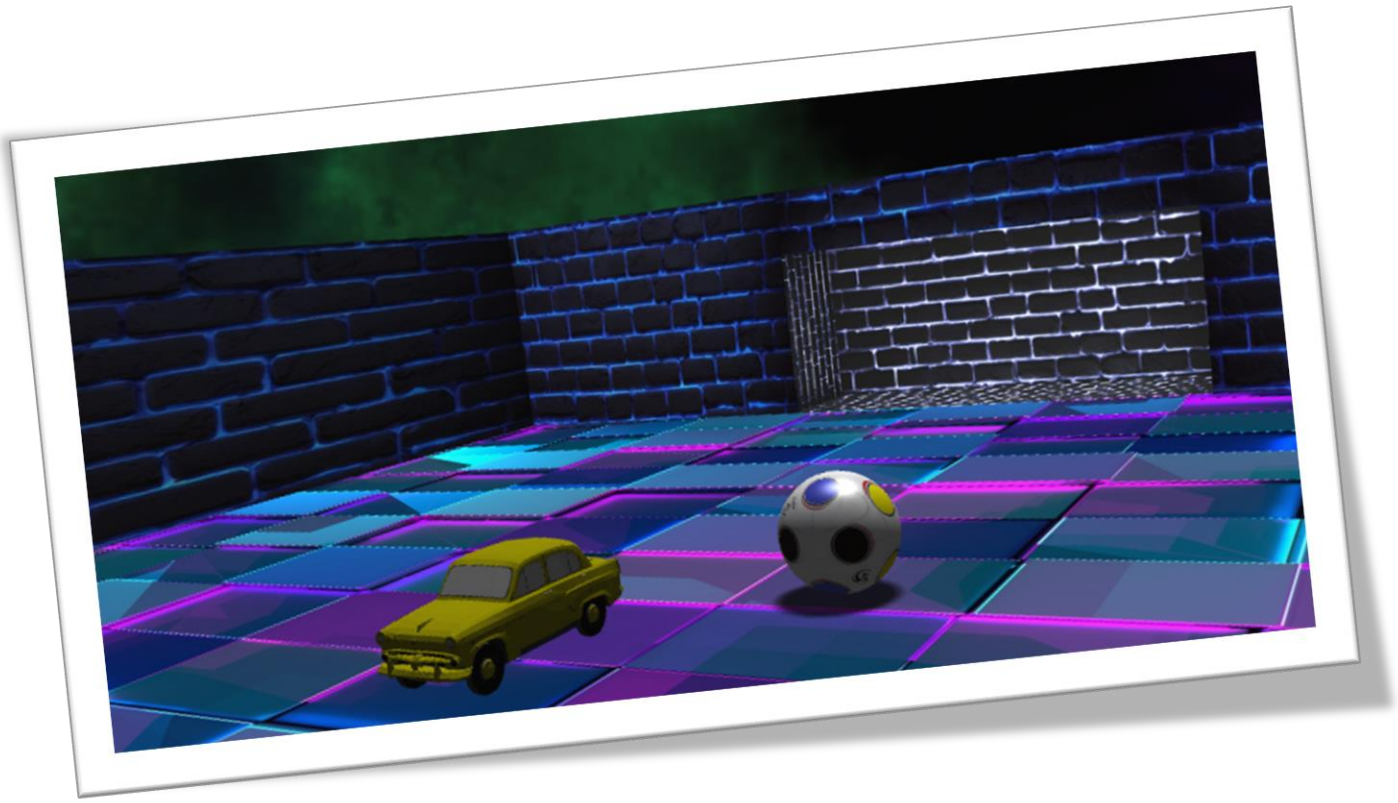
Trabajo de Realidad Virtual



Entrenamiento de Rocket League

Antón Maestre Gómez

Índice:

1 Introducción

El propósito de este proyecto es crear manualmente un juego inspirado en el conocido videojuego Rocket League. La idea es diseñar un modo de juego para un solo jugador, en el que el objetivo sea marcar goles utilizando un solo coche, funcionando como un entrenamiento del juego original.

2 Descripción de la portería y paredes

Para el desarrollo de la portería y las paredes que componen el terreno de juego, he empleado dos clases que heredan de una llamada **CGFigure**.

