

Realidad Virtual

Curso 2023/2024

Trabajo:

Entrenamiento de Rocket League



Autor:

Miguel Quiroga Campos

Índice

| | |
|---|---|
| Introducción..... | 3 |
| Descripción de los objetos del terreno de juego..... | 3 |
| Descripción de la clase que desarrolla la escena..... | 4 |
| Cálculo de la posición de la cámara..... | 4 |
| Cálculo del movimiento del balón y del coche..... | 5 |
| Detección de choques con las paredes y goles..... | 7 |
| Pruebas del funcionamiento..... | 9 |

Introducción

En el marco de la asignatura de Realidad Virtual, se ha desarrollado un proyecto inspirado en el popular videojuego Rocket League. El objetivo principal de este proyecto es crear un modo de juego de un único jugador que sirva como entrenamiento del juego original. En este modo, el jugador controla un coche con el objetivo de marcar goles en una portería, interactuando con un balón dentro de un campo delimitado por paredes.

El desarrollo de este juego se ha llevado a cabo utilizando la biblioteca OpenGL, lo que nos ha permitido implementar gráficos 3D y simular la física necesaria para recrear de manera realista las interacciones entre el coche, el balón y los elementos del campo de juego. La memoria que se presenta a continuación detalla los aspectos técnicos y de implementación del proyecto, abarcando desde la descripción de los objetos que conforman el escenario hasta las pruebas realizadas para verificar su correcto funcionamiento.

Descripción de los objetos del terreno de juego

El terreno de juego está compuesto por dos objetos clave: la portería y las paredes.

La portería (CGPorteria) se ha diseñado como una estructura rectangular que define el área en la que se debe introducir el balón para marcar un gol. Ha sido creada a partir de la clase que definía un rectángulo en las prácticas iniciales, con una ligera modificación, se le ha quitado una de sus caras, haciendo ver el interior del rectángulo, como si fuera una caja de zapatos sin la tapa. Posteriormente se le añadió la textura de portería contenida en los archivos auxiliares.

Las paredes del campo de juego, por su parte, son superficies planas verticales que delimitan el espacio de juego y evitan que el balón salga del área permitida. Se han diseñado como objetos de tipo CGGround, a los que se les han aplicado las traslaciones y rotaciones necesarias para que delimiten correctamente el terreno de juego. La textura escogida ha sido la de tipo cemento incluida en los archivos auxiliares.

Descripción de la clase que desarrolla la escena

La clase que desarrolla la escena se ocupa de inicializar y visualizar todos los objetos y elementos que componen el campo de juego, incluyendo la luz, las paredes, la portería, el suelo, el coche y el balón, junto a sus texturas y sombras correspondientes; redimensionando y aplicando las traslaciones y rotaciones necesarias a cada uno de ellos para que queden en su lugar correspondiente.

Cálculo de la posición de la cámara

La posición de la cámara se calcula dinámicamente para ofrecer una vista adecuada del terreno de juego. La cámara sigue al coche desde una perspectiva trasera fija que permite al jugador tener una visión óptima del coche y del resto del terreno de juego.

Para ello se reutiliza la clase cámara implementada en la práctica 11 de la asignatura. La posición y dirección de la cámara es ajustada adecuadamente en la función **CGModel::update()**, donde a su vez se llama a la siguiente función, que es la encargada de realizar el cálculo necesario:

```
void CGModel::colocarCamara() {  
    glm::vec3 position = scene->getCoche()->GetRealPosition();  
    glm::vec3 forward = scene->getCoche()->getForwardDirection();  
    glm::vec3 up = scene->getCoche()->getUpDirection();  
  
    glm::vec3 cameraPosition = position - forward * 3.0f - up * 60.0f;  
  
    camera->SetPosition(cameraPosition.x, (cameraPosition.y + 15.0f), (cameraPosition.z));  
    camera->SetDir(-up);  
}
```

La posición de la cámara (**cameraPosition**) es calculada restando de la posición del coche una cantidad proporcional a la dirección hacia adelante y a la dirección hacia arriba. Esto sitúa la cámara detrás y ligeramente por encima del coche, proporcionando una vista óptima del coche y su entorno.

