que las mecánicas principales se centran en el movimiento, sobretodo en el salto, pues en estos juegos el jugador debe llegar desde el punto A hasta el punto B, esquivando a los enemigos y consiguiendo la mayor cantidad de recompensas y puntos.

Por otro lado, el género metroidvania, aun estar basado en el género de plataformas puesto que también se centra mucho en el movimiento, en este no se podrá acceder a todos los lugares o zonas desde el principio, ya que es necesario avanzar en el juego para conseguir habilidades que te permitirán acceder a estas, en ocasiones viéndonos obligados a volver a niveles anteriores para poder proseguir con la trama con ayuda de las habilidades adquiridas durante el transcurso del juego.

### **A.4. Gameplay**

El gameplay del videojuego se centra en la exploración de los niveles, la recolección de los jamoncitos y llegar al final del nivel para completar esa fase.

El cocodrilo puede enfrentarse a sus enemigos o esquivarlos para huir de ellos, durante el transcurso del nivel se encuentran diferentes puntos de control repartidos por el mapa que le permiten reaparecer en el punto que encontró dicho punto de control cuando este cae al vacío, pero si se quedase a cero corazones el nivel se reinicia desde el principio, tanto en el avance como en la puntuación de jamoncitos conseguidos durante la exploración.

Cabe destacar que la cámara se mantiene siempre centrada en el personaje principal, pues esta le sigue en todo momento para no perderlo de vista.

Por último, para terminar cada uno de los niveles se debe llegar al final de estos y atravesar la cueva que ahí se encuentra, lo cual nos llevará al siguiente nivel.

### **A.5. Target**

El target principal de este juego es muy diverso, dado que al ser un juego de plataformas con ligeros toques de metroidvania, con mecánicas sencillas y

divertidas es la puerta de entrada para todos aquellos que quizás nunca hayan probado un juego anteriormente o para aquellos que buscan una aventura rápida y entretenida con la que matar el tiempo.

Nuestro público objetivo se encuentra en una franja de edad entre los siete y treinta y cinco años aproximadamente, aunque cualquiera puede disfrutar de este videojuego debido a sus sencillas mecánicas.

## A.6. Unique Selling Points (USP)

- No todo es lo que parece.
- Una aventura que te sorprenderá.
- Un juego alegre y colorido, o quizás no...
- El descenso a lo más oscuro de la corrupción.
- Hazte más fuerte, avanza y vence.

## A.7. Mecánicas

- Desplazamiento lateral con las teclas A y D o las flechas laterales.
- Salto y doble salto con la tecla W o flecha hacia arriba.
- Lanzallamas con la E.
- Dash con el Shift Izquierdo.
- Bola de fuego con la Q.
- Balancearse con el Espacio.
- Caminar de las tortugas
- Volar de los murciélagos.
- Vigilancia de las arañas y los castores.
- Obtener los jamoncitos.

## A.8. Dinámicas

- Usar el doble salto para esquivar a un enemigo o llegar a un punto elevado.
- Atacar a enemigos con el lanzallamas y la bola de fuego.
- Usar el dash para pasar por huecos o impulsarse en el aire.
- Balancearse para recorrer distancia sin caerse.
- La tortuga se esconde ante el fuego.
- El murciélago ataca cuando el jugador está debajo de él.
- Las arañas persiguen al jugador.
- El castor tira proyectiles al jugador.

## A.9. Aesthetics

- Colores vivos y brillantes.
- Píxel art.
- Evolución de la estética del juego a medida que se avanza por los niveles.
- A partir de cierto nivel se torna oscura y retorcida.

## A.10. Finalidad

La finalidad de este título es la de que el jugador se aprenda las diferentes mecánicas que aparecen durante el transcurso de los diferentes niveles y que este sea capaz de usarlas en el momento oportuno para poder seguir avanzando en su aventura. De esta forma conseguirá un gameplay fluido y entretenido en todo momento.

## A.11. Objetivos

El objetivo principal del jugador es completar el nivel actual para desbloquear el siguiente, solamente consiguiendo esta hazaña si es capaz de llegar al final del nivel, donde le espera un gran arco de rocas, al atravesarlo se da por

finalizado el nivel y se desbloquea el siguiente. Como objetivo secundario, el jugador puede intentar obtener todos los jamoncitos y lograr una puntuación perfecta en todos los niveles.

