PROYECTO FINAL - BLAST VOID

1. Introducción y temática

El proyecto final que he realizado se titula *Blast Void,* y se trata de un pequeño videojuego prototipo creado utilizando la librería de físicas Physx de NVIDIA.
En Blast Void el jugador se encuentra en un planeta remoto en el que debe jugar a un deporte llamado *blasting,* el cual consiste en que hay que meter una pelota en la *zona de blast* para conseguir un punto*,* utilizando únicamente diferentes armas cuyas balas darán un empuje a la pelota.
Cada arma tiene sus propias características: tiempo de cadencia, formas en las que se disparan las balas, número de disparos cada vez que se aprieta el gatillo, etc. En este torneo, el jugador dispone de dos armas diferentes por las que tendrá que conseguir hacer un punto en cada una de las tres fases del torneo. Cada una de las armas acaba generando una colisión diferente con la pelota y es el jugador el que debe elegir la mejor estrategia para superar el nivel en el menor tiempo posible.
Pero, recordemos que nos encontramos en un planeta inhóspito alejado del nuestro, no podremos estar tan tranquilamente. Por ello, el jugador dispondrá de un TEV o "Traje Especial de Vuelo" que le proporcionará un jetpack de tecnología punta con el que podrá elevarse del terreno para llegar a puntos que no podría llegar de otro modo y realizar disparos y jugadas nunca vistas.

2. Efectos incorporados

El jugador comienza en una escena de menú en la cual se encuentra con los tres portales a los tres diferentes niveles. Para acceder a cualquiera de esos niveles debe disparar a uno de esos portales y se teletransportará a dicho nivel.

Para volver al menú antes de terminar un nivel, el jugador podrá acercarse a la partícula que se encuentra **flotando sobre un líquido** en una de las esquinas del escenario.

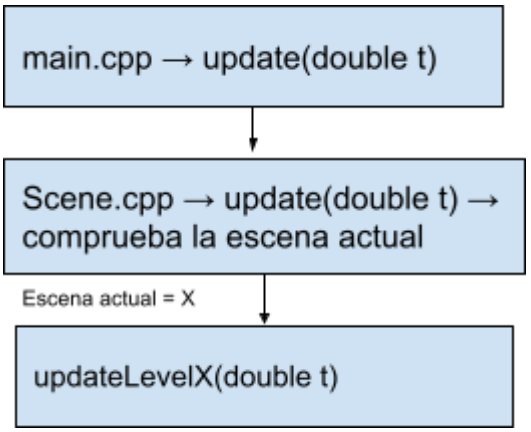
El jugador podrá caminar con normalidad hacia delante y hacia los lados, y además, dispondrá de un jetpack que le impulsará si así lo desea, consumiendo combustible para ello. Dicho combustible se recargará (más lentamente de lo que se gasta) mientras no utilice el jetpack.

En los niveles encontraremos diferentes elementos: obstáculos estáticos (**objetos sólidos**) en los que la pelota rebotará, **generadores de fuerzas de viento y torbellino** que empujarán a la pelota (acompañados de **efectos de partículas**) y unos **fuegos artificiales** que se lanzarán cuando el jugador meta la pelota en la *zona de blast.*

Además de los generadores de partículas de los generadores de viento y torbellino, **cada arma funciona como un generador de sólidos rígidos**.

