Práctica 3

Programación Avanzada

"Shooting Star"

Universidad Rey Juan Carlos

Luis Javier Moreno Rojas
Lucía Álvarez Blázquez
Isabel Sánchez Benito

Índice

Introducción 4
Creación del juego 4
Clases usadas existentes 4
Clases propias 5
Clase entidad 5
Clase jugador 6
Clase enemigo1 7
Clase enemigo 2 8
Clase torreta 8
Clase Scene0 9
Clase Scene1 10
Clase Scene2 10
Clase Scene3 10
Mecánicas del juego y controles 10
Niveles 11
UML y Plantilla de requisitos 13
Conclusión y preguntas 15
Bibliografía 17

Tabla de ilustraciones

Ilustración 1: Clases existentes usadas 4
Ilustración 2: Clases nuevas 5
Ilustración 3: Imagen del jugador en función de su vida 6
Ilustración 4: Enemigo 1 7
Ilustración 5: Enemigo 2 8
Ilustración 6: Enemigo torreta 9
Ilustración 7: Bala 11
Ilustración 8: Ejemplo mecánica 11
Ilustración 9: Nivel 1 11
Ilustración 10: Nivel 2 12
Ilustración 11: Nivel 3 12
Ilustración 12: UML de las clases base 14
Ilustración 13: UML de clases propias 14
Ilustración 14: UML de los modelos 15
Ilustración 15: UML de Scene 15

Introducción

En esta práctica se mostrará el desarrollo paso a paso de la creación de un videojuego propio usando un motor ya proporcionado y aplicando como base los conceptos ya vistos en clase, siguiendo el paradigma de la programación orientada a objetos.

Para la creación de nuestro juego se han usado las herramientas de Visual Studio en lenguaje C++ y varias clases y objetos previamente creados (además de otros nuevos).

El juego se llama Shooting star, y es un juego arcade del género Shoot`em up en el que el jugador tendrá que superar una serie de niveles antes de ser derrotados para poder ganar. Cada nivel está formado por una pantalla en la que aparecen varios enemigos a los cuales tendremos que derrotar para pasar de nivel. El jugador cuenta con 3 vidas.

Creación del juego

Clases usadas existentes

Estas son las clases usadas con las que ya se contaba:

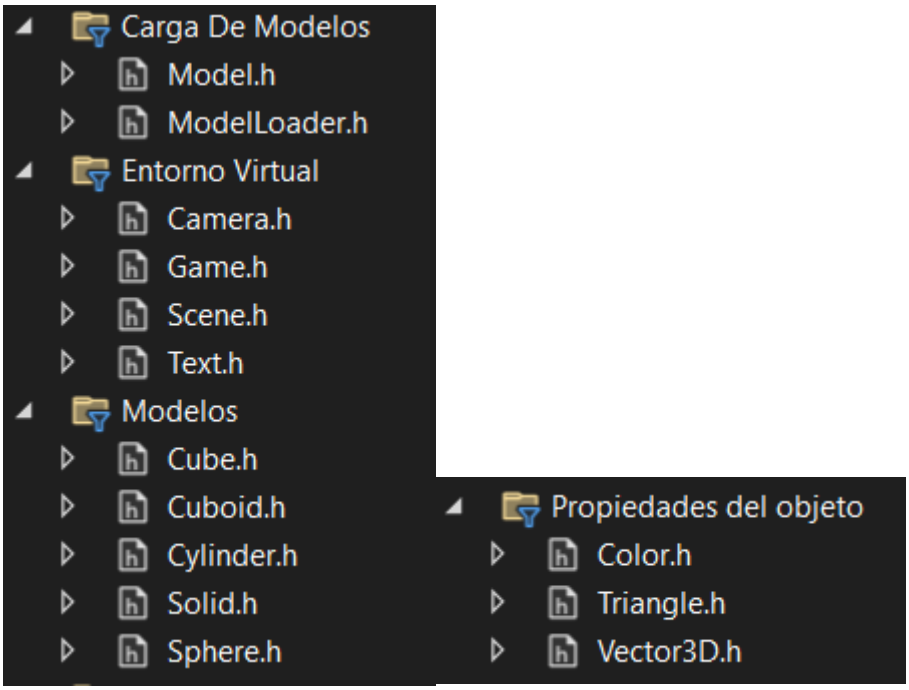

Ilustración 1: Clases existentes usadas

Como se puede observar están divididas en función de su funcionalidad del proyecto, desde la carga de modelos hasta propiedades del objeto como color o vector.