Además, se establece la dirección de la cámara hacia abajo (**SetDir(-up)**), de modo que la cámara mire hacia el coche desde arriba.

Cálculo del movimiento del balón y del coche

El movimiento del balón y del coche se calcula utilizando principios como la aceleración, fricción y colisión con otros objetos, sólo que en un marco simplificado.

El movimiento del coche se lleva a cabo a través de la siguiente función, llamada también en la función **CGModel::update()**:

```
void CGObject::MoveFront()
{
    if (moveStep > 0.0f)
        moveStep -= 0.002f;
    else if (moveStep < 0.0f)
        moveStep += 0.002f;

    Pos = glm::vec3(0.0f, 0.0f, 0.0f);
    Pos += moveStep * Dir;
    Translate(Pos);
}
```

Los bloques if-else tienen como objetivo simular la fricción del balón con el suelo, frenando su velocidad; a su vez, el resto es únicamente calcular la nueva posición del coche, y moverlo.

Como se puede intuir, al pulsar las teclas arriba y abajo, el atributo modificado es el **moveStep**, y al pulsar las teclas derecha e izquierda, simplemente se efectúa una leve rotación al coche, siempre y cuando este tenga algo de velocidad (para evitar que gire estando parado).

```
case GLFW_KEY_UP:
    scene->getCoche()->SetMoveStep(scene->getCoche()->GetMoveStep() + 0.1f);
    break;
case GLFW_KEY_DOWN:
    scene->getCoche()->SetMoveStep(scene->getCoche()->GetMoveStep() - 0.1f);
    break;
case GLFW_KEY_LEFT:
    if (scene->getCoche()->GetMoveStep() >= 0.15f || scene->getCoche()->GetMoveStep() <= -0.15f) {
        scene->getCoche()->Rotate(4.0f, glm::vec3(0.0f, 0.0f, 1.0f));
    }
    break;
case GLFW_KEY_RIGHT:
    if (scene->getCoche()->GetMoveStep() >= 0.15f || scene->getCoche()->GetMoveStep() <= -0.15f) {
        scene->getCoche()->Rotate(-4.0f, glm::vec3(0.0f, 0.0f, 1.0f));
    }
    break;
}
```

El balón, por su parte, sigue una trayectoria influenciada por la gravedad, los impactos con el coche y las colisiones con las paredes del campo de juego.

De forma similar a la del coche, el movimiento del balón se lleva a cabo de la siguiente manera:

```
void CGFigure::MoveFront()
{
    if (moveStep > 0.0f)
        moveStep -= 0.001f;

    else if (moveStep < 0.0f)
        moveStep += 0.001f;

    if (this->GetRealPosition().y > 3.0f)
        upStep-=0.005f;
    else if (upStep < 0.0f)
        upStep = -upStep * 0.8;

    Pos = glm::vec3(0.0f, 0.0f, 0.0f);
    Pos += glm::vec3(moveStep * Dir.x, upStep, moveStep * Dir.z);
    Translate(Pos);
}
```

Con la diferencia de que, para el balón, hemos definido también una velocidad vertical (**upStep**), que nos ayudará a hacer que la trayectoria sea parabólica, de tal forma que este valor se vaya reduciendo, simulando la fuerza de la gravedad, hasta que en el momento de tocar el suelo, se invierta y se reduzca, para que vaya botando cada vez menos.

El **moveStep**, **upStep** y la dirección son determinados según el impacto recibido por el coche:

```
void CGModel::CarBallImpact()
{
    glm::vec3 posB = scene->getBalon()->GetRealPosition();
    glm::vec3 posC = scene->getCoche()->GetRealPosition();

    float dx = posB.x - posC.x;
    float dy = posB.y - posC.y;
    float dz = posB.z - posC.z;
    float distancia = sqrt(dx * dx + dy * dy + dz * dz);

    if (distancia <= 5.0f)
    {
        if (scene->getCoche()->GetMoveStep() < 0.0)
            scene->getBalon()->SetDirection(glm::vec3(-dx, 0.0f, -dz), scene->getBalon()->GetUpDirection());
        else
            scene->getBalon()->SetDirection(glm::vec3(dx, 0.0f, dz), scene->getBalon()->GetUpDirection());

        scene->getBalon()->SetMoveStep(scene->getCoche()->GetMoveStep() * 0.5f);
        scene->getBalon()->SetUpStep(scene->getCoche()->GetMoveStep() * 0.4f);
    }
}
```

Donde la dirección que tomará el balón se calcula a través de un vector, como la diferencia de las posiciones de ambos objetos. Las velocidades (**moveStep** y **upStep**) vienen definidas por un factor de reducción respecto a la del coche.