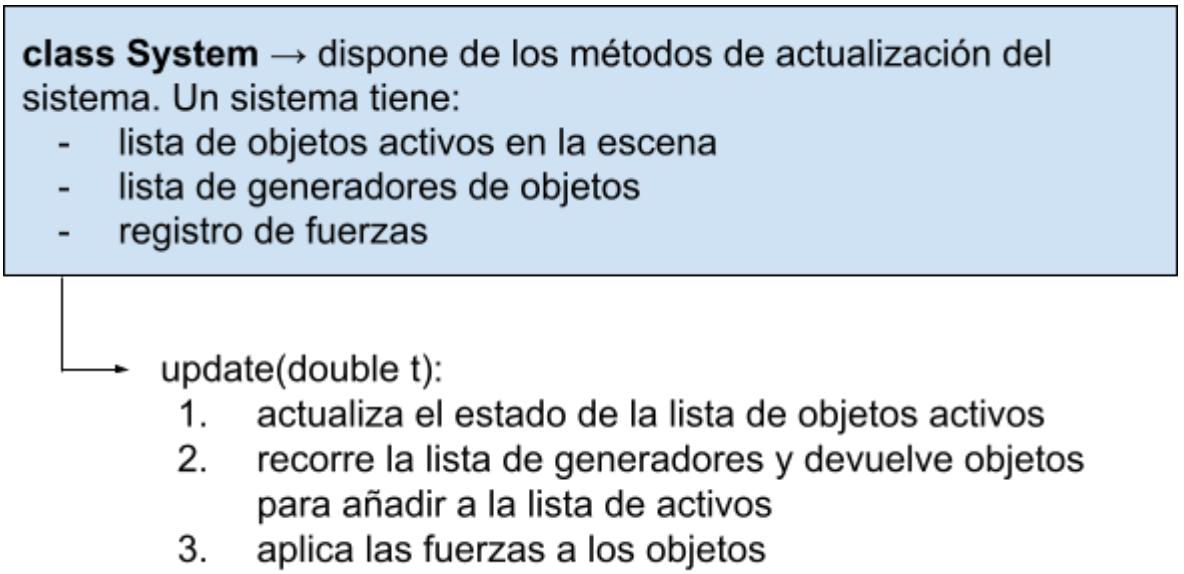
3. Diagrama de clases del proyecto

En el proyecto he implementado un sistema de gestor de escenas. En el main.cpp se encuentran las llamadas del ciclo de vida de physx integradas en el proyecto, y el gestor de escenas de la clase Scene tiene un método homónimo para cada una de esas llamadas (update, release, onKeyChanged, etc). Dentro de la clase Scene, existen también las llamadas a estos métodos para cada una de las escenas de juego.

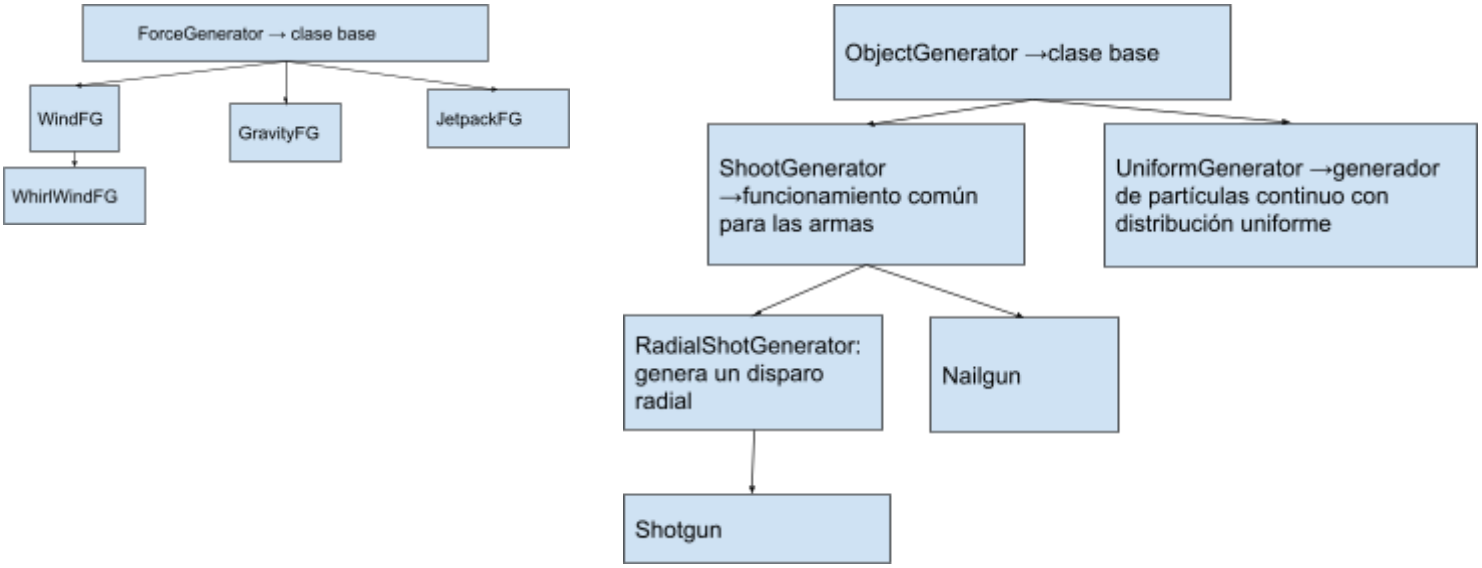


Por lo tanto, main.cpp hará una llamada a Scene.cpp, que funcionará como gestor de escenas llamando a los métodos correspondientes según la escena de juego.

En cuanto a los sistemas implementados, disponemos de un sistema de partículas y un sistema de sólidos rígidos dinámicos. El funcionamiento es el mismo, así que por simplicidad lo explicaré como uno único, “Object”, pero que realmente se divide en dos: “Particle” y “RigidBody”. La única diferencia es que uno trabaja con objetos de la clase Particle y otro trabaja con objetos de la clase DynamicRigidBody (clase propia que hereda de RigidBody y de la cual surgen Dynamic y Static Rigidbody, que funcionan como una abstracción de los PxRigidbody que ofrece Physx).



En cuanto al resto de elementos, el esquema es el siguiente:



4. Fórmulas físicas y parámetros usados en las ecuaciones

a. Generador de partículas uniforme

Para utilizar el generador de partículas uniforme, lo que indicamos es un ancho en cuanto a la velocidad y la posición que queremos para nuestras partículas, es decir, los valores que queremos utilizar a la hora de realizar la distribución uniforme para cada una de las componentes (x,y,z) de la velocidad y posición de las partículas generadas.
Como mínimo y máximo para la distribución uniforme utilizaremos -1 y 1, y utilizaremos el valor dado por dicha distribución para multiplicarlo por el ancho de la componente, para finalmente sumarlo al origen.

$$Posición = posición\ origen + (Ancho\ de\ posición * distribución\ uniforme)$$
$$Velocidad = velocidad\ origen + (Ancho\ de\ velocidad * distribución\ uniforme)$$

b. Arma

Ambas armas (Shotgun y Nailgun) son clases que heredan de ShotGenerator, el cual generará sólidos rígidos siempre de la misma manera. Cada arma tiene unos valores numéricos para indicar:

- el número de balas que disparará en cada disparo ejercido,
- cuántos disparos se realizarán en cada ráfaga,
- cuánto tiempo de retroceso hay entre cada ráfaga,
- cuánto tiempo de retroceso hay entre cada disparo de la ráfaga.

Cambiando a nuestro gusto cualquiera de estos valores y personalizando la generación de cada bala según el funcionamiento ideado, podemos utilizar esta clase base para crear las armas que queramos según el diseño del videojuego. En este caso he creado una escopeta y una “pistola de clavos”.

i. Shotgun

La boca de la escopeta tiene tres orificios que forman un triángulo. No obstante, no siempre formará un triángulo perfecto, para así poder generar pequeñas diferencias en cómo ocurrieron los disparos y que la reacción al colisionar con la pelota no sea siempre igual: dependerá de la orientación de la cámara ligeramente.

Pese a formar un triángulo, realmente eso ocurre porque he indicado que el arma tenga tres orificios. No obstante, la fórmula matemática lo que hace es colocar x puntos en una circunferencia, por lo que añadiendo más orificios obtendremos diferentes formas, como un pentágono, hexágono o figuras más cercanas a una circunferencia. Este funcionamiento está descrito en la clase RadialShotGenerator, funcionando así la clase Shotgun para simplemente dar los valores de velocidad (0,8,0, masa, damping, color de las balas, tiempos entre ráfagas y disparos, número de balas.

