

# Trabajo Unity

<b>Introducción</b>	<b>2</b>
<b>Nivel 1</b>	<b>3</b>
Creación del escenario de juego	3
Creación del jugador	13
Creación de los pinchos	30
<b>Nivel 2</b>	<b>43</b>
<b>Nivel 3</b>	<b>53</b>
<b>Nivel 4</b>	<b>61</b>
<b>Nivel 5</b>	<b>66</b>
<b>Aspectos adicionales</b>	<b>67</b>
Creación de la música de fondo del juego	67
Creación de un contador de muertes	68
Corrección de la mecánica de muertes del juego	74

# Introducción

En primer lugar, voy a realizar una breve descripción del juego que voy a desarrollar para la práctica entregable de Unity.

El juego que voy a desarrollar será en 2D un juego de plataforma donde tendremos un personaje con sus correspondientes animaciones y el objetivo de este personaje es avanzar por los distintos niveles que tiene el juego. Cada nivel es un conjunto de plataformas y obstáculos que el personaje debe atravesar hasta una puerta que le llevará al siguiente nivel. Cada nivel contará con un grado de dificultad mayor ya que el número de obstáculos aumentará y así como la complejidad de los distintos obstáculos.

Cada vez que nuestro personaje muera por causa de algún obstáculo o enemigo, nuestro pesaje volverá a aparecer en la puerta de inicio del nivel correspondiente. El objetivo del juego es conseguir que nuestro personaje supere todos los niveles con el mínimo número de muertes posible.

Nota: Aunque el desarrollo de este juego está solamente realizado por mi (el trabajo es individual), a lo largo del todo el documento hablaré en 1º persona del plural para hacer referencia a las acciones que voy realizando a lo largo del desarrollo del juego.

# Nivel 1

## Creación del escenario de juego

Lo primero que vamos a hacer para desarrollar nuestro juego, es realizar el diseño de los distintos niveles que conformarán dicho juego. Para ello, vamos a utilizar un Asset gratuito de la Asset Store de Unity llamado ***Medieval pixel art asset FREE*** el cual se encuentra disponible para su descarga gratuita.

De esta manera, vamos a ir a la pestaña de “Window” en la barra de herramientas superior y vamos a seleccionar la opción de “Asset Store”. Una vez hecho esto, se nos desplegará en el editor de Unity una pestaña como la siguiente (Figura 1):

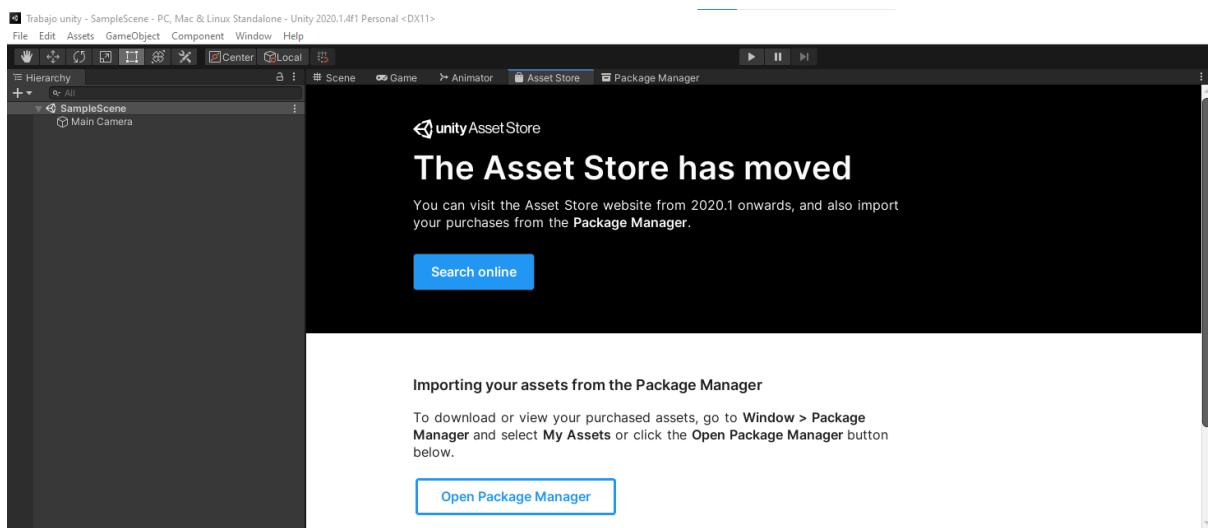


Figura 1. **Asset Store**

De esta manera, seleccionamos el botón de Search Online para buscar en la tienda de Asset de Unity el asset que queremos importar. Una vez hemos encontrado el Asset que queremos importar (Figura 2), haremos click en el botón de “Add to My Assets”.

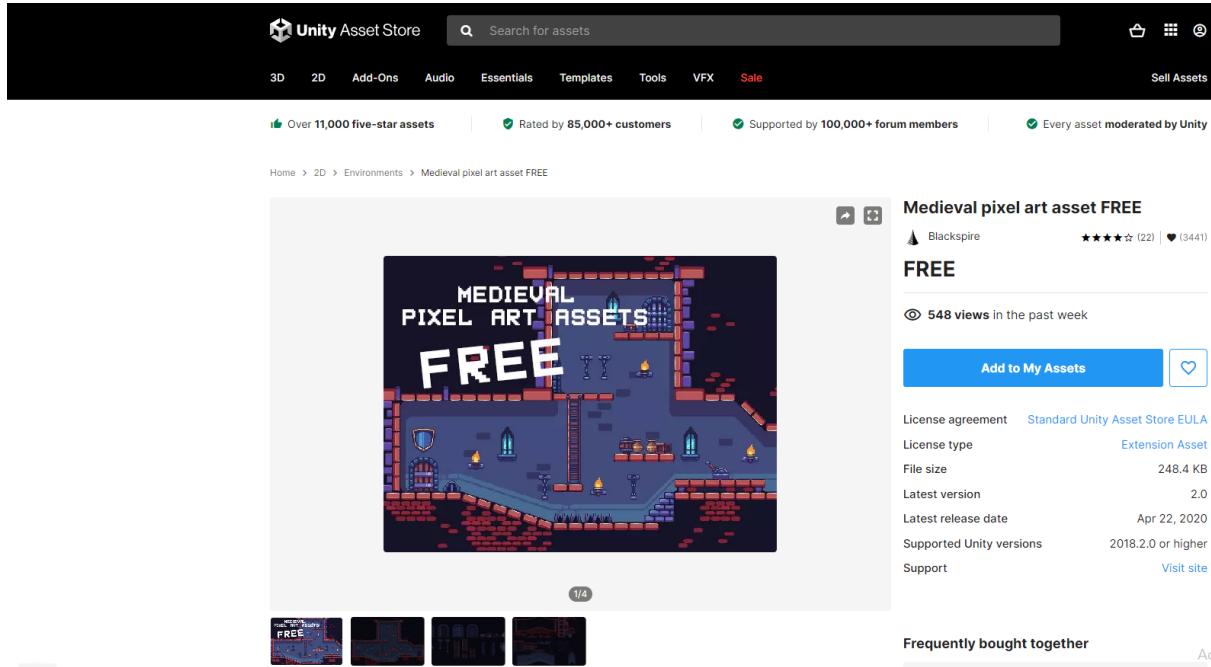


Figura 2. Medieval pixel art asset FREE

Una vez hecho esto, se nos abrirá una nueva pestaña (Package Manager) en nuestro editor de Unity (Figura 3) que nos permitirá en primer lugar descargar el contenido del Asset y posteriormente importarlo a nuestro proyecto.

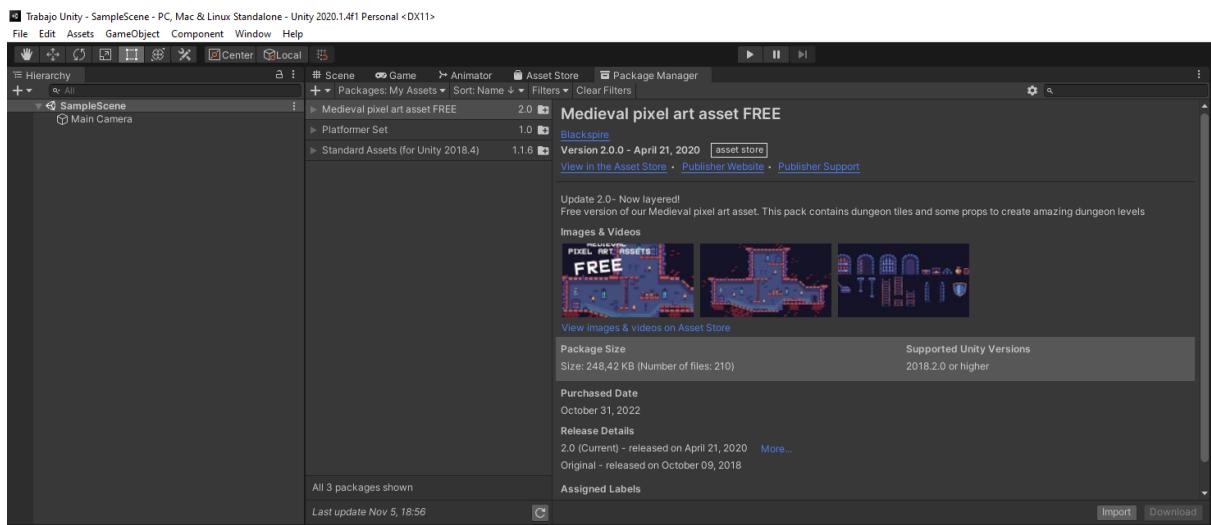


Figura 3. Package Manager

Una vez hemos descargado e importado a nuestro proyecto correctamente el contenido del Asset tendremos a disposición todos los recursos que contiene este Medieval Asset.

Para crear la infraestructura del nivel por decirlo así lo primero que vamos a realizar es crear un gameobject de tipo tilemap (mapa de mosaicos), haciendo click derecho en la barra izquierda en la opción 2D object→Tilemap. Con esta opción se podrá editar el escenario como si fuera un pincel. En nuestro caso, vamos a llamar al Tileset recién creado “Suelo”.

Una vez hemos creado el tilemap, para poder editar el contenido de este tilemap el siguiente paso es crear una paleta, y para ello se abre la ventana window→2d→Tile Palette.

Sin embargo, este Asset ya cuenta con una paleta ya creada y que se encuentra dentro de la carpeta “palettes”. De esta manera, al acceder a la ventana de Tile Palette ya tendremos cargada por defecto la paleta que nos incluye el propio Asset.

De esta manera, al acceder a ventana de Tile Palette tendremos lo siguiente (Figura 4):

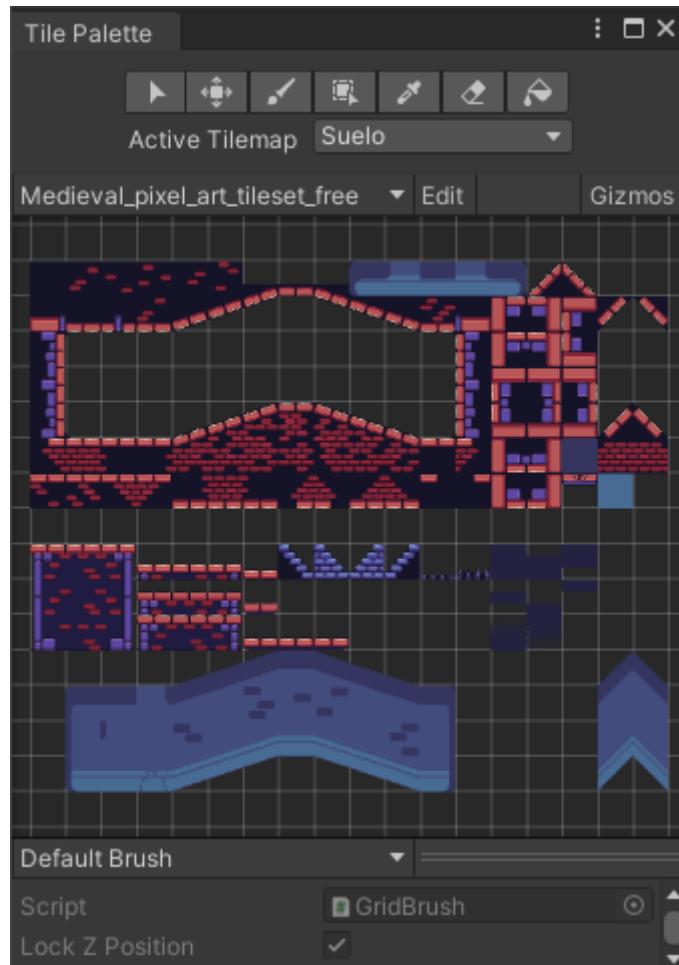


Figura 4. Paleta ya incluida con el asset

Ahora, para dibujar en el escenario como si una brocha se tratase seleccionaremos en la barra de herramientas la brocha (botón con un pincel encima) y seleccionamos dentro de la paleta la sección de imagen que deseamos “pintar” en el escenario. Una vez seleccionamos una cuadrícula, podremos dibujar esta cuadrícula en el escenario las veces que queramos.

Sin embargo, con un solo Tilemap no vamos a poder realizar un correcto diseño del escenario del nivel ya que el fondo y el suelo se van a encontrar en el mismo nivel y el personaje no va a poder moverse dentro del escenario del nivel. Para ello vamos a crear una nuevo Tilemap llamado Fondo en el que se va a dibujar el fondo del escenario del juego. De esta manera, volvemos a crear otro Tilemap llamado Fondo haciendo click derecho en la barra izquierda en la opción 2D object→Tilemap.

Al crear este segundo tilemap, se creará dentro de otro Grid distinto. Como en nuestro caso, ya teníamos otro Grid creado cuando creamos el Tilemap “Suelo” lo que vamos a hacer es mover el Tilemap “Fondo” al Grid en el que se encuentra el Tilemap “Suelo” y eliminar este segundo Grid ya que se encontrará vacío y no lo usaremos para nada.

Nos seguir estos pasos nos debería quedar algo como lo siguiente (Figura 5):

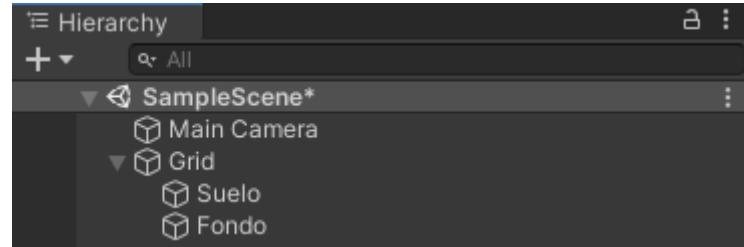


Figura 5. Agrupación de los 2 tilemap en un solo grid

Una vez ya hemos creado los tilemaps de Suelo y de Fondo, el siguiente paso es definir su orden de estos Tilemaps en la escena. En nuestro caso, el Tilemap “Suelo” nos interesa que se “dibuje” por delante del Tileset “Fondo”. De esta manera, seleccionamos el Tileset “Suelo” y en el componente “Tilemap Renderer” buscamos la propiedad “Order in Layer” y le asignamos un valor de 1 (Figura 6) mientras que en el Tileset “Fondo” le asignamos un valor de 0 (Figura 7). De esta manera, la capa del suelo se representará “por delante” de la capa del fondo.

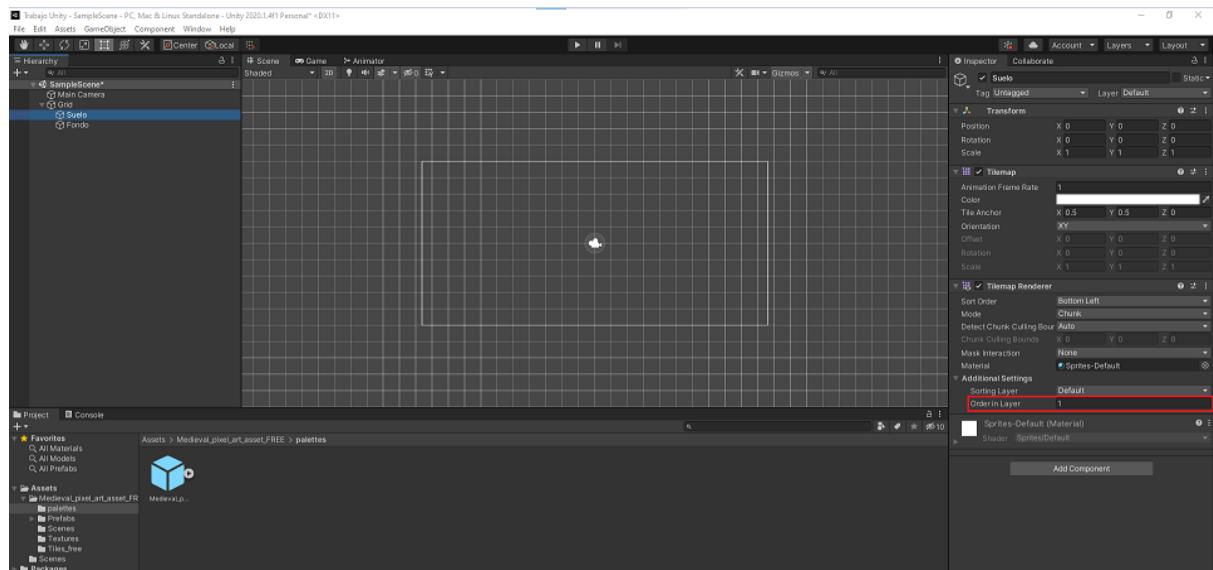


Figura 6. Asignación del orden al tilemap Suelo

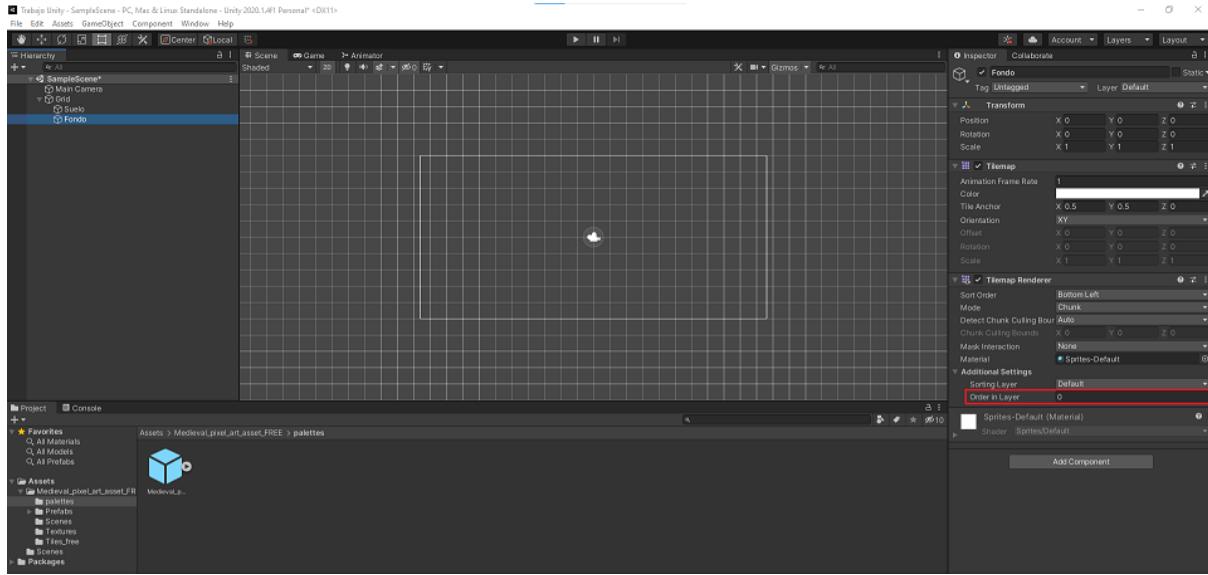


Figura 7. Asignación del orden al tilemap Fondo

Ahora, una vez ya hemos configurado los Tilemap correctamente, el siguiente paso es utilizar la paleta que nos proporciona el Asset que hemos importado y dibujar tanto el suelo como el fondo de los distintos niveles. Hay que tener cuidado de seleccionar en cada momento en cual de los dos Tilemap se desea dibujar ya que si no podemos dibujar sin querer partes del suelo en el fondo y viceversa.

En el juego el diseño de los distintos niveles va a ser incremental. Esto quiere decir que cada nivel va a contar con la estructura básica del nivel anterior y además se le van a añadir un conjunto de componentes nuevos. De esta manera, en primer lugar vamos a diseñar el escenario del primer nivel y luego utilizaremos esta estructura del primer nivel para ir desarrollando los siguientes niveles.

En nuestro caso, el infraestructura del primer nivel va a quedar de la siguiente manera (Figura 8):

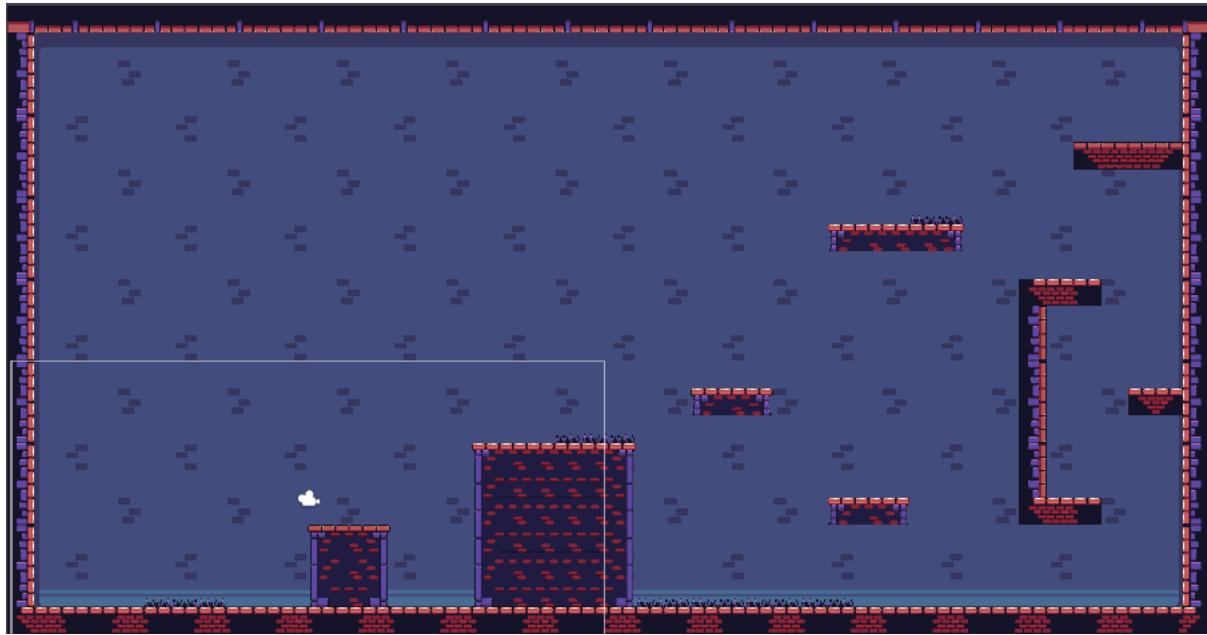


Figura 8. Estructura del nivel 1 (sin accesorios)

Una vez hecho el escenario con el que va a interactuar el personaje, vamos a añadir los colliders al suelo para que el personaje pueda caminar sobre las distintas plataformas del suelo (Figura 9). Para ello sólo hay que seleccionar el componente Tilemap→Tilemap collider 2D y se puede ver de manera automática que se crean los colliders para toda la capa del suelo (Figura 10).

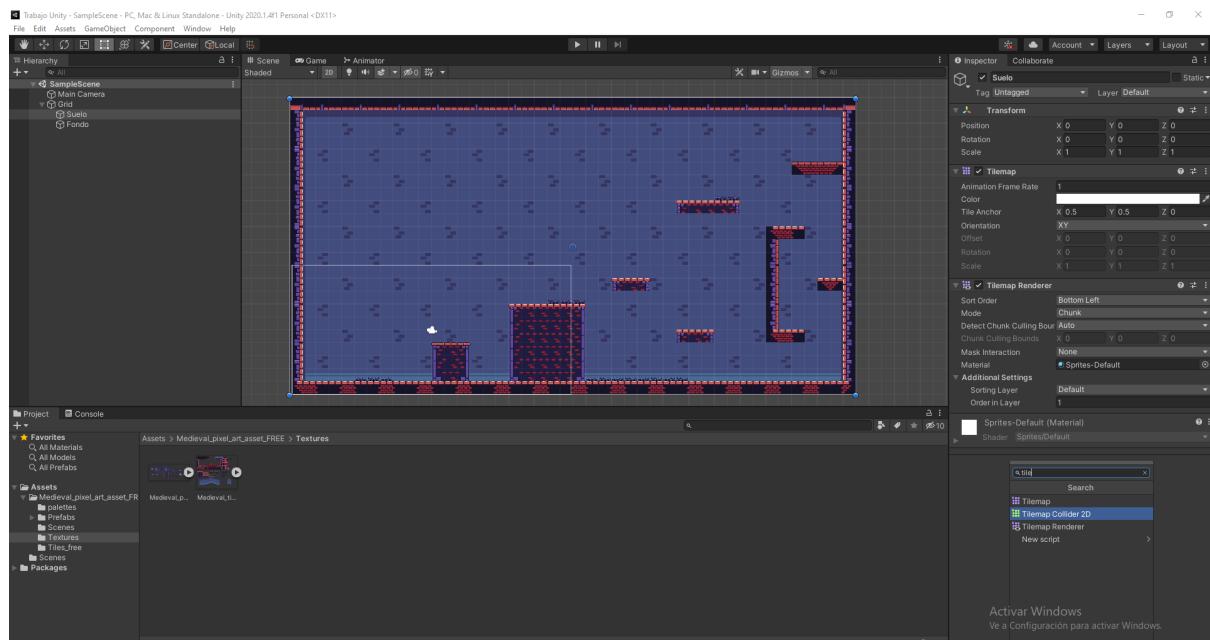


Figura 9. Tilemap Collider 2D

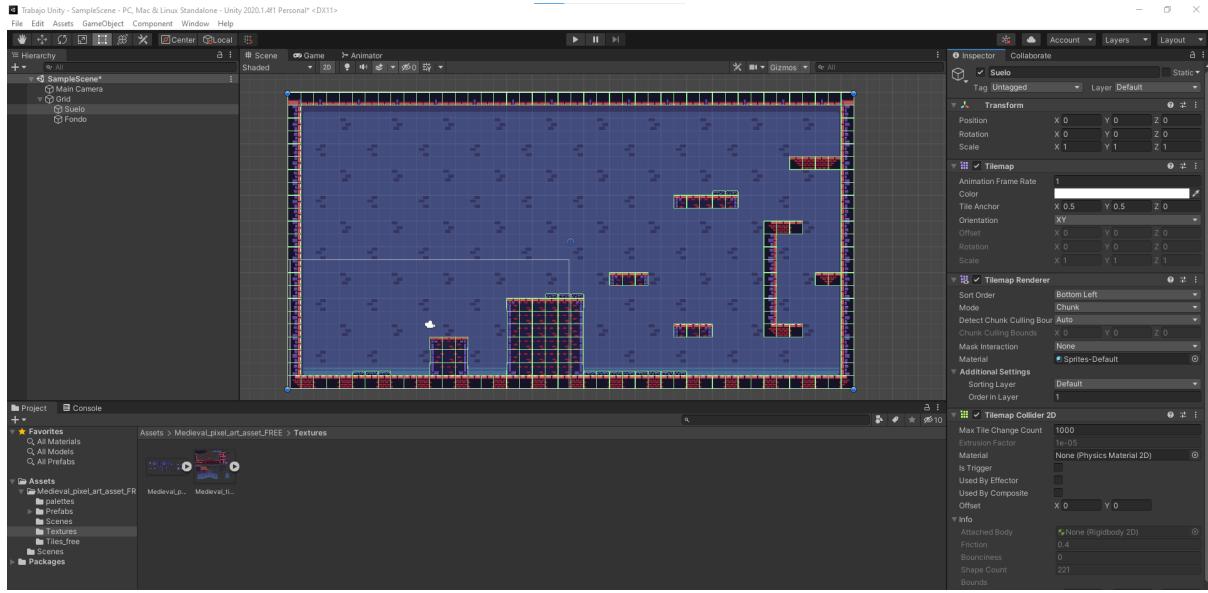


Figura 10. **Tilemap Collider 2D**

Sin embargo, aún nos queda un pequeño detalle por hacer para terminar con la infraestructura del primer nivel. Este detalle consiste en añadir 2 puertas que indiquen el principio y el final del nivel. La puerta de inicio, es el lugar donde aparecerá nuestro personaje y la puerta de final es la puerta a la que debe ir el personaje para completar el nivel.