Clases propias

Se han implementado 5 clases propias las cuales se explicarán a continuación.

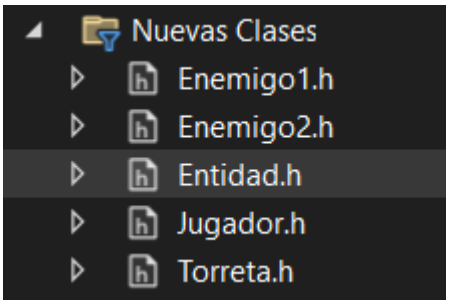

Ilustración 2: Clases nuevas

Clase entidad

En primer lugar, se ha implementado la clase entidad, la cual servirá como base de todas las entidades de nuestro juego, por lo que todas las demás clases heredarán de ella.

Esta clase cuenta con 3 atributos, (vida, daño y posición) y con varios métodos abstractos que se definirán sus clases hijas.

```
class Entidad
{
private:
	int vida;
	int dano;
	Vector3D posicion;


public:
	Entidad() : vida(1), dano(1), posicion(Vector3D(1,1,1)) {}

	Entidad(int vida, int dano, Vector3D posicion) : vida(vida), dano(dano), posicion(posicion) {}

	inline int GetVida() const { return this->vida; }
	inline int GetDano() const { return this->dano; }
	inline Vector3D GetPosicion() const { return this->posicion; }


	inline void SetVida(const int& vidaToSet) { this->vida = vidaToSet; }
	inline void SetDano(const int& danoToSet) { this->dano = danoToSet; }
	inline void SetPosicion(const Vector3D& posicionToSet) { this->posicion = posicionToSet; }


	virtual void Desplazamiento(const float& timeIncrement) = 0;

	virtual bool Colision(Cylinder* Bala) = 0;

	virtual void Render() = 0;

	virtual void Update(const float& timeIncrement) = 0;

};
```

Clase jugador

Se ha implementado la clase jugador, que como su nombre indica es la que usaremos para crear a nuestro personaje principal. Esta clase cuenta con un modelo propio de una estrella y con varios atributos como posición de disparo o velocidad de ataque y movimiento.

```
class Jugador : public Entidad
{
private:

	Vector3D posDisparo;
	float velocidadAtaque;
	float velocidadMovimiento;
	Model modelo;

public:
	Jugador() : Entidad(3, 1, Vector3D(1, 1, 1)), posDisparo(Vector3D(6, 1, 5)), velocidadAtaque(0.1),
		velocidadMovimiento(0.5){}
	Jugador( Vector3D posDisparo, float velocidadAtaque, float velocidadMovimiento) : posDisparo(posDisparo),
		velocidadAtaque(velocidadAtaque),
velocidadMovimiento(velocidadMovimiento) {}

	inline Vector3D GetposDisparo() const { return this->posDisparo; }
	inline float GetVelocidadAtaque() const { return this->velocidadAtaque; }
	inline float GetVelocidadMovimiento() const { return this->velocidadMovimiento; }
	inline Model GetModel() const { return this->modelo; }

	inline void SetPosDisparo(const Vector3D& posDisparoToSet) { this->posDisparo = posDisparoToSet; }
	inline void SetVelocidadAtaque(const float& velocidadAtaqueToSet) { this->velocidadAtaque = velocidadAtaqueToSet; }
	inline void SetVelocidadMovimiento(const float& velocidadMovimientoToSet) { this->velocidadMovimiento = velocidadMovimientoToSet; }
	inline void SetModel(const Model& modeloToSet) { this->modelo = modeloToSet; }


	Cylinder* Disparo(const float& timeIncrement);

	bool Colision(Cylinder* Bala);

	void Desplazamiento(const float& timeIncrement);

	void Render();

	void Update(const float& timeIncrement);
};
```

Ilustración 3: Imagen del jugador en función de su vida

Las últimas tres clases están relacionadas entre sí (en especial las dos primeras), pues cada una representa a un tipo de enemigo, aunque con diferentes características.

