
# Práctica 3-PA-Memoria

**Diseño y Desarrollo de Videojuegos**
**Programación Avanzada**
**Curso: 2º (2021-2022)**

**Juan Higuero López**
**Ivan Pérez Ciruelos**

**Fecha entrega: 10/1/2022**

# ÍNDICE

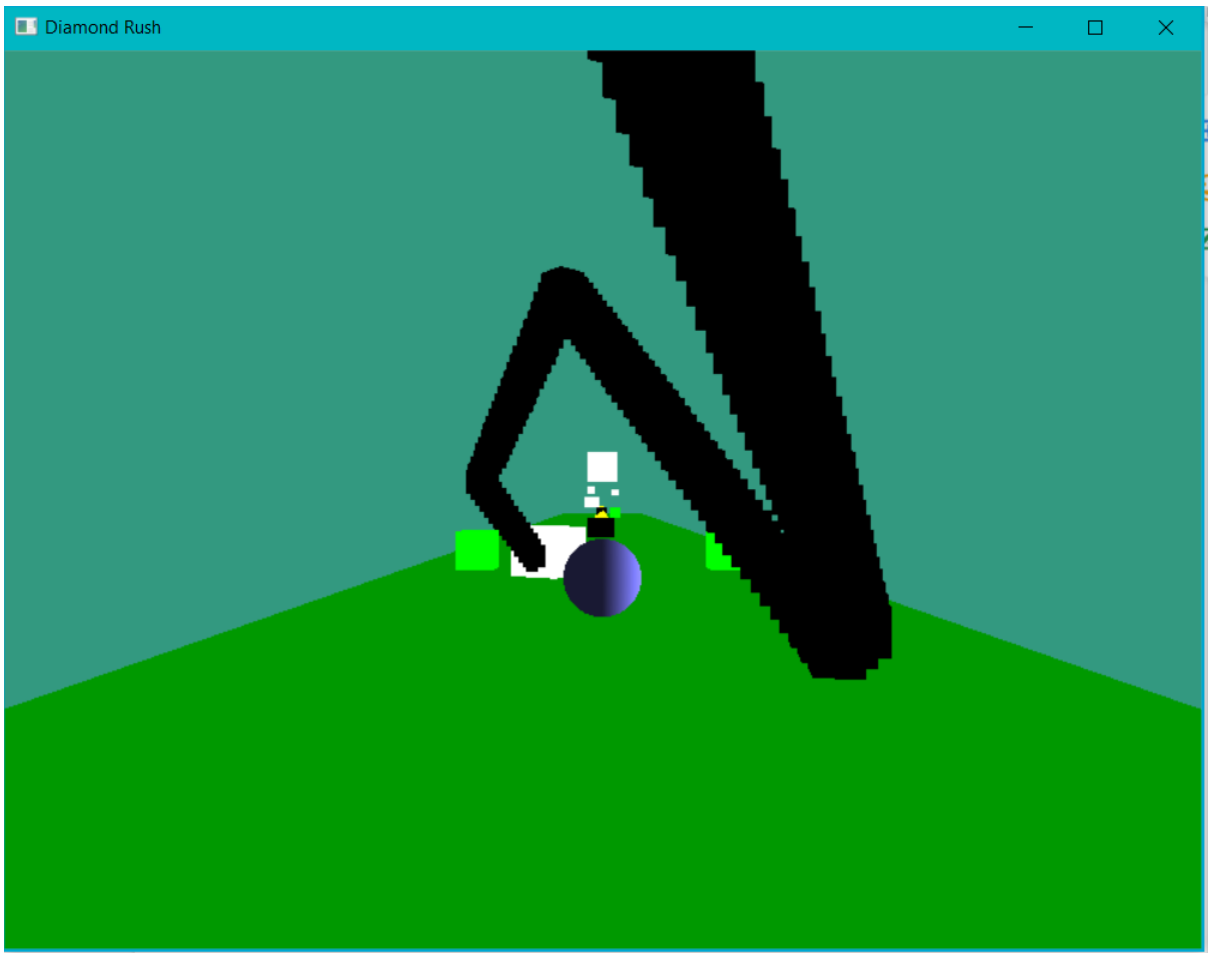
# 1. Base del juego

La base del juego radica en la idea de un juego de avanzar sin parar, en el que tienes que evitar los obstáculos generados a lo largo del escenario.

