



# VNiVERSiDAD D SALAMANCA

CAMPUS DE EXCELENCIA INTERNACIONAL

## **Práctica 1 Unity: "The Odissey"**

**Animación Digital**

**Grado en Ingeniería Informática**

**Luis Blázquez Miñambres, 70910465Q**

## Contenido

1.	Introducción .....	3
2.	Manual de Juego .....	4
3.	Desarrollo del Juego .....	5
4.	Características .....	16
<a href="#">a.</a>	Jugador principal.....	16
<a href="#">b.</a>	Enemigos.....	19
<a href="#">c.</a>	Tilemaps.....	21
<a href="#">d.</a>	Cofres.....	23
<a href="#">e.</a>	Monedas y diamantes.....	25
5.	Fallos e implementaciones futuras .....	26
6.	Bibliografía .....	27

## 1. Introducción

El juego que pretendía realizar es una especie de juego de acción y aventuras RPG en 2D al estilo del conocido juego “*The Legend of Zelda*” donde vas luchando contra diferentes enemigos y recogiendo objetos hasta enfrentarte a un enemigo final. En mi caso le he añadido mi propia historia, tomando como base para el desarrollo de este los conocimientos y algunas de los recursos aprendidos y utilizados en las prácticas en 2D de los aviones y de plataformas como la utilización de la vida de los personajes, los enemigos, disparos, etc.

Aunque es verdad que he desarrollado el juego desde cero, pero partiendo de muchas cosas ya hechas en clase, para el diseño del escenario he tomado referencia de algunos videos pertenecientes a esta guía.

En el juego controlamos a un personaje principal que tiene la posibilidad de elegir entre tres armas principales y jugar como tres personajes distintos con diferentes habilidades cada uno para ir avanzando a través de las dos zonas del juego luchando contra los enemigos y recogiendo monedas hasta llegar al enemigo final, al que se enfrentará de una manera u otra dependiendo del arma que haya escogido.



*Imagen 1.1: Vista previa de una ejecución del juego*

## 2. Manual de Juego

Para controlar al personaje principal, en la primera pantalla del juego nada más empezar, en la parte derecha muestro los controles del juego. En concreto:



**Atacar con arma principal:** Botón Izquierdo del ratón



**Atacar con arma secundaria/poder especial:** Botón derecho del ratón



**Mover al personaje:** Teclas W, A, S, D o flechas de dirección



**Interactuar con cofres y postes:** Tecla E

### 3. Desarrollo del Juego

Al comienzo del juego traté de buscar los *sprites* adecuados para mi personaje y los enemigos a los que se iba a enfrentar. Aunque para la idea que quería hacer, necesitaba un *sprite* que se pudiese personalizar de alguna manera o tuviera diferentes *skins*. Es por ello, que encontré una [colección de sprites](#) orientada a este tipo de juegos en la página OpenGameArt.org. Y a partir de realizar varias búsquedas sobre esta colección, conocida como LPC, encontré una [herramienta](#) que permitía crear tus propias *sprites*, personalizándolos y creándolos a tu antojo de una manera más simple y sencilla que la utilizada en la herramienta RPGMaker creada por un usuario y publicada en su repositorio de Github.

Así fue como cree al personaje principal, así como sus diferentes sprites de movimiento y ataque tanto para él como para los otros 3 personajes. Todos los sprites referentes a estos personajes se pueden encontrar en la carpeta **Sprites > MyAssets > Player** en la carpeta Assets.



Imagen 3.1: Personaje principal



Imagen 3.2: Personaje principal convertido en Jedi

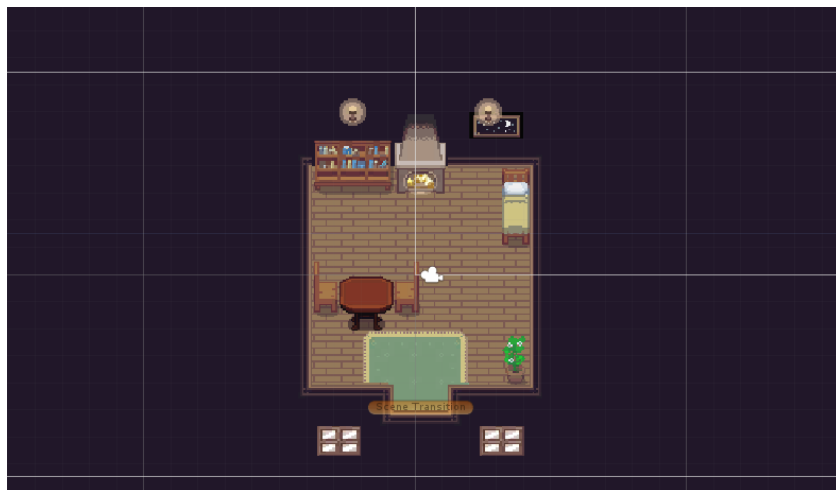


Imagen 3.3 Personaje principal convertido en Thor

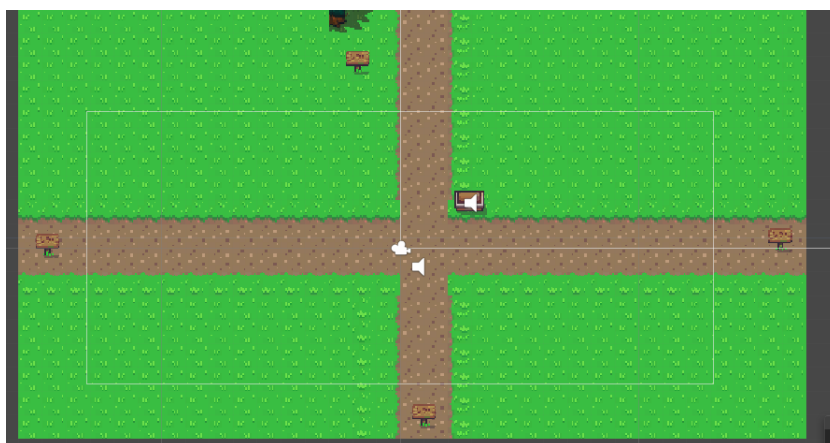


Imagen 3.4: Personaje principal convertido en Caballero Medieval

A partir de aquí empecé a plantear las distintas zonas que tendría el juego, en concreto 3: una principal en un bosque, otra en el desierto y otra subterránea volcánica, cada una con distintos tipos de enemigos y un jefe en cada zona. Finalmente, por falta de tiempo, realicé tres escenas principales: el menú principal (que es la casa del personaje), la zona del bosque y la zona volcánica, desechando la zona del desierto.



*Imagen 3.5: Versión principal y final del menú de juego*



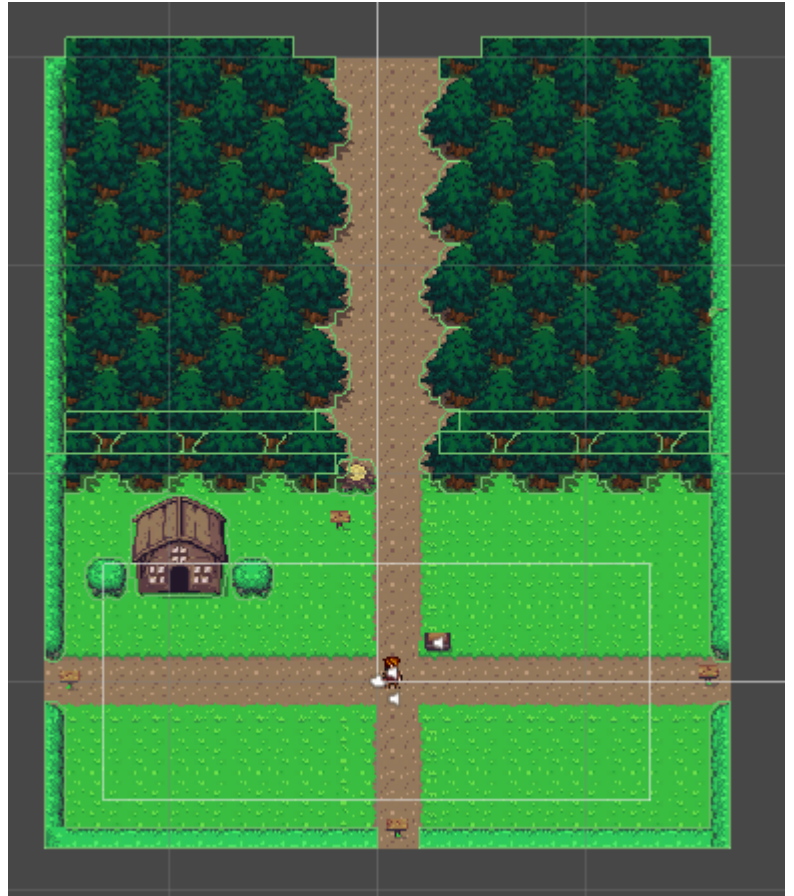
*Imagen 3.6: Versión inicial del primer escenario de la zona 1*