Cabe recalcar que todos los factores de reducción o aumento definidos previamente están determinados manualmente, a través de numerosas pruebas, con el objetivo de que la experiencia de usuario resulte lo más realista posible.

Detección de choques con las paredes y goles

La detección de choques con las paredes se realiza de una forma muy sencilla, obteniendo la posición del objeto en cuestión (coche o balón) y comprobando si se encuentra en los límites del terreno de juego (se permite situarse dentro de la portería), donde en caso afirmativo, se calcula la nueva trayectoria basada en la dirección de impacto, invirtiendo la componente correspondiente de la dirección del objeto.

Además, para ambos objetos, se aplica un coeficiente de amortiguación a la velocidad, que simula la energía cinética absorbida por la pared en el impacto.

```
void CGModel::CarConstraints()
{
    glm::vec3 pos = scene->getCoche()->GetRealPosition();
    int constraint = 0;
    if (pos.x > 93.0f) { constraint = 1; }
    if (pos.x < -93.0f) { constraint = 1; }
    if (pos.z > 143.0f) { constraint = 1; }
    if (pos.z < -143.0f) { constraint = 1; }

    if (pos.x >= -20.0f && pos.x <= 20.0f) {

        if (pos.z > -160.0f)
            constraint = 0;
        else
            constraint = 1;
    }

    if (constraint == 1 && fueraCoche == false)
    {
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->GetMoveStep() * 0.4));
        fueraCoche = true;
    }
    else if (constraint == 0)
        fueraCoche = false;
}
```

```

void CGModel::BallConstraints()
{
    glm::vec3 pos = scene->getBalon()->GetRealPosition();
    int constraint = 0;

    glm::vec3 dir = scene->getBalon()->GetDirection();

    //if (pos.y < 1.0f) { pos.y = 1.0f; constraint = 1; }
    //if (pos.y > 40.0f) { pos.y = 40.0f; constraint = 1; }
    if (pos.x > 93.0f || pos.x < -93.0f) { constraint = 1; }
    if (pos.z > 143.0f || pos.z < -143.0f) { constraint = 2; }

    if ((pos.x >= -23.0f && pos.x <= 23.0f) && (pos.y >= 0.0f && pos.y <= 20.0f))
        constraint = 0;

    if (constraint == 1 && fueraBalon == false) {
        scene->getBalon()->SetDirection(glm::vec3(-dir.x, 0.0f, dir.z), scene->getBalon()->GetUpDirection());
        scene->getBalon()->SetMoveStep(scene->getBalon()->GetMoveStep() * 0.8f);
        fueraBalon = true;
    }
    else if (constraint == 2 && fueraBalon == false) {
        scene->getBalon()->SetDirection(glm::vec3(dir.x, 0.0f, -dir.z), scene->getBalon()->GetUpDirection());
        scene->getBalon()->SetMoveStep(scene->getBalon()->GetMoveStep() * 0.8f);
        fueraBalon = true;
    }
    else if (constraint == 0)
        fueraBalon = false;
}

```

Por otro lado, a la hora de detectar el gol, se realiza de forma similar:

```

void CGModel::GoalDetection()
{
    glm::vec3 pos = scene->getBalon()->GetRealPosition();

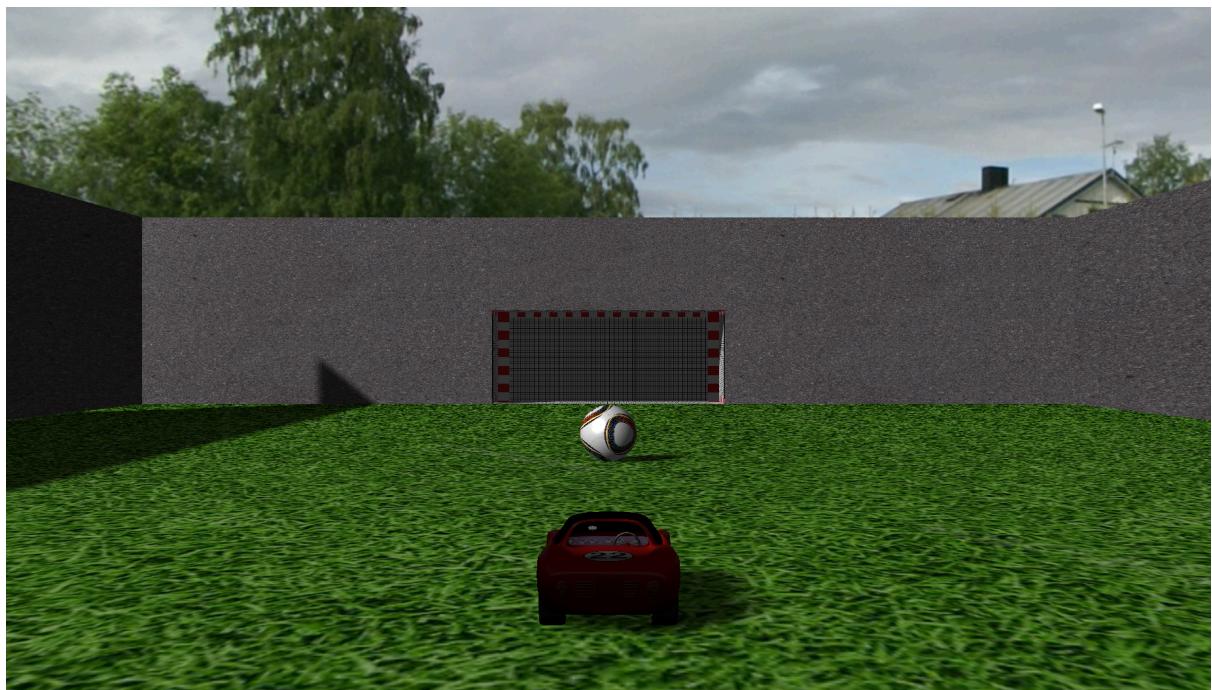
    if ((pos.x >= -23.0f && pos.x <= 23.0f) && (pos.y >= 0.0f && pos.y <= 20.0f) && (pos.z <= -160.0f))
        exit(0);
}