La clase CGFigure es una clase abstracta que representa un objeto para su renderizado mediante shaders, utilizando un Vertex Array Object (VAO).

Esta clase maneja los datos de los vértices, normales, índices y coordenadas de textura necesarios para representar un objeto en 3D.

```
GLushort* indexes; // Array of indexes
GLfloat* vertices; // Array of vertices
GLfloat* normals;  // Array of normals
GLfloat* textures; // Array of texture coordinates

GLuint numFaces; // Number of faces
GLuint numVertices; // Number of vertices
GLuint VBO[4];
GLuint VAO;

glm::mat4 location; // Model matrix
CGMaterial* material;
glm::vec3 Dir;
glm::vec3 Pos;
glm::vec3 Up;
glm::vec3 Right;

GLfloat moveStep;
GLfloat upStep;
GLfloat turnStep;
GLfloat cosAngle;
GLfloat sinAngle;
```

Además, cuenta con una matriz que representa el objeto en el espacio y una serie de variables que servirán para hacer funcionar el movimiento del objeto.

Utiliza VAO y VBO para optimizar el renderizado con shaders, y permite aplicar transformaciones como:

- **Translate:**

Esta función aplica una transformación de traslación a la matriz de modelo (location) del objeto.

```
void CGFigure::Translate(glm::vec3 t)
{
    location = glm::translate(location, t);
}
```

- **Rotate:**

Aplica una rotación a la matriz de modelo (location) del objeto en torno a un eje especificado.

```
void CGFigure::Rotate(GLfloat angle, glm::vec3 axis)
{
    location = glm::rotate(location, glm::radians(angle), axis);
}
```

Además, dispone de una función **Draw** que se encarga de dibujar el objeto en la escena principal. Para ello, calcula la matriz Model-View-Projection (MVP) combinando las matrices de proyección, vista y modelo (localización). Esta matriz transforma las coordenadas del modelo desde el espacio local al espacio de recorte. Seguidamente, realiza la configuración de los uniforms del Shader, configura el material y finaliza con el renderizado del objeto.

```
void CGFigure::Draw(CGShaderProgram* program, glm::mat4 projection, glm::mat4 view, glm::mat4 shadowViewMatrix)
{
    glm::mat4 mvp = projection * view * location;
    program->SetUniformMatrix4("MVP", mvp);
    program->SetUniformMatrix4("ViewMatrix", view);
    program->SetUniformMatrix4("ModelViewMatrix", view * location);
    program->SetUniformMatrix4("ShadowMatrix", shadowViewMatrix * location);
    material->SetUniforms(program);

    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, numFaces * 3, GL_UNSIGNED_SHORT, NULL);
}
```

La función **DrawShadow** genera la sombra correspondiente al objeto en la escena principal. Así pues, esta función configura las matrices de transformación necesarias para las sombras, establece los uniformes correspondientes en el shader program, y realiza la llamada a OpenGL para renderizar la sombra del objeto.

```
void CGFigure::DrawShadow(CGShaderProgram* program, glm::mat4 shadowMatrix)
{
    glm::mat4 mvp = shadowMatrix * location;
    program->SetUniformMatrix4("MVP", mvp);

    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, numFaces * 3, GL_UNSIGNED_SHORT, NULL);
}
```

Una vez definida esta clase abstracta, generamos dos clases que la implementan y que nos servirán para representar las paredes y la portería:

- **CGGround**:

Representa una superficie plana en el espacio tridimensional y tras definir su geometría mediante vértices, normales, coordenadas de textura e índices en base a unos parámetros, inicializa los buffers de OpenGL para que el objeto pueda ser renderizado correctamente en la escena gráfica (InitBuffers()).

Será la clase que nos sirva para generar tanto el suelo como las paredes del terreno de juego.

- **CGPorteria**:

Define una portería en un espacio tridimensional utilizando también vértices, normales, coordenadas de textura e índices para representar su geometría. Recibe dos parámetros, en base al primero se toman las medidas de ancho y alto; y en base al segundo se establece la profundidad de la portería. Finalmente, y al igual que el CGGround, inicializa los buffers.

Usaremos dicha clase para crear nuestra portería.

3 Descripción de la clase escena

Para la representación de los objetos que deseamos generar, emplearemos una clase denominada **CGScene**. Esta clase maneja la creación y manipulación de figuras, materiales, luces y objetos, y proporciona métodos para dibujar tanto la escena como sus sombras.

```
class CGScene {
public:
    CGScene();
    ~CGScene();
    CGObject* getCoche();
    CGFigure* getBall();
    void Draw(CGShaderProgram* program, glm::mat4 proj, glm::mat4 view, glm::mat4 shadowViewMatrix);
    void DrawShadow(CGShaderProgram* program, glm::mat4 shadowMatrix);
    glm::mat4 GetLightViewMatrix();

private:
    CGFigure* ground;
    CGFigure* wall1;
    CGFigure* wall2;
    CGFigure* wall3;
    CGFigure* wall4;
    CGFigure* wall5;
    CGFigure* wall6;

    CGFigure* fig0;
    CGFigure* fig4;

    CGMaterial* matb;
    CGMaterial* matg;
    CGMaterial* mat0;
    CGMaterial* mat1;
    CGMaterial* mat2;
    CGMaterial* mat3;
    CGMaterial* mat4;

    CGLight* light;
    CGObject* object;
};
```

En el constructor de la clase, se inicializa la escena de forma que:

- Creamos la iluminación, inicializando tanto la dirección de la luz como las configuraciones de las luces ambiental, especular y difusa.

```
// Iluminacion

glm::vec3 Ldir = glm::vec3(0.0f, -1.0f, 0.8f);
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));
```

- Cada material es creado y configurado con sus propiedades reflectantes y una textura asociada.

- Se crean y configuran los objetos de la escena, entre los que están las paredes, el suelo, la portería, el coche y el balón. El balón también hereda de la clase **CGFigure**, por lo que su creación y rasterización interna es similar a las paredes y portería. Por otra parte, el coche es un modelo importado que hereda de la clase **CGObject**, que se genera a partir de un conjunto de instancias de la clase **CGPiece**.

```
// Texturas

matg = new CGMaterial();
matg->SetAmbientReflect(1.0f, 1.0f, 1.0f);
matg->SetDifusseReflect(1.0f, 1.0f, 1.0f);
matg->SetSpecularReflect(0.8f, 0.8f, 0.8f);
matg->SetShininess(16.0f);
matg->InitTexture("textures/ground.jpg");
```

```
// Coche

object = new Moskvitch();
object->Rotate(180.0f, glm::vec3(0.0f, 1.0f, 0.0f));
object->Init();
```

El destructor se encarga de liberar la memoria de todos los objetos creados mediante el comando “delete”.

Las funciones Draw y DrawShadow se encargan de dibujar los objetos y sus correspondientes sombras en las escena, realizando llamadas a las funciones internas de cada objeto.

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

    ground->Draw(program, proj, view, shadowViewMatrix);
    wall1->Draw(program, proj, view, shadowViewMatrix);
    wall2->Draw(program, proj, view, shadowViewMatrix);
    wall3->Draw(program, proj, view, shadowViewMatrix);
    wall4->Draw(program, proj, view, shadowViewMatrix);
    wall5->Draw(program, proj, view, shadowViewMatrix);
    wall6->Draw(program, proj, view, shadowViewMatrix);
    fig0->Draw(program, proj, view, shadowViewMatrix);
    fig4->Draw(program, proj, view, shadowViewMatrix);
    object->Draw(program, proj, view);
}
```

```
void CGScene::DrawShadow(CGShaderProgram* program, glm::mat4 shadowMatrix)
{
    wall1->DrawShadow(program, shadowMatrix);
    wall2->DrawShadow(program, shadowMatrix);
    wall3->DrawShadow(program, shadowMatrix);
    wall4->DrawShadow(program, shadowMatrix);
    wall5->DrawShadow(program, shadowMatrix);
    wall6->DrawShadow(program, shadowMatrix);
    fig0->DrawShadow(program, shadowMatrix);
    fig4->DrawShadow(program, shadowMatrix);
    object->DrawShadow(program, shadowMatrix);

}
```

Finalmente, encontramos algunos get de interés como uno para devolver el objeto del balón, otro para el coche y otro para obtener la matriz de posicionamiento de la luz.

Esta última se utiliza en el sombreado, donde se necesita transformar las coordenadas del mundo al espacio de la luz para determinar si un punto está iluminado o en sombra.

```
glm::mat4 CGScene::GetLightViewMatrix()
{
    glm::vec3 Zdir = -(light->GetLightDirection());
    glm::vec3 Up = glm::vec3(0.0f, 1.0f, 0.0f);
    glm::vec3 Xdir = glm::normalize(glm::cross(Up, Zdir));
    glm::vec3 Ydir = glm::cross(Zdir, Xdir);
    glm::vec3 Zpos = 150.0f * Zdir;
    glm::vec3 Center = glm::vec3(0.0f, 0.0f, 0.0f);

    glm::mat4 view = glm::lookAt(Zpos, Center, Ydir);
    return view;
}
```

4 Descripción del cálculo de la posición de la cámara

Para la gestión y control de la cámara en mi proyecto, he implementado una clase específica denominada CGCamera. Esta clase está diseñada para ofrecer un control preciso sobre la posición y orientación de la cámara en la escena. Esto se logra mediante varios métodos y atributos que permiten establecer y obtener las coordenadas de posición y dirección sobre los ejes, además de generar la matriz de vista de la cámara, que es crucial para transformar las coordenadas del mundo al espacio de la cámara, permitiendo así una correcta creación de la escena y del skybox.

```
class CGCamera {
public:
    CGCamera();
    glm::mat4 ViewMatrix();

    void SetPosition(GLfloat x, GLfloat y, GLfloat z);
    void SetDirection(GLfloat xDir, GLfloat yDir, GLfloat zDir, GLfloat xUp, GLfloat yUp, GLfloat zUp);
    void SetD(glm::vec3 up);

    glm::vec3 GetPosition();
    glm::vec3 GetDirection();
    glm::vec3 GetUpDirection();

private:
    glm::vec3 Pos;
    glm::vec3 Dir;
    glm::vec3 Up;
    glm::vec3 Right;
};
```

La clase cámara es instanciada junto con la escena y el skybox en la clase **CGModel**, la cual se encarga de gestionar y controlar la representación gráfica: la inicialización, renderizado, actualización, y manejo de entradas del usuario.

En dicha clase, existe una función **update** que anima la escena actualizándola en cada momento. Es aquí donde llamaremos a la función **colocarCamara**.

```
void CGModel::colocarCamara() {
    glm::vec3 posR = scene->getCoche()->GetRealPosition();
    glm::vec3 forward = scene->getCoche()->getFWDirection();
    glm::vec3 up = scene->getCoche()->GetUpDirection();
    glm::vec3 right = scene->getCoche()->GetRightDirection();
    glm::vec3 camPos = posR - (forward * 75.0f) + (up * 15.0f);

    camera->SetPosition(camPos.x, camPos.y, camPos.z);

    camera->SetD(-forward);

}
```

Esta función se encarga de posicionar y orientar la cámara detrás del coche en cada momento. Su propósito es mantener una vista en tercera persona del coche, de modo que la cámara siempre siga al coche a una distancia fija y con una orientación específica.

La posición de la cámara se calcula restando a la posición del coche un vector proporcional a la dirección hacia adelante del coche (**forward * 75.0f**) y sumando un vector correspondiente a la dirección hacia arriba del coche (**up * 15.0f**). Esto coloca la cámara 75 unidades detrás y 15 unidades por encima del coche.

posicionCamara = posicionCoche - (forward * 75) + (up * 15)

Por otro lado, la inversión de la dirección **(-forward**) es crucial porque queremos que la cámara mire hacia el coche. Si forward apunta hacia donde el coche está mirando, entonces -forward apunta hacia el coche. Al establecer esta dirección como la orientación de la cámara, nos aseguramos de que la cámara esté siempre orientada hacia el coche.

direccionCámara = -forward

Al llamar a la función colocarCamara en la función update, garantizamos que la cámara se mantenga en la posición y dirección que queremos en todo momento.

5 Descripción de los movimientos y choque Coche-Balón

Para describir los movimientos del juego, debemos analizar el desplazamiento del coche y del balón, así como el desplazamiento del balón al colisionar con el coche:

- **Movimiento del coche:**

El movimiento hacia delante y hacia atrás del coche se basa en el aumento (movimiento hacia atrás) y disminución (movimiento hacia delante) de la variable moveStep. Cuanto mayor sea el valor de esta variable tanto positivo como negativo, mayor será la velocidad del coche:

```
void CGObject::MovAdelante() {
        moveStep -= 0.1f;
}
```

```
void CGObject::MovAtras() {
        moveStep += 0.1f;
}
```

El movimiento hacia la derecha y hacia la izquierda se fundamenta en el uso de una variable (**turnStep**) que indica el ángulo de giro con cada acción de rotación solicitada por teclado. El nuevo valor de la dirección se obtiene a partir del seno y coseno de dicho ángulo. Cabe resaltar que he establecido un limite de velocidad mínimo para realizar un giro y que así el juego se vea más realista.

```
void CGObject::MoveLeft()
{
    Dir.x = cosAngle * Dir.x + sinAngle * Right.x;
    Dir.y = cosAngle * Dir.y + sinAngle * Right.y;
    Dir.z = cosAngle * Dir.z + sinAngle * Right.z;
    Right = glm::cross(Up, Dir);

    this->Rotate(turnStep, glm::vec3(0.0f, 1.0f, 0.0f));
}
```

```
void CGObject::MoveRight()
{
    Dir.x = cosAngle * Dir.x - sinAngle * Right.x;
    Dir.y = cosAngle * Dir.y - sinAngle * Right.y;
    Dir.z = cosAngle * Dir.z - sinAngle * Right.z;
    Right = glm::cross(Up, Dir);

    this->Rotate(-turnStep, glm::vec3(0.0f, 1.0f, 0.0f));
}
```

Finalmente, definimos la función Actualizar, que será llamada en el update de la clase CGModel y que irá aplicando una fuerza de rozamiento (reduciendo la velocidad del coche) en cada momento. De esta forma, el coche será más realista al no tener un movimiento uniforme.

Después de aplicar este rozamiento, recalculamos la nueva posición del coche acorde a su moveStep y dirección:

Los movimientos de acelerar, decelerar, girar a la izquierda y girar a la derecha se ejecutarán al pulsar sobre las teclas correspondientes a las flechas de dirección del teclado.

```
void CGObject::Actualizar() {
    if (moveStep != 0.0f) {
        GLfloat rozamiento = 0.002f;
        if (moveStep > 0.0f) {
            if ((moveStep - rozamiento) >= 0.0f) {
                moveStep -= rozamiento;
            }

        }
        else if (moveStep < 0.0f){
            if ((moveStep - rozamiento) <= 0.0f) {
                moveStep += rozamiento;
            }

        }
    }
    Pos = glm::vec3(0.0f, 0.0f, 0.0f);
    Pos += moveStep * Dir;
    if (moveStep != 0.0f) {
        this->Translate(Pos);
    }

}
```

- **Movimiento del balón:**

La pelota contará también con la variable moveStep para definir su velocidad de movimiento sobre los ejes x-z. Además, contará con la variable upStep para establecer también su movimiento sobre el eje y (ya que el balón podrá botar y elevarse sobre el terreno). Para ello, definiremos la función moveFront en la clase abstracta CGFigure (la cual implementa).

Lo primero que realiza esta función es aplicar la fuerza de rozamiento sobre la variable moveStep al igual que hicimos con el coche. Después, actúa sobre el upStep haciendo que si la posición Y es mayor que 6.5 (el radio de la pelota), se reduce upStep en 0.005 unidades, simulando un efecto de caída (gravedad). Si upStep es negativa (la pelota se está moviendo hacia abajo), se invierte y se multiplica por 0.8, simulando un rebote con pérdida de energía.

Finalmente se actualiza la posición de la pelota en base al moveStep sobre los ejes x-z, y al upStep sobre el eje y. Esta función también será llamada en el update de CGModel.

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

    if (this->GetRealPosition().y > 6.5f)
        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);
}
```

- **Movimiento de colisión Coche-Balón:**

Con la idea de simular el golpeo de la pelota con el coche, se crea una función de la clase CGModel que se encargue de actualizar los nuevos valores de desplazamiento del balón en el momento del choque.

La función **BallCarConstraints** calcula en primer lugar tanto las diferencias de posición en los ejes X, Y y Z entre la pelota y el coche, como la distancia euclidiana en 3D entre la pelota y el coche usando la fórmula de la distancia euclídea.

$$d = \sqrt{((x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2)}$$

Si la distancia es menor o igual a 12 supondrá una colisión, por lo que, dependiendo de la orientación del coche, estableceremos la nueva dirección de la pelota:

- Si el coche va marcha atrás, la pelota deberá tener la dirección contraria al coche.
- Si el coche golpea de frente, la pelota tendrá la misma dirección que el coche.

Por último, se actualiza la nueva velocidad del balón (moveStep) con relación a la que lleve el coche, y se establece el movimiento vertical (upStep) del balón a partir de la velocidad del coche y de la velocidad del eje y que llevase ya la pelota.

```
void CGModel::BallCarConstraints() {
    glm::vec3 posBall = scene->getBall()->GetRealPosition();
    glm::vec3 posCar = scene->getCoche()->GetRealPosition();

    float disx = posBall.x - posCar.x;
    float disy = posBall.y - posCar.y;
    float disz = posBall.z - posCar.z;
    float disxyz = sqrt((disx * disx) + (disz * disz) + (disy * disy));

    if (disxyz <= 12.0f) {
        if(scene->getCoche()->GetMoveStep() < 0.0f) // si el coche marcha atras
            scene->getBall()->SetDirection(glm::vec3(-disx, -disy, -disz));
        else
            scene->getBall()->SetDirection(glm::vec3(disx, disy, disz));

        scene->getBall()->SetUpStep((scene->getCoche()->GetMoveStep() * 0.30f)
            + scene->getBall()->GetUpStep());
    }

}
```

6 Descripción del choque con las paredes y detección de gol

Para evitar que tanto el coche como el balón salgan de los límites del terreno de juego, se han diseñado dos funciones en el CGModel:

- **Choque del coche con las paredes**:

La función **CarConstraints** se encarga de establecer las físicas de choque del coche con las paredes dentro de la escena gráfica. Para ello, utiliza una variable constraint que indicará si el coche está colisionando con las paredes (se encuentra fuera de los parámetros definidos).

Por otra parte, usamos dos variables booleanas para ajustar la dirección de movimiento del coche cuando se detecta una colisión con las paredes de frente (fueraCoche) o marcha atrás (mAtras).

De esta forma, si detectamos un choque invertiremos la velocidad del coche con una cierta pérdida y activaremos la booleana correspondiente. Si por el contrario no se detecta ningún choque, simplemente reiniciaremos las variables booleanas fueraCoche y mAtras.

De esta forma conseguimos las físicas de colisión del coche con las paredes y portería del terreno de juego.

```
void CGModel::CarConstraints() {
    glm::vec3 pos = scene->getCoche()->GetRealPosition();
    int constraint = 0;
    if (fueraCoche && scene->getCoche()->GetMoveStep() > 0)
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->GetMoveStep()));

    if (mAtras && scene->getCoche()->GetMoveStep() < 0)
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->GetMoveStep()));

    if (pos.x > 110.0f || pos.x < -110.0f || pos.z > 150.0f) {
        constraint = 1;
    }
    if (pos.z < -150.0f && (pos.x > 44.0f || pos.x < -44.0f))
        constraint = 1;
    if (pos.z < -165.0f && (pos.x <= 44.0f || pos.x >= -44.0f))
        constraint = 1;

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

}
```

- **Choque de la pelota con las paredes y detección de gol**:

La función **BallConstraints** se encarga de aplicar las físicas de choque del balón con las paredes de la escena. De la misma forma que la función CarConstraints, esta función utilizará una variable constraint para indicar si el balón está colisionando con las paredes y dos variables booleanas para ajustar debidamente la dirección del balón cuando se produce la colisión.

```
void CGModel::BallConstraints() {
    glm::vec3 pos = scene->getBall()->GetRealPosition();
    int constraint = 0;
    bool px = false;
    bool pz = false;
    glm::vec3 dir = scene->getBall()->GetDirection();



    if (fueraBalon == true && scene->getBall()->GetMoveStep() > 0)
        scene->getBall()->SetMoveStep(-(scene->getBall()->GetMoveStep()));

    if (mAtrasB == true && scene->getBall()->GetMoveStep() < 0)
        scene->getBall()->SetMoveStep(-(scene->getBall()->GetMoveStep()));
```

Al comprobar si el balón se encuentra dentro de los límites definidos, analizamos si ha entrado en la portería. Si es así, el programa finaliza indicando que se ha marcado gol.

```
if (pos.x > (120.0f - 6.5f)) {
    constraint = 1;
    px = true;
}
if (pos.x < (6.5f - 120.0f)) {
    constraint = 1;
    px = true;
}
if (pos.z > (160.0f - 6.5f)) {
    constraint = 1;
    pz = true;
}
if (pos.z < (6.5f - 160.0f) && (pos.x > (-6.5f + 44.0f)
    || pos.x < (6.5f - 44.0f))) {
    constraint = 1;
    pz = true;
}
if (pos.z < (-160.0f) && ((pos.x <= (-6.5f + 44.0f)
    || pos.x >= (6.5f - 44.0f)))) {
    std::cout << "GOL";
    exit(0);
}
```

Si se detecta una colisión (constraint == 1), se analiza la orientación del balón al chocar. Dependiendo de esto y de las variables px y pz, se ajusta la dirección del balón para que rebote en la pared y se invierte la velocidad estableciendo una cierta pérdida.

Las variables pz y px nos ayudarán a saber si la pelota ha chocado con una pared del eje x o del eje z. De esta forma, invertiremos la coordenada de la dirección necesaria para que el movimiento tras el choque sea lo más realista posible.

Si por el contrario no se detecta ningún choque, simplemente reiniciaremos las variables booleanas fueraBalon y mAtrasB.

```
if (constraint == 1) {
    if (scene->getBall()->GetMoveStep() > 0 && !mAtrasB && !fueraBalon) {
        if (px == true) {
            scene->getBall()->SetDirection(glm::vec3(dir.x, dir.y, -dir.z));
        }
        else if (pz == true) {
            scene->getBall()->SetDirection(glm::vec3(-dir.x, dir.y, dir.z));
        }
        scene->getBall()->SetMoveStep(-(scene->getBall()->GetMoveStep() * 0.4f));
        fueraBalon = true;
    }
    else if (scene->getBall()->GetMoveStep() < 0 && !fueraBalon && !mAtrasB) {
        if (px == true) {
            scene->getBall()->SetDirection(glm::vec3(dir.x, dir.y, -dir.z));
        }
        else if (pz == true) {
            scene->getBall()->SetDirection(glm::vec3(-dir.x, dir.y, dir.z));
        }
        scene->getBall()->SetMoveStep(-(scene->getBall()->GetMoveStep() * 0.4f));
        mAtrasB = true;
    }
}
else if (constraint == 0) {
    fueraBalon = false;
    mAtrasB = false;
}
```

Estas dos funciones también son llamadas en el update del CGModel, para que las comprobaciones se realicen en todo momento y logremos asegurar tanto el correcto funcionamiento de las físicas del terreno de juego como la comprobación de gol.

7 Referencias

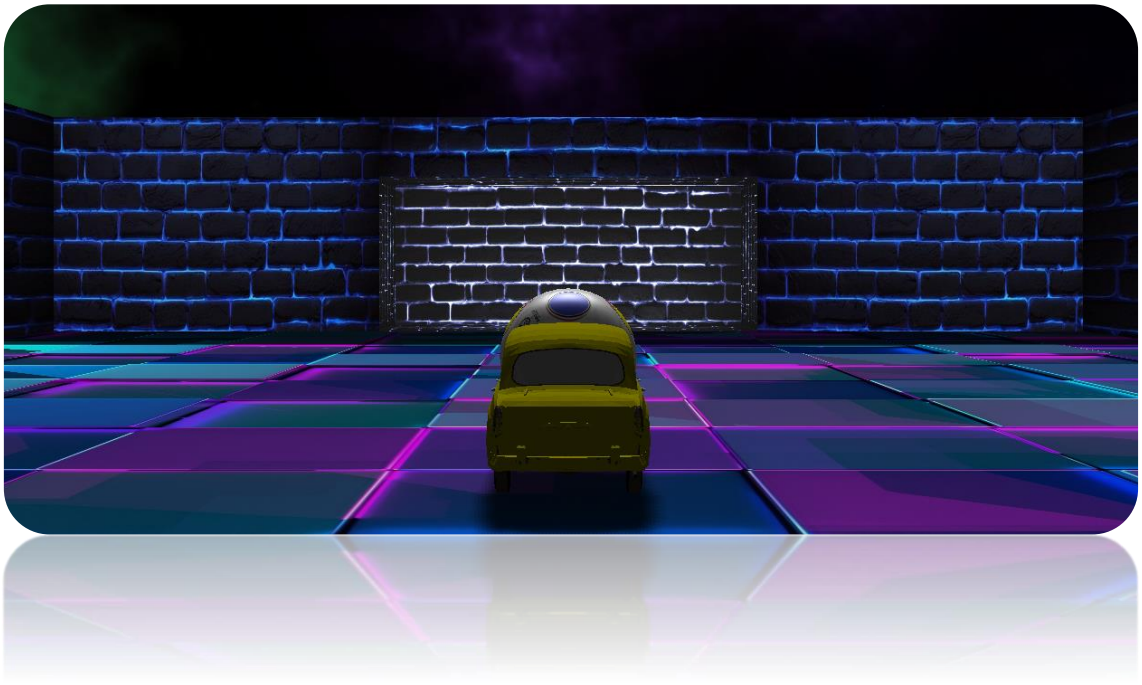
Para la realización del proyecto, ciertos recursos han sido tomados de las siguientes plataformas digitales:

- Texturas -> https://www.freepik.com
- Skybox -> https://opengameart.org/content
- Coche (Moskvitch) -> Material proporcionado para el proyecto.

8 Pruebas del funcionamiento

- **Escena inicial**:

Al iniciar el programa, el coche se encuentra mirando hacia la portería y la pelota justo delante de él, ambos sin movimiento inicial y reposando sobre el suelo del terreno de juego:

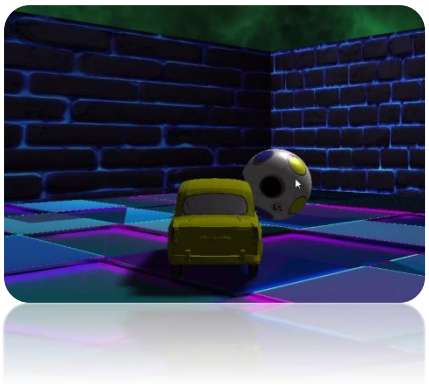


- **Movimiento del coche y cámara**:

Al realizar movimientos con el coche de giro y desplazamiento, vemos cómo la cámara sigue perfectamente colocada detrás del coche y orientada hacia el mismo:

- **Golpeo del balón y choque con las paredes:**

Cuando el coche se aproxima al balón, vemos cómo lo golpea y la pelota sale disparada con una velocidad y altura proporcionales a la velocidad de impacto:

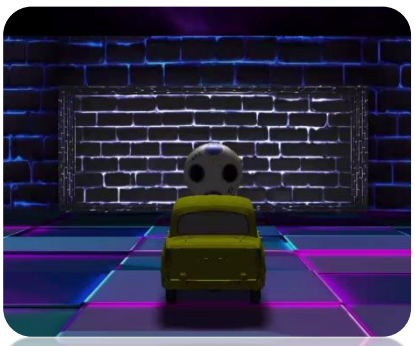




Al chocar tanto el coche como la pelota contra una pared, se observa la colisión correspondiente y cómo la velocidad (una vez invertida tras el choque) se ve reducida. Cabe destacar que la pelota rebota más que el coche:



- **Detección de gol:**

Finalmente, cuando el balón entra por completo en la portería, el programa termina indicando que se ha marcado gol:

```
Consola de depuración de Microsoft Visual Studio
GOL
C:\Proyecto RV\x64\Release\ProyectoRV.exe (proceso 19232) se cerró con el código 0.
Para cerrar automáticamente la consola cuando se detiene la depuración, habilite Herramientas ->Opciones ->Depuración ->
Cerrar la consola automáticamente al detenerse la depuración.
Presione cualquier tecla para cerrar esta ventana. . .
```