*Imagen 3.7: Segunda versión del primer escenario de la zona1. El jugador sale del menú principal a la puerta de la casa*



*Imagen 3.8: Versión primaria del bosque que conecta el primer escenario con el segundo*



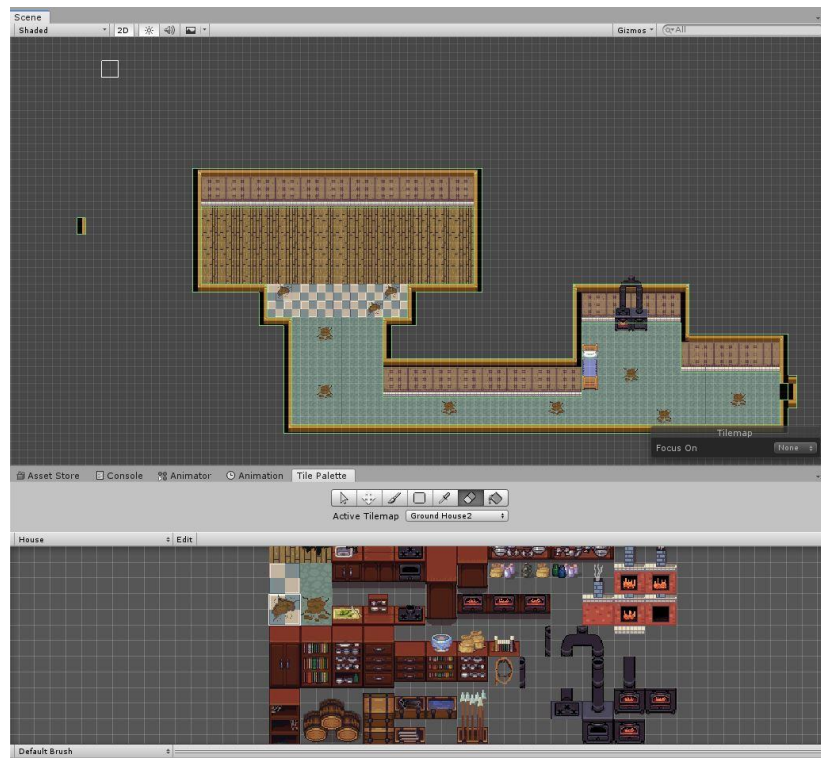
*Imagen 3.9: Versión final del primer escenario de la zona1 juntando el escenario 1 y el escenario del bosque.*



*Imagen 3.10: Versión del que primero sería el segundo escenario y acabaría siendo el tercero. Después de entrar en la casa y escoger un personaje*



Fue entonces cuando decidí que el jugador escogiera un personaje a partir de unos cofres que estarían en otro escenario de la primera zona: una casa dentro de otro escenario. Que quedarían de la siguiente manera, como zonas intermedias, donde a través de una puerta el jugador accedería a la casa.



*Imagen 3.10: Primer fase de desarrollo de la casa interior con los cofres*



*Imagen 3.11: Versión final del interior de la Casa donde se encuentran los cofres que contienen la selección de personajes*





Imagen 3.12: Segundo escenario de la zona1 al que se transporta el jugador

Para realizar el teletransporte de un escenario a otro de cada una de las zonas, ya fuera la primera zona que he mostrado anteriormente o la zona de lava, utilicé un *GameObject* que posteriormente convertí en un *Prefab*, ya que iba a utilizarlo muchas veces de la misma manera, que estaba compuesto por dos óvalos, uno azul y otro rojo. Al *GameObject* del óvalo azul le añadí un componente *BoxCollider* con *IsTrigger* activado de manera que me sirviera para detectar cuando el personaje se chocara con él y realizar una acción desde un script asignado a este objeto.

En este caso, lo que haría el código en el método *OnEnterTrigger2D* sería ver si es el jugador el que ha chocado con él y cambiar su posición a la de inicio de cada zona.

Por tanto, el óvalo azul indica la salida del personaje de esa zona y el óvalo rojo es la posición al que el personaje se va a mover desde la zona origen desde la que se ha teletransportado (los sprites de los óvalos los escondo ya que simplemente me servían para colocar correctamente las zonas de transporte).

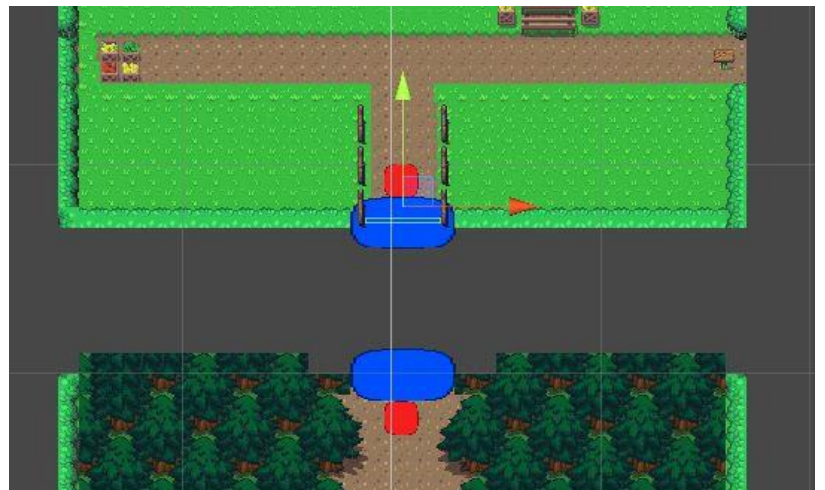


Imagen 3.13: Objetos de transporte entre escenarios de las zonas



Imagen 3.13: Vista previa de la primera fase de desarrollo de la zona 2

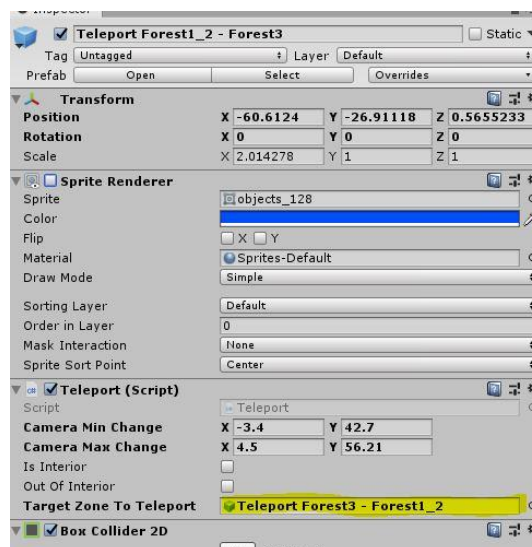


Imagen 3.14: Inspector del objeto de teletransporte. En concreto el que pasa del escenario 1 al escenario 2 de la casa, para ello se pasan las coordenadas de cambio de la cámara y el objeto de teletransporte de destino

Es importante añadir que las coordenadas a las que se transportaba el personaje las determiné yo manualmente para el transporte a cada una de las zonas, así como la altura y anchura máxima de cada escenario, con el fin de que no se vieran los bordes fuera de lo que era el Tilemap. Esto es debido a que, al tratarse de Tilemaps creados manualmente por mi dentro de Unity utilizando la herramienta *Tile Palette* en la que recortaba los sprites en trozos de 16 bits para poder tener más libertad a la hora de coger terrenos o texturas de una parte de un sprite y no un sprite entero.

Todo esto con tamaños de ancho y alto distintos para cada escenario de la escena, por lo que no había manera de tener un control sobre las dimensiones de cada escenario en concreto

desde el código, lo que permitiría automatizar el proceso de transporte sumando o restando posiciones de una manera sencilla a la posición de la cámara y el jugador de manera automática. Esto se podría hacer si se creasen los Tilemaps predefinidos con otros programas como “Tiled Map Editor”.

Una vez creadas todos los escenarios de la primera zona, los escenarios de la siguiente escena o zona, la referente a la lava, la podría realizar de manera análoga.

