THEATER ANTARTICA 2D

1. Temática, historia y objetivo del juego

La historia subyacente a nuestro juego es la de dos pingüinos que viven en una región de permanentes nevadas, veloces torbellinos e icebergs, que buscan eliminarse mutuamente con el fin de coronarse rey de esos gélidos parajes, para ello habrán de destruir al oponente sin piedad.

Theatre Antartica 2D es un pequeño “shooter” táctico en vista 2D donde el objetivo simple de ambos jugadores es impactar al otro mediante el lanzamiento de un proyectil. Cuando uno de ellos consiga impactar en el otro, las plataformas y los jugadores cambiarán de lugar, y el turno pasará al contrario.

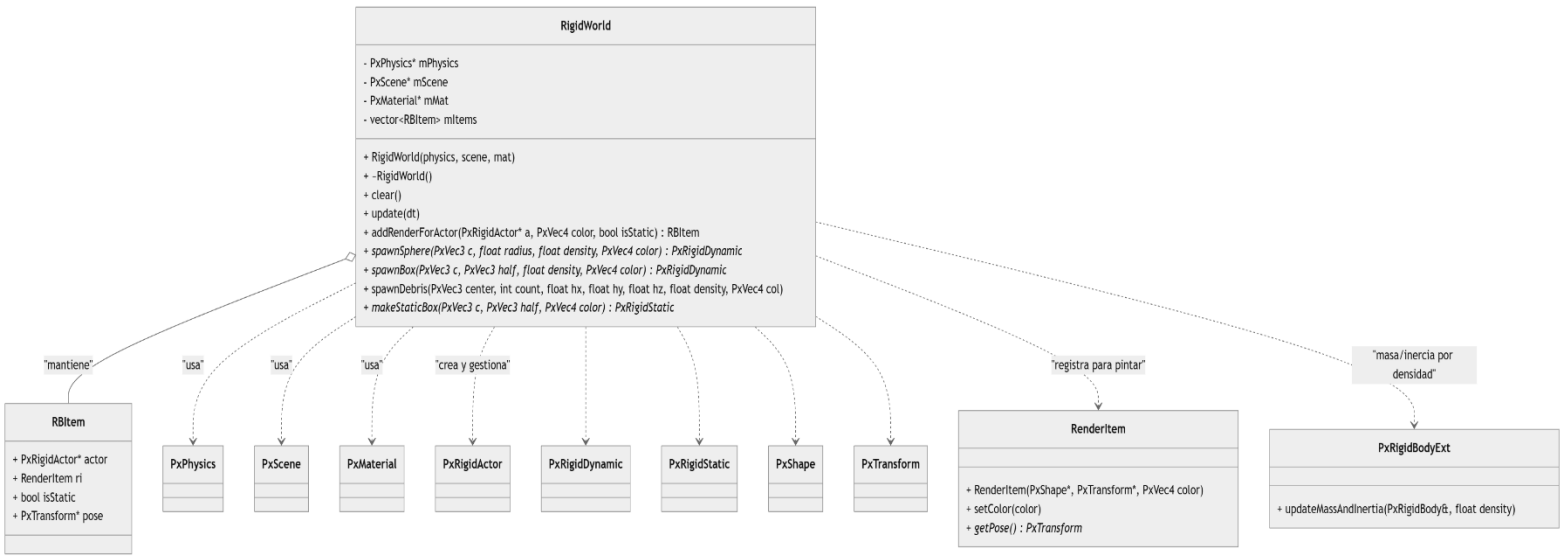
Tenemos varios tipos de proyectil, los cuales varían entre sí en cuanto a masa y tamaño y por ende se ven influenciados por el viento de cada ronda el cual varia en cuanto en intensidad y sentido de un turno a otro.

Además, a la hora de realizar el lanzamiento podemos modificar el ángulo de lanzamiento, en un abanico de entre 10 y 80 grados, así como la fuerza que se le aplica al disparo en función de cuanto mantengamos presionada la tecla de lanzamiento.

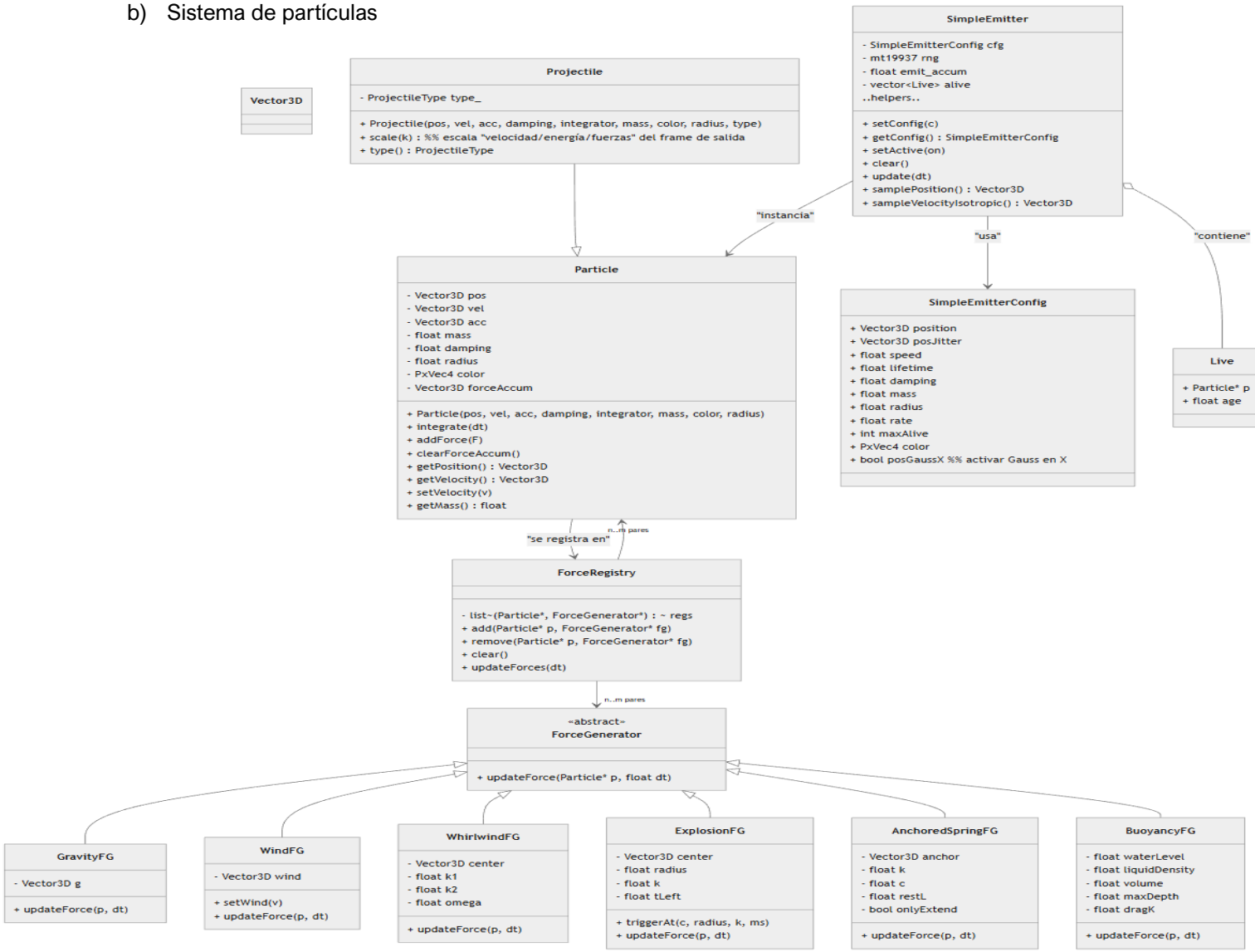
Al ser un juego infinito y sin límite de puntuación los jugadores pueden pasar entre rondas indefinidamente.

2. Diagramas de clase del sólido rígido y del sistema de partículas

a) Sólido rígido



b) Sistema de partículas



3. Breve explicación de las ecuaciones físicas usadas, así como de los valores dados a los parámetros de dichas ecuaciones, y líneas y ficheros donde está implementado el requisito/efecto/ecuación indicados (he juntado aquí los apartados C y E):

a. Dos generadores de partículas distintos (uniforme y gaussiano):

Uniforme→ $x \sim U[x0-jx, x0+jx]$

Gauss truncado en X → $x \sim N(x0,\sigma)$ y luego para x se hace →clamp(x,x0−jx,x0+jx)

```
Vector3D SimpleEmitter::samplePosition() {
    // X: uniforme por defecto; normal si se activa en cfg
    float x;
    if (cfg.posGaussX) {
        const float xmin = cfg.position.x - cfg.posJitter.x;
        const float xmax = cfg.position.x + cfg.posJitter.x;
        const float sigma = std::max(0.0001f, 0.35f * cfg.posJitter.x); // campana visible
        float g;
        // normal truncada
        do { g = randNormal(cfg.position.x, sigma); } while (g < xmin || g > xmax);
        x = g;
    }
    else {
        x = cfg.position.x + randUniform(-cfg.posJitter.x, +cfg.posJitter.x);
    }
    // Y y Z uniformes
    const float y = cfg.position.y + randUniform(-cfg.posJitter.y, +cfg.posJitter.y);
    const float z = cfg.position.z + randUniform(-cfg.posJitter.z, +cfg.posJitter.z);
    return { x, y, z };
}
```

```
struct SimpleEmitterConfig {
    Vector3D position{ 0,0,0 };
    Vector3D posJitter{ 0,0,0 };      // +/- por eje
    float    speed = 5.0f;            // velocidad escalar
    float    lifetime = 2.0f;         // s
    float    damping = 0.99f;
    physx::PxVec4 color{ 0.2f,0.6f,1.0f,1.0f };
    float    radius = 0.12f;
    float    rate = 6.0f;         // partículas por segundo
    int      maxAlive = 300;          // cap por emisor
    bool     active = false;       // inicia apagado
    float mass = 1.0f;
    bool posGaussX = false;
    bool gaussX = false;
    float sigmaX = 1.5f; //desviacion tip
};
```

Como en mi juego solo se utilizan los ejes X e Y para que sea 2D y los emisores se disponen en una línea imaginaria del eje X y luego caen a creando un efecto de nieve tengo dos emisores normales en **blanco** que esparcen de manera uniforme y uno gaussiano en **azul** que concentra más en zona central.

- SimpleEmitter.cpp: 74-110
- Main.cpp: 679-714

b. **Gestión de creación y destrucción de instancias (TTL)**

A cada partícula que tiene una duración limitada se le fija un tiempo de vida (y a mayores están acotados unos límites invisibles como segunda barrera de seguridad), cada frame se integra, acumula edad y cuando supera su lifetime esta partícula se borra.

SimpleEmitter.cpp: 141-213
RigidWorld.cpp: 26-44

```cpp
void SimpleEmitter::update(float dt) {
    if (dt <= 0.0f) return;

    // integrar y borrar por lifetime
    for (auto it = alive.begin(); it != alive.end(); )
    {
        Particle* p = it->p;
        p->integrate(dt);
        it->age += dt;

        const auto pos = p->getPosition();
        const bool deadTime = (it->age >= cfg.lifetime);
        const bool deadKill = (pos.y < -30.0f); // mismo kill-plane que en main

        if (deadTime || deadKill) {
            if (gForceReg) gForceReg->removeAll(p);
            delete p;
            it = alive.erase(it);
        }
        else {
            ++it;
        }
    }
```

c. **Destrucción de elementos al salir de escena**

Al cambiar de modo o escena se liberan todas las instancias, registros, contenedores y memoria (fuerzas, emisores, proyectiles, sólidos, etc…) Hay diferentes métodos auxiliares de limpieza.

Main.cpp: 231-238, 538-543, 757-778, 1237-1280

```cpp
static void ExitTheaterMode() {
    // limpiar proyectiles del juego
    for (auto* p : gProjectiles) { KillParticle(p); }
    gProjectiles.clear();
    //borrar jugadores
    if (gPlayerVis[0]) { delete gPlayerVis[0]; gPlayerVis[0] = nullptr; }
    if (gPlayerVis[1]) { delete gPlayerVis[1]; gPlayerVis[1] = nullptr; }

    ClearPlatforms();

    // eliminamos los generadores de F del juego
    delete gGravGame; gGravGame = nullptr;
    delete gWindGame; gWindGame = nullptr;

    //eliminamos los decorados del juego
    DestroyTheaterStage();
    ClearSpringBobs();
    for (auto* p : gFloaters) { KillParticle(p); }
    for (auto* b : gFloatersBuoy) { delete b; }
    gFloaters.clear();
    gFloatersBuoy.clear();
}
```

```cpp
//LIMPIEZA PRINCIPAL
ClearScene();
ClearProjectiles();
ClearEmitters();
```

```cpp
// Rigid Body ++++++++++++++++++++++++++++++++++++++++
if (gRigid) { gRigid->clear(); delete gRigid; gRigid = nullptr; }
```

```cpp
static void ClearSpringBobs()
{
    for (auto* p : gSpringBobs) { KillParticle(p); }
    gSpringBobs.clear();

    for (auto* fg : gSpringFGs) { delete fg; }
    gSpringFGs.clear();
}
```

d. Partículas con distinta masa (ej. Proyectiles y emisores)

Por ejemplo, tanto los proyectiles que spawnean los jugadores, como las partículas de los emisores tienen una masa customizable. En el caso de los disparos es más representativo por haber una gran diferencia, no como en los emisores que les doy una masa más homogénea para que el efecto sea consistente.

Main.cpp: 573-577, 688,

```
switch (projType) {
case 0: mass = 1.0f; damp = 0.990f; radius = 0.35f; color = physx::PxVec4(1.f, 0.85f, 0.2f, 1.f); break; // bala
case 1: mass = 1.5f; damp = 0.995f; radius = 0.80f; color = physx::PxVec4(0.2f, 0.2f, 0.2f, 1.f);   break; // canon
case 2: mass = 0.3f; damp = 0.995f; radius = 0.30f; color = physx::PxVec4(0.7f, 0.9f, 1.f, 1.f); break; // helio
}
```

e. Sistema de sólidos-rígidos

Se crean “actores” dinámicos de dos formas (y dentro de ellas cierta aleatoriedad en cuanto a volumen, para que con la densidad que tiene cada cuerpo varíe su tensor → cosa que calcula el propio motor). Teniendo posición **x**, velocidad lineal **v**, velocidad angular **ω,** masa **m**, y tensor de incercia **I.**

$mv\dot{}=\sum F$ ---- $I\omega\dot{}+\omega\times(I\omega)=\sum\tau$

Cuando “explota” a cada “esquirla” se le da un impulso lineal muy grande en un espacio muy corto de tiempo, de la misma manera que se hace con los proyectiles cuando se lanzan **Δv=J/m,** ese impulso lo lanza y hace colisión con los objetos estáticos rígidos de la escena (véase las paredes y otras “esquirlas” y como hace fricción por punto de contacto genera un torque que es lo que los hace girar).

```
void RigidWorld::spawnDebrisExplosion(const physx::PxVec3& center, int count, float v0)
{
    using namespace physx;

    auto frand = [](float a, float b) { return a + (b - a) * (rand() / (float)RAND_MAX); };
    auto rand01 = []() { return rand() / (float)RAND_MAX; };

    for (int i = 0; i < count; ++i) {
        // color aleatorio para ver bien las piezas
        PxVec4 col(0.35f + 0.65f * rand01(),
            0.35f + 0.65f * rand01(),
            0.35f + 0.65f * rand01(), 1.0f);

        // ligera dispersión de origen para evitar overlap
        PxVec3 c = center + PxVec3(frand(-0.10f, 0.10f),
            frand(-0.10f, 0.10f),
            frand(-0.02f, 0.02f));

        // vida 4–8 s para auto-limpieza
        const float life = frand(4.0f, 8.0f);

        // 50% cubos 50% bolitas
        const bool makeSphere = (i % 2 == 0);

        PxRigidDynamic* body = nullptr;
        if (makeSphere) {
            // esfera: radio aleatorio pequeñito
            const float r = frand(0.08f, 0.14f);
            const float density = 300.0f; // mitad que la caja para variar inercia y que se vean mejor ambos
            body = spawnSphere(c, r, density, col);
        }
        else {
            // caja: medidas aleatorias pero con poca variación y pequeñas tabmien
            const float hx = frand(0.08f, 0.24f);
            const float hy = frand(0.08f, 0.24f);
            const float hz = frand(0.08f, 0.24f);
            const float density = 600.0f;
            body = spawnBox(c, PxVec3(hx, hy, hz), density, col); // spawnBox
        }

        // marca TTL para que desaparezcan
        if (auto r = findItemByActor(body)) r->life = life;

        // impulso radial y pequeño giro
        const float ang = (float)i / (float)count * 6.2831853f;
        PxVec3 dir(cosf(ang), sinf(ang), 0.0f);

        body->addForce(v0 * dir * body->getMass(), PxForceMode::eIMPULSE);
        body->addTorque(PxVec3(frand(-2.0f, 2.0f),
            frand(-2.0f, 2.0f),
            frand(-2.0f, 2.0f)), PxForceMode::eIMPULSE);
    }
}
```

f. Sólidos rígidos de distinto tamaño, forma, masa y tensor de inercia.

Continuando con el apartado anterior, tenemos esferas y cajitas que hacen las veces de “esquirlas de metralla” diferentes entre si en tamaño, forma, masa y por ende tensor de inercia.

Pero el cálculo real de las fórmulas, lo hace el propio PhysX que calcula masa e inercia cuando se le pasan la densidad y la geometría del cuerpo en particular que es lo que se define:

Esfera de radio r: $\mathbf{I} = \frac{2}{5}mr^2\,\mathbf{I}_3$

Caja (a, b, c = lados):

$$I_x = \frac{1}{12}m(b^2 + c^2),\ I_y = \frac{1}{12}m(a^2 + c^2),$$

$$I_z = \frac{1}{12}m(a^2 + b^2)$$

Paramétricamente se dan:

- Densidad: 600.0f para caja 300.0f para las esferas y que haya diferencia visible en cuanto a la inercia.
- Radio de esfera 0.08f-0.12f
- Mitades de las caras en x,y,x para que estén centradas y no partan de un vértice: también entre 0.08f y 0.12f.

RigidWorld.cpp: 62-140

```
PxRigidDynamic* RigidWorld::spawnSphere(const PxVec3& c, float radius, float density,
    const PxVec4& color) {
    PxRigidDynamic* body = mPhysics->createRigidDynamic(PxTransform(c));
    PxShape* shp = mPhysics->createShape(PxSphereGeometry(radius), *mMat);
    body->attachShape(*shp);
    shp->release();

    PxRigidBodyExt::updateMassAndInertia(*body, density);
    mScene->addActor(*body);
    addRenderForActor(body, color,false);
    return body;
}

PxRigidDynamic* RigidWorld::spawnBox(const PxVec3& c, const PxVec3& half,
    float density, const PxVec4& color) {
    PxRigidDynamic* body = mPhysics->createRigidDynamic(PxTransform(c));
    PxShape* shp = mPhysics->createShape(PxBoxGeometry(half.x, half.y, half.z), *mMat);
    body->attachShape(*shp);
    shp->release();

    PxRigidBodyExt::updateMassAndInertia(*body, density);
    mScene->addActor(*body);
    addRenderForActor(body, color, /*static*/false);
    return body;
}
```

g. Dos generadores de fuerzas diferentes

Los dos más visibles y que no haya mencionado son el **viento** que afecta a los proyectiles de los jugadores, y el **torbellino** en mitad de la escena. (porque el impulso de la explosión y del lanzamiento como tal ya son uno cada uno aunque similares puesto que se aplica el mismo principio de aplicar una fuerza muy elevada en un espacio muy pequeño de tiempo, lo que causa una impulso inicial muy fuerte, que luego decrece en función del tiempo y las fuerzas intervinientes)

Viento: $\mathbf{F_w = k(w-v)}$ se aplica una fuerza resultante del drag lineal y la diferencia de velocidades del viento y del cuerpo en cuestión, en este caso solo en el eje X porque está dispuesto para que solo intervenga en la componente X de los proyectiles (entre -1.0f y 1.0f para que sea jugable y no frustrante).

```
void WindFG::updateForce(Particle* p, float /*dt*/) {
    if (!p) return;

    // arreglado con early return
    if (k1_ == 0.0f && k2_ == 0.0f) return;

    // v_rel = v_viento - v_particula
    Vector3D vRel = (vWind_ - p->getVelocity());

    // F = k1*vRel + k2*|vRel|*vRel
    float speed = vRel.length();
    Vector3D F = vRel * k1_;
    if (k2_ != 0.0f && speed > 0.0f) F += vRel * (k2_ * speed);

    p->addForce(F);
}
```

A nivel decorativo se le da **0.3f** también en X para que haya una mínima brisa.

WindFG.cpp: 4-18

```cpp
void WhirlwindFG::updateForce(Particle* p, float /*dt*/) {
    if (!p) return;

    const Vector3D& pos = p->getPosition();
    if (!zone_.contains(pos)) return;

    // Vector desde el centro
    Vector3D r = pos - zone_.getCenter();

    // Componente tangencial en el plano XZ (giro alrededor de Y)
    Vector3D rXZ(r.x, 0.0f, r.z);
    float rlen = fastLenXZ(rXZ);

    Vector3D vTang(0, 0, 0);
    if (rlen > 1e-6f) {
        // tangente = rotación 90° de rXZ en XZ: (-z, 0, x) normalizada
        Vector3D t = Vector3D(-rXZ.z, 0.0f, rXZ.x) * (1.0f / rlen);
        // v_tangencial = omega * r * t
        vTang = t * (omega_ * rlen);
    }


    Vector3D vRad(0, 0, 0);
    if (radialIn_ > 0.0f && rlen > 1e-6f) {
        Vector3D dirIn = (rXZ * (-1.0f / rlen)); // hacia el centro en XZ
        vRad = dirIn * radialIn_;
    }

    // Updraft vertical
    Vector3D vUp(0.0f, updraft_, 0.0f);

    // Velocidad de viento efectiva en la posición
    Vector3D vWind = vTang + vRad + vUp;

    // Fuerza tipo viento: F = k1 * vRel + k2 * |vRel| * vRel
    Vector3D vRel = vWind - p->getVelocity();
    float speed = vRel.length();
    Vector3D F = vRel * k1_;
    if (k2_ != 0.0f && speed > 0.0f) F += vRel * (k2_ * speed);

    p->addForce(F);
```

Torbellino: en la práctica intermedia había un torbellino convencional donde las partículas al estar en la proximidad de la base de este subían y se veía un torbellino de facto, en el teatrillo, para dar un efecto de campana que esparce la nieve ha decidido invertirlo pero la implementación sigue siendo la misma que ya hubiera.

Se aplica **F total= Comp.Tangencial + Comp.Radial + Updraft vertical.** De tal manera que a cada partícula (si está dentro de la esfera de acción) se le aplica dicha fuerza sumatoria

Whirlwind.cpp y Whirlwind.h
Main.cpp: 677

```cpp
private:
    ZoneSphere zone_;
    float omega_;    // rad/s (giro en torno a Y)
    float updraft_;  // m/s (viento vertical +Y)
    float radialIn_; // m/s (succión hacia el centro en el plano XZ)
    float k1_, k2_;  // coeficientes laminar y turbulento
```

h. Interacción con el usuario a través de teclado/ratón

Tenemos únicamente controles de teclado:

- **J**: disminuye el ángulo de lanzamiento.
- **L**: aumenta el ángulo de lanzamiento.
- **Espacio**: carga la potencia del disparo.
- **R**: reinicia la ronda (y con ello cambia el viento de la misma)
- **1,2,3**: selección de proyectil

i. **Ejemplo de muelle o de flotación.**

La flotación en este caso es **Fempuje = p * g * Vsub * ^y** siendo (densidad/módulo de la gravedad/volumen sumergido del cuerpo/ dirección vertical). Ya que solo aplica en el eje Y, no hay desplazamiento ni función en X ni Z. La diferencia entre las boyas y los icebergs viene por la densidad de ambas partículas, las boyas al ser más pequeñas y menos densas tienen una flotación más rápida y los icebergs más lenta.

BuoyancyFG.h y Buoyancy.cpp

```cpp
void BuoyancyFG::updateForce(Particle* p, float /*dt*/)
{
    if (!p) return;
    if (p->getMass() <= 0.0f) return;
    if (height_ <= 0.0f || volume_ <= 0.0f || rho_ <= 0.0f) return;

    // Centro del cuerpo y sus extremos en Y
    const float y = p->getPosition().y;
    const float hh = 0.5f * height_;     // mitad
    const float top = y + hh;          // parte superior
    const float bottom = y - hh;         // parte inferior

    // Casos (inmersión parcial)
    float subRatio = 0.0f; // fracción sumergida [0,1]

    if (bottom >= h0_) {
        // totalmente por encima del agua -> 0
        subRatio = 0.0f;
    }
    else if (top <= h0_) {
        // totalmente sumergido -> 1
        subRatio = 1.0f;
    }
    else {
        // parcial: desde bottom hasta h0
        // fracción = (h0 - bottom) / height
        subRatio = (h0_ - bottom) / height_;
        if (subRatio < 0.0f) subRatio = 0.0f;
        if (subRatio > 1.0f) subRatio = 1.0f;
    }

    if (subRatio <= 0.0f) return;

    // Empuje hacia +Y: E = rho * g * V_sub
    const float V_sub = volume_ * subRatio;
    const float E = rho_ * g_mag() * V_sub;

    Vector3D F(0.0f, E, 0.0f);
    p->addForce(F);
}
```

4. Efectos incorporados en el proyecto:

a. Viento (aplicado como brisa muy tenue para facilitar la jugabilidad y como viento cambiante solo en X .
b. Gravedad (uniforme para todo el proyecto e invariable).
c. Torbellino (estático e invertido en mitad del teatro de juego).
d. Explosión de partículas de forma, masa, tamaño y color aleatorio (cuando un jugador impacta en otro)
e. Lanzamiento de proyectil desde una posición determinada.
f. Generadores de partículas con dispersión uniforme y gaussiana (nieve blanca y azul).
g. Generación de cuerpos con flotabilidad (boyas e icebergs) y muelles (los muelles meramente ejemplificados).
h. Sistema de sólidos-rígidos y registro de colisiones entre elementos estáticos (paredes y suelo) y elementos móviles (metralla de cubitos y bolitas cuando un jugador es impactado).
i. Partículas y cuerpos con diferentes parámetros (masa, volumen, forma, densidad e inercia)

5. Efectos o experimentos extra incluidos

Creo que no he incluido ningún apartado extra más allá de todo lo mencionado anteriormente, tal vez algo lo sea, pero de serlo no soy consciente de ello.