MEMORIA PROYECTO FINAL FÍSICA

ANTONIO POVEDANO ORTIZ

1.- Temática

En el juego controlaremos a una pelota que tendrá que recorrer un pequeño laberinto hasta llegar a la plataforma de victoria. Por el camino habrá tanto obstáculos, que dificultan su llegada, como ayudas.
A lo largo del recorrido por el laberinto la pelota se verá afectada por diferentes fuerzas que modifican o empeoran el movimiento de esta por el laberinto.
Además este último muestra de forma clara cuál es el destino final del jugador, así como diferentes elementos que sirven para mejorar la estética principal del juego.

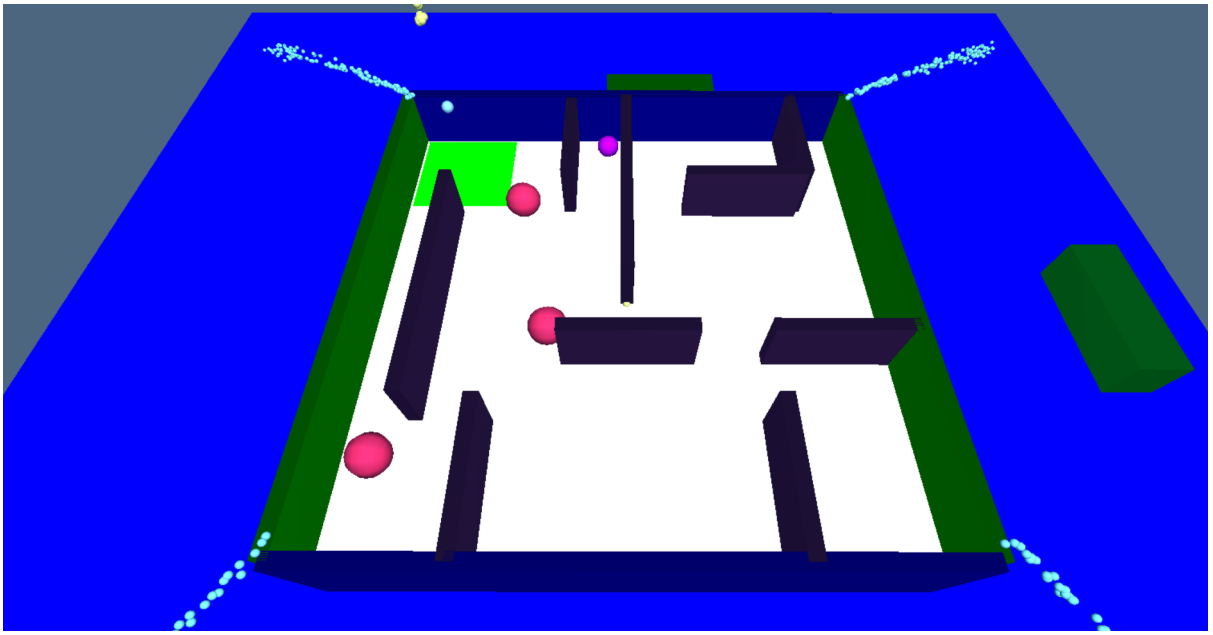
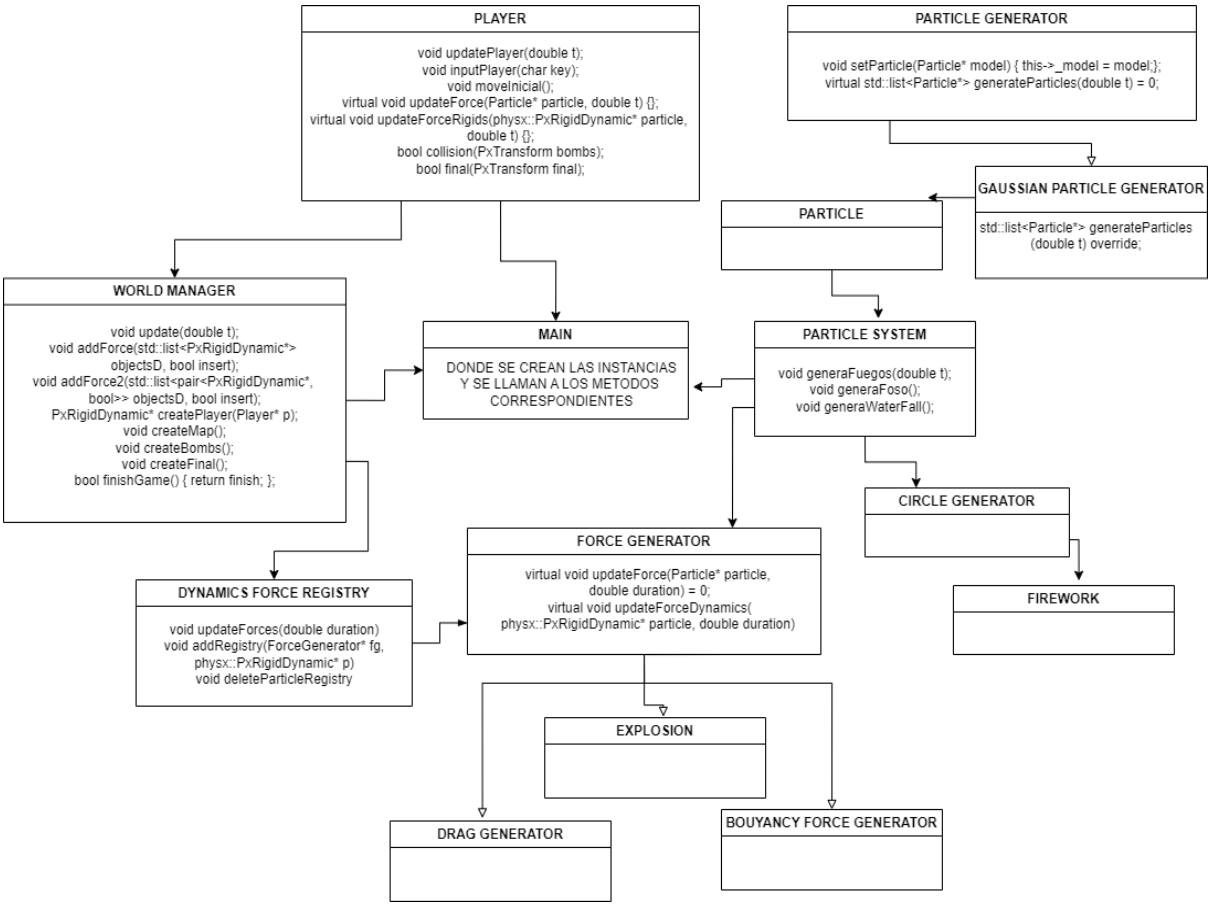


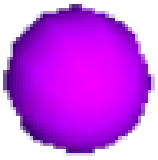
Imagen principal de una partida

2.- Diagrama de clases



3.- Efectos incorporados

- **Player**: el jugador está representado con una esfera (objeto dinámico) de color morado. Esta se moverá a partir de las teclas del teclado establecidas. El jugador tiene una clase propia en la que se encarga del movimiento de este, las colisiones con los demás objetos del mapa y comprobaciones como la de final de partida además de actualizar las fuerzas que actúan sobre este.



Al iniciar el juego el jugador está siendo influenciado por una fuerza (drag Generator), que moverá al jugador. La fórmula utilizada para mover al jugador es:

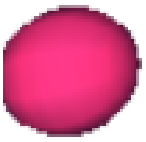
```
void DragGenerator::updateForceDynamics(physx::PxRigidDynamic* rigid, double t)
{
    if (!active)
        return;

    Vector3 v = _wind;
    float drag_coef = v.normalize();
    Vector3 dragF;
    drag_coef = (_k1 * drag_coef) + _k2 * drag_coef * drag_coef;
    dragF = -v * drag_coef;
    rigid->addForce(dragF);
}
```

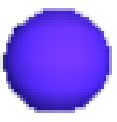
Drag Generator para los objetos dinámicos

- **Mapa:** los límites del laberinto están hechos con objetos estáticos. Estos solo sirven para evitar que el jugador se caiga del laberinto en cualquier momento (imagen principal de la partida).

- **Enemigos**: están representados por objetos dinámicos de color rojo. Si el jugador colisiona con alguno de estos será transportado automáticamente a la posición inicial teniendo este que repetir todo el proceso.



- **Ayuda**: están representados por objetos dinámicos de color azul. Si el jugador colisiona con este la fuerza que estaba influenciando al jugador (drag) se desactiva. Esto se realiza a través del booleano active de esta misma fuerza (imagen superior). Al colisionar esta se auto destruye, para que el jugador no pueda utilizarla de nuevo.



- **Colisiones:** las colisiones para el jugador con los objetos explicados anteriormente se realizan en el método collision del jugador.

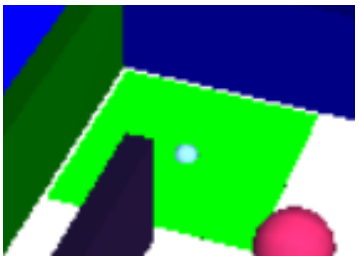
```
bool Player::collision(PxTransform bombs)
{
    if (abs(bombs.p.x - player_->getGlobalPose().p.x) > sizeEnemies + sizePlayer
        || abs(bombs.p.y - player_->getGlobalPose().p.y) > sizeEnemies + sizePlayer ||
        abs(bombs.p.z - player_->getGlobalPose().p.z) > sizeEnemies + sizePlayer)
        return false;

    float dx = bombs.p.x - player_->getGlobalPose().p.x;
    float dy = bombs.p.y - player_->getGlobalPose().p.y;
    float dz = bombs.p.z - player_->getGlobalPose().p.z;
    float distance = sqrt(dx * dx + dy * dy + dz * dz);

    return distance < (sizePlayer + sizeEnemies);
}
```

Método de la colisión en el player

- **Final de juego**: la plataforma para el final del juego esta representada por un plano de color verde.



Sobre esta habrá un generador de fuegos artificiales que se generan cada 2 segundos, para dar así más feedback visual al jugador. Para crear estos fuegos artificiales se utiliza el método generaFuegos del Particle System:

```
void ParticleSystem::generaFuegos(double t)
{

    std::shared_ptr<CircleGenerator> gen1(new CircleGenerator(this, "fireWork1",
        fireworkP, 10, { 0, 10,0 }, { 0, 0, 0 }));


    currentTime += t;

    if (lSpawn + tSpwan <= currentTime)
    {
        lSpawn = currentTime;

        particles.push_back(new Firework({ 150,0,-150 }, { 0, 200 ,0 }, 5,
            { 0, -2, 0 }, 0.5, 1, { 0.6,0.9,1 }, { gen1 }));

    }

}
```

GeneraFuegos del Particle System

Este método es llamado en el update de main.

Cuando el jugador colisiona con la plataforma final se activa la explosión que afecta a todos los elementos que se encuentran en el laberinto. Para crear esta explosión se ha utilizado la siguiente fórmula:

```
void Explosion::updateForceDynamics(physx::PxRigidDynamic* rigid, double t)
{

    if (!active)
        return;

    const double euler = std::exp(1.0);
    auto pos = rigid->getGlobalPose().p;
    auto difX = pos.x -ax_;
    auto difY = pos.y - ay_;
    auto difZ = pos.z - az_;

    auto r2 = pow(difX, 2) + pow(difY, 2) + pow(difZ, 2);

    if (r2 > pow(R_, 2))
    {
        return;
    }

    auto x = (k_ / r2) * difX * pow(euler, (-t / w_));
    auto y = (k_ / r2) * difY * pow(euler, (-t / w_));
    auto z = (k_ / r2) * difZ * pow(euler, (-t / w_));

    Vector3 force(x, y, z);


    rigid->addForce(force);
}
```

Explosiones para los objetos dinámicos

- **Elementos estéticos**: para complementar el juego, el laberinto se encuentra situado encima de un lago con tres cocodrilos en este. Además en las esquinas del laberinto hay un chorro de agua que apunta hacia el lago.
  Para los cocodrilos se han utilizado partículas con forma rectangular de color verde, a las cuales se les aplica una fuerza de flotación, para simular que está nadando por el lago. El método para crearlos está en el ParticleSystem:

```
void ParticleSystem::generaFoso()
{
    Particle* p = new Particle({ -400 , 0, 0 }, { 0, 0, 0 }, 2, { 0, 0, 0 }, 0.99, 100, { 0, 1, 0 }, 2, {50, 20, 20});
    _particles.push_back(p);
    p->setMass(20);
    p->setVolumen(50 * 20 * 20);
    Particle* p1 = new Particle({ 0 , 0, 400 }, { 0, 0, 0 }, 2, { 0, 0, 0 }, 0.99, 100, { 0, 1, 0.2 }, 2, { 60, 20, 20 });
    _particles.push_back(p1);
    p->setMass(1);

    p1->setVolumen(60  * 20 * 20);
    Particle* p2 = new Particle({ 300 , 0, 40 }, { 0, 0, 0 }, 2, { 0, 0, 0 }, 0.99, 100, { 0, 1, 0 }, 2, { 30, 20, 60 });
    _particles.push_back(p2);
    p->setMass(50);

    p2->setVolumen(30 * 20 * 60);


    buoyancy = new BuoyancyForceGenerator(2, 10, 0.1);
    pfr->addRegistry(buoyancy, p);
    pfr->addRegistry(buoyancy, p1);
    pfr->addRegistry(buoyancy, p2);

    GravityForceGenerator* g = new GravityForceGenerator({ 0, -9.8, 0 });
    pfr->addRegistry(g, p);
    pfr->addRegistry(g, p1);
    pfr->addRegistry(g, p2);

}
```

GeneraFoso del ParticleSystem

Para la fuerza de flotación se ha utilizado la siguiente fórmula:

```
void BuoyancyForceGenerator::updateForce(Particle* particle, double t)
{
    float h = particle->pos.p.y;
    float h0 = _liquid_particle->pos.p.y;

    Vector3 f(0, 0, 0);
    float immersed = 0.0;

    if (h - h0 > _height * 0.5)
    {
        immersed = 0.0;
    }
    else if(h0 - h > _height * 0.5){
        immersed = 1.0;
    }
    else
    {
        immersed = (h0 - h) / _height + 0.5;
    }

    f.y = _liquid_density * particle->getVolume() * immersed * 9.8;

    particle->addForce(f);
}
```

Flotación para las partículas

Para los chorros de agua se ha utilizado un generador gaussiano.

```
void ParticleSystem::generaWaterFall()
{
    addParticleGenerator(Gau, { 125, 100, 170 }, { 60, -25, 60 }, 1);
    addParticleGenerator(Gau, { -200, 100, -170 }, { -60, -25, -60 }, 1);
    addParticleGenerator(Gau, { -200, 100, 170 }, { -60, -25, 60 }, 1);
    addParticleGenerator(Gau, { 125, 100, -170 }, { 60, -25, -60 }, 1);
}
```

GeneraWaterFall del ParticleSystem

```
std::list<Particle*> GaussianParticleGenerator::generateParticles(double t)
{
    _currentTime += t;
    std::list<Particle*> _particles = std::list<Particle*>();

    //Se crea o no
    std::normal_distribution<double> dol{ 0, 1 };


    if (_timeGen + _lastGen <= _currentTime)
    {
        _lastGen = _currentTime;

        std::random_device r;
        std::default_random_engine gnd(r());
        for (auto i = 0; i < _num_particles; i++)
        {
            if (dol(gnd) > 0.75)
            {
                _particles.push_front(_model->clone());
                _particles.front()->pos.p += Vector3(_devTip_pos.x + _d(gnd), _devTip_pos.y + _d(gnd), _devTip_pos.z + _d(gnd));
                _particles.front()->vel += Vector3(_devTip_vel.x + _d(gnd), _devTip_vel.y + _d(gnd), _devTip_vel.z + _d(gnd));
            }
        }

        return _particles;
    }
    return _particles;
}
```

GenerateParticles del generador gaussiano

4.- Controles

Los controles usados son únicamente destinados para el movimiento del jugador. Como se ha explicado antes estos están recogidos en el método input Player del jugador.
Los controles son:

- A: desplazamiento hacia la izquierda.
- S: desplazamiento hacia abajo.
- D: desplazamiento hacia la derecha.
- W: desplazamiento hacia arriba.