A continuación, realicé toda la interfaz visual del HUD, con la vida, barra de energía, acumulador de monedas y cajas de dialogo o contexto. Para la barra de energía azul cogí un sprite del paquete LPC que indiqué anteriormente. Esta barra al tratarse de una imagen, lo que hacía para simular que la energía bajaba cada vez que el personaje realizaba su habilidad secundaria era reducir las dimensiones de la imagen a lo largo hacia la izquierda, proporcionalmente a la energía consumida. Esto lo he cogido del juego realizado en clase con los aviones, en la que reducíamos la barra de vida, posicionada estáticamente a la parte izquierda de la interfaz (como la figura del canvas).

```
// Change the image display of the barr  
manaBarr.rectTransform.sizeDelta = new Vector2( x: manaBarr.rectTransform.sizeDelta.x - 30, manaBarr.rectTransform.sizeDelta.y);
```

Imagen 3.15: Reducción del tamaño de la imagen para reducir la energía en la barra de energía

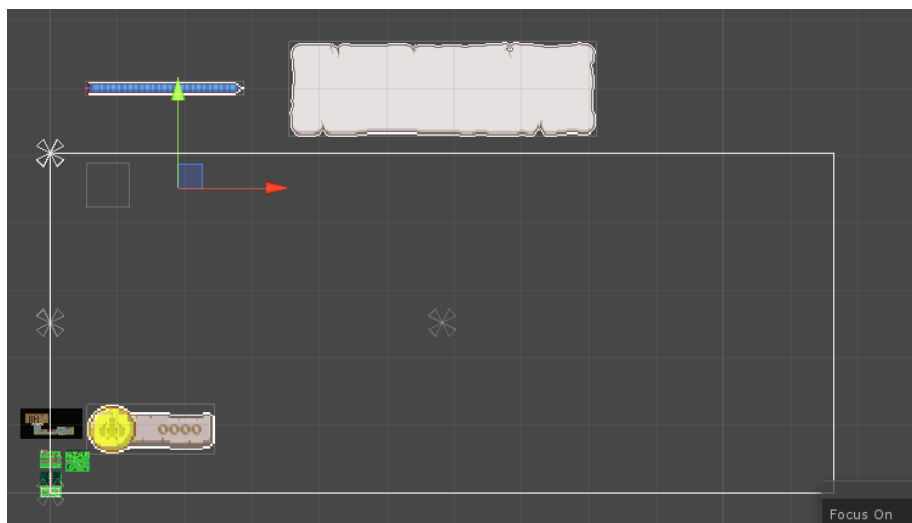


Imagen 3.16: Elementos del Canvas que forman parte del HUD del juego

En cuanto a la vida del jugador, he utilizado unos sprites del paquete LPC. Para escoger la vida del jugador he utilizado elementos llamados *ScriptableObject* a partir de un script que he llamado *FloatValue.cs* que hereda de *ScriptableObject* que indica la vida del jugador y permite indicar cuantos corazones tendrá en la interfaz.

Estos son objetos que se pueden crear en Unity que tienen propiedades o atributos determinadas y se pueden utilizar como las clases en un lenguaje orientado a objetos, pero de manera gráfica. Se encuentran en la carpeta **ScriptableObjects** en la carpeta Assets. Estos corazones cambian del sprite con el corazón relleno completamente al corazón partido a la mitad cuando el jugador recibe un golpe y del corazón medio lleno al corazón vacío cuando recibe otro golpe. De manera que cada corazón en la interfaz equivale a dos vidas. El valor de la vida del jugador se puede cambiar desde la propiedad del *ScriptableObject*.

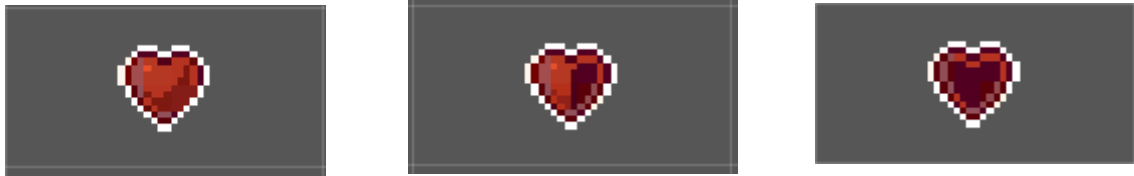


Imagen 3.17: Sprites de la vida del jugador

Estos objetos los añado, en este caso, al script que controla el movimiento y acciones del jugador “PlayerMovement” y me permite coger el valor que yo indique desde la referencia a la instancia de ese objeto, cambiando únicamente el valor que desee desde el inspector sin tener que hacer cambios en el código permitiendo un mayor nivel de abstracción y desacoplamiento.

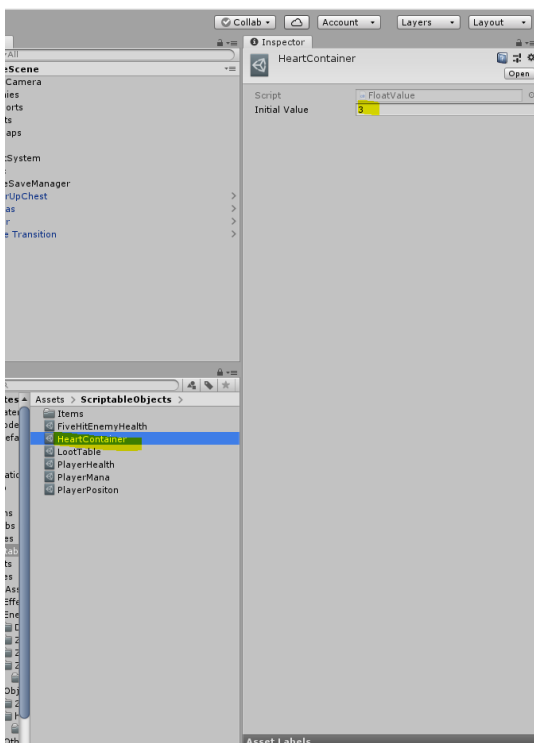


Imagen 3.18: ScriptableObject del numero de corazones del HUD

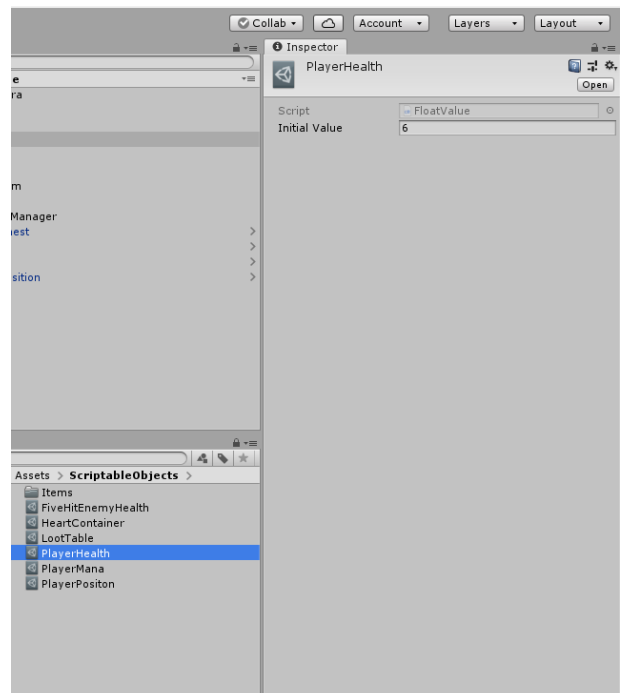


Imagen 3.19: ScriptableObject del valor la vida del jugador



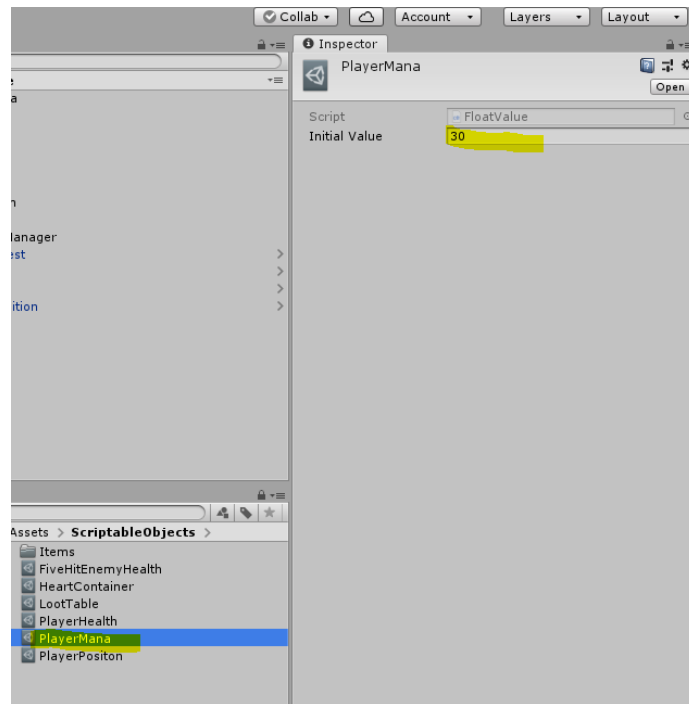


Imagen 3.20: ScriptableObject del valor del maná o energía del jugador

Finalmente falta añadir los enemigos a la escena, pero las características de cada uno lo explicaré más adelante en siguientes apartados.