```

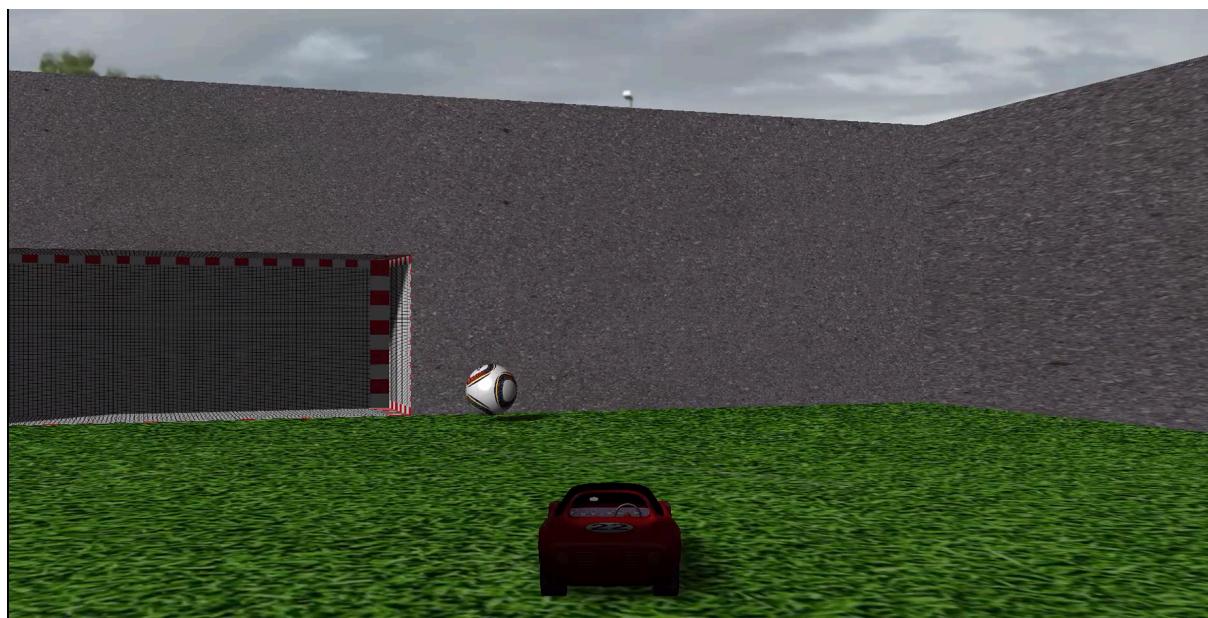
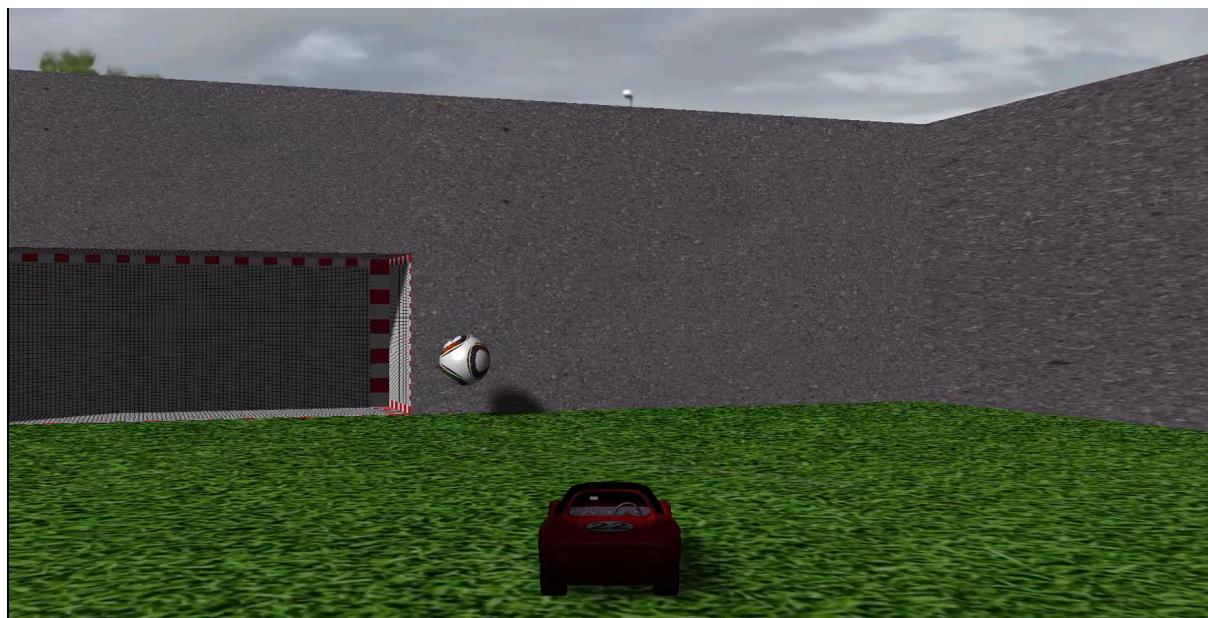
Sólo que, una vez se comprueba que el balón está dentro de la portería, el programa termina.

Pruebas del funcionamiento

Para verificar el correcto funcionamiento de la aplicación, se han realizado diversas pruebas que incluyen capturas de imágenes en diferentes situaciones del juego. Estas pruebas muestran el comportamiento del coche y del balón, la detección de colisiones y la actualización de la posición de la cámara, asegurando que todos los componentes interactúan de manera coherente y realista.









En resumen, este proyecto representa una implementación simplificada y funcional de un modo de entrenamiento inspirado en Rocket League, utilizando OpenGL para gestionar los gráficos y la física del juego, proporcionando una base sólida para el desarrollo de futuros modos de juego más complejos.