Empleando el motor de juego aprendido en clase, y, a partir de este, se han creado diferentes clases para poder llevar a cabo la implementación de los diferentes objetos que aparecen en el juego.

Para la creación de estas clases se ha optado por crear clases heredadas de las clases más genéricas albergadas en el motor de juego (Clase solid y sus clases heredadas, tales como, cube, cuboid o sphere). En estas clases heredadas se han añadido las nuevas funcionalidades que nos han permitido crear obstáculos de diferentes tipos además de un jugador y power ups.

Una vez creados todos los obstáculos se ha implementado una nueva cámara dinámica y una detección de colisiones para darle el aspecto de videojuego y que el jugador sienta que sus acciones tienen consecuencias gracias al procesamiento de teclas y detección de colisiones.



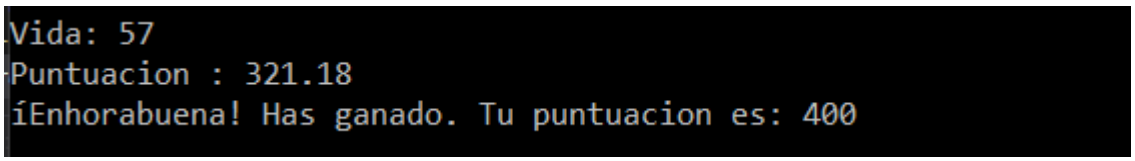
# 2. Condición de vitoria

La condición de victoria del juego consiste en llegar a una meta creada al final del nivel. Esta meta ha sido creada con dos cilindros colocados a cada lado del escenario y un rectángulo a modo de bandera que los une.

Al conseguir la condición de victoria (cruzar la meta), cambiaremos automáticamente de pantalla (y escena), indicando al jugador que ha completado el nivel.

Además, cuando se consigue la victoria, se muestra por la consola un aspecto similar al que se muestra en la nueva escena conseguida en el juego. En la consola podremos observar la puntuación obtenida por el usuario.

Además, se crea un fichero txt con el mensaje de victoria y la puntuación.



# 3. Condición de derrota

La condición de derrota del juego consiste en perder la toda la vida mediante una detección de colisiones del objeto principal, la esfera, con los diferentes obstáculos que hay en el nivel.

Al igual que sucede con la condición victoria, cuando logramos la condición de derrota, cambiamos automáticamente de pantalla (y escena), indicando al jugador que ha perdido y dándole las mismas opciones de la condición de victoria. También aparecerá por consola un mensaje de derrota.

A parte de esta nueva escena, al chocar con algún objeto, se muestra por la consola del juego cual ha sido nuestra puntuación en el juego.

Además, se crea un fichero .txt con el mensaje de derrota y la puntuación.

```
Vida: 0
Puntuacion : 150.94
Has perdido. Tu puntuacion es: 150
```

