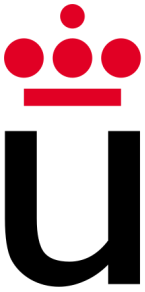

Universidad
Rey Juan Carlos

Práctica Final:

Desarrollo de un videojuego

Diseño y Desarrollo de Videojuegos

Rubén Díaz Borrego
Félix Vilches Guzmán

ÍNDICE

1. Introducción

Esta memoria tiene como finalidad la documentación de todo el proyecto sobre la práctica final: desarrollo de un videojuego aplicando programación orientada a objetos. Para esta práctica, se ha partido del motor proporcionado en clase, y se ha realizado un juego que trata sobre esquivar obstáculos mientras se avanza por un escenario lineal. El jugador puede moverse hacia los lados y esquivar dichos obstáculos, así como recoger algunos objetos que darán ciertas bonificaciones. En caso de chocar con un obstáculo, se acabará la partida. Se puede almacenar un nombre de usuario y, tras ver un ranking, se podrá iniciar una partida nueva. A continuación, se detallarán todas las clases introducidas.

2. Nuevas funcionalidades

Para la realización de esta práctica, se ha partido del motor de juego creado en las anteriores prácticas. Se han añadido varias funcionalidades nuevas, para conseguir que el videojuego funcione correctamente. Dichas funcionalidades se han añadido creando clases nuevas o modificando las ya existentes, como la clase Main.

Clases nuevas:

- Jugador
- Ranking
- PowerUp
- CamaraFija
- ObstaculoMeteorito
- ObstaculoBasuraEspacial
- EscenaJuego (hereda de Scene)

3. Clase Jugador

Esta clase se encarga de gestionar el jugador en el videojuego. En el encabezado, se definen las variables a utilizar y se carga el modelo del jugador. Se le asignan varios parámetros, así como su velocidad, y se le desactiva la gravedad.

En el cuerpo (Jugador.cpp) se establecen los métodos. Mediante una función, se asignan las teclas necesarias para su movimiento a izquierda y derecha (en este caso A para izquierda y D para derecha), al igual que se establecen unos límites para que no se pueda salir por los bordes de la zona de juego.

```
void Jugador::ProcessKeyPressed(unsigned char key, int px, int py) {
	switch (key)
	{
	case 'a':
	case 'A':
		if (this->getPos().GetX() > -1.0f) {   //para no salirse de los bordes
			this->setP(Vector3D(this->getPos().GetX() - velocidad, 0.0f, 10.0f));

		}
		break;

	case 'd':
	case 'D':
		if (this->getPos().GetX() < 10.0f) {
			this->setP(Vector3D(this->getPos().GetX() + velocidad, 0.0f, 10.0f));

		}
		break;

	default:
		break;
	}

}
```

4. Clase Ranking

Esta clase se encarga de mostrar un ranking al final de la partida, después de que el jugador introduzca su nombre. En dicho ranking se mostrará el nombre del jugador junto con su puntuación.

En Ranking.h, se gestionan las diferentes variables, entre las que se incluyen dos vectores, para puntos y nombres.

En el cuerpo, hay varias funciones para que el ranking se muestre correctamente. En la función agregar, se agregan el nombre y los puntos en un fichero.

En la función addToVector, se lee el fichero y se introducen los valores a un vector de strings (nombres) y a uno de doubles (puntos).

Por otra parte, en la función eliminarMinVec, se obtiene el elemento con la puntuación más baja y se elimina. Para ello, dentro de un bucle for, se va intercambiando el primer valor del array por el actual, en caso de que el actual sea menor:

```
void Ranking::eliminarMinVec() {

	for (int i = 0; i < ptsVector.size(); i++)
	{
		if (ptsVector[i] < ptsVector[0]) {
			swap(ptsVector[0], ptsVector[i]);
			swap(nombres[0], nombres[i]);
		}
	}
	ptsVector.erase(ptsVector.begin());
	nombres.erase(nombres.begin());
}
```

En la función ordenarRanking, se ordenan los vectores de mayor a menor, para mostrar primero al jugador con más puntos.

