Pablo Fernández Ivars

REALIDAD VIRTUAL: ENTRENAMIENTO ROCKET LEAGUE

DESARROLLO DE LA PORTERÍA Y LAS PAREDES DEL CAMPO

Para hacer las paredes del estadio, he reutilizado el tipo CGGround con el que implementamos el suelo, usando sus métodos Rotate() y Translate() para formar el "estadio", además de crear diferentes materiales para sus texturas usando algunas imágenes.

- Ejemplo creación material ->

```
matg3 = new CGMaterial();  //pared porteria
matg3->SetAmbientReflect(5.0f, 5.0f, 5.0f);
matg3->SetDifusseReflect(1.0f, 1.0f, 1.0f);
matg3->SetSpecularReflect(0.8f, 0.8f, 0.8f);
matg3->SetShininess(16.0f);
matg3->InitTexture("textures/roca3.jpg");
//matg->InitTexture(IDR_IMAGE2);
```

- Ejemplo creación pared ->

```
pared2 = new CGGround(38.0f, 155.0f);
pared2->SetMaterial(matg2);
pared2->Translate(glm::vec3(128.0f, 38.0f, 0.0f));
pared2->Rotate(90.0f, glm::vec3(0.0f, 0.0f, 1.0f));
```

La portería ha sido creada a partir de la clase Cubo, la cual fue modificada para que tuviera unas aristas más largas (y asi asemejarse a un rectángulo), además de quitarle una de sus caras, para formar el vacío que simula la profundidad de una portería. Esta fue colocada en el hueco entre 3 paredes para que quedara empotrada en la pared

- Creación Portería ->

```
porteria = new Porteria(20.0f, 15.0f);
porteria->SetMaterial(matg4);
porteria->Translate(glm::vec3(-32.0f, 20.0f, -170.0f));
porteria->Rotate(180.0f, glm::vec3(1.0f, 0.0f, 0.0f));
```

<- Resultado Final

CLASE “CGSCENE”

La clase CGScene representa una escena gráfica del programa, incluye luces, objetos (como el coche y la pelota), materiales y otros elementos geométricos como paredes y el suelo.

Esta clase tiene tres funciones principales: el constructor CGScene(), el destructor ~CGScene(), y las funciones para dibujar la escena y sus sombras.

- Constructor CGScene():

  Este constructor se encarga de inicializar todos los elementos de la escena.

  1.- Inicialización de la dirección de la luz: creando un vector LDir y normalizándolo.

  2.- Creación y configuración del objeto de luz: estableciendo su dirección y sus tipos de luces.

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

  3.- Creación y configuración del objeto principal (coche): Se crea el objeto coche y se establece su posición inicial, rotándolo en el proceso.

```
object = new Chrysler_Saratoga(); // COCHE
object->Translate(glm::vec3(0.0f, 0.0f, 0.0f));
object->Rotate(-180.0f, glm::vec3(0.0f, 10.0f, 0.0f));
```

  4.- Creación y configuración de los materiales: crea los materiales que serán usados por el suelo, paredes y portería, además de agregarle sus texturas a través de una foto que tenemos guardada en el proyecto.

  5.- Creación del suelo, paredes y portería: se crean las 6 paredes utilizadas, el suelo y la portería, inicializando sus texturas y posición en el mapa.

```
ground = new CGGround(128.0f, 155.0f);
ground->SetMaterial(matg);

matg2 = new CGMaterial();  //fondos
matg2->SetAmbientReflect(5.0f, 5.0f, 5.0f);
matg2->SetDifusseReflect(1.0f, 1.0f, 1.0f);
matg2->SetSpecularReflect(0.8f, 0.8f, 0.8f);
matg2->SetShininess(16.0f);
matg2->InitTexture("textures/roca1.jpg");
//matg->InitTexture(IDR_IMAGE2);
```

```
pared6 = new CGGround(46.0f, 18.0f);
pared6->SetMaterial(matg3);
pared6->Translate(glm::vec3(-2.0f, 58.0f, -155.0f));
pared6->Rotate(90.0f, glm::vec3(1.0f, 0.0f, 0.0f));

porteria = new Porteria(20.0f, 15.0f);
porteria->SetMaterial(matg4);
porteria->Translate(glm::vec3(-32.0f, 20.0f, -170.0f));
porteria->Rotate(180.0f, glm::vec3(1.0f, 0.0f, 0.0f));
```