# 4. Diseño de detección de colisiones

Para detectar las colisiones entre el jugador y los obstáculos y los power ups hemos usado un método distance implementado clase Vector3D, el cual calcula la distancia entre dos puntos (en nuestro caso entre el jugador y los obstáculos y power ups). Para facilitar este proceso guardamos los obstáculos en un vector y comparamos la distancia entre el jugador y cada componente del vector mediante un bucle for. Realizamos el mismo proceso con los power ups.

```
//colisiones con los obstaculos
 if (this->jugadores[0]->getVida() > 0) {
     for (int i = 0; i < obs.size(); i++) {
         if((this->jugadores[0]->getPos().distance(
               this->obs[i]->getPos()) < 1.5)) {
             puntuacionActual = this->jugadores[0]->getPuntuacion();
             resta = this->obs[i]->getPuntuacionResta();
             nuevaPuntuacion = puntuacionActual - resta;
             this->jugadores[0]->setPuntuacion(nuevaPuntuacion);
             vidaNueva = this->jugadores[0]->getVida() -
               this->obs[i]->getVidaResta();
             this->jugadores[0]->setVida(vidaNueva);
             this->escenario1->ColorSegunVida(this->jugadores[0]
               ->getVida());
             Text* mensajePuntos = obs[i]->mensajePuntuacion();
             this->activeScene->addGameObjects(mensajePuntos);
             Text* mensajeVida = obs[i]->mensajeVida();
             this->activeScene->addGameObjects(mensajeVida);
             cout << "Vida: " << this->jugadores[0]->getVida() << endl;
             cout << "Puntuacion : " <<
               this->jugadores[0]->getPuntuacion() << endl;
         }

     }
 }
 int suma;
 //colisiones con los powerUps
 for (int i = 0; i < PUs.size(); i++) {
     if ((this->jugadores[0]->getPos().distance(this->PUs[i]->getPos())
             < 1.5)) {
         puntuacionActual = this->jugadores[0]->getPuntuacion();
         suma = this->PUs[i]->getPuntuacionSuma();
         nuevaPuntuacion = puntuacionActual + suma;
         this->jugadores[0]->setPuntuacion(nuevaPuntuacion);
         vidaNueva = this->jugadores[0]->getVida() +
               this->PUs[i]->getVidaSuma();
         this->jugadores[0]->setVida(vidaNueva);
         this->escenario1->ColorSegunVida(this->jugadores[0]->getVida())
         Text* mensajePuntos = PUs[i]->mensajePuntuacion();
         this->activeScene->addGameObjects(mensajePuntos);
```

```
            Text* mensajeVida = PUs[i]->mensajeVida();
            this->activeScene->addGameObjects(mensajeVida);
            cout << "Vida: " << this->jugadores[0]->getVida() << endl;
            cout << "Puntuacion : " << this->jugadores[0]->getPuntuacion()
                  << endl;
        }
    }
```

# 5. Diseño de nivel

Para el diseño de nivel se ha optado por un diseño sencillo, teniendo un nivel compuesto por un escenario, un personaje, los diferentes tipos de obstáculos, una luz y la cámara.

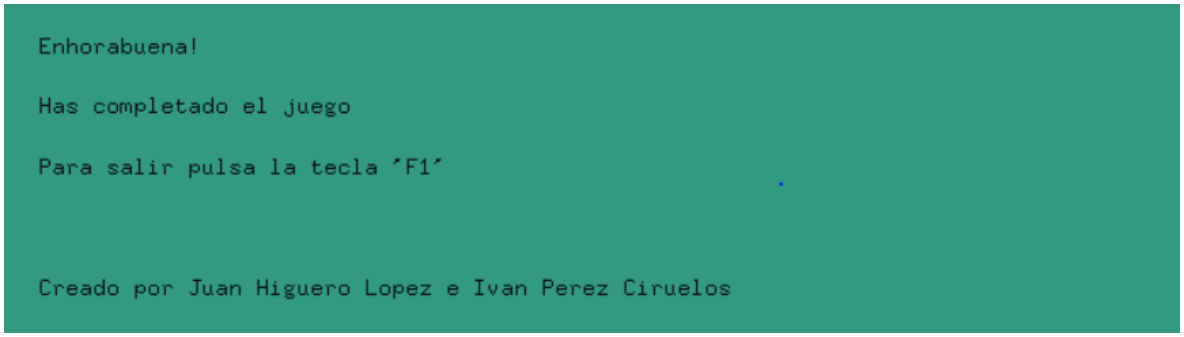
Para la creación del escenario se ha creado otra clase la cual se encarga de crear un escenario genérico que es llamado desde game para poder implementarlo en el juego, este escenario está compuesto por un cuboid personalizado que realiza la función de suelo. A modo de meta se han colocado dos cilindros y un cuboid al final de todos los obstáculos para indicar la condición de victoria y el final de nivel.