Posteriormente, en agregarFichero, se meten los elementos de los vectores en el fichero. Los elementos que se añaden sobreescriben los que había anteriormente.

Se utiliza la función visualizar para leer el fichero, y la función crearRanking para usar los métodos anteriores ordenados para crear el ranking. Aquí, además, se añade una condición para no mostrar más de 5 jugadores en el ranking. En caso de que haya más de 5, se elimina el que tenga menos puntos.

5. Clase PowerUp

La clase PowerUp consiste en la carga de un objeto que funcionará añadiendo una bonificación al jugador, en caso de que lo recoja. No colisionará como el resto de obstáculos, sino que el jugador pasará por encima y este objeto desaparecerá. Al recogerlo, el jugador aumentará su velocidad de movimiento hacia los lados, haciendo que sea más fácil esquivar obstáculos.

En PowerUP.h, se hace la carga de un objeto, se le asigna un color y se pone su gravedad en falso. El resto de la funcionalidad se implementa en EscenaJuego.

```
class PowerUP:public Model
{
public:
	PowerUP() {

		ModelLoader* loader = new ModelLoader();
		loader->LoadModel("..\\3dModels\\Star.obj");
		this->triangles = loader->getModelo().triangles;
		this->setC(Color(0, 120, 0));
		this->SetIsAffectedByGravity(false);
	}
};
```

6. Clase CamaraFija

Esta clase sirve para establecer la cámara de juego. Será posible cambiar de cámara durante el juego, pulsando alguna tecla determinada, que se indicará a continuación.

En CamaraFija.h, se crea el constructor y se establecen los elementos que serán utilizados en el cuerpo.

En el cpp, en la función Render, se establece la posición y orientación inicial de la cámara fija. Por otra parte, en la función ProcessKeyPressed, se asigna una tecla a cada cámara de la escena, de forma que al pulsarlas, sea posible cambiar entre ellas:

- Cámara frontal: tecla “F”.
- Cámara aérea: tecla “C”.
- Cámara lateral: tecla “L”.

Se puede cambiar entre las distintas cámaras en cualquier momento de la partida.

```
void CamaraFija::Render() {
	glRotatef(getOri().GetX(), 1.0, 0.0, 0.0);
	glRotatef(getOri().GetY(), 0.0, 1.0, 0.0);
	glRotatef(getOri().GetZ(), 0.0, 0.0, 1.0);
	glTranslatef(-1 * this->getPos().GetX(), -1 * this->getPos().GetY(), -1 * this->getPos().GetZ());
}

void CamaraFija::ProcessKeyPressed(unsigned char key, int px, int py)
{
	switch (key) {
	case 'f':
	case 'F':
		this->setP(Vector3D(5.0f, 10.0f, 20.0f)); //camara frontal
		this->setOri(Vector3D(20.0f, 0.0f, 0.0f));
		break;

	case 'l':
	case 'L':
		this->setP(Vector3D(5.0f, 30.0f, -6.0f)); //camara Lateral
		this->setOri(Vector3D(90.0f, -90.0f, 0.0f));
		break;

	case 'c':
	case 'C':
		this->setP(Vector3D(5.0f, 30.0f, 1.0f)); //camara Aerea
		this->setOri(Vector3D(90.0f, 0.0f, 0.0f));
		break;
}}
```

7. Obstáculos

Hay dos tipos de obstáculos diferentes: basura espacial y meteoritos. Su implementación es muy similar. Los meteoritos heredan de la clase Sphere, mientras que la basura especial hereda de la clase Cylinder.