6.- Creación de la pelota: la pelota es una instancia de la clase Sphere, a la que se le ha dado una textura de balón de fútbol.

```
mat4 = new CGMaterial();
mat4->SetAmbientReflect(1.0f, 1.0f, 1.0f);
mat4->SetDifusseReflect(1.0f, 1.0f, 1.0f);
mat4->SetSpecularReflect(0.8f, 0.8f, 0.8f);
mat4->SetShininess(16.0f);
mat4->InitTexture("textures/ball.jpg"); //textura pelota
//mat4->InitTexture(IDR_IMAGE3);

fig4 = new CGSphere(20, 40, 6.5f); //pelota
fig4->SetMaterial(mat4);
fig4->Translate(glm::vec3(2.0f, 30.0f, -50.0f));
```

❖ Destructor CGScene::~CGScene():

El destructor CGScene::~CGScene() asegura que todos los recursos dinámicamente asignados en el constructor sean liberados, evitando fugas de memoria.

```
CGScene::~CGScene()
{
    delete ground;
    delete fig4;
    delete light;
    delete matg;
    delete matg2;
    delete matg3;
    delete matg4;
    delete mat4;
    delete object;
    delete pared1;
    delete pared2;
    delete pared3;
    delete pared4;
    delete pared5;
    delete pared6;
    delete porteria;
}
```

- CGScene::DrawShadow (CGShaderProgram* program, glm::mat4 shadowMatrix):

  Este método se encarga de dibujar las sombras de la escena utilizando un programa de shaders específico para sombras y una matriz de sombras. Dibuja las sombras de las paredes, la pelota y el coche.

- CGScene::GetLightViewMatrix():

  Este método calcula y devuelve la matriz de vista de la luz, que es utilizada para las sombras.

```
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;
}
```

- Getters coche y balón:

  Sirven para acceder a los dos objetos fuera de la clase GCScene;

CÁLCULO DE LA POSICIÓN DE LA CAMARA

El método colocarCamara() que se encuentra en la clase CGModel la uso para configurar la posición y la dirección de la cámara en una escena hacia la parte trasera del coche.

Este método obtiene la posición real del coche y las direcciones fundamentales forward (obtiene la dirección hacia adelante del coche, que indica hacia dónde está "mirando" el coche), up (obtiene la dirección hacia arriba del coche, que generalmente apunta hacia el cielo) y right (orientación hacia la derecha e izquierda) que describen la orientación del coche en el espacio 3D.

```
glm::vec3 posicionReal = scene->getCoche()->GetRealPosition();
glm::vec3 forward = scene->getCoche()->getForwardDirection();
glm::vec3 up = scene->getCoche()->getUpDirection();
glm::vec3 r = scene->getCoche()->getRightDirection();
```

Luego, calcula la posición de la cámara usando estas direcciones multiplicados por unos índices que, a base de prueba y error, han demostrado ser los mejores para ver el terreno de juego correctamente y que no fuera obstaculizada por la parte trasera del coche.

Finalmente, la dirección de la cámara se ajusta con la dirección -forward, para asegurar que la cámara esté todo el tiempo mirando hacia el coche desde atrás y así proporcionar una vista dinámica y coherente durante el movimiento del coche en la escena.

```
glm::vec3 cameraPosition = posicionReal - forward * 70.0f + up * 15.0f;

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

camera->SetDir(-forward);
```

CÁLCULO DEL MOVIMIENTO DEL BALON, DEL COCHE Y SUS IMPACTOS

Por orden de realización, el movimiento del coche lo conseguí basándome en los movimientos ya existentes en la clase Camera.

Dentro de la clase Object podemos encontrar una serie de atributos y métodos que nos ayudaran con esto. El primero de ellos es el atributo tipo CGFloat llamado moveStep, que por así decirlo es el "paso" que realiza el coche en el mapa hacia la dirección en la que está colocado.

Para hacer efectivo este paso, debemos usar el método MoveFront(). Este método se ejecuta continuamente en la clase CGModel, debido a que este ejecuta las translaciones del coche usando el atributo moveStep y el atributo Dir. Además de esto, este método nos sirve para generar el efecto del rozamiento del coche con el aire, ya que cada vez que se ejecuta reduce un poco el moveStep, si este es mayor que 0. Por último, detecté que al dejar de mover el coche este seguía manteniendo una pequeña velocidad, casi imperceptible, pero que a la larga podría causar algún error de traspasamiento de paredes, por lo que comprobar la velocidad a la que se movía el coche en ese caso y establecer una condición para frenar completamente el coche en esa situación.

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

    if (moveStep < 0.001 && moveStep > -0.001)
        moveStep = 0;

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

Además, en esta misma clase tenemos los métodos SetMoveStep(GLfloat step) y GetMoveStep() para poder tratar con el parámetro fuera de esta.