Por último, cree la segunda zona de la que no entraré en detalles ya que la cree en base a lo que había creado en la primera zona, pero con distintos sprites y distintos diseños. Se puede ver como realicé la transición entre distintas escenas, para ello cree un objeto (que más tarde convertiría en prefab) que se llama *Scene Transition* con varias propiedades que explicaré a continuación. Este tipo de objeto lo utilicé tanto para la transición de la escena del menú principal en la casa del jugador a la zona 1 como del paso de la zona 1 a la zona 2.

La diferencia es que, en la transición del menú principal a la zona1, no mantuve la referencia del Jugador ya que no había empezado el juego aún. Mientras que en la transición de la zona 1 a la zona 2 permito que el jugador recargue la vida y la barra de energía entre escenas, pero mantengo el inventario (esto son número de monedas y arma y poder del personaje elegida en uno de los cofres).

Para que la transición se realice de manera menos forzada y más fluida cree una animación con un panel blanco que aparecía y desaparecía y las metí como dos argumentos *FadeInPanel* y *FadeOutPanel* en el código del script *SceneTransitionManegment.cs*, respectivamente. Esto lo hice mediante la realización de una subrutina que permitía realizar dos acciones de manera paralela.

Por un lado, crear el panel blanco de transición de escenas mientras se cargaba la escena y se movía la posición del jugador y se configuraba el cambio. Para que esto se pudiese hacer de manera paralela y asíncrona he utilizado el método *LoadSceneAsync()* junto con el nombre de la escena a la que se quiere hacer la transición.

Por otro lado, como ya comentaba, para mantener objetos entre escenas lo que he hecho ha sido añadir el método *Awake()*, que se ejecuta una vez por escena, en los scripts de los objetos que quería mantener. En este caso, este es el método *Awake()* perteneciente al jugador principal en su script *PlayerMovement.cs*, indicando mediante el método *DontDestroyOnLoad()* de no destruir la referencia al objeto del jugador al cargarlo en la escena. También añadir que he hecho lo similar para mantener el inventario del jugador, sus moedas y todo lo que se quiera mediante el script *GameSaveManagerObjects.cs* en un *GameObject* llamado *GameSaveManager*.

```
private void Awake()
{
    playerInventory.coins = 0;

    if (SceneManager.GetActiveScene().name.Equals("SampleScene"))
    {
        DontDestroyOnLoad(target: this);
    }
}
```

Imagen 3.21: Método *Awake()* del script del jugador principal

Por ejemplo, en el caso de la transición de la escena de la zona1 a la escena final o zona2 del volcán. Aquí podemos ver donde está colocado el *gameObject* con el *BoxCollider* que detecta cuando el jugador pasa por él.

Cabe destacar, que otro de los objetos que no se destruyen al pasar de una escena a otra es el objeto *PowerUpChest*, que es un cofre que aparece justo antes de pelear contra el jefe final y convierte al jugador en el personaje que ha elegido. Pero para mantener la referencia al jugador principal desde el inicio del juego lo que hago es mantener el cofre dentro de la zona1 inhabilitado y lo mantengo para la zona 2 en la posición que yo determino habilitándolo.

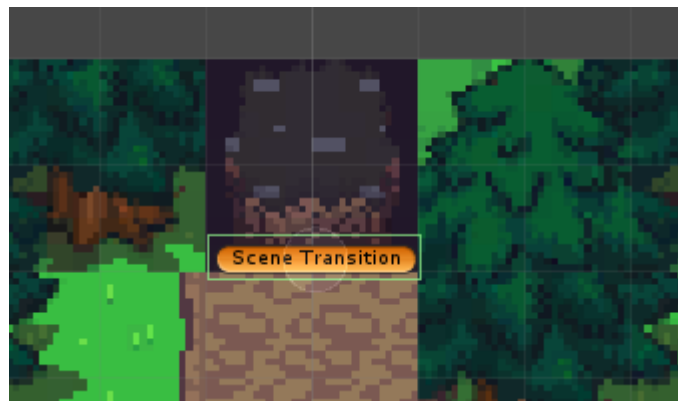


Imagen 3.22: Objeto *Scene Transition* de transición a la siguiente escena



Para preparar todo lo necesario para la zona 2, aunque no creo que sea el método más adecuado para hacerlo, es crear un `GameObject` en la escena llamado *SetupScene* al que le asocio un script que en su método *Start()*, realiza todas las acciones propicias para preparar la escena. Esto es, actualizar el contador de monedas del nuevo *Canvas* con las monedas del inventario del jugador, asociar la cámara de esta escena al jugador (esto lo hice porque al pasar la cámara de una escena a otra tuve problemas donde la cámara se iba siempre a una posición fuera del mapa de juego) y mover el, ya mencionado, *PowerUpChest* a su posición.

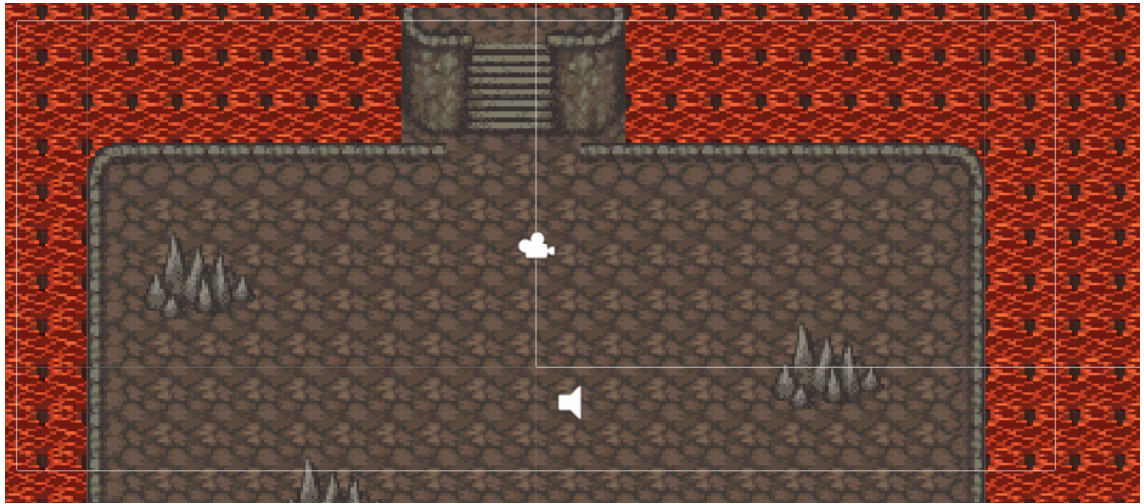


Imagen 3.23: Comienzo de la escena final *FinalScene*

## 4. Características

A continuación, se detallarán las características de los objetos utilizados en el juego, ya estén relacionados o no con el jugador, como los enemigos, objetos coleccionables, o los mapas creados.

### a. Jugador principal

En cuanto al personaje principal, caben destacar varias cosas. La primera es que, como ya mencioné en la introducción, el jugador puede elegir ser uno de los tres personajes que el juego permite, una vez elige abrir uno de los tres cofres en el escenario de la casa en la zona 1. Una vez elegido, este pasa a utilizar su arma y sus habilidades. Para ello, pensé en crear 3 *GameObjects* distintos (uno para cada personaje) que estuvieran dentro del objeto *Player*, pero me pareció una solución ineficiente y algo compleja.

Por lo que después de probar varias soluciones, encontré la solución de cambiar al *AnimatorController* del personaje correspondiente cuando el jugador abriese el cofre *PowerUpChest*. Pero antes de llegar a ese punto, el jugador podía utilizar las habilidades de su personaje.

En concreto:

- Tanto para el ataque principal como para el ataque secundario he utilizado corutinas, ya que permitían hacer varias cosas de manera paralela sin parar al personaje, haciendo sus movimientos y animaciones más creíbles.
- El personaje del Jedi es capaz de lanzar arcos de fuerza como habilidad especial y el personaje del Caballero Medieval es capaz de lanzar flechas. Para ello, dentro de la corutina *SpecialPowerCoroutine()*, después de crear la animación de lanzamiento, instancio el objeto correspondiente en la escena y les añado dos componentes fundamentales, *RigidBody* para que tengan físicas y *BoxCollider* para que al impactar contra el enemigo le hagan daño. Es por ello por lo que a través del script *Throwable Object.cs* añado velocidad y movimiento al proyectil de cada personaje y a través del script *Knockback Melee* añado empuje al impactar contra el enemigo (este script también está añadido para el ataque con el arma principal).
- El personaje de Thor tiene un comportamiento diferente en cuanto al lanzamiento de su poder secundario. Lo primero cree una animación para el lanzamiento del martillo, de manera que pareciese que al lanzarlo llega e impacta en el enemigo y vuelve de nuevo al jugador. Para ello, dentro de la animación, tuve que cambiar la posición del martillo para cada una de las direcciones en las que apuntase el personaje. Esto lo utilicé añadiendo el componente *Position* a la animación y luego a través de la opción *Curve* modificar la curva de color

rojo (eje X o Y dependiendo de la posición) en función de como quería que se moviese el martillo.

