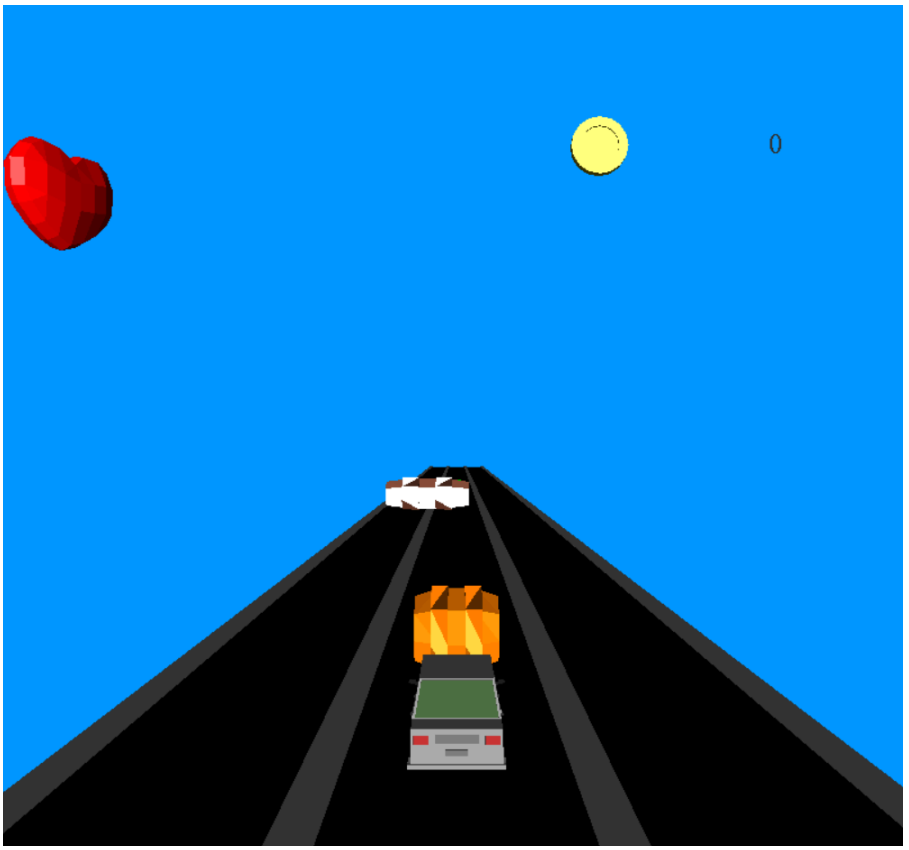
PROGRAMACIÓN AVANZADA - PRÁCTICA FINAL



GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

GRUPO 6:

Roberto Rojas Benítez

Daniel Mazo Sola

Alejandro Marín Sánchez

Miguel Amado Camuñas

INTRODUCCIÓN

En la siguiente memoria, se explicará la implementación de las diversas clases requeridas para la práctica, relatando los aspectos clave de su desarrollo y configuración, mediante un diagrama UML del programa, y las decisiones de diseño e implementación adoptadas para asegurar su funcionalidad y eficiencia.

Asimismo, también se proporcionará información acerca del juego a nivel conceptual mediante una versión simplificada del Game Concept Document, empleado habitualmente en la industria del videojuego, y cuya versión de mayor extensión se entregará en un documento distinto, así como una tabla de la plantilla de realización de requisitos de la práctica.

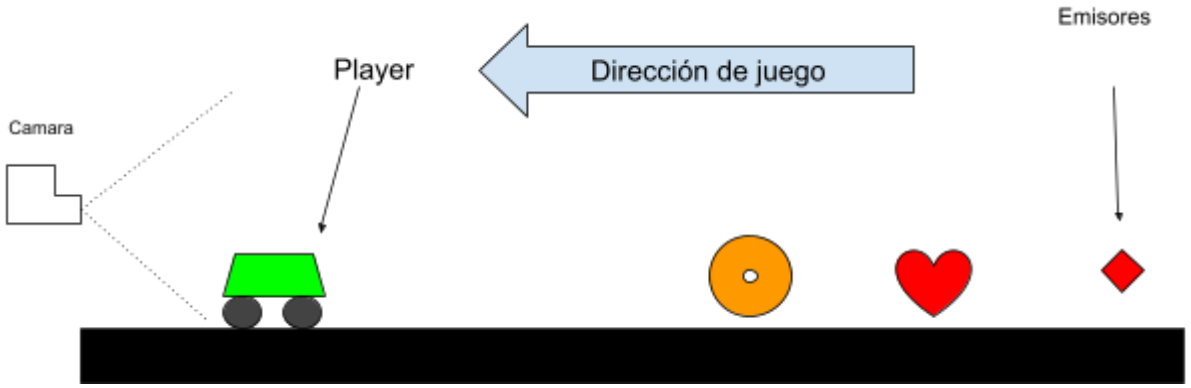
Durante la memoria, se empleará el siguiente formato de color para el código:

- Azul: Designa variables de una clase.
- Amarillo: Usado principalmente para constructores, caracterizados por utilizar en gran medida parametros con y sin valores por defecto, o por ser constructores por defecto.
- Verde: Representa los métodos de acceso de las diferentes clases.
- El color predeterminado se utiliza para todos los demás elementos.

1. CONCEPTO DE JUEGO

El concepto de juego se basa en un "*endless runner*", un género que se basa en correr a lo largo de varios carriles mientras el jugador esquiva obstáculos. Concretamente, en el juego desarrollado, el jugador dispone de tres carriles diferentes, representados mediante carreteras, para esquivar los obstáculos que aparecen aprovechando las clases desarrolladas en la práctica anterior.