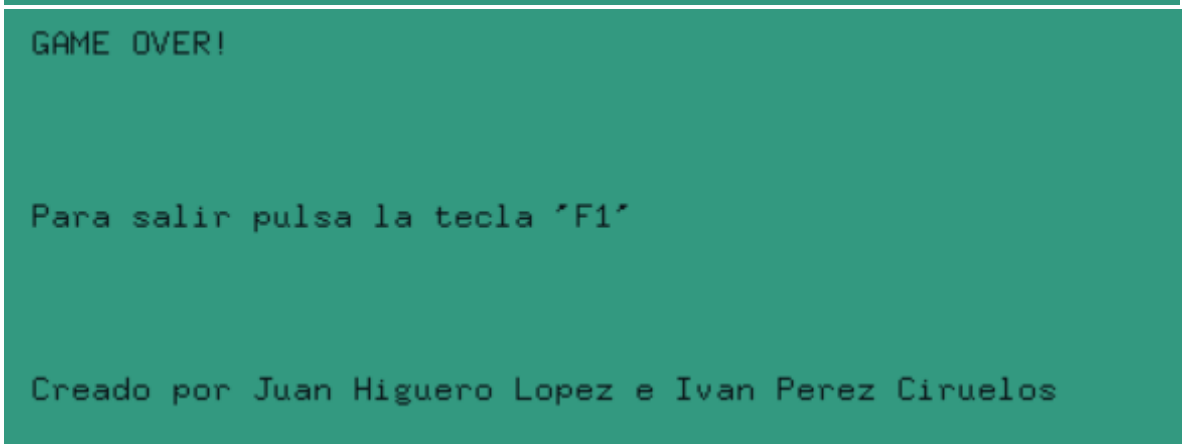
Por otro lado, se ha creado una esfera (jugador) que avanza sin parar a una velocidad determinada y desplazándose de izquierda a derecha según vaya rebotando con los laterales del escenario, aunque este movimiento puede ser interrumpido mediante las teclas ‘A’ y ‘D’.

```
void Game::Init() {
 // Luz
 Light* light = new Light(Vector3D(1, 0, 0));
 mainScene->addGameObjects(light);

 cam->setPos(Vector3D(0, 3, 0));
 mainScene->addGameObjects(cam);

 // Jugador
 player = new Jugador();
 mainScene->addGameObjects(player);
 this->jugadores.push_back(player);

 // Escenario
 escenario1 = new Escenario();
 escenario1->CrearArcoMeta(-400, mainScene);
 mainScene->addGameObjects(escenario1);

 //Crear obstáculos genéricos
 Obstaculos* obsGen1 = new Obstaculos();
 obsGen1->setPos(Vector3D(10, 1, -50));
 mainScene->addGameObjects(obsGen1);
 this->obs.push_back(obsGen1);
```

# 6. Diseño de menús

Para crear los menús se ha creado un ‘cuboid’ y varias líneas que te muestran información de la partida.



# 7. Diseño de cámaras - clase cámara

Para el uso de cámaras primero hemos creado una nueva clase llamada “CamaraJuego”, a la cual se le puede poner una velocidad determinada y tiene su propio método update para que se mueva automáticamente. La idea de esta cámara es ponerle la misma velocidad que al jugador para que le siga uniformemente.

```
class CamaraJuego:public Camera
{
    Vector3D vel = Vector3D(0, 0, 0);
public:
    CamaraJuego(Vector3D p = Vector3D(0, 0, 0)) : Camera() {
        setPos(p);
        setVel(vel);
    }
    void Update(double time);
    inline Vector3D getVel() { return vel; }
    inline void setVel(Vector3D v) { vel = v; }
};
```

# 8. Diseño de luces - clase light

Para el uso de luces en nuestra escena hemos creado una nueva clase llamada “Light”, heredera de la clase “Solid”. Esta nueva clase crea un foco de luz gracias al atributo “glLightfv”, pudiendo colocar dicha luz en cualquier ubicación de la escena.

```
class Light : public Solid {
public:
    Light(Vector3D p = Vector3D(50, 150, 50)) : Solid(p) {}
    void Render() {
        float x = getPos().getX();
        float y = getPos().getY();
        float z = getPos().getZ();

        GLfloat lightpos[] = { x,y,z,0 };
        glLightfv(GL_LIGHT0, GL_POSITION, lightpos);
    }
};
```