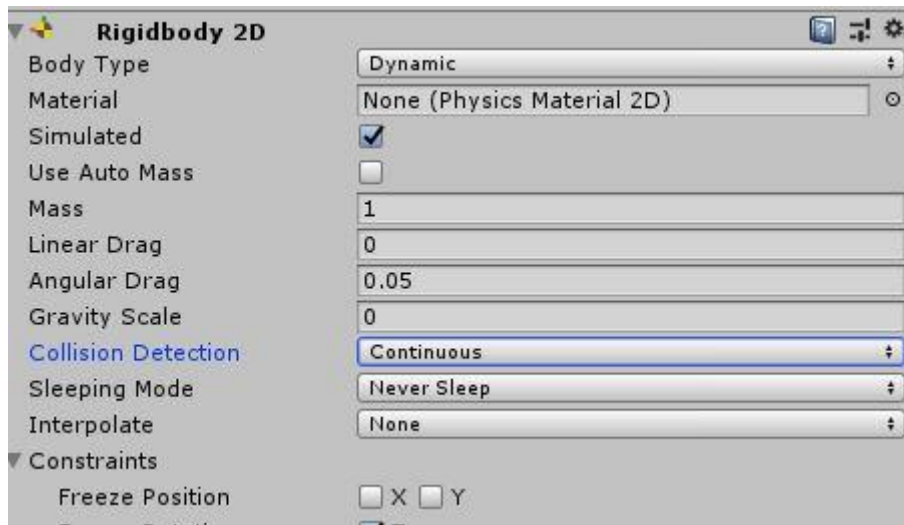


Imagen 4.a.1: Componente RigidBody del martillo

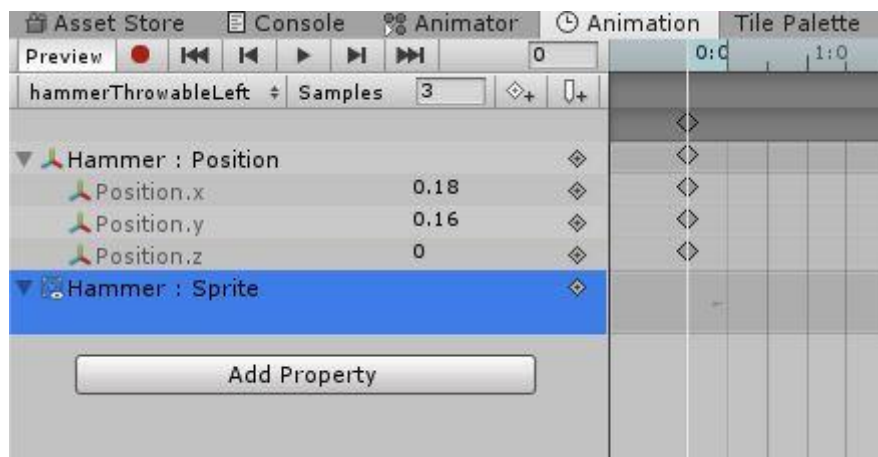


Imagen 4.a.2: Animación de movimiento del martillo

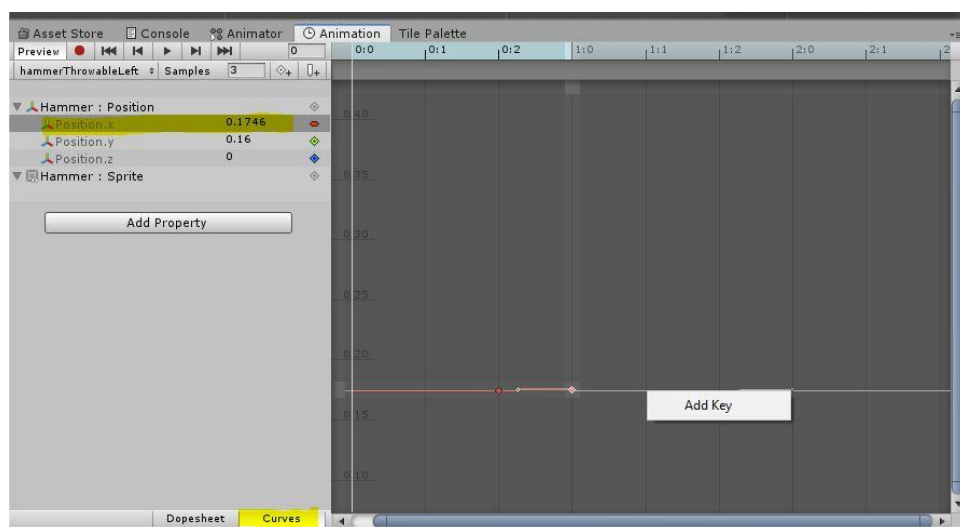


Imagen 4.a.3: Seleccionamos el componente de la posición que queremos modificar en Curves

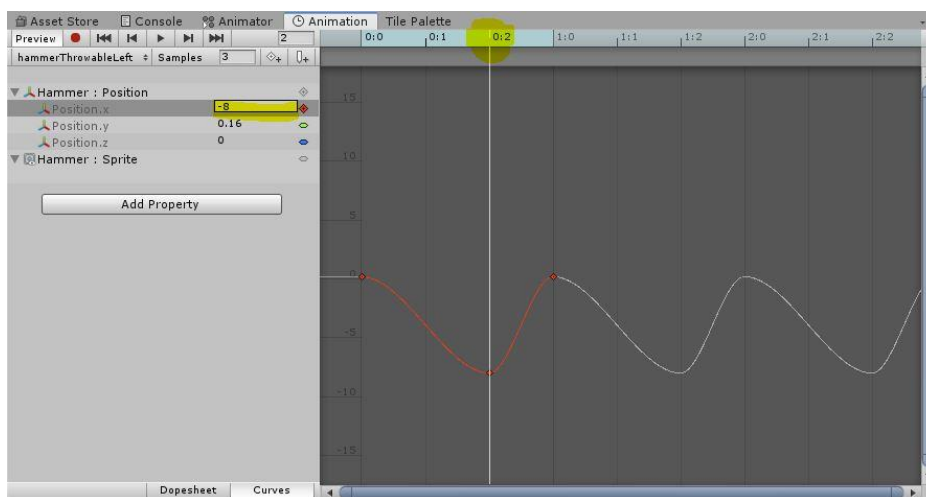


Imagen 4.a.4: Animación de movimiento del martillo

A continuación, explicaré los objetos y valores que se le pasan al script del jugador. Como podemos observar, aparte de los *ScriptableObjects* para la vida y el maná o energía, añadimos la imagen del canvas de la barra de energía para reducir su tamaño a medida que utilizamos el poder secundario, así como el inventario del jugador para almacenar las monedas y el arma principal del personaje que se escoja. En cuanto a los proyectiles, son los objetos prefab que se instanciarán cuando se utilice el poder secundario. A continuación, los *AnimatorController* de cada uno de los personajes, en función de cual se escoja. Y los clips o sonidos que se reproducirán en el componente *AudioSource* al atacar con el arma principal.

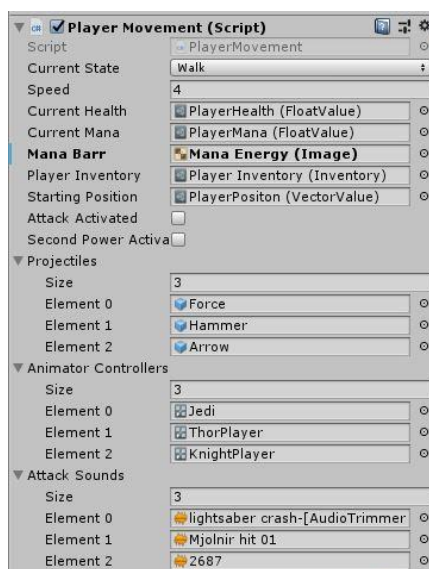
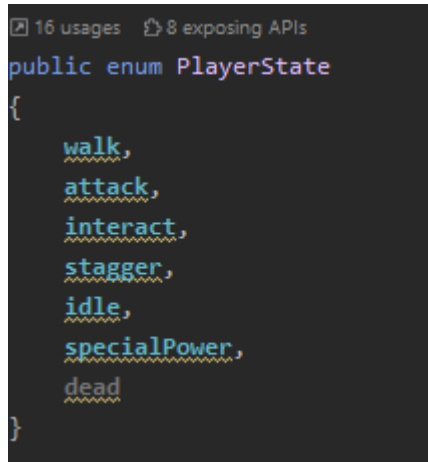


Imagen 4.a.5: Inspector del script de control de jugador principal

También cabe destacar, que añadí una máquina de estados en el código del jugador para saber en cada momento el estado en el que se encontraba, y hacer más versátil y fácil que el personaje pudiese realizar determinadas acciones en función de su estado en ese momento. De manera que programar en el script se hace más sencillo.

