Documentación SG-TrainRunner

Práctica realizada por:
- Pedro Pernías Martínez.
- Francisco Antonio Pino Muñoz.

1. Descripción.

Nosotros hemos desarrollado un juego con la esencia de un subway surfers. Consta de un escenario compuesto por 3 carriles, en los que el personaje que controlamos puede cambiarse entre ellos y además tendrá que evitar los distintos obstáculos que se irán generando de forma aleatoria. Cuando éste personaje colisione con uno de los obstáculos finaliza la partida y te muestra la puntuación obtenida.

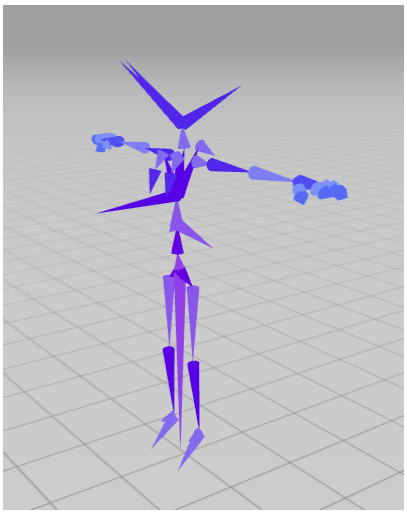
2. Diseño.

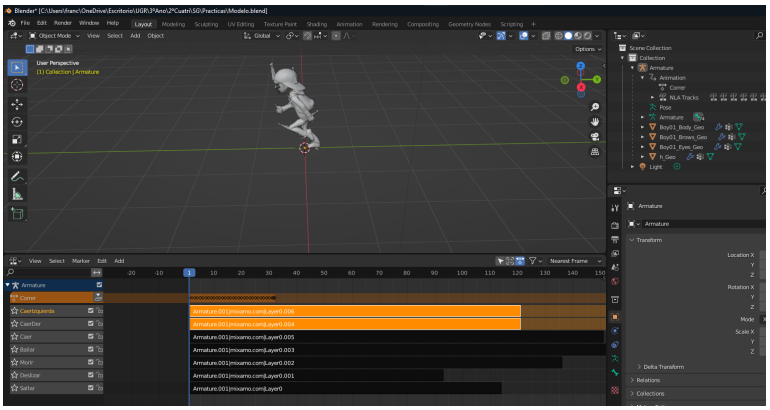
Esta sección podemos dividirla en cuatro subsecciones principales:

2.1. Modelos:

Los modelos principales para el desarrollo de la práctica son los siguientes:

- **Humano**: en un principio, empezamos a desarrollar nuestro propio modelo de un humano y a trabajar con él, más tarde observamos que programar unas animaciones realistas para el juego (correr, saltar, colisionar...) era una tarea demasiado compleja y extensa, por lo que optamos por descargar un modelo de internet con un **esqueleto**. Tras añadir la **textura/forma/skin** que iba a tener, le buscamos unas animaciones adecuadas y juntamos con la aplicación **Blender**, para finalmente añadir a nuestro programa como un modelo GLTF.
  (cont).





- **Tren**: un CSG compuesto por una base rectangular con un tejado cilíndrico con la mitad sustraída.
  La base tiene una textura del juego original, al que hemos creado un bump editando manualmente los colores para aplicar volumen a la textura.

- **Valla**: es una Object3D en el que le añadimos las 3 partes de la valla y le aplicamos las texturas a cada una de ellas.

- **Muro**: es un cubo con textura de ladrillo y bump para el volumen.

- **Carril**: un cubo con la textura de una vía de tren con el bump creado manualmente.

- **Fondo**: un cubo delgado con una imagen de fondo, a la que le modificamos el color para que quedase del mismo tono que el césped de los laterales para dar una mejor sensación.
-

Salvo e**l bump del Muro,** los bumps de los demás modelos han sido creados a mano por medio de programas de edición de fotos e imágenes.

Por añadir, al comenzar con el juego decidimos incluir la implementación del **modelo en forma diferida.**

Animaciones

El objetivo de este diseño de implementación era poder establecer una animación inicial, en este caso de la Bailar una vez implementado el modelo del humano.

Durante el desarrollo del juego iniciaremos una animación en bucle de correr, en el caso de realizar un salto hacemos un fade **InOut** entre ambas animaciones para poder realizar el salto y luego seguir corriendo.

Cuando el Juego finaliza aplicaremos unas animaciones de choque atendiendo al caso en el que ocurre el choque(Izquierda, derecha y de frente)

2.2. Movimiento y colisiones:

- **Movimiento:** Todo el movimiento ha sido creado y definido en la clase Escena. El **movimiento básico** estaba basado en el cambio de carriles, es decir, mover a Izquierda y derecha conforme se pulsan las teclas de las flechas o por medio de un movimiento con el ratón.

  Para realizar este movimiento era necesario trasladar el conjunto de: el **humano** que habíamos creado previamente y los **3 RayCasters** y poder trasladar el sistema de colisiones de forma correcta. Este movimiento se realizaba sobre el eje X con la función setPositionHumano que realizaba una animación de TWEEN para trasladar el conjunto.

  Para realizar el movimiento de **Salto** recurrimos inicialmente a la implementación del mismo mediante un movimiento de TWEEN con la opción de **Yoyo** activada para que el movimiento fuese más fluido.

  Sin embargo, una vez teníamos implementados los movimientos básicos era necesario implementar movimientos más complejos que cortaban la animación de los más simples, por lo tanto, tuvimos que modificar el salto para establecer dos movimientos en vez de uno con la opción de **YOYO.**

Y saber en qué momento del salto estábamos subiendo y descendiendo. Los movimientos son:

- Saltar (Ascenso) y Derecha o Izquierda
- Saltar (Descenso) y Derecha o izquierda
- Derecha o Izquierda y Saltar

Para ello era necesario crear una función que cortase los movimientos del salto y definir los sucesos en base la posición del humano y Booleanos de control de terminación del movimiento.

- **Colisiones**: En la función update de la escena, llamamos a una función que nos detecta las colisiones con los obstáculos. Las **colisiones**, las detectamos mediante **3 RayCasters**, como nuestro personaje siempre mira hacia el **eje Z negativo,** uno de los rayos apunta hacia el Eje Z(Colisiones frontales), Eje X(colisiones por la izquierda) y Eje -X(colisiones por la derecha).

  En un principio, las colisiones se detectaban mediante los atributos **near y far** de los distintos rayos para observar las colisiones con los obstáculos que se encontraran en una posición lo suficientemente cerca como para detectar que ha colisionado. Este método solamente era efectivo con velocidades pequeñas de generación de terreno, ya que con velocidades más altas, este rayo al ser tan pequeño y actualizarse el terreno tan rápidamente, no detectaba correctamente las colisiones. **Para arreglar esto**, optamos por cambiar el sistema de detección de obstáculos, utilizando un **rayo** que proyecte hacia el **infinito**, y simplemente comprobamos el atributo **distance** con el que se encuentra del obstáculo, y en caso de que este sea menor que una cifra, se detecta como colisión.

  Las colisiones han supuesto mucho trabajo, ya que al principio, detectaba siempre una colisión aunque no existiera un obstáculo en ese carril con el atributo visible = true. Tras muchas pruebas y comprobaciones, llegamos a la conclusión de que el rayo detecta las colisiones con los hijos de los modelos principales. Es decir, si el tren está compuesto por dos subobjetos (cubo y cilindro), el rayo detectaba la colisión con el cubo, cuya visibilidad es true y no con el tren, que la visibilidad es false. Para solucionar esto, tuvimos que agregar al array de obstáculos en el que

comprobaba la colisión, que comprobase la colisión con el parent en lugar del objeto en sí.

El correcto funcionamiento de las colisiones con el movimiento del personaje,
se debe a que estos rayos se desplazan junto con el personaje cuando se interactúa con él.

2.3. Escenario:

El escenario (clase MyScene), está formado por un background de un cubeMap para ambientar el terreno, el modelo del humano utilizado, el fondo donde ponemos la imagen que se ve en la partida y finalmente el mapa, que contiene el resto de elementos del escenario.

El mapa es la clase que contiene la funcionalidad de la generación del terreno, la generación de obstáculos y la puntuación de la partida. El mapa está compuesto por 3 terrenos (terreno 1, terreno 2 y terreno 3), los cuáles están formados por 3 carriles (carril0, carril1, carril2), que cada uno de ellos puede tener un obstáculo dependiendo del patrón aleatorio de obstáculos que se genere. La función update de esta clase, se encarga de desplazar los tres terrenos en el sentido positivo de la Z, de forma que genere la ilusión de que el personaje se está moviendo y cuando este llegue a un límite, establecido por la zona de regeneración, el terreno llamará a su función de generar obstáculos y se desplazará justo detrás del fondo, para que no se vea directamente como se va generando el terreno.

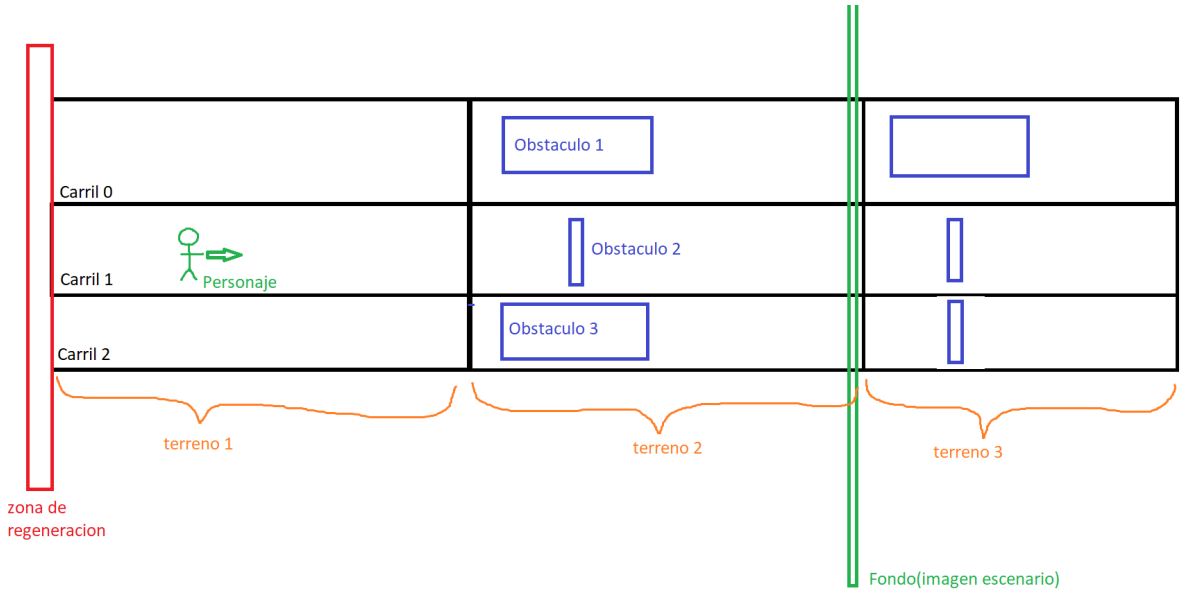


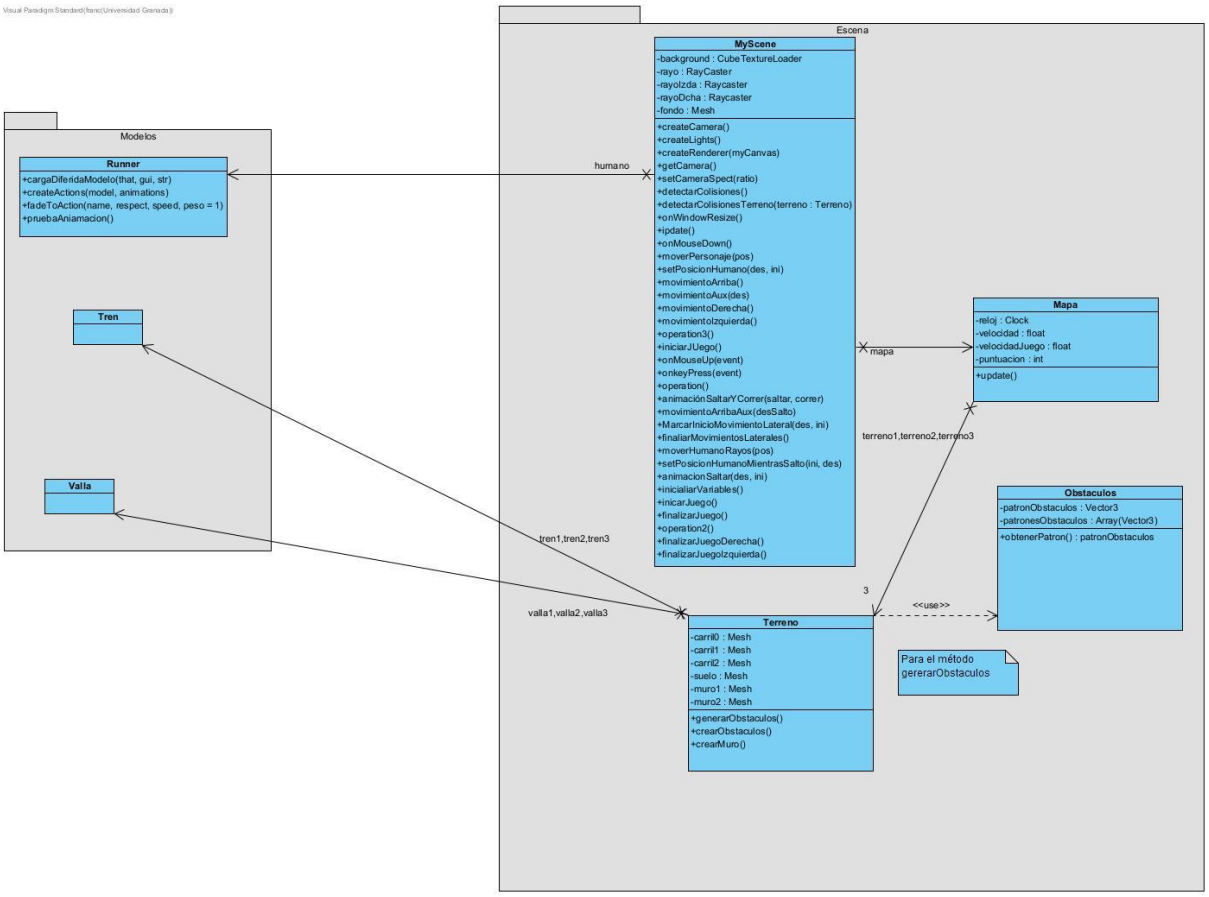
Imagen del mapa y sus componentes.

La clase terreno, está compuesta por 3 carriles, cada uno de ellos mantiene en una posición, los modelos de los obstáculos tren y valla, que por defecto tiene el atributo visible = false para que no aparezcan en la escena. El terreno puede llamar a la función de genera obstáculos, que se encargará de crear un patron en la clase Obstáculos, de forma que nos devuelve un array de 3 elementos, elegidos de forma aleatoria entre un conjunto de patrones. Comprobamos los valores de los elementos del array y establecemos el atributo visible dependiendo de si el obstáculo es una valla, un tren o no es nada.

La clase obstáculo simplemente tiene un array de 3 obstáculos, que pueden ser valla, tren y nada. También tiene un array de patrones del cual se obtiene un patrón cuando se llama a la función obtenerPatron de manera aleatoria.

Problemas con la generación de terreno. Para desplazar los distintos terrenos por el mapa, simplemente modificamos su posición en el método update, pero al llegar a la zona de regeneración y situarse de nuevo al final de los carriles, si la velocidad del juego es muy alta, se generaban unos huecos entre terreno y terreno. Para solucionarlo, simplemente añadimos una longitud extra sobre los terrenos, de forma que se sobrepongan un poco entre ellas para sopesar el desfase producido por la modificación de la posición.

Diagrama de Clases



3. Referencia Modelos.

Modelo Humano y animaciones:
https://www.mixamo.com/#/?page=1&type=Motion%2CMotionPack
Unión de GLTF : Aplicación Blender

4. Manual

Al cargar el juego, puedes pulsar espacio para empezar la partida o pulsar la A para mostrar los controles. Una vez comienza la partida, puedes controlar el movimiento del personaje con las flechas del teclado o haciendo click con el ratón, moviéndolo en alguna dirección y finalmente soltarlo.