MEMORIA DE PROYECTO C#

Programación Avanzada.

Samuel González Linde

Máster Universitario en Computación Gráfica, Realidad Virtual y Simulación

Diciembre 2021

Contents

1 Introducción

El proyecto consiste en hacer uso de la herramienta Unity junto con el lenguaje de programación C# para crear una aplicación gráfica interactiva. La idea sobre la que se basa la solución del proyecto es un juego de disparos, con características de juego de supervivencia y una vista cenital.

A continuación se enumeran las diferentes requisitos que se han planteado para llevar a cabo la solución:

1. Cámara con una vista cenital que siga al jugador.
2. Movimiento del personaje haciendo uso de las teclas ASDW.
3. Capacidad del personaje para apuntar en la dirección del cursor.
4. Disparo del personaje en la dirección del cursor al pulsar click izquierdo.
5. Diferentes comportamientos de los enemigos:
   (a) Permanecer estático si el jugador no se encuentra al alcance del enemigo.
   (b) Perseguir al jugador al entrar en el rango de visión.
   (c) Atacar al jugador al acercarse lo suficiente a él.
6. Interfaz de usuario para mostrar la información del juego:
   (a) Vidas restantes del jugador.
   (b) Balas restantes del jugador.
   (c) Puntuación conseguida durante la partida.

2 Solución propuesta

Para comenzar a explicar como se ha implementado cada uno de los comportamientos anteriormente descritos, es necesario explicar primero el sistema de comunicación que se ha empleado para algunos objetos de la escena.

2.1 Sistema de Eventos

Este sistema se ha implementado para poder establecer una comunicación de forma sencilla y desacoplada con la interfaz de usuario del juego, aunque finalmente algunos otros objetos de la escena han acabado usandolo debido a su utilidad.

Este sistema de eventos se basa en el patrón Publicador-Subscriptor el cual haciendo uso de un canal de eventos, permite que unos Subscriptores permanezcan a la escucha hasta que algún Publicador mande un aviso por este canal de

eventos. Está compuesto por dos Scripts, un MonoBehaviour que hace de Subscriptor llamado **GameEventListener** y un ScriptableObject que nos permitirá crear los canales de eventos (**GameEvent**). Los publicadores serán aquellos otros MonoBehaviour de la escena que hagan uso de uno de estos canales de eventos para lanzar los avisos.

Hay que tener en cuenta que estos canales de eventos permiten avisar a los subscriptores y pasarles estructuras de datos simples (int, boolean, string) desde el editor, pero no desde el propio código de forma dinámica. Por lo tanto, podemos decir que estos eventos son de tipo void. Para poder pasar de forma dinámica hay que crear otros canales de eventos que hacen uso de UnityEvents de forma dinámica. En este caso se han creado **IntGameEvent** y **IntGameEventListener** para poder enviar los puntos que son sumados por cada enemigo matado.

2.2 Cámara

2.2.1 Seguimiento de la Cámara

Para conseguir que la cámara siga al jugador, el script **CameraFollow** hace uso de la Instancia del propio jugador y la función Lerp, que nos permitirá desplazar la posición de la cámara hasta la posición del jugador (junto con un offset para mantener la cámara alejada) durante un tiempo que indiquemos.

2.2.2 Temblor de Cámara

Esta característica se ha logrado haciendo uso del script CameraShake, que a su vez hace uso de la librería DOTween para hacer vibrar a la cámara cada vez que el jugador recibe un golpe.

Para avisar de que se ha recibido ese golpe, la cámara está a la escucha del canal de eventos *DamagedEvent* que es lanzado por el script **Player** cada vez que el jugador recibe daño.

2.3 Jugador

El jugador hace uso del patrón Singleton, definido en la clase **Singleton** para poder utilizar su instancia desde cualquier clase y poder asi acceder (principalmente) a su posicion. Esta clase es heredada desde el script **Player** para este propósito y también implementa la interfaz **IDamageable** que nos permite definir las funciones para recibir daño si nos atacan.

El resto de scripts que utiliza el jugador son **PlayerShooting**, **PlayerMovement** y **Health**. Este ultimo se utiliza para llevar un control de la vida máxima que puede tener el jugador y la vida actual que tiene.

2.3.1 Movimiento

Para el movimiento se ha hecho uso del nuevo Sistema de Input y los UnityEvents que van enviando la información captada por las teclas pulsadas a los métodos que nosotros tengamos definidos. En este caso, el script **PlayerMovement** se encarga de recibir ese input, leerlo y actualizar su posición en el juego.

2.3.2 Apuntado y disparo

El script **PlayerShooting** se encarga de obtener los inputs del movimiento del ratón y del click izquierdo. Teniendo la información de la posición del ratón, se realiza una proyección de la pantalla a 3D a través de un raycast que impacta contra el suelo y nos indica la posición a la que tiene que mirar el jugador.

Al hacer click, una referencia de la clase **Weapon** del arma que tenemos es encargada de realizar la lógica del disparo.

2.4 Armas

La clase **Weapon** es la encargada de tener un ScriptableObject con toda la información del arma que se está usando, así como referencias para publicar los eventos de disparo y recarga o el punto de salida de las balas del arma.

El ScriptableObject que utiliza es un **ScriptableWeapon** que tiene el daño, las balas del arma, los tiempos de recarga y entre disparo; y los sonidos.

Todo ello es utilizado por **Weapon** para realizar el disparo. A través de un raycast, se instancia un rayo para observar de forma visual el viaje de la bala y en caso de golpear a un enemigo que implemente la interfaz **IDamageable**, le hará daño.

También se reducirá el numero de balas en cada tiro así como recargar cuando sea necesario.

Figure 1: Prefab utilizado para el Revolver

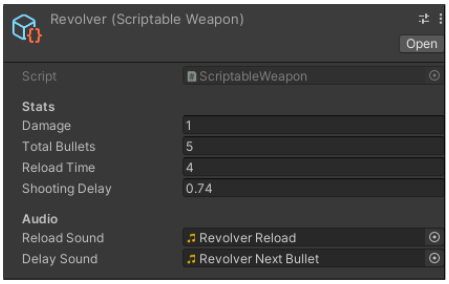

Figure 2: Scriptable Object con la información del Revolver

2.5 Enemigos

Para los enemigos se ha optado por crear una máquina de estados que definiesen cada uno de los comportamientos que van a tener.

Para ello se ha creado una máquina de estado genérica haciendo uso de la interfaz **IEstate** que define los métodos EnterState y Update que van a ser llamados al entrar en el estado y a cada frame a través de un MonoBehaviour. Este MonoBehaviour es el script **StateMachine** que es el encargado de transicionar entre los distintos estados.

Tiene una referencia al estado actual en el que nos encontramos y el resto de estados posibles. Para facilitar su uso se hace almacenan en un diccionario y como clave se hace uso de una clase **Enum**.

La clase **Enemy** tiene la información de los enemigos que es proporcionada a través de un **ScriptableEnemy** y hereda de la clase **StateMachine**. Haciendo uso de la interfaz **IEstate** se han creado para los enemigos los scripts para los estados **EnemyIdle**, **EnemyChasing**, **EnemyAttack** y **EnemyDeath**.

La máquina de estados, hace de contexto para los estados y va a proporcionarle toda la información que necesitan para que puedan realizar sus acciones.

Al igual que en el caso del jugador, implementan la interfaz **IDamageable** para poder recibir daño de los disparos del jugador. A su vez, implementan la interfaz **ISpawneable** que nos permite realizar acciones cada vez que la instancia del **PoolManager** hace Spawn o Despawn de estos objetos.

En este caso, al sapawnear, los enemigos escalan desde cero a su tamaño normal haciendo uso de DOScale del paquete **DOTween**.

2.5.1 Zombies

Figure 3: Prefab utilizado para los Zombies

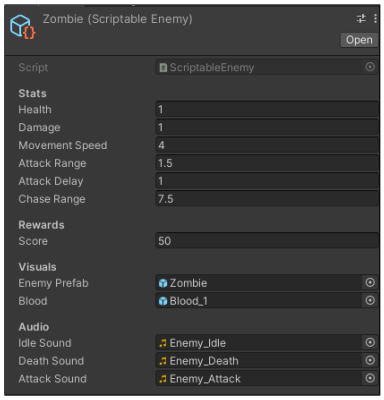

Figure 4: Scriptable Object con la información de los Zombies

Al igual que en el caso de las armas, se hace uso de ScriptableObjects para

tener la información de cada uno de los enemigos que se van a poder crear y asi establecer los parametros que van a usar cuando sean Spawneados.

Estos son el único tipo de enemigo que ha dado tiempo a crear durante el desarrollo.

2.6 Interfaz de Usuario

En la interfaz de usuario encontramos la puntuación que se ha conseguido, las vidas restantes del jugador y las balas que tiene.

Figure 5: Interfaz de usuario del juego

Los tres elementos que se han mencionado funcionan de manera muy similar: permanecen a la escucha de ciertos eventos y se actualizan. A continuación se va a mostrar como se ha llevado a cabo el control dinamico de la información en la interfaz.

2.6.1 Vidas

Al comenzar el juego, **HealtUI** instanciar tantos prefabs de *Heart* como vida máxima tenga el jugador y cada vez que este reciba daño, a través de los avisos del canal *DamagedEvent* se decrementará el número de vidas.

2.6.2 Munición

Al igual que en el caso anterior, el escript **AmmoUI** se encarga de instanciar tantos prefabs de *Bullet* como balas máximas tengamos en el ScriptableObject del arma utilizada. Cada vez que el arma hace un disparo, al estar a la escucha del canal ShootEvent, este script actualizará la interfaz y eliminará una de las balas. Cuando no queden balas el arma comenzará a recargar, avisando a través del canal ReloadEvent haciendo que el script **AmmoUI** vaya añadiendo una bala cada X tiempo. Este tiempo es calculado dividiendo el tiempo total de recarga que necesita el arma entre las balas totales que tiene.

2.6.3 Puntuación

La puntuación hace uso de un script **ScoreUI** que se encarga de ir sumando la nueva puntuación conseguida y mostrarla en pantalla haciendo uso de TextMesh-Pro. También contiene un componente que permanece a la escucha de *EnemyKilledEvent*, encargado de enviarle la puntuación de cada enemigo que va mueriendo durante la partida y **ScoreUI** se encarga de actualizarlo .