Realidad Virtual – Trabajo Final

Desarrollo de un videojuego mediante OpenGL

Memoria Técnica

Universidad de Huelva

Realizado por:

Álvaro Esteban Muñoz

Índice

Introducción

Debemos desarrollar un videojuego basado en un modo de entrenamiento para el conocido juego "Rocket League" haciendo uso directo de las funciones que nos ofrece la librería OpenGL. Las reglas son sencillas, ser un modo para un único jugador en el que el objetivo es marcar gol con un único coche a modo de entrenamiento del juego original.

Debemos mencionar que se ha tomado como punto de partida una versión de las prácticas realizadas en clase, las clases básicas como "*CGGround*", "*CGApplication*", "*CGFigure*", etc…, vienen ya implementadas.

Descripción de los modelos diseñados

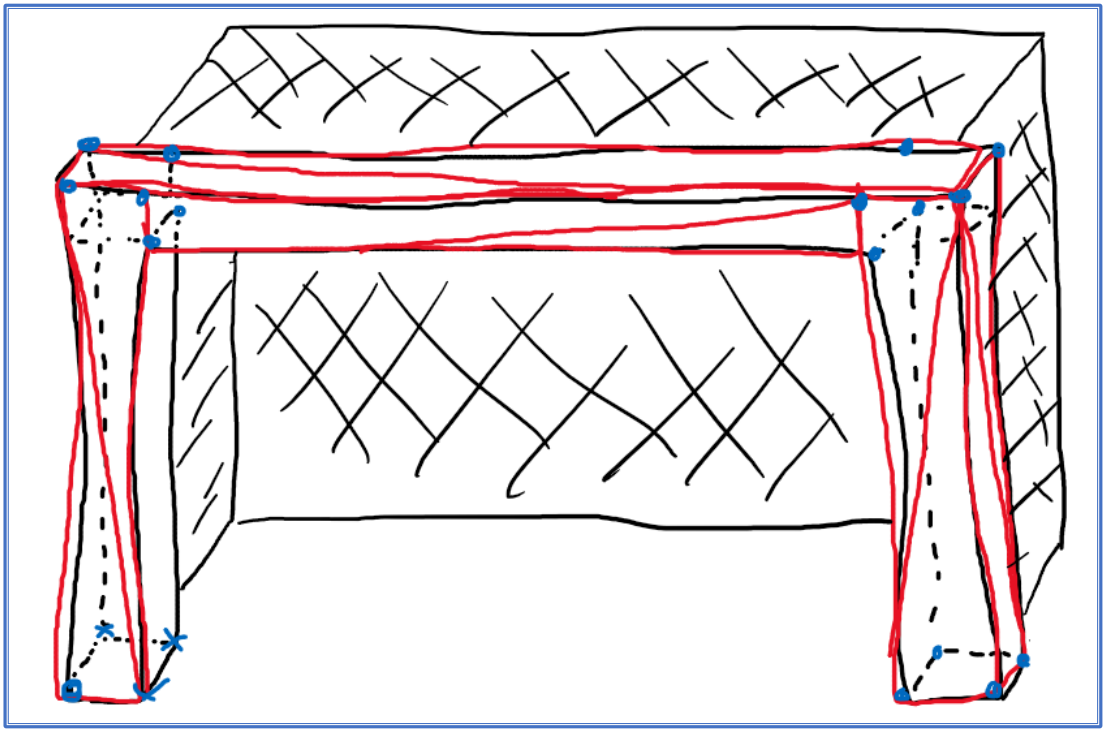
Para el **terreno de juego** hemos desarrollado una clase nueva que se denomina *"Pared"*, que nos ahorrará la rotación de reutilizar la clase "*CGGround*" y nos permitirá tener un código más organizado. El campo está compuesto por una figura de la clase "*CGGround*", que representará el suelo, tres figuras de la clase "*Pared*" y una figura de la clase "*ParedFrontal*", que difiere de la clase "*Pared*" en que además añade un hueco en medio para incrustar la **portería**. También tenemos una portería que hemos desarrollado dividiéndola en dos partes, arco y red.