Para mover efectivamente el coche, dentro de la clase CGModel existe un método key_pressed(int key), que sirve para ejecutar métodos cuando una tecla sea pulsada por el usuario y donde hemos asignado las teclas de dirección para poder hacer los movimientos y giros efectivamente. Para las teclas de arriba y abajo, hemos incrementado o decrementado respectivamente el moveStep del coche y para las teclas de derecha e izquierda he realizado un rotate() que modificara ligeramente su trayectoria hacia la dirección deseada. Se contempló poner un mínimo de velocidad para poder girar hacia los lados el coche, pero debido a que el mapa es relativamente corto, hay muchas veces que se dificultaría el movimiento del coche, por lo que finalmente esta medida fue descartada.

```
void CGModel::key_pressed(int key)
{
    switch (key)
    {
    case GLFW_KEY_UP:
        scene->getCoche()->SetMoveStep(scene->getCoche()->GetMoveStep() + 0.2f);
        break;
    case GLFW_KEY_DOWN:
        scene->getCoche()->SetMoveStep(scene->getCoche()->GetMoveStep() - 0.2f);
        break;
    /*case GLFW_KEY_LEFT:
        if(scene->getCoche()->GetMoveStep() >= 0.4 || scene->getCoche()->GetMoveStep() <= -0.4)
           scene->getCoche()->Rotate(4.0f, glm::vec3(0.0f, 1.0f, 0.0f));
        break;
    case GLFW_KEY_RIGHT:
        if (scene->getCoche()->GetMoveStep() >= 0.4 || scene->getCoche()->GetMoveStep() <= -0.4)
            scene->getCoche()->Rotate(-4.0f, glm::vec3(0.0f, 1.0f, 0.0f));
        break;
    }*/
    case GLFW_KEY_LEFT:
        //if (scene->getCoche()->GetMoveStep() >= 0.15 || scene->getCoche()->GetMoveStep() <= 0.15)
            scene->getCoche()->Rotate(4.0f, glm::vec3(0.0f, 1.0f, 0.0f));
        break;
    case GLFW_KEY_RIGHT:
        //if (scene->getCoche()->GetMoveStep() >= 0.15 || scene->getCoche()->GetMoveStep() <= 0.15)
            scene->getCoche()->Rotate(-4.0f, glm::vec3(0.0f, 1.0f, 0.0f));
        break;
    }
}
```

Lo siguiente que realice fue el movimiento de la pelota, por lo que esta vez la clase modificada fue la CGFigure, que es la clase padre de la clase CGSphere de la que la pelota es una instancia.

Se implementan exactamente los mismos métodos y atributos que en CGObject pero con algunas modificaciones en el MoveFront() y se añade el atributo upStep, que nos sirve para añadirle movimiento en la componente y al balón. Las modificaciones en el método se basan en incluirle gravedad a la bola, es decir, que cuando sea golpeada o choque contra algo, su movimiento vaya describiendo una parábola para añadirle algo de realismo. También, al añadir upStep dentro del vector la dirección a la que se mueve cambia.

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

    if (moveStep < 0.001 && moveStep > -0.001)
        moveStep = 0;

    if (this->GetRealPosition().y > 6.5f)
        upStep -= 0.006f;
    else if (upStep < 0.0f)
        upStep = -upStep * 0.6;

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

Para crear los impactos del coche y del balón, usé un método BallHits(), el cual se basa en, mediante las posiciones del coche y del balón, calcular una distancia de impacto en cada una de las componentes, que se combinará en una variable llamada distanciaXYZ, la cual determinará si la pelota ha sido golpeada por el coche. Si es golpeada, primero se calcula la dirección en la que ira disparado el balón (depende de si fue golpeado marcha atrás o no) y después transmitirle parte de la velocidad que llevaba el coche en el momento del golpeo, además de hacer que se levante un poco la pelota del suelo subiéndole el upStep.

```
void CGModel::BallHits() {
    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 distanciaXYZ = sqrt((dx * dx) + (dz * dz) + (dy * dy));

    if (distanciaXYZ <= 15.0f) {
        if(scene->getCoche()->GetMoveStep() < 0.0f)
            scene->getBalon()->SetDireccion(glm::vec3(-dx, -dy, -dz));
        else
            scene->getBalon()->SetDireccion(glm::vec3(dx, dy, dz));

        scene->getBalon()->SetMoveStep(scene->getCoche()->GetMoveStep() * 0.3f);
        scene->getBalon()->SetupStep(0.25f);
    }

}
```

DETECCIÓN DE CHOQUES CON PAREDES Y GOL

Se han usado dos funciones de la clase CGModel principalmentes: CarConstraints() y BallConstraints(). Estas funciones se encargan de gestionar las restricciones de movimiento para el coche y el balón, respectivamente.