A screenshot of a code editor showing a C# enum definition. At the top, it says '16 usages' and '8 exposing APIs'. The code is: 

```
public enum PlayerState
{
    walk,
    attack,
    interact,
    stagger,
    idle,
    specialPower,
    dead
}
```

Imagen 4.a.6: Máquina de estados del jugador principal

## b. Enemigos

En cuanto al comportamiento de los enemigos, cree un objeto prefab *Enemy* del cual se pueden crear otros prefab que hereden de este. En concreto, el juego contiene tres enemigos con las mismas características salvo algunas excepciones.

Todos tienen su propio script de control que heredan de una clase padre en el script *Enemy.cs*, ya que todos comparten las mismas características y atributos.

El comportamiento que todos estos enemigos comparten es que, dentro de su *Animator Controller*, se mantienen en el estado *Sleep* que indica que están en reposo. Y cuando el jugador aparece en su radio de ataque estos pasan al estado de ataque, acercándose al jugador y tratando de atacarle. Si el jugador se aleja de su radio de ataque, estos vuelven a su estado inicial de reposo o *Sleep*. Cabe destacar que dentro del script de este prefab se puede cambiar tanto el valor del daño del enemigo como su radio de ataque y su velocidad, de manera similar a como ocurre con el jugador.

El cálculo del radio de ataque lo he hecho comparando la distancia entre las coordenadas de la posición del jugador y las coordenadas de posición del enemigo y comprobar si están son menor que el valor del radio de ataque que se ha dado en el script.

En función de la posición del jugador, ya mencionada, el enemigo se moverá en la dirección y hacia la posición del jugador con el método:

*MoveTowards(origen, destino, velocidad)*

Y en función de la posición del jugador en los ejes X e Y , el enemigo cambiará de animación hacia una dirección u otra, de manera que el enemigo siga al jugador para atacarlo.

De manera análoga a como ocurría con el jugador, el ataque se realiza mediante una subrutina con el mismo funcionamiento que como ocurría con el jugador, y una máquina de estados que facilita la comprensión y realización de la lógica interna del código.

```

1 usage -> More
void CheckDistanceOfPlayer()
{
    if (!playerDead)
    {
        if (Vector3.Distance(a: target.position, b: transform.position) <= chaseRadius &&
            Vector3.Distance(a: target.position, b: transform.position) > attackRadius)
        {
            if (currentState == EnemyState.idle ||
                currentState == EnemyState.walk && currentState != EnemyState.stagger)
            {
                Vector3 temp = Vector3.MoveTowards(current: transform.position,
                                                    target.position,
                                                    (maxDistanceDelta: moveSpeed * Time.deltaTime));
                ChangeAnim( direction: temp - transform.position);
                myRigidBody.MovePosition(temp);
                ChangeState(EnemyState.walk);
                _animator.SetBool( name: "wakeUp", value: true);
            }
        }
        else if (Vector3.Distance(a: target.position, b: transform.position) > chaseRadius)
        {
            _animator.SetBool( name: "wakeUp", value: false);
            _animator.SetBool( name: "attackPlayer", value: false);
        }
    }
}

```

Imagen 4.b.1: Código de los enemigos donde se comprueba si el jugador está cerca o no y como mover el enemigo

Adicionalmente, cuando el enemigo se produce una animación que al finalizar llama a un evento que destruye la instancia del objeto del Enemigo y llama a la función *MakeLoot()* que deja caer, o no, uno de los objetos añadidos al ScriptableObject “*LootTable*” dependiendo de su probabilidad.

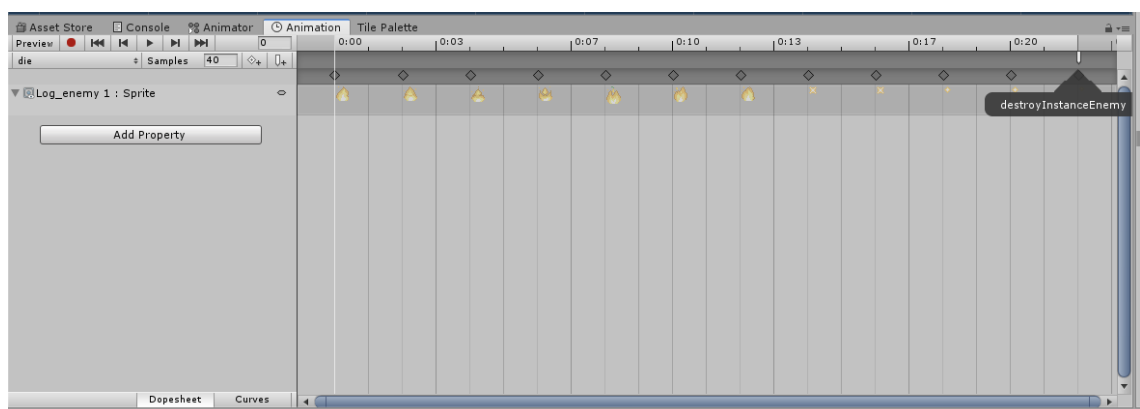


Imagen 4.b.2: Animación de muerte del enemigo. Al final de la animación se llama mediante un evento al método que destruye la instancia del objeto del enemigo

La única diferencia con el resto es que el enemigo *Log* no tienen ninguna animación de ataque solo camina hacia el enemigo y lo ataca mientras que el resto



al estar cerca del enemigo comienzan una animación de ataque. Otra diferencia es que el jefe final tiene un *Box Collider* más pequeño para que le sea más difícil al jugador atacarle, así como contar con más salud y más daño de ataque.

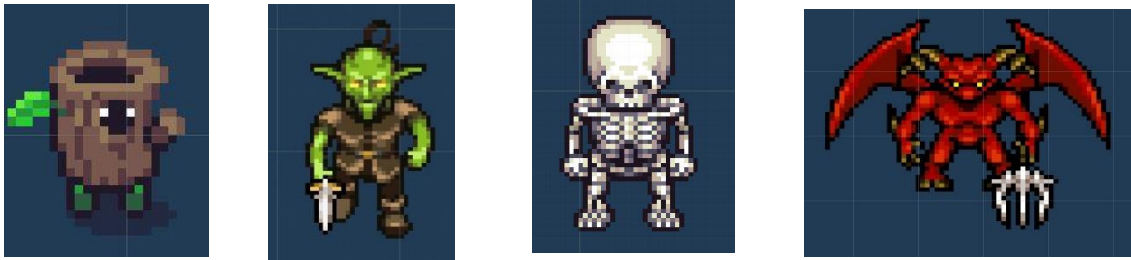


Imagen 4.b.3: Enemigos del juego. En orden: Log, Goblin, Skeleton, The Demon (jefe final)

### c. Tilemaps

En el caso de los Tilemaps, lo primero que hice fue configurar sus sprites con los parámetros que se ven en la *imagen 4.c.1* y que señalo, todo ello para conseguir dividir los elementos de los sprites en cuadrados de 16 pixeles y sin ningún tipo de filtros ni compresión, para conseguir la calidad original del sprite.

Posteriormente, para cada una de las zonas era imprescindible que añadiese dos tipos de objetos. Por un lado, un tipo de *GameObject* de tipo Tilemap que sería el que he utilizado para pintar el suelo de todos los escenarios. Y, por otro lado, otro tipo de *GameObject* con el que pintaría los objetos delimitantes de una escena que el jugador no podría atravesar (los bordes de los escenarios, paredes, mesas, etc) que contenían los componentes *RigidBody2D*, *TilemapCollider* y *Collider2D*, estos dos últimos con la opción de *Composite* activada indicando que el collider se activará por composición de los elementos que dibuje en la escena.

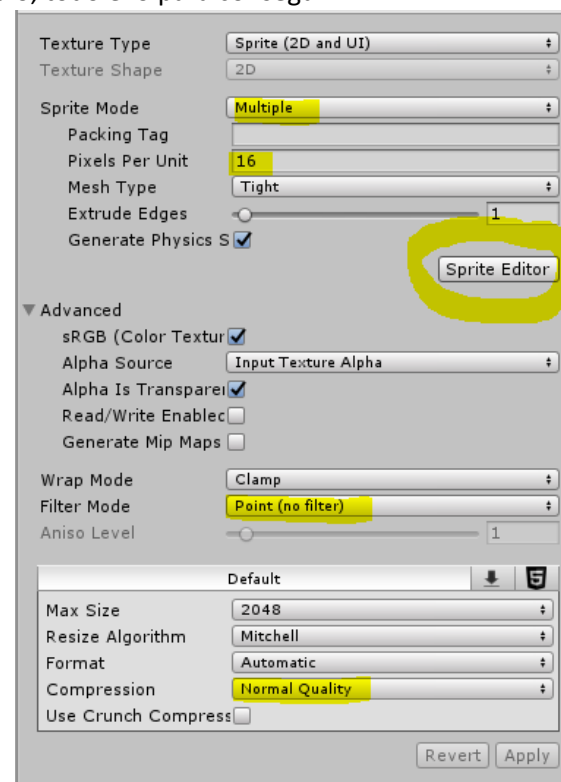


Imagen 4.c.1: Configuración para editar los sprites de los Tilemaps

Por tanto todos los elementos que aparecen los escenarios los he pintado yo en las escenas utilizando los sprites a través del Tilemap Editor, como ya comenté en la introducción.



Imagen 4.c.1: Ejemplo de la capa de elementos con los collider de los Tilemaps

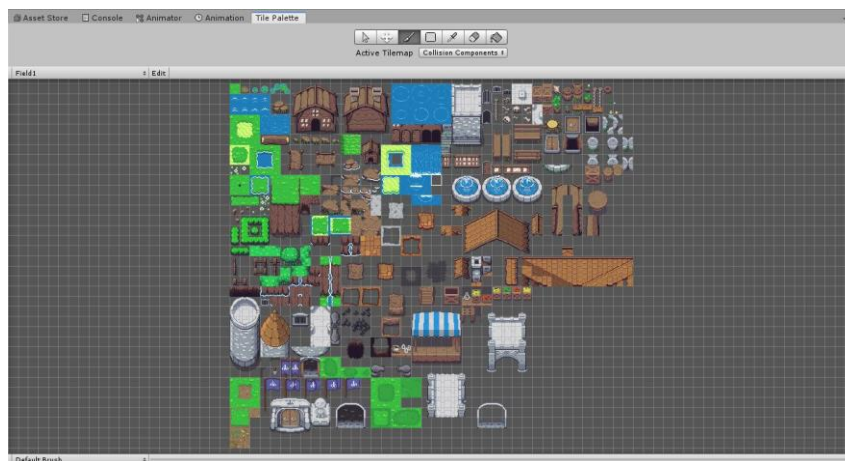


Imagen 4.c.2: Tilemap usado para crear los escenarios de la zona 1 (escena SampleScene)

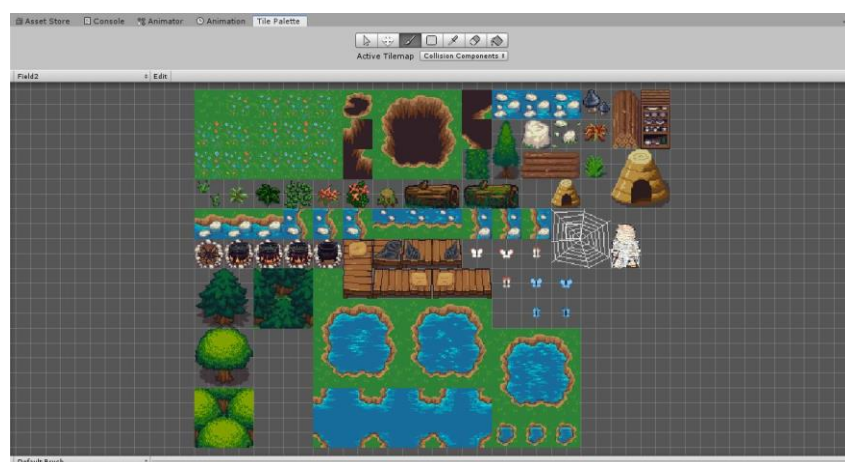


Imagen 4.c.3: Tilemap usado para crear los escenarios de la zona 1 (escena SampleScene)

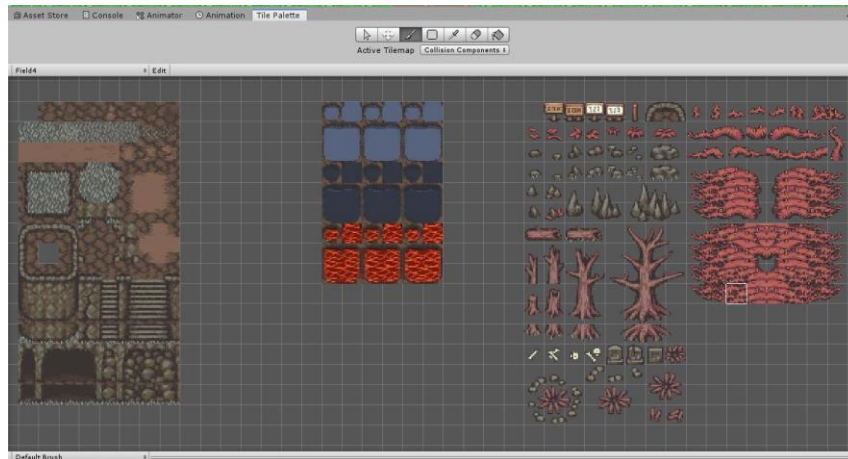


Imagen 4.c.4: Tilemap usado para crear los escenarios de la zona 2 (escena FinalScene)

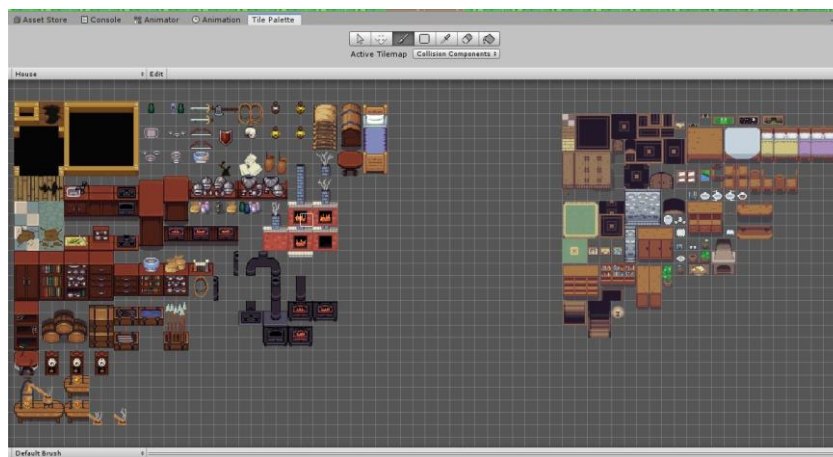


Imagen 4.c.5: Tilemap usado para crear los escenarios de las escenas de interiores (casas)

## d. Cofres

Uno de los elementos más importantes del juego es el cofre, del cual hay dos tipos. Cabe mencionar dos tipos de cofre tienen el mismo tipo animación.

Por un lado, están los cofres de selección de personaje que se encuentran al principio del juego y que permiten escoger el personaje con el que jugaremos el resto del juego. Y por otro lado está el cofre llamado *PowerUpChest*, que convierte al jugador principal en el personaje que ha elegido con su skin, antes de pelear contra el jefe final.

Cuando se abren los primeros tipos de cofres, de selección de personajes, cada cofre tiene asignado un ID (0, 1 o 2) en función del tipo de personaje. Este valor se le asigna al elemento *weaponID* del inventario del jugador para saber que personaje ha elegido y cual está utilizando en cada momento ya que las animaciones de los tres personajes, así como las habilidades es distinta. En este caso hay métodos más eficientes de hacerlo, por ejemplo con clases (una para

cada tipo de personaje) y un array de objetos de esta clase, pero con el fin de hacerlo menos complejo, decidí en primera instancia hacerlo así.

El comportamiento de ambos es el siguiente, una vez el jugador interactúa con él (como ya expliqué anteriormente) a través de un *Collider* con *IsTrigger* activado que detecta al jugador en la zona y despliega el *GameObject* con el bocadillo encima de su cabeza con la interrogación, este abre el cofre y ocurren varias cosas. La primera es, que el jugador tiene un *GameObject* consigo con el elemento *Sprite Renderer* desactivado que se llama "Received Item" y que es el objeto que recibirá el jugador al abrir un cofre, cada cofre tiene un objeto distinto. Además cada cofre tiene asignada una música que sonará en el componente *AudioSource* asignado cuando este es abierto y se parará cuando el jugador pulsa el botón **E**, cerrando el cuadro de dialogo. En el caso del *PowerUpChest*, la música sonará indefinidamente.

Cuando un cofre es abierto, el script del jugador recoge el sprite asignado en el script del cofre y se lo asigna al *GameObject* del jugador con nombre "Received Item" y lo muestra. Lo segundo que pasa es que se reproduce la animación *ReceiveItem* activando el elemento de tipo bool *receiveItem* en el Animator Controller del jugador, que refleja que el jugador está sosteniendo en el aire el objeto.

Finalmente cabe destacar, que en el caso del cofre *PowerUpChest*, además de tener el mismo comportamiento, este activa la animación que ya mencioné antes de que muestra un destello blanco para la transición entre escenas y cambia el Animator Controller al del personaje que haya escogido en función del valor del *weaponID* en el inventario del jugador, que como ya mencioné, indica el personaje escogido por el jugador.



Imagen 4.d.1 Acción de apertura de un cofre (izq). Obtención del objeto(centro). Cambio de Animator Controller en PowerUpChest (der)

## e. Monedas y diamantes

Como todo juego de este estilo es usual añadir un sistema de recogida de objetos coleccionables, en este caso un recuento de monedas, que, aunque no tienen ningún valor real dentro del juego más allá de su recogida y recolección, estaba pensado para servir para otro propósito como el comercio por otros objetos. Aunque esta idea se descartó por falta de tiempo.

En el juego, habrá dos objetos que hagan este recuento. Estos objetos contarán con un *BoxCollider* con la opción *IsTrigger* activado para que, únicamente si el jugador “colisiona” con ellos, hagan desaparecer la instancia del *GameObject* y contarlos al colector de monedas en el HUD. Por un lado, las monedas que contarán por el valor de 1 moneda, y por otro lado los diamantes, que tendrán el valor de 5 monedas. Estos objetos únicamente serán dejados por los enemigos al derrotarlos.

Para realizar esto, cree un *ScriptableObject*, ya mencionado anteriormente con su script correspondiente que hereda de *ScriptableObject* *LootTableManagement.cs*, que se llamaba “*LootTable*” en la carpeta **ScriptableObjects** y que contiene las referencias de los *GameObject* de los objetos que los enemigos pueden dejar caer al ser derrotados (monedas o diamantes) y la probabilidad de que estos objetos aparezcan. Cabe destacar que sobre el 100% de loot, hay un 25% de probabilidades de que los enemigos no suelten nada. Este método permite ser lo más extensible posible y añadir cuantos objetos de *loot* queramos y su probabilidad de aparición.



Imagen 4.e.1: Objetos coleccionables prefab diamantes y moneda

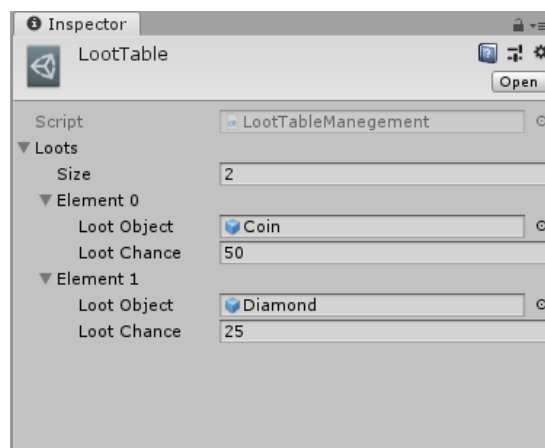


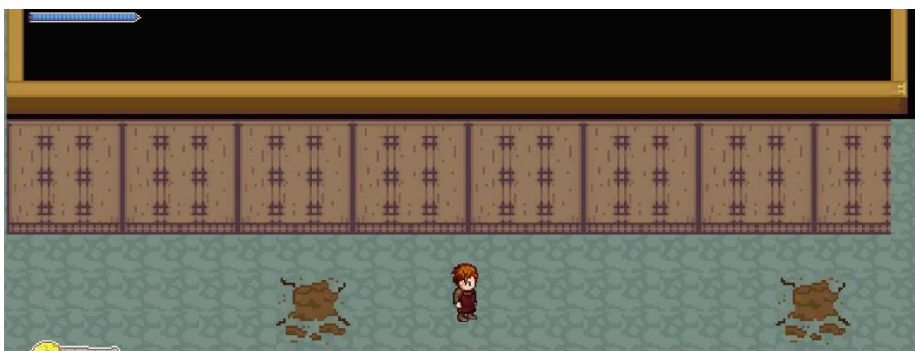
Imagen 4.e.2: Propiedades del ScriptableObject LootTable

## 5. Fallos e implementaciones futuras

En la versión final del juego ha habido dos errores o *bugs* que he encontrado haciendo pruebas que no he podido solucionar por falta de tiempo, pero lo dejo aquí indicado y, en caso de conocer su solución, dejarla indicada.

El primer fallo es que los objetos que el jugador lanza con su habilidad especial, cuando atraviesan el collider de las zonas de teletransporte, las toma como el *GameObject* del jugador y hacen que el jugador se teletransporte sin haber chocado con los elementos de teletransporte. He estado revisando y los elementos que el jugador lanza no tienen la posibilidad de ser identificados como objetos *Player*, pero deduzco que al ser hijos del *GameObject* del jugador principal, estos son reconocidos por los collider de este tipo.

El segundo fallo tiene que ver con las texturas del interior del escenario de la casa donde se encuentran los cofres de selección de personajes. Como se puede ver en la *imagen 5.1*, cuando el jugador abre uno de los cofres de selección del personaje, las texturas con objetos de decoración desaparecen del mapa y se buguean. Este comportamiento carece de sentido ya que, tal y como revisé en el código, no existe ninguna relación entre el *GameObject* de los cofres y los *GameObject* de los escenarios ni realiza ninguna acción en los scripts de mantenimiento de los cofres *ChestBehaviour.cs* que pueda afectar a las texturas del Tilemap, por lo que entiendo que se trata de un bug y no de un error de lógica.



*Imagen 5.1: Fallo de las texturas del interior del escenario de la casa*



## 6. Bibliografía

- Guia Unity 2D. Diseño Juego Tipo Zelda, *Youtube*:  
<https://www.youtube.com/watch?v=s81hQDATvBE&list=PLiplyDjUMtthNIEmdbP7E6-ZaWTZKyAQ5>
- Guia Unity 2D RPG and Tilemaps, *Youtube*:  
<https://www.youtube.com/watch?v=F5sMq8PrWuM&list=PL4vbr3u7UKWp0iM1WIfRjCDTI03u43Zfu>
- *OpenGameArt*, web oficial: <https://opengameart.org/>
- Colección de sprites LPC, *OpenGameArt*: <https://opengameart.org/content/lpc-collection>
- Herramienta de creación de sprites, *Universal-LPC-SpriteSheet*:  
<http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/>
- Sprites Adicionales 2D, *OpenGameArt*: <http://lpc.opengameart.org/static/lpc-style-guide/assets.html>
- “The Lord of the Rings: The Two Towers CR – 08. The Wolves of Isengard”, *Youtube*:  
<https://www.youtube.com/watch?v=Vk3HTqZjWAs&t=115s>
- “Star Wars – The Force Suite (Theme)”, *Youtube*:  
<https://www.youtube.com/watch?v=eb2zuegwcwk>
- “John Williams – Duel of Fates (Star Wars Soundtrack) [HQ]” *YouTube*:  
<https://www.youtube.com/watch?v=xLYCxbBZUCY>
- “Two Towers Soundtrack-05-The Uruk-Hai”, *Youtube*:  
[https://www.youtube.com/watch?v=9Vetg\\_e5fJQ](https://www.youtube.com/watch?v=9Vetg_e5fJQ)
- “Avengers Suite (Theme)”, *Youtube*:  
<https://www.youtube.com/watch?v=FOabQZHT4qY>
- “Avengers: Infinity War Trailer Music #2”, *Youtube*:  
[https://www.youtube.com/watch?v=K48wOUe2\\_Ns](https://www.youtube.com/watch?v=K48wOUe2_Ns)
- Electrical Disintegration Animation, *OpenGameArt*:  
<https://opengameart.org/content/electrical-disintegration-animation>
- Lightning Shock Spell, *OpenGameArt*: <https://opengameart.org/content/lightning-shock-spell>
- 2D effects list, *OpenGameArt*: <https://opengameart.org/content/2d-effects>
- *Time Fantasy* , (Free Graphics RPG Game Development):  
<http://finalbossblues.com/timefantasy/free-graphics/>
- Jungle Tileset, *Itch.io*: <https://finalbossblues.itch.io/tf-jungle-tileset>
- LPC terrains, *OpenGameArt*: <https://opengameart.org/content/lpc-terrains>
- LPC 16x16 terrains, *OpenGameArt*: <https://opengameart.org/content/lpc-16x16-tiles-extended>
- “How to use Sprite Editor correctly”, *Unity Docs*:  
<https://docs.unity3d.com/Manual/SpriteEditor.html>
- Wind animation, *Itch.io*: <https://kvsr.itch.io/wind>
- Ashlands Tileset, *Itch.io*: <https://finalbossblues.itch.io/ashlands-tileset>