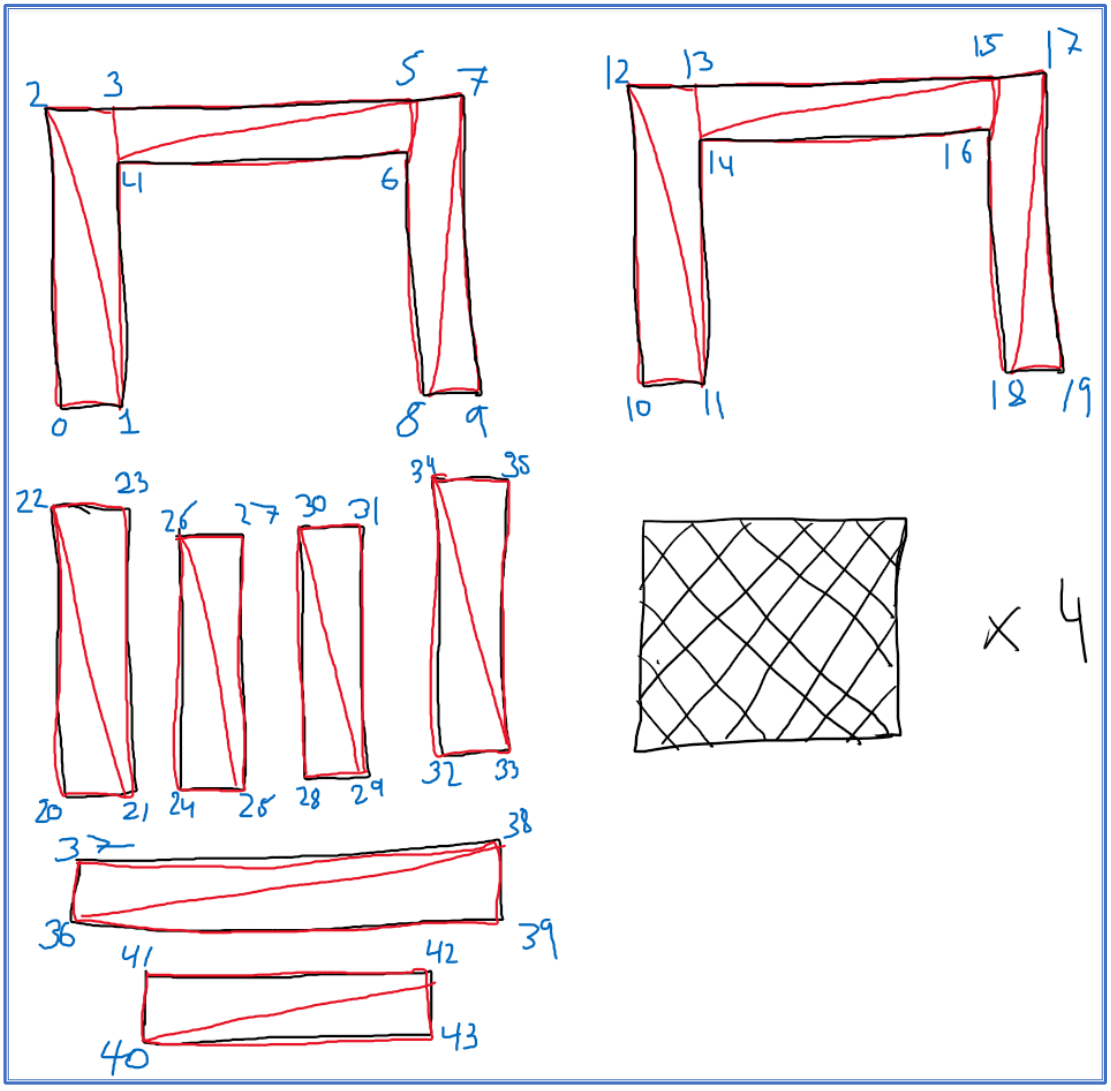
La **red de la portería** está compuesta por tres figuras *"Pared"*, y una figura "*CGGround"*, las paredes han sido rotadas debidamente al igual que el "*CGGround*" que es el que hace de parte superior.

Para el **arco de la portería** hemos desarrollado una clase "*Porteria*" que basará las coordenadas de sus vértices en el ancho, el alto y el grosor de sus postes.

```
1   #pragma once
2
3   #include <GL/glew.h>
4   #include "CGFigure.h"
5
6   /*
7   * Portería de 'w' de ancho, 'h' de alto y postes de 'd' de grosor
8   */
9   class Porteria : public CGFigure {
10  public:
11      Porteria(GLfloat w, GLfloat h, GLfloat d);
12  };
13
14
```

Para crear el arco de la portería se ha dividido la figura en **ocho caras planas**, aunque repetiremos vértices, hacer esto nos ayudará a tener el código más organizado y nos facilitará la implementación de su textura.




0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43
x 4

La relación entre estos vértices y las medidas del modelo vienen en el array de vértices de la figura.

```
GLfloat p_vertices[44][3] = {
    // Cara delantera
    { 0, 0, d },              // 0
    { d, 0, d },              // 1
    { 0, h, d },              // 2
    { d, h, d },              // 3
    { d, h - d, d },          // 4
    { w - d, h, d },          // 5
    { w - d, h - d, d },      // 6
    { w, h, d },              // 7
    { w - d, 0, d },          // 8
    { w, 0, d },              // 9
    // Cara trasera
    { w, 0, 0 },              // 10
    { w - d, 0, 0 },          // 11
    { w, h, 0 },              // 12
    { w - d, h, 0 },          // 13
    { w - d, h - d, 0 },      // 14
    { d, h, 0 },              // 15
    { d, h - d, 0 },          // 16
    { 0, h, 0 },              // 17
    { d, 0, 0 },              // 18
    { 0, 0, 0 },              // 19
    // Palo izquierdo - Lateral exterior
    { 0, 0, 0 },              // 20
    { 0, 0, d },              // 21
    { 0, h, 0 },              // 22
    { 0, h, d },              // 23
    // Palo izquierdo - Lateral interior
    { d, 0, d },              // 24
    { d, 0, 0 },              // 25
    { d, h - d, d },          // 26
    { d, h - d, 0 },          // 27
    // Palo derecho - Lateral interior
    { w - d, 0, 0 },          // 28
    { w - d, 0, d },          // 29
    { w - d, h - d, 0 },      // 30
    { w - d, h - d, d },      // 31
    // Palo derecho - Lateral exterior
    { w, 0, d },              // 32
    { w, 0, 0 },              // 33
    { w, h, d },              // 34
    { w, h, 0 },              // 35
    // Palo superior - Cara superior
    { 0, h, d },              // 36
    { 0, h, 0 },              // 37
    { w, h, 0 },              // 38
    { w, h, d },              // 39
    // Palo superior - Cara inferior
    { d, h - d, 0 },
    { d, h - d, d },
    { w - d, h - d, d },
    { w - d, h - d, 0 }
};
```