Las clases enemigo1 y enemigo2 cuentan con casi los mismos atributos, aunque con diferentes movimientos y vidas.

Clase enemigo1

El enemigo 1 tiene el modelo de un cubo, cuentan con una vida y se moverán exclusivamente de izquierda a derecha o de arriba debajo de forma aleatoria.

```
class Enemigo1 : public Entidad
{
private:
	float velocidadMovimiento;
	Cube figura;

public:
	Enemigo1() : Entidad(), velocidadMovimiento(0.001), figura(Cube()) {}

	inline float GetVelocidadMovimiento() const { return this->velocidadMovimiento; }
	inline Cube GetFigura() const { return this->figura; }

	inline void SetVelocidadMovimiento(const float& velocidadMovimietnoToSet) { this->velocidadMovimiento = velocidadMovimietnoToSet; }
	inline void SetFigura(const Cube& figuraToSet) { this->figura = figuraToSet; }

	void Desplazamiento(const float& timeIncrement);

	bool Colision(Cylinder* Bala);

	void Render();

	void Update(const float& timeIncrement);

	bool Impacto(float limite);

};
```

Ilustración 4: Enemigo 1

Clase enemigo 2

El enemigo 2 tiene el modelo de una esfera, cuenta con dos vidas y puede moverse adicionalmente de forma diagonal.

```
class Enemigo2 : public Entidad
{
private:
	float velocidadMovimiento;
	Sphere figura;

public:
	Enemigo2() : Entidad(2, 1, Vector3D(1, 1, 1)), velocidadMovimiento(0.001), figura(Sphere()) {}

	inline float GetVelocidadMovimiento() const { return this->velocidadMovimiento; }
	inline Sphere GetFigura() const { return this->figura; }

	inline void SetVelocidadMovimiento(const float& velocidadMovimietnoToSet) { this->velocidadMovimiento = velocidadMovimietnoToSet; }
	inline void SetFigura(const Sphere& figuraToSet) { this->figura = figuraToSet; }

	void Desplazamiento(const float& timeIncrement);

	bool Colision(Cylinder* Bala);

	void Render();

	void Update(const float& timeIncrement);

	bool Impacto(float limite);

};
```

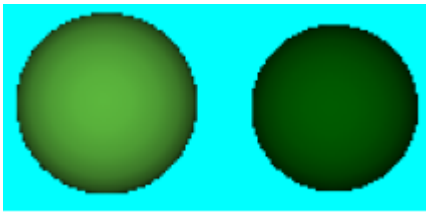

Ilustración 5: Enemigo 2

Clase torreta

El último enemigo se diferencia bastante de los demás, pues este se encuentra estático en un punto del escenario, pero disparará balas que pueden dar al jugador y quitarle una de sus vidas. Hablamos del enemigo torreta que está formado por un cubo y un cuboide que cuenta con 4 vidas.

```
class Torreta : public Entidad
{
private:
	float velocidadMovimiento;
	Cuboid canon;
	Cube figura;
	Vector3D posDisparo;
```

```
public:
	Torreta():	Entidad(4,1,Vector3D(1,1,1)),	velocidadMovimiento(0.01),
figura(Cube()), canon(Cuboid()) {}
	Torreta(float xf, float yf, float zf, float xc, float yc, float zc) :
Entidad(),		velocidadMovimiento(0.01),		figura(Cube(xf,yf,zf)),
canon(Cuboid(xc,yc,zc)) {}

	inline	float	GetVelocidadMovimiento()	const	{	return	this-
>velocidadMovimiento; }
	inline Cube GetFigura() const { return this->figura; }
	inline Cuboid GetCanon() const { return this->canon; }
	inline Vector3D GetposDisparo() const { return this->posDisparo; }

	inline void SetVelocidadMovimiento(const float& velocidadMovimietnoToSet) {
this->velocidadMovimiento = velocidadMovimietnoToSet; }
	inline	void	SetFigura(const	Cube&	figuraToSet)	{	this->figura	=
figuraToSet; }
	inline void SetCanon(const Cuboid& canonToSet) { this->canon = canonToSet;
}
	inline	void	SetPosDisparo(const	Vector3D&	posDisparoToSet)	{	this-
>posDisparo = posDisparoToSet; }

	Cylinder* Disparo(const float& timeIncrement);

	void Desplazamiento(const float& timeIncrement);

	bool Colision(Cylinder* Bala);

	void Render();

	void Update(const float& timeIncrement);

};
```

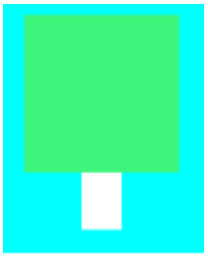

Ilustración 6: Enemigo torreta

Clase Scene0

Esta clase hereda de Scene y cuenta con la escena principal de menú donde se indica al jugador que tecla pulsar para jugar

```
#pragma once
#include "Scene.h"
using namespace std;
class Scene0 : public Scene
{
private:
	ModelLoader* loader;
public:
	Scene0(){}
	void Init0();
	void CreateText();
	void CreateModel();
};
```

Clase Scene1

Esta clase también hereda de Scene y cuenta con la implementación de cada nivel. Se puede indicar la cantidad de enemigos de cada nivel (no puede haber más de un enemigo de tipo torreta).

```
#pragma once
#include "Scene.h"
class Scene1 : public Scene
{
private:
	ModelLoader* loader;
public:
	Scene1(){}
	void Init1(bool vertex, bool box, float x, float y, float z,Color color,
bool fondo);
	void CreateEnemies(int numEnemigos1, int numEnemigos2,int numEnemigos3);

};
```

Clase Scene2

También hereda de Scene y se refiere a la pantalla de victoria.

```
#pragma once
#include "Scene.h"
using namespace std;
class Scene2 : public Scene
{
private:
	ModelLoader* loader;
public:
	Scene2() {}
	void Init2();
	void CreateText();
	void CreateModel();
};
```

Clase Scene3

También hereda de Scene y se refiere a la pantalla de derrota.

```
#pragma once
#include "Scene.h"
using namespace std;
class Scene3 : public Scene
{
private:
	ModelLoader* loader;
public:
	Scene3() {}
	void Init3();
	void CreateText();
	void CreateModel();
};
```

Mecánicas del juego y controles

Los controles del juego son bastante sencillos, el jugador puede desplazarse lateralmente usando las teclas “a” y “d” y podrá disparar usando la tecla “t”.

Como mecánica principal del juego está la mecánica del disparo. Tanto el jugador como el enemigo torreta podrá disparar una bala que quitará uno de vida. La bala aparecerá en la posición en la que la entidad disparó y seguirá una trayectoria

vertical. Si la bala no impacta contra ninguna entidad desaparecerá cuando salga de los límites del escenario.

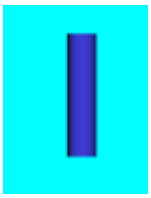

Ilustración 7: Bala

Otra mecánica importante implementada es la de que cuando una entidad choca contra los limites laterales del escenario, atravesará el muro y aparecerá al otro lado de la pantalla.

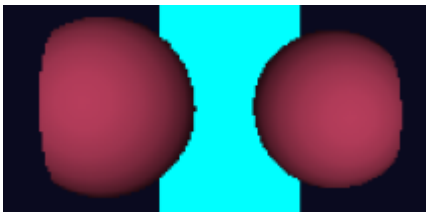

Ilustración 8: Ejemplo mecánica

Niveles

El juego está formado por tres niveles. En el primer nivel, el escenario es más grande y aparecen 3 enemigos del tipo 1.

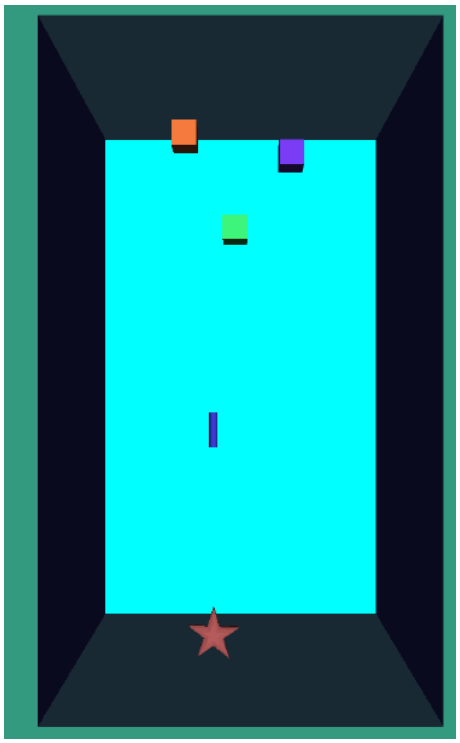

Ilustración 9: Nivel 1

En el segundo nivel, el escenario empequeñece un poco y aparecen 3 enemigos del tipo 2.

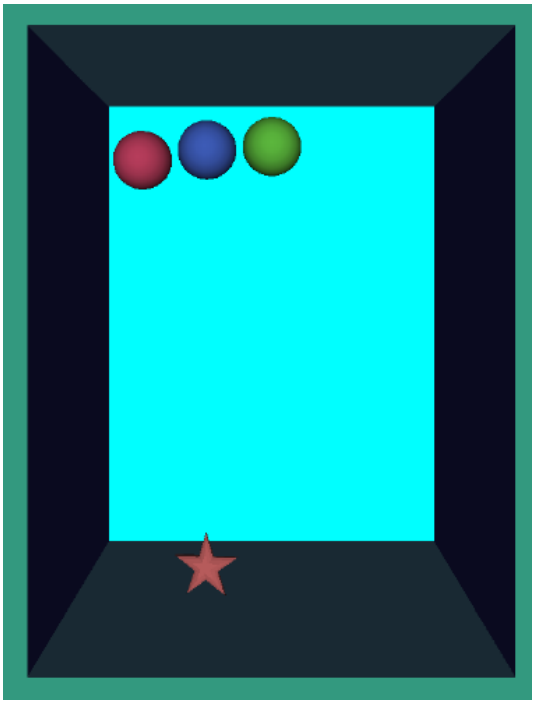

Ilustración 10: Nivel 2

En el tercer nivel, el escenario aún se hace más pequeño y cuenta con un enemigo de cada tipo (incluido una torreta).

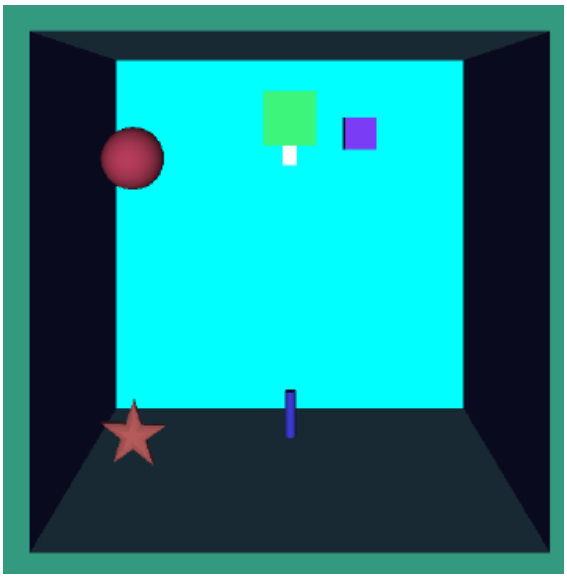

Ilustración 11: Nivel 3

UML y Plantilla de requisitos

Tipo de requisito	Requisito	Implementado (Sí/No)	Comentarios
Mínimo	Uso del motor	Sí	Motor implementado en clase
Mínimo	Condición de victoria y/o derrota	Sí	Ambas, al perder o a superar los niveles
Mínimo	Número de clases nuevas	Sí	5 clases nuevas explicadas en la memoria
Mínimo	Gestión de múltiples escenas	Sí	Escena de inicio, derrota, victoria y niveles
Mínimo	Uso de modelos obj	Sí	Estrella del jugador
Mínimo	Texto	Sí	Puntuaciones e indicaciones
Mínimo	Control de actualización	Sí	Implementado en el juego
Mínimo	Diagrama UML	Sí	En la memoria
Mínimo	Documento de concepto de juego	Sí	En la entrega
Notable	Vector 3D como plantilla	No	
Notable	Volumen de objetos en colisiones	Sí	Colisión entre objetos y límites del escenario
Notable	Varios niveles de juego	Sí	3 niveles propios
Notable	Sobrecarga del operador de flujo	No	
Notable	Ranking de puntuaciones en disco	No	
Sobresaliente	Modelos obj con materiales	No	
Sobresaliente	Generación procedural con memoria dinámica	No	
Sobresaliente	Físicas propias	No	
Sobresaliente	Otras	Sí	Mecánicas extra de disparo y atravesar paredes

A continuación, se muestran los diagramas UML correspondientes a nuestro juego:

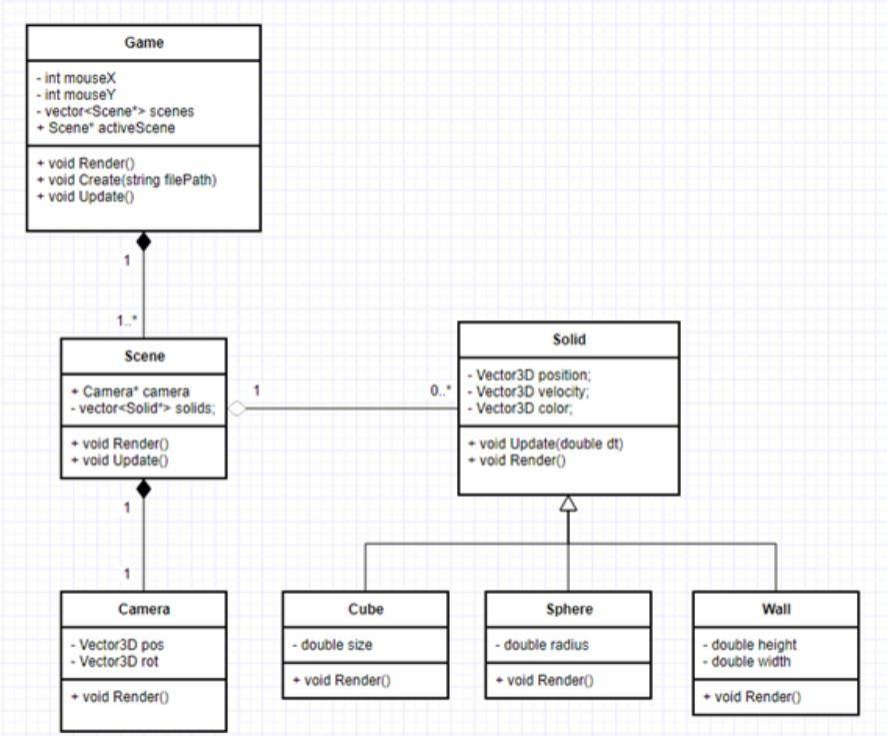

Ilustración 12: UML de las clases base

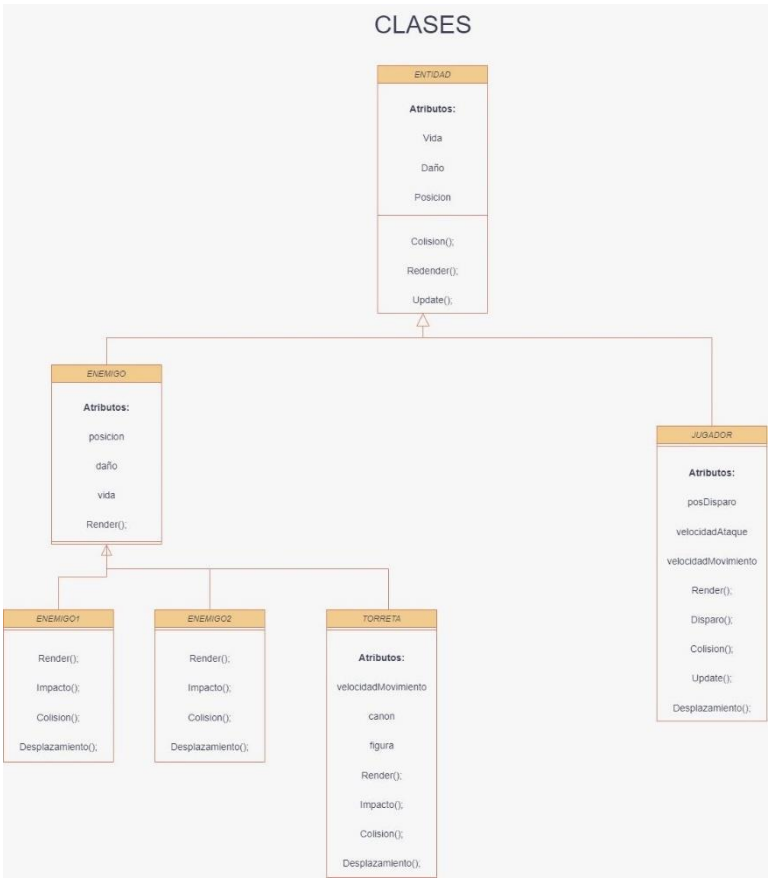

Ilustración 13: UML de clases propias

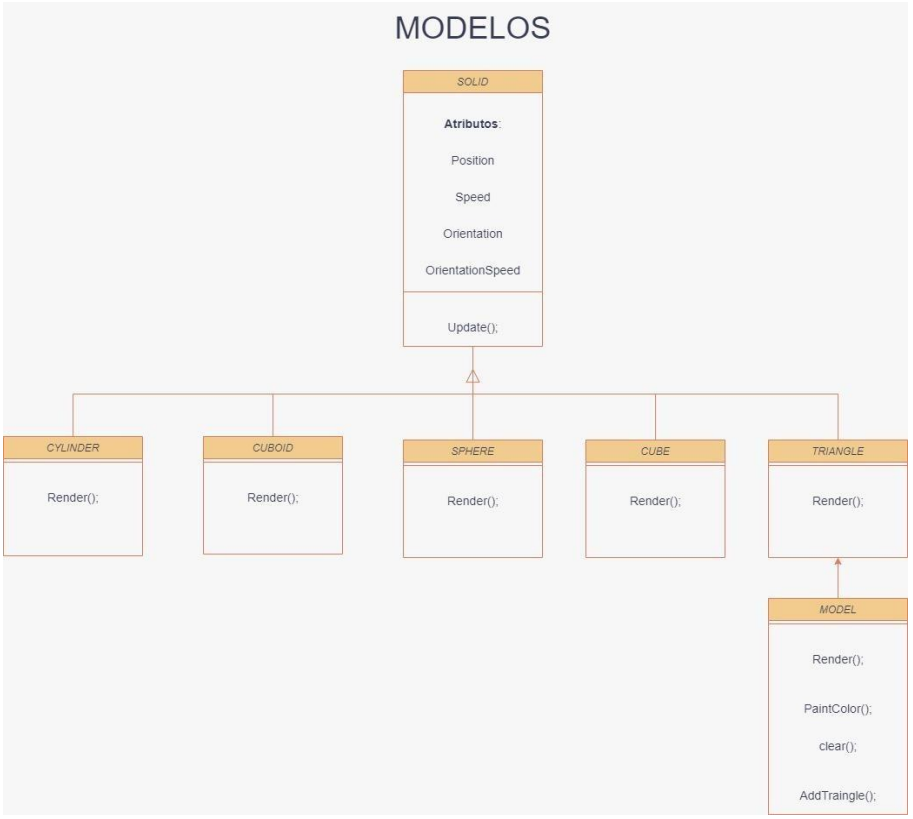

Ilustración 14: UML de los modelos

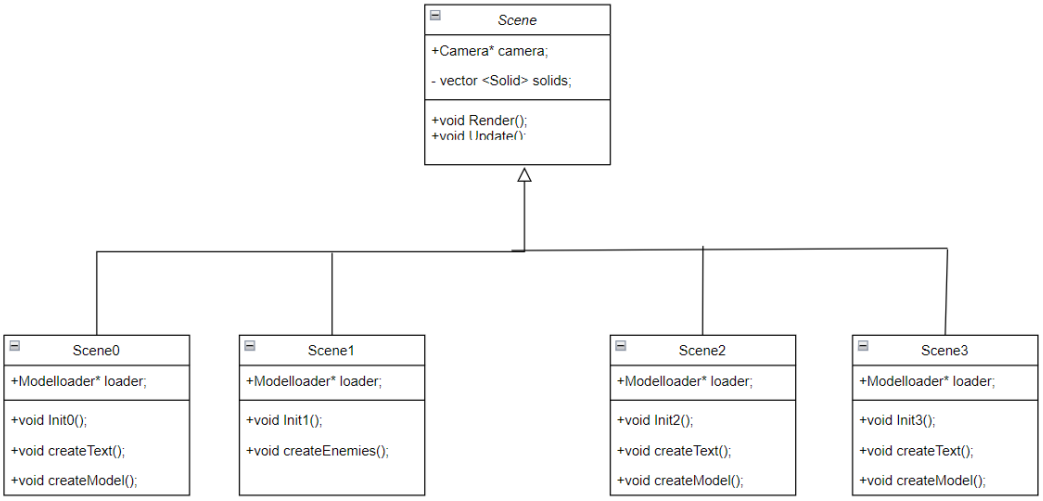

Ilustración 15: UML de Scene

Conclusión y preguntas

PREGUNTA 4.1.a: ¿Cuál es la finalidad del carácter & que aparece en la línea 28? ¿Qué ocurre si no se pone?
Se pone con la finalidad de evitar que se copien los objetos contenidos en triangles. El carácter & indica que es una referencia del objeto contenido y si este carácter no se pusiese, en su lugar se copiarían los objetos del vector, lo que puede llegar a suponer un problema para el proyecto de memoria o rendimiento.

PREGUNTA 4.1.b: ¿Qué elementos del código de este método están relacionados con el tratamiento de excepciones?
¿Qué implica este tipo de implementación para el programa? ¿Qué ocurriría en caso de que el archivo no existiera?
¿y si no existiera el código de tratamiento de la excepción?
Tanto el if que comprueba que el archivo está abierto como el catch y todo lo que contiene tienen que ver con el tratamiento de excepciones.
Esta implementación hace que, en caso de cualquier tipo de error, el control de excepciones nos muestre por consola que error nos ha ocurrido y así poder arreglarlo. Si el archivo no existiera, nos saltaría una excepción y en la consola aparecería un mensaje explicándonoslo.
Si no existiese el control de excepciones, en primer lugar, el código estaría incompleto ya que try necesita de catch para funcionar y, además pasaría que cuando el código nos de error, la excepción no se mostraría por consola y por ende nos costaría más descubrir como corregir el error.

PREGUNTA 4.1.c: Explica qué es ifstream, para qué se usa y cómo se utiliza.
Ifstream es una clase de C++ que nos indica que vamos a trabajar un fichero de entrada de datos. Esto normalmente se usa para leer los datos de un fichero existente y guardarnos en nuestro proyecto. En videojuegos, por ejemplo, se puede usar para retomar una partida desde un punto de guardado en un fichero de datos.
Para utilizar ifstream debemos incluir en nuestro proyecto la cabecera <fstream>, tras esto debemos crear un objeto ifstream con el nombre del archivo que se quiere leer, comprobar que ha sido abierto correctamente, leer los datos del archivo, cerrar el fichero y gestionar un control de excepciones para este.

PREGUNTA 4.1.d: Explica qué es istringstream, para qué se usa y cómo se utiliza.