## A.12. Blueprint

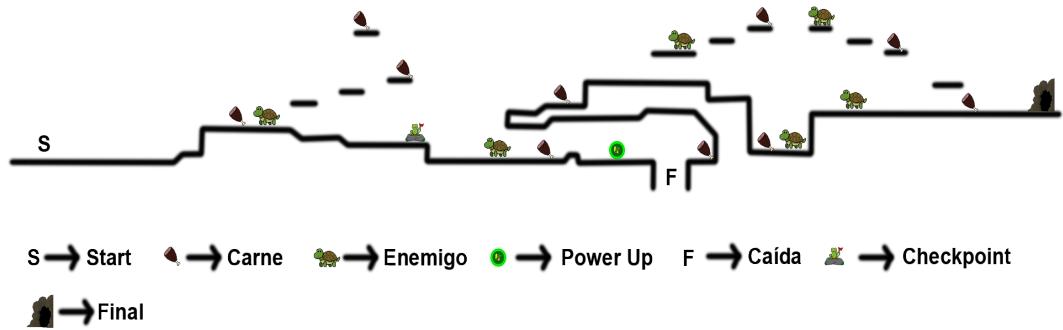


Figura A.1: Pantano

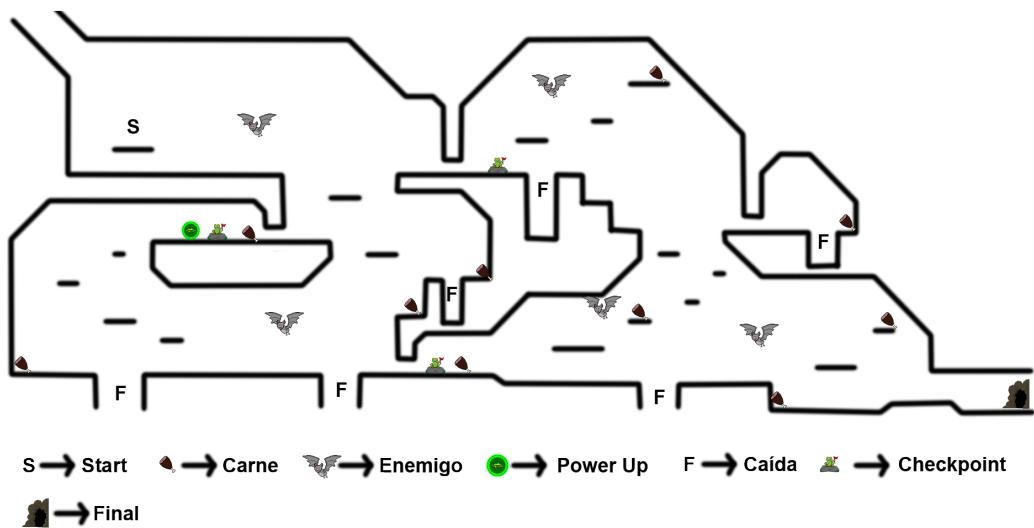


Figura A.2: Cueva

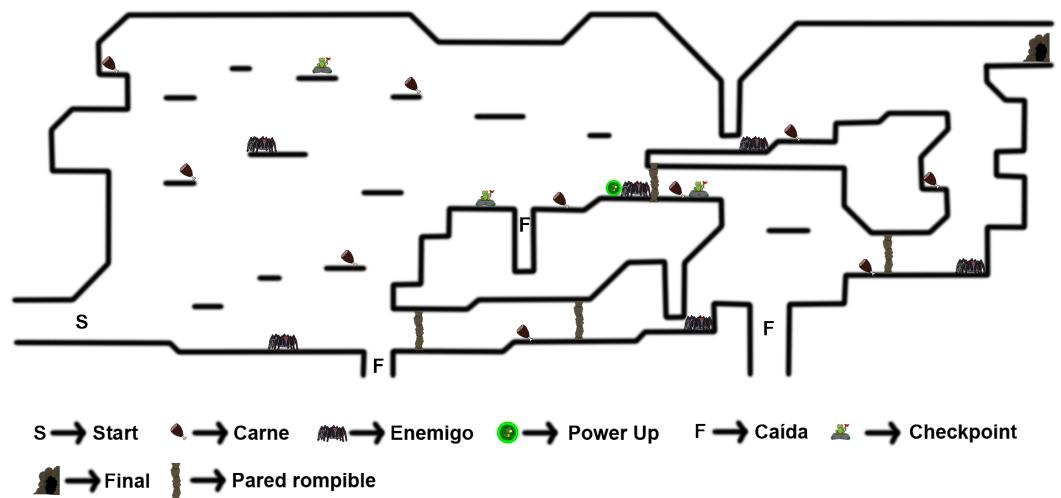


Figura A.3: Cueva oscura

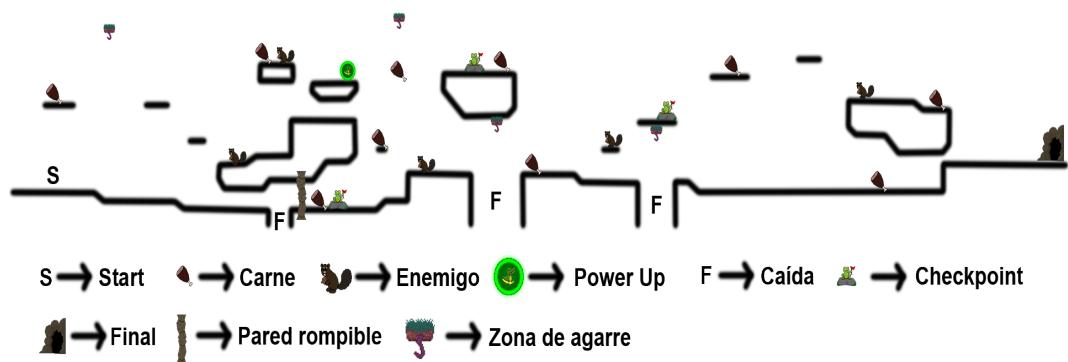


Figura A.4: Pantano oscuro

## A.13. Referencias de arte

Como referencias de arte podemos encontrar principalmente juegos de la vieja escuela, como “Super Mario” o el “Castlevania”, juegos los cuales también pertenecen a los géneros que se han utilizado como referencia para llevar a cabo el juego, siendo los géneros de plataformas y “metroidvania” concretamente.

El arte está totalmente basado en pixel art, puesto que queríamos representar una estética de videojuego de 8 o 16 bits, además de usar colores vivos y coloridos en el primer nivel, oscuros para los dos niveles de cueva y, por último, tonalidades violáceas y mortecinas en el último de los niveles. De esta forma conseguimos un claro contraste entre cada uno de los niveles, además de crear un gran contraste en el feeling del juego, puesto que uno de los escenarios es muy amigable, otros más oscuros y tensos, y el último de estos siendo tétrico y frío.

Los personajes siguen las líneas generales de la estética del videojuego, puesto que el cocodrilo y la tortuga son de aspecto alegre y desenfadado, con colores vivos y brillantes.

Por otro lado, encontramos los enemigos de las cuevas, cuyos colores son más neutros y apagados, aunque en el caso concreto de la araña estos denotan peligro en ellos, puesto que este enemigo es más peligroso que el murciélago. Por último, con el castor encontramos un diseño sencillo, un castor al uso, que está pensado para que el jugador se acerque confiado y se sorprenda al ver que este es malvado y le lanza troncos para acabar con él.

## A.14. Personaje principal

Este videojuego consta de un único personaje jugable disponible desde el primer momento, un cocodrilo pequeño y regordete que camina a dos patas, llamado Cocodrilo Waton. Este sigue la estética pixel art del videojuego y su objetivo es obtener el máximo posible de jamoncitos mientras llega al final del nivel.

Las habilidades de las que dispone son bastante variadas, empezando con desplazamiento lateral, salto y un lanzallamas y más adelante durante el transcurso de su aventura desbloqueando otras gracias a los power ups que encuentra por los niveles, como el doble salto, el dash, la bola de fuego y un gancho para balancearse.

El cocodrilo dispone de la capacidad de dañar a sus enemigos saltando encima de estos a la par que recibir daño en caso de entrar en contacto o ser alcanzado por sus proyectiles.

Tiene cinco corazones de vida y pierde uno de ellos cada vez que es golpeado hasta quedar a cero, momento en que cae derrotado y el jugador debe volver a empezar el nivel.

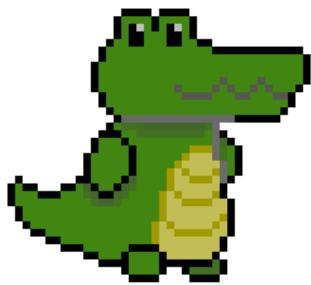


Figura A.5: Cocodrilo Waton

## A.15. Enemigos

### A.15.1. Tortuga

Las tortugas son el primer enemigo que encontrará el jugador a la vez que el más simple de ellos, ya que su único cometido es ir de un punto a otro, sin atacar de forma directa al cocodrilo. Aún así, si el personaje entra en contacto con la tortuga, este recibirá daño a causa de ello. Si la tortuga está mirando hacia el cocodrilo y este usa el lanzallamas o dispara una bola de fuego, se esconderá en su caparazón para no recibir daño.

### A.15.2. Murciélagos

Los murciélagos aparecen en el segundo nivel, con un funcionamiento similar al de las tortugas, con la diferencia de que estos vuelan y atacan al jugador si este se encuentra debajo de ellos. Una vez hayan atacado al cocodrilo, reanudarán su vuelo hasta que pueda repetir el ataque.

### A.15.3. Araña

Las arañas aparecen en el tercer nivel, son completamente diferente a los dos previamente mencionados, puesto que en este caso es mucho más agresivo respecto al jugador.

La araña hostiga al jugador con sus ataques sin detenerse, se abalanza por sorpresa y a gran velocidad sobre él, para dañarlo con el impacto.

#### **A.15.4. Castor**

El castor es el enemigo que aparece en el cuarto nivel, este no guarda ninguna similitud con los anteriores, puesto que es el único de estos que ataca a distancia, lanzando troncos sin descanso al jugador para acabar con él.

### **A.16. Power ups**

#### **A.16.1. Doble salto**

Te permite saltar por segunda vez en el aire, este efecto no podrá ser repetido una vez se haya realizado hasta que el cocodrilo vuelva a posarse sobre el suelo.

#### **A.16.2. Dash**

Te permite lanzarte hacia la dirección en la que se está mirando, avanzando rápidamente un pequeño espacio de tiempo, lo que ayuda a llegar más lejos tras un salto y pasar por zonas estrechas que de otra forma sería imposible.

#### **A.16.3. Bola de fuego**

Esta habilidad permite lanzar grandes bolas de fuego que permiten destruir muros concretos, para así poder abrir el camino, o atacar a los enemigos, siendo este un ataque de gran poder.

#### **A.16.4. Gancho**

Esta habilidad te permite lanzar un gancho a puntos concretos del nivel, que da la posibilidad de balancearse y así llegar a lugares muy lejanos, que no es posible alcanzar solamente con el doble salto o el das.

### **A.17. Cutscenes**

Cuando el jugador llega al final del nivel y está cerca de la cueva, en ese momento se lanza un cutscene en el que podemos ver al cocodrilo adentrándose en ella, representando así que pasa hacia el siguiente nivel.

## **A.18. Assets**

Los assets que encontramos en este juego están compuestos por las diferentes capas que forman los fondos de cada uno de los niveles, en los que dependiendo de a cual de estos pertenezcan podemos encontrar agua, vegetación, rocas y árboles, pasando por stalactitas, stalagmitas y columnas en los niveles ambientados en cuevas hasta llegar al último nivel, el cual es una versión corrupta del primero, en el que la vegetación tiene colores apagados, el agua se encuentra contaminada y los árboles ahora retorcidos y malditos muestran rostros siniestros en sus troncos.

Por otro lado, encontramos el sonido ambiental, el cual cambia en cada uno de los niveles, siendo melodías cortas y repetitivas, encontrando que en el primer nivel es rápida y animada mientras que en las cuevas es lúgubre, siendo más oscura y lenta cuanto más te adentras, mientras que en el último nivel, para coincidir con la estética del mismo, la melodía se encuentra corrupta, ralentizada y reverberada para convertir la musiquilla alegre en retorcida y decadente.