Para dibujar las primitivas tendremos que dejar claro cuales son los vértices que van a participar en cada una, para eso está el array de índices.

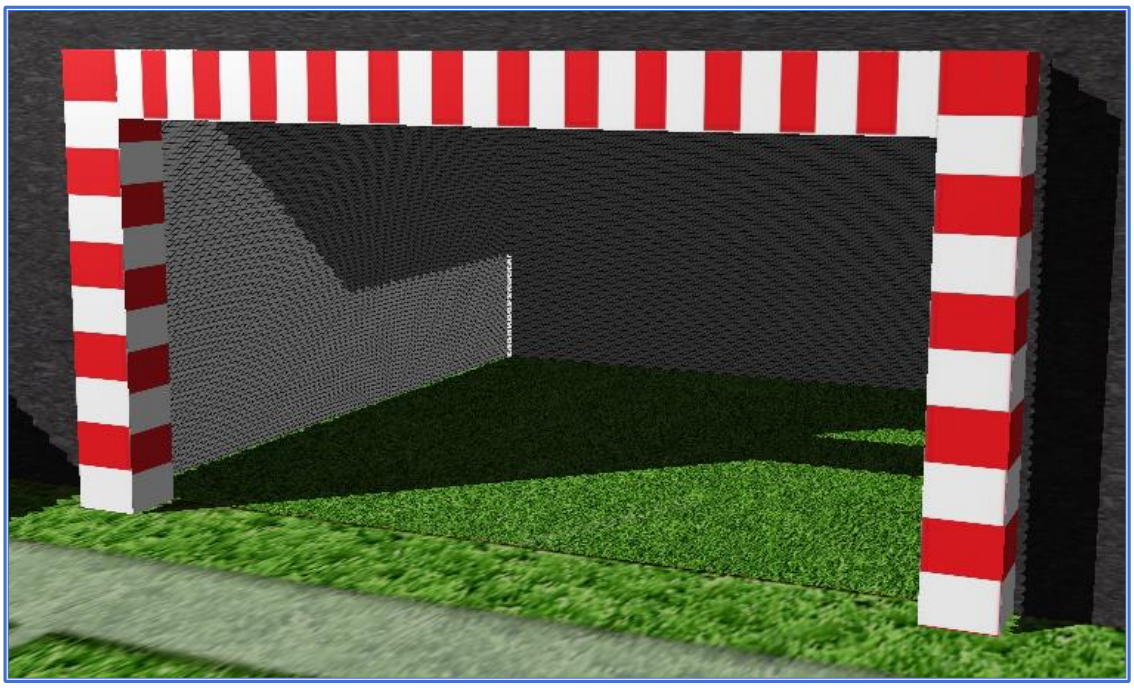
```
GLushort p_indexes[24][3]{
    // Cara delantera
    {0, 1, 2},
    {2, 1, 3},
    {3, 4, 5},
    {5, 4, 6},
    {5, 8, 7},
    {7, 8, 9},
    // Cara trasera
    {10, 11, 12},
    {12, 11, 13},
    {13, 14, 15},
    {15, 14, 16},
    {15, 18, 17},
    {17, 18, 19},
    // Palo izquierdo - Lateral exterior
    {20, 21, 22},
    {22, 21, 23},
    // Palo izquierdo - Lateral interior
    {24, 25, 26},
    {26, 25, 27},
    // Palo derecho - Lateral interior
    {28, 29, 30},
    {30, 29, 31},
    // Palo derecho - Lateral exterior
    {34, 32, 33},
    {34, 33, 35},
    // Palo superior - Cara superior
    {36, 38, 37},
    {38, 36, 39},
    // Palo superior - Cara inferior
    {40, 42, 41},
    {42, 40, 43}
};
```