Istringstream es otra clase de C++ que se utiliza para operaciones de entrada con cadenas de caracteres o strings. Esto se usa para tratar una cadena de caracteres como flujo de entrada y extraer valores de esta. Para usarlo, debemos incluir la cabecera <sstream> y crear un objeto stringstream. Podemos extraer datos de la cadena usando >>.

PREGUNTA 4.1.e: ¿Qué cambios habría que hacer en el código para poder procesar archivos obj con este formato?
En primer lugar, habría que añadir al método la forma de obtener las coordenadas de textura del triángulo y restarles el centro del modelo a estas coordenadas de texturas y, por último, habría que añadir el método centeredTriangle() estas coordenadas.

Gracias a esta práctica, hemos aprendido especialmente acerca de la programación orientada a objetos y optimizar la encapsulación en proyectos.

Ha sido interesante pensar en las mecánicas del juego, condición de victoria… En general, crear un motor gráfico desde 0 para hacer un videojuego de creación propia ha resultado muy gratificante y esperamos que nos sirva para el futuro.

Por último, hemos aprendido programar usando C++ y manejar visual studio para detectar errores, ayudarnos a la hora de hacer código y buscar e importar librerías o recursos externos que pueden ayudarnos en nuestros trabajos.

Bibliografía

https://cplusplus.com/

https://vandal.elespanol.com/rankings/videojuegos/shoot-em-up

https://es.wikipedia.org/wiki/Matamarcianos

https://www.aulavirtual.urjc.es/moodle/course/view.php?id=2067