# 9. Clases de obstáculos y PowerUp

Para crear los obstáculos hemos creado la clase “Obstáculos”, la cual hereda de la clase “Cube” y hace la misma función que la clase “Solid”, siendo la clase padre de todos los tipos de obstáculos. Aparte hemos creado las clases “Obstaculo1”, “Obstaculo2” y “Obstaculo3” las cuales heredan de la clase “Obstáculos” y cada una tiene unas características particulares aparte de las características generales que heredan de la clase “Obstáculos”.

Siguiendo la misma lógica hemos creado la clase “PowerUps” para crear los recolectables aunque en este caso solo hay un tipo que es el genérico.

```
class Obstaculos : public Cube
{
private:
    int puntuacionResta;
    float vidaResta;
public:
    Obstaculos(Color col = Color(0, 10, 0)) : Cube() { // Constructor
        this->setColor(col);
        this->setPos(Vector3D(0, 0, 0));
        this->setSpeed(Vector3D(0, 0, 0));
        this->setOrientationSpeed(Vector3D(0, 0, 0));
        this->setOrientation(Vector3D(0, 0, 0));
        this->SetSize(1.5);
        this->setPuntuacionResta(10);
        this->setVidaResta(0.001);
    }

    inline float getPuntuacionResta() const { return this->puntuacionResta;
}
    inline void setPuntuacionResta(const float& puntuacionRestaToSet) {
this->puntuacionResta = puntuacionRestaToSet; }
    float getVidaResta() const { return this->vidaResta; }
    inline void setVidaResta(const float& puntuacionToSet) { this-
>vidaResta = puntuacionToSet; }

    Text* mensajePuntuacion(Vector3D posJugador);
    Text* mensajeVida(Vector3D posJugador);
};



class PowerUps : public Cube
{
private:
    float puntuacionSuma;
    int vidaSuma;
public:
```

```
    PowerUps() : Cube() { // Constructor
        this->setColor(Color(10, 10, 0));
        this->setPos(Vector3D(3, 1, 3));
        this->setSpeed(Vector3D(0, 0, 0));
        this->setOrientationSpeed(Vector3D(-10, 20, 5));
        this->SetSize(1);
        this->setVidaSuma(1);
        this->setPuntuacionSuma(10);
    }

    float getPuntuacionSuma() const { return this->puntuacionSuma; }
    void setPuntuacionSuma(const float& puntuacionSumaToSet) { this-
>puntuacionSuma = puntuacionSumaToSet; }
    float getVidaSuma() const { return this->vidaSuma; }
    void setVidaSuma(const float& puntuacionToSet) { this->vidaSuma =
puntuacionToSet; }

    Text* mensajePuntuacion();
    Text* mensajeVida();
};
```

## 9.1 Clase Obstáculo 1

La clase Obstaculo 1 hereda de Obstaculo y permite la creación de nuevos obstáculos que interfieren con el objetivo de nuestro jugador, ganar el juego. Esta clase crea un cubo negro, que cambia de tamaño dependiendo del tamaño actual del mismo, creciendo y decreciendo progresivamente.

```
#include "Obstaculo1.h"
void Obstaculo1::Update() {
    if (tamaño == true) {
        this->SetSize(GetSize() + 0.01);
        if (this->GetSize() > 3) {
            tamaño = false;
        }
    }

    if (tamaño == false) {
        this->SetSize(GetSize() - 0.01);
        if (this->GetSize() < 0.5) {
            tamaño = true;
        }
    }
}
```