El cálculo de las coordenadas de textura es muy simple, hemos medido los píxeles en la imagen que queremos poner de textura y hemos normalizado dichas coordenadas.

```
GLfloat p_textures[44][2] = { // Array of te
    // Cara delantera
    { 0.0f, 0.0f },         // 0
    { 0.0518f, 0.0f },      // 1
    { 0.0f, 1.0f },         // 2
    { 0.0518f, 1.0f },      // 3
    { 0.0518f, 0.9482f },   // 4
    { 0.9482f, 1.0f },      // 5
    { 0.9482f, 0.9482f },   // 6
    { 1.0f, 1.0f },         // 7
    { 0.9482f, 0.0f },      // 8
    { 1.0f, 0.0f },         // 9
    // Cara trasera
    { 0.0f, 0.0f },         // 10
    { 0.0518f, 0.0f },      // 11
    { 0.0f, 1.0f },         // 12
    { 0.0518f, 1.0f },      // 13
    { 0.0518f, 0.9482f },   // 14
    { 0.9482f, 1.0f },      // 15
    { 0.9482f, 0.9482f },   // 16
    { 1.0f, 1.0f },         // 17
    { 0.9482f, 0.0f },      // 18
    { 1.0f, 0.0f },         // 19
    // Palo izquierdo - Lateral exterior
    { 0.0f, 0.0f },         // 20
    { 0.0518f, 0.0f },      // 21
    { 0.0f, 1.0f },         // 22
    { 0.0518f, 1.0f },      // 23
    // Palo izquierdo - Lateral interior
    { 0.0f, 0.0f },         // 24
    { 0.0518f, 0.0f },      // 25
    { 0.0f, 0.9482f },      // 26
    { 0.0518f, 0.9482f },   // 27
    // Palo derecho - Lateral interior
    { 0.0f, 0.0f },         // 28
    { 0.0518f, 0.0f },      // 29
    { 0.0f, 0.9482f },      // 30
    { 0.0518f, 0.9482f },   // 31
    // Palo derecho - Lateral exterior
    { 0.0f, 0.0f },         // 32
    { 0.0518f, 0.0f },      // 33
    { 0.0f, 1.0f },         // 34
    { 0.0518f, 1.0f },      // 35
    // Palo superior - Cara superior
    { 0.0f, 0.9482f },      // 36
    { 0.0f, 1.0f },         // 37
    { 1.0f, 1.0f },         // 38
    { 1.0f, 0.9482f },      // 39
    // Palo superior - Cara inferior
    { 0.0518f, 0.9482f },   // 40
    { 0.0518f, 1.0f },      // 41
    { 0.9482f, 1.0f },      // 42
    { 0.9482f, 0.9482f }    // 43
};
```

Al ser una figura con caras planas, el cálculo de los vectores normales también es muy simple, se reduce al vector de la superficie en cada punto.

```
GLfloat p_vertices[44][3] = {
  // Cara delantera
  { 0, 0, d },              // 0
  { d, 0, d },              // 1
  { 0, h, d },              // 2
  { d, h, d },              // 3
  { d, h - d, d },          // 4
  { w - d, h, d },          // 5
  { w - d, h - d, d },      // 6
  { w, h, d },              // 7
  { w - d, 0, d },          // 8
  { w, 0, d },              // 9
  // Cara trasera
  { w, 0, 0 },              // 10
  { w - d, 0, 0 },          // 11
  { w, h, 0 },              // 12
  { w - d, h, 0 },          // 13
  { w - d, h - d, 0 },      // 14
  { d, h, 0 },              // 15
  { d, h - d, 0 },          // 16
  { 0, h, 0 },              // 17
  { d, 0, 0 },              // 18
  { 0, 0, 0 },              // 19
  // Palo izquierdo - Lateral exterior
  { 0, 0, 0 },              // 20
  { 0, 0, d },              // 21
  { 0, h, 0 },              // 22
  { 0, h, d },              // 23
  // Palo izquierdo - Lateral interior
  { d, 0, d },              // 24
  { d, 0, 0 },              // 25
  { d, h - d, d },          // 26
  { d, h - d, 0 },          // 27
  // Palo derecho - Lateral interior
  { w - d, 0, 0 },          // 28
  { w - d, 0, d },          // 29
  { w - d, h - d, 0 },      // 30
  { w - d, h - d, d },      // 31
  // Palo derecho - Lateral exterior
  { w, 0, d },              // 32
  { w, 0, 0 },              // 33
  { w, h, d },              // 34
  { w, h, 0 },              // 35
  // Palo superior - Cara superior
  { 0, h, d },              // 36
  { 0, h, 0 },              // 37
  { w, h, 0 },              // 38
  { w, h, d },              // 39
  // Palo superior - Cara inferior
  { d, h - d, 0 },
  { d, h - d, d },
  { w - d, h - d, d },
  { w - d, h - d, 0 }
};
```



Descripción de la clase que desarrolla la escena

La mayoría de las cosas que se ven en la escena se inicializan en el constructor de la clase "*CGScene*" este constructor tendrá la tarea de inicializar los siguientes elementos:

Elemento de la escena	Subelementos	Clase
Base	base	CGFigure
	matb	CGMaterial
Suelo del campo	ground	CGFigure
	matg	CGMaterial
Pelota	pelota	CGFigure
	matPelota	CGMaterial
Coche	coche	CGObject
	matCoche	CGMaterial
Porteria	portería	CGFigure
	sueloPorteria	CGFigure
	matPorteria	CGMaterial
	matSueloPort	CGMaterial
	matRed	CGMaterial
	redPorteria_01	CGFigure
	redPorteria_02	CGFigure
	redPorteria_03	CGFigure
	redPorteria_04	CGFigure
Paredes del campo	pared_01	CGFigure
	pared_02	CGFigure
	pared_03	CGFigure
	pared_04	CGFigure

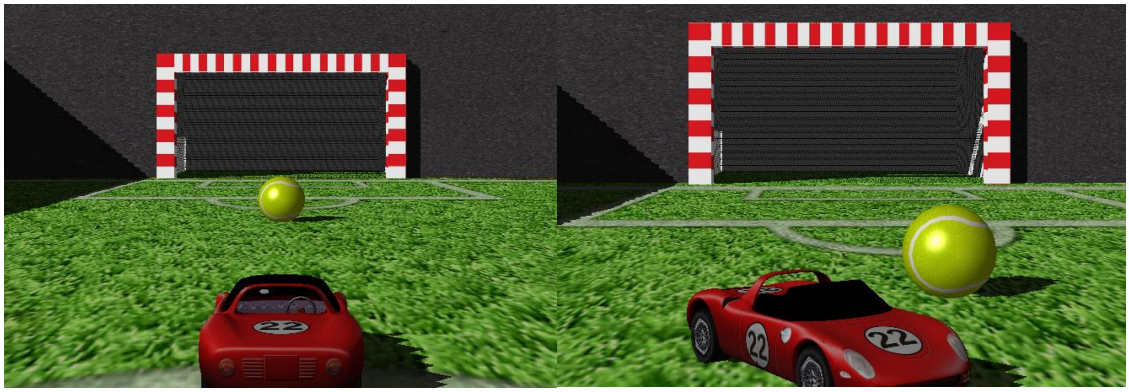
	matPared	CGMaterial
Luz	light	CGLight
Niebla	fog	CGFog
Skybox	skybox	CGSkybox

Cálculo de la posición de la cámara

Para la cámara se ha optado por un diseño en **tercera persona**, siempre se ve al coche desde atrás en una dirección **fija** en el eje Z, esto permite una sencilla implementación ya que no es necesario realizar el reajuste de la dirección de la cámara cada vez que se rota el coche, aunque si que implique el recalculo de su posición respecto de la posición del coche.

Puesto que la cámara siempre sigue al coche, no hemos necesitado una clase que se encargue de manejarla, simplemente hemos realizado la inicialización de la matriz **View** dentro de la clase del coche, esto simplifica mucho el código y la estructura de clases.

Para calcular la posición de la cámara simplemente hemos necesitado saber la posición del coche cada vez que esta se ve alterada y a partir de ahí se actualiza la posición de la matriz View como la posición del coche más la distancia a la que queremos tener la cámara de este.



Físicas

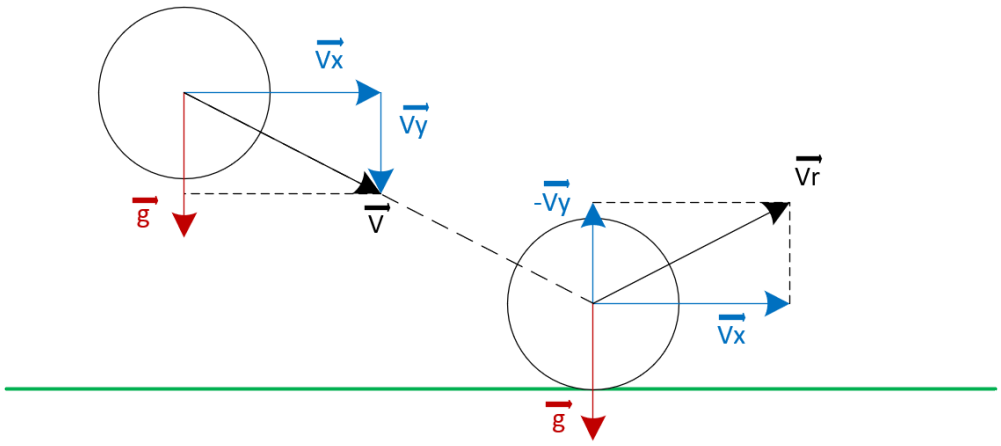
Para animar la escena hemos tenido que programar las físicas de cada objeto de la escena, se ha intentado imitar un modelo similar al de la realidad.