En ObstaculoMeteorito.h y ObstaculoBasuraEspacial.h, se crea el constructor y se pasan los parámetros a cada objeto. De forma similar al power up, estos objetos se crearán en EscenaJuego.

```
class ObstaculoMeteorito: public Sphere
{
public:
	ObstaculoMeteorito(){}
	ObstaculoMeteorito(Vector3D pos, Color col, Vector3D vel, Vector3D rot):
		Sphere(pos, col, vel, rot){}

};

class ObstaculoBasuraEspacial: public Cylinder
{

	public:
		ObstaculoBasuraEspacial() {}
		ObstaculoBasuraEspacial(Vector3D pos, Color col, Vector3D vel, Vector3D rot, float rdB =
0.5, float rdPS = 1.5, float al = 1, float sli = 5, float sla = 5) :
			Cylinder(pos, col, vel, rot) {}
	};
```

8. EscenaJuego

Esta clase se ha creado utilizando como base la clase “Scene”, que se ha utilizado en las prácticas anteriores.

En su EscenaJuego.h, se han declarado diversas variables, así como su constructor. Se crean el mapa, los obstáculos, el jugador y los power ups. Posteriormente, se crean los elementos de la escena y los elementos de progreso del juego.

En el cpp, se utilizan varias funciones. ClearScene se encarga de borrar la escena. Las funciones de add sirven para añadir los elementos al juego, como son los obstáculos y el jugador. Mediante un push_back, se añaden los elementos a la lista de gameObjects. En CrearEscenario, se añaden los elementos sobre los que se encontrarán los gameObjects, además de añadir los bordes izquierdo y derecho. En la función crearObstaculos, se añaden tanto basura espacial como meteoritos, para dar un poco de variedad a los obstáculos que se encuentran:

```
void EscenaJuego::crearObstaculos()
{

	ObstaculoBasuraEspacial* basura = new(nothrow) ObstaculoBasuraEspacial[numeroObst];
	if (basura != nullptr) {
		for (int index = 1; index < numeroObst; index++) {
			basura[index] = ObstaculoBasuraEspacial(
				Vector3D((rand() % 13), (0.5), -25),
				Color(120.0f, 1.0f, 1.0f),
				Vector3D((0.07 + index * 0.001), (0), (0.6 * index)),
				0.1 + 2 * 0.1
			);
			basura[index].SetIsAffectedByGravity(false);
			this->addObstaculos(basura + index);
		}
	}

	ObstaculoMeteorito* meteoritos = new(nothrow) ObstaculoMeteorito[numeroObst];
	if (meteoritos != nullptr) {
		for (int index = 1; index < numeroObst; index++) {
			meteoritos[index] = ObstaculoMeteorito(
				Vector3D((rand() % 13), (0.8), -25),
				Color((120), (0), (120)),
				Vector3D((0.07 + index * 0.001), (0), (0.75 * index)),
				0.1 + 2 * 0.1
			);
			meteoritos[index].SetIsAffectedByGravity(false);
			this->addObstaculos(meteoritos + index);
		}}}
```

De forma similar se añaden tanto los power ups como el jugador, en sus respectivas funciones.

```
void EscenaJuego::crearPowerUps() {

	PowerUP* powerUp = new PowerUP();
	powerUp->setP(Vector3D(6.0f, -4.2f, -25.0f));
	powerUp->setOri(Vector3D(0.0f, 20.0f, 0.0f));
	this->setPowerUP(powerUp);
}

void EscenaJuego::crearJugador() {

	Jugador* jugador = new Jugador();
	jugador->setP(Vector3D(5.0f, 0.0f, 10.0f));
	jugador->setVel(Vector3D(0.45f, 0, 0));
	this->setJugador(*jugador);
}
```

En Render, se recorren los gameObjects y se van renderizando por pantalla. En la función Update, se van actualizando los elementos necesarios. Se ha utilizado para hacer diversas comprobaciones constantemente, como las colisiones con los obstáculos o que estos no se salgan de los bordes laterales. También se controla aquí la condición de derrota: si el jugador pierde, se muestra el ranking y se reinicia la partida.

En la función spawnearPowerUP(), se hace que aparezca el anillo power up a los 100 puntos. Este anillo nos aumenta la velocidad de movimiento.

