

Realidad Virtual



Trabajo final

Entrenamiento de Rocket League

Autor: Daniel Linfon Ye Liu

Curso 2023/2024

Índice

Índice.....	2
Introducción.....	3
Elementos que componen el juego.....	3
Suelo.....	3
Portería.....	3
Paredes.....	4
Coche.....	5
Balón.....	6
Skybox.....	6
Clase Scene.....	7
Posición de la cámara.....	8
Movimientos del coche y el balón.....	9
Coche.....	9
Balón.....	9
Impactos del coche al balón.....	10
Detección de los choques con las paredes y gol.....	11
Coche.....	11
Balón.....	12
Pruebas del funcionamiento.....	13
Referencias.....	15
Texturas.....	15
Figuras.....	15

Introducción

El objetivo de este proyecto es desarrollar manualmente una versión simplificada y en solitario inspirada en el popular videojuego Rocket League. La meta es crear un modo de juego para un solo jugador, en el cual el objetivo principal es marcar goles con un único coche, funcionando como un entrenamiento del juego original.

Elementos que componen el juego

En este proyecto, he implementado una serie de texturas para cada elemento del entorno, todas ellas relacionadas con la temática de invierno. Cada objeto y superficie ha sido texturizado para reflejar este tema, asegurando una experiencia visual que transporta al jugador a un mundo gélido y helado.

Suelo

Se ha utilizado la clase **CGGround** para crear y gestionar el suelo del entorno del juego. Para nuestra escena, las medidas del terreno de juego son 100.0f de largo y 40.0f de ancho. En este caso, al ser la base de nuestro entorno, no ha sido necesario aplicar ningún tipo de rotación o traslado. Se le ha aplicado una textura de hielo que refleja la temática:



Textura 1: Suelo

Portería

La clase **CGPorteria** ha sido creada para diseñar una portería en forma de prisma, con una de sus caras laterales eliminada para permitir el paso del balón. Al instanciar esta clase, se deben proporcionar dos parámetros: la altura de la portería y la profundidad de la misma.

```
CGPorteria::CGPorteria(GLfloat a, GLfloat p)
```

Para asegurar que las caras laterales tengan forma de rectángulo, la longitud de la base se define como cuatro veces la altura de la portería. Esto garantiza que las proporciones de la portería sean consistentes y que la geometría del objeto sea adecuada para su propósito en el juego. En nuestra escena, las medidas de la portería son: $a = 8.0f$, $b = 32.0f$, $p = 10.0f$.

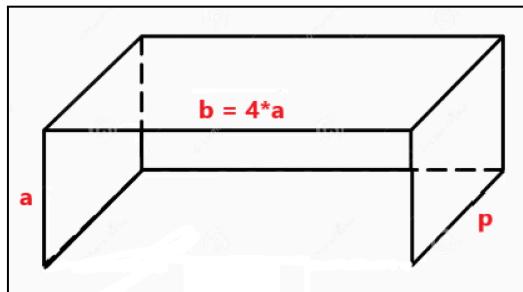


Figura 1: Estructura de la portería.



Textura 2: Portería.

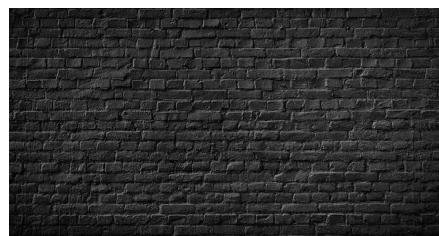
Paredes

De nuevo, se ha utilizado la clase **CGGround** para diseñar las paredes que delimitan el terreno de juego. Para lograr esto, se han aplicado rotaciones de 90° para orientar las paredes de forma vertical y se han realizado los traslados correspondientes para posicionar cada pared en su lugar adecuado.

En nuestra escena, hemos utilizado un total de 6 objetos de la clase **CGGround**. Tres de estos objetos se han empleado para cubrir los tres lados del campo, mientras que los tres restantes se han utilizado para el área de la portería. Esta distribución se debe a que la portería está incrustada en la pared, por lo que hemos tenido que colocar las paredes alrededor de ella para delimitar correctamente el área de juego. La altura de las paredes es de $15.0f$.



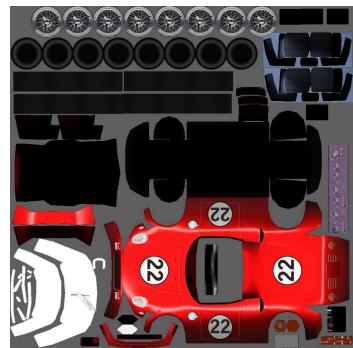
Figura 2: Paredes de la portería.



