RED DEAD REDEMPTION 3 – INFORME

Enrique Fernández-Baíllo Rodríguez de Tembleque

Jacobo Banús Bertram

Jugador

El desarrollo del jugador principal comenzó dibujando manualmente a estilo pixel art el personaje, en diferentes formas para posteriormente ser animado. En la carpeta RedDeadRedemption3/character se encuentran los archivos de Photoshop .psd y los archivos .png finalmente exportados del personaje en cada una de sus formas.

El personaje se divide en los siguientes scripts:

- **Walk:** que gestiona la dirección en la que anda el personaje y la orientación del Sprite en función de hacia qué dirección ande.
- **Shoot:** que instancia balas mágicas o normales en función del botón del ratón presionado.
- **BulletManager:** que se encarga de que las balas avancen uniformemente en la dirección correcta, y se destruya cada una a su respectivo tiempo.
- **BulletCollisionManager:** que, en ambos tipos de bala, destruye la bala si esta choca con una pared o un monstruo. En caso de la bala mágica, si choca con una pared (tag “Ground” o “Wall”), teletransporta al personaje hacia esa pared, al estilo del juego indie Out There Somewhere.
- **Jump:** El script más complejo del personaje, el cual gestiona el salto y el dash. Funciona mediante una máquina de estados, los cuales son: en el suelo, saltando, dasheando por el aire y dasheando por el suelo.
  Si está dasheando, no permite realizar ninguna otra acción en el update. Durante el dash, el cual se gestiona con una corrutina, se anula la gravedad y se avanza uniformemente hacia el lado. También se cambia momentáneamente la layer de colisión del personaje para poder así permitir dashear los proyectiles de los enemigos y pasar a través de las paredes finas. Esto se ha configurado en la Layer Collision Matrix.
  Si está en el suelo, puede saltar y se activa la posibilidad de hacer un doble salto un poco menos potente. Además, si está en el suelo, pero deja de estarlo sin saltar, solo permite ejecutar este segundo salto. El salto se lleva a cabo aplicando una fuerza al Rigidbody2D en la coordenada Y. El cambio de estados entre “grounded” y “jumping” se lleva a cabo mediante colisiones con el suelo.
- **Gun Start:** Inicializa correctamente la posición del Sprite de la pistola.
- **Gravity Fix:** En caso de que una caída libre haga al personaje ir demasiado rápido, fija la velocidad máxima a la que puede caer el personaje. De esta manera se evita que el personaje atraviese tan rápido una colisión que el motor no sea capaz de calcularlo, atravesando así el objeto sin colisionar.
- **Collision2D:** El jugador tiene dos colliders bidimensionales: uno para interactuar con el escenario y otro para la interacción con enemigos. Gestiona las colisiones con los enemigos y sus proyectiles. Esta decisión se hizo para balancear el juego,

ya que en caso contrario el jugador era impactado por muchos más proyectiles de los que visualmente resultan naturales.

- **Collision Manager:** Gestiona las colisiones del personaje con el entorno. En caso de que el jefe golpee al personaje, empujará a este hacia atrás haciendo uso del Rigidbody2D. En caso de que un proyectil o un enemigo toque al jugador, este llamará al script que se encarga de gestionar la vida para restarle vida restante. Cuando el personaje recibe daño, se cambia mediante una corrutina su material, para que cambie de color durante un periodo de 0.35 segundos, dando la sensación visual de estar recibiendo daño.
- **Health Manager:** El encargado de asignar al personaje una vida inicial y de restarle vida cuando recibe la orden. Se ha asignado un cooldown a la pérdida de vida para evitar interacciones no deseadas entre las colisiones del jefe final y el jugador. Además, es tradicional en este género que cuando pierdas vida seas invulnerable durante un breve instante de tiempo. Este script se encarga también de llamar la función que cambia el display gráfico de la vida en la pantalla, llamando al método UpdateHealth de la clase HealthDisplay, la cual sigue un patrón Singleton para facilitar su acceso al ser un elemento único y evitar así la búsqueda innecesaria de componentes.

Por otro lado, el personaje se ha animado haciendo manualmente los clips de animación, alternando entre los fotogramas dibujados. Las transiciones entre clips de animación se hicieron en Animator Controller y dependen de booleanos que cambian en función del estado del personaje. Por ejemplo, la animación de dash se llama durante cualquiera de los dos estados de dash, y se vuelve a la postura por defecto al terminar. Se ha utilizado también un Trail Renderer, el cual se activa al principio del dash y se desactiva al final. Se utiliza para potenciar la sensación de desplazamiento, y se ha configurado ligeramente el color y la forma.