## 9.2 Clase Obstáculo 2

La clase obstaculo 2 hereda de Obstáculo y permite la creación de los obstáculos que interfieren con el objetivo de nuestro jugador, ganar el juego.  Esta clase posee una funcionalidad la cual consiste en realizar un rebote del obstáculo de forma que baje rápido y suba más lento,

```
void Obstaculo2::Update() {
    if (this->getPos().getY() > 7 && this->getSpeed().getY() > 0) {
        this->setSpeed(Vector3D(this->getSpeed().getX(), this-
>getSpeed().getY() * -3, this->getSpeed().getZ()));
    }

    if (this->getPos().getY() < 1 && this->getSpeed().getY() < 0) {
        this->setSpeed(Vector3D(this->getSpeed().getX(), this-
>getSpeed().getY() * -0.33, this->getSpeed().getZ()));
    }
}
```

## 9.3 Clase Obstáculo 3

La clase Obstaculo 3 hereda de Obstaculo y permite la creación de nuevos obstáculos que interfieren con el objetivo de nuestro jugador, ganar el juego. Esta clase crea un cubo blanco, que se mueve hacia arriba y abajo; y de izquierda a derecha continuamente, además, detecta cuándo el jugador se está acercando para comenzar a disparar una ráfaga continuada de obstáculos negros que el jugador debe esquivar.

```
Obstaculo3* Obstaculo3::Disparar(Scene* mainScene) {
    Obstaculo3* obs = new Obstaculo3();
    mainScene->addGameObjects(obs);
    obs->setPos(Vector3D(getPos().getX(), getPos().getY(),
getPos().getZ()));
    obs->setSpeed(Vector3D(0, 0, 5));
    obs->SetSize(1);
    obs->setColor(Color(0, 0, 0));
    return obs;
}
```

# 10. Clase escenario

La clase “Escenario” la hemos creado para facilitar la construcción de todo lo relacionado con el escenario del juego y elementos que no tienen ninguna propiedad especial. Tal y como está, instanciar la clase crea el suelo, pero tiene un método que crea la meta a una distancia determinada.

```
class Escenario : public Cuboid
{
public:
    Escenario(Color col = Color(0, 3, 0), float largo = 3000, float ancho =
20) : Cuboid() {
        this->setColor(col);
        this->setPos(Vector3D(5, -1, -20));
        this->setSpeed(Vector3D(0, 0, 0));
        this->setOrientationSpeed(Vector3D(0, 0, 0));
        this->setOrientation(Vector3D(0, 0, 0));
        this->setAltura(1);
        this->setAnchura(largo);
        this->setLongitud(ancho);
    }

    void CrearArcoMeta(double dist, Scene* mainScene);
    void ColorSegunVida(float vida);
};
```

# 11. Clase jugador

Para la creación de nuestro jugador hemos optado por crear una esfera, para ello hemos creado una clase Jugador que heredase de esfera para poder crearlo.

En esta nueva clase Jugador, además de crear la esfera, se añaden nuevas funcionalidades como vida y puntuación los cuales pueden ir disminuyendo y aumentando dependiendo de la recogida de power ups o la colisión con obstáculos.

Respecto a la vida, esta va a depender de las veces que colisionamos con los diferentes objetos que hay en el escenario (para reducir la vida) o con los diferentes power ups (para aumentar la vida).

Respecto a la puntuación, esta se calcula mediante la distancia que recorre el jugador antes de perder toda la vida, para ello se solicita continuamente la posición z del jugador (coordenada en la que se mueve) y se va sumando continuamente.

```
class Jugador : public Sphere
{
    double vida;
    double puntuacion;
public:
    Jugador(Color col = Color(0.5, 0.5, 1), int vida = 100, int puntuacion
= 0) :Sphere() { // Constructor
        this->setColor(col);
        this->setPos(Vector3D(5, 1, -15));
        this->setSpeed(Vector3D(0, 0, -2));
        this->setOrientationSpeed(Vector3D(0, 20, 0));
        this->setVida(vida);
        this->SetSize(1);
    }
    inline double getVida() { return vida; }
    inline void setVida(double v) { vida = v; }
    inline double getPuntuacion() { return -this->getPos().getZ(); }
    inline void setPuntuacion(double p) { puntuacion = p; }
    void Update(double dt);
    void CambiarDireccion(float d);
};
```

# 12. Diagrama de clases UML