El diagrama inferior muestra la distribución de los objetos puestos en la escena:



Se dibuja con unos pocos cuboides una carretera y unos separadores para delimitar los carriles de la carretera. La cámara se posiciona como una tercera persona encima del Player, con un plano en perspectiva, mostrando toda la carretera y al Player desplazándose a los largo de los 3 carriles.

Para el Player y el resto de Items utilizados en la práctica, se han utilizado modelos creados por terceros ajenos, los cuales han sido modificados en el software de código abierto Blender para centrar los modelos y cargarlos en el motor de juego.

El objetivo para ganar el juego consiste en recoger una cantidad de monedas mínimas para superar el nivel. Si se consigue, el Player habrá cumplido la condición de victoria y podrá pasar de nivel. En caso contrario, si el jugador pierde sus vidas, se le llevará a la escena de derrota y se le dará la opción de reiniciar el juego.

2. PLANTILLA DE CUMPLIMIENTO DE REQUISITOS

Tipo de requisitos	Requisito	Implementado (Sí/No)	Comentarios
Mínimo	Uso del motor	Sí	Se partió del código realizado en clase para la realización de la práctica
Mínimo	Condición de victoria y/o derrota	Sí	La condición de victoria se alcanza cuando se llega a un número mínimo de monedas recogidas en la escena. En el caso del GameScene lvl1 es de 30 monedas. Por el contrario, la condición de derrota se alcanza cuando se pierdan todas las vidas del jugador
Mínimo	Número de clases nuevas	Sí	El requisito mínimo de 8 clases se alcanza y supera con la jerarquía que se ha implementado en la realización de la práctica
Mínimo	Gestión de múltiples escenas	Sí	En la clase game del juego se gestionan las escenas, entre las que se encuentran: MenuScene, GameScene, LoseScene y WinScene
Mínimo	Uso de modelos obj	Sí	Los diferentes objetos del juego implementan su propio modelo obj. Entre ellos se encuentran: Heart, Ray, SpeedReduce, Shield, Player, Coin, Barrel, WideBarrel
Mínimo	Texto	Sí	Las escenas del juego utilizan texto para dar

			información sobre el estado del juego y las distintas opciones que el usuario puede decidir hacer.
Mínimo	Control de actualización	Sí	Para el control de la correcta actualización, cada escena llama a su método Update().
Mínimo	Diagrama UML	Sí	En la entrega, se proporciona el diagrama UML de las clases de las que hace uso el videojuego descrito en este documento.
Mínimo	Documento de concepto de juego	Sí	En la entrega de la tarea, se facilita a su vez un documento del concepto del juego, el cual transmite las principales características de este.
Notable	Vector3D como plantilla	No	
Notable	Volumen de objetos en colisiones	No	
Notable	Varios niveles de juego	Sí	El juego consta de 5 niveles
Notable	Sobrecarga del operador de flujo	No	
Notable	Ranking de puntuaciones en disco	No	
Sobresaliente	Modelos obj con materiales	Sí	Un ejemplo de esto es la clase Player que utiliza materiales.
Sobresaliente	Generación procedural con memoria dinámica	No	
Sobresaliente	Físicas propias	No	
Sobresaliente	Otras	No	

3. NUEVAS CLASES IMPLEMENTADAS / MODIFICADAS

Para implementar el juego, se han creado nuevas clases para su desarrollo:

- Clase Time:

Esta clase es una clase que en su mayor parte utiliza la librería <chrono>, está basada en la implementación de deltaTime del motor de Unity:

```
class Time
{
private:
	milliseconds interTime;

	float time;
	float timeScale;
	double deltaTime;
	double unscaledDeltaTime;

public:
	Time() : time(0), timeScale(1), deltaTime(0), unscaledDeltaTime(0)
	{
		interTime =
duration_cast<milliseconds>(system_clock::now().time_since_epoch());
	}

	inline float GetTime() { return time; }
	inline float GetDeltaTime() { return deltaTime; }
	inline double GetUnscaledDeltaTime() { return unscaledDeltaTime; }
	void SetTimeScale(float& timeScale);//Establece la velocidad a la que
pasa el tiempo

	void Run();//Actualiza el tiempo del juego
};
```

Con esta clase se puede poner un cronómetro interno al tiempo de uso de los *PowerUps*. De esta forma, se consigue que los *PowerUps* tengan una duración limitada, así como también se puede alterar la velocidad de transcurso del tiempo.

- Clase Player:

Esta clase permite controlar como el jugador interactúa directamente con el juego:

```
class Player : public Solid
{

public:

    enum PowerUp { None, Ray, Shield, SpeedReduce };

private:

    MaterialModel model;
    Sphere* shield = new Sphere(1.2, 15, 10);

    int lives;
    int coinsValue;

    bool activeShield = false;

    int carril;


    const float distanceColission = 1;
    const float distanceColissionWideBarrel = 5;

public:

    UICanva* uiCanva = nullptr;


    PowerUp powerUp;

    Player() : Solid(), lives(3), coinsValue(0), powerUp(None) {}

    inline float getDistanceColission() const { return
this->distanceColission; }
    inline float getDistanceColissionWide() const { return
this->distanceColissionWideBarrel; }

    inline void setPowerUp(const Player::PowerUp& pow) { this->powerUp = pow;
}

    inline int getLives() const { return this->lives; }

    int getCurrentPowerUp();

    inline int getCarril() const { return this->carril; }
    inline void setCarril(const int& carrilSet) { this->carril = carrilSet; }
```

```
    bool hasPowerUp() const;
    inline int getCoins() const { return coinsValue; }

    inline bool hasShieldActivated() const { return this->activeShield; }

    inline void setModelPlayer(const MaterialModel& mS) { this->model = mS; }

    inline void setShield(const bool& s) { this->activeShield = s;}

    int usePowerUp();
    void setUICanva(UICanva* ui);
    void notifyUICanva();


    void ResetPowerUp() { this->powerUp = None; }

    void applyCollisionEffect(CollisionEffect col);


    void Update(const float& timeUpdate);
    void Render();

    Solid* Clone(){

        return new Player(*this);
    }

};
```