También se han tenido en cuenta las colisiones, con las funciones comprobarColisisionObstaculos() y comprobarColisionPowerUp(). Funcionan para las comprobaciones tanto con los obstáculos como con el power up. La manera de implementar las colisiones para los obstáculos es la siguiente:

```
void EscenaJuego::comprobarColisionObstaculos() {
	for (int idx = 0; idx < this->obstaculos.size(); idx++)
	{
		if (obstaculos[idx]->getPos().GetZ() < 11.7) {
			bool collisionX = abs(obstaculos[idx]->getPosX() - this->jugador.getPosX()) < jugador.getColisionIzq();
```

```
                bool collisionZ = abs(obstaculos[idx]->getPosZ() - this->jugador.getPosZ()) <
jugador.getColisionDer();

                if (collisionX && collisionZ) {
                        this->jugador.setVel(Vector3D(0, 0, 0));
                        for (int idx = 0; idx < this->obstaculos.size(); idx++)
                        {
                                this->obstaculos[idx]->setVel(Vector3D(0, 0, 0));
                        }
                        derrota = true;
                }}}}
```

Colisionar con un obstáculo es una condición de derrota, por lo que al hacerlo, se pasa el boolean derrota a true y se termina el juego.
Para evitar que los obstáculos se salgan por los lados del mapa, se utiliza la función limitesObstaculos. Además, para que sigan apareciendo obstáculos continuamente mientras se juega, se les devuelve al principio del mapa una vez que llegna al final, de forma que nunca dejan de aparecer.

```
void EscenaJuego::limitesObstaculos(const float& time) {

        for (int idx = 0; idx < this->obstaculos.size(); idx++)
        {
                this->obstaculos[idx]->Update(time, this->GetGravity());
                if (this->obstaculos[idx]->getPos().GetX() > this->GetSize().GetX()
                        || this->obstaculos[idx]->getPos().GetX() < 0) {

                        this->obstaculos[idx]->setVel(Vector3D(this->obstaculos[idx]->getVel().GetX() * -1,
                                this->obstaculos[idx]->getVel().GetY(),
this->obstaculos[idx]->getVel().GetZ()));
                }
                if (this->obstaculos[idx]->getPos().GetY() > this->GetSize().GetY() ||
                        this->obstaculos[idx]->getPos().GetY() < 0) {

                        this->obstaculos[idx]->setVel(Vector3D(this->obstaculos[idx]->getVel().GetX(),
                                this->obstaculos[idx]->getVel().GetY()                  *                  -1,
this->obstaculos[idx]->getVel().GetZ()));
                }
                if (this->obstaculos[idx]->getPos().GetZ() > 20 ||
                        this->obstaculos[idx]->getPos().GetZ() < -30) {
                        this->obstaculos[idx]->setP(Vector3D((rand() % 13), (0.5), -25));
                }
        }
}
```

9. UML

Se adjunta al final de este documento. Adicionalmente, también puede revisarse en el siguiente enlace:

- https://lucid.app/lucidchart/2ce2d424-dc6f-4245-8aa8-272d12277bea/edit?viewport_loc=-1326%2C666%2C7168%2C3256%2CHWEp-vi-RSFO&invitationId=inv_32c0ed06-b3d3-4228-9b7e-2bc2d573bd54#

Conclusiones

Esta práctica nos ha servido para aprender a crear un videojuego desde cero, incluyendo el motor de juego. Nos ha parecido que es especialmente útil si se quiere tener un control total de lo que se está haciendo, ya que tener acceso al motor y conocer bien cómo se ha creado, permite añadir nuevas funcionalidades al videojuego de manera muy cómoda y accesible. Hemos notado una gran diferencia con respecto al uso de otros motores, en los que cambiar su funcionamiento nos resultaría prácticamente imposible. A pesar de sus limitaciones, consideramos que es de utilidad conocer cómo funciona la creación de un videojuego desde un nivel tan bajo.

Bibliografía

- Learn cpp https://www.learncpp.com/
- C plus plus http://www.cplusplus.com/doc/tutorial/

UML Practica PA