Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

DESARROLLO DE UN VIDEOJUEGO EN PIXEL ART

DEVELOPMENT OF A VIDEOGAME WITH PIXEL ART STYLE

Daniel Eduardo González Marrero

La Laguna, 10 de septiembre de 2020

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43826207-Y, profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

“Desarrollo de un videojuego en pixel art”

ha sido realizada bajo su dirección por D. **Daniel Eduardo González Marrero**, con N.I.F. 79095244-S.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a **10 de septiembre de 2020**

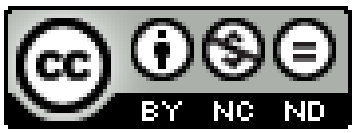
AGRADECIMIENTOS

- Quiero agradecer a mi familia y a mi pareja por el apoyo continuo durante el desarrollo del trabajo.

- Quiero agradecer a compañeros y amigos por realizar la fase de beta-testing para eliminar errores del juego.

- Y a mi tutor, Jesús Miguel Torres Jorge, por la ayuda y guía en el proceso.

LICENCIA



RESUMEN

El objetivo principal de este proyecto de fin de grado es el de **desarrollar un videojuego en 2D basándonos en el estilo visual "pixel art"**. Este videojuego ha sido realizado utilizando el motor gráfico de Unity, el cual nos proporciona una variedad de herramientas prediseñadas para poder realizar nuestro proyecto. Además, cuenta con un conjunto de recursos, bibliografía y documentación muy variado que nos facilita el proceso de desarrollo.

Se ha utilizado una **perspectiva top-down**, de manera que la cámara se sitúe en la parte superior del mundo, mirando hacia abajo. Gracias a esto, podemos ver como el personaje se mueve en vertical y horizontal (ejes X e Y).

Este proyecto se ha basado en la idea de realizar unos enemigos cuyo sistema de combate hagan que el videojuego suponga un reto para el jugador, y que intente sacar el máximo partido a los reflejos del usuario.

Además, se ha realizado el diseño de todos los elementos del mundo a mano, utilizando, como anteriormente se ha mencionado, la estética "pixel art". Estos diseños se han hecho con una herramienta externa y luego han sido importados a la aplicación de Unity como sprites.

Se desarrolló un nivel completo en el que varios enemigos intenten impedir el paso del personaje principal, el cual busca llegar al final del camino para eliminar a un poderoso jefe final.

- **Palabras clave:** videojuego, pixel art, difícil, 2d, top-down, IA, sprites, unity

ABSTRACT

The main target of this final degree Project is **to develop a 2D videogame using a "pixel art" visual style**. This videogame was developed with Unity. Unity gives us a wide variety of predefined tools to create our project. It also relies on some really useful documentation, bibliography and resources that can make the development much easier.

A **top-down perspective** was used, so the camera always point downwards. This will allow us to see the player moving vertically and horizontally (X and Y axis).

This project relies on the idea of the development of a combat system that makes this videogame a challenge for the player, making reflexes an important ability for the user.

Also, we designed every element of the project by our own hand, using "pixel art" aesthetic. These designs were made with an external tool, and then added to Unity as sprites.

A complete level was built, where numerous enemies will block the way to the end of the road, where a powerful final boss will be waiting for us.

- **Key Words:** videogame, pixel art, hard, 2d, top-down, AI, sprites, unity

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1
INTRODUCCIÓN

1.1 Introducción al proyecto

Este proyecto cuenta con una multitud de elementos a ser desarrollados, por tanto, necesitamos dividir el trabajo para mantenerlo organizado y hacer el desarrollo más ágil y menos caótico.

Debemos abordar cada elemento por separado utilizando las herramientas de las que disponemos.

Por otro lado, debemos centrarnos también en los antecedentes: proyectos anteriores similares que han tenido relativo éxito, y que pueden ayudar para inspirarnos o, incluso, dar ideas de cómo solucionar problemas que podamos encontrarnos.

CAPÍTULO 2
ANTECEDENTES Y SITUACIÓN ACTUAL

2.1 Antecedentes e influencia

Han sido varios proyectos los que han inspirado el desarrollo de este trabajo de fin de grado.

- **Hyper Light Drifter**: es un videojuego en dos dimensiones, con perspectiva top down, y cuyo estilo artístico se basa en pixel art. Es el principal antecedente de este proyecto. Su sistema de combate se basa en golpe con espada, disparo y dash[1].
- **Sekiro**: es un videojuego en tres dimensiones desarrollado por From Software. Su combate se sirve de reconocer patrones y tener reflejos para bloquear y realizar parry[2].
- **Wizard of Legend**: es un videojuego en dos dimensiones con perspectiva top down, que se basa más en el combate por magia, y supone un antecedente interesante en el que fijarse, puesto que en un futuro se puede ampliar el sistema de combate del proyecto.

2.2 Situación actual

Si hablamos de un proyecto con las características que buscamos, el más cercano es el anterior mencionado **Hyper Light Drifter**. Sin embargo, puede quedarse corto en cuanto a profundidad.

Es por eso por lo que surge este proyecto: crear una base ampliable que permita fusionar distintos conceptos de videojuegos muy distintos como lo son Hyper Light Drifter o Sekiro, en un solo proyecto que suponga un reto desafiante a la vez que adictivo.

[1] Término común en videojuegos. Es una habilidad que permite realizar un salto hacia la dirección en la que mira el personaje, generalmente a gran velocidad. Es una mecánica que se suele utilizar para esquivar golpes en juegos de acción en tiempo real.

[2] Supone un bloqueo en el tiempo justo, y generalmente viene seguido de una ventaja para el que lo realiza.

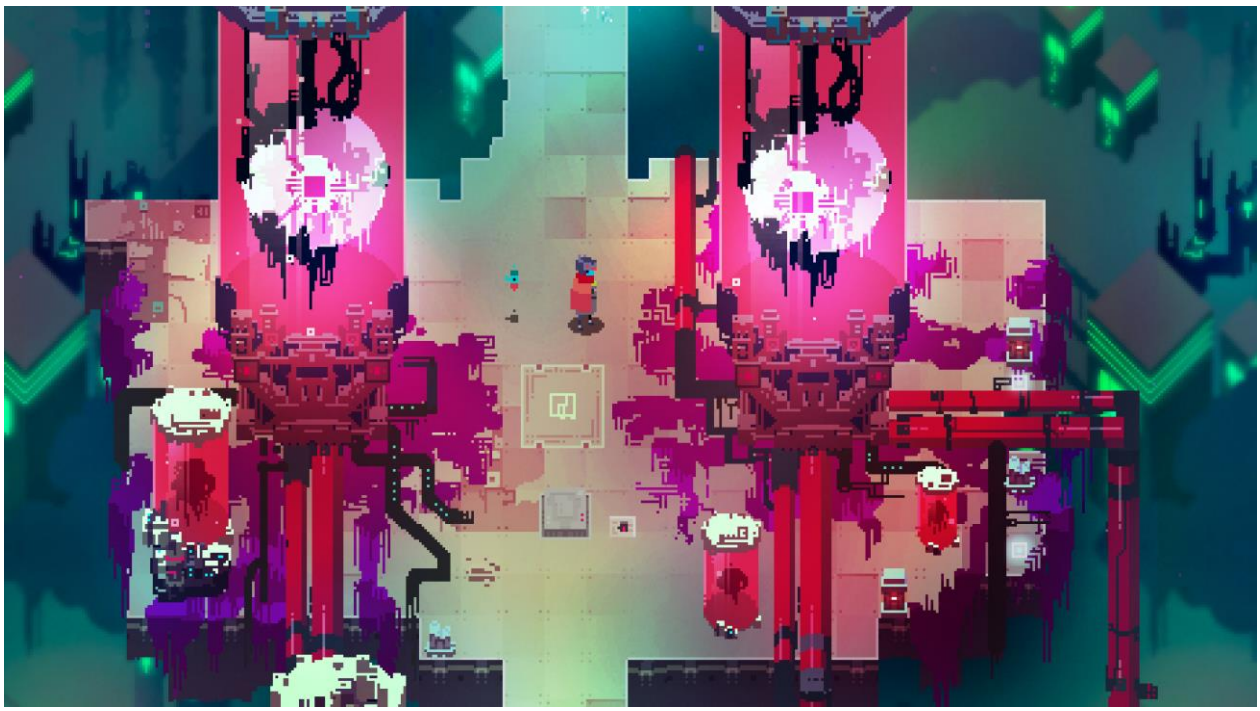


Figura 1: Hyper Light Drifter

Figura 2: Sekiro

CAPÍTULO 3
OBJETIVOS

3.1 Objetivos generales

El principal objetivo del proyecto es desarrollar un videojuego en dos dimensiones. Vamos a plantear de manera más concreta y puntualizada los objetivos principales:

- Desarrollar un videojuego en 2D utilizando la plataforma de Unity.
- Desarrollar un sistema de combate variado para el personaje principal:
  - Combate cuerpo a cuerpo con espada (o melee).
  - Sistema de disparo (pistola).
  - Bloqueo (con posibilidad de "parry").
- Desarrollar un sistema de combate desafiante para la IA.
- Desarrollar un nivel completo con distintos enemigos.
- Diseñar tanto el entorno como los elementos y personajes del videojuego utilizando un estilo pixel art.
- Desarrollar un algoritmo de generación procedural de mundos.
- Desarrollar elementos extra: salto, puzles, dash…

3.2 Objetivos específicos

Por otro lado, contamos con objetivos específicos:

- Aprender cómo se gestionan los sprites en Unity.
- Desarrollar un entorno basándonos en el mapa de tiles que trae Unity por defecto.
- Comprender el funcionamiento de las animaciones y del "Animator" de Unity para poder transmitir el comportamiento de los personajes a su animación.
- Desarrollar un sistema de colisiones[3] para el combate cuerpo a cuerpo y a distancia.
- Desarrollar un sistema de combate que permita bloquear y hacer "parry".
- Aprender a realizar sprites[4] en pixel art mediante Photoshop.

[3] Los colliders son los elementos que utiliza Unity para sus sistema propio de físicas.

[4] Un gráfico digital, utilizado comúnmente para representar objetos y personajes en videojuegos.

CAPÍTULO 4
ORGANIZACIÓN DEL PROYECTO

Como mencionamos en la introducción, debemos organizar el proyecto en distintas fases, de manera que podamos trabajar con cada una por individual y luego enlazarlas al conjunto del proyecto.

También debemos enumerar las herramientas que tenemos disponibles para nuestro proyecto.

4.1 Herramientas disponibles

- **Unity**: En primer lugar, disponemos del motor gráfico de Unity. Ya hemos comentado anteriormente que su cantidad de recursos y documentación nos hará mucho más fácil el desarrollo, además de tener una gran cantidad de assets creados por usuarios que nos podrían servir para nuestro proyecto.
- **Visual Studio**: Contamos también con el editor de Visual Studio, que tiene instalado el componente para trabajar en C# con Unity. De esta manera, el reconocimiento de errores en el código es amplio y rápido, lo que aumenta la productividad.
- **Adobe Photoshop**: Esta aplicación de diseño gráfico nos ha permitido desarrollar los sprites que hemos utilizado, tanto para el entorno como para los distintos personajes. También hemos diseñado los menús del videojuego.

4.2 Fases del proyecto

El proceso se ha dividido en distintas fases que trataremos por individual. Una vez terminada cada fase, se ha enlazado al proyecto completo. Sólo una actividad se ha realizado en paralelo con el resto y es la de diseño de sprites, de forma que se han ido diseñando los sprites necesarios para cada fase.

	Fase
1	**Documentación**
2	**Diseño de sprites**
3	**Desarrollo del entorno con tilemap**
4	**Desarrollo del personaje principal**
5	**Desarrollo de enemigo nº 1**
6	**Desarrollo de enemigo nº 2**
7	**Desarrollo de enemigo nº 3**
8	**Desarrollo de generación procedural, extras + corrección de errores**

Tabla 1: Fases de desarrollo

1							
2							
3							
4							
5							
6							
7							
8							

Tabla 2: Cronograma del proyecto

CAPÍTULO 5
DESARROLLO DEL PROYECTO

Para mantener una memoria limpia y organizada, vamos a explicar cada proceso del desarrollo por separado, siguiendo el orden establecido en el Capítulo 4, donde hemos definido las distintas fases.

5.1 Documentación

Para este paso, hemos recurrido a la documentación de Unity para poder establecer un formato correcto para el videojuego.

También hemos averiguado la manera más sencilla para realizar los sprites de los videojuegos y qué tipo de planteamiento podíamos darle al combate cuerpo a cuerpo.

Hemos probado otros videojuegos (Hyper Light Drifter) para poder buscar similitudes que desearíamos incluir en el proyecto, y ver como se tratan ciertos temas.

Una vez teniendo una idea muy concisa en la cabeza, procedemos a crear el documento del juego.

Creamos un nuevo proyecto en Unity, vamos al menú **Edit** y en **project settings**, vamos hasta Editor, y seleccionamos **Behaviour mode** a 2D.

Finalmente, el editor ya está preparado para comenzar con el proyecto.

5.2 Diseño de sprites

El desarrollo de los sprites se ha desarrollado con la herramienta de Adobe Photoshop, el cual, con los ajustes indicados, nos permite trabajar con píxeles de manera individual.

Las fases para el desarrollo de un Sprite en pixel art son las siguientes:

- Diseño conceptual en papel: se realiza un boceto del diseño, no necesariamente en pixel art, pero que sirve como guía para digitalizarlo.
- Creación del documento
- Diseño del elemento
- Reescalado del documento
- Exportación del documento

Explicaremos paso a paso las distintas fases, pero primero debemos configurar Adobe Photoshop para poder trabajar con píxeles.

El primer paso será abrir Adobe Photoshop. Pulsamos Ctrl + K para acceder a las preferencias. En “GENERAL”, en Interpolación de imagen, pondremos “por aproximación”.

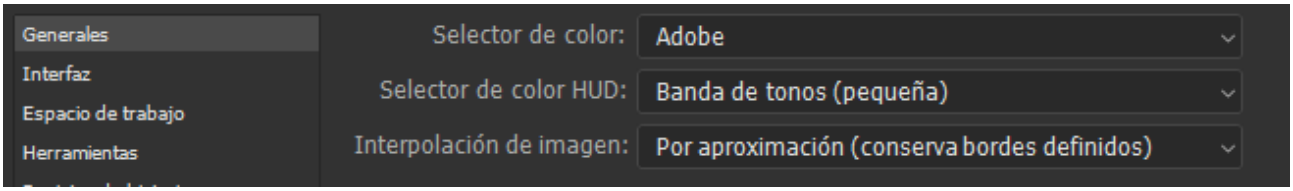


Figura 3: Seleccionamos "Por aproximación"

La interpolación es el algoritmo que utiliza Photoshop para rediseñar una imagen cuando se cambia su tamaño, sea para agrandarla o para hacerla más pequeña. Por defecto, está seleccionado “**Bicúbica automática**”, que permite difumina los bordes para intentar hacer el cambio de tamaño más natural. Nosotros buscamos que, al cambiar la imagen de tamaño, los píxeles se mantengan en su proporción y que se conserven los bordes de manera definida.

Esta opción nos proporcionará justo lo que buscamos.

Vamos ahora a diseñar nuestro Sprite. Lo primero que haremos será dibujarlo en un papel para poder tener una guía.

Ahora debemos crear un nuevo documento con Ctrl + N. Debemos definir un número de pixeles para cada unidad de nuestro Sprite. Este número será 40 pixeles de ancho por 40 pixeles de alto. Por tanto, cada unidad en nuestro mundo contará con un diseño de 40 pixeles por 40 pixeles.

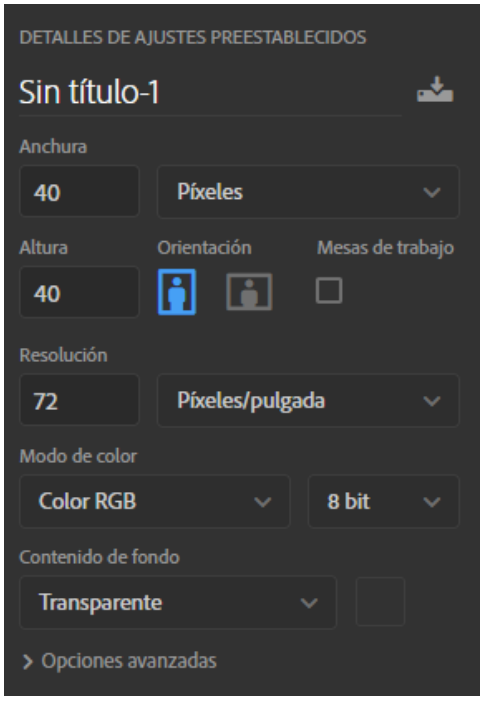


Figura 4: Opciones para documento (Sprite de 1ud x 1ud)

Le pondremos un nombre identificativo, colocaremos 40 pixeles por 40 pixeles de tamaño por cada unidad que suponga nuestro diseño, y escogeremos 72 PPP (píxeles por pulgada) de resolución.

Como el objetivo es una pantalla digital, el modelo RGB es el que debemos escoger. Además, pondremos fondo transparente.

Tras haber hecho esto, escogeremos la herramienta "**LÁPIZ**", escogeremos un color y dibujaremos nuestro Sprite.

Para las animaciones, será mucho más sencillo incluir todos los diseños en una sola hoja, llamada "sheet". Luego, desde Unity, podemos importar la imagen completa y recortarla según necesitemos.

Para este ejemplo, que es el del jefe final del nivel, vamos a escoger 80 pixeles por 80 pixeles, puesto que, al ser un enemigo grande, debe ocupar dos unidades de nuestro mundo de ancho y dos de alto.

Además, como vamos a realizar dos diseños (su animación estándar oscilará entre arriba y abajo), multiplicamos el ancho por dos, para que quepan dos sprites. Por tanto, para este ejemplo, el diseño tendrá un tamaño de 160 de ancho por 80 de alto.

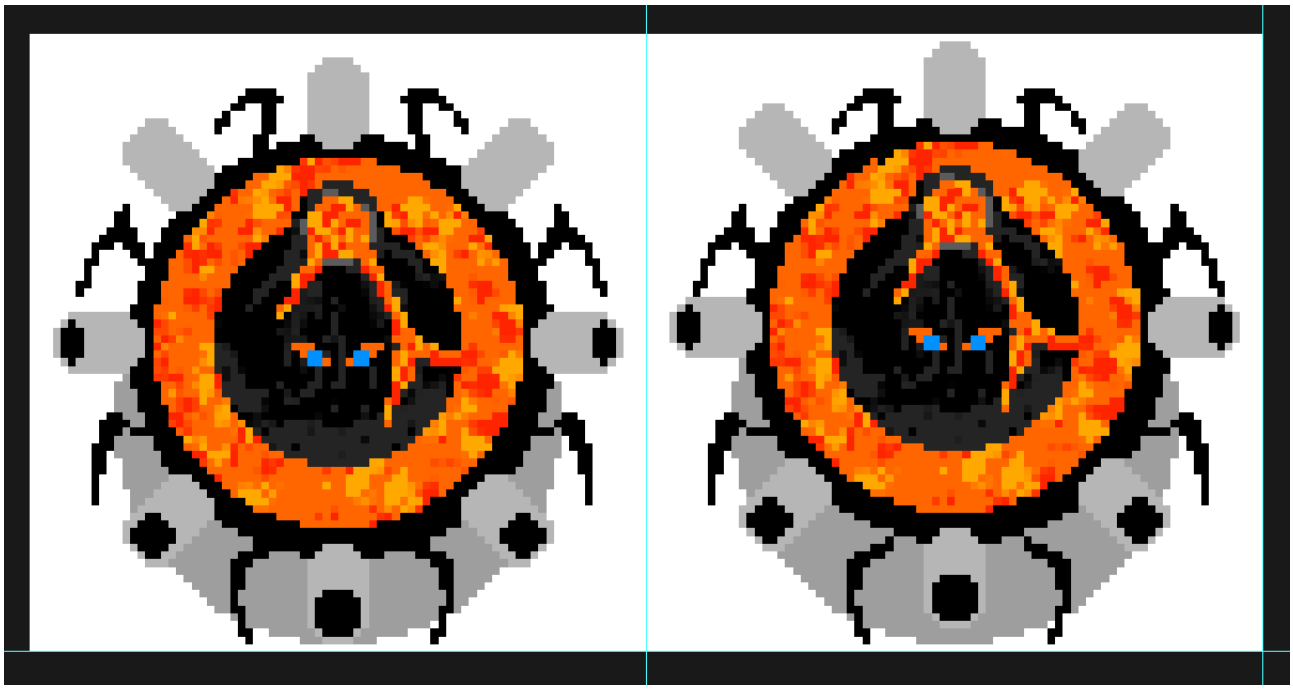


Figura 5: Diseño de sprite de jefe final

Una vez terminado nuestro diseño, ahora tenemos que reescalarlo. Esto es debido a que, al importarlo a Unity, un diseño de 160 pixeles por 80 pixeles acabaría mostrándose muy pequeño.

Debemos reescalar nuestro diseño a 512 pixeles por cada 40 pixeles. Para ello, hacemos Ctrl + Alt + I, y con la cadena marcada (mantiene las proporciones), colocamos en donde pone 160 pixeles, 2048. Automáticamente abajo se colocará 1024 pixeles.

A continuación, guardamos nuestro documento con Ctrl + Shift + S, y seleccionamos el formato PNG. Debemos guardarlo en la carpeta de Assets del proyecto de Unity, dentro de “Sprites”.

Ahora, desde Unity, con el navegador de proyecto, vamos hasta donde esté nuestro Sprite. Al hacer click se activará el inspector, y debemos colocar “Pixels per unit” a 512 (debido al reescalado).

Si se trata de un sheet como en este ejemplo, en donde pone **Sprite mode**, debemos colocar “Multiple”. Aplicamos los cambios y vamos al **Sprite editor**. Desde ahí seleccionamos **Slice**, colocamos el **pivot** (el centro de gravedad de nuestro Sprite) abajo (bottom). Esto es debido a que nuestro juego es top-down, y nos interesa al dibujar las capas de los sprites se realicen desde ese punto. Guardamos y ya tenemos dividido nuestro sheet en varios sprites.

Por el contrario, si se trata de un solo Sprite, tan solo será necesario, en el inspector, poner en **Pivot**: Bottom, Dejamos **Sprite mode** en Single, y **pixels per unit** a 512.

El proceso de animación lo explicaremos con el personaje principal, y será el mismo para los enemigos.

5.3 Desarrollo del entorno con Tilemap

Tilemap

En la jerarquía de la escena se crea un nuevo objeto: 2D Object -> Tilemap.

Para crear las baldosas, o tiles, que serán los cuadros que conformarán nuestro mundo, debemos diseñarlos primero en Photoshop, como hemos indicado en el punto 5.2.

Las he desarrollado de 3 x 3 para poder formar todas las combinaciones posibles.

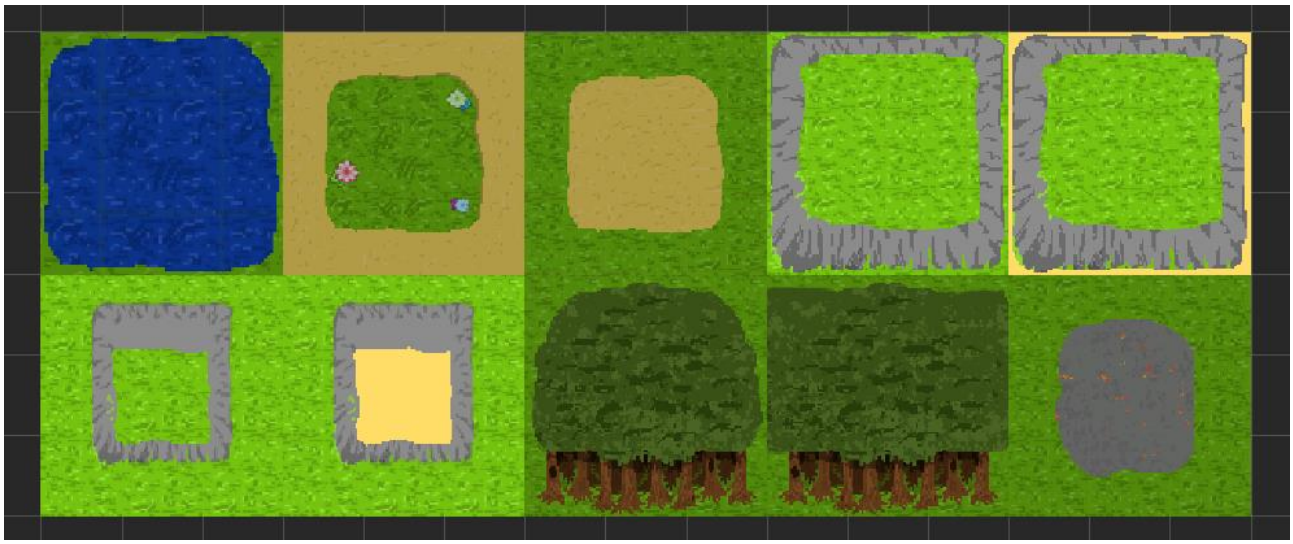


Figura 6: Baldosas creadas para el mundo

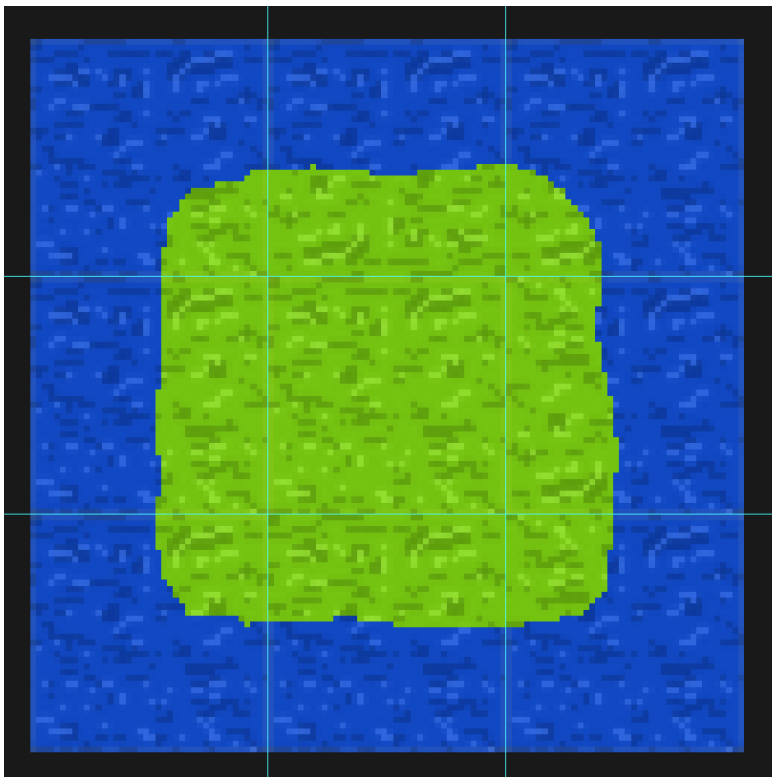


Figura 7: 9 baldosas distintas

Aquí podemos ver como he diseñado 9 baldosas distintas. Estas 9 baldosas las podemos interconectar para realizar las formas que necesitemos. Luego, como explicamos en el paso anterior, las dividimos y creamos distintas baldosas. A continuación, bastará con arrastrar nuestro Sprite al **Tile Palette** de Unity, y ya podemos seleccionar cada baldosa y aplicarlo en nuestra "**Scene view**"

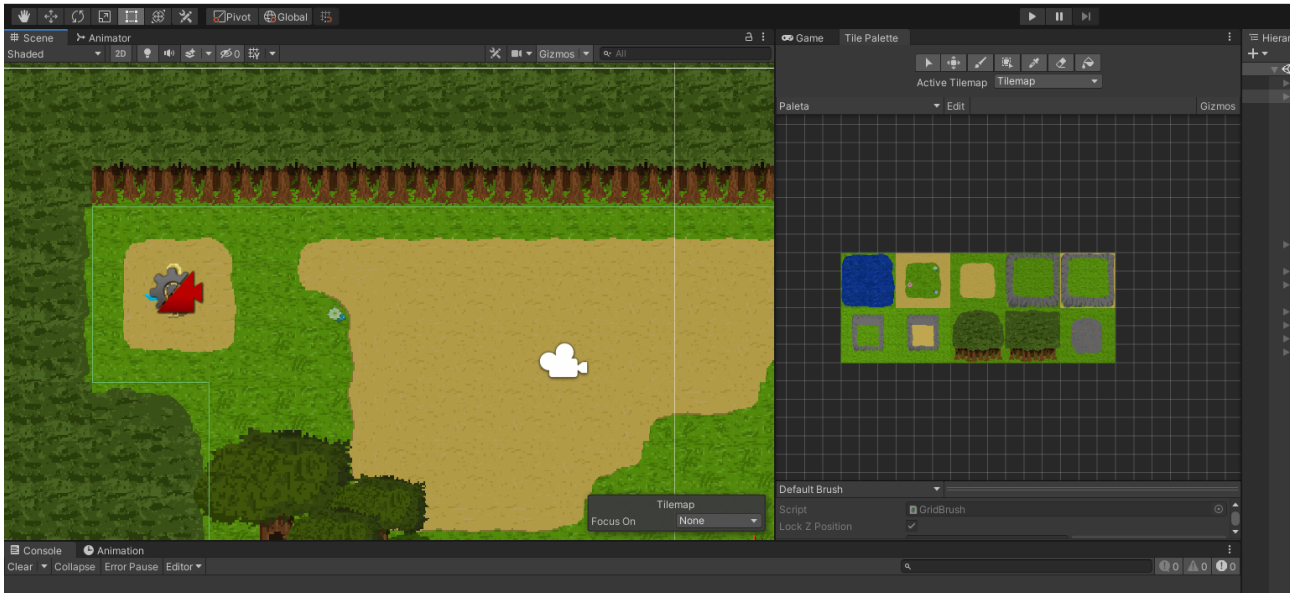


Figura 8: Mapa creado con las baldosas

Colisionadores

Ahora, para añadir colisionadores a nuestro mapa, debemos seleccionar el gameobject de nuestro tilemap, y añadirle un Rigidbody2D y seleccionar su modo a estático, puesto que el mapa nunca va a moverse.

También añadimos un TileMap Collider2D, y ahora tan solo debemos ir a los objetos tile de nuestras baldosas y seleccionar aquellos que SÍ puedan ser transitables, y en el inspector, en **Collider Type**, poner “none”. Las baldosas que no se hayan seleccionado tienen puesto en esa opción “Sprite”, por lo que ahora mismo son un collider que bloquearán el paso.

Hay una manera de optimizar este proceso. Ahora mismo cada cuadrado es un collider. Podemos conseguir crear un único collider con todos los que hemos creado anteriormente.

Esto lo conseguimos añadiendo a nuestro TileMap un Composite Collider 2D. Luego tan solo en TileMap Collider 2D marcamos “**Used By Composite**” y ya tenemos nuestro mundo listo.

Objetos y decoraciones

Para crear objetos y decoraciones lo que haremos será crear un sprite. En nuestro caso, uno de los que hemos creado es un conjunto de cajas. Este sprite lo arrastramos hasta nuestra ventana de jerarquía. Añadimos un rigidbody2d, ponemos la gravedad a 0 y luego añadimos un box collider 2d. Ajustamos el collider a la mitad inferior del objeto (recordemos que estamos en perspectiva Top-down).

Figura 9: Cajas y su collider

Con esta técnica hemos creado también los árboles y las rocas. La única diferencia es que tenemos que ajustar el collider a un tamaño proporcional al objeto. Cada uno de estos objetos luego los podemos arrastrar a nuestra carpeta “Prefab” en el proyecto, de manera que podamos recurrir a ellos y tener copias idénticas de cada uno

Figura 10: Árboles

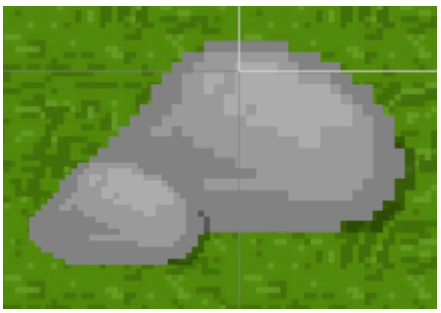


Figura 11: Rocas

5.4 Desarrollo del personaje principal

El desarrollo del personaje principal ha sido el más largo de todos. No sólo por la complejidad de su control sino también por equilibrar sus parámetros, y que no tuviera ventaja ni desventaja contra sus enemigos. Empezaremos hablando de la creación.

Creación del personaje

Una vez creado el sprite del personaje, simplemente tenemos que importarlo a la escena y hacer exactamente los mismos ajustes que realizamos para las cajas, las rocas o los árboles, de manera que tenemos un prefab con un rigidbody2D y un boxcollider2D.

Luego de haber creado nuestro personaje, debemos darle movimiento.

Movimiento del personaje

Vamos a crear nuestro primer script, el cual he llamado **MoonboyController**, y que finalmente albergará todas las funciones de nuestro personaje principal.

En principio, hemos creado su atributo principal: velocidad.

En el método **update()** hemos recibido el input de los botones de dirección (configurados por defecto por Unity, WASD o Flechas de dirección). Una vez obtenidos, simplemente movemos el **transform.position** de nuestro personaje, sumándole un vector con el componente horizontal y vertical llamado **change**.

Guardamos el script y se lo añadimos como componente a nuestro personaje principal.

Problema encontrado: El personaje se mueve, pero vibra al caminar contra un objeto como las cajas o las rocas. Esto es debido a que las colisiones se generan un plano distinto (sistema de físicas de unity) al que podemos ver, así que cuando el rigidbody entra en colisión, tiene que corregir también el transform.position de nuestro personaje. Además, al chocar con un elemento, empieza a rotar sobre el elemento.

Solución: En lugar de mover el transform.position, obtenemos el componente rigidbody2d de nuestro personaje y lo guardamos en una variable. Lo que haremos será mover de posición el rigidbody directamente, y esto eliminará la vibración. Para eliminar la rotación, tenemos que ir al componente rigidbody2d de nuestro personaje y hacer click en **Freeze Rotation: Z**.

```
rigidbody2d.MovePosition(act_pos + change * speed);
```

Problema encontrado: Al cambiar el movimiento al rigidbody2d, la velocidad de movimiento se reduce. Además, dependiendo de la velocidad de fotogramas de cada ordenador, se moverá a velocidad distinta.

Solución: Cambiamos el algoritmo de movimiento a **FixedUpdate()** en lugar de en el Update(). Multiplicamos el movimiento por Time.deltaTime (el tiempo que dura un fotogama).

```
rigidbody2d.MovePosition(act_pos + change * speed * Time.deltaTime);
```

Para realizar las animaciones para el movimiento, debemos importar y recortar en Unity todos nuestros sprites. Luego, debemos añadir un componente Animator a nuestro personaje principal. Luego, en la carpeta de Assets, creamos una carpeta Animations. Dentro, desde Unity, creamos un componente Animator Controller. En el componente Animator de nuestro personaje, seleccionamos como controlador este último que hemos creado. Para realizar las animaciones, todo componente necesita un “Animator”, que se encarga de decidir qué animación se ejecuta en todo momento.

Figura 12: Hoja de movimiento para Moonboy

Estando ya importadas nuestras animaciones, abrimos la ventana Animation. Creamos un nuevo clip, por ejemplo, para estar en reposo mientras mira hacia abajo. Esta animación consta de dos frames[5]. Arrastramos esos dos frames al clip en animation y seleccionamos el número de muestras a 2 (este es el número de sprites que tiene la animación).

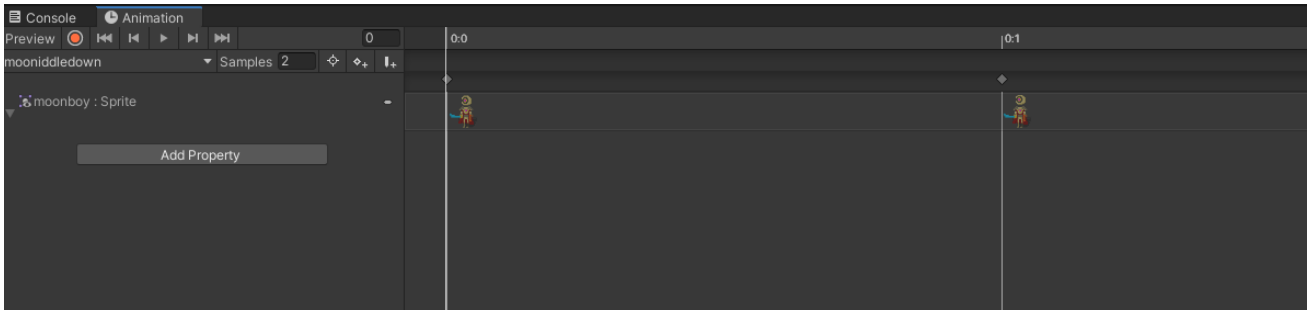


Figura 13: Ventana de animación

[5] Fotogramas.

Una vez hecho esto, creamos otro clip para el resto de las direcciones. También crearemos otros cuatro clips para que nuestro personaje corra. Las animaciones para correr arriba y abajo constan de dos muestras, mientras que las horizontales constan de cuatro muestras. Realizamos el mismo proceso de antes configurando el número de muestras al que necesitemos.

Cuando esté terminado, podemos pasar al Animator. Necesitamos, de alguna manera, decirle al Animator que nuestro personaje está mirando hacia X dirección para que el Animator pueda seleccionar la animación correspondiente. Hemos creado dos parámetros, Look X y Look Y, que se encargan de decir hacia dónde está mirando nuestro personaje.

Ahora, en nuestro mapa sólo debemos eliminar todas las animaciones que hemos creado y crear dos nuevos **blend tree**. Los **blend tree** se encargan de elegir animaciones en base a parámetros. Un **blend tree** se llamará **Idle**, y se encargará de elegir la animación para cuando Moonboy esté en reposo. El otro se llamará **Moving**. Ambos usarán los dos parámetros que hemos creado.

Ya sólo es cuestión de entrar en cada blend tree y configurar qué animación se activará para cada valor de los parámetros. La configuración será exactamente la misma para **Idle** y para **Moving**.

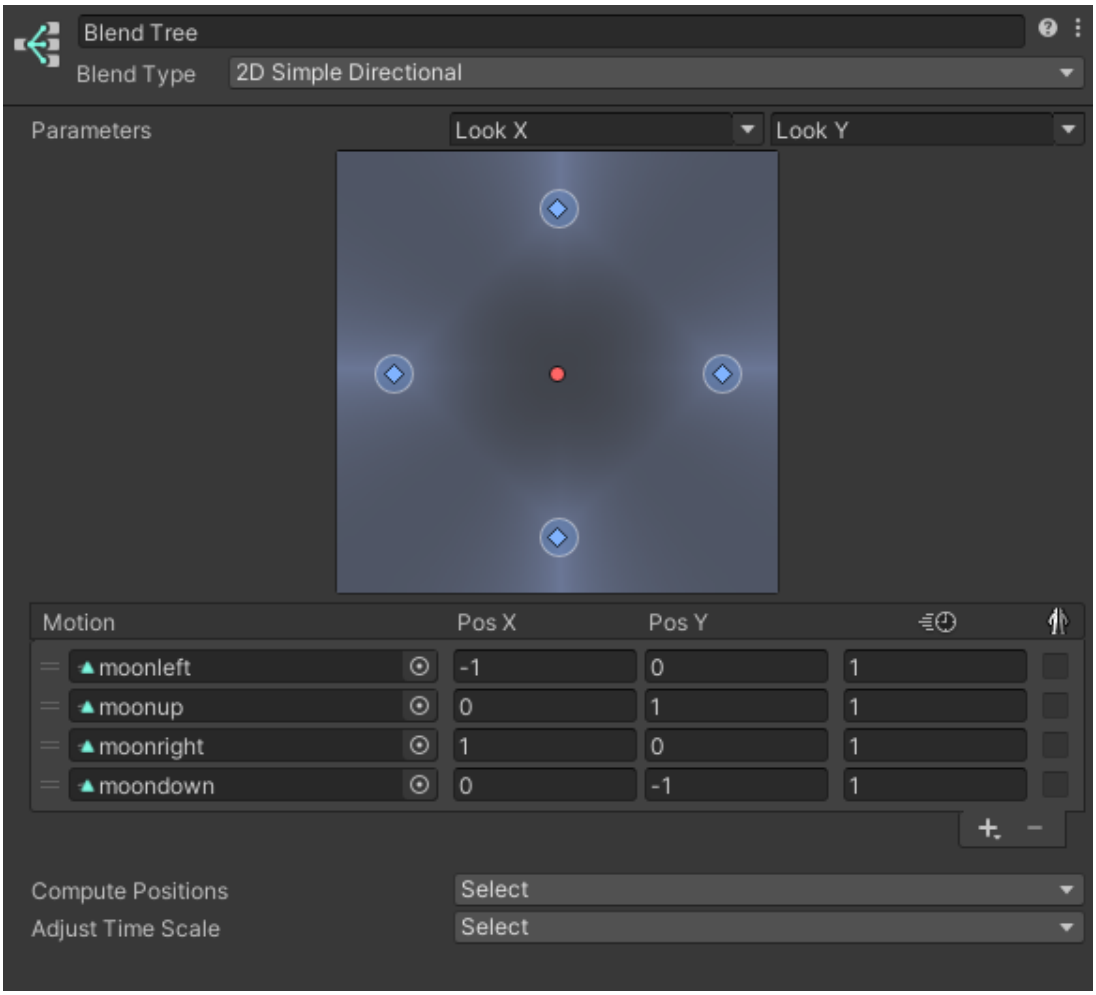


Figura 14: Configuración para blend tree

Luego, en nuestro mapa principal del Animator, debemos seleccionar que desde el estado de reposo podemos pasar al estado **Moving**, y viceversa. Debemos crear un nuevo parámetro en el Animator llamado **Speed**, de manera que si la velocidad es mayor que 0.1 (debemos definir este parámetro para cuando el personaje principal deje de correr y su velocidad todavía no sea cero) entonces pase hacia **Moving**. Si es menor que 0.1, entonces pasará de **Moving** a **Idle**. El Animator trabaja con una especie de máquina de estados que a su vez trabaja con otra máquina de estados en su interior.

Desde el script de nuestro personaje principal, en cada fotograma debemos enviarle hacia dónde está mirando nuestro personaje, y eso lo conseguimos enviando la entrada de teclado al Animator.

```
animator.SetFloat("Look X", lookDirection.x);
animator.SetFloat("Look Y", lookDirection.y);
animator.SetFloat("Speed", change.magnitude);
```

Le pasamos ambas direcciones y también le pasamos a la variable Speed la magnitud del vector de movimiento, que no es sino un vector con los valores de entrada horizontal y vertical.

Problema encontrado: En este punto pudimos darnos cuenta de que la animación de nuestro personaje no se mostraba correctamente. Es decir, cuando se suponía que debía estar delante de una caja, realmente se mostraba detrás.

Solución: Esto se arregla yendo a las opciones del proyecto y configurar para que las capas se muestren según su eje Y. Es decir, aquello que esté más abajo en el eje Y se mostrará delante, y lo que esté más arriba, se mostrará detrás.

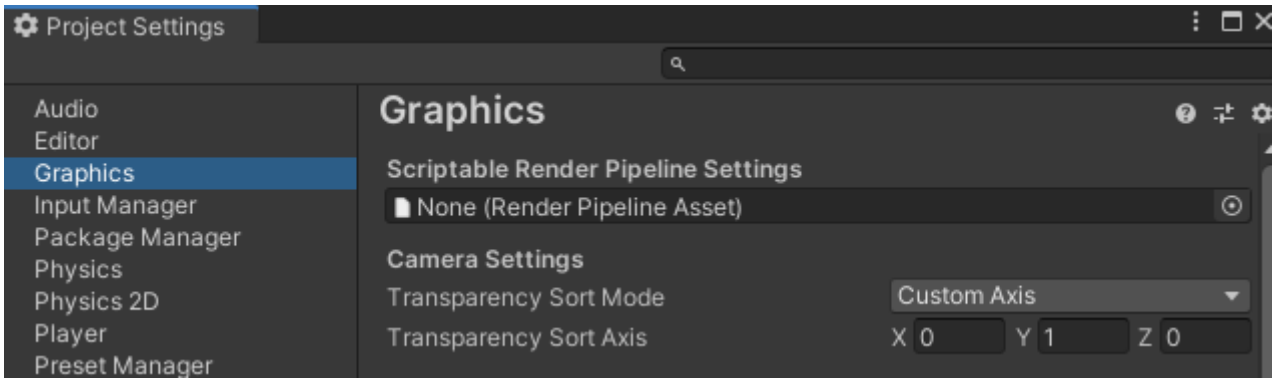


Figura 15: Configuración para las capas

Una vez configuradas las animaciones de movimiento, podemos pasar al sistema de combate.

Sistema de combate

El primer paso sería conseguir que nuestro personaje dispare a distancia. Esto lo conseguimos creando un prefab de un proyectil.

A este prefab le añadiremos un rigidbody2D y un boxcollider2D.

La diferencia es que en el boxcollider marcaremos “Is Trigger”. Lo que conseguiremos con esto será que realmente no se detenga al chocar con algo, sino que simplemente mande un aviso de que se ha traspasado otro collider.

Una vez creado, debemos crear un script llamado “Projectile”. En aspectos generales, lo que haremos en este script será recibir ese aviso de que se ha traspasado un collider y actuar en base a ello.

En este script lo que haremos será obtener el collider con el que ha chocado, y si es un personaje enemigo, restarle uno de vida mediante un método público.

Más adelante, cuando estén creados los enemigos, así quedaría la función:

```
Mensaje de Unity | 0 referencias
private void OnTriggerEnter2D(Collider2D collision)
{
    //we also add a debug log to know what the projectile touch
    Debug.Log("Projectile Collision with " + collision.gameObject);

    Destroy(gameObject);

    GameObject splashhit = Instantiate(splashobject, rigidbody2d.position, Quaternion.identity);
    EnemyController e = collision.GetComponent<EnemyController>();

    if (e != null)
    {
        e.Damage(1);
    }
}
```

Figura 16: Script para pistola

Cuando se active esta función, destruimos el propio proyectil y generamos un “splash”, que no es sino un sprite de una pequeña explosión convertido en prefab.

Ahora debemos ir al script del personaje principal y comprobar si pulsa la tecla C. Si es así, vamos a instanciar uno de estos proyectiles y les vamos a añadir una fuerza hacia la dirección en la que apunte el personaje.

```
1 referencia
void Launch()
{
    GameObject projectileObject = Instantiate(projectilePrefab, rigidbody2d.position + Vector2.up * 0.5f, Quaternion.identity);

    Projectile projectile = projectileObject.GetComponent<Projectile>();
    projectile.Launch(lookDirection, 500);

    animator.SetTrigger("Launch");
}
```

Figura 17: Script para lanzar una bala

Para la animación seguimos los pasos definidos anteriormente: creamos las animaciones y lo definimos en nuestro Animator. A continuación, crearemos un nuevo parámetro "**trigger**", el cual sólo sirve para indicar que ha sido activado y en base a ello ejecutar una acción. Para ejecutar nuestro **blend tree** de animaciones de disparo, tiene que activarse. Y eso es lo que hacemos en nuestro script.

Ahora podemos pasar al sistema de combate cuerpo a cuerpo, a priori el más complicado de planear.

Hemos desarrollado cuatro animaciones más para el ataque cuerpo a cuerpo. Lo que haremos ahora será crear cuatro hijos dentro del gameobject de nuestro jugador. Cada uno se llamará hit + dirección, de manera que cada uno corresponda con un hitbox en cada dirección.

Una vez definido, para cada uno debemos añadir un **"polygon collider 2D"**. Y ejecutando la animación con el loop[6] de la ventana Animation, creamos nuestro collider en base al ataque del jugador. Seleccionamos también que sea "Is Trigger"

Figura 18: Creando hitboxes

Cuando los cuatro colliders estén listos, simplemente tenemos que definir en el script de nuestro personaje que cuando se presione la tecla X, active la animación de ataque (creamos un nuevo parámetro llamado Attack, en este caso booleano). Además, debemos crear la función "**OnTriggerEnter2D**"[7] en un nuevo script que añadiremos a los cuatro hitboxes. Dentro de estos hitboxes se detectará si la colisión se ha producido con un enemigo, y si es así, les restará uno de vida.

[6] Repetición continua. En este caso, de una animación.

[7] Esta función se activa cuando un collider entra en el recinto de otro collider que tiene "Is Trigger" activado.

Problema encontrado: Al atacar se daña en los cuatro hitboxes. Esto es porque no hemos definido cuál se debe activar en cada situación, sino que están los cuatro activos en todo momento.

Solución: En el panel Animation tenemos una opción que nos permite “grabar acciones”. Desactivamos los cuatro hitboxes que hay dentro de nuestro personaje. Modificamos nuestras animaciones de ataque: le damos al botón “record” y en el primer frame activamos el hitbox[8] que corresponda. Movemos la línea de tiempo al último frame y desactivamos el hitbox. Quitamos el botón “record” y ya tenemos configurada la animación. Realizamos el mismo paso con las demás direcciones. Si el hitbox no está activado, entonces no podrá hacer daño a nadie. Sólo se activará en la dirección a la que mira el personaje principal cuando pulse la X.

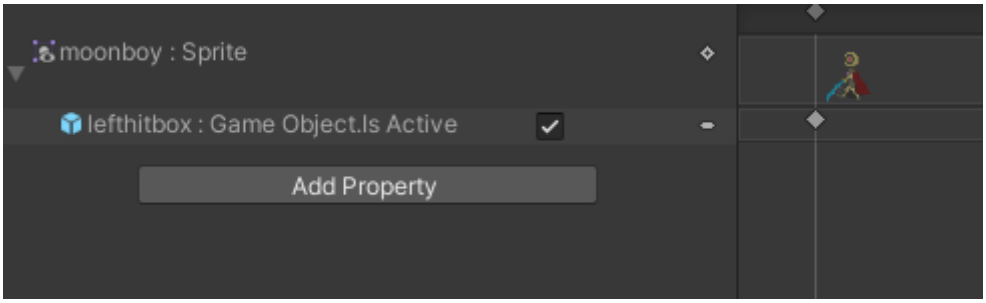


Figura 19: Activando hitbox hacia la izquierda

El último elemento para terminar el sistema de combate será el bloqueo. Esto constará de crear la animación y activarla cuando el usuario pulse la letra Z. De esta manera, se activa un booleano que evita que el personaje principal reciba daño durante un segundo.

Problema encontrado: El código resulta muy caótico y la manera de definir lo que va a hacer el personaje según la entrada de teclado es muy primitiva y complicada de modificar.

Solución: Implementar una máquina de estados. Hemos definido un enumerable dentro del script del personaje principal que permite los siguientes estados: idle, running, attacking, dashing, shooting, blocking.

Comprobar su estado en todo momento nos ayuda a definir lo que puede o no puede hacer el personaje según las circunstancias. Esto en adición a la creación de subrutinas, nos ha permitido definir tiempos de espera sin parar el motor del juego.

[8] Un hitbox es un recinto delimitador. Se utiliza para definir la zona en la que puede un personaje puede recibir daño.

5.5 Desarrollo de enemigo nº1

El primer enemigo desarrollado constará de un personaje humanoide muy parecido a nuestro protagonista, cuyo nombre es Sunboy.

Constará de combate cuerpo a cuerpo y pistola. Sus animaciones serán muy parecidas, y el sistema cuerpo a cuerpo se ha desarrollado exactamente igual que como se ha hecho con el personaje principal, así que obviaremos todo este proceso pues ya está explicando en el punto 5.4.

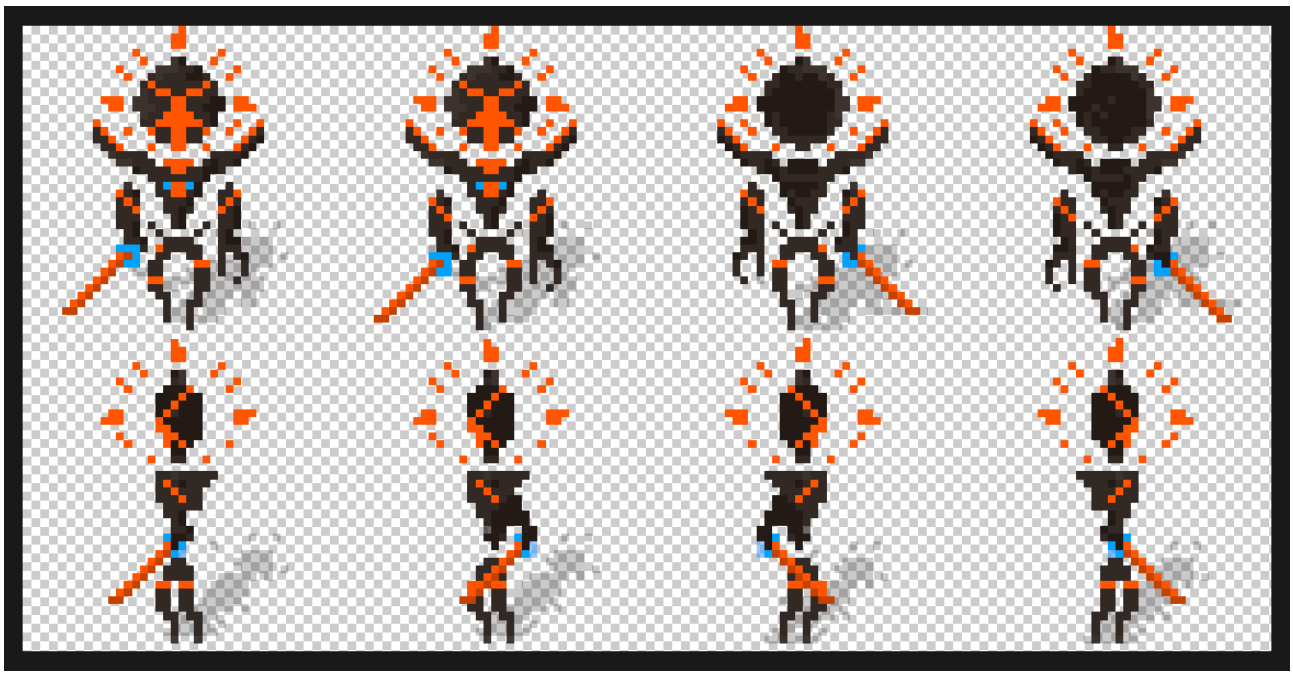


Figura 20: Sheet de Sunboy

Lo que quedaría por explicar es su IA para atacar al personaje principal.

Debemos definir tres distancias. La primera es la distancia de persecución, es la distancia radial en la que el enemigo se activa y persigue al personaje principal. Luego tenemos la distancia de ataque, que es la distancia en la que se detiene para atacar cuerpo a cuerpo. Por último, tenemos la distancia de disparo, que es la distancia en la cual decide disparar.

También debemos decirle al enemigo quién es su objetivo, así que definimos una variable **target**, a la que se asocia el personaje principal. El enemigo sabe continuamente donde se encuentra el enemigo siempre y cuando se encuentre dentro de su rango de acción.

Una vez el personaje principal entra dentro de su rango de acción (5 unidades a la redonda), se acerca hacia él. Cuando está a una distancia 3 del objetivo, dispara. Cuando llega a la distancia de ataque (0.5 unidades), ataca cuerpo a cuerpo.

Sin embargo, si ataca cuerpo a cuerpo mientras el personaje principal está bloqueando, se realiza un parry a favor del personaje principal, y queda bloqueado sin acción durante dos segundos.

Este enemigo también cuenta con una máquina de estados que nos ayuda a definir toda la estructura interior del algoritmo, y que nos facilita poder añadir nuevas habilidades en un futuro.

Para poder saber cuánto de vitalidad le queda al enemigo, hemos añadido un gameobject hijo del enemigo, el cual tiene un componente **TextMesh**.

Desde el script del enemigo, simplemente obtenemos este componente y cambiamos su texto para que refleje la vitalidad actual y la máxima separados por una barra.

```
3 referencias
public virtual void Damage(int amount)

{

    currenthealth -= amount;

    if (currenthealth > 0)
    {
        string life = currenthealth + "/" + maxhealth;

        texto.text = life;

    }
    else if(currenthealth == 0)
    {
        string life = currenthealth + "/" + maxhealth;

        texto.text = life;
        currentState = EnemyState.dying;
        StartCoroutine("Waiter");
    }
```

Figura 21: Algoritmo para recibir daño de enemigo

5.6 Desarrollo de enemigo nº2

Avanzado el desarrollo, se ha decidido realizar más enemigos con diferentes capacidades. Se puede apreciar que muchos de los atributos y funciones que utilizaríamos en este nuevo enemigo ya se usaban en el primer enemigo, es por ello por lo que hemos decidido darle un componente de herencia y polimorfismo a nuestro proyecto, permitiendo además en un futuro poder añadir más enemigos con distintas habilidades.

Nuestro segundo enemigo es una araña que se lanza constantemente contra el enemigo, y hemos hecho que herede todo del primer enemigo.

Sin embargo, su forma de actuar no es la misma, así que habría que sobrecargar las funciones de ataque de este nuevo enemigo.

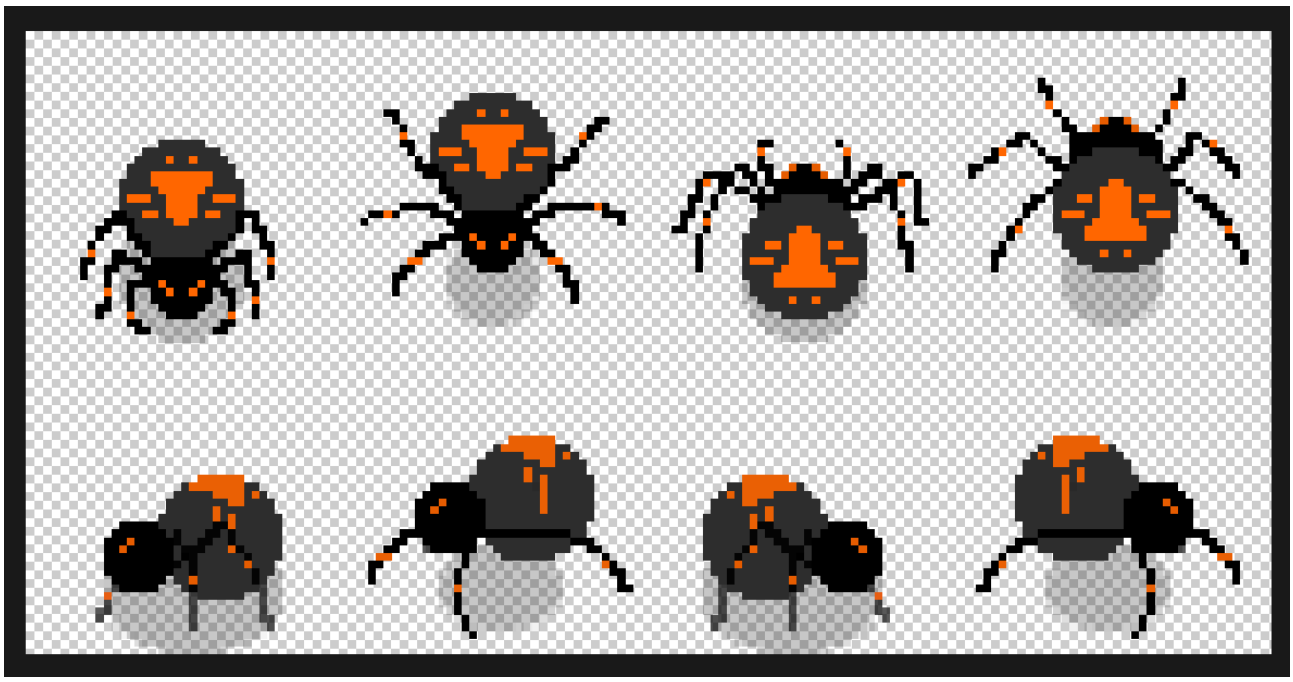


Figura 22: Sheet de la araña

Al igual que el primer enemigo, debe tener un objetivo, el cual es el personaje principal. También debe tener un rango de acción, aunque este va a ser mucho más amplio.

Debe obtener la posición del enemigo, y lanzarse hacia esa dirección a una velocidad desmedida. El rigidbody de este enemigo es “Is Trigger” para que pueda atravesar a nuestro personaje principal y aun así hacerle daño.

Problema encontrado: La araña actualiza su posición a mitad de trayecto, y en su salto hace curvas, acabando SIEMPRE en la posición del enemigo. Por tanto, nunca falla en su ataque.

Solución: Este enemigo tiene un temporizador de un segundo para sus ataques. Vamos a tomar cada ataque por individual. Cada vez que se disponga a atacar, debe guardar la posición en la que se encuentra el personaje principal en ese momento. Esta dirección, durante el resto del tiempo que dura el ataque, va a ser siempre la misma. Por tanto, damos un pequeño tiempo al personaje principal para poder esquivarlo.

5.7 Desarrollo de enemigo nº3

Llegados a este punto hemos creado dos enemigos muy distintos y que requieren una reacción distinta. Este tercer y final enemigo no es una excepción. Se encontrará al final del nivel y no se moverá. Su único objetivo será disparar balas hasta que lo mates, al estilo de los antiguos videojuegos de naves (Figura 5).

Este enemigo utilizará un sistema parecido al del enemigo nº1. Pero este, una vez entre dentro del rango de acción, empezará a disparar automáticamente hasta que lo maten o hasta que el personaje principal salga del rango de acción.

Por tanto, debemos utilizar la técnica que usamos con Moonboy o Sunboy: instanciaremos balas y les daremos fuerza con **rigidbody2d.addForce**.

En este caso, con una variable aleatoria, dispararía balas formando una cruz, o bien una X. Todo esto con un tiempo de espera entre cada bala.

Problema encontrado: Alejándose lo suficiente del enemigo, fuera de su rango de acción, se puede matar al enemigo a disparos desde lejos.

Solución: Aumentar el rango de acción.

Problema encontrado: Al disparar las balas solo en 8 direcciones, puede ser fácil atacar al enemigo.

Solución: Añadir una bala más que dispara en una dirección aleatoria cada segundo. Esto añade dificultad.

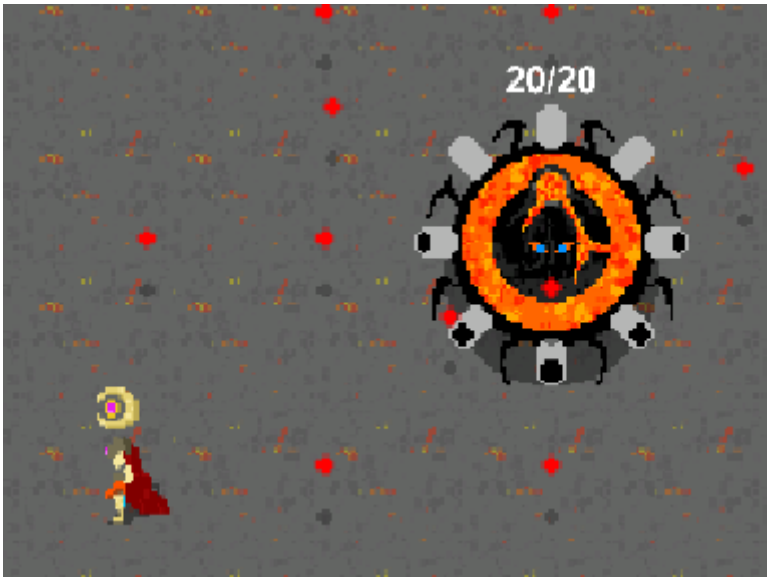


Figura 23: Jefe final atacando

5.8 Desarrollo de extras y corrección de errores

Una vez terminado el grueso del proyecto, se ha desarrollado el dash para el personaje principal. El dash consiste aumentar su velocidad exponencialmente durante una fracción de segundo cuando se pulse la barra espaciadora, de manera que parezca que realiza un salto hacia adelante muy rápido.

Además, para que sea realmente útil, hemos conseguido que mientras el estado del personaje principal sea “dashing”, no pueda recibir daño. Algo muy útil contra el jefe final del nivel, cuando el personaje pueda encontrarse en apuros.

Hemos añadido también elementos que añaden y eliminan vida en el entorno, siendo estos los corazones y los pinchos, respectivamente. Se añaden igual que el resto de los objetos del entorno, pero estos tienen "**Is Trigger**" activado.

En el caso del corazón, al pasar por él, aumenta 1 la vida del personaje principal y se destruye.

Los pinchos, por otro lado, causan daño 1 al personaje principal. No se destruyen.

Se ha añadido un menú principal, en el que se puede quitar la aplicación o comenzar el juego. Si el personaje principal, se ejecuta otro menú en el que se puede reintentar el nivel o salir. También se ha realizado otro menú para cuando se derrota al jefe final indicando al usuario que ha ganado.

Finalmente, se ha añadido también un sistema de Interfaz de Usuario (UI). Mediante un objeto **Canvas**[9] de Unity, hemos podido colocar un sprite diseñado para la vida del personaje principal. A este se le ha añadido una barra verde justo encima. Se ha colocado el **pivot** (es decir, el centro) en su parte izquierda. De esta manera, si cambiamos esta barra de tamaño de manera vertical, lo haría de derecha a izquierda (parecido a como si se estuviera vaciando). Luego ha sido cuestión de vincular la vida del personaje principal con esta barra, de manera que la barra cambiará su tamaño a (**vida actual / vida total) * tamaño de la barra.**

Figura 24: Menú principal

[9] Canvas es un tipo de objeto en Unity que nos permite representar elementos respecto a la pantalla, pudiendo o no cambiar debido a la interacción con el mundo del videojuego.

Corrección final de errores:

Problema encontrado: Se encontró un error relacionado con las colisiones.

Solución: La forma más fácil de arreglarlo fue añadir tipos de capa o layer a los distintos elementos del entorno. De esta manera, gracias a la matriz de físicas 2D de unity, podemos definir cuáles deben interactuar entre sí y cuáles no. Por ejemplo, no nos interesa que la bala de un enemigo golpee a otro enemigo.

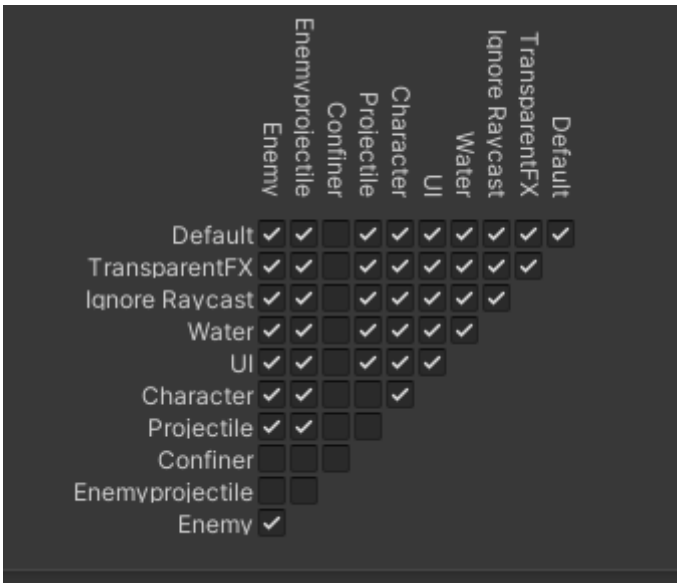


Figura 25: Matriz de colisiones de Unity

Problema encontrado: El dash puede atravesar paredes.

Solución: Definir en el rigidbody del personaje principal "**Collision detection**" a "Continuous".

CAPÍTULO 6
CONCLUSIONES Y LÍNEAS FUTURAS

Una vez finalizado el proyecto, nos encontramos con un nivel completo, diseñado 100% por en Adobe Photoshop y en el que podemos observar un total de tres tipos de enemigos, cada uno más complicado de afrontar que el anterior.

Tras varias pruebas de juego podemos considerar que el objetivo principal del proyecto está cumplido. Se ha realizado el desarrollo de un videojuego en dos dimensiones, de manera que el sistema de combate sea variado y además suponga un reto para el usuario.

Además, se ha desarrollado de manera que dé cabida a futuras ampliaciones y la implementación no sólo de nuevos niveles o habilidades, sino de nuevos enemigos y nuevas mecánicas tales como opciones de diálogo y personaje no jugables que apoyen al usuario en su aventura.

Como amante de los videojuegos, ha sido un placer por fin desarrollar uno propio. Debido a restricciones de trabajo y de tiempo, no ha sido posible completar algunos objetivos tales como la generación procedural de mundos, o mecánicas como salto o puzles.

Sin embargo, quedo muy satisfecho con el resultado y tengo muy claro que, aun habiéndolo completado como proyecto de fin de grado, seguiré desarrollándolo y ampliándolo en un futuro.

Los controles finales del videojuego serían los siguientes:

- WASD o flechas de dirección: moverse
- Z: bloquear
- X: golpeo cuerpo a cuerpo
- C: disparo con pistola
- Barra espaciadora: dash
- Escape: salir del juego

CAPÍTULO 7

SUMMARY AND CONCLUSIONS

Once we have finally finished this project, we encounter ourselves in front a complete level, designed 100% by Adobe Photoshop, in which we can see three different types of enemies, each one more difficult than the last.

After several gaming tests we can considere that the main objective of the project is fulfilled. We've developed a two dimensional videogame, making a challenging and diverse combat system.

Also, we've developed this project so we can expand it further on the future. And we're not talking just about levels or skills, but whole new enemies and new mechanics such as dialogue options and non-player characters that support the protagonist on his adventure.

As a true gamer, it's been a pleasure to finally develop one by my own. Due to time and work restrictions, we couldn't finish some features like procedural world generation, or different mechanics like jumping or puzles.

However, I'm very satisfied with the result and I am really sure that I will continue this project and expanding it on the future.

The final controls for this videogame will be the following:

- WASD or Arrows: move
- Z: block
- X: melee Attack
- C: gun
- Spacebar: dash
- Escape: close the application

CAPÍTULO 8
PRESUPUESTO

Para poder definir un presupuesto tenemos que evaluar, además de la mano de obra, el precio de las aplicaciones que hemos utilizado.

Por un lado, Unity es una herramienta gratuita.

Por otro, Adobe Photoshop tiene un precio de 24.19€ al mes. Desde febrero hasta septiembre, supondría un total de 193.52€.

La aplicación de Visual Studio resulta gratuita para estudiantes universitarios.

Por último, el sueldo por hora de un informático junior torna sobre los 9€ la hora. Partiendo de la base de que este proyecto se ha terminado de realizar en unas 350h, el presupuesto total del proyecto ascendería a los **3.343,52€.**

CAPÍTULO 9
REPOSITORIO DEL PROYECTO

9.1 Enlace al repositorio

https://github.com/alu0100971385/tfg-moonboy.git

BIBLIOGRAFÍA

Unity Learn. (n.d.). Unity Learn Service. Consultado constantemente a lo largo del proceso del proyecto - https://learn.unity.com/

Mister Taft. (2017). Mister Taft Creates youtube channel. Consultando constantemente a lo largo del proceso del proyecto.
https://www.youtube.com/channel/UCZczqDvepgNqy80gTMGnUXw

Stack Overflow. (n.d.). Stack Overflow en español. Consultando constantemente a lo largo del proceso del proyecto - https://es.stackoverflow.com/