El jugador tiene un número inicial y máximo de 3 vidas. Por otra parte, registra el número actual de coins y presenta un enumerador en el que se muestra que *PowerUp* ostenta para que, cuando desee utilizarlo, el usuario pulse el input correspondiente para ello. Los distintos powerUps implementados son los siguientes:

- **Rayo:** Destruye todos los barriles que haya en la escena en el momento que es activado.
- **Escudo:** Durante 10 segundos, el jugador no se verá afectado por los barriles de la escena. Se puede considerar como un periodo de invulnerabilidad.
- **Reducción de velocidad:** Durante 10 segundos los barriles reducirán su velocidad al 50%, ofreciendo al jugador un mayor tiempo de reacción.

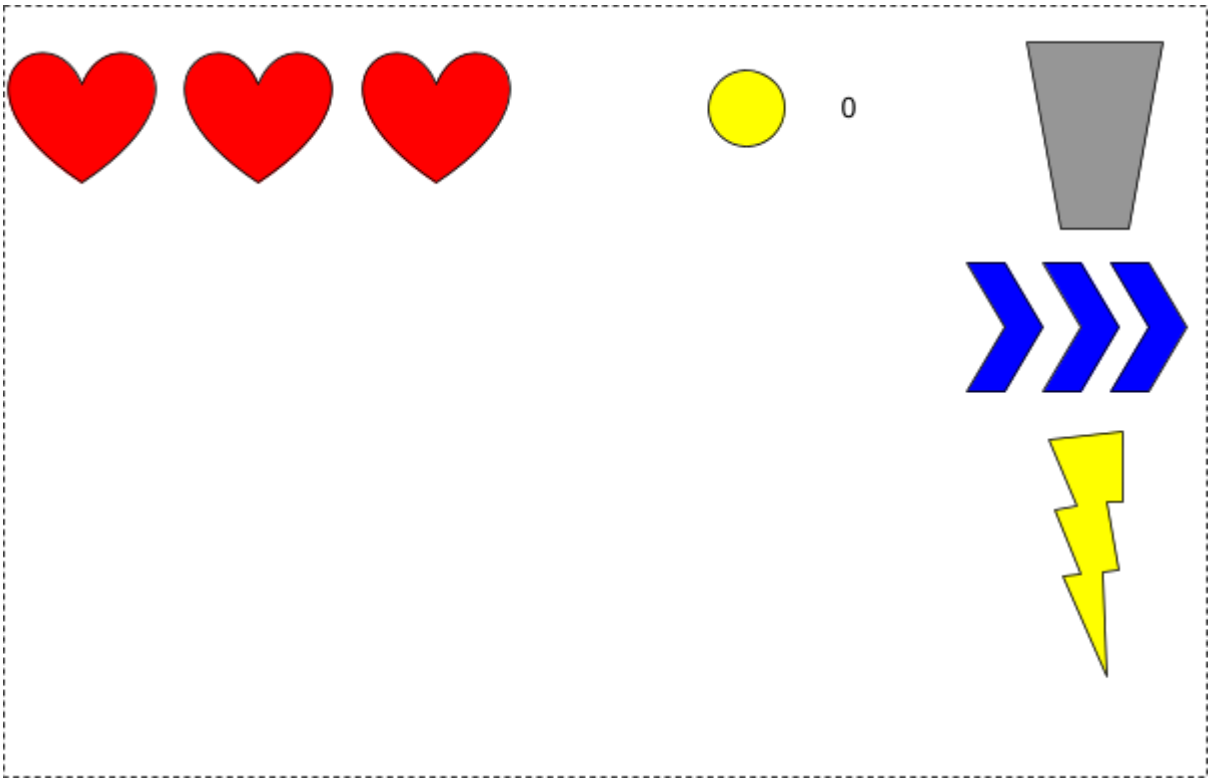
El player se encarga también de recibir las colisiones con los demás objetos de la escena y de notificar a su interfaz de actualizar el contenido de sus atributos. Además, al heredar de *Solid*, puede adquirir los atributos básicos de posición, rotación, velocidad, entre otros, lo que permite un mejor control del jugador.
El Player tiene un objeto llamado *MaterialModel model*, que representa gráficamente al jugador mediante un coche, debido a que este no tiene una malla poligonal que le

representa dentro del motor de juego. En el método abstracto *Render()*, se renderiza esta malla que adquirimos del modelo mediante la clase vista en el desarrollo guiado de *MaterialModel.h* y *.cpp*.

Para actualizar y gestionar los atributos del Player, se han utilizado getters y setters básicos. Además, este dispone de un constructor básico que al crearse, lo crea por defecto con 3 vidas, 0 monedas y ningún *PowerUp* asignado.

- Clase UICanva:

Esta clase se encarga de mostrar de manera gráfica los atributos del jugador, facilitando la comunicación visual con el mismo, basándose en las vidas y monedas actuales, y *PowerUp* disponible:



El *UICanva* se ha implementado heredando de *Solid*, de tal manera que se pueda posicionar con la cámara.

Los objetos son renderizados en el *UICanva* mediante el uso de punteros, en base a la información que reciba el *Canva* del Player. De esta manera, se logra ahorrar memoria en ejecución de forma óptima.

```
class UICanva : public Solid
{


private:

    ModelLoader loader;


    Text* coinsText;

    Model* heart1 = new Model();
    Model* heart2 = new Model();
    Model* heart3 = new Model();

    Model* rayPowerUpUI = new Model();
    Model* shieldPowerUpUI = new Model();
    Model* speedPowerUpUI = new Model();

    Model* activePowerUp = nullptr;

    Model* activeHeart1 = heart1;
    Model* activeHeart2 = heart2;
    Model* activeHeart3 = heart3;

    Model* coinUI = new Model();

    int coins;
    int lives;
    int powerUi;

public:


      UICanva() : coinsText(new Text( int(0) , Text::TimesNewRoman24)) ,
activePowerUp(nullptr) , powerUi(0){}


    void InitUI();
                void    UpdateCoinsText(const    int&    coinsValue)    {
this->coinsText->setText(coinsValue); }
    void UpdateHeartsUI(const int& currentLive);
    void SetActivePowerUpUI(const int& power);


    void Update(const float& timeUpdate);
    void Render();
    Solid* Clone();

};
```

Esta implementación de renderizar objetos mediante punteros sigue la misma filosofía que utiliza el vector de Escenas en la clase *Game.cpp*, que seleccionaba mediante un puntero que escena queríamos renderizar.

En todo momento, cuando se llama al método *NotifyUiCanva()* de Player, el canva recibe la información actual de Player y la muestra actualizando los punteros y cambiando atributos como el texto de las monedas.

- Clase Item:

Esta clase está diseñada a modo de “clase mediadora” entre su clase base Solid (que define las características generales de los objetos del juego) y los objetos del juego (*Barrel*, *PowerUp* y *Coin*). Su objetivo consiste en la simplificación de la gestión de colisiones de los elementos interactuables.

Se trata de una clase abstracta pura de la que no se instancian objetos, puesto que su funcionalidad es informar durante las colisiones de con qué tipo de objeto se choca, gracias al objeto de la clase *CollisionEffect* que permite saber qué cambios aplicarle al Player:

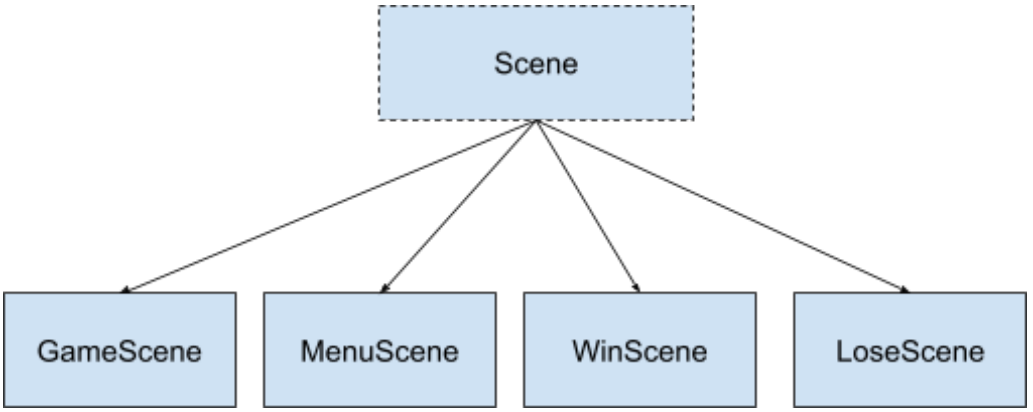
```
class Item : public Solid
{

public :

    Item() {}

    virtual CollisionEffect getCollisionEffect() = 0;
    virtual Item* Clone() = 0;
};
```

4. HERENCIA Y POLIMORFISMO

En la práctica, estos dos conceptos se han aplicado en diferentes partes:

- **Escenas**: se ha implementado una jerarquía de 4 diferentes tipos de escenas, las cuales heredan de su clase padre *Scene*.

  El polimorfismo en la jerarquía de escenas se ha implementado mediante el redefinición de sus métodos *Init()*, *Render()*, *Update()*, *CambioEscena()* y los métodos de gestión de la interacción entre el usuario y el programa




```
class Scene
{
private:
	vector<Solid*> gameObjects;
	Camera camera;
	bool sceneHasEnded;

public:

	Scene()
	{
		this->camera.SetPosition(Vector3D(0, 2, 20));
	}

	virtual void AddGameObject(Solid* gameObject);

	inline Camera GetCamera() const { return this->camera; }

	inline bool hasEndedScene() const { return this->sceneHasEnded; }


	inline void endScene(const bool& End) { this->sceneHasEnded = End; }
```

```
        virtual void Init();
        virtual void Render();
        virtual void Update(const float& timeUpdate);
        //virtual void Reset();

        virtual void cambioEscena(){}

        void ClearGameObject();


        virtual void ProcessKeyPressed(unsigned char key, int px, int py);
        virtual void ProcessSpecialKeyPressed(int key, int px, int py);
        virtual void ProcessMouseMovement(int x, int y) ;
    virtual void ProcessMouseClicked(int button, int state, int x, int y);
};
```

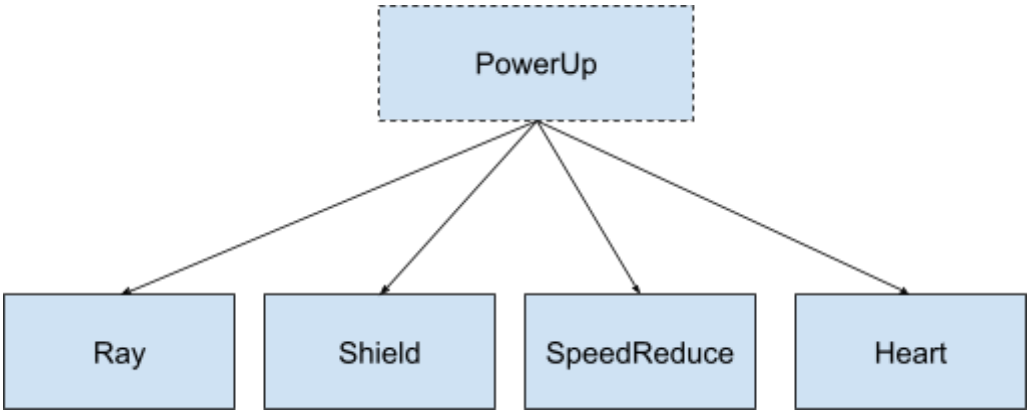
- **PowerUps**: Para los *PowerUps*, la herencia considerada ha recibido un enfoque más estético. Es decir, tiene en cuenta qué modelo renderiza el *PowerUp* y qué colisión devuelve al entrar en contacto con el Player.

```
class PowerUp : public Item {

private:

    Model itemModel;
    //Sphere* esfera = new Sphere(1);

public:

    PowerUp() {}

    inline void SetModel(const Model& mdl) { this->itemModel = mdl; }

    inline void PaintPowerUp(const Color& c) { this->itemModel.PaintColor(c);
}

    void Render() {

        this->itemModel.SetPosition(this->GetPosition());
        this->itemModel.SetOrientation(this->GetOrientation());
        this->itemModel.SetSpeed(this->GetSpeed());
        this->itemModel.SetOrientationSpeed(this->GetOrientationSpeed());
        this->itemModel.Render();
       //this->esfera->SetPosition(this->GetPosition());
       //this->esfera->Render();

    }

    virtual CollisionEffect getCollisionEffect() = 0;

    virtual  Item* Clone() = 0;
};
```

El Render de *PowerUp* muestra el modelo elegido, de forma que, al igual que en la implementación de Player, el modelo se posiciona en la posición real del objeto.

Redefinimos el método de colisión de tal manera que se obtiene un ID del *PowerUp* con el que se ha colisionado, siguiendo esta lógica numérica:

- **[1]**: Si el poder con el que se ha colisionado es un **Rayo**.
- **[2]**: En el caso que se trate de un **Escudo**.
- **[3]** : Si es una **Reducción de velocidad**.



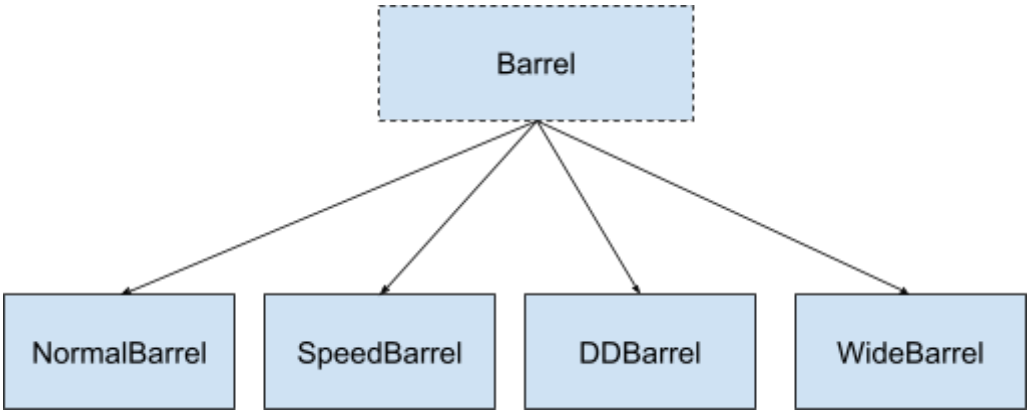
En el caso de colisionar con un **Corazón**, no se consigue un *PowerUp*, sino que se modifica la vida, conforme a la jerarquía interna de la clase Heart: cuando se genera un nuevo corazón, este puede ser de 3 tipos diferentes, y la colisión que se gestiona varía en función del tipo de corazón:

- **[+1] vida** si es un corazón normal (rojo)
- [**+2] vidas** si es un super corazón (amarillo)
- **[-1] vida** si es un corazón envenenado (morado oscuro, actuaría como una trampa para el jugador)

❖ **Barrels:** Al igual que en PowerUp, la jerarquía de clases diseñada para los barriles tiene como objetivo optimizar el aspecto visual de las clases, facilitando la selección y uso de diferentes modelos. Además, permite asignar una colisión específica para cada tipo de barril, lo que mejora la gestión y funcionalidad del sistema.

```
class Barrel : public Item
{

private:
    //Sphere* esfera = new Sphere(0.7);    //Esta esfera es para renderizar
las posiciones reales de los barriles en debug
    Model model;
```

```
public:

    Barrel() {}


    inline void SetModel(const Model& mdl) { this->model = mdl; }

    inline void PaintBarrel(const Color& c) { this->model.PaintColor(c); }


    void Render() {

        this->model.SetPosition(this->GetPosition());
        this->model.SetOrientation(this->GetOrientation());
        this->model.SetSpeed(this->GetSpeed());
        this->model.SetOrientationSpeed(this->GetOrientationSpeed());
        this->model.Render();
        //this->esfera->SetPosition(this->GetPosition());
        //this->esfera->Render();
    }

    virtual CollisionEffect getCollisionEffect() = 0;

    virtual  Item* Clone() = 0;
};
```



Al colisionar con los diferentes barriles de la jerarquía, se perderá 1 vida, exceptuando el caso del *DDBarrel* (Double Damage), con el que se perderán 2 vidas en lugar de 1.

Con respecto a los demás atributos, tienen una función estética y consiguen establecer una jerarquía de barriles sólida y consistente. Al crear diferentes instancias de estos objetos, su constructor tiene diferentes atributos aplicados:

- **Color:** Diferenciamos los diferentes tipos de barriles pintándolos con distintos colores para distinguirlos de forma clara y explícita.
- **Velocidad:** Cuando se instancian los barriles, se crean con una velocidad predefinida, simplificando el trabajo a la clase *Emitter*, que previamente se encargaba de asignar la velocidad a los objetos emitidos.

    - En este caso, se asigna en el constructor de cada objeto una velocidad específica, destacando *SpeedBarrel*, una clase de barril cuya velocidad es mayor al resto, con el objetivo de añadir dificultad y variedad al juego.
- **Malla poligonal:** Todos los barriles tienen un objeto asignado llamado *ModelLoader*, observado en el desarrollo guiado, que se encarga de cargar el modelo y asignarlo a la clase. Todos los barriles compartirán la misma malla, salvo *WideBarrel*, que es una instancia del modelo encalado en un eje (separador de la carretera) para que este barril ocupe 2 carriles y dificulte el gameplay.

★ **Caso especial: Clase Coin**

Este es un caso especial en el que se hereda directamente de Item, sin pasar por un filtro especial. Se ha considerado la posibilidad de agregar una jerarquía parecida a la de los objetos anteriores. Por ejemplo, monedas que sumen 2 al contador en lugar de 1 o incluso monedas de valor x5. No obstante, en este caso se ha optado por abstenerse para asegurar algo de duración de las partidas. Por lo tanto, la colisión que devuelve suma 1 al contador de monedas.

5. EMITTER Y EMITTER CONFIGURATION

En este apartado se explicarán qué cambios se han implementado en la clase *Emitter* para hacer de ésta un emisor de objetos más robusto y versátil:

```
class EmitterConfiguration
{
private:

    int maxParticles;
    int minBurstSize;
    int maxBurstSize;
    int minInterval;
    int maxInterval;
    long lifetimePerParticle;
    bool loopEmitter;

    vector<pair<Item*, float>> modelsProbability;

    bool useModels;

public:


    // Constructor por defecto
    EmitterConfiguration()
        :
        maxParticles(100),
        minBurstSize(1),
        maxBurstSize(5),
        minInterval(500),
        maxInterval(1500),
        lifetimePerParticle(5000),
        loopEmitter(false),
        useModels(false)
    {}

    EmitterConfiguration(
        const vector<pair<Item*, float>>& modelsWithProbabilities,
        int maxParticles = 10,
        int minBurst = 1,
        int maxBurst = 5,
        int minInt = 500,
        int maxInt = 1500,
        long particleLT = 5000,
        bool loop = true
    ) :
        modelsProbability(modelsWithProbabilities),
        maxParticles(maxParticles),
        minBurstSize(minBurst),
        maxBurstSize(maxBurst),
```

```
        minInterval(minInt),
        maxInterval(maxInt),
        loopEmitter(loop),
        lifetimePerParticle(particleLT),
        useModels(true)
    {
        // Validar que el vector no esté vacío y su tamaño sea válido
        if (modelsWithProbabilities.empty() || modelsWithProbabilities.size()
> 6) {
            throw invalid_argument("El vector de modelos debe contener entre
1 y 6 elementos.");
        }

        // Validar que todas las probabilidades sean positivas
        for (const auto& pair : modelsWithProbabilities) {
            if (pair.second <= 0.0f) {
                throw invalid_argument("Todas las probabilidades deben ser
mayores que 0.");
            }
        }

        // Verificar que la suma de las probabilidades sea igual a 1 (o
cercana)
        float sumProbabilities = 0.0f;
        for (const auto& pair : modelsWithProbabilities) {
            sumProbabilities += pair.second;
        }

        if (abs(sumProbabilities - 1.0f) > 0.01f) { // Tolerancia de ±0.01
            throw invalid_argument("La suma de las probabilidades debe ser
igual a 1.");
        }
    }

    inline bool IsUsingModels() const { return this->useModels; }

    inline int GetMaxParticles() const { return this->maxParticles; }
    inline int GetMinBurstSize() const { return this->minBurstSize; }
    inline int GetMaxBurstSize() const { return this->maxBurstSize; }
    inline int GetMinInterval() const { return this->minInterval; }
    inline int GetMaxInterval() const { return this->maxInterval; }

    inline bool GetIsLooped() const { return this->loopEmitter; }

    inline int getLifetimeParticle() const { return
this->lifetimePerParticle; }

    inline vector<pair<Item*,float>> getVectorAndProbabilites() const {
return this->modelsProbability; }
    inline int getVectorSize() const { return this->modelsProbability.size();
}

    inline Item* GetParticle(int index) const {
```

```
        if (index < 1 || index > modelsProbability.size()) {
            throw out_of_range("El índice está fuera del rango permitido.");
        }
        return modelsProbability.at(index - 1).first;

    }
};
```

- ☐ **Implementación de vector con Partícula y probabilidad.**

Para la generación de barriles, se ha implementado un sistema con el que se agrega a un vector un objeto que contiene un puntero a Item y un float con su probabilidad de aparición. Esto resulta útil porque se puede definir con qué probabilidades se van a generar ciertos barriles u *PowerUps* en cada uno de los niveles, facilitando la gestión de dificultad del juego.