Textura 3: Pared.

Coche

Para el coche hemos generado las clases **CGSlotCar** y **CGSlotCar_pieces** a través de el **Ejecutar.bat** y el **ObjParser.jar** proporcionados en clase.



Textura 4: Coche.

Balón

Para crear el balón hemos utilizado la clase **CGSphere**, proporcionada en las prácticas de la asignatura. Se ha configurado el tamaño del balón para que sea ligeramente mayor que el del coche.



Textura 5: Balón.

Skybox

El Skybox consiste en renderizar un cubo alrededor de toda la escena del juego y aplicar texturas panorámicas a sus seis caras interiores. Esto nos sirve para que el jugador perciba que está inmerso en un espacio tridimensional con un horizonte lejano y un cielo expansivo que se extiende hasta el infinito.

En nuestra escena, apenas podemos contemplar el cielo ya que las paredes nos impiden ver el resto de texturas.

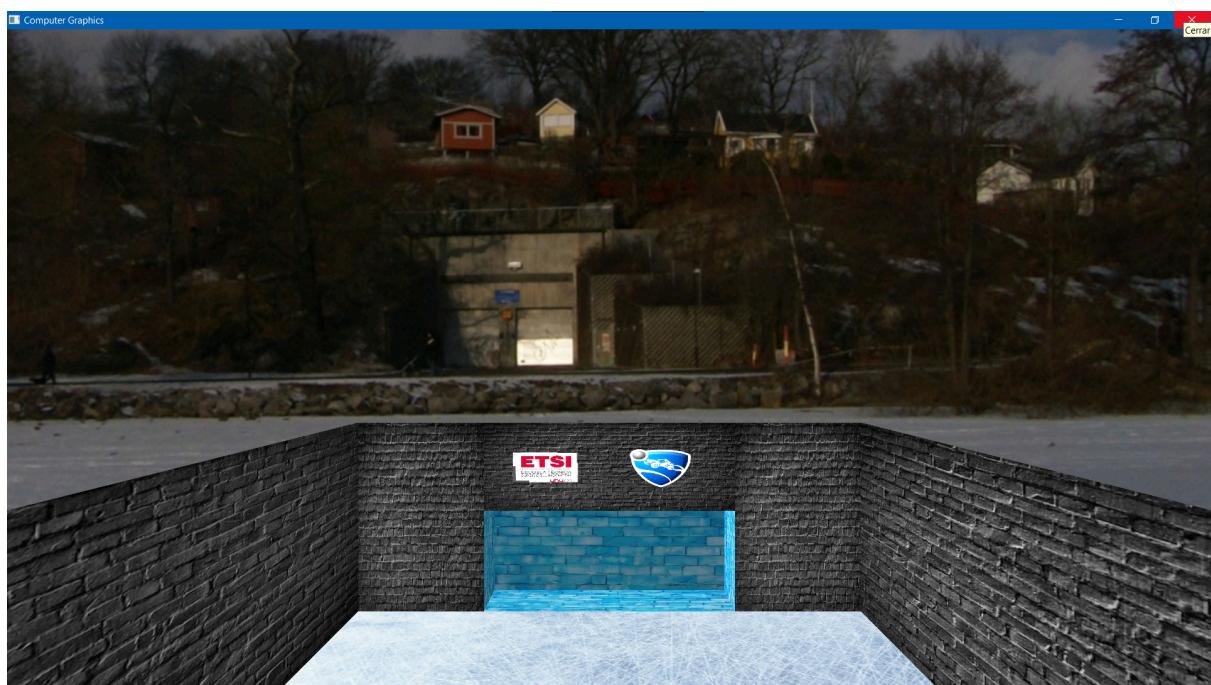


Figura 3: Vistas del Skybox en la portería.

Clase Scene

El constructor inicializa todos los elementos de la escena, incluyendo la luz, los materiales y los objetos mencionados previamente. Para cada elemento, se define su tamaño y se asigna el material correspondiente. Además, se aplican las transformaciones necesarias, como traslaciones y rotaciones, para posicionar correctamente cada objeto en la escena.

```
//Suelo
ground = new CGGround(40.0f, 100.0f);
ground->SetMaterial(matg);

//Pared 1
pared1 = new CGGround(15.0f, 100.0f);
pared1->Translate(glm::vec3(-40.0f, 15.0f, 0.0f));
pared1->Rotate(-90.0f, glm::vec3(0.0f, 0.0f, 1.0f));
pared1->SetMaterial(matParedes);

//Pared 2
pared2 = new CGGround(15.0f, 100.0f);
pared2->Translate(glm::vec3(40.0f, 15.0f, 0.0f));
pared2->Rotate(90.0f, glm::vec3(0.0f, 0.0f, 1.0f));
pared2->SetMaterial(matParedes);

//Pared 3
pared3 = new CGGround(40.0f, 15.0f);
pared3->Translate(glm::vec3(0.0f, 15.0f, 100.0f));
pared3->Rotate(-90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
pared3->SetMaterial(matParedes2);

//Porteria
porteria = new CGPorteria(8.0f, 10.0f);
porteria->Translate(glm::vec3(12.0f, 8.0f, -110.0f));
porteria->Rotate(180.0f, glm::vec3(0.0f, 1.0f, 0.0f));
porteria->SetMaterial(matPorteria);
```

```
//Paredes Porteria
paredIzq = new CGGround(10.0f, 15.0f);
paredIzq->Translate(glm::vec3(-30.0f, 15.0f, -100.0f));
paredIzq->Rotate(90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
paredIzq->SetMaterial(matParedes2);

paredDer = new CGGround(10.0f, 15.0f);
paredDer->Translate(glm::vec3(30.0f, 15.0f, -100.0f));
paredDer->Rotate(90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
paredDer->SetMaterial(matParedes2);

paredMid = new CGGround(20.0f, 7.0f);
paredMid->Translate(glm::vec3(0.0f, 23.0f, -100.0f));
paredMid->Rotate(90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
paredMid->SetMaterial(matParedMid);

//Coche
figCoche = new CGSlotCar();
figCoche->Rotate(-90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
figCoche->Translate(glm::vec3(0.0f, -50.0f, 0.0f));

//Pelota
figPelota = new CGSphere(20, 40, 2.3f);
figPelota->SetMaterial(matPelota);
figPelota->Translate(glm::vec3(0.0f, 2.3f, 0.0f));
```

Una vez creados todos los objetos, sólo nos falta dibujarlo en la escena y dibujar sus sombras.

```
void CGScene::Draw(CGShaderProgram* program, glm::mat4 proj, glm::mat4 view, glm::mat4 shadowViewMatrix)
{
    light->SetUniforms(program);

    ground->Draw(program, proj, view, shadowViewMatrix);
    pared1->Draw(program, proj, view, shadowViewMatrix);
    pared2->Draw(program, proj, view, shadowViewMatrix);
    pared3->Draw(program, proj, view, shadowViewMatrix);
    paredIzq->Draw(program, proj, view, shadowViewMatrix);
    paredDer->Draw(program, proj, view, shadowViewMatrix);
    paredMid->Draw(program, proj, view, shadowViewMatrix);
    porteria->Draw(program, proj, view, shadowViewMatrix);
    figPelota->Draw(program, proj, view, shadowViewMatrix);
    figCoche->Draw(program, proj, view);
}
```

```
void CGScene::DrawShadow(CGShaderProgram* program, glm::mat4 shadowMatrix)
{
    figPelota->DrawShadow(program, shadowMatrix);
    porteria->DrawShadow(program, shadowMatrix);
    pared1->DrawShadow(program, shadowMatrix);
    pared2->DrawShadow(program, shadowMatrix);
    pared3->DrawShadow(program, shadowMatrix);
    paredIzq->DrawShadow(program, shadowMatrix);
    paredDer->DrawShadow(program, shadowMatrix);
    paredMid->DrawShadow(program, shadowMatrix);
}
```

La configuración de la luz es la siguiente:

```
glm::vec3 Ldir = glm::vec3(1.0f, -0.8f, -1.0f);
Ldir = glm::normalize(Ldir);
light = new CGLight();
light->SetLightDirection(Ldir);
light->SetAmbientLight(glm::vec3(0.2f, 0.2f, 0.2f));
light->SetDifusseLight(glm::vec3(0.8f, 0.8f, 0.8f));
light->SetSpecularLight(glm::vec3(1.0f, 1.0f, 1.0f));
```

Estos pasos aseguran que la fuente de luz en la escena tenga la dirección correcta y las propiedades adecuadas para simular la iluminación ambiental, difusa y especular, lo cual contribuye a un entorno visualmente realista.

Posición de la cámara

Para posicionar la cámara detrás del coche en la escena, hemos creado la función `CGModel::colocarCamara()`:

```
void CGModel::colocarCamara()
{
    glm::vec3 posicionReal = scene->getCoche()->getRealPosition();
    glm::vec3 forward = scene->getCoche()->getForwardDirection();
    glm::vec3 up = scene->getCoche()->getUpDirection();

    glm::vec3 cameraPosition = posicionReal + forward * -25.0f + up * 5.0f;

    camera->SetPosition(cameraPosition.x, cameraPosition.y, cameraPosition.z);
    camera->SetDirection(-forward);
}
```

Para colocar la cámara detrás del coche, debemos obtener la posición real del vehículo y disminuir el valor de su eje Z. Además, tenemos que incrementar el valor de su eje Y para que se sitúe ligeramente por encima del coche. Obtenemos la fórmula:

$$cameraPosition = posicionReal + forward * -distanciaZ + up * altura$$

- **posicionReal**: Obtiene la posición actual del coche en el espacio 3D.
- **forward**: Obtiene la dirección hacia adelante del coche (el eje Z del coche).
- **up**: Obtiene la dirección hacia arriba del coche (el eje Y del coche).

Una vez calculada la posición de la cámara, establecemos la posición de esta en las coordenadas calculadas y le indicamos la dirección opuesta a "forward", haciendo que mire hacia el coche. Esto asegura que la cámara esté mirando hacia la parte trasera del coche, siguiendo su movimiento.

Movimientos del coche y el balón

Coche

Para configurar el movimiento del coche, utilizamos una variable llamada `moveStep`, la cual incrementamos o disminuimos progresivamente para simular la aceleración y desaceleración. Esta variable también nos permite aplicar el efecto de rozamiento.

```
void CGObject::MoverHaciaDelante() {
    SetMoveStep(this->getMoveStep() + 0.2f);
}

void CGObject::MoverHaciaDetras() {
    SetMoveStep(this->getMoveStep() - 0.2f);
}

void CGObject::MoverHaciaIzq() {
    Rotate(2.0f, glm::vec3(0.0f, 0.0f, 1.0f));
}

void CGObject::MoverHaciaDer() {
    Rotate(-2.0f, glm::vec3(0.0f, 0.0f, 1.0f));
}
```

```
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);
}
```

Cuando `moveStep` es mayor que 0, significa que el coche se está moviendo hacia adelante; reducimos su valor gradualmente para que el coche acabe deteniéndose, simulando así el efecto de rozamiento. De manera similar, cuando `moveStep` es menor que 0, significa que el coche se está moviendo hacia atrás; también en este caso, reducimos su valor poco a poco para que eventualmente se detenga. Para controlar los giros del coche, simplemente aplicamos rotaciones en el eje Z.

Balón

Para el balón, aplicamos la misma teoría que para el coche pero con la adición una variable llamada `upStep`, el cual tiene la misma función que el `moveStep` pero para el eje Y.

```

void CGFigure::MoveFront()
{
    if (moveStep > 0.0f)
        moveStep -= 0.0008f;

    else if (moveStep < 0.0f)
        moveStep += 0.0008f;

    if (this->getRealPosition().y > 2.3f)
        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);
}

```

Esta variable nos permite regular el movimiento ascendente y descendente del balón. Cuando la altura de la pelota supera los 2.3f (la altura de su centro), lo interpretamos como que está en el aire. En este caso, disminuimos gradualmente el valor de **upStep** para que el balón descienda suavemente.

Para simular el rebote del balón al contactar con el suelo, necesitamos hacer que rebote con una velocidad ligeramente inferior a la velocidad de contacto, en nuestro caso, un 80%. Para ello, detectamos que debe rebotar cuando **upStep** es menor que 0, lo que indica que está a la altura del suelo. En ese momento, invertimos el valor de **upStep** (haciendo que sea positivo) y lo reducimos en un 20% para simular la pérdida de energía en el rebote.

Impactos del coche al balón

Para detectar la colisión entre un coche y una pelota, y ajustar la dirección y velocidad de la pelota en consecuencia, hemos implementado la siguiente función:

```

void CGModel::impactoCochePelota() {
    glm::vec3 coche = scene->getCoche()->getRealPosition();
    glm::vec3 pelota = scene->getPelota()->getRealPosition();

    glm::vec3 diff = pelota - coche;

    // Calcular la distancia en el plano XZ
    float distanciaXZ = sqrt(pow(diff.x, 2) + pow(diff.z, 2));

    // Verificar si la distancia es menor o igual al umbral de impacto
    if (distanciaXZ <= 5.0f) {

        glm::vec3 direccionImpacto = glm::vec3(diff.x, 0.0f, diff.z);

        // Marcha atrás
        if (scene->getCoche()->getMoveStep() < 0)
            direccionImpacto = glm::vec3(-diff.x, 0.0f, -diff.z);

        scene->getPelota()->SetDirection(direccionImpacto);

        scene->getPelota()->SetMoveStep(scene->getCoche()->getMoveStep() * 0.5f);
        scene->getPelota()->setUpStep(0.2f);
    }
}

```

En primer lugar, obtenemos las posiciones actuales del coche y de la pelota en el espacio 3D y calculamos la diferencia de posición entre ellas. Luego se calcula la distancia entre la pelota y el coche en el plano XZ (horizontal). Esto se hace utilizando el teorema de Pitágoras en las componentes x y z de diff.

Una vez calculada la distancia entre el coche y la pelota, establecemos un umbral de 5.0f, que aproximadamente es la distancia que hay entre los centros de los objetos, para detectar una colisión.

Si se detecta colisión, se calcula la dirección del impacto como un vector en el plano XZ (horizontal), ignorando la componente Y (vertical). Si el coche golpea el balón marcha atrás (**moveStep** es negativo), se invierte la dirección del impacto.

Finalmente, ajustamos la velocidad de la pelota a la mitad de la velocidad del coche en el momento del impacto y establecemos un pequeño valor para **upStep** para simular un ligero salto vertical de la pelota tras el impacto.

Detección de los choques con las paredes y gol

Coche

Para implementar la detección de choques del coche con las paredes hemos implementado la siguiente función:

```
void CGModel::carConstraints() {
    glm::vec3 pos = scene->getCoche()->getRealPosition();
    int constraint = 0;
    if (pos.x > 35.0f || pos.x < -35.0f || pos.z > 95.0f || pos.z < -95.0f) {
        constraint = 1;
    }

    if (pos.z < -95.0f && pos.z > -112.0f && pos.x > -18.0f && pos.x < 18.0f) {
        constraint = 0;
    }

    if (constraint == 1 && fueraCoche == false) {
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->getMovestep() * 0.2));
        fueraCoche = true;
    }
    else if (constraint == 0) {
        fueraCoche = false;
    }
}
```

La función controla si el coche está dentro de los límites del campo o dentro del área de la portería (**constraint = 0**). Si el coche sale de los límites permitidos (**constraint = 1**), su velocidad se reduce a una quinta parte de su velocidad actual y su dirección se invierte. La variable **fueracoche** se utiliza para asegurarse de que la velocidad y la dirección solo se ajusten una vez cuando el coche cruza los límites.

Balón

Para implementar la detección de choques del balón con las paredes hemos implementado la siguiente función:

```
void CGModel::BallConstraints() {
    glm::vec3 pos = scene->getPelota()->getRealPosition();
    glm::vec3 dir = scene->getPelota()->getDirection();

    int constraint = 0;
    if (pos.x > 35.0f || pos.x < -35.0f){
        constraint = 1;
    }else if (pos.z > 95.0f || pos.z < -95.0f) {
        constraint = 2;
    }

    //Gol
    if (pos.z < -95.0f && pos.z > -112.0f && pos.x > -18.0f && pos.x < 18.0f) {
        constraint = 0;
        if (pos.z < -102.0f) {
            exit(0);
        }
        //scene->getPelota()->ResetLocation(); //Gol
    }

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

La función es muy similar a la anterior salvo que ahora controlamos si la pelota sale de los límites del campo, específicamente en el eje X (`constraint = 1`) o Z (`constraint = 2`). Si se cumple, su dirección se invierte solo en su eje correspondiente y su velocidad se reduce un 20%.

Cuando la pelota está dentro del área de gol, se finaliza el programa.

Pruebas del funcionamiento



Escena inicial



Interior de la portería



Golpeo del balón



Choque contra paredes



Tiro a puerta

Referencias

Texturas

Textura 1: Suelo: <https://www.freepik.es/fotos-vectores-gratis/textura-hielo>

Textura 2: Portería:

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTSrGxTV1A_usucZDtfRZQh0ffkt5Qg91z_aA&s

Textura 3: Pared: https://cdn.pixabay.com/photo/2017/03/25/03/18/white-2172682_640.jpg

Textura 4: Coche: Archivos auxiliares.

Textura 5: Balón: Archivos auxiliares.

Figuras

Figura 1: Estructura de la portería.

Figura 2: Paredes de la portería.

Figura 3: Vistas del Skybox en la portería.