$$\text{a} = Ángulo\ de\ rotación\ = \frac{360}{num.\ balas} \times \frac{\pi}{180}$$

$$\text{s} = Pos.\ inicio\ de\ rotación(x, y)\ = (0 - radio\ ,\ 0)$$

$$x\ =\ cos(a\ *\ bala\ actual)\ *\ sx\ -\ sin(a\ *\ bala\ actual)\ *\ sy$$

$$y\ =\ cos(a\ *\ bala\ actual)\ *\ sy\ -\ sin(a\ *\ bala\ actual)\ *\ sx$$

Nueva Posición = (x,y)

$$Posición\ final(x, y, z)\ =\ Posición\ origen\ +\ Nueva\ Posición$$

ii. Nailgun

Este arma no hereda de RadialShotGenerator puesto que la intención era disparar tres proyectiles (clavos) formando una línea recta. El único valor a indicar a la hora de construir una nueva pistola de clavos es la distancia que quieres que haya entre cada uno de los clavos disparados, formando una línea recta con sus elementos más o menos separados. Además, podemos indicar el número de balas (clavos) que queremos que haya, pudiendo diseñar incluso cierta progresión en el videojuego (mejorar el arma consiguiendo más o menos orificios para disparar).

c. Generador de fuerzas: viento uniforme

$$\vec{Fv} = k1(vv - v) + k2\|vv - v\|(vv - v)$$

vv = velocidad del viento. v = velocidad del objeto. k1, k2 = coeficientes de rozamiento del aire.
Este generador de fuerzas simulará un viento uniforme que seguirá la dirección indicada por el vector de velocidad del viento.
Valores usados:

- Para las partículas: k1 = (0.5-1], k2= {0,0.01}
- Para los sólidos rígidos: k1 = [15,18], k2 = 2

d. Generador de fuerzas: torbellino

$$vtorbellino(x, y, z) = K[-(z - zc), 50 - (y - yc), x - xc]$$

c = posición del centro del torbellino; x,y,z = posición del objeto
Al igual que el generador de fuerzas de viento uniforme, también usa un coeficiente de rozamiento del viento (no utiliza el segundo).
Valores usados:

- Para las partículas: K = 5
- Para los sólidos rígidos: K = 0.6

e. Generador de fuerzas: flotación

$$E = Fg \cdot Pagua \cdot Vsumergido$$

Para este generador de fuerzas, la fuerza final ejercida dependerá de lo que el objeto se ha llegado a sumergir en nuestro fluido simulado (simplemente una posición), además de la masa y damping del objeto.
Valores usados:

- Para el objeto que flota: m = 5.0, damping = 0.8
- Para el líquido: volumen = 1, densidad = 1000, posición = posición del objeto -2 (eje y)

f. Generador de fuerzas: jetpack

Para permitir al jugador elevarse del juego he implementado un generador de fuerza vertical para simular un jetpack. Para que no sea simplemente una fuerza que te eleva verticalmente como el viento, existe un sistema de combustible que limita la potencia de vuelo.

- Si el combustible >= ⅔ del combustible máximo:
  f = {0,potencia,0}
- Si el combustible es ⅓ máximo >= combustible actual <= ⅔ combustible máximo
  f = {0, potencia* 0.66, 0}
- Si el combustible es 0.5/3 máximo >= combustible actual <= ⅓ combustible máximo
  f = {0,potencia*0.4,0}
- Si el combustible es = 0
  fuerza = {0,0,0}

Así, teniendo una constante de potencia para el jetpack, podemos dar diferentes valores de fuerza ejercida dependiendo del combustible que nos quede en el depósito: cuanto menos combustible haya, menos potencia de vuelo tendremos. De esta manera, se deberá hacer buen uso del combustible de vuelo para así perder la mínima potencia posible, pero siendo imposible mantenerla del todo si se pretende volar mucho, obligando al jugador a organizar sus jugadas y pensar bien el siguiente movimiento, aprovecharse de las fuerzas de viento para dejar la pelota en el aire y bajar al suelo a recargar el depósito, etc.
El jetpack tendrá también una constante *coste*, la cual indicará cuánta energía del depósito gastará por fotograma. Además, también indica a qué velocidad se recuperará la energía, puesto que la velocidad de recarga es *0.5*coste.*