Para ello, se utilizará como ejemplo el nivel 1: Los barriles normales tienen una probabilidad del 50% de aparecer, mientras que los barriles rápidos tienen un 30% de aparición y los barriles de doble daño el 20% restante.

```
Item* Emitter::generateSolidByProbability(const vector<pair<Item*, float>>&
solidsWithProbabilities) {
	// Calcular la suma de las probabilidades (por si no suman exactamente
1.0)
	float totalProbability = 0.0f;
	for (const auto& entry : solidsWithProbabilities) {
		totalProbability += entry.second;
	}

	// Generar un número aleatorio entre 0.0 y el total de probabilidades
	random_device rd;
	mt19937 gen(rd());
	uniform_real_distribution<float> dis(0.0f, totalProbability);

	float randomValue = dis(gen);
	float cumulative = 0.0f;

	// Evaluar qué probabilidad corresponde al valor generado
	for (const auto& entry : solidsWithProbabilities) {
		cumulative += entry.second;
		if (randomValue <= cumulative) {
			return entry.first->Clone();
		}
	}

	return nullptr; // No debería llegar aquí
}
```

El método superior es un método de la clase *Emitter* que permite generar los objetos en función de sus probabilidades.

Recibe como argumento el vector que se configura en el *EmitterConfiguration*, genera un float entre 0 y 1 mediante una distribución uniforme, y después compara el valor mediante una variable *cumulative*, que si está dentro de los rangos establecidos de la probabilidad, generará la partícula de la probabilidad correspondiente llamando a su método *Clone()*.

De esta manera, se puede llamar al siguiente método en el *Update* y asignar la nueva partícula al vector:

```
Item* newParticle =
generateSolidByProbability(conf.getVectorAndProbabilites());
```

- ☐ **Posibilidad de loopear el emisor y tiempo de vida de partículas.**

Para evitar entrar en un punto muerto en el gameplay, se ha añadido una funcionalidad extra, que permite seguir instanciando objetos hasta que el jugador pueda cumplir la condición de victoria o derrota. En el caso de que se supere el número máximo de partículas que puede tener el vector de partículas, éste no generará más aunque marquemos el loop del emisor. Para solucionar este problema, se ha implementado la eliminación de las partículas considerando un tiempo de vida, como se explica a continuación:

Creamos, además del vector de partículas, un vector adicional de <long> para guardar la vida de los objetos instanciados. Con esto conseguimos que el emisor registre el tiempo de los objetos, y si se quiere eliminar,se hace una comparación del tiempo de vida con el tiempo de vida establecido en la configuración del emisor:

```
	if (particleAge > conf.getLifetimeParticle())
	{
		// Liberar memoria y marcar como nullptr
		delete particlesVector[i];
		particlesVector[i] = nullptr;

		// Si el loop está habilitado, eliminar del vector
		if (conf.GetIsLooped())
		{
			particlesVector.erase(particlesVector.begin() + i);
			particleCreationTimes.erase(particleCreationTimes.begin()
+ i);
			// Ajustar índice
			i--;
		}
	}
	else
	{
		// Actualizar la partícula sólo si sigue siendo válida
		particlesVector[i]->Update(timeUpdate);
	}
}
```

De esta manera, se logra liberar espacio en el vector para seguir creando nuevos objetos.

- ☐ **Implementación de Ráfagas e intervalos de tiempo.**

Para algunos casos, como el de las monedas, se busca instanciar en una iteración más de una partícula para evitar que el gameplay se sienta lento y tedioso. Por esta razón, se implementa en la configuración un mínimo y un máximo de partículas en cada ráfaga, consiguiendo que puedan aparecer de 1 a 5 monedas en cada iteración.

Para que no se superpongan las instancias, se ha implementado un método auxiliar llamado *randomPositionOffsetZ()*, que se encarga de separar ligeramente las instancias, para que el jugador las pueda visualizar correctamente.

***NOTA:** En este método auxiliar se está utilizando static_cast, pero no se está empleando para obtener polimorfismo, sino que se trata de una ayuda implementada en el método*.

```
Vector3D Emitter::randomPositionOffsetZ(int particleId) {
     mt19937 generator(static_cast<unsigned int>(time(nullptr)) +
particleId);
     uniform_real_distribution<float> distribution(-10.0f, 10.0f);

     float offsetZ = distribution(generator);

     return Vector3D(0, 0, offsetZ);
}
```

```
int burstSize = generateRandom(conf.GetMinBurstSize(),
conf.GetMaxBurstSize());
```

Con estas comprobaciones previas a la generación de partículas mediante un bucle es posible instanciar varios objetos en una misma iteración de intervalo.

```
for (int i = 0; i < burstSize && particlesVector.size() <
conf.GetMaxParticles(); i++)
```

6. GESTIÓN DE COLISIONES

En este apartado, se analizará cómo se ha gestionado la información de las colisiones:

➔ **Cálculo matemático de vectores:**

En la clase Solid, se ha incluido un método llamado *Distance()*, que calcula la magnitud del vector que forma con otro sólido, accediendo a las posiciones de ambos objetos y calculando su magnitud:

```
float Solid::Distance(const Vector3D& other) {

    float x = other.GetX() - this->GetPosition().GetX();
    float xx = x * x;
    float y = other.GetY() - this->GetPosition().GetY();
    float yy = y * y;
    float z = other.GetZ() - this->GetPosition().GetZ();
    float zz = z * z;

    return sqrt(xx + yy + zz);

}
```

➔ **El emitter comprueba las colisiones:**

En la clase Emitter, hay un método denominado *CheckCollisions()*, cuya función es recorrer, mediante un bucle, los diferentes objetos instanciados y revisar las magnitudes de vectores explicadas en el apartado anterior

```
void checkCollisionsPlayer(Player player) {

    for (Item* particle : particlesVector) {

        if (particle->Distance(player.GetPosition()) <
player.getDistanceColission()) {

            player.applyCollisionEffect(particle->getCollisionEffect());
            player.notifyUICanva();
            this->removeParticle(particle);
        }


     }
}
```

En el caso de que la magnitud sea menor a la distancia predefinida en el Player, ocurrirá la colisión, por lo que se llamará a un método que tiene Player designado como *ApplyCollisionEffect()* y se destruirá la partícula, notificando al *Canva* del jugador para que actualice la información visual de la interfaz del juego.

