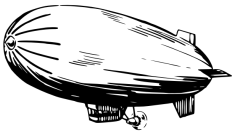


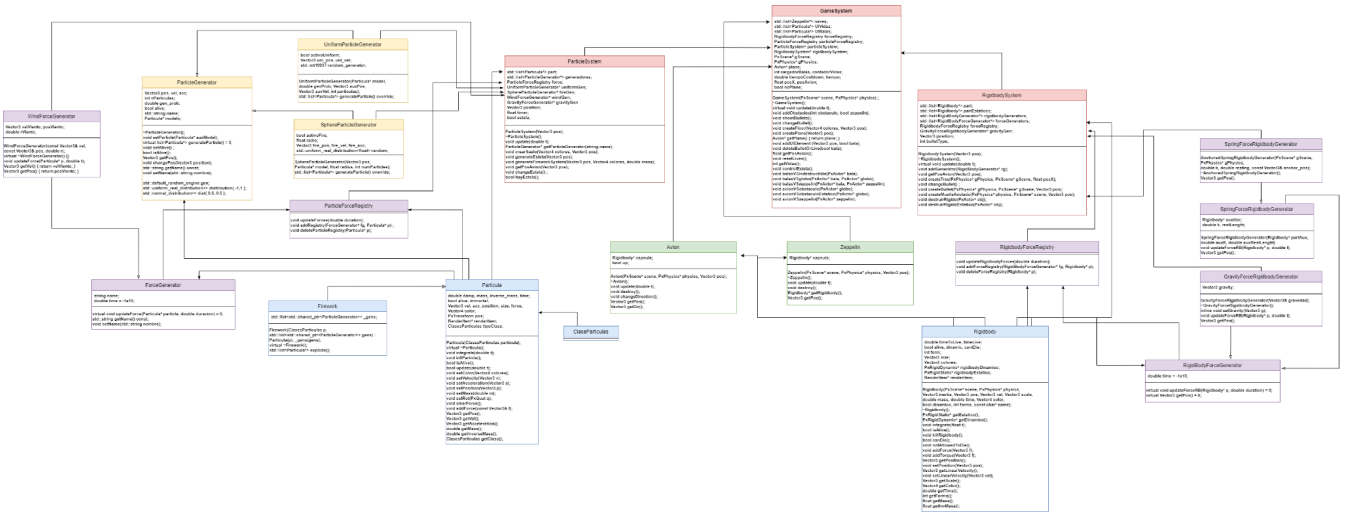
Tiny Zeppelin - Memoria

1. Descripción del juego:

“Tiny Zeppelin” es un videojuego que pertenece a la categoría de Endless Run, con una perspectiva 2D.

En el transcurso de una partida, el jugador tendrá que sortear los diferentes obstáculos (árboles, globos y naves) que aparecerán en pantalla, hasta que pierda todas sus vidas y obtenga su puntuación final. Tras terminar, podrá reiniciar el juego e intentar conseguir mejorar sus resultados previos.

2. *Diagrama de clases:



*Para ver la imagen con mejor resolución, hacer click aquí

3. Ecuaciones:

- **Avion + Zeppelin**

    - **Movimiento:** El avión posee un movimiento ininterrumpido hacia la derecha, pero el jugador puede influir en el desplazamiento vertical del personaje. Cambiando el booleano de dirección, dependiendo de la posición ‘y’, la velocidad será positiva, negativa o nula en el eje. Los zepelines tendrán una posición ‘y’ fija, desplazándose con velocidad en ‘x’ negativa.

```
Vector3 pos = capsula->getPosition() + capsula->getLinearVelocity();
capsula->setPosition(pos);
```

- **Particula**

    - Actualizamos la posición de la partícula de forma proporcional al tiempo que ha transcurrido y la velocidad (que depende de las fuerzas externas, la masa y el tiempo) que lleva.

```
pos.p += vel * t;
auto totalacc = force * inverse_mass;
vel = vel * pow(damp, t) + totalacc * t;
(...)
clearForce();
```

- **RigidbodySystem**

    - **Cuando creamos un globo, establecemos valores aleatorios para k y resting, manteniéndolos ambos dentro de un rango establecido para que no salgan por los bordes de la pantalla.**

```
int resting = rand() % 10 + 30;
int k = rand() % 40 + 60;
```

    - **Comprobamos si el rigidbody, el generador o la partícula del System ha sido eliminada (isAlive) o está situada detrás del avión, para borrar estas de las listas (En el ParticleSystem también funciona con esta fórmula)**

```
!(*it)->isAlive() || (*it)->getPosition().x + 100.0 < position.x)
```

- **UniformParticleGenerator**

    - Realizamos el cálculo de las nuevas posiciones y velocidades de las partículas creadas por el generador, teniendo en cuenta la distribución normal.

```
auto gen = std::uniform_int_distribution<int>(0, 100);
auto px = std::uniform_real_distribution<float>(pos.x - uni_pos.x / 2, pos.x + uni_pos.x / 2); (...);
auto vx = std::uniform_real_distribution<float>(vel.x - uni_vel.x / 2, vel.x + uni_vel.x / 2); (...);
for (int i = 0; i < nParticulas; i++) {
        int cr = gen(random_generator);
        if (cr <= gen_prob) {
        Vector3 pos = { px(random_generator), py(random_generator), pz(random_generator)};
        Vector3 vel = { vx(random_generator), vy(random_generator), vz(random_generator) };
        Particula* p = new Particula(modelo->getClass());
        p->setVelocity(vel); p->setPosition(pos);
        listParticles.push_back(p); }}
```