Físicas de la pelota

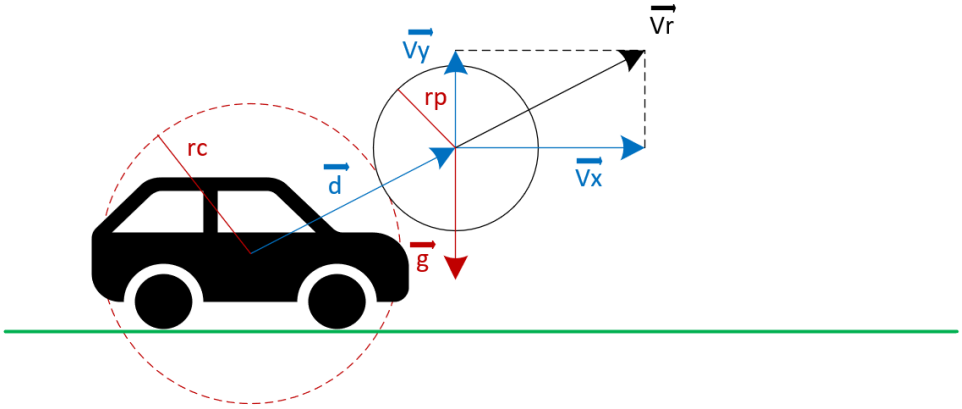
Para el desarrollo de las físicas de la pelota se ha implementado un modelo basado en el movimiento uniformemente acelerado, es decir, hemos definido un atributo para la **velocidad** y otro para la **aceleración**.

La **aceleración** de la pelota en el eje de las ordenadas será siempre la **gravedad** terrestre, de forma que cuando la pelota sea levantada, esta tienda a caer al suelo. Por otro lado, la pelota no debe rebasar nunca el suelo ni las paredes, para ello, hemos decidido que cuando se detecte la posición de la pelota a la distancia de su radio de alguno de los

límites del campo, la **velocidad** de su componente perpendicular a la superficie se verá **invertida**, provocando así un efecto de rebote como el que se ve en el dibujo.



El impacto del coche contra la pelota se ha simplificado a la colisión de dos esferas, esto limita mucho los cálculos ya que solo tendremos que comprobar que el módulo del vector distancia entre los centros de las esferas no sea igual a la suma de los radios.



Si |d| == rc + rp → Colisión

En caso de **colisión** se actuará proporcionándole al vector dirección de la pelota al coche la velocidad del coche, y el resultado será la velocidad aplicada sobre la pelota.

Físicas del coche

El caso del coche es muy similar al de la pelota, puesto que este solo se puede mover en el plano X-Z, nos ahorramos los cálculos relacionados con la gravedad.

El coche puede ser acelerado por el usuario mediante los controles, la tecla **W** permite acelerar el coche mientras que la tecla **S** permite decelerar el coche (deceleraciones por debajo de 0 invierten el sentido de la velocidad, provocando la marcha atrás del coche).

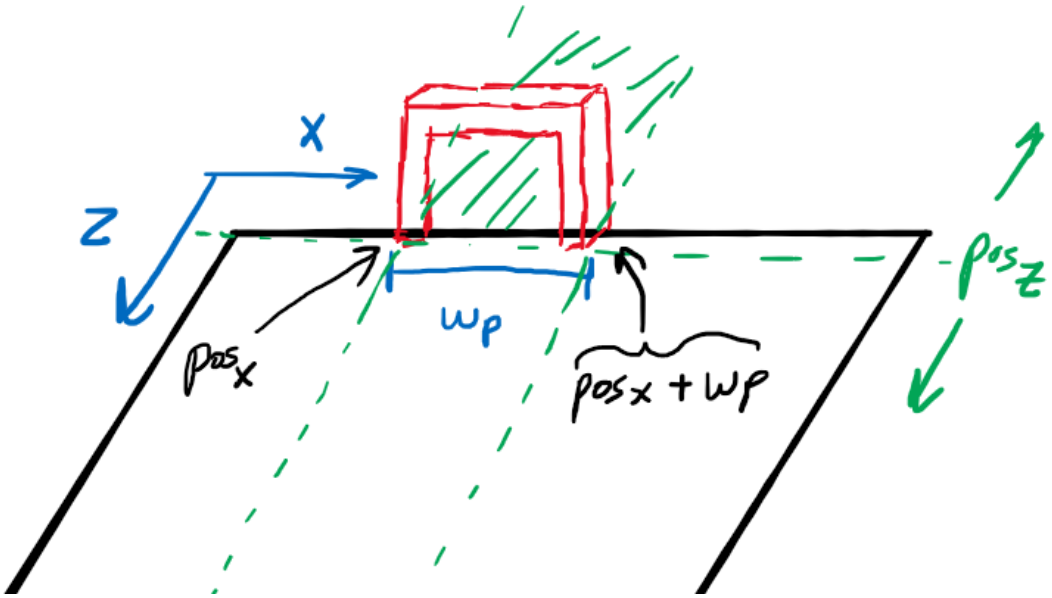
Así mismo, el coche también puede ser rotado mediante las flechas de control **Derecha** e **Izquierda**.

La velocidad del coche ha sido **limitada**, si se alcanza dicho límite, la velocidad se mantendrá estática y la aceleración se bloqueará a cero. Para crear un control más realista se ha añadido un efecto de **rozamiento**, es decir, si el usuario no acelera, la velocidad irá disminuyendo poco a poco hasta que el coche se detenga. El rozamiento se calcula como el 20% de la velocidad y esta se aplicará en sentido opuesto.