En el método CarConstraints() detectamos colisiones entre el coche y las paredes del campo de juego. Primero comprobamos si el coche se encuentra entre los limites establecidos en el mapa (siempre son un poco antes que las paredes debido a que no sería realista que el coche chocara con la pared sabiendo que su eje está en el centro y la parte delantera de este la atravesaría) y, si estamos empezando a salir de él, producimos un rebote del coche invirtiendo el moveStep de este, además de aplicarle un coeficiente de rozamiento para que no salga disparado a la misma velocidad a la que chocó.

```
void CGModel::CarConstraints() {
    glm::vec3 posS = scene->getCoche()->GetRealPosition();
    int constraint = 0;

    //std::cout << " movestep: " << scene->getCoche()->GetMoveStep() << std::endl;
    //std::cout << " x: " << posS.x << " y: " << posS.y << " z: " << posS.z << std::endl;

    if (fueracoche == true && scene->getCoche()->GetMoveStep() > 0)
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->GetMoveStep()));

    if (marchaatras == true && scene->getCoche()->GetMoveStep() < 0)
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->GetMoveStep()));

    if (posS.x > 113.0f) {
        constraint = 1;
    }
    if (posS.x < -113.0f) {
        constraint = 1;
    }
    if (posS.z > 145.0f) {
        constraint = 1;
    }
    if ((posS.x > 40 || posS.x < -40) && (posS.z < -145))
        constraint = 1;

    if ((posS.x < 40 || posS.x < -40) && (posS.z < -180))
        constraint = 1;
```

```
if (constraint == 1) {
    if (scene->getCoche()->GetMoveStep() > 0 && !marchaatras && !fueracoche) {
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->GetMoveStep() * 0.3f));
        fueracoche = true;

    }
    else if (scene->getCoche()->GetMoveStep() < 0 && marchaatras == false && !fueracoche)
    {
        scene->getCoche()->SetMoveStep(-(scene->getCoche()->GetMoveStep() * 0.4f));
        marchaatras = true;
    }

}
else if (constraint == 0) {
    fueracoche = false;
    marchaatras = false;
}
}
```

Para evitar traspasar la pared, he incluido una serie de condiciones para que, si en la ejecución anterior del CarContrains se estaba intentando salir del mapa y continuas asi, bloqueé el movimiento, ya que comprobé que, sin esto, al insistir contra la pared, el coche terminaba atravesándola sin mucho esfuerza. Si no se encuentra fuera del mapa, simplemente pone las variables de comprobación a 0.

BallConstrains() es prácticamente una copia de CarConstrains(), pero aplicado para la pelota. La única modificación que presenta con respecto a este es la adición de una ultima comprobación, y es que si la bola se encuentra en la portería y traspasa totalmente la línea de gol, es decir, se encuentra completa dentro, el programa se cierra en señal de gol.

```
if ((posS.x < 40 || posS.x < -40) && (posS.z < -164)) {
    std::cout << " G000000000000000L " << std::endl;
    exit(0);
}
```

REFERENCIAS TEXTURAS Y MODELOS

Enlaces :

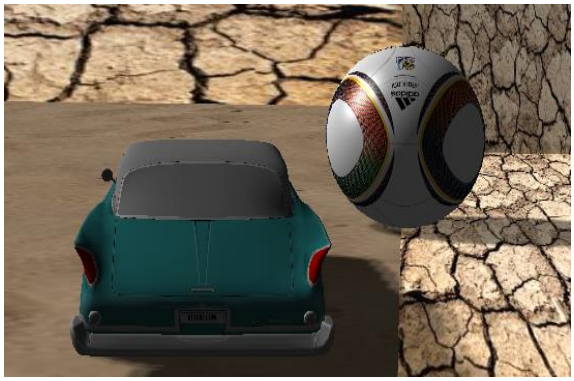
- https://www.pinterest.es/ Texturas paredes, porterías y suelos.
- https://www.humus.name/ Skybox;

FUNCIONAMIENTO DE LA APLICACIÓN

VISTAS DEL COCHE:

CHOQUE DE LA PELOTA CON EL COCHE:

VISTAS PORTERÍA Y BALÓN:



CHOQUE Y REBOTE CON LA PARED: