Miguel Norberto Pérez Ballester

SIMULACIÓN FÍSICA PARA VIDEOJUEGOS

PROYECTO FINAL MIGUEL

- **Temática:**
  - El juego está inspirado el los típicos plataformas que debes recorrer un nivel hasta llegar a la meta.
  - El objetivo del juego es superar los obstáculos del nivel, sin caerse de la plataforma o tocar una superficie roja, ya que el jugador muere y debe volver a empezar desde el inicio, y llegar a la meta, que es una gran columna de color verde.
- **Movimiento del jugador**
  - El jugador puede moverse con las teclas WASD podiendo tomar 4 direcciones, mas las 4 diagonales. Las fuerzas se aplican tomando como vector inicial la dirección de apuntado de la cámara y según la tecla pulsada se modifica. Se puede ir hacia adelante o atrás y simultáneamente

```
Vector3 cam_dir = cam->getDir();
cam_dir.normalize();

Vector3 totalForce = Vector3(0, 0, 0);
// FORWARD BACKWARD MOVEMENT
if (key == 'W')
    totalForce += Vector3(cam_dir.x, 0, cam_dir.z);
else if (key == 'S')
    totalForce += Vector3(-cam_dir.x, 0, -cam_dir.z);

// LEFT RIGHT MOVEMENT
if (key == 'A')
    totalForce += Vector3(cam_dir.z, 0, -cam_dir.x);
else if (key == 'D')
    totalForce += Vector3(-cam_dir.z, 0, cam_dir.x);


if (!totalForce.isZero()) {
    // Normalizar para que siempre se mueva a la misma velocidad
    totalForce.normalize();
    player->getRigidBody()->addForce(totalForce * speed);
}
```

- **Sistema de escenas**
  - Utilizo un sistema de escenas que maneja la actualización de la escena que se esta usando en el momento. En caso de cambiar la escena, elimina a todos los sistemas de rigidbodys y partículas que se han añadido correctamente para no dejar ningún residuo.

```
virtual ~SceneRB() {
    for (auto sys : rb_systems) {
        sys->killAllRB();
    }
    rb_systems.clear();

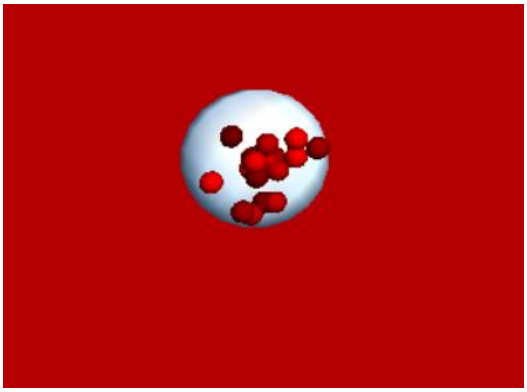
    for (auto sys : p_systems) {
        delete sys;
    }
    p_systems.clear();

    rb_generators.clear();
    p_generators.clear();
}
```

- Esto se puede ver con el cambio de escena entre el menú, el nivel y la escena de victoria.
- En caso de querer añadir más niveles, se podría hacer bastante fácil gracias a la herencia de la clase Scene_RB

• Animación de muerte

- Cuando el jugador toca la lava o cualquier obstáculo rojo, se activará un generador de partículas justo en la cabeza del player, spawneando partículas con dirección aleatoria X,Z y dirección vertical para la Y durante 3 segundos, luego se desactivará y eliminará todos las partículas restantes.
- Las partículas tienen un tiempo de vida de 0.5 segundos



• Física del viento

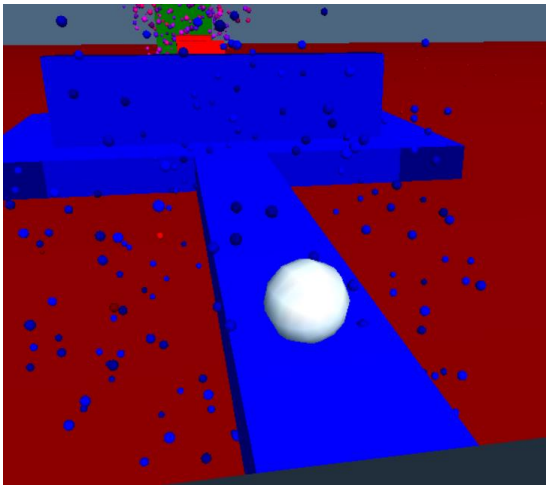
- El viento aplica la siguiente fórmula al sistema de rigidbodys que pertenece el jugador: (k1 = coef rozamiento aire, k2 = coeficiente de drag)
- Strength = 7 Radius = 10

```
void RB_WindGenerator::applyForce(std::shared_ptr<RB> rb) {

    if ((rb->getRigidBody()->getGlobalPose().p - tr->p).magnitude() < radius) {
        Vector3 windVelocity = direction * strength;
        float k1 = 0.5; // coef rozamiento aire
        float k2 = 0;
        Vector3 aux = rb->getRigidBody()->getLinearVelocity() - windVelocity;

        Vector3 windForce = k1 * aux + k2 * (aux.magnitude() * aux);
        rb->getRigidBody()->addForce(windForce);
    }
}
```

- El viento además tiene un sistema de partículas con un generador que posiciona partículas en el borde de la semiesfera imaginaria que se crea con el radio de acción del viento.
- Se mueven hacia la izquierda y su condición de eliminación es que la distancia con el centro sea mayor que el radio.
- Su color es aleatorio.



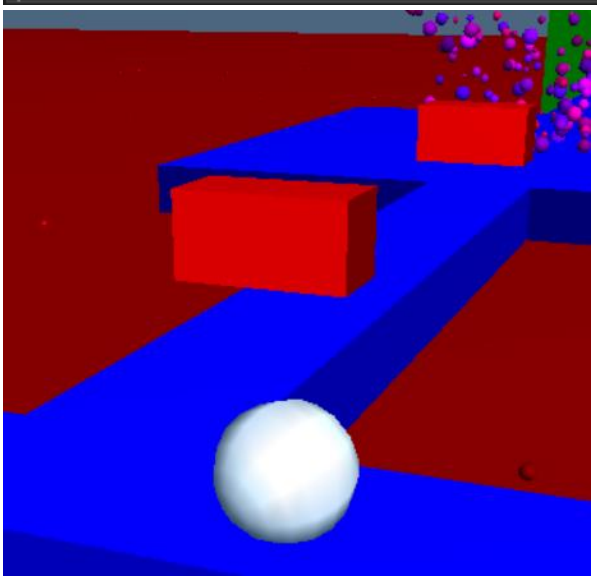
- **Física del muelle**
  - Hay dos cubos rojos que al tocarlos mueres. Estos cubos se mueven gracias a la fórmula del muelle, que los ata a un nodo imaginario a una altura superior a ellos. Elasticidad: 1.3 Longitud en descanso: 4

```
void RB_SpringAnchorGenerator::applyForce(std::shared_ptr<RB> rb) {
    Vector3 posRelativeOrigin = origin - rb->getRigidBody()->getGlobalPose().p;

    float length = posRelativeOrigin.normalize();
    float deltaX = length - restingLength;

    Vector3 springForce = posRelativeOrigin * deltaX * elasticity;

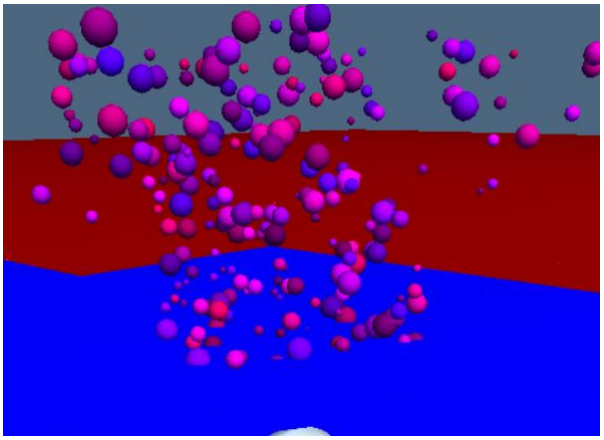
    rb->getRigidBody()->addForce(springForce);
}
```



- # Física del remolino
    - El remolino usa la siguiente fórmula para aplicar fuerza al player:
    - Height = 10 Strength = 6
    - ```
      void RB_WhirlWind::applyForce(std::shared_ptr<RB> rb) {

          if ((rb->getRigidBody()->getGlobalPose().p - origin).magnitudeSquared() < radius * radius)
          {
              Vector3 whirlwindVel = strength * Vector3(
                  -(rb->getRigidBody()->getGlobalPose().p.z - origin.z),
                  height - (rb->getRigidBody()->getGlobalPose().p.y - origin.y),
                  rb->getRigidBody()->getGlobalPose().p.x - origin.x);

              rb->getRigidBody()->addForce(whirlwindVel);
          }
      }
      ```
    - Además tiene un sistema de partículas que utiliza dos generadores de fuerzas, uno de gravedad y otro de remolino que aplican al sistema de partículas.

- # Manual de usuario
    - El objetivo es tocar el pilar verde
    - Si tocas cualquier plataforma de color rojo mueres y empiezas desde el spawn
    - -W ir hacia adelante
      -S ir hacia atrás
      -A ir hacia la izquierda
      -D ir hacia la derecha
      -Click izquierdo y arrastrar: mover la cámara
- # Extras
    - He tenido que modificar algunos métodos del render para poder mostrar texto.
      ```

- He tenido que modificar algunos métodos de la cámara para que siga al transform del jugador desde una inclinación y una distancia deseadas.

```
void Camera::updateFollow(const physx::PxTransform& tr) {
    // Actualizar el transform del objeto que la cámara sigue
    lookingAtTransform = tr;

    // Calcular la nueva posición de la cámara basándonos en el radio y el vector de dirección actual
    PxVec3 lookingAt = tr.p; // Posición del objeto
    PxVec3 cameraToTarget = mEye - lookingAt; // Dirección actual desde la cámara al objeto

    // Asegurar que el vector esté normalizado para calcular la posición de la cámara
    if (cameraToTarget.magnitude() == 0) {
        // Si la cámara ya está en el centro del objeto, la movemos a una posición inicial
        cameraToTarget = PxVec3(0, 0, -1); // Mirando hacia adelante por defecto
    }
    cameraToTarget.normalize();

    // Actualizar la posición de la cámara para mantener el radio de órbita
    mEye = lookingAt + cameraToTarget * FOV;

    mEye.y = INCLINATION;

    // Actualizar la dirección de la cámara para que siga mirando al objeto
    mDir = (lookingAt - mEye).getNormalized();
}

void Camera::lookAt(const physx::PxTransform& tr) {
    // Ajustar la dirección hacia un objetivo específico
    mDir = tr.p - mEye;
}
```

- Sistema de colisión que utiliza los nombres dados a los rigidbodys para manejar colisiones

```
void Level1::onCollision(physx::PxActor* actor1, physx::PxActor* actor2) {
    if ((actor1->getName() == "player" && actor2->getName() == "death") ||
        (actor2->getName() == "player" && actor1->getName() == "death")) {
        kill = true;
        if (!start_kill_timer) {
            start_kill_timer = true;
            kill_timer_end = std::chrono::steady_clock::now() + std::chrono::milliseconds(KILL_ANIM_TIME);
            death_anim_generator->toggleActive();
        }
    }
    else if ((actor1->getName() == "player" && actor2->getName() == "goal") ||
        (actor2->getName() == "player" && actor1->getName() == "goal")) {
        goToWinScreen();
    }
}
```