- **SphereParticleGenerator**

    - Generamos las partículas en una posición aleatoria dentro de una esfera definida, añadiendo velocidad en esas direcciones para que salgan propulsadas fuera del rango del rango.

```
std::random_device r{};
std::mt19937 gen{ r() };
float theta = 0, phi = 0;
for (int i = 0; i < nParticulas; i++) {
        phi = 2 * 3.14 * random(gen);
        theta = 2 * 3.14 * random(gen);
        float x = cos(theta) * sin(phi);
        float y = sin(theta) * sin(phi);
        float z = cos(phi);
        Vector3 vel(x, y, z);
        Particula* p = new Particula(modelo->getClass());
        p->setPosition(pos);
        p->setVelocity(vel.getNormalized() * 20);
        listParticles.push_back(p); }
```

- **GravityForceRigidbodyGenerator**

    - Aumentamos el peso de la masa con el vector de gravedad, y añadimos el resultado como fuerza que afecta al sólido. El valor usado en el muelle es: {0.0f, -6.0f, 0.0f}

```
p->addForce(gravity * p->getMass());
```

- **SpringForceRigidbodyGenerator**

    - Calculamos la fuerza teniendo en cuenta la distancia que hay entre un sólido y otro (en este caso, la sujeción del muelle), restando dicho resultado a la longitud máxima del muelle en reposo. Después, multiplicamos por la constante de elasticidad y añadimos la fuerza.

```
Vector3 force = auxiliar->getPosition() - p->getPosition();
const float lenght = force.normalize();
const float deltaX = lenght - restLenght;
force *= deltaX * k;
p->addForce(force);
```

- **WindForceGenerator**

    - Calculamos la diferencia de velocidad entre el viento y la partícula, multiplicando el resultado por las fuerzas de rozamiento del aire establecidas, que harán que la partícula se mueva en una dirección u otra.

```
Vector3 v = velViento - p->getVel();
p->addForce(0.2 * v + 0.001 * v.magnitude() * v);
```

- **GameSystem**

    - Para indicar que se ha recuperado una de las balas perdidas, hay un método que añade una partícula en la esquina superior de la pantalla, teniendo que usar para ello la posición actual del jugador y el tamaño del vector de balas de la UI, para no posicionar esta fuera de los bordes o debajo de otra bala.

```
float((plane->getPos().x + 85.0) - 5.0 * UIBalas.size())
```

    - Todos los elementos de la interfaz y la cámara se desplazan junto con el jugador, actualizando su posición con cada update que se realiza.

```
float((*it)->getPos().x + 0.01)
```

    - Cuando sucede una colisión con elementos declarados en el GameSystem (avión o zepelines), se genera una explosión de fuegos dada la posición que tenía el objeto que ha sido eliminado.

```
particleSystem->generateFireworkSystem((*i)->getPos(), { 1.0, 0.5, 0.0, 1.0 }, 5.0);
```

4. Efectos y experimentos

- Hay un sistema de generación aleatoria de objetos cada X segundos, siempre que haya un jugador presente. Estos están posicionados en una posición predeterminada, que aumenta con cada objeto añadido en la escena.
- Existen diferentes tipos de disparos, cada uno con valores de velocidad, escala, inercia y peso diferentes.
- Puede activarse y desactivarse la estela del avión, apareciendo siempre detrás del jugador.
- Las teclas de movimiento de la cámara están desactivadas, teniendo esta un desplazamiento constante que sigue al jugador. Además, la rotación y posición de la cámara han sido modificadas para dar aspecto de juego 2D.
- Existe diferentes métodos para lidiar con las colisiones entre objetos, teniendo en cuenta si son entre unos u otros (por ejemplo, una colisión entre una bala y un árbol destruye solo a la bala, pero un impacto entre el jugador y un árbol elimina a ambos)
- Cuando el jugador es destruido, el movimiento y la creación de objetos queda detenida hasta que el jugador vuelva a retomar la partida.
- Hay implementada una IU que indica el número de balas y vidas que dispone el jugador, regenerándose las balas cada cierta cantidad de tiempo.
- Por último, hay un sistema de puntuación que está determinado por la distancia recorrida más el número de obstáculos destruidos con balas.

5. Manual de usuario:

A	Inicio de partida y seguir jugando después de perder una vida
Z	Cambio de dirección en el eje vertical
X	Disparo
C	Cambio de proyectil
S	Desactivar o activar estela

6. Anotaciones extras:

- Existe un fallo con las colisiones, habiendo detección después de haber sucedido esta y pudiendo impedir que, en el caso de que sea el avión uno de los afectados, pueda seguir jugando.
- Después de un rato, el juego puede comenzar a ralentizarse. Tener la estela activada también puede provocar un efecto similar.