Dentro de nuestro Asset, tenemos una carpeta llamada “Textures” que en su interior tiene 2 conjuntos de Tileset. El primer Tileset ya lo utilizamos para dibujar el escenario con la paleta que ya venía creada por defecto en el Asset (carpeta palettes) mientras que el segundo Tileset (Figura 11) tiene un conjunto de elementos adicionales decorativos (puertas, escaleras, palancas, etc).

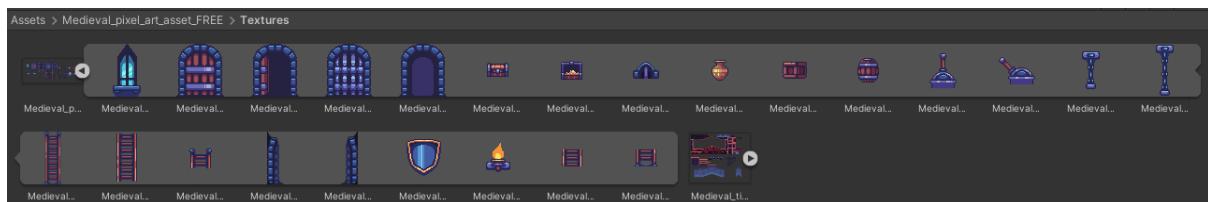


Figura 11. **Tileset elementos decorativos**

Con este conjunto de objetos decorativos podemos crear una nueva paleta para poder dibujar en el escenario cualquiera de estos objetos decorativos como nosotros queramos. Para crear una paleta, volvemos a abrir la ventana que nos permite gestionar las paletas (window→2d→Tile Palette) y en vez de usar la paleta ya existente, seleccionamos la opción de crear una nueva paleta (Figura 12).

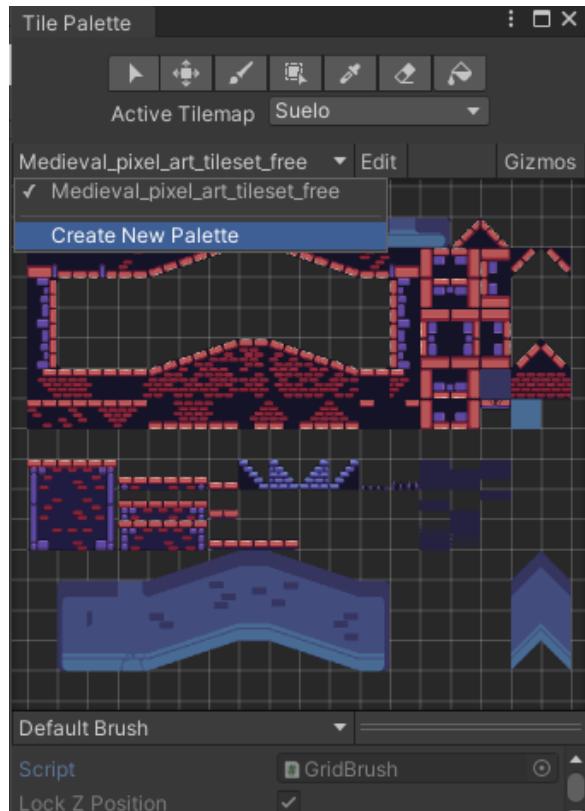


Figura 12. Creación de una nueva paleta

Al hacer esto se nos desplegará una pestaña, donde podemos configurar distintos parámetros como el nombre de la paleta (que en nuestro caso será Accesorios) el tipo de la celda o el tamaño de la celda (Figura 13). Nosotros cambiaremos el nombre y dejaremos el resto de parámetros por defecto.

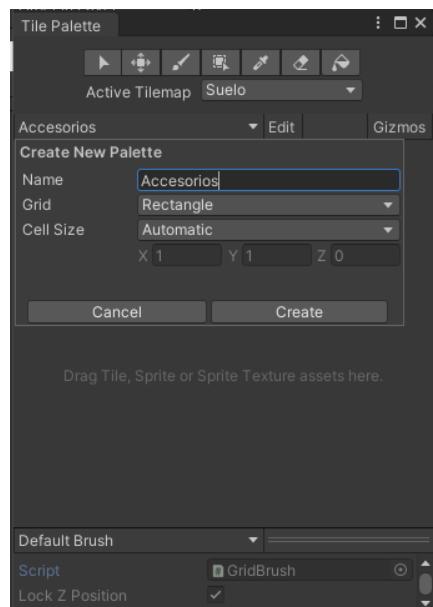


Figura 13. Creación de una nueva paleta

Al hacer esto ya tendremos creada nuestra nueva paleta “Accesorios” (Figura 14). Ya lo único que nos queda es arrastrar el contenido del Tileset que contiene los accesorios a la paleta y ya podremos dibujar los distintos accesorios decorativos en la escena con esta paleta.

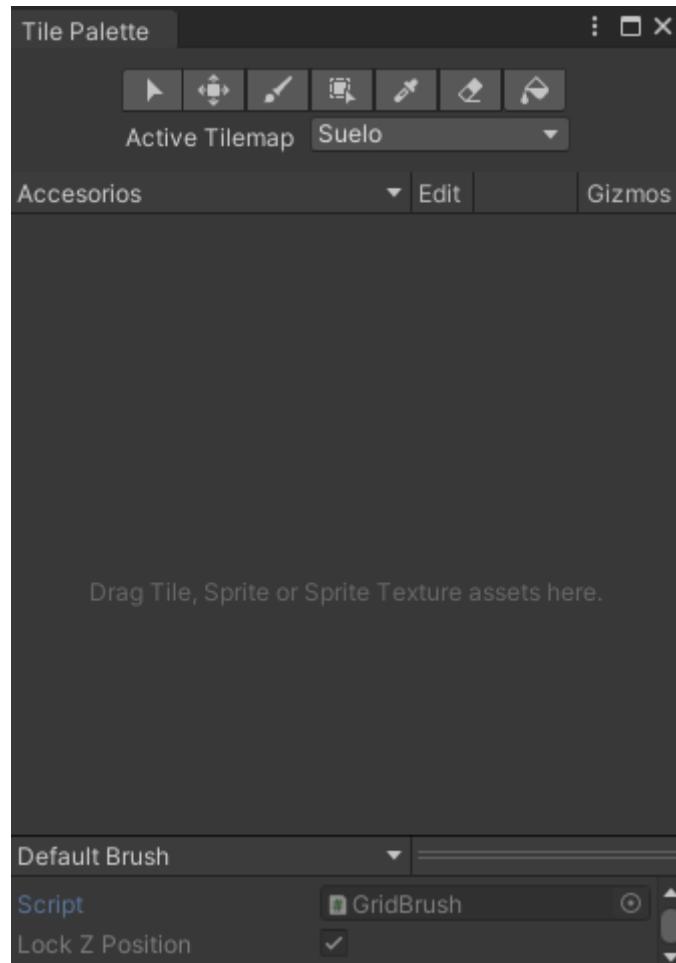


Figura 14. Paleta de los accesorios

Al arrastrar el contenido del Tilemap que contiene los accesorios decorativos a la paleta, no quedará lo siguiente (Figura 15). Aunque visualmente en la paleta pueda parecer que algunos objetos se superponen con otros, a la hora de seleccionar una cuadrícula para dibujar un objeto podemos dibujarlo sin ningún tipo de superposición.

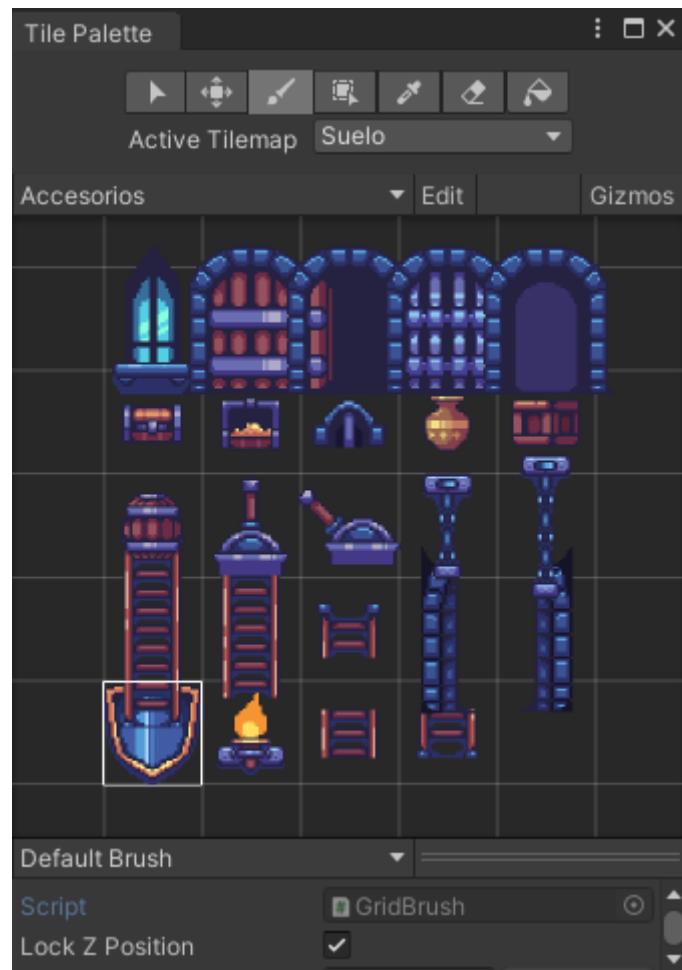


Figura 15. Paleta de los accesorios

Ahora, lo único que nos queda es decorar la escena de este primer nivel con distintos elementos decorativos como pueden ser puertas, antorchas o escudos. El resultado es el siguiente (Figura 16):

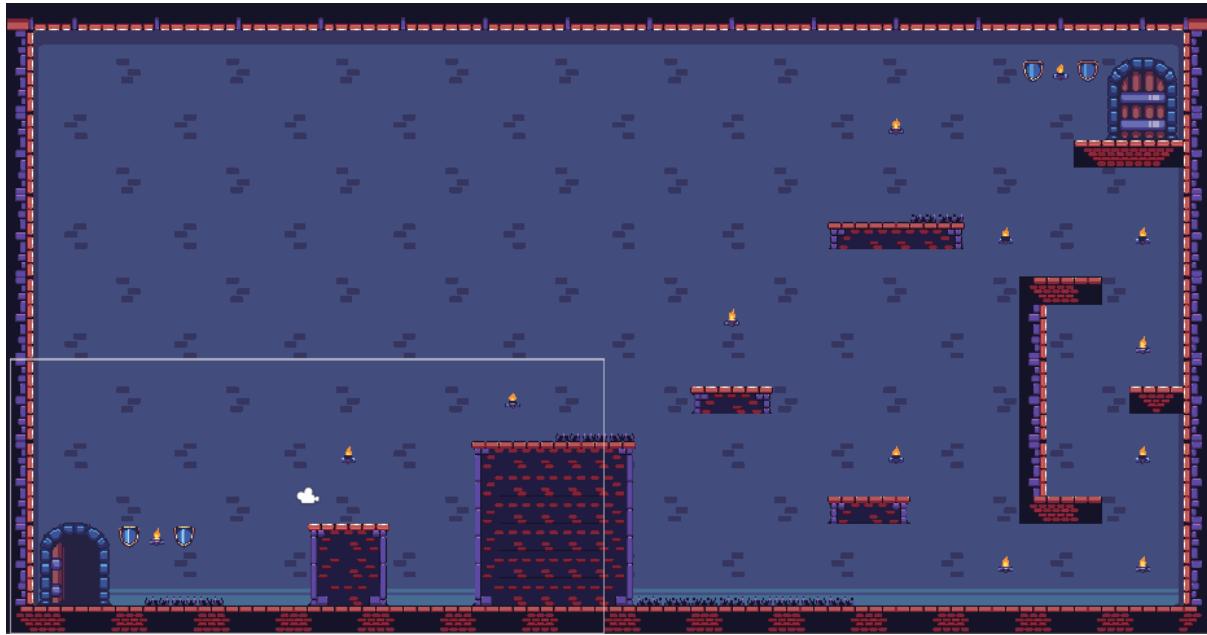


Figura 16. Estructura del nivel 1 con accesorios

A modo de curiosidad, en la escena se puede observar como las antorchas que hemos colocado van señalando de alguna manera el recorrido que tiene que seguir el personaje para llegar a la puerta que le permita acceder al siguiente nivel.

## Creación del jugador

Una vez ya hemos diseñado el escenario del nivel, el siguiente paso será crear nuestro jugador, realizar las animaciones correspondientes a este jugador y realizar la máquina de estados del jugador. Para ello vamos a utilizar un Asset llamado “Medieval King Pack 2” (Figura 17) el cual contiene distintas animaciones de un personaje Rey.

Figura 17. Medieval King Pack 2

De esta manera, vamos a descargar e importar este Asset a nuestro proyecto siguiendo los mismos pasos que hicimos con “Medieval pixel art asset FREE”. Una vez hayamos importado el Asset en nuestro proyecto, podemos observar que hay 3 carpetas en este Asset. La primera carpeta contiene animaciones, la segunda carpeta contiene una escena de prueba y la tercera carpeta contiene sprites (imágenes de las distintas animaciones).

Para crear nuestro jugador, seleccionamos la opción de Create2D→ObjectSprite, cambiamos el nombre del Sprite para que se llame “Jugador” y le asociamos una imagen a este Sprite. En nuestro caso, vamos a asociar el primer Sprite de la animación de Idle a este Sprite “Jugador”. El resultado de este proceso se muestra en la siguiente imagen (Figura 18):

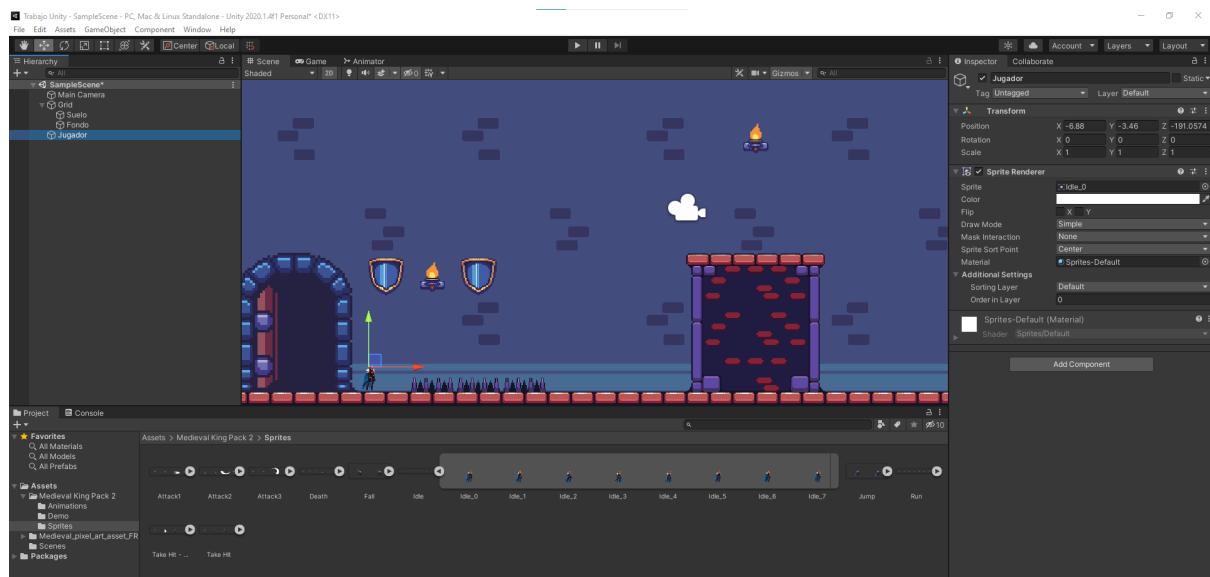


Figura 18. Sprite Jugador

Como se puede observar en la imagen anterior (Figura 18), nuestro personaje es pequeño en relación con el resto de objetos de la escena. Sin embargo, esto se puede arreglar fácilmente cambiando la escala del personaje a través de la propiedad Scale del componente Transform. Tras haber realizado estos ajustes en la escala del jugador nos queda lo siguiente (Figura 19):

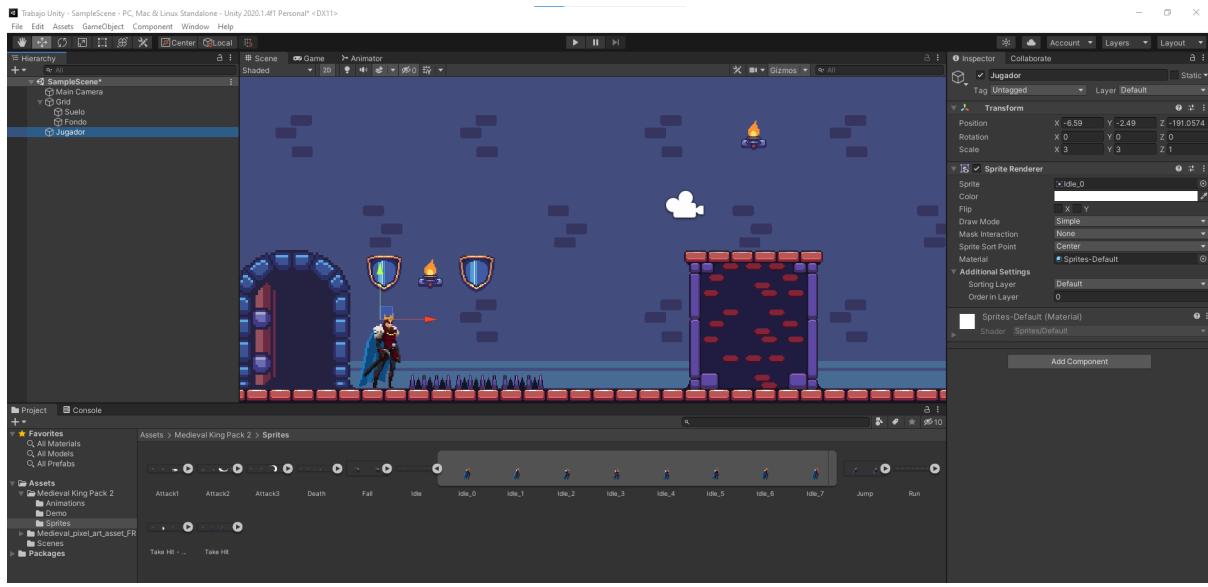


Figura 19. Sprite Jugador

Una vez tenemos ya dispuesto nuestro jugador en la escena el siguiente paso es crear las animaciones del personaje. Como hemos dicho antes, el Asset “Medieval King Pack 2” contaba con una carpeta de animaciones donde se encuentran las distintas animaciones de Idle, Run, Jump, Fall, etc ya realizadas (Figura 20) por lo que este paso no lo podemos ahorrar. Sin embargo, cada animación tiene asociado su propio Animator Controller por lo que debemos crear un nuevo Animator Controller que nos permita combinar todas las transiciones entre las distintas animaciones.



Figura 20. Animaciones

De esta manera, vamos a crear un nuevo Animator Controller llamado “Jugador” en esta misma carpeta de animaciones (Create→Animator Controller) y vamos a asociar este Animator Controller al jugador. Para ello, seleccionamos al Jugador y añadimos un nuevo componente llamado Animator. Una vez hecho esto, arrastramos nuestro Animator Controller “Jugador” a la propiedad Controller del componente Animator (Figura 21). Con esto ya tenemos asociado el Animator Controller que gestiona todos los cambios en las animaciones con el jugador.

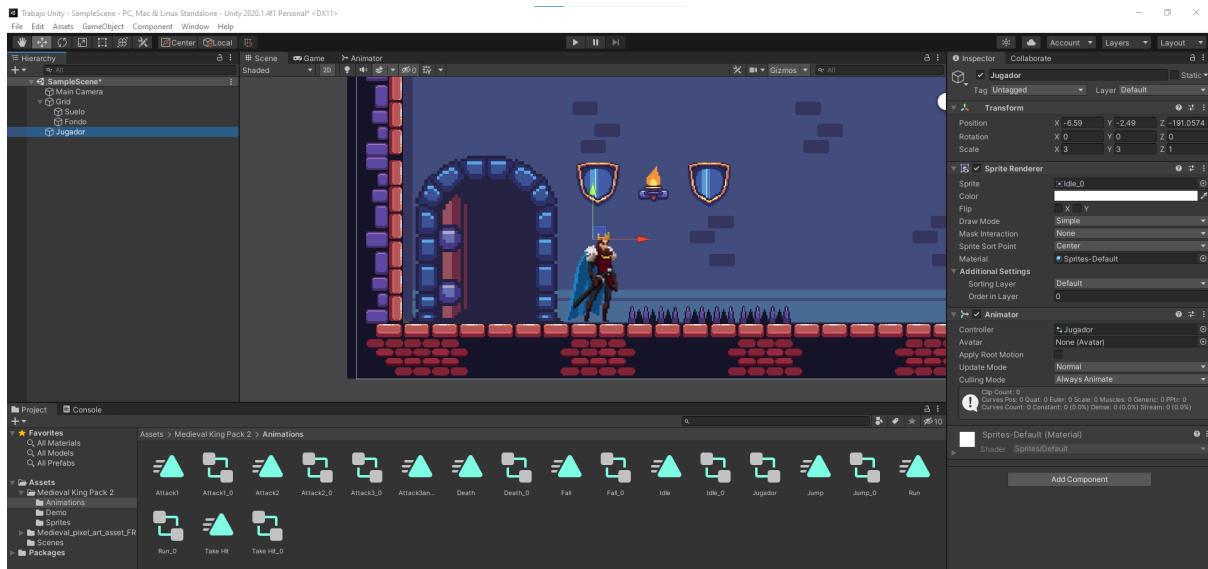


Figura 21. Asociando el Animator Controller “Jugador” al jugador

Ahora vamos a crear la máquina de estados que defina las animaciones del Animator Controller “Jugador” y las transiciones entre dichas animaciones. De esta manera, vamos a arrastrar las animaciones que nos interesen en el Animator Controller y vamos a crear una serie de parámetros booleanos que serán utilizados para definir las transiciones entre animaciones.

En nuestro juego, queremos que el jugador tenga las animaciones de Idle, Run, Jump, Fall y Death. Por lo tanto arrastramos estas 5 animaciones al Animator Controller (Figura 22) de forma que haya una transición desde el estado de entrada y la animación de Idle, que es la primera animación que queremos que se ejecute cuando arranque el juego.

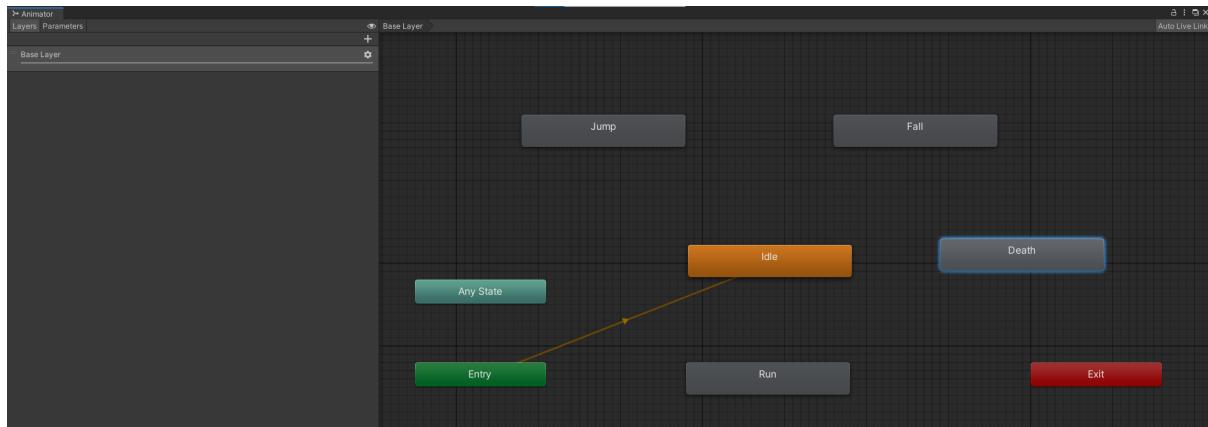


Figura 22. Animator Controller con los estados

Antes de realizar el resto de transiciones entre los distintos estados, vamos a definir una serie de parámetros (en su mayoría booleanos) en la sección de parámetros para que así se produzcan las transiciones cuando un parámetro tome un determinado valor. En nuestro caso, vamos a definir 3 parámetros booleanos llamados “caer”, “correr” y “saltar” y un trigger llamado “caerTierra” (Figura 23).

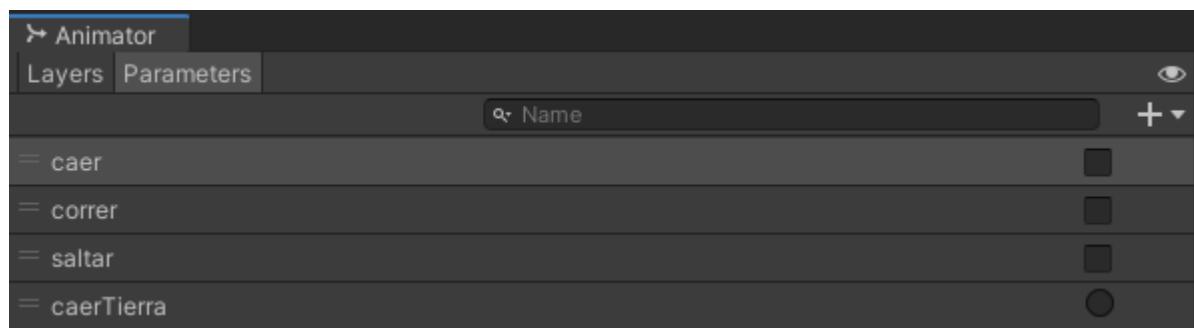


Figura 23. Lista de parámetros del Animator Controller

Una vez hecho esto vamos a crear las distintas transiciones. Para crear una transición simplemente hay que hacer click derecho en el estado donde se desea iniciar una transición y seleccionar la opción “Make transition”. Una vez se tiene una transición creada, se selecciona dicha transición y en el apartado de Conditions añadir las condiciones que se consideren necesarias (Figura 24). Estas condiciones van a trabajar con los parámetros que hemos creado anteriormente.

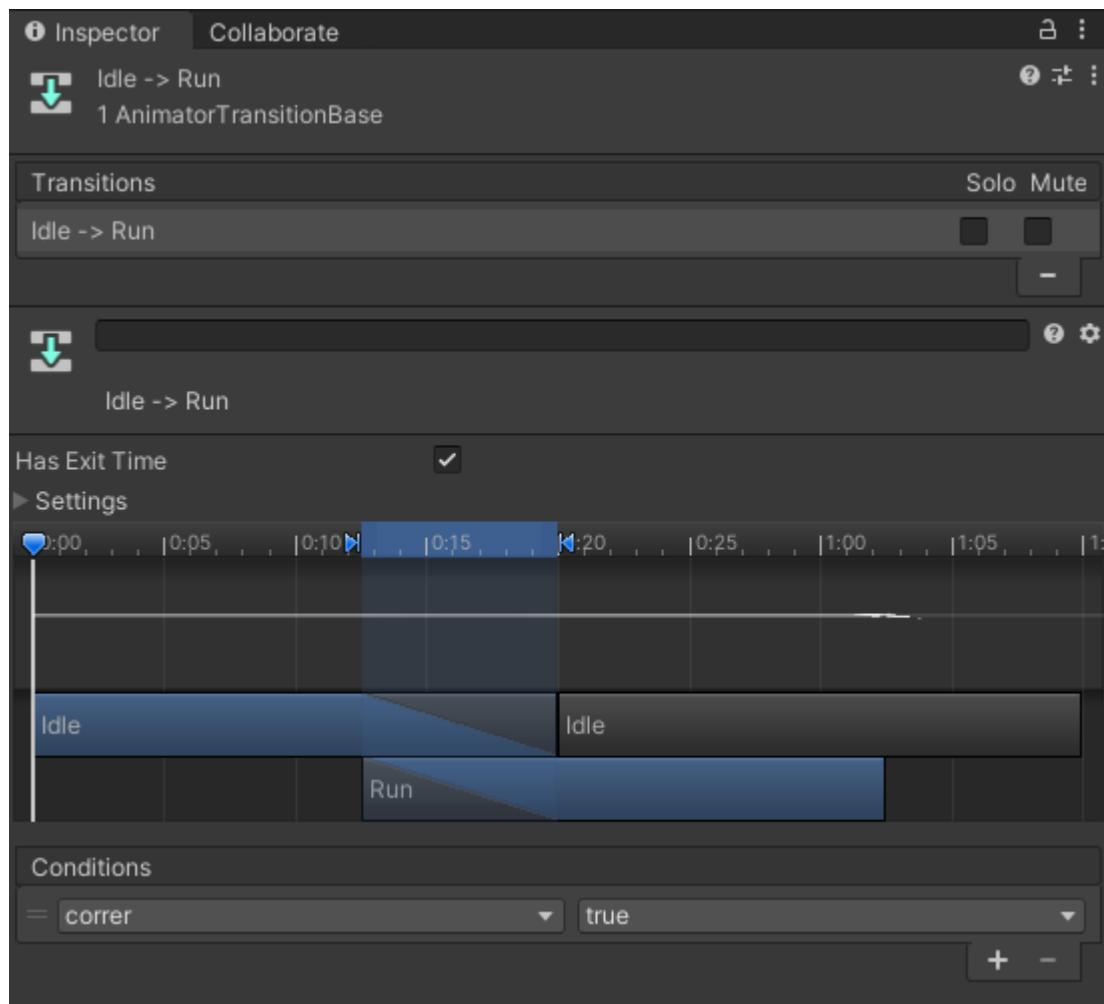


Figura 24. Ejemplo de transición del Animator Controller

En nuestro caso, vamos a crear las siguientes transiciones:

- Una transición desde el estado “Idle” hasta el estado “Run” que se produzca cuando la variable booleana correr sea cierta.
- Una transición desde el estado “Run” hasta el estado “Idle” que se produzca cuando la variable booleana correr sea falsa.
- Una transición desde el estado “Idle” hasta el estado “Jump” que se produzca cuando la variable booleana saltar sea cierta.
- Una transición desde el estado “Run” hasta el estado “Jump” que se produzca cuando la variable booleana saltar sea cierta.
- Una transición desde el estado “Jump” hasta el estado “Fall” que se produzca cuando la variable booleana caer sea cierta.
- Una transición desde el estado “Fall” hasta el estado “Idle” que se produzca cuando active el disparador caerTierra.
- Una transición desde el estado “Any State” hasta el estado “Death” que se produzca en cualquier momento (no tiene ningún tipo de condición).

Una vez hemos creado todas las transiciones que hemos indicado y hemos añadidos las condiciones necesarias a cada transición, tendremos una máquina de estado como se muestra en la siguiente imagen (Figura 25)

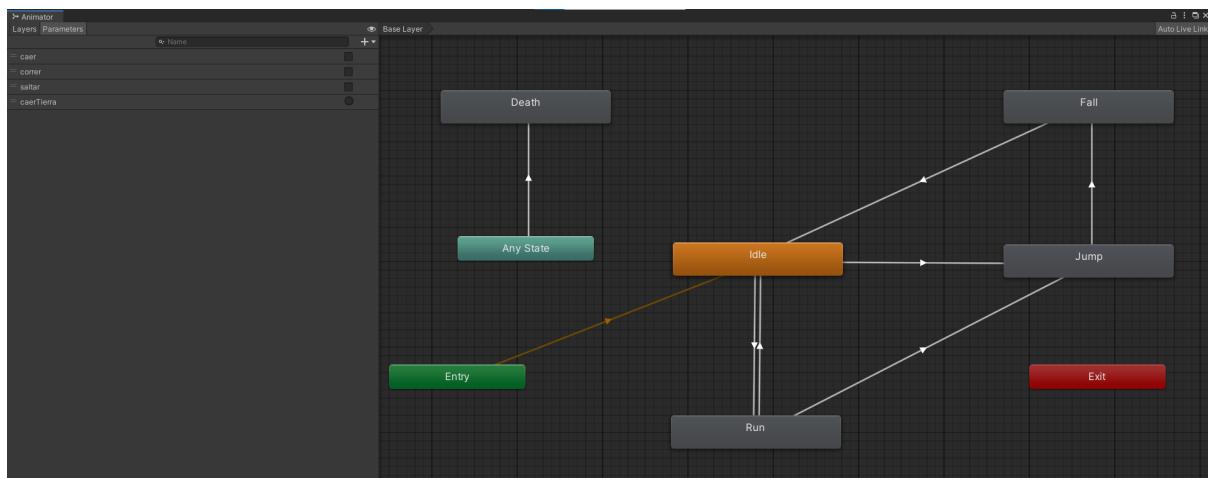


Figura 25. Animator Controller “Jugador”

Por lo general, al crear una transición, la opción “Has Exit Time” está habilitada. Este parámetro si está activado espera que finalice la animación antes de realizar la transición a la animación siguiente. Como en nuestro caso, no nos interesa (en general) que la animación acabe antes de saltar a la siguiente, vamos a desmarcar esta opción de todas las transiciones.

El siguiente paso es conseguir que nuestro personaje tenga masa propia. Para ello vamos a seleccionar al Jugador y añadir el componente “Rigidbody 2D” (Figura 26). Al añadir este componente podemos editar algunos parámetros como la masa del personaje o la escala de gravedad. De momento, dejaremos estos parámetros por defecto.

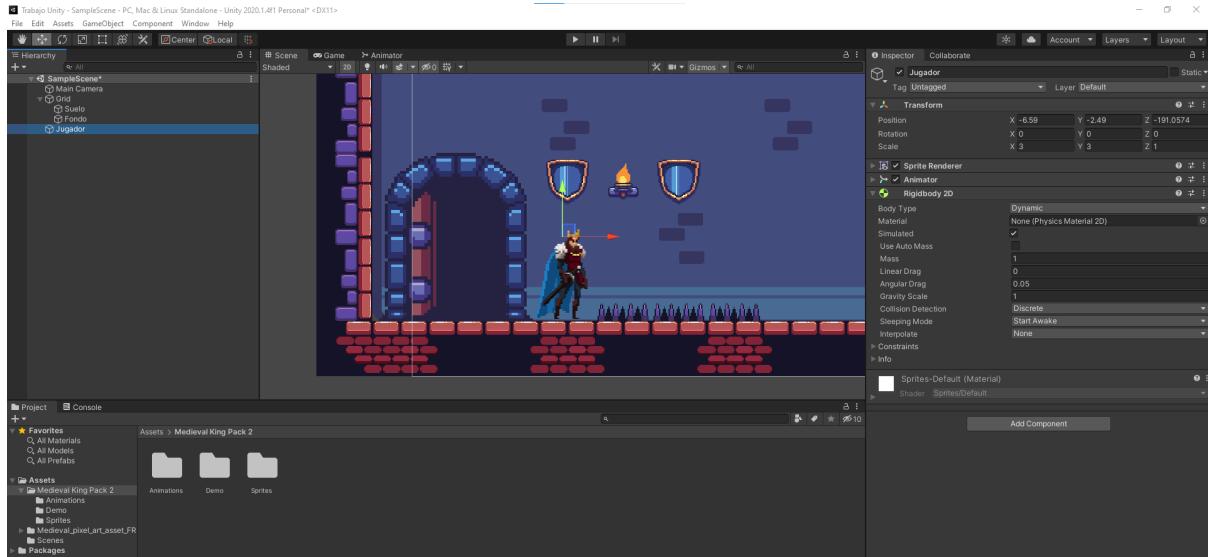


Figura 26. Rigidbody 2D

Para que nuestro personaje pueda mantenerse en el suelo y detectar las colisiones con este, vamos a añadir un componente “Capsule Collider 2D” al nuestro jugador (Figura 27). En este componente, vamos a tener distintos parámetros para ajustar este área del collider a nuestro personaje. El resultado se puede ver en la siguiente imagen:

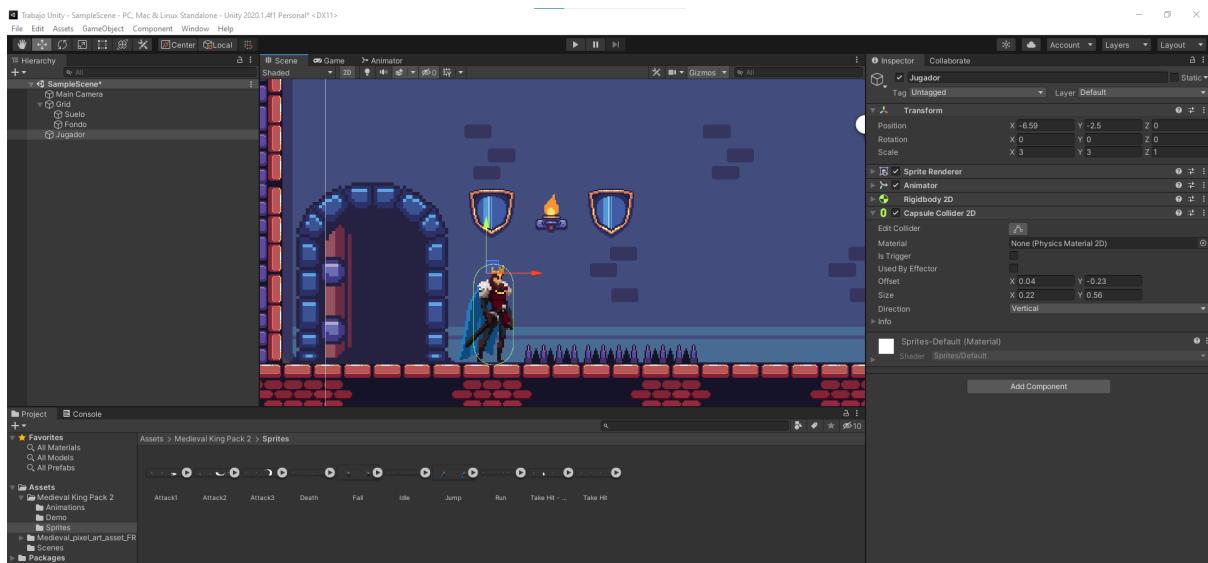


Figura 27. Capsule Collider 2D

Para evitar que el personaje se quede atascado en las plataformas debido a la fricción, vamos a crear un nuevo material (al que llamaremos “Hielo” por ejemplo) que no tenga ningún tipo de fricción y asociaremos dicho material a nuestro suelo del nivel. Para crear el material haremos click derecho (Create→Physics Material 2D) y le asignamos el nombre de “Hielo” al nuevo material. Después seleccionamos dicho material, y en la propiedad de fricción escribimos un 0 (Figura 28).

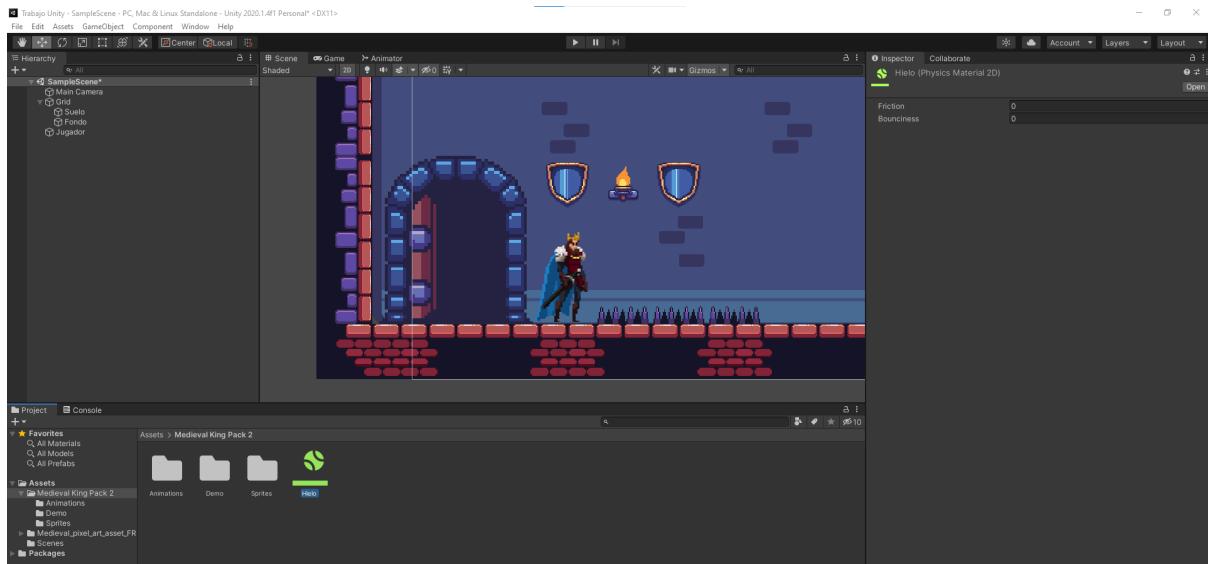


Figura 28. Material sin fricción

Ahora, para asociar este material que acabamos de crear al suelo de nuestro escenario vamos a seleccionar el suelo y en el parámetro Material del componente Tilemap Collider 2D vamos a arrastrar (o seleccionar) el material que hemos creado (Figura 29).

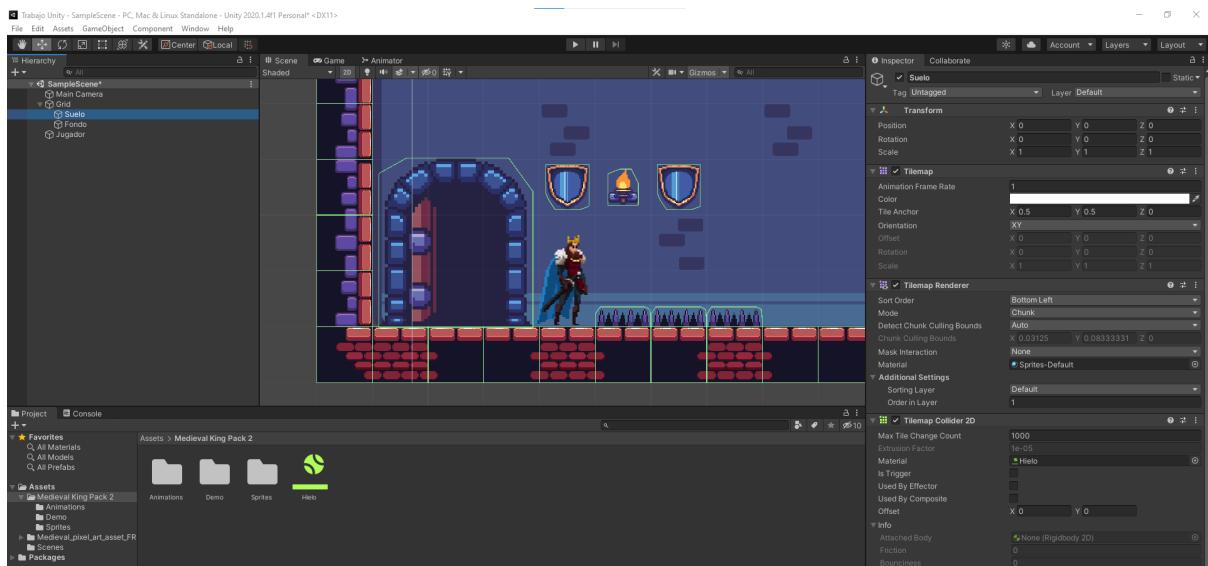


Figura 29. Asociamos el material al suelo

Ahora, el siguiente paso, es crear un script en C# (Create→C# Script) llamado “Jugador” que se encargue de manejar el movimiento de nuestro jugador. Este script lo vamos a guardar en una carpeta llamada Scripts junto con otros scripts que crearemos posteriormente.

Para este script “Jugador” vamos a reutilizar el script que utilizamos en clase en el tema 10 para realizar el movimiento del personaje. Más adelante realizaremos varias modificaciones sobre este script pero de momento lo vamos a dejar así. El contenido de este Script será el siguiente:

```

using UnityEngine;
using System.Collections;

public class Jugador : MonoBehaviour
{
    public float velocidad = 5f;
    public float salto = 15f;
    protected Rigidbody2D rigidBody2D;
    protected Animator animator;
    protected Collider2D collider2d;

    // Use this for initialization
    void Start()
    {
        rigidBody2D = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        collider2d = GetComponent<Collider2D>();
    }

    // Update is called once per frame
    void Update()
    {

    }

    void FixedUpdate()
    {
        rigidBody2D.velocity = new Vector2(velocidad * Input.GetAxis("Horizontal"),
        rigidBody2D.velocity.y);
        animator.SetBool("correr", rigidBody2D.velocity.x != 0);

        //roto al personaje para es lo escalo a -1 o lo giro 180 en el eje y
        if (rigidBody2D.velocity.x < 0)
            transform.rotation = Quaternion.Euler(0, 180, 0);
        else if (rigidBody2D.velocity.x > 0)
            transform.rotation = Quaternion.Euler(0, 0, 0);

        //saltar arriba
        if (rigidBody2D.velocity.y > 0 && Physics2D.OverlapBox(transform.GetChild(0).position, new
        Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) == null)
        {
            animator.SetBool("saltar", true);
        }
    }
}

```

Una vez tenemos creado el script, tenemos que asociar el script al Jugador. Para ello, vamos a seleccionar el Jugador y vamos a arrastrar el script Jugador a la barra derecha de los componentes (Figura 30).

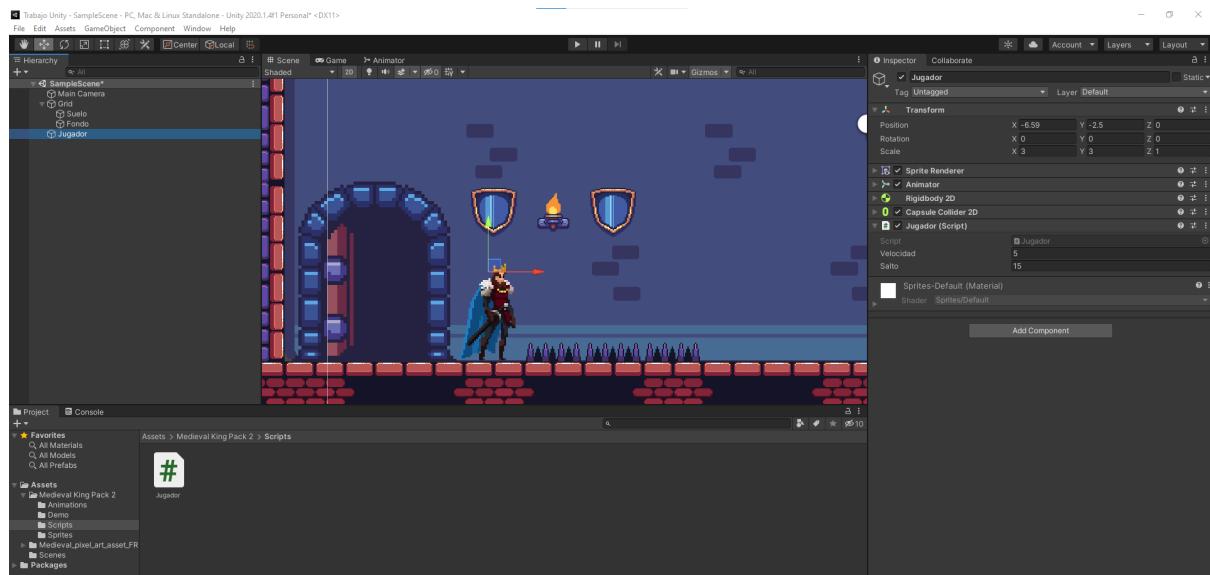


Figura 30. Script Jugador

Como se puede observar, al asociar el script al jugador podemos seleccionar la velocidad y el salto que queramos que tenga el personaje cuando estemos jugando. Para encontrar un valor correcto de velocidad y de salto simplemente iremos probando el juego con diferentes valores hasta encontrar unos valores que nos parezcan adecuados para el juego. En nuestro caso, hemos elegido un valor de velocidad de 5 y un valor de salto de 7.5.

Si ejecutamos el juego en el estado actual, al caminar el personaje, este se girara y se tumbara. Esto se debe a que debemos habilitar una opción dentro del Rigidbody del Jugador. De esta manera, vamos a ir a los parámetros avanzados del Rigidbody 2D y vamos a habilitar la opción de Freeze Rotation Z (Figura 31).

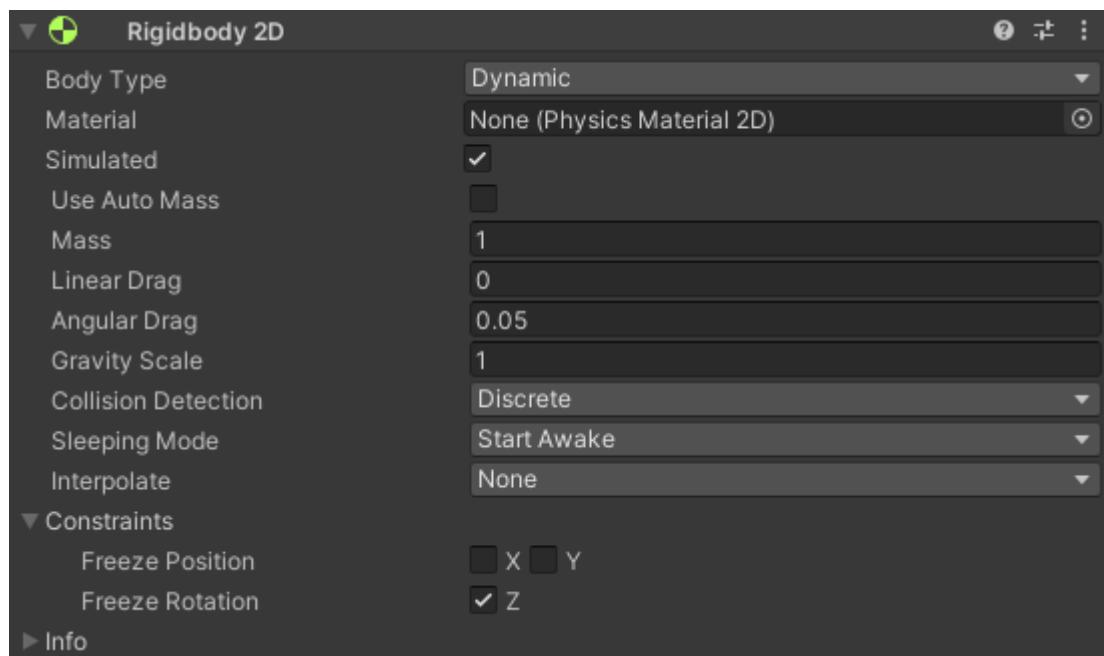


Figura 31. Freeze Rotation Z

Ahora, en el estado actual del juego, podemos movernos con las flechas por el escenario pero no podemos saltar. Esto se debe a que el script utiliza un objeto hijo del jugador para realizar el salto. Para poder hacer esto vamos a crear un nuevo gameObject (GameObject→Create Empty), lo vamos a renombrar para que se llame Suelo, lo vamos a colocar justo en los pies del jugador y vamos a colocar este gameObject como un hijo del Jugador (Figura 32).

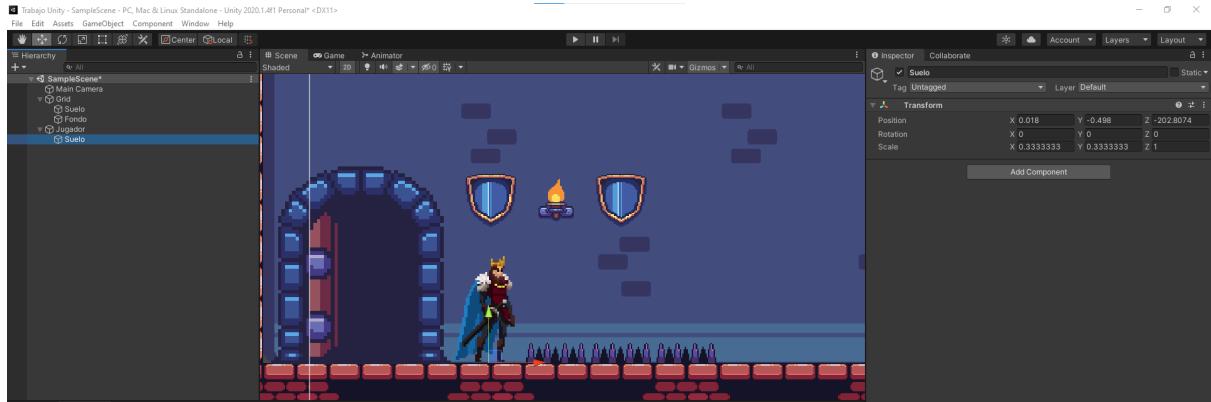


Figura 32. GameObject Suelo

Una vez hemos conseguido que nuestro personaje pueda moverse por el escenario, el siguiente paso es conseguir que la cámara del juego siga al personaje. Para ello vamos a crear un nuevo script llamado Camara2D. Este script será el mismo que el utilizado en los temas 9 y 10 de la asignatura. El contenido de este Script será el siguiente:

```
using UnityEngine;
using System.Collections;

public class Camara2D : MonoBehaviour
{
    public GameObject gameObjectSeguir;
    protected float margenX;
    protected float margenY;
    protected float currentVelocityX;
    protected float currentVelocityY;

    // Use this for initialization
    void Start()
    {
        Vector2 vector2min = Camera.main.ViewportToWorldPoint(new
        Vector2(0, 0));
        Vector2 vector2max = Camera.main.ViewportToWorldPoint(new
        Vector2(1, 1));
        margenX = (vector2max.x - vector2min.x) / 2.5f;
        margenY = (vector2max.y - vector2min.y) / 2.5f;
    }
}
```

```

// Update is called once per frame
void Update()
{
    moverCamara();
}

void moverCamara()
{
    Vector2 vector2min = Camera.main.ViewportToWorldPoint(new Vector2(0, 0));
    Vector2 vector2max = Camera.main.ViewportToWorldPoint(new Vector2(1, 1));
    Vector3 nuevaPosicion = new Vector3();

    //el jugador siempre entre lo límites en el ejeX e ejeY
    //calculo desplazamiento eje x y eje y
    float desplazamientoJugadorX = Mathf.Clamp(gameObjectSeguir.transform.position.x,
vector2min.x + margenX, vector2max.x - margenX);
    desplazamientoJugadorX = gameObjectSeguir.transform.position.xdesplazamientoJugadorX;
    float desplazamientoJugadorY = Mathf.Clamp(gameObjectSeguir.transform.position.y,
vector2min.y + margenY, vector2max.y - margenY);
    desplazamientoJugadorY = gameObjectSeguir.transform.position.y - desplazamientoJugadorY;

    //para hacer que el movimiento de la camara sea suave
    nuevaPosicion.x = Mathf.SmoothDamp(transform.position.x,
transform.position.x + desplazamientoJugadorX, ref currentVelocityX, 0.1f);
    nuevaPosicion.y = Mathf.SmoothDamp(transform.position.y,
transform.position.y + desplazamientoJugadorY, ref currentVelocityY, 0.1f);
    nuevaPosicion.z = Camera.main.transform.position.z;
    Camera.main.transform.position = nuevaPosicion;
}
}

```

Una vez tenemos creado el script, tenemos que asociar el script al Jugador. Para ello, vamos a seleccionar el Jugador y vamos a arrastrar el script Camara2D a la barra derecha de los componentes. Una vez hecho esto, tenemos que arrastrar la cámara principal al parámetro Game Object Seguir del script para indicar al script cuál es la cámara que tiene que manejar (Figura 33).

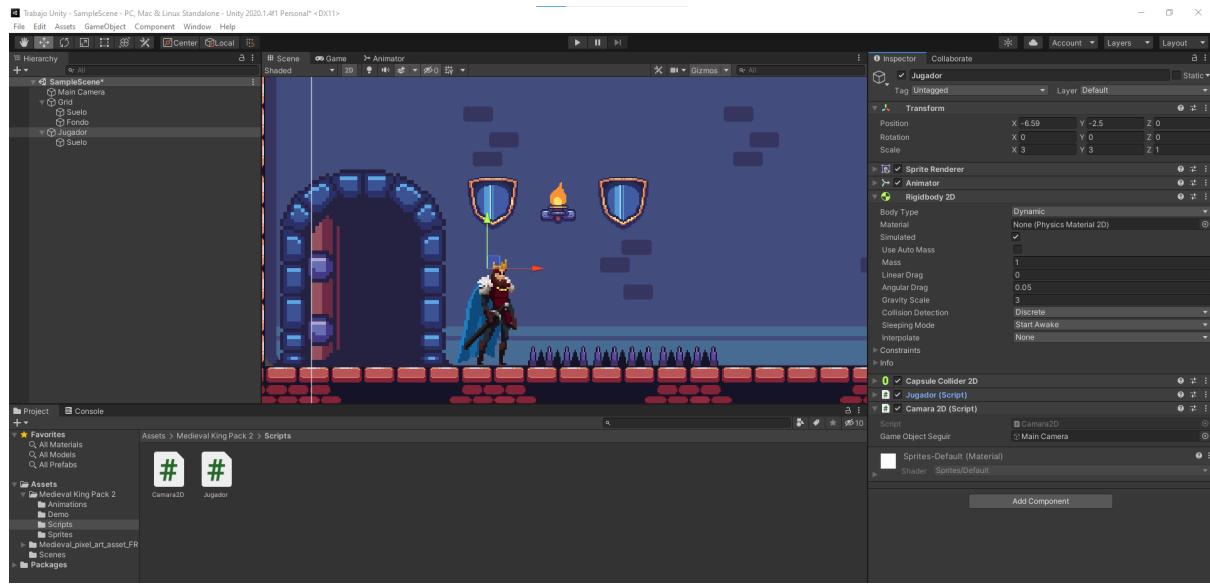


Figura 33. Script Camara2D

Con esto, la cámara principal ya sigue a nuestro personaje mientras este se mueve por el escenario.

Una vez hemos podido mover el personaje por el escenario, hemos detectado un error que hemos cometido a la hora de crear la infraestructura del nivel. Al añadir los elementos decorativos a la escena (antorchas, escudos, puertas), lo hemos hecho sobre el Tilemap suelo, por lo que al añadir el componente Tilemap Collider 2D al suelo, estos elementos decorativos también detectan las colisiones y esto no lo queremos.

De esta manera, vamos a crear un tercer Tilemap llamado Accesorios y a través de la paleta de accesorios que creamos antes, dibujaremos los accesorios tal como estaban antes. En último lugar, estableceremos la propiedad Order in Layer de este Tilemap a 1 al igual que el Fondo (Figura 34).

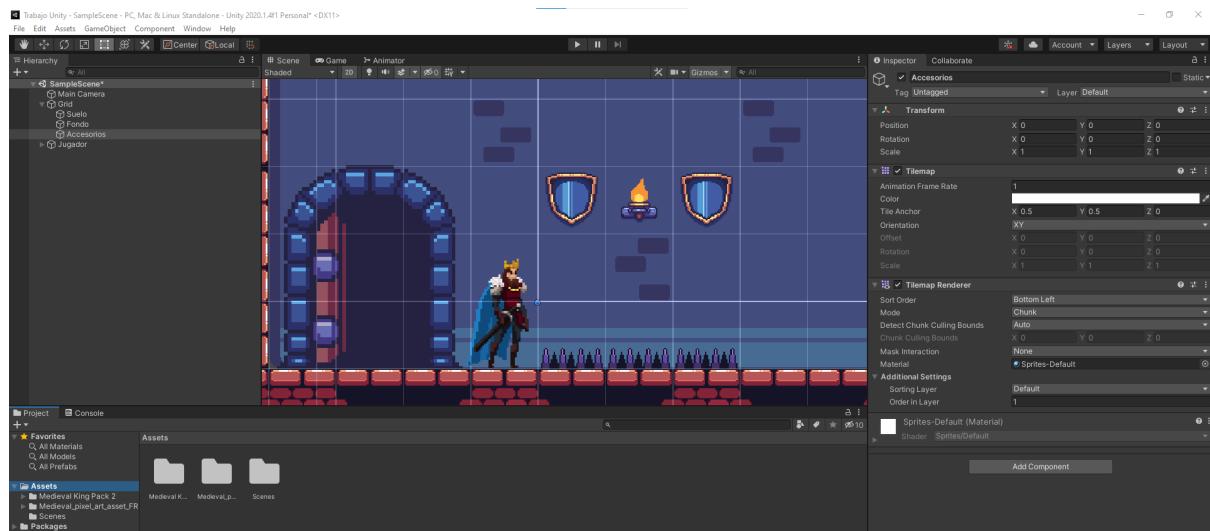


Figura 34. Tilemap Accesorios

Podríamos haber dibujado estos accesorios en el propio Tilemap Fondo pero al contar estos accesorios con un color de fondo distinto, estropearía en cierta medida el resultado del fondo y esto no es lo que queremos.

Ahora, vamos a modificar el script de movimiento del jugador con el objetivo de poder realizar dobles saltos. Para ello vamos a abrir el código del script Jugador y vamos a introducir las siguientes modificaciones que van a estar sombreadas en amarillo.

```
using UnityEngine;
using System.Collections;

public class Jugador : MonoBehaviour
{
    public float velocidad = 5f;
    public float salto = 15f;
    protected Rigidbody2D rigidBody2D;
    protected Animator animator;
    protected Collider2D collider2d;
    public int dobleSalto;

    // Use this for initialization
    void Start()
    {
        rigidBody2D = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        collider2d = GetComponent<Collider2D>();
        dobleSalto = 2;
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.UpArrow) && dobleSalto > 0)
        {
            Debug.Log(rigidBody2D.velocity.y);
            rigidBody2D.AddForce(new Vector2(0, (salto * 2) - rigidBody2D.velocity.y),
ForceMode2D.Impulse);
            Debug.Log("Salto " + Time.time);
            dobleSalto = dobleSalto - 1;
        }
    }

    void FixedUpdate()
    {
```

```

        rigidBody2D.velocity = new Vector2(velocidad * Input.GetAxis("Horizontal"),
        rigidBody2D.velocity.y);
        animator.SetBool("correr", rigidBody2D.velocity.x != 0);

//roto al personaje para es lo escalo a -1 o lo giro 180 en el eje y
if (rigidBody2D.velocity.x < 0)
    transform.rotation = Quaternion.Euler(0, 180, 0);
else if (rigidBody2D.velocity.x > 0)
    transform.rotation = Quaternion.Euler(0, 0, 0);

//saltar arriba
if (rigidBody2D.velocity.y > 0 && Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) == null)
{
    animator.SetBool("saltar", true);
    if (animator.GetBool("caer"))
    {
        animator.SetBool("caer", false);
    }
}

//cayendo
else if (rigidBody2D.velocity.y < 0 && animator.GetBool("saltar"))
{
    animator.SetBool("saltar", false);
    animator.SetBool("caer", true);
}

//toco tierra
else if (animator.GetBool("caer") && Physics2D.OverlapBox(transform.GetChild(0).position,
new Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
{
    animator.SetBool("saltar", false);
    animator.SetBool("caer", false);
    animator.SetTrigger("caerTierra");
    dobleSalto = 2;
}

//el overlapbox se hace la mitad del collider para evitar que roce en lateral con una plataforma y
me deje saltar de nuevo, (collider2d.bounds.max.x - collider2d.bounds.min.x) / 2
if (Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
{
    //Debug.Log("SAlto " + Time.time);
}

```

```
if (Input.GetAxis("Jump") != 0)
{
    rigidBody2D.AddForce(new Vector2(0, salto), ForceMode2D.Impulse);
    Debug.Log("SAlto " + Time.time);
}
```

Las modificaciones dentro del script del jugador consisten en lo siguiente:

- Hemos añadido un parámetro entero llamado dobleSalto que nos va a permitir gestionar el doble salto.
  - En el método Start hemos asignado a la variable dobleSalto el valor 2 ya que es el número máximo de saltos permitidos sin tocar el suelo.
  - En el método Update hemos añadido unas líneas de código las cuales nos permiten saltar con la flecha arriba. Este código evalúa el número de saltos realizados de forma que cuando llevemos 2 saltos realizados no podremos realizar otro salto hasta que hayamos caído al suelo y se haya restablecido el valor inicial de la variable dobleSalto.
  - Cuando el personaje toca el suelo, hemos añadido una línea que nos establece el valor de dobleSalto a su valor original de 2.

Además, hemos descomentado una línea en la que establecemos el parámetro Saltar a falso. Esta línea con el código original no es necesaria incluirla pero al realizar el doble salto, se puede dar la posibilidad de que demos el segundo salto cuando el personaje ya está cayendo y tal como está configurado el Animator Controller no hay manera de poder volver desde la animación de caída a la animación de salto.

Es por esto por lo que vamos a incluir una nueva transición desde la animación de Fall a la animación de Jump que se produzca cuando la variable saltar sea verdadera (Figura 35). Y para que esto funcione es necesario poner esta variable a falso cuando el personaje empiece a caer.

También se ha incluido una comprobación en la que se cambia el valor de la variable caer de varadero a falso en el caso de que se dé el segundo salto cuando el personaje estuviera cayendo para así evitar que se produzca una transición de Jump a Fall antes de que el personaje estuviera cayendo realmente.

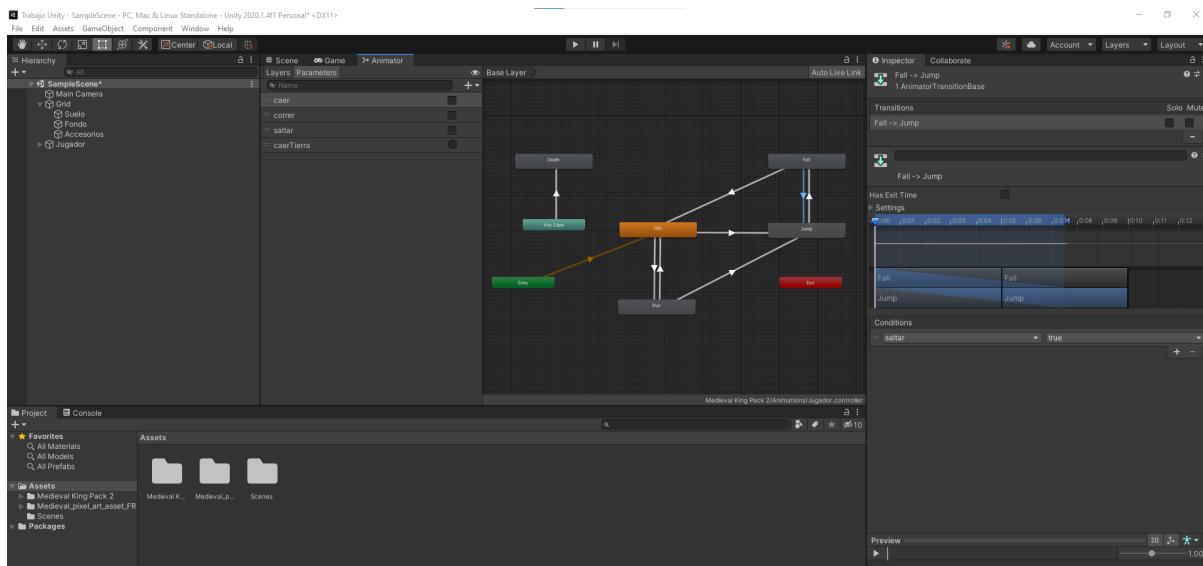


Figura 35. Animator Controller Actualizado

## Creación de los pinchos

Una vez hemos configurado el doble salto del Jugador, ahora vamos a crear otro script que nos permita gestionar las colisiones con los pinchos de forma que cada vez que mi personaje toque los pinchos, muera automáticamente y vuelva a aparecer al principio del nivel. De esta manera, vamos a modificar el script Jugador de forma que se encargue de realizar las funciones que hemos comentado.

Sin embargo, antes de modificar el script debemos realizar un cambio en la estructura del nivel. Al crear el Tilemap Suelo del nivel incluimos todos los pinchos en este Tilemap Suelo. Para poder gestionar las colisiones únicamente con todos los pinchos (y no con el resto del suelo) vamos a crear un nuevo Tilemap llamado Pinchos que contendrá todos los pinchos del nivel y eliminaremos los pinchos existentes del Tilemap Suelo.

Después, incluiremos la propiedad Tilemap Collider 2D de la misma manera que hicimos con el Tilemap Suelo para detectar las colisiones con los pinchos y en la propiedad Order in Layer del componente Tilemap Renderer le daremos el mismo valor que al Tilemap Suelo que en nuestro caso este valor es de 1 (Figura 36).

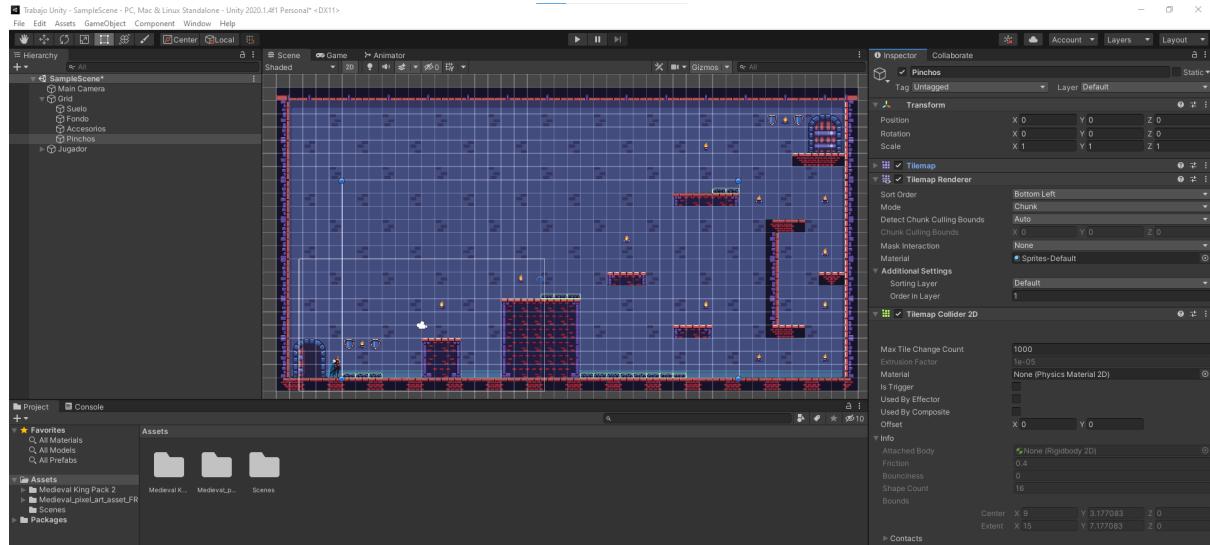


Figura 36. Tilemap Pinchos

Ahora, ya podemos editar el contenido del script Jugador que se encargará de gestionar las colisiones con los pinchos. De manera adicional queremos que se reproduzca un sonido de muerte cuando el Jugador entre en contacto con los pinchos. Para ello, vamos a buscar un sonido de muerte en internet, nos lo vamos a descargar en formato mp3 y lo vamos a importar a nuestro proyecto. Una vez hayamos hecho esto, agregaremos al Jugador un componente Audio Source y arrastraremos nuestro sonido a la propiedad AudioClip de este componente. También desactivaremos la opción Play on Awake de este componente ya que en caso de no hacerlo, el sonido se reproducirá nada más iniciar el juego (Figura 37).

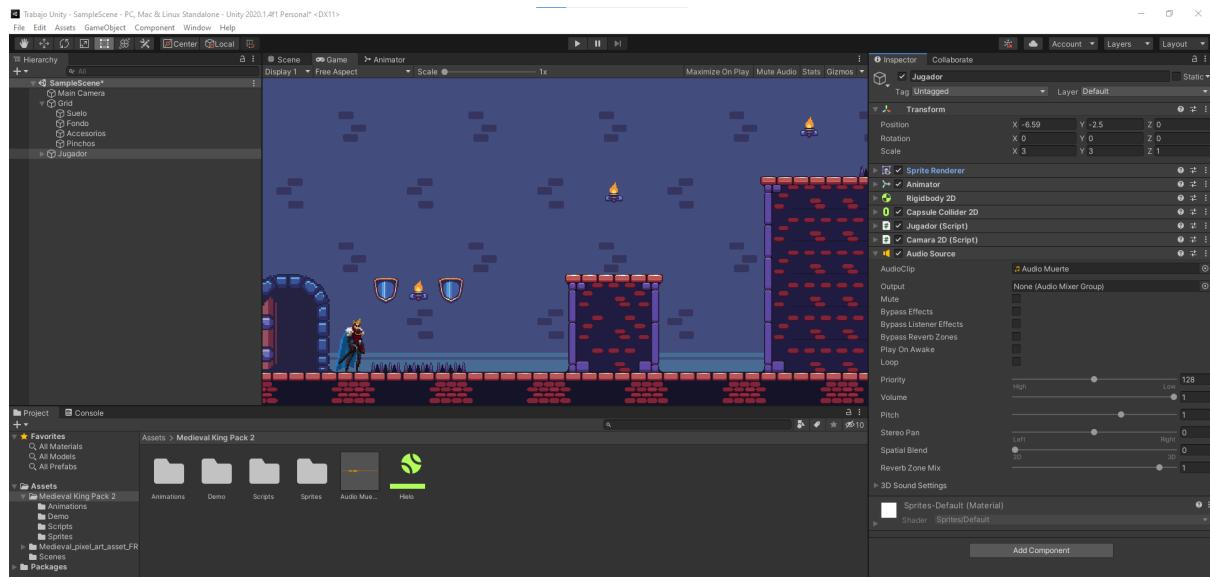


Figura 37. Audio Source

Ahora vamos a editar el contenido del script Jugador. El contenido de este script será el siguiente:

```
using UnityEngine;
```

```

using System.Collections;

public class Jugador : MonoBehaviour
{
    public float velocidad = 5f;
    public float salto = 15f;
    protected Rigidbody2D rigidBody2D;
    protected Animator animator;
    protected Collider2D collider2d;
    public int dobleSalto;
    AudioSource audioSource;

// Use this for initialization
void Start()
{
    rigidBody2D = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
    collider2d = GetComponent<Collider2D>();
    audioSource = GetComponent<// Update is called once per frame
void Update()
{
    if (Input.GetKeyDown(KeyCode.UpArrow) && dobleSalto > 0)
    {
        Debug.Log(rigidBody2D.velocity.y);
        rigidBody2D.AddForce(new Vector2(0, (salto * 2) - rigidBody2D.velocity.y),
ForceMode2D.Impulse);
        Debug.Log("Salto " + Time.time);
        dobleSalto = dobleSalto - 1;
    }
}

void FixedUpdate()
{
    rigidBody2D.velocity = new Vector2(velocidad * Input.GetAxis("Horizontal"),
rigidBody2D.velocity.y);
    animator.SetBool("correr", rigidBody2D.velocity.x != 0);

//roto al personaje para es lo escalo a -1 o lo giro 180 en el eje y
if (rigidBody2D.velocity.x < 0)
    transform.rotation = Quaternion.Euler(0, 180, 0);
else if (rigidBody2D.velocity.x > 0)
}

```

```

        transform.rotation = Quaternion.Euler(0, 0, 0);

//saltar arriba
if (rigidBody2D.velocity.y > 0 && Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) == null)
{
    animator.SetBool("saltar", true);
    if (animator.GetBool("caer"))
    {
        animator.SetBool("caer", false);
    }
}

//cayendo
else if (rigidBody2D.velocity.y < 0 && animator.GetBool("saltar"))
{
    animator.SetBool("saltar", false);
    animator.SetBool("caer", true);
}

//toco tierra
else if (animator.GetBool("caer") && Physics2D.OverlapBox(transform.GetChild(0).position,
new Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
{
    animator.SetBool("saltar", false);
    animator.SetBool("caer", false);
    animator.SetTrigger("caerTierra");
    dobleSalto = 2;
}

//el overlapbox se hace la mitad del collider para evitar que roce en lateral con una plataforma y
me deje saltar de nuevo, (collider2d.bounds.max.x - collider2d.bounds.min.x) / 2
if (Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
{
    //Debug.Log("SAlto " + Time.time);
    if (Input.GetAxis("Jump") != 0)
    {
        rigidBody2D.AddForce(new Vector2(0, salto), ForceMode2D.Impulse);
        Debug.Log("SAlto " + Time.time);
    }
}
}

private void OnCollisionEnter2D(Collision2D collision)

```

```

    {
        if(collision.gameObject.tag == "Pinchos")
        {
            audioSource.Play();
            animator.SetTrigger("morir");
        }
    }
}

```

Básicamente, en este Script cogemos la referencia de nuestro audio de muerte en la función Start y la reproducimos cuando se produce la colisión del personaje con los pinchos. Además de esto, al colisionar el personaje con los pinchos también se va a disparar un trigger llamado morir. Este trigger será utilizado dentro del Animator Controller para realizar la transición entre cualquier estado y el estado de muerte.

Para que esto funcione, primero añadiremos el parámetro morir como un trigger y después añadiremos una condición en la transición que nos lleva al estado de Death en la que se produzca la transición cuando se active el disparador morir (Figura 38).

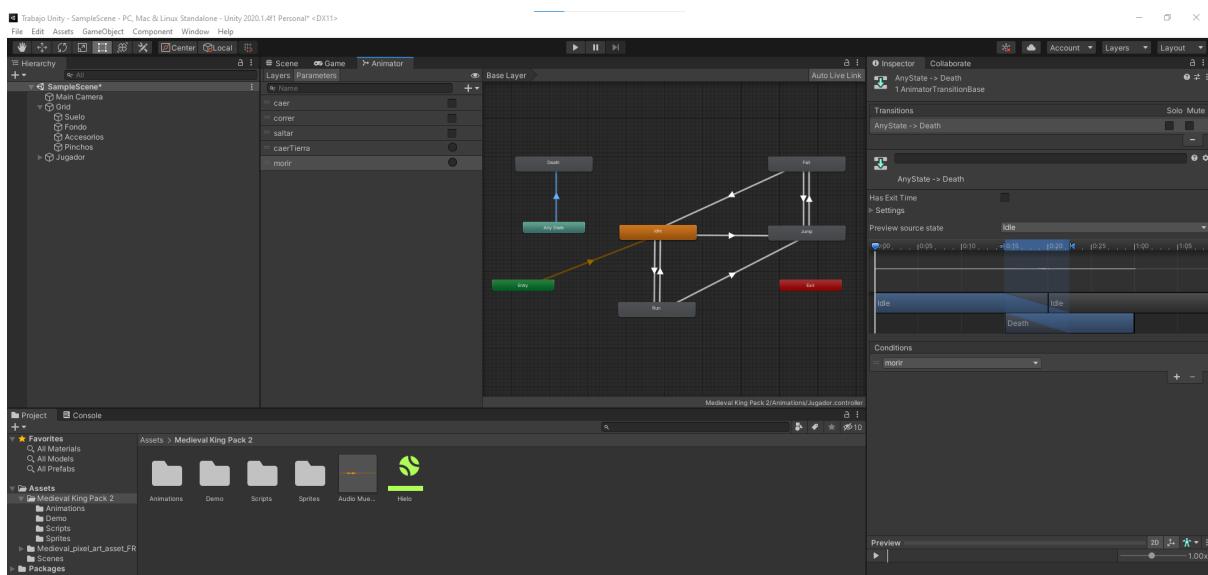


Figura 38. Transacción al estado de Death

Como se puede observar en el código del script Jugador, solo “matamos” al personaje cuando este entra en contacto con los pinchos. Para conseguir esto, hemos añadido una etiqueta al Tilemap Pinchos (Figura 39) para que solo entremos en el if cuando se haya producido una colisión con un objeto que tenga como etiqueta “Pinchos”.

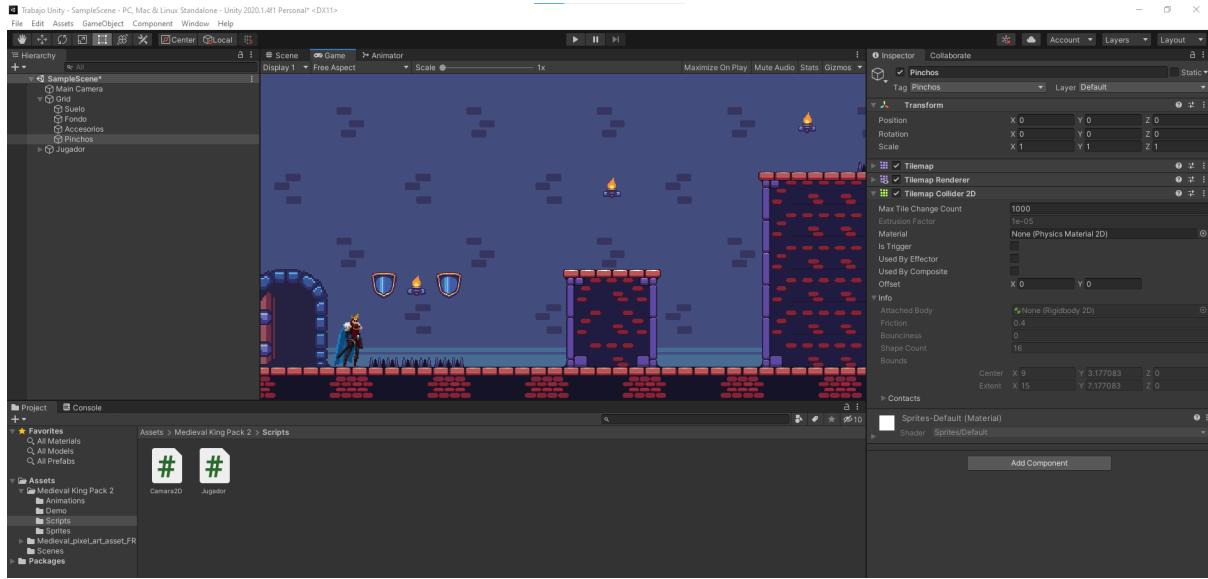


Figura 39. Etiqueta “Pinchos”

Ahora, añadiremos un evento al final de la animación de Death para colocar al Jugador al inicio del nivel al acabar la animación de muerte. Para ello, abriremos la animación de Death, y al final del último fotograma haremos click derecho y pulsaremos la opción “Add Animation Event”. Al hacer esto se nos desplegará una ventana en la que elegiremos la función que queremos que se dispare cuando ocurra el evento (Figura 40). En nuestro caso, añadiremos una función en nuestro script Jugador llamada recolocar que nos coloque al principio del nivel y restablezca las animaciones.

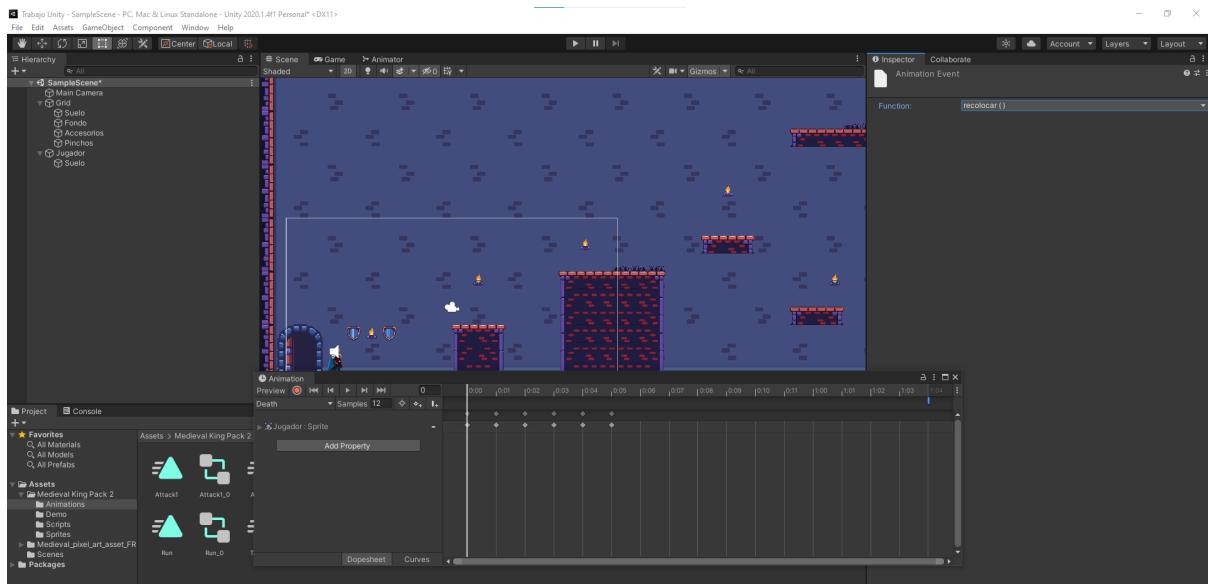


Figura 40. Evento recolocar en el final de la animación Death

Para darle más “dramatismo” a la muerte del jugador hemos añadido unos cuantos huecos sin fotografías en la animación para que transcurra un breve periodo de tiempo desde que muere el jugador hasta que se recoloca en su posición inicial. De no hacer esto, la transición sería muy abrupta y poco estética visualmente.

En cuanto al contenido del script Jugador se han añadido las siguientes líneas de código:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Jugador : MonoBehaviour
{
    public float velocidad = 5f;
    public float salto = 15f;
    protected Rigidbody2D rigidBody2D;
    protected Animator animator;
    protected Collider2D collider2d;
    public int dobleSalto;
    AudioSource audioSource;
    private Vector3 posicionInicial;

    // Use this for initialization
    void Start()
    {
        rigidBody2D = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        collider2d = GetComponent<Collider2D>();
        audioSource = GetComponent<
```

```

        rigidBody2D.velocity = new Vector2(velocidad * Input.GetAxis("Horizontal"),
        rigidBody2D.velocity.y);
        animator.SetBool("correr", rigidBody2D.velocity.x != 0);

//roto al personaje para es lo escalo a -1 o lo giro 180 en el eje y
if (rigidBody2D.velocity.x < 0)
    transform.rotation = Quaternion.Euler(0, 180, 0);
else if (rigidBody2D.velocity.x > 0)
    transform.rotation = Quaternion.Euler(0, 0, 0);

//saltar arriba
if (rigidBody2D.velocity.y > 0 && Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) == null)
{
    animator.SetBool("saltar", true);
    if (animator.GetBool("caer"))
    {
        animator.SetBool("caer", false);
    }
}

//cayendo
else if (rigidBody2D.velocity.y < 0 && animator.GetBool("saltar"))
{
    animator.SetBool("saltar", false);
    animator.SetBool("caer", true);
}

//toco tierra
else if (animator.GetBool("caer") && Physics2D.OverlapBox(transform.GetChild(0).position,
new Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
{
    animator.SetBool("saltar", false);
    animator.SetBool("caer", false);
    animator.SetTrigger("caerTierra");
    dobleSalto = 2;
}

//el overlapbox se hace la mitad del collider para evitar que roce en lateral con una plataforma y
me deje saltar de nuevo, (collider2d.bounds.max.x - collider2d.bounds.min.x) / 2
if (Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
{
    //Debug.Log("SAlto " + Time.time);
}

```

```

if (Input.GetAxis("Jump") != 0)
{
    rigidBody2D.AddForce(new Vector2(0, salto), ForceMode2D.Impulse);
    Debug.Log("SAlto " + Time.time);
}
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Pinchos")
    {
        audioSource.Play();
        animator.SetTrigger("morir");
    }
}

public void recolocar()
{
    animator.SetTrigger("reinicio");
    rigidBody2D.velocity = Vector3.zero;
    transform.position = posicionInicial;
}
}

```

Los cambios realizados en el script Jugador se basan en la instrucción de una nueva función llamada recolocar la cual se va a encargar de gestionar la colocación del personaje al principio del nivel. Como hemos visto, esta función se ejecuta cuando acaba la animación de muerte del personaje. Para recolocar el personaje en la posición inicial, en la función start() se almacena el valor de la posición para posteriormente usar este valor para recolocar al jugador.

Al recolocar el jugador, debemos asegurarnos de que la animación que se esté ejecutando es la de Idle y no la de Death. Es por esto por lo que vamos a añadir una nueva transacción que parte de Death hasta Idle (Figura 41). Esta transición se activará cuando se active el disparador reinicio (no olvidar de añadir este parámetro al Animator Controller). Es por esto por lo que en la función recolocar activamos el disparador de reinicio.

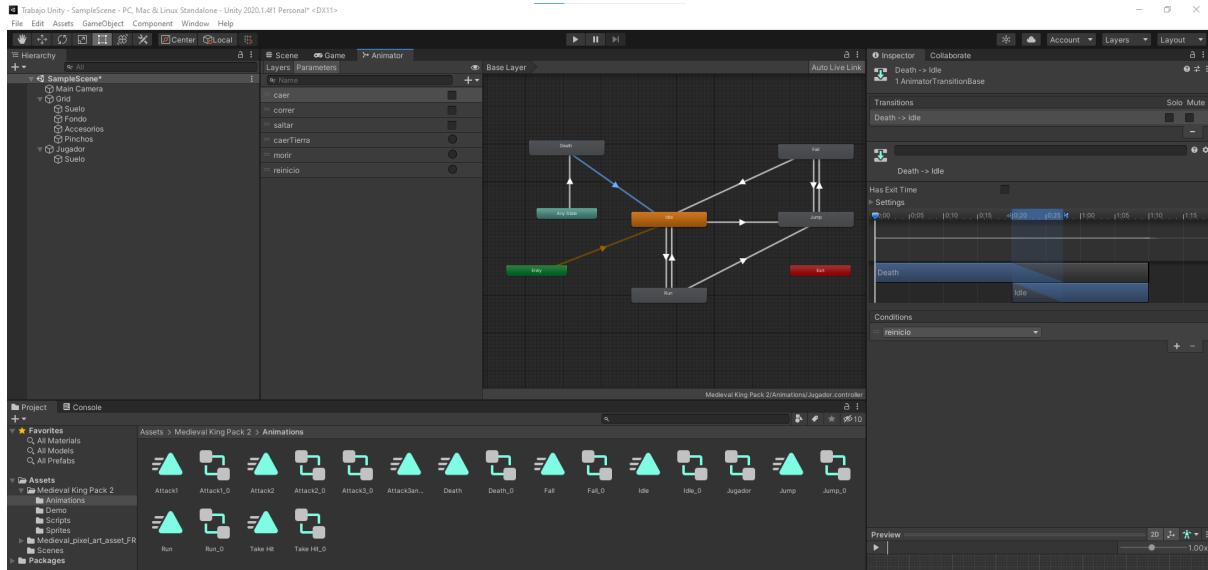


Figura 41. Transición de reinicio en el Animator Controller

Con esto ya tenemos toda la gestión de los pinchos correctamente realizada y se podrá reutilizar para todos los niveles del juego.

Por último, debemos configurar el juego para realizar el cambio de nivel cuando el Jugador haya llegado a la puerta que marca el final del nivel. Para ello vamos a utilizar una solución muy sencilla. Vamos a borrar la puerta que marca el final del nivel y que se encontraba en el Tilemap Accesorios y la vamos a añadir al escenario como un objeto independiente. Con esto conseguimos que la puerta de fin tenga un collider independiente y cuando se detectan colisiones con esta puerta, se ejecutarán los procedimientos necesarios para cambiar la escena y por lo tanto el nivel.

De esta manera, vamos a agregar esta puerta de salida como un objeto independiente arrastrando el tile directamente al escenario. Una vez hecho esto, añadiremos un componente Box Collider 2D que se ajuste al tamaño de la puerta para que detecte las colisiones (Figura 42).

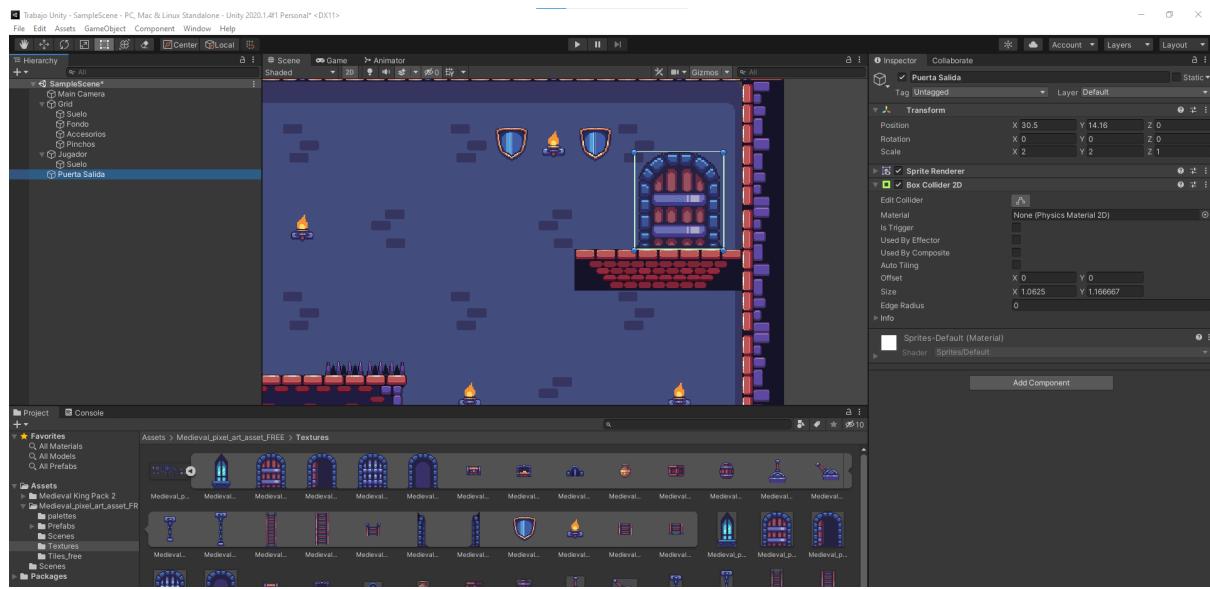


Figura 42. Puerta de Salida

Ahora crearemos un pequeño script llamado cambioNivel que hará las gestiones necesarias para realizar este cambio de escenario. El contenido de este script será el siguiente:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class cambioNivel : MonoBehaviour
{
    public int numeroNivel;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        SceneManager.LoadScene("Nivel" + numeroNivel);
    }
}

```

}

Ahora, lo último que tenemos que hacer es asociar este script a la puerta de salida. Como ya hemos visto, podemos asociar directamente este script arrastrándolo a la ventana de componentes de la puerta de salida (Figura 43).

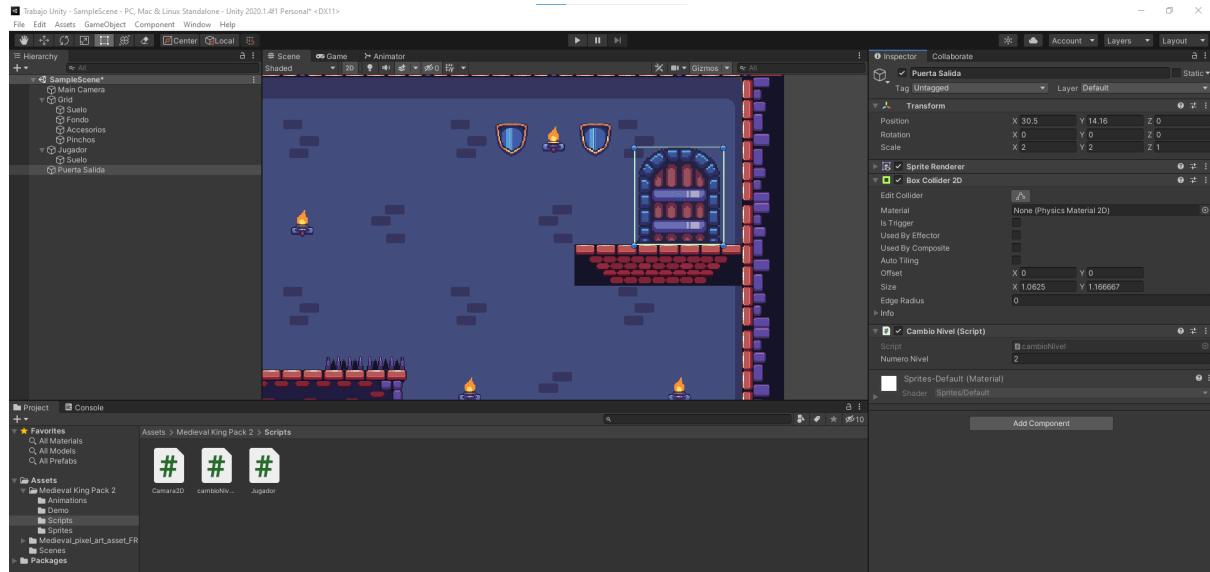


Figura 43. Puerta de Salida con el script de cambioNivel asociado

Como se observa en la figura 43, al asociar el script a la puerta de salida, podemos elegir directamente a qué nivel queremos ir. Para ello, lo único que tenemos que hacer es cambiar el parámetro “Número Nivel” pero el número del nivel al que queremos cambiar. De esta manera, podemos reutilizar este script para resaltar los cambios de niveles ya que lo único que tenemos que cambiar es este parámetro.

Para poder realizar correctamente el cambio al nivel 2, tenemos que tener creada ya la escena (aunque esté vacía) del nivel 2 y tenemos que construir el juego incluyendo este segundo nivel. Para incluir la escena del nivel 2 dentro de la construcción del juego, accederemos a File→Build Settings.

Una vez hecho esto, se nos desplegará una ventana en la que se muestran las escenas que forman parte del juego. En nuestro caso, sólo aparecerá la escena del nivel 1. Para agregar la escena del nivel 2 a la construcción del juego, accederemos a la escena del nivel 2 y pulsaremos el botón de “Add Open Scenes”. El resultado será como el de la siguiente imagen (Figura 44):

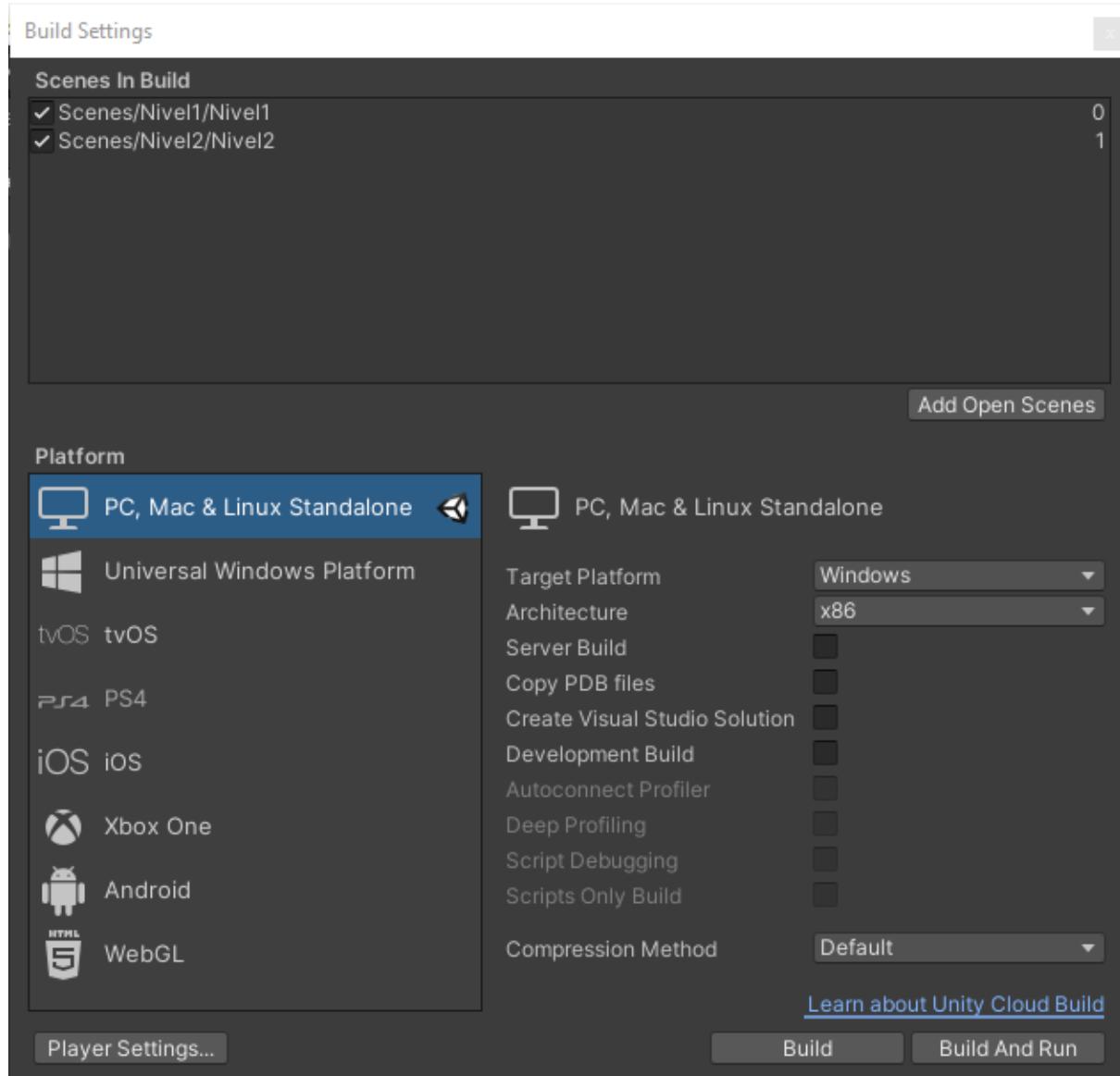


Figura 44. Puerta de Salida con el script de cambioNivel asociado

Esto se tendrá que realizar para todas las escenas de juego para poder así realizar los cambios de nivel. Cuando se construyen los siguientes niveles, también se añadirán a la construcción del juego pero no volveremos a mencionar la realización de este paso en el informe por que ya se da por explicada.

Antes de pasar al desarrollo del Nivel 2, quería comentar un pequeño detalle. El parámetro Order In Layer del componente Sprite Renderer de nuestro Jugador tiene un valor de 0. Si mantenemos este valor de 0, las decoraciones se verán por delante del juego y eso estéticamente es bastante feo. Para solucionar esto simplemente, hay que cambiar este valor del Order In Layer del personaje a 2 para que así el Jugador se represente por delante de las decoraciones.

## Nivel 2

Hasta ahora, hemos realizado todos los pasos necesarios para mover a nuestro personaje por el escenario y hemos realizado tanto la infraestructura como la funcionalidad del primer nivel del juego. Ahora, vamos a definir la infraestructura y funcionalidad del segundo nivel. Para ello, vamos a partir de la estructura del nivel 1 y vamos a introducir un par de cambios que supondrán un aumento en la dificultad.

En primer lugar, vamos a crear una nueva escena llamada Nivel2 la cual va ser originalmente una copia exacta de lo que es el nivel 1. Una vez hemos creado esta nueva escena y hemos copiado todos los elementos de la escena anterior, vamos a añadir el siguiente componente de dificultad. De esta manera, en este nivel, además de haber pinchos que matan al Jugador, van a existir 2 cañones que van a estar disparando balas a un punto fijo con una frecuencia determinada.

Para añadir esta funcionalidad, en primer lugar, vamos a buscar en Internet una imagen del cañón y una imagen de las balas que dispara el cañón. En nuestro caso, hemos encontrado una imagen en la que se incluye tanto el caño como la bala (Figura 45).

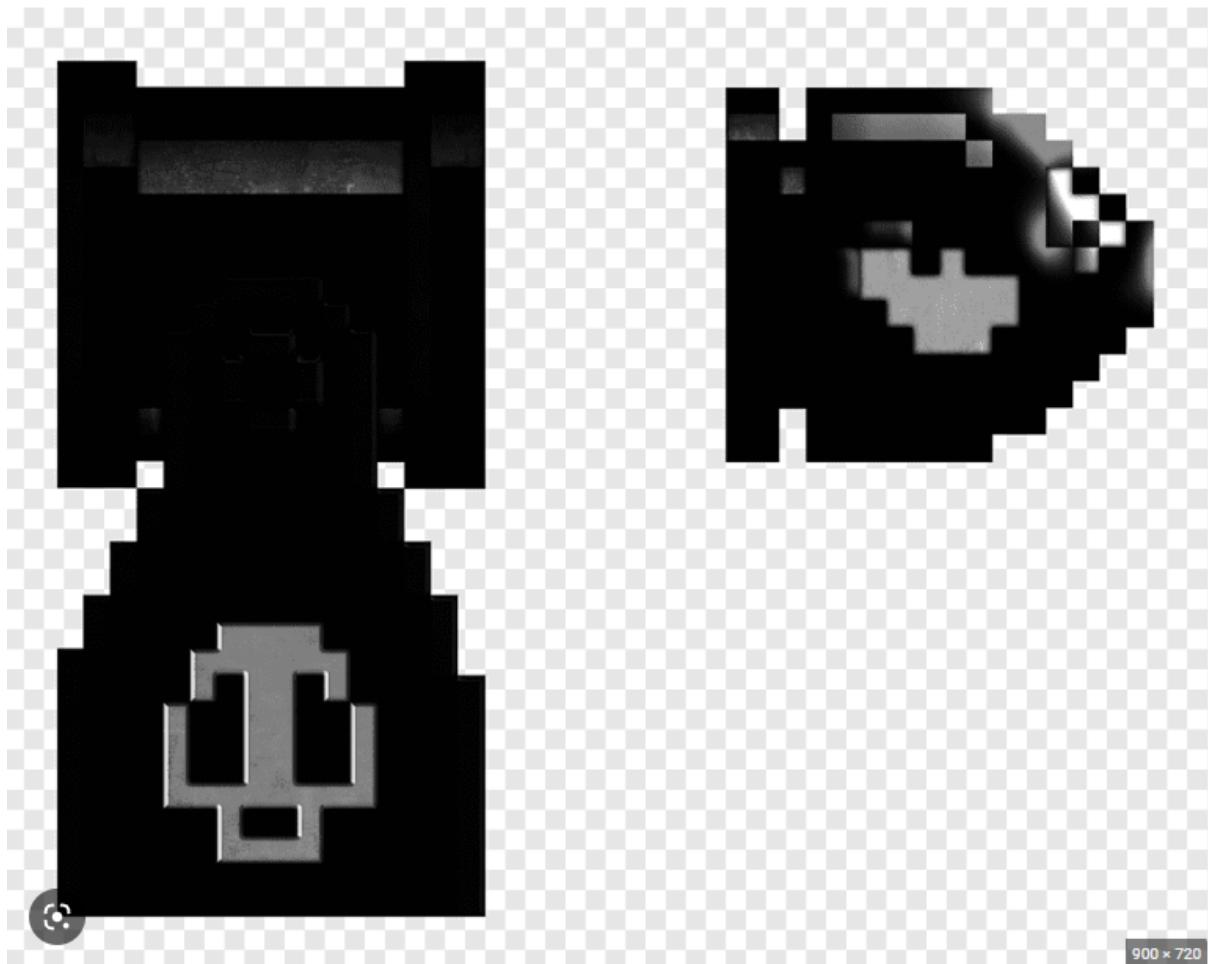


Figura 45. Imagen Cañón y Bala

De esta manera, vamos a descargarnos esta imagen y la vamos a incluir en nuestro proyecto. Al añadir esta imagen al proyecto tenemos un sprite simple que contiene tanto la bala como el cañón. El siguiente paso es conseguir un sprite separado de la bala y un sprite separado del cañón. Para ello, vamos a seleccionar el sprite y en la propiedad Sprite Mode vamos a cambiar su valor de “Single” a “Multiple” (Figura 46).

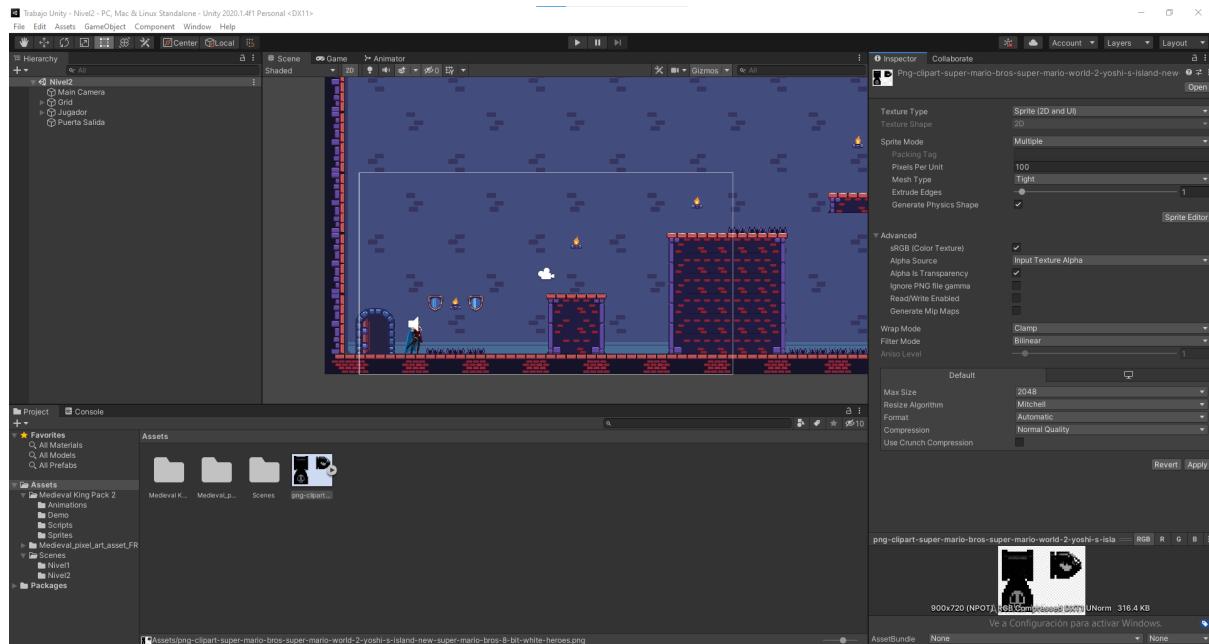


Figura 46.      Sprite Cañón y Bala

Ahora, vamos a seleccionar la opción de “Sprite Editor” y vamos a recortar de esta imagen los sprites de la bala y del cañón respectivamente. Para ello simplemente hay que seleccionar el área de la imagen que queremos recortar y el nombre del recorte. Una vez se quiera hacer el recorte simplemente hay que pulsar el botón de Apply (Figura 47).

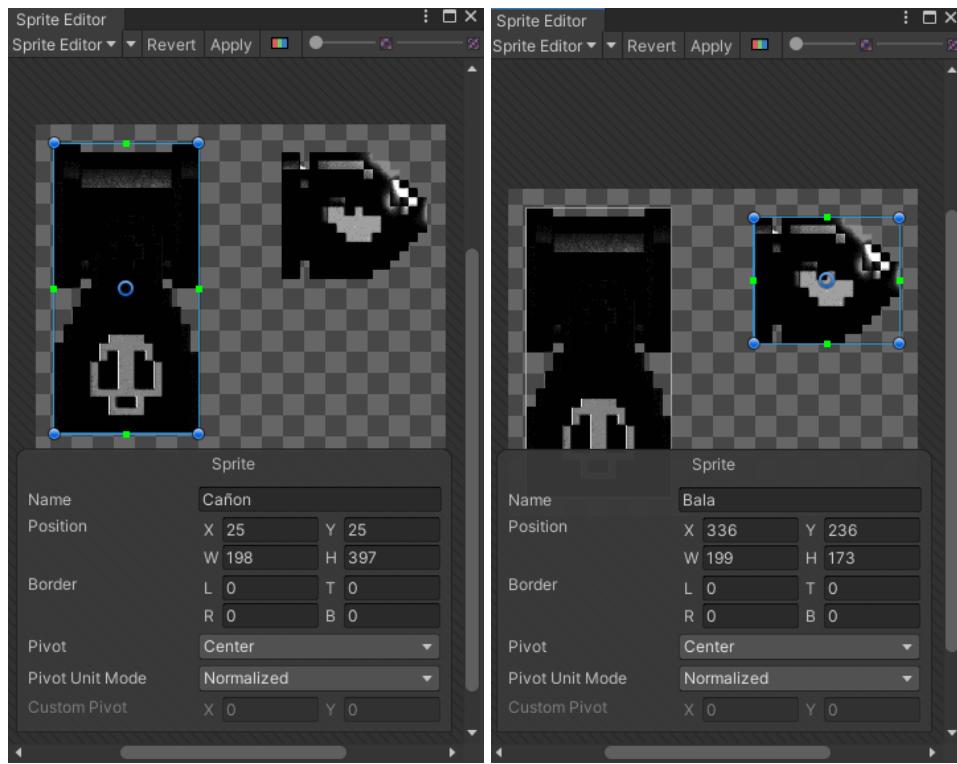


Figura 47. Creando los Sprites Bala y Cañón

Ahora, vamos a ir al tilemap Suelo y vamos a agregar tanto la infraestructura en la que se van a soportar los cañones como los propios cañones en sí (Figura 48). Para crear los cañones arrastraremos el sprite directamente al escenario y ajustaremos la posición, la escala y la rotación a los valores deseados.