➔ **Clase CollisionEffect:**

Esta es una clase auxiliar contenedora de información que será transmitida al Player. Esta implementación favorece, en gran medida, el encapsulamiento de información.

```
class CollisionEffect
{

private:

	int livesUpdate;
	int coinUpdate;
	int powerUpID;

public :

	CollisionEffect(int lU , int coins , int IDpu)  : livesUpdate(lU) , coinUpdate(coins) , powerUpID(IDpu){}

	inline int ColLives() const { return this->livesUpdate; }
	inline int ColCoins() const { return this->coinUpdate; }
	inline int ColPowerUpID() const { return  this->powerUpID; }

};
```

Esta clase solo tiene 3 variables, un constructor y 3 getters sencillos con los que se puede acceder a la información transmitida.

Cuando se llama a cualquier *Item* de juego, cada uno tiene un implementación diferente del método *CollisionEffect()*, devolviendo un efecto diferente en función de cada *Item*:

- **(-1,0,0):** El jugador perderá una vida (puede ser un barril o un corazón envenenado).
- **(1,0,0):** El jugador recupera una vida.
- **(2,0,0):** El jugador recupera dos vidas al haber recogido un supercorazón.
- **(0,1,0):** El jugador ha cogido una moneda.
- **(0,0,1):** El jugador ha conseguido un *PowerUp* de Rayo.
- **(0,0,2):** El jugador ha recogido un *PowerUp* de Escudo.
- **(0,0,3)**: El jugador ha obtenido un *PowerUp* de Reducción de velocidad.

Ejemplo de colisión con el *PowerUp* de Rayo:

```
CollisionEffect getCollisionEffect() { return CollisionEffect(0, 0, 1); }
```

➔ **Método del Player de ApplyCollisionEffect():**

Cuando ya se ha recibido una colisión, el Player procesa esa información y actualiza su atributos en función de lo que haya recibido:

```
void Player::applyCollisionEffect(CollisionEffect col) {


    if(!activeShield){
    this->lives += col.ColLives();
    }

    this->coinsValue += col.ColCoins();

    if (col.ColPowerUpID() != 0 && this->powerUp != None) { // solo podemos
tener un powerup a la vez

        switch (col.ColPowerUpID()) {

        case 1:
            this->setPowerUp(Ray);
            break;
        case 2:
            this->setPowerUp(Shield);
            break;
        case 3:
            this->setPowerUp(SpeedReduce);
            break;
        }
    }
    this->notifyUICanva();
}
```

7. CAMBIO DE ESCENAS

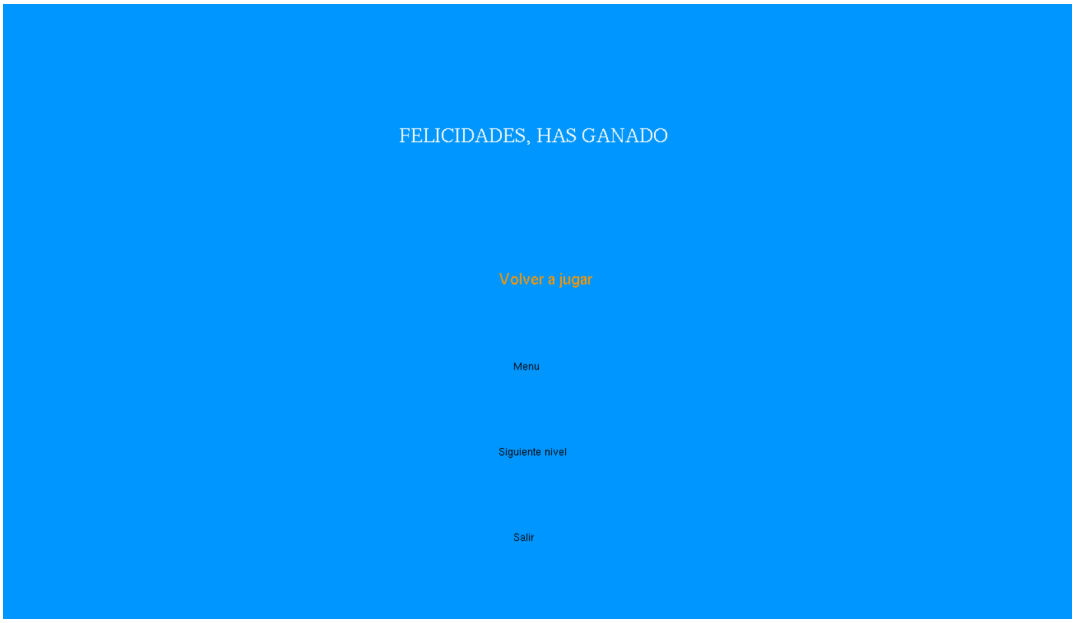
Los cambios de escena se han implementado en la clase Game, mediante la actualización del puntero de la escena activa:

- **Método Reset():**

El método *Reset()* se ha implementado únicamente en la clase *GameScene*, al ser ésta la única que requería que los atributos de la escena volvieran a sus valores iniciales. Para ello se llamó al método *ClearGameObject()* de escena que borra el vector de objetos de la clase padre *Scene* y los vectores de los emisores de barriles, monedas e *Items*. Además, se borró mediante delete todos los elementos de la *GameScene* y se pusieron a nulos (nullptr) para que no apuntarán a nada. Tras esto, se volvieron a inicializar todos ellos llamando al método *Init()* de