Por otra parte, también se debe controlar que el coche no sobrepase los límites del campo, puesto que no existe velocidad en el eje de las ordenadas, no tendremos que controlar que el coche no sobrepase el suelo. Con las paredes es diferente, tendremos que asegurarnos que el coche no sobrepasa los límites del campo, para eso usamos el radio de colisión establecido anteriormente para controlar el choque con la pelota.

Detección de Gol

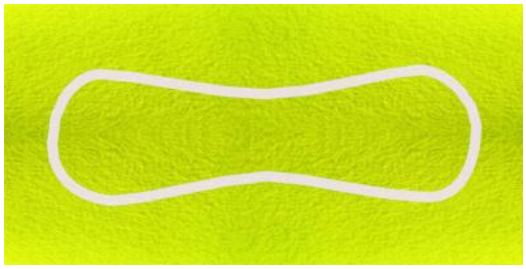
El juego termina en cuanto se consigue marcar un gol, la detección de este suceso se consigue comprobando que el balón se encuentra pasando la posición de la portería en eje Z y que se encuentra entre los dos palos, es decir, entre la posición de la portería en el eje X y la posición de la portería en el eje X más el ancho de la portería.



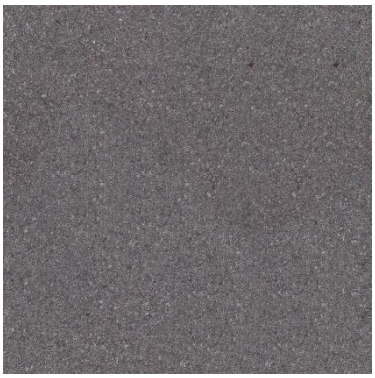
Si se detecta un gol, la aplicación sale del bucle principal de detección de eventos y por lo tanto el juego finaliza.

Texturas

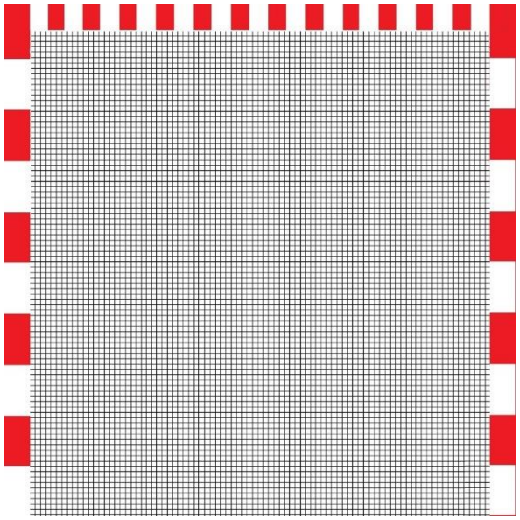
La mayoría de las texturas utilizadas para la aplicación las hemos sacado de los materiales aportados junto al enunciado de la práctica.