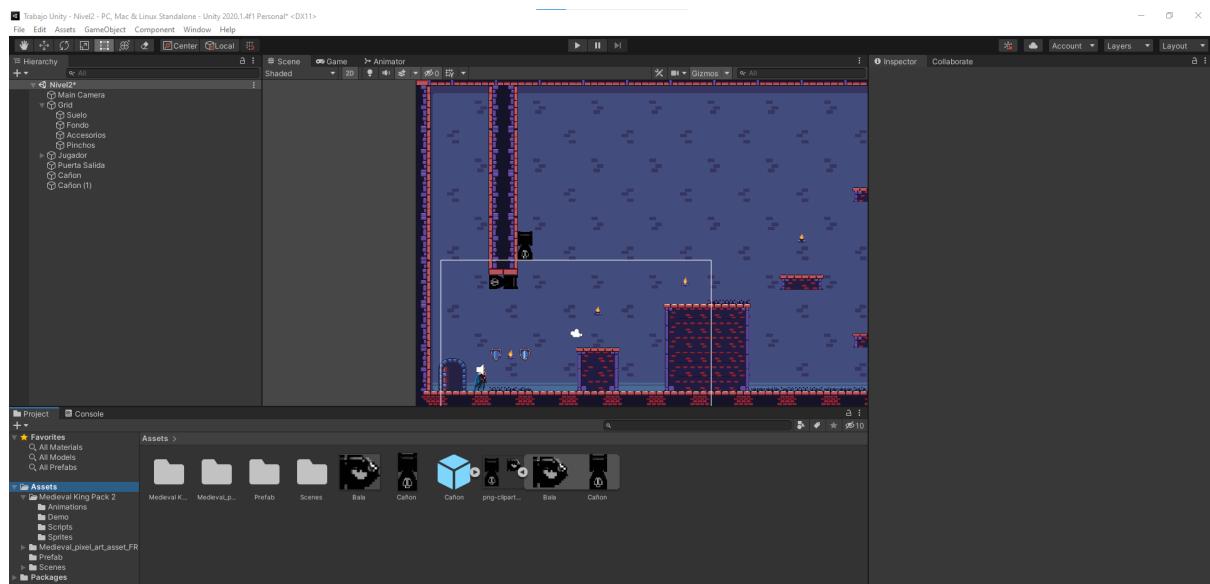


Figura 48. Nivel 2 con la infraestructura de los cañones

Ojo: No olvidar asociar a estos cañones un Box Collider para que detecte las colisiones con el Jugador aunque esto es poco probable que suceda.

Ahora, vamos a crear un nuevo GameObject de tipo Sprite llamado Bala que representarán las balas que disparan los cañones. Una vez hemos ajustado el tamaño de la bala, se creará un prefab de la bala (Figura 49).

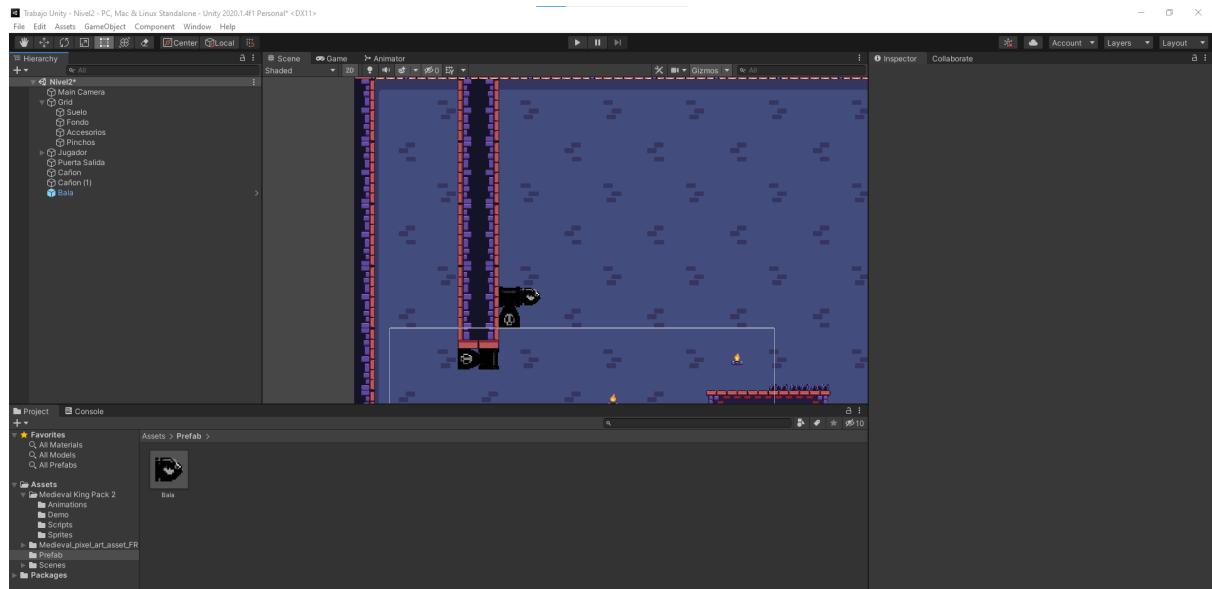


Figura 49.      **Prefab de la bala**

Para saber desde dónde tienen que salir las balas, se van a crear 2 sprites, de modo que queden ubicados al lado de los cañones. Cada bala se asignará como hijo de uno de los cañones. Una vez se tenga esto, se desactivaran ambos gameObejct ya que solo queremos estos gameObejct para marcar la posición de donde van a salir las balas (Figura 50).

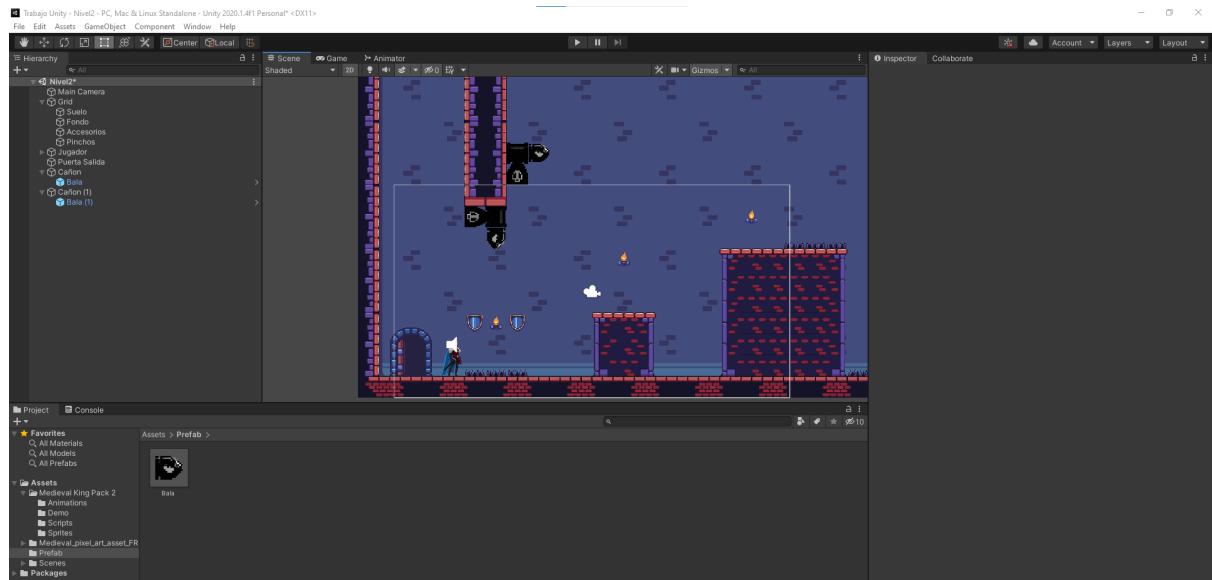


Figura 50.      **Sprites que marcan el sitio donde salen las balas**

En el prefab Bala que hemos creado anteriormente hay que añadir el componente RigidBody 2D para que la bala tenga masa y deshabilitar la gravedad para poder mover el

objeto por la escena asignando una velocidad. Además, también hay que añadir el componente Box Collider para que la bala detecte las colisiones y activar la opción de Is Trigger (Figura 51).

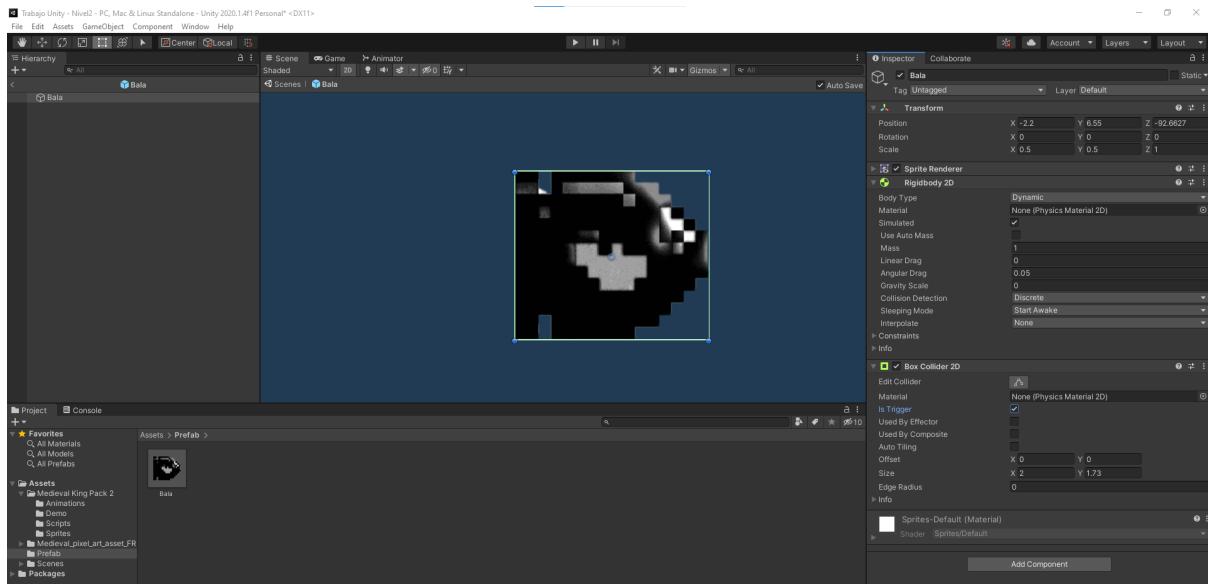


Figura 51.      **Prefab Bala**

Ahora, vamos a crear un script llamado instanciarBala el cual se encargará de crear instancias de las balas de manera periódica. El contenido del script será el siguiente:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class instanciarBala : MonoBehaviour
{
    protected Transform trans;
    public GameObject prefabBala;
    private IEnumerator corutina;

    // Use this for initialization
    void Start()
    {
        trans = GetComponent<Transform>();

        corutina = corutinaAvisiones(2);
        StartCoroutine(corutina);
    }

    // Update is called once per frame
    void Update()
```

```
{  
}  
  
}  
  
private IEnumerator coroutinaAvisiones(float waitTime)  
{  
    while (true)  
    {  
        if (trans.gameObject.tag == "Abajo")  
        {  
            Instantiate(prefabBala, new Vector2(trans.GetChild(0).position.x,  
trans.GetChild(0).position.y), Quaternion.Euler(0, 0, 270));  
        }  
        else  
        {  
            Instantiate(prefabBala, new Vector2(trans.GetChild(0).position.x,  
trans.GetChild(0).position.y), Quaternion.identity);  
        }  
        yield return new WaitForSeconds(waitTime);  
    }  
}
```

Básicamente, en este script se tiene una corrutina que nos va a generar balas automáticamente cada 2 segundos. La parte interesante del script es la orientación de las balas ya que si la bala es instancia desde el cañón que apunta hacia abajo, se rotará la bala para que “mire” hacia abajo. Para hacer este rotado, hemos añadido una etiqueta al cañón que apunta hacia abajo para saber que balas hay que rotar y cuáles no.

Una vez tenemos hecho el script, vamos a asociar este script a los 2 cañones que hemos añadido al escenario anteriormente. No olvidar de pasarle el prefab de la bala que hemos creado al script como parámetro para que el script utilice este prefab de la bala para realizar las instancias de las balas (Figura 52).

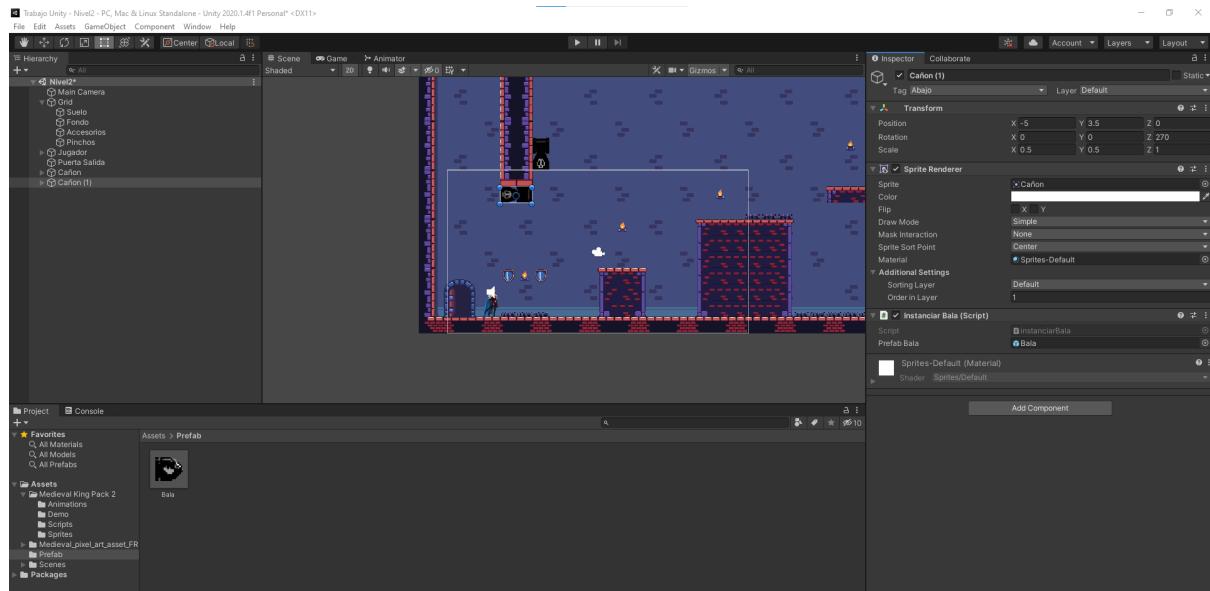


Figura 52. Asociando el script instanciarBala

El siguiente paso, es crear un script llamado movimientoBala que se encargue de gestionar el movimiento de la bal por el escenario así como las colisiones con nuestro personaje y con los muros del escenario. El contenido del script es el siguiente:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class movimientoBala: MonoBehaviour
{
    public float velocidad = 1.0f;
    protected Rigidbody2D rigid;
    protected static Transform trans;
    protected SpriteRenderer spriteRender;
    public static GameObject prefabBala;
```

```
// Use this for initialization
void Start()
{
    rigid = GetComponent<Rigidbody2D>();
    trans = GetComponent<Transform>();
    if (trans.rotation == Quaternion.identity)
    {
        rigid.velocity = new Vector2(velocidad, 0);
    }
    else
    {
```

```

        rigid.velocity = new Vector2(0, -velocidad);
    }
    spriteRender = GetComponent<SpriteRenderer>();
}

// Update is called once per frame
void Update()
{
    Vector2 vector2min = Camera.main.ViewportToWorldPoint(new Vector2(0, 0));
    Vector2 vector2max = Camera.main.ViewportToWorldPoint(new Vector2(1, 1)); //tengo en
cuanta el borde inferior del sprite
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Suelo")
    {
        Destroy(this.gameObject);
    }
}
}

```

En este script, en primer lugar, se analiza la rotación de la bala para saber en qué dirección hay que mover la bala. Una vez se conoce la rotación se mueve la bala en la dirección correspondiente.

Por otra parte, se tiene un evento OnTriggerEnter2D el cual se produce cuando la bala colisiona contra algo. Al activar el parámetro de Is Trigger en el Collider de la Bala, la Bala en vez de simular la colisión, activa este evento. De esta manera, si la bala colisiona contra algún muro del escenario, se destruirá directamente la bala.

Para que este código funcione, no hay que olvidarse de acceder al Tilemap Suelo y asignarle una etiqueta llamada “Suelo”.

Una vez se tiene hecho el script del movimiento de la Bala, asignaremos este Script al prefab de la bala (para que todas las balas del juego utilicen este Script) y asignaremos el componente de velocidad que nosotros consideramos conveniente para el juego. En nuestro caso, hemos elegido una velocidad de 8 (Figura 53).

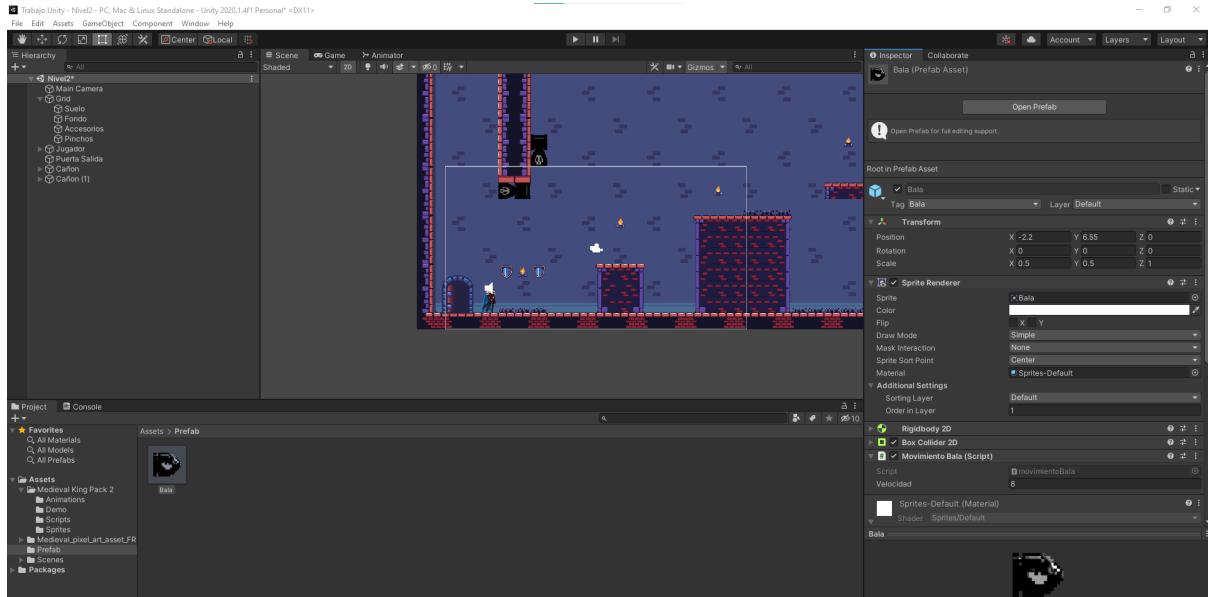


Figura 53. Asociando el script movimientoBala al prefab de la Bala

Por último, nos queda realizar unos pequeños cambios al script del Jugador ya que queremos que las balas cuando impacten al jugador, maten a este. Además queremos que las balas desaparezcan cuando impacten al jugador. Para ello vamos a introducir las siguientes modificaciones en el script del Jugador:

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Bala")
    {
        Animator animator = GetComponent<Animator>();
        animator.SetTrigger("morir");
        Destroy(other.gameObject);
    }
}
```

La modificación consiste en la introducción de un evento OnTriggerEnter2D de forma que cuando una bala impacte con nuestro jugador, se active el disparador de morir (lo que desencadena en la muerte del personaje). Después de activar el disparador de muerte, se destruirá el objeto bala.

Para que este código funcione, no hay que olvidarse de acceder al prefab de la bala y asignarle una etiqueta llamada “Bala”.

De manera adicional, vamos a añadir un efecto de sonido para cuando se disparan las balas. Para ello, nos vamos a descargar un sonido de bala en formato mp3 y lo vamos a importar a nuestro proyecto. Una vez hecho esto, vamos a crear un componente Audio Source en ambos cañones en el que vamos a asociar el sonido de nuestro audio (Figura 54) y realizaremos los cambios necesarios en el script instanciarBala para que se reproduzca este efecto de sonido.

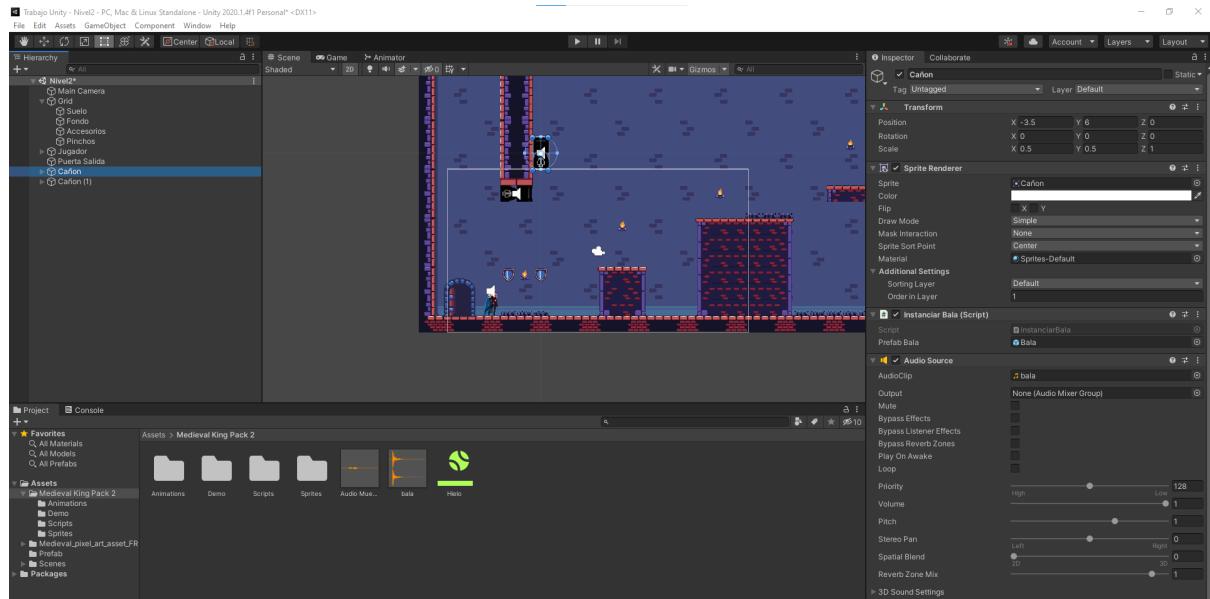


Figura 54. Añadiendo efectos de sonido bala

No olvidar de desactivar el parámetro Play on Awake para que el sonido solo se reproduzca cuando nosotros lo hayamos indicado y no al principio.

Los cambios realizados sobre el script instanciarBala son los siguientes:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class instanciarBala : MonoBehaviour
{
    protected Transform trans;
    public GameObject prefabBala;
    private IEnumerator corutina;
     AudioSource audioSource;

    // Use this for initialization
    void Start()
    {
        trans = GetComponent<Transform>();
         AudioSource audioSource = GetComponent<

        corutina = corutinaAvisiones(2);
        StartCoroutine(corutina);
    }
}

```

```

// Update is called once per frame
void Update()
{
}

private IEnumerator coroutinaAvisiones(float waitTime)
{
    while (true)
    {
        if (trans.gameObject.tag == "Abajo")
        {
            audioSource.Play();
            Instantiate(prefabBala, new Vector2(trans.GetChild(0).position.x,
trans.GetChild(0).position.y), Quaternion.Euler(0, 0, 270));
        }
        else
        {
            audioSource.Play();
            Instantiate(prefabBala, new Vector2(trans.GetChild(0).position.x,
trans.GetChild(0).position.y), Quaternion.identity);
        }
        yield return new WaitForSeconds(waitTime);
    }
}
}

```

Con esto, ya tendríamos el nivel 2 del juego totalmente operativo. Como último paso, vamos a cambiar el parámetro del script cambioNivel para que podamos cambiar al nivel 3.

## Nivel 3

En los 2 niveles anteriores hemos creado una infraestructura, un personaje que se mueve por el escenario y varios obstáculos (pinchos y balas). Para este nivel 3, vamos a partir de la infraestructura del nivel 2 y vamos a añadir un par de sierras que se van a ir moviendo por el escenario siguiendo un patrón repetitivo y rectilíneo. Con esto agregamos un grado de dificultad más alto a este nivel.

Para poder hacer esto, lo primero que vamos a hacer es buscar en internet una imagen de una sierra, descargarla e importarla a nuestro proyecto. Al importar la imagen de la sierra, el proyecto nos quedará en forma de sprite (Figura 55). A diferencia del nivel 2, no tenemos que recortar nada ya que el sprite contiene únicamente la sierra que es lo que nos interesa.

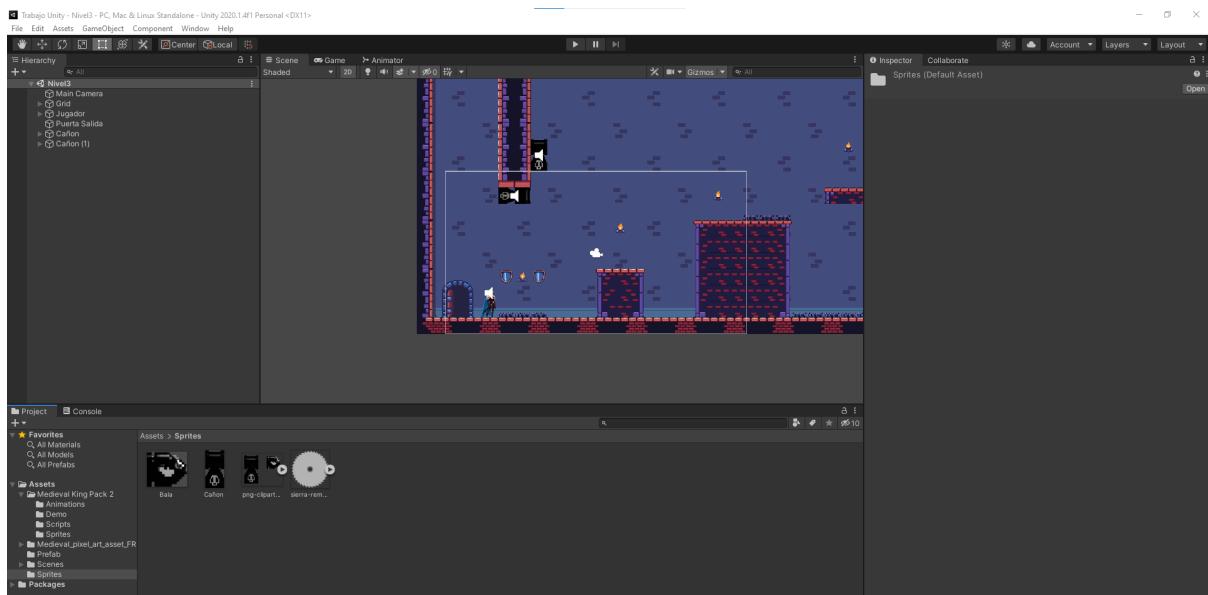


Figura 55. Sprite Sierra

Ahora vamos a arrastrar la sierra en el escenario, y vamos a añadir a la sierra un componente Rigidbody 2D para que la bala tenga masa y deshabilitar la gravedad para poder mover el objeto por la escena asignando una velocidad. Además, también hay que añadir el componente Circle Collider para que la sierra detecte las colisiones y activar la opción de Is Trigger (Figura 56).

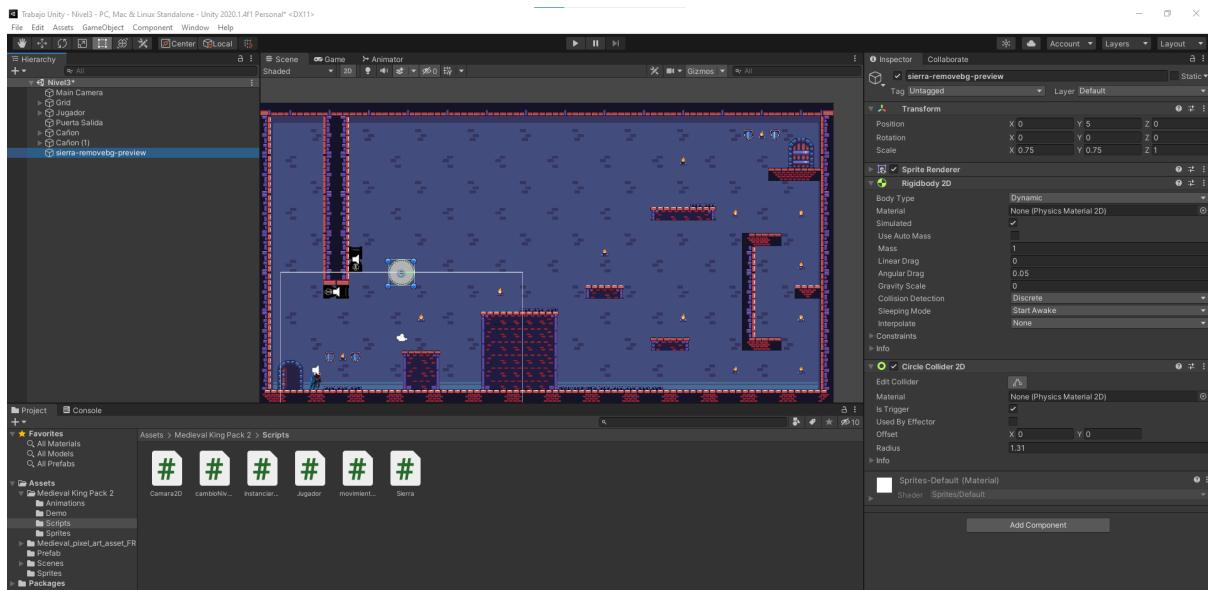


Figura 56. Sprite Sierra

Ahora el siguiente paso es crear un script llamado Sierra que gestione los movimientos de las sierras y las colisiones con los personajes. En nuestro caso, vamos a querer mover 2 sierras en el escenario de forma diferente, por lo que en el script se diferenciarán ambas sierras por su nombre. El script que va a gestionar el movimiento de las sierra como sus colisiones con el personaje será el siguiente:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Sierra : MonoBehaviour
{
    public float velocidad = 1.0f;
    protected Rigidbody2D rigid;
    protected static Transform trans;
    protected SpriteRenderer spriteRender;
    public GameObject sierra;

    // Use this for initialization
    void Start()
    {
        rigid = GetComponent<Rigidbody2D>();
        trans = GetComponent<Transform>();
        spriteRender = GetComponent<SpriteRenderer>();

        if (sierra.name.StartsWith("Sierra2"))
        {
            rigid.velocity = new Vector2(0, velocidad);
        }
        else
        {
            rigid.velocity = new Vector2(velocidad, 0);
        }
    }

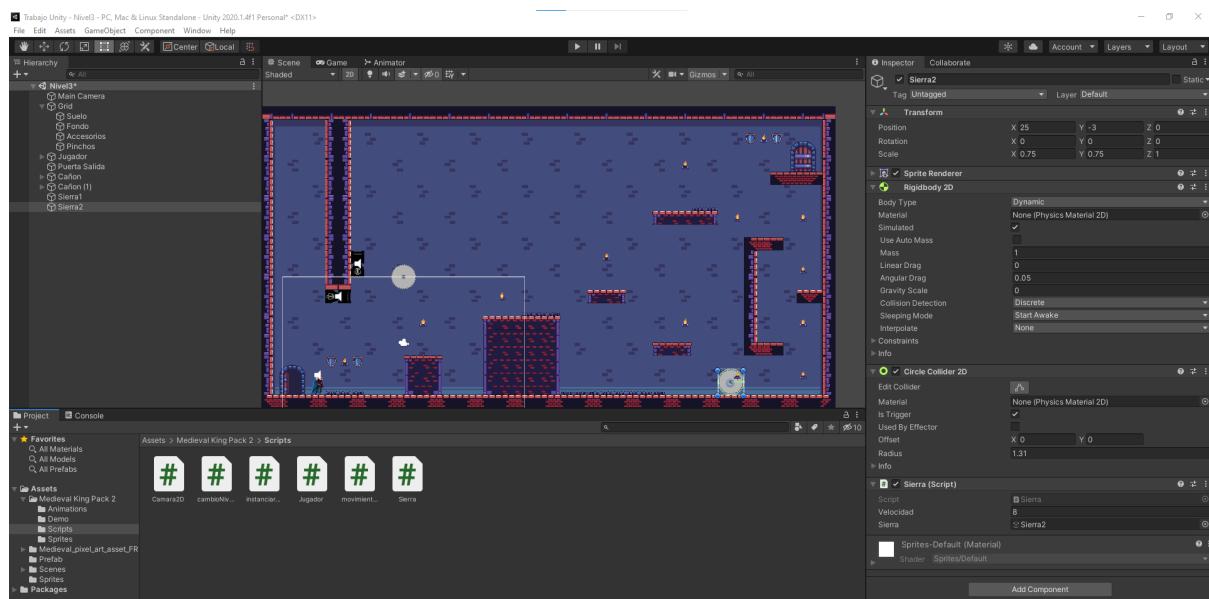
    // Update is called once per frame
    void Update()
    {
        if (sierra.name.StartsWith("Sierra2"))
        {
            if (sierra.transform.position.y >= 7)
            {
                rigid.velocity = new Vector2(0, -velocidad);
            }
            if (sierra.transform.position.y <= -3)
            {
                rigid.velocity = new Vector2(0, velocidad);
            }
        }
        else
    }
}

```

```
{  
    if (sierra.transform.position.x >= 24)  
    {  
        rigid.velocity = new Vector2(-velocidad, 0);  
    }  
    if (sierra.transform.position.x <= 0)  
    {  
        rigid.velocity = new Vector2(velocidad, 0);  
    }  
}  
}
```

Básicamente con este código, cuando se llega a un límite superior o inferior se cambia el sentido en el que se mueve la sierra. Como tenemos 2 sierras que se mueven en direcciones diferentes, tenemos que distinguir el código de movimiento de una del código de movimiento de la otra. Para diferenciar correctamente las dos sierras, vamos a pasar al script el gameObject de cada sierra directamente.

Por último, vamos a colocar los dos objetos sierra en las posiciones deseadas y con el tamaño deseado y a ambas sierras les vamos a asociar el script de Sierra y darle una velocidad de 8 (Figura 57) además de pasar el gameObject de la sierra correspondiente como parámetro.



**Figura 57.** **Sprite Sierra**

Ahora, al igual que hicimos con las balas, tenemos que realizar una pequeña modificación en el script del jugador, para que este se muera cuando se produzca una colisión con

alguna de las sierras. Esta modificación consiste en agregar lo siguiente en el método OnTriggerEnter2D:

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Bala")
    {
        Animator animator = GetComponent<Animator>();
        animator.SetTrigger("morir");
        Destroy(other.gameObject);
    }

    if (other.gameObject.name.StartsWith("Sierra"))
    {
        animator.SetTrigger("morir");
    }
}
```

Al igual que hicimos con las balas, como los componentes Circle Collider de las sierras tiene el parámetro Is Trigger activado, cada vez que se produzca una colisión con estas se llamará al método OnTriggerEnter2D. De esta manera, cada vez que se detectan colisiones de la sierra con el jugador, se llamará a este método y se activará el disparador morir, lo que desencadena en la muerte del personaje.

Es extraño que en el juego las sierras vayan flotando por el escenario, para eliminar esta percepción de que las sierras van flotando, vamos a crear un par de líneas que describan el rango de movimiento de las sierras (Create→Effects→Line). Una vez hayamos creado las líneas, a través de la propiedad Positions, podemos definir las coordenadas de cada uno de los dos puntos de la línea. De esta manera, podremos colocar las líneas en la posición deseada. Por último, le daremos un color blanco a las líneas y en la propiedad Order In Layer le daremos un 1 (al igual que las sierras, cosa que olvide mencionar). El resultado es el siguiente (Figura 58):

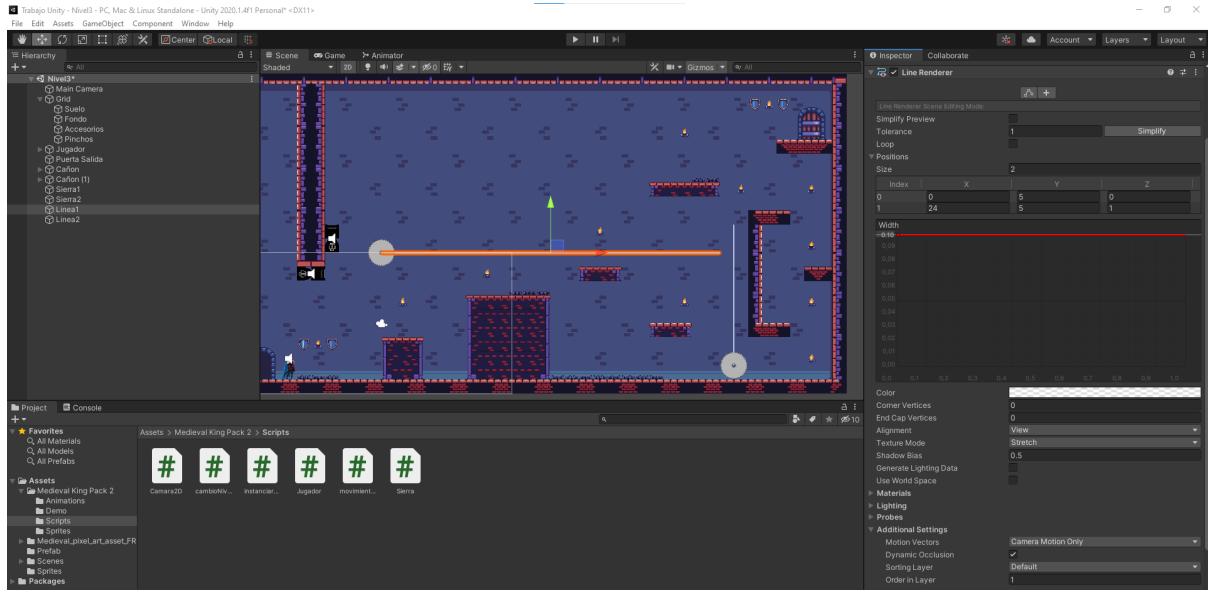


Figura 58. Líneas que describen el movimiento de las sierras

Para mejorar la decoración de las líneas, vamos a crear unos puntos o círculos en el principio y final de la línea para delimitar bien cual es la trayectoria de la sierra. Para ello, en primer lugar creamos un círculo Assets→Create→Sprite→Circle. Una vez hemos creado el sprite arrastramos los círculos en las posiciones deseadas (principio y fin de las líneas) y cambiamos las dimensiones y color de estos círculos al gusto. Por último, no olvidar de cambiar la propiedad Order in Layer a 1 para todos los círculos (Figura 59).

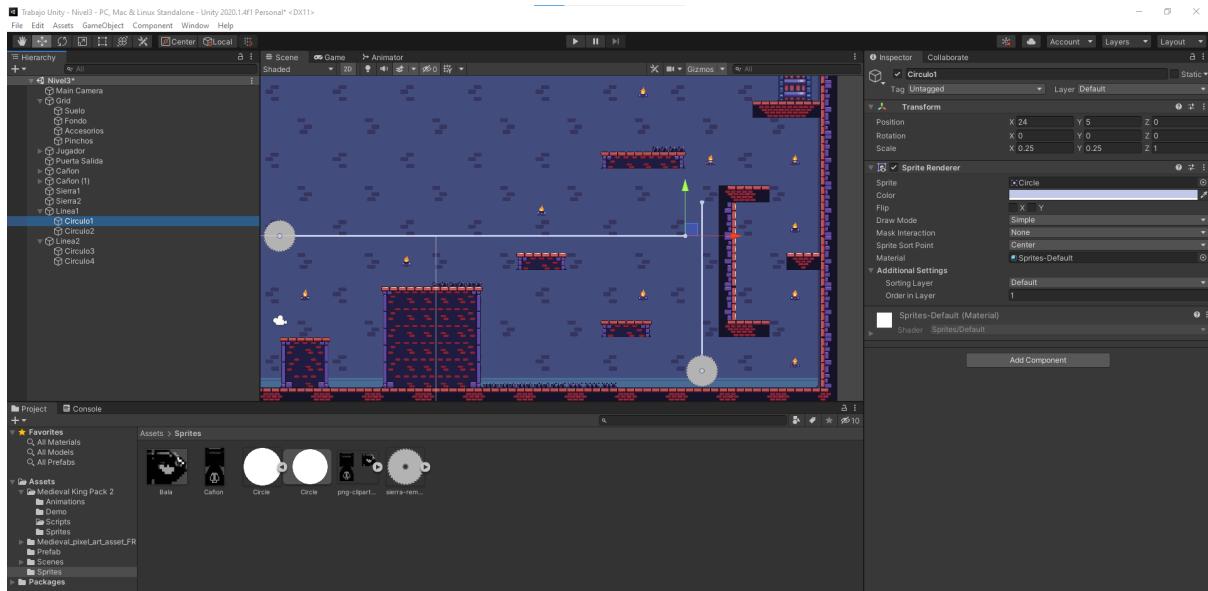


Figura 59. Líneas con sus respectivos círculos

En nuestro caso, para mejorar la manejabilidad de las líneas, hemos añadido estos círculos como hijos de las líneas.

De manera adicional, vamos a añadir un efecto de sonido para cuando el personaje choque con la sierra. Para ello, nos vamos a descargar un sonido de sierra en formato mp3 y lo

vamos a importar a nuestro proyecto. Una vez hecho esto, vamos a crear un componente Audio Source en las 2 sierras en el que vamos a asociar el sonido de nuestro audio (Figura 60) y realizaremos los cambios necesarios en el script Sierra para que se reproduzca este efecto de sonido.

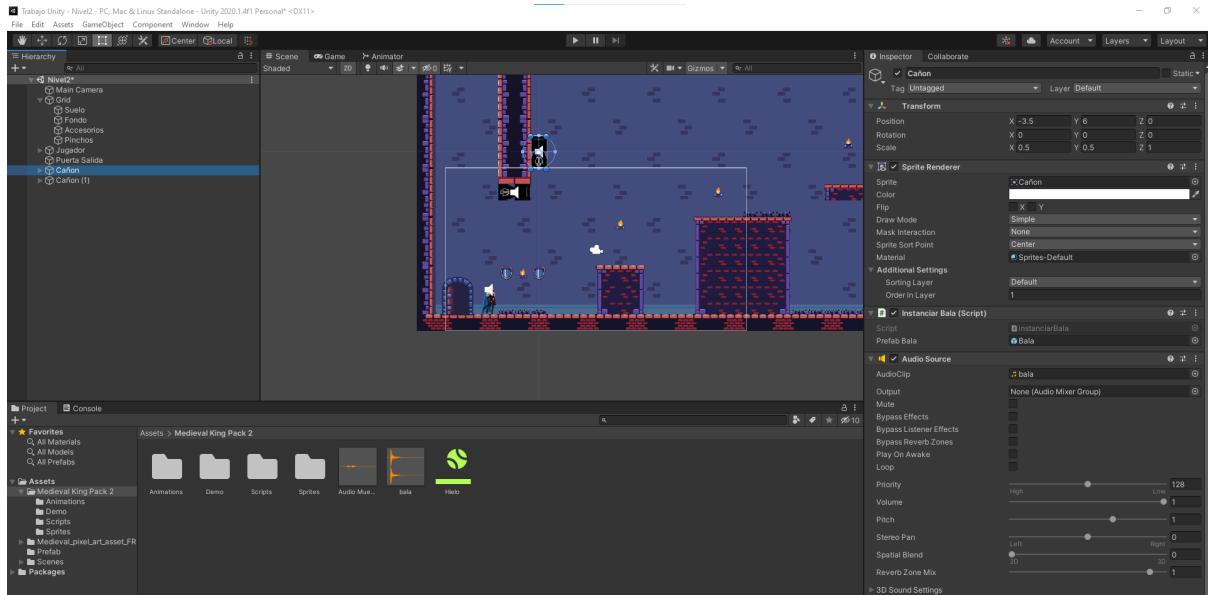


Figura 60. Añadiendo efectos de sonido bala

No olvidar de desactivar el parámetro Play on Awake para que el sonido solo se reproduzca cuando nosotros lo hayamos indicado y no al principio.

Los cambios realizados sobre el script Sierra son los siguientes:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Sierra : MonoBehaviour
{
    public float velocidad = 1.0f;
    protected Rigidbody2D rigid;
    protected static Transform trans;
    protected SpriteRenderer spriteRender;
    public GameObject sierra;
    AudioSource audioSource;

    // Use this for initialization
    void Start()
    {
        rigid = GetComponent<Rigidbody2D>();
        trans = GetComponent<Transform>();
```

```

spriteRender = GetComponent<SpriteRenderer>();
audioSource = GetComponent< AudioSource >();

if (sierra.name.StartsWith("Sierra2"))
{
    rigid.velocity = new Vector2(0, velocidad);
}
else
{
    rigid.velocity = new Vector2(velocidad, 0);
}

// Update is called once per frame
void Update()
{
    if (sierra.name.StartsWith("Sierra2"))
    {
        if (sierra.transform.position.y >= 7)
        {
            rigid.velocity = new Vector2(0, -velocidad);
        }
        if (sierra.transform.position.y <= -3)
        {
            rigid.velocity = new Vector2(0, velocidad);
        }
    }
    else
    {
        if (sierra.transform.position.x >= 24)
        {
            rigid.velocity = new Vector2(-velocidad, 0);
        }
        if (sierra.transform.position.x <= 0)
        {
            rigid.velocity = new Vector2(velocidad, 0);
        }
    }
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.name.StartsWith("Jugador"))
    {
        audioSource.Play();
    }
}

```

```

        }
    }
}

```

Con esto, ya tendríamos el nivel 3 del juego totalmente operativo. Como último paso, vamos a cambiar el parámetro del script cambioNivel para que podamos cambiar al nivel 4.

## Nivel 4

En los 3 niveles anteriores hemos creado una infraestructura, un personaje que se mueve por el escenario y varios obstáculos (pinchos y balas). Para este nivel 4, vamos a partir de la infraestructura del nivel 3 y añadiremos un componente más de dificultad. Este componente será la utilización de plataformas con “gravedad”, es decir, plataformas que se caigan una vez el Jugador se ha apoyado sobre ellas. Para este nivel sólo se establecerán un par de plataformas con esta mecánica ya que la complejidad del nivel puede ser demasiado elevada si se establecen todas las plataformas con esta mecánica.

Para poder hacer esto, lo primero que vamos a hacer es acceder al tilemap Suelo y a través de la paleta correspondiente, vamos a eliminar las plataformas a las cuales queremos cambiar su mecánica (Figura 61). Eso se hace porque nos interesa gestionar esta par de plataformas por separado para simular la “gravedad” de las plataformas.

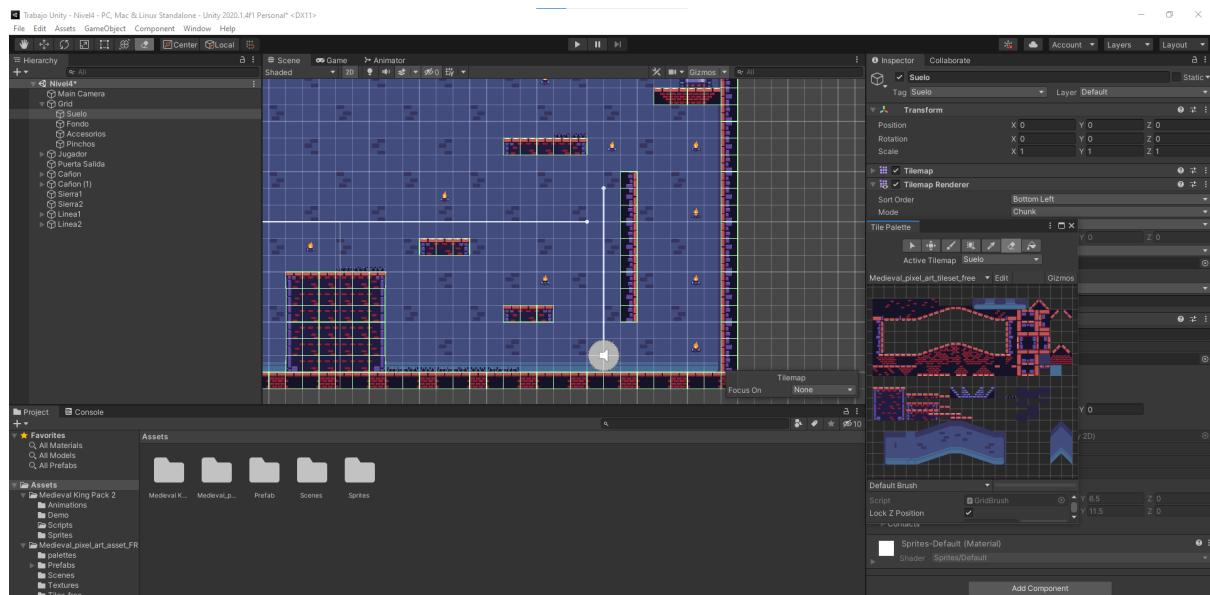


Figura 61. Redefiniendo las plataformas del escenario

En segundo lugar, vamos a acceder al sprite compuesto que contiene todos los materiales del escenario y vamos a extraer un sprite que representa a nuestra plataforma “gravitatoria” (Figura 62).

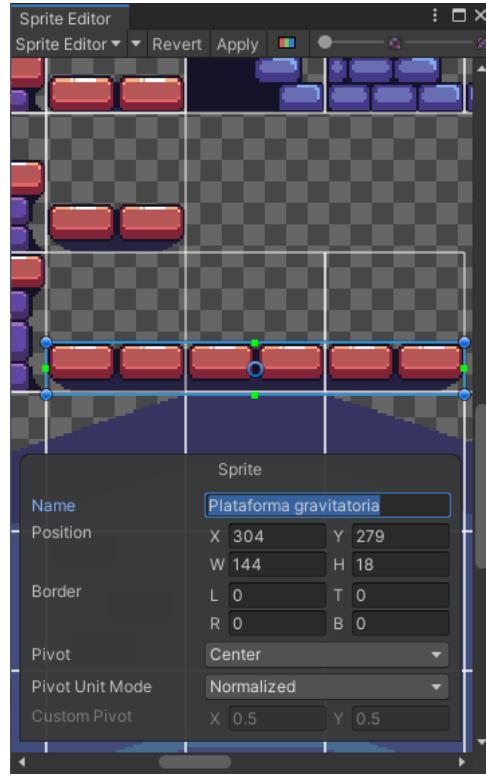


Figura 62. Extrayendo el sprite de la plataforma “gravitatoria”

Ahora vamos a arrastrar esta plataformas “gravitatoria” al escenario y le vamos a asociar un Box Collider para detectar las colisiones con el Jugador al igual que lo haría una plataforma normal y un rigidBody 2D para asociar una masa a la plataforma cuando el Jugador se apoye en ella (Figura 63). Inicialmente la propiedad Gravity Scale del Rigidbody será de 0 para que no se caiga por sí misma. La idea es cambiar la gravedad de la plataforma cuando el personaje se apoye en ella.

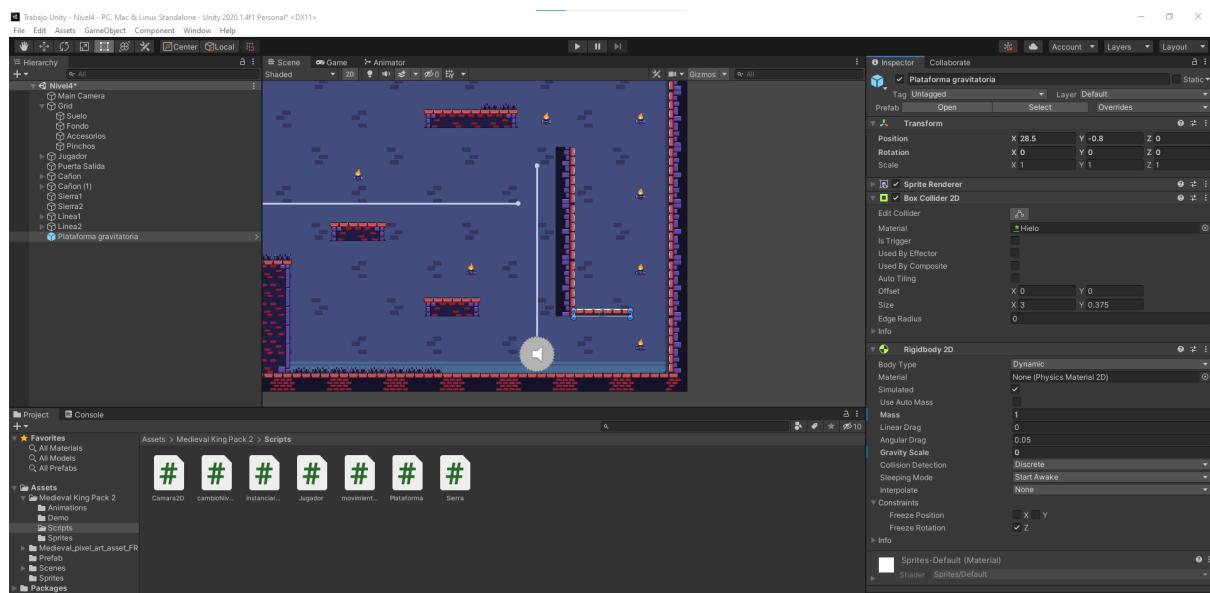


Figura 63. Estableciendo las propiedades de la plataforma “gravitatoria”

No olvidar se asociar nuestro material de hielo al componente Collider al igual que hicimos con las demás plataformas en su momento. En este caso, el parámetro Order In Layer de esta plataforma será 0 (valor por defecto) al igual que el resto del suelo.

También vamos a habilitar las opciones Freeze Rotation Z y Freeze Rotation X para que las plataformas no giren sobre sí mismas o se vayan a la izquierda o derecha cuando el Jugador colisione con ellas. Solo no interesa que las plataformas se muevan en el eje Y. (Nota: En la figura 63 olvide congelar las rotaciones para el eje Y).

Una vez hecho esto, vamos a crear un script llamado Plataforma que gestione esta gravedad de las plataformas. El contenido del script será el siguiente:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Plataforma : MonoBehaviour
{
    protected Rigidbody2D rigid;
    protected static Transform trans;
    protected Collider2D collider2d;
    protected SpriteRenderer spriteRender;

    // Start is called before the first frame update
    void Start()
    {
        rigid = GetComponent<Rigidbody2D>();
        trans = GetComponent<Transform>();
        collider2d = GetComponent<Collider2D>();
        spriteRender = GetComponent<SpriteRenderer>();
    }

    // Update is called once per frame
    void Update()
    {
        if (spriteRender.bounds.min.y < -4)
            Destroy(this.gameObject);
    }

    void OnCollisionStay2D(Collision2D collision)
    {
        if (collision.gameObject.name.StartsWith("Jugador"))
        {
            rigid.gravityScale = 0.8f;
            collider2d.isTrigger = true;
        }
    }
}
```

```

    }
}

}

```

Básicamente, en este código cuando el jugador se apoye en la plataforma, se ejecutará el método OnCollisionStay2D de forma que se dotará de gravedad a la plataforma y esta caerá. También se establece el collider de la plataforma en Is Trigger para que en vez de rebotar contra el suelo, poder atravesarlo. Una vez la plataforma atraviesa el suelo, se destruirá para que no quede perdida por el juego.

Lo último que nos queda es asociar el Script Plataforma a la plataforma. Ahora, de la misma manera que hicimos con las balas, vamos a crear un prefab de esta plataforma ya que nos interesa que todos las plataformas tengan la misma mecánica. Con esto podremos crear otras 2 plataformas con las mismas propiedades y colocarlas en el escenario como nosotros queramos. Al final el resultado será el siguiente (Figura 64):

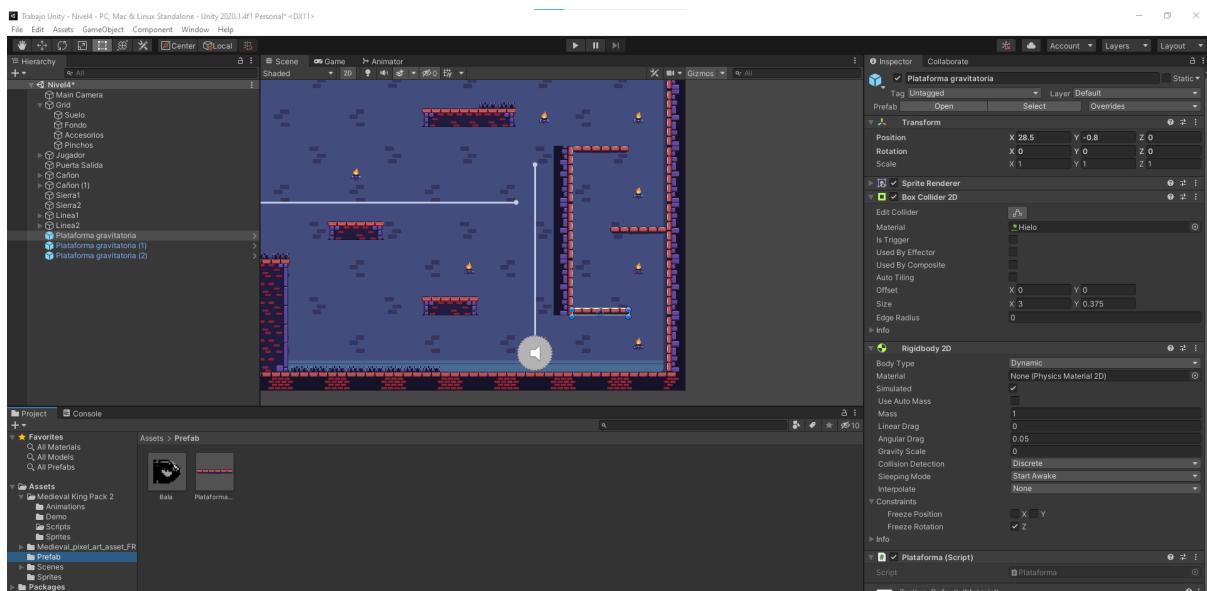


Figura 64. Estableciendo las propiedades de la plataforma “gravitatoria”

Sin embargo, nos queda un aspecto muy importante que gestionar. Imaginemos el caso en el que el Jugador muere después de superar las plataformas gravitatorias. Cuando se restablezca la posición del Jugador al inicio del nivel y el Jugador vuelva a la parte del mapa donde se encuentran las plataformas gravitatorias, estas ya no estarán por lo que no podrá superar el nivel.

Debido a este problema, vamos a implantar una corutina que se ejecute cada 10 segundos y se encargue de restablecer las plataformas a su posición original con sus propiedades originales. Para ello, vamos a realizar las siguientes modificaciones en el script Plataforma:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

```

```

public class Plataforma : MonoBehaviour
{
    protected Rigidbody2D rigid;
    protected Transform trans;
    protected Collider2D collider2d;
    protected SpriteRenderer spriteRender;
    private IEnumerator corutina;
    private Vector3 posicionInicial;

    // Start is called before the first frame update
    void Start()
    {
        rigid = GetComponent<Rigidbody2D>();
        trans = GetComponent<Transform>();
        collider2d = GetComponent<Collider2D>();
        spriteRender = GetComponent<SpriteRenderer>();

        posicionInicial = trans.position;

        corutina = corutinaAvisiones(10);
        StartCoroutine(corutina);
    }

    // Update is called once per frame
    void Update()
    {
        if (spriteRender.bounds.min.y < -4)
        {
            rigid.gravityScale = 0f;
            rigid.velocity = new Vector2(0, 0);
            //Destroy(this.gameObject);
        }
    }

    void OnCollisionStay2D(Collision2D collision)
    {
        if (collision.gameObject.name.StartsWith("Jugador"))
        {
            rigid.gravityScale = 0.8f;
            collider2d.isTrigger = true;
        }
    }

    private IEnumerator corutinaAvisiones(float waitTime)

```

```

    {
        while (true)
        {
            trans.position = posicionInicial;
            collider2d.isTrigger = false;
            yield return new WaitForSeconds(waitTime);
        }
    }
}

```

Como se puede observar, al iniciar el script se almacena la posición actual de la plataforma y en la corutina, cada 10 segundos se establece la posición de las plataformas en la posición inicial. Obsérvese como ahora, en vez de destruir la plataforma cuando llega al suelo, se frena en esta posición y cuando se ejecute la corutina se coloca de nuevo en su posición.

Con esto, ya tendríamos el nivel 4 del juego totalmente operativo. Como último paso, vamos a cambiar el parámetro del script cambioNivel para que podamos cambiar al nivel 5.<

## Nivel 5

Realmente el juego consta de 4 niveles pero al completar el nivel 4 queremos que se nos lleve a una escena final (llamada nivel 5 para poder reutilizar el script de cambio de nivel) en la que se nos dará la enhorabuena por acabar el juego. Este nivel por lo tanto estará formado por un texto de enhorabuena y una foto decorativa.

Para crear la imagen de fondo utilizaremos la opción Create→UI→Image y asociaremos la foto correspondiente. En nuestro caso, vamos a importar a nuestro proyecto una imagen de nuestro jugador y esta será la foto que vayamos a asociar al componente Image (Figura 65).

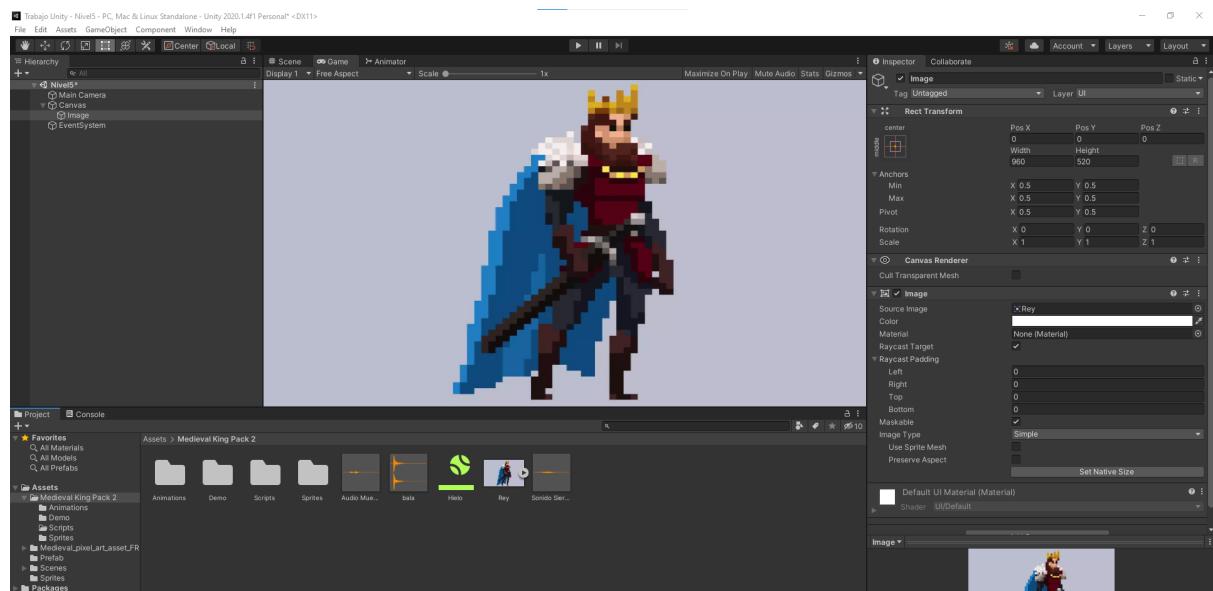


Figura 65. Imagen de fondo

Para crear el texto de enhorabuena utilizaremos la opción Create→UI→Text y añadiremos el texto de enhorabuena. En nuestro caso, el texto sera de color amarillo negrita, tendra tamaño 80 y estara centrado en el medio de la pantalla (Figura 66).

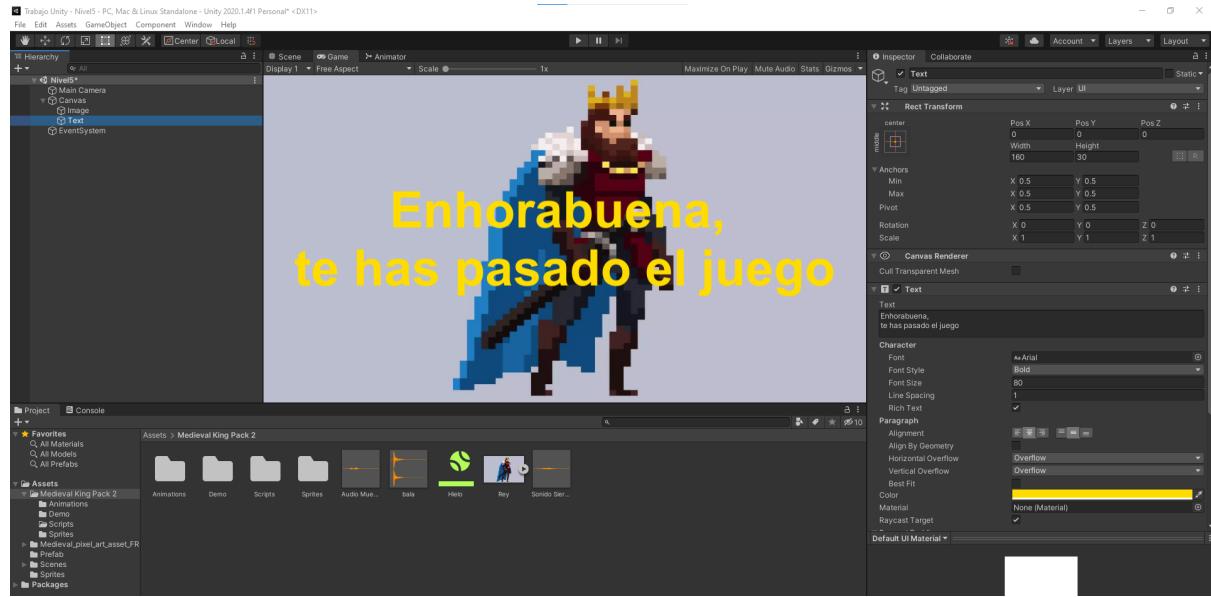


Figura 66. Texto de enhorabuena

## Aspectos adicionales

### Creación de la música de fondo del juego

A lo largo del desarrollo del juego se han introducido sonidos para los diferentes efectos del juego. Una vez hemos desarrollado el juego por completo y tenemos todos los niveles, ahora vamos a asociar una música de fondo para el juego la cual se ejecuta en el transcurso de los distintos niveles. Hemos descargado 4 canciones (una por nivel).

Para que las distintas canciones se reproduzcan al arrancar el nivel simplemente tenemos que asociar un componente Audio Source a cualquier objeto que permanezca siempre en el nivel (como la cámara por ejemplo) y en este caso sí nos interesa mantener activado el parámetro Play on Awake para que la música se reproduzca al empezar el nivel (Figura 67).

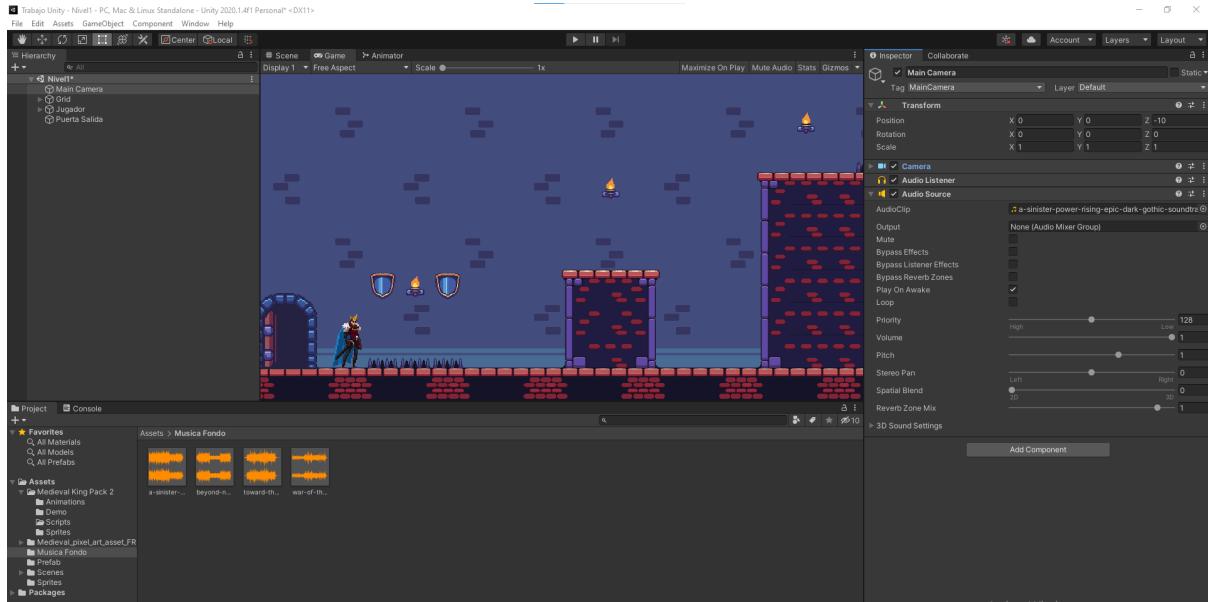


Figura 67. **Musica de fondo - Nivel 1**

Para el resto de niveles se hará de manera similar pero utilizando un audio distinto para así tener música distinta para cada nivel.

## Creación de un contador de muertes

Por otro lado, hemos decidido que es interesante llevar un pequeño contador de las muertes que lleva nuestro jugador y mostrarlas en todo momento en una esquina de la pantalla. También hemos decidido mostrar en el nivel final el número total de muertes a modo de “puntuación” siendo el objetivo que el número de muertes sea el más bajo posible.

Para ello, vamos a crear un Texto en todos los niveles (Create→UI→Text) que indique el número de muertes y lo colocaremos en la esquina superior izquierda. Elegimos para el texto un color que presente un buen contraste con el fondo del juego para que se pueda leer con facilidad (Figura 68).

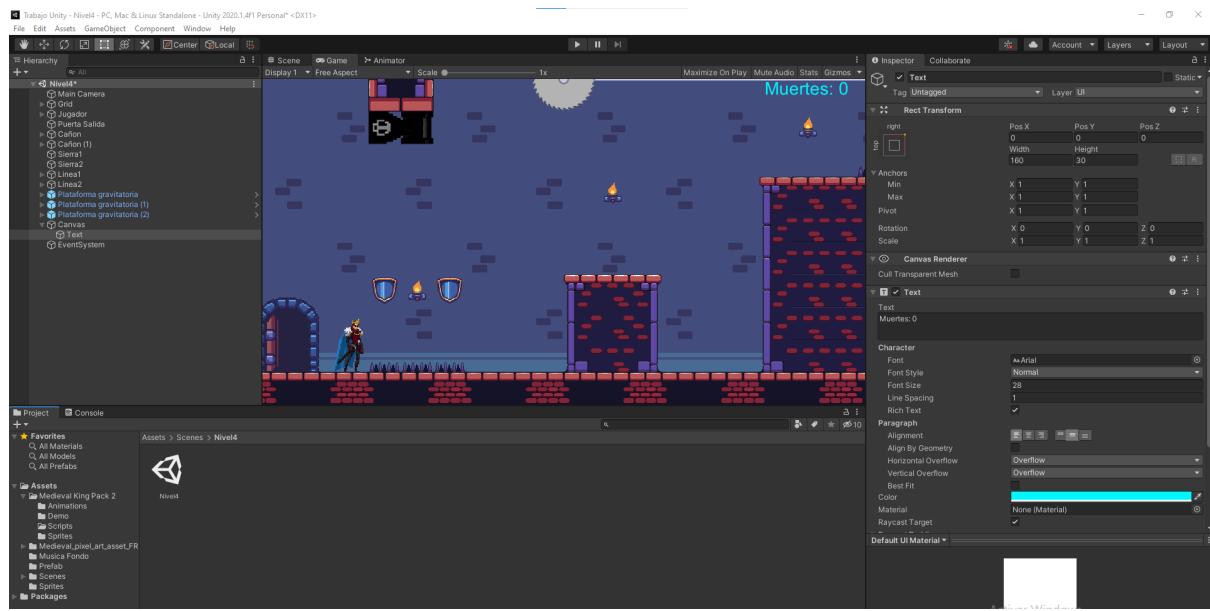


Figura 68. **Texto de muertes**

El siguiente paso es crear un script modelo llamado Muertes que controle y almacene el número de muertes totales del jugador. El código de este Script será el siguiente:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Muertes : MonoBehaviour
{
    protected static int muertes = 0;
    public static int pMuertes
    {
        get
        {
            return muertes;
        }
        set
        {
            muertes = value;
        }
    }
}
```

El siguiente paso es añadir unas modificaciones al script del jugador de forma que cuando muera el personaje, se incremente el valor de muertes en una unidad y se refresque este cambio en el texto de la esquina superior izquierda. Para ello realizaremos las siguientes modificaciones:

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class Jugador : MonoBehaviour
{
    public float velocidad = 5f;
    public float salto = 15f;
    protected Rigidbody2D rigidBody2D;
    protected Animator animator;
    protected Collider2D collider2d;
    public int dobleSalto;
    AudioSource audioSource;
    private Vector3 posicionInicial;
    public GameObject text;

    // Use this for initialization
    void Start()
    {
        rigidBody2D = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        collider2d = GetComponent<Collider2D>();
        audioSource = GetComponent<2;
        posicionInicial = transform.position;
        text = GameObject.Find("Text");
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.UpArrow) && dobleSalto > 0)
        {
            Debug.Log(rigidBody2D.velocity.y);
            rigidBody2D.AddForce(new Vector2(0, (salto * 2) - rigidBody2D.velocity.y),
ForceMode2D.Impulse);
            Debug.Log("Salto " + Time.time);
            dobleSalto = dobleSalto - 1;
        }
    }

    void FixedUpdate()

```

```

{
    rigidBody2D.velocity = new Vector2(velocidad * Input.GetAxis("Horizontal"),
    rigidBody2D.velocity.y);
    animator.SetBool("correr", rigidBody2D.velocity.x != 0);

    //roto al personaje para es lo escalo a -1 o lo giro 180 en el eje y
    if (rigidBody2D.velocity.x < 0)
        transform.rotation = Quaternion.Euler(0, 180, 0);
    else if (rigidBody2D.velocity.x > 0)
        transform.rotation = Quaternion.Euler(0, 0, 0);

    //saltar arriba
    if (rigidBody2D.velocity.y > 0 && Physics2D.OverlapBox(transform.GetChild(0).position, new
    Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) == null)
    {
        animator.SetBool("saltar", true);
        if (animator.GetBool("caer"))
        {
            animator.SetBool("caer", false);
        }
    }

    //cayendo
    else if (rigidBody2D.velocity.y < 0 && animator.GetBool("saltar"))
    {
        animator.SetBool("saltar", false);
        animator.SetBool("caer", true);
    }

    //toco tierra
    else if (animator.GetBool("caer") && Physics2D.OverlapBox(transform.GetChild(0).position,
    new Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
    {
        animator.SetBool("saltar", false);
        animator.SetBool("caer", false);
        animator.SetTrigger("caerTierra");
        dobleSalto = 2;
    }

    //el overlapbox se hace la mitad del collider para evitar que roce en lateral con una plataforma y
    me deje saltar de nuevo, (collider2d.bounds.max.x - collider2d.bounds.min.x) / 2
    if (Physics2D.OverlapBox(transform.GetChild(0).position, new
    Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
    {
}

```

```

//Debug.Log("SAlto " + Time.time);
if (Input.GetAxis("Jump") != 0)
{
    rigidBody2D.AddForce(new Vector2(0, salto), ForceMode2D.Impulse);
    Debug.Log("SAlto " + Time.time);
}
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Pinchos")
    {
        audioSource.Play();
        animator.SetTrigger("morir");
    }
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Bala")
    {
        Animator animator = GetComponent<Animator>();
        animator.SetTrigger("morir");
        Destroy(other.gameObject);
    }

    if (other.gameObject.name.StartsWith("Sierra"))
    {
        animator.SetTrigger("morir");
    }
}

public void recolocar()
{
    animator.SetTrigger("reinicio");
    rigidBody2D.velocity = Vector3.zero;
    transform.position = posicionInicial;

    Muertes.pMuertes += 1;
    text.GetComponent<Text>().text = "Muertes: " + Muertes.pMuertes;
}
}

```

Con estas modificaciones, básicamente cada vez que muere el personaje se actualiza el valor de muertes del modelo (Script Muertes) y se actualiza el valor del texto de la esquina con las muertes correspondientes.

Por último, nos colocamos en el nivel 5 y añadiremos otro cuadro de texto que indique las muertes totales del Jugador en el juego. Este cuadro de texto se colocará en la esquina inferior derecha y tendrá un color rojo intenso para que exista un buen contraste con el fondo y se pueda leer sin problemas (Figura 69).



Figura 69. Muertes - Final Juego

Por último, se creará un pequeño script llamado Final que se encargará de acceder al modelo Muertes y mostrar por la pantalla el número de muertes total del Jugador. El contenido de este script será el siguiente:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Final : MonoBehaviour
{
    public Text text;

    // Use this for initialization
    void Start()
    {
        text.text = "Muertes: " + Muertes.pMuertes;
    }
}
```

```
// Update is called once per frame
void Update()
{
}
```

Este script se asociará a un gameObject vacío que crearemos con el único objetivo de que ejecute este script Final (Figura 70). A este script se le pasará como parámetro el texto que queremos modificar (que en nuestro caso es el texto de “muertes”).

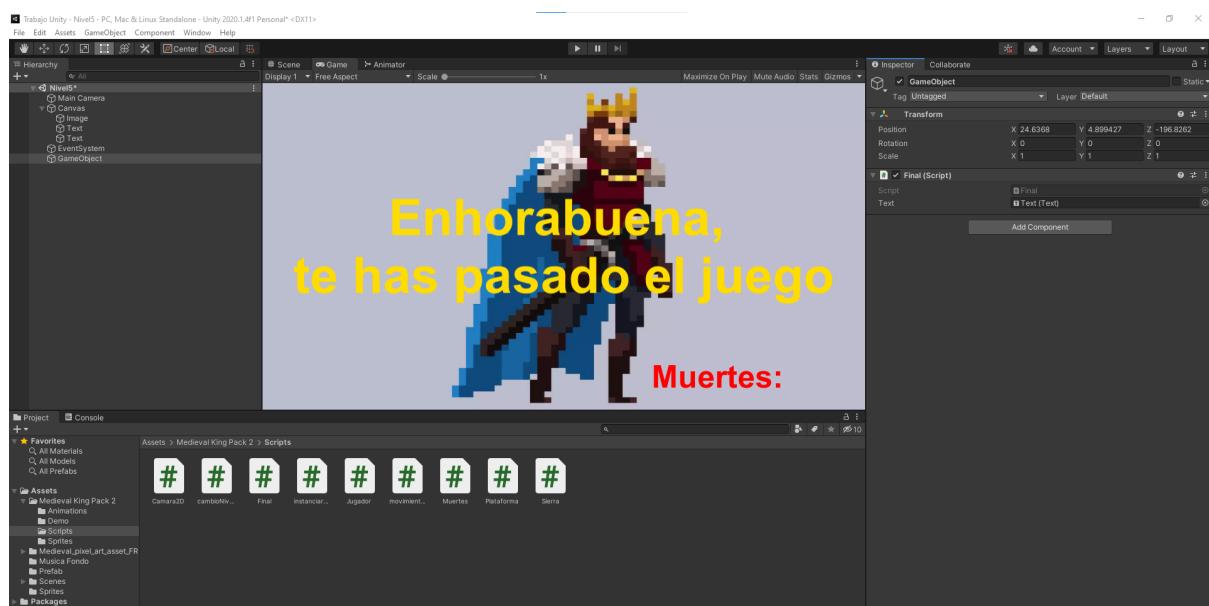


Figura 70. Asociando el Script Final

Esto que hemos hecho con el nivel 5, se debe hacer con todos los niveles excepto con el primero ya que de no hacer esto, al cambiar de nivel, no se nos mostrará el número real de muertes (se mostrará un 0 por defecto) hasta que el Jugador muera y se refresque el texto con la información correcta. Si incluimos el script Final en este nivel, al ejecutar el nivel se mostrará desde el principio el número de muertes actualizado.

## Corrección de la mecánica de muertes del juego

Por otra parte, hemos detectado un pequeño error en el funcionamiento del juego. Este error consiste en la posibilidad de seguir moviendo o manejando al Jugador cuando este se encuentra en el estado de Death. Esta mecánica es muy cutre y puede confundir al propio Usuario, además de ralentizar el reposicionamiento del Jugador en el inicio del nivel.

Para corregir este problema, vamos a agregar en el Animator Controller un nuevo parámetro booleano llamado muerte que nos va a indicar si el Jugador está muerto o no. Esto tiene como objetivo usar esta variable booleana para determinar si el Jugador puede moverse por el escenario o no. La mecánica es tan simple como impedir que el Jugador se pueda mover cuando esté muerto y la variable muerte sea true.

De esta manera, añadiremos el parámetro booleano muerte y en el script Jugador realizaremos las siguientes modificaciones:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class Jugador : MonoBehaviour
{
    public float velocidad = 5f;
    public float salto = 15f;
    protected Rigidbody2D rigidBody2D;
    protected Animator animator;
    protected Collider2D collider2d;
    public int dobleSalto;
    AudioSource audioSource;
    private Vector3 posicionInicial;
    public GameObject text;

    // Use this for initialization
    void Start()
    {
        rigidBody2D = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        collider2d = GetComponent<Collider2D>();
        audioSource = GetComponent<
```

```

        }

    }

void FixedUpdate()
{
    if (!animator.GetBool("muerte"))
    {
        rigidBody2D.velocity = new Vector2(velocidad * Input.GetAxis("Horizontal"),
rigidBody2D.velocity.y);
        animator.SetBool("correr", rigidBody2D.velocity.x != 0);

        //roto al personaje para es lo escalo a -1 o lo giro 180 en el eje y
        if (rigidBody2D.velocity.x < 0)
            transform.rotation = Quaternion.Euler(0, 180, 0);
        else if (rigidBody2D.velocity.x > 0)
            transform.rotation = Quaternion.Euler(0, 0, 0);

        //saltar arriba
        if (rigidBody2D.velocity.y > 0 && Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) == null)
        {
            animator.SetBool("saltar", true);
            if (animator.GetBool("caer"))
            {
                animator.SetBool("caer", false);
            }
        }

        //cayendo
        else if (rigidBody2D.velocity.y < 0 && animator.GetBool("saltar"))
        {
            animator.SetBool("saltar", false);
            animator.SetBool("caer", true);
        }

        //toco tierra
        else if (animator.GetBool("caer") && Physics2D.OverlapBox(transform.GetChild(0).position,
new Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
        {
            animator.SetBool("saltar", false);
            animator.SetBool("caer", false);
            animator.SetTrigger("caerTierra");
            dobleSalto = 2;
        }
    }
}

```

```

//el overlapbox se hace la mitad del collider para evitar que roce en lateral con una plataforma
y me deje saltar de nuevo, (collider2d.bounds.max.x - collider2d.bounds.min.x) / 2
if (Physics2D.OverlapBox(transform.GetChild(0).position, new
Vector2((collider2d.bounds.max.x - collider2d.bounds.min.x) / 2f, 0.01f), 0) != null)
{
    //Debug.Log("SAlto " + Time.time);
    if (Input.GetAxis("Jump") != 0)
    {
        rigidBody2D.AddForce(new Vector2(0, salto), ForceMode2D.Impulse);
        Debug.Log("SAlto " + Time.time);
    }
}
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Pinchos" && !animator.GetBool("muerte"))
    {
        audioSource.Play();
        animator.SetBool("muerte", true);
        animator.SetTrigger("morir");
    }
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Bala" && !animator.GetBool("muerte"))
    {
        Animator animator = GetComponent<Animator>();
        animator.SetBool("muerte", true);
        animator.SetTrigger("morir");
        Destroy(other.gameObject);
    }

    if (other.gameObject.name.StartsWith("Sierra") && !animator.GetBool("muerte"))
    {
        animator.SetBool("muerte", true);
        animator.SetTrigger("morir");
    }
}

public void recolocar()

```

```

{
    animator.SetBool("muerte", false);
    animator.SetTrigger("reinicio");
    rigidBody2D.velocity = Vector3.zero;
    transform.position = posicionInicial;

    Muertes.pMuertes += 1;
    text.GetComponent<Text>().text = "Muertes: " + Muertes.pMuertes;
}
}

```

Básicamente, con esta modificaciones, cuando el jugador, muere se pone la variable muerte a true y cuando el jugador se recoloca en el inicio del nivel, se cambia el valor de esta variable a false. Cuando el contenido de la variable muerte sea true, impediremos con un if tanto movernos por el escenario como saltar ya que queremos que el usuario mueva el Jugro cuando esté muerto.

También hemos utilizado esta variable muerte, para evitar por decirlo de alguna manera que el personaje muera cuando esté muerto. Es decir, en la versión anterior del juego, por ejemplo, si una sierra mata al Jugador y el cadáver del Jugador cae en los pinchos, se reproduce la animación de muerte dos veces y esto no es lo que queremos. Con este control que hemos añadido con la variable muerte, evitamos que se repita la animación de morir cuando el personaje ya está muerto.

Con esto, concluimos con el desarrollo de este videojuego de plataforma en Unity