Enemigo

En paralelo al desarrollo del movimiento del personaje, empezamos a trabajar en el enemigo básico del juego. Para ello empezamos importando unas fotos de sprites de internet que nos gustaron, y creamos las animaciones: vuelo, ataque, daño y muerte. La lógica del enemigo la separamos de las animaciones, que se gestionan en un script aparte. La lógica del enemigo se basa en una máquina de estado. De hecho, el enemigo tiene varios conjuntos de estados que se superponen, como por ejemplo los estados que gestionan la disponibilidad del ataque y los que gestionan la vida del enemigo. Inicialmente, el enemigo únicamente se movía de izquierda a derecha y viceversa de manera senoidal. Añadimos proyectiles en forma de bolas de fuego que el enemigo disparaba. Para ello descargamos fotos de sprites para la animación de recorrido y explosión de la bola de fuego, y de nuevo separamos lógica y gestión de la animación. En un principio, el enemigo instanciaba directamente las bolas de fuego, pero luego implementamos factorías para los distintos proyectiles, de manera que la lógica de creación se delegó en ellas. Un reto al que nos enfrentamos fue gestionar la orientación del enemigo, para que éste estuviera apuntado al jugador si este estaba en su rango de ataque y lanzara las bolas de fuego en su dirección. Para ello, definimos una orientación que toma el valor de 1 o –1, y reescalamos el transformer del enemigo con ella. Además, añadimos un estado de seguimiento al jugador, de manera que, si el jugador se acerca

mucho al enemigo, éste comienza a seguirle hasta que el jugador se aleje lo suficiente. Cabe destacar que tuvimos que crear dos colliders diferentes para el enemigo: uno que controla el rango de disparo y otro que controla la colisión con el propio enemigo. Cuando el collider de rango registra un entrada del jugador, el enemigo dispara en esa dirección. El collider de cuerpo, por otra parte, se usa para gestionar la colisión del enemigo con las balas del jugador, que le quitan vida. Las bolas de fuego, por su parte, vuelan hasta colisionar o llegar a un límite de recorrido, y luego explotan. Con su colisión con el jugador tuvimos dificultades, ya que decidimos que la mecánica de dash las esquivaría. Terminamos por utilizar las layers de colisión 2D para solucionar este problema. Finalmente, también añadimos un movimiento.

Slime y jefe final

Decidimos añadir un jefe final cuando ya teníamos prácticamente implementado al enemigo. Encontramos en internet unas fotos de sprites que nos gustaron mucho (y nos costaron 10€ que esperamos que ICAI nos reembolse). Estos incluían un slime que transicionaba a un jefe final. Importamos las fotos y creamos las diferentes animaciones. La primera decisión fue decidir si el slime y el boss debían ser un mismo objeto con dos estados principales, ya que el jefe final no dejaba de ser lo que salía después del slime. No obstante, descartamos rápidamente esta idea, debido a que creaba una complejidad muy artificial en el código, por lo que separamos slime y jefe en objetos diferentes.

El slime únicamente se mueve de un lado a otro con una pequeña animación de salto, y recibe daño por parte del jugador. Su animación de muerte es precisamente la transición hacia el jefe final. Justo antes de morir (destruirse el objeto), instancia al prefab del jefe final. Este tiene varios scripts asociados. En primer lugar, cabe recalcar que el collider utilizado es el poligonal, con un pequeño script que lo adapta a cada sprite de las animciones del jefe. Esto fue importante ya que, en los distintos ataques del jefe, la forma del sprite cambia sustancialmente. Asimismo, dado que el collider va junto con la animación, la lógica del monstruo está separada del collider, por lo que hicimos un script específico que gestionara las colisiones. En cuanto a la lógica del jefe, ésta también se rige por una máquina de estados que definen el estado del movimiento del jefe. Dado que el collider queda definido por las animaciones, la lógica del jefe consiste precisamente en cuando ejecutar cada animación. En particular, el jefe tiene un estado básico en el que persigue al jugador, tres ataques cuerpo a cuerpo, y un ataque a distancia. El ataque a distancia es el preferido por el jefe, y lo ejecutará siempre que el tiempo de cooldown lo permita y el jugador esté suficientemente lejos. El ataque consiste en la creación de cuatro pequeños proyectiles de fuego con posición vertical aleatoria. Igual que con el enemigo, en un principio el jefe instanciaba los proyectiles personalmente, pero decidimos delegar su creación a una factoría específica. Por otro lado, cuando el jefe está suficientemente cerca del jugador, realiza aleatoriamente uno de sus tres ataques cuerpo a cuerpo: salto con impacto contra el suelo, aliento de fuego, o tajo con su espada. Si el jugador colisiona con el monstruo o con alguno de sus ataques, recibe daño y un empuje en dirección opuesta al jefe (con los proyectiles unicamente recibe daño, y además pueden ser esquivados mediante el dash).

Escenario principal

Para diseñar el escenario princial, descargamos un conjunto de assets (carpeta Cainos) de la tienda de Unity. En primer lugar, utilizamos el tilemap que venía con ellos, que nos permitió diseñar la distribución de las plataformas del nivel. Aquí, decidimos añadir unas paredes especiales que el jugador podría atravesar con el dash, para lo cual aprovechamos el sistema de layers de colisión 2D de Unity. El diseño del nivel fue orientado a aprovechar todas las mecánicas del jugador, además de crear un buen escenario para la batalla contra el jefe final. Los enemigos básicos están esparcidos por el nivel. Además, utilizamos varios prefabs visuales que venían con los assets descargados para adornar el entorno, como hierba, piedras o lámparas. Por último, incluimos en el escenario varios cuadros de diálogo para que funcionaran como un pequeño tutorial que explicara el uso de las mecánicas al jugador.

Transición entre escenas mediante eventos

El juego consta de 4 escenas diferentes (aparte de las utilizadas para testeo): menú principal, pantalla de muerte, pantalla de victoria y escenario principal. El menú principal, la pantalla de muerte y la pantalla de victoria son muy parecidas: Un canvas con botones para jugar o salir del juego. El botón de jugar carga el escenario principal en todos los casos. Para gestionar la transición del escenario principal a otras escenas, hicimos uso de eventos, y gestionamos las transiciones mediante el objeto SceneTransition. En primer lugar, el jugador invoca un evento cuando muere, al que SceneTransition está suscrito con una función que carga la pantalla de muerte. Además, debajo del nivel hay un gran collider (VerticalLimit) que controla la caída del jugador. Cuando se activa su trigger, el collider lanza un evento al que SceneTransition está suscrito, de nuevo con la función que carga la pantalla de muerte. Por último, la muerte del jefe final invoca un evento, al cual SceneTransition está suscrito para cargar la pantalla de victoria. Aquí nos enfrentamos a varias dificultades: Dado que el jefe no está de entrada en la escena, sino que es instanciado tras la muerte del slime, SceneTransition no se puede suscribir directamente al evento de su muerte. Nuestra primera solución fue hacer que el evento fuera estático, de manera que SceneTranstion lo puedirera referenciar directmente en Start(). No obstante, esto causaba problemas al rejugar el juego: Dado que el evento era estático, este no se reiniciaba con los demás objetos de la escena principal, y daba problemas al volver a invocarse tras una segunda vitoria contra el jefe final. Finalmente, decidimos que el evento no fuera estático, e ideamos la siguiente solución: Aprovechando el evento de muerte del slime que utilizamos para la cámara (explicado más adelante), SceneTransition se suscribe a él, y al invocarlo busca al jefe final en la escena, que ahora si está presente. Así se puede suscribirse al evento de muerte de esa instancia de jefe final en particular, y cargar la pantalla de victoria cuando éste invoca el evento. De esta forma, no teníamos problemas para rejugar el juego las veces que quisiéramos.

Cámara

El código de la cámara hace que siga al jugador, dejando mayor espacio en la dirección al que éste apunte. Además, la cámara está suscrita al evento de muerte del slime, ya que

en ese momento se acerca al jugador para hacer la batalla contra el jefe final más cinemática.

Audio

Tanto la música del menú principal como la música que suena durante el nivel son originales del juego, compuestas mi amiga Minas (amiga de Enrique). No fueron expresamente compuestas para el videojuego, ya que las tenía perdidas en el disco duro, pero desde el equipo de desarrollo se ha decidido que eran adecuadas para sus respectivas escenas. La música del jefe final es Vordt Of The Boreal Valley, del Dark Souls III, y la canción de la pantalla de victoria es Outlaws From The West, del Red Dead Redemption II. El equipo de desarrollo ha adoptado una postura de cero clemencia para con los derechos de autor de estas canciones. El sonido de ambiente es únicamente un sonido de viento, que encaja con el paisaje montañoso y solitario que presenta el juego.

A cada escena se le ha asignado un audio base: en el menú principal, la canción del menú; en la pantalla de muerte y el nivel, el viento; en el nivel, también la canción de nivel; y Outlaws From The West para la pantalla de victoria. Para las transiciones de audio, como por ejemplo la transición de canción de nivel a silencio antes del boss, y de silencio a la canción del boss final, se han hecho funciones de FadeIn y FadeOut de sonido, para introducir el audio de forma menos abrupta. Ambas funciones son corrutinas que interpolan linealmente el volumen desde un volumen de inicio hasta un volumen final. Estas funciones deberían haber sido implementadas mediante una interfaz, pero se subestimó el tiempo que se tardaría en hacer este proyecto. Algunas transiciones de audio fueron lanzadas en conjunto con los estados del programa (por ejemplo, la canción del boss empieza cuando muere el slime). Para otras, se usaron colliders de tipo trigger (por ejemplo, la transición de canción de nivel a silencio se hace cuando el jugador llega a cierta plataforma).

UI

La interfaz gráfica de la escena del juego consta de un canvas que contiene un corazón con el número de vidas indicado encima. Este número se modifica mediante el script HealthDisplayer, el cual ya se ha explicado que se trata de un Singleton que invoca el Health Manager. De esta forma, se consigue separar la lógica detrás de la vida de su interfaz gráfica, lo cual es recomendable para la escalabilidad del programa. Para el texto del número y para el resto de los textos interactivos y la mayoría de los textos del juego, se han utilizado componentes de tipo TextMeshPro, para mejor calidad del texto. Estos componentes han utilizado una fuente estilo pixelada para ir en consonancia estética con el estilo del juego. Los elementos de las interfaces gráficas se han anclado mediante anchor points a sus respectivas esquinas más cercanas para hacer que la interfaz gráfica se mantenga adecuadamente situada aún en diferentes resoluciones y equipos.

Tanto la pantalla de muerte como de victoria como de menú constan de una imagen de fondo, una hecha a mano y otra tomada de la pantalla de muerte del Red Dead Redemption 2 y editada posteriormente. Constan también de dos botones, que permiten reiniciar (o iniciar en caso del menú) el juego o salir de la aplicación, los cuales son invisibles y son componentes de tipo Button. Estos botones tienen su respectivo texto

encima. Los botones tienen un componente de tipo Event Trigger mediante los que se ha conseguido aumentar el tamaño del texto al posicionar el ratón encima del cuadrado del botón que tienen debajo, dándole a los menús un toque visual dinámico.

Secreto

A lo largo de este informe y en el diagrama UML, se ha evitado intencionalmente mencionar los scripts y los elementos pertenecientes al Easter Egg que contiene este juego, por la propia naturaleza secreta del mismo. Este pequeño detalle, al cual solo se puede acceder (sin hacer trampa) mediante un bug del juego, el cual no se ha resuelto a propósito, ha llevado un rato de trabajo y ha quedado demasiado bien, por lo que desde el equipo de desarrollo se espera encarecidamente que se busque. (Si se llegase a encontrar, no moverse durante más de 10 segundos desde que empiece el cambio de canción).

UML

Como podemos observar, y tras explicar el desarrollo del juego, el UML final es muy distinto al inicial (y bastante más amplio). El slime y jefe final, por ejemplo, son adiciones completamente nuevas, ya que decidimos añadirlas durante el desarrollo del juego. Por otro lado, el jugador tiene muchos más componentes, ya que su movimiento terminó siendo mucho más complejo de lo ideado inicialmente. Además, la cámara esta ahora presente en el diagrama, al tener un papel activo en el seguimiento al jugador.

También son adiciones las distintas factorías, patrón de diseño que nos ayudó en la lógica de creación de proyectiles. El enemigo también tiene más componentes, al separar la lógica de la animación, además de contener varios colliders.

En conclusión, dada la creciente ambición del proyecto durante su desarrollo, el UML terminó creciendo en gran medida.

No se incluye foto del diagrama ya que es demasiado grande como para que se pueda leer el texto, pero el archivo .drawio se encuentra en este mismo directorio del repositorio.