Textura de la pelota

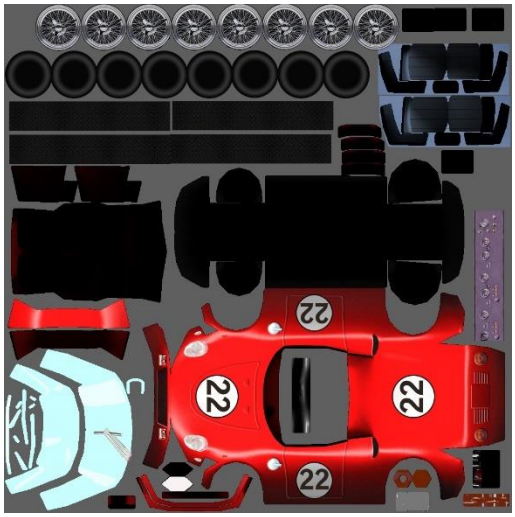


Textura de las paredes



Textura de la portería

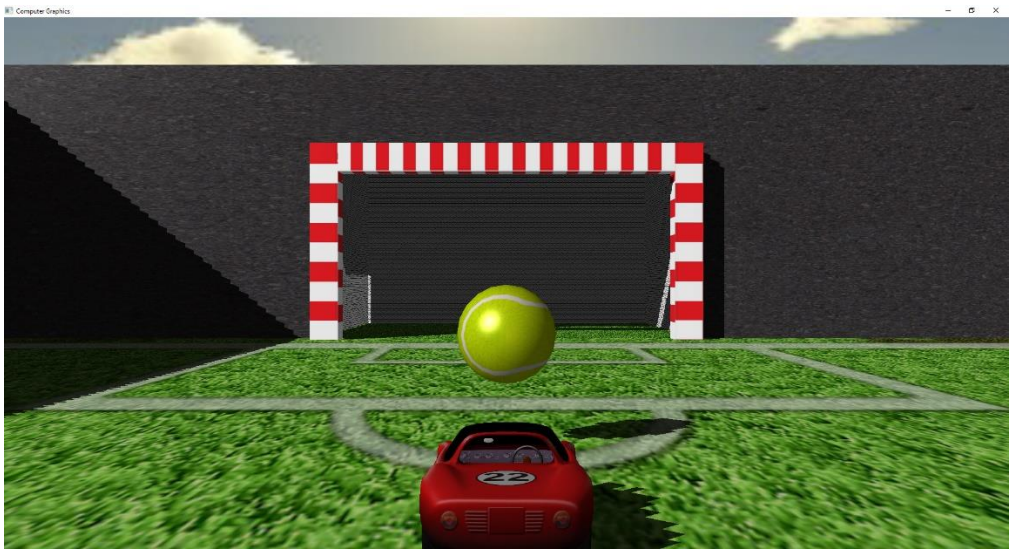
Textura del campo

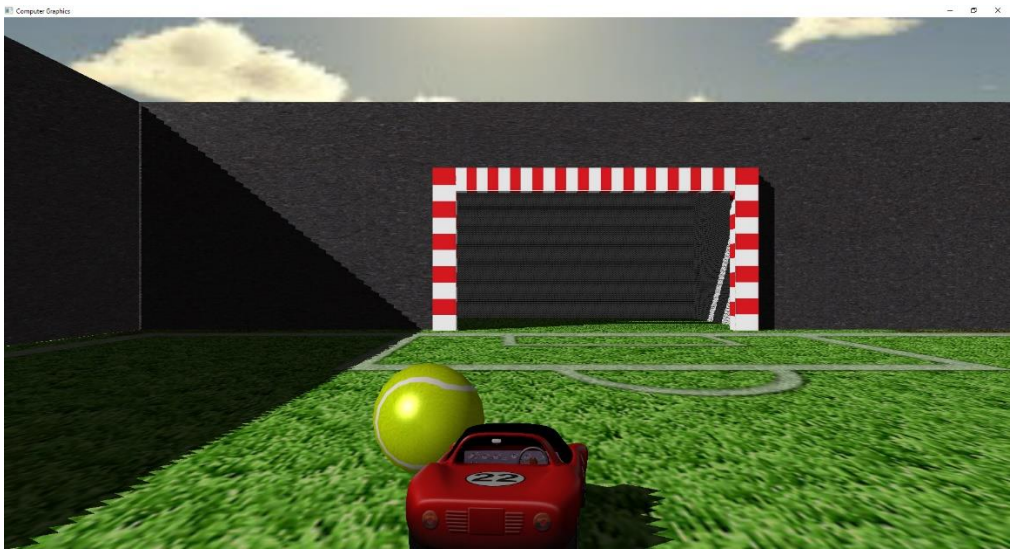


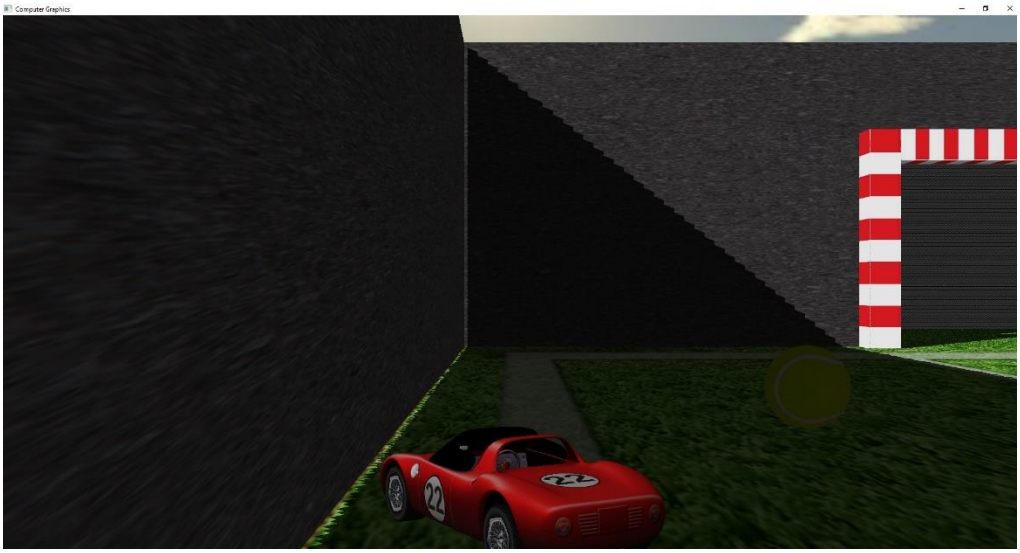
Textura del coche

Pruebas sobre la aplicación

Para comprobar que la aplicación funciona de forma correcta, hemos estado haciendo algunas pruebas.







La aplicación tiene algunos bugs, a veces cuando la pelota choca con las paredes, esta la atraviesa y desaparece, por el resto va bastante bien, la pelota se mueve al recibir el impacto del coche, se detectan los goles sin problema y si el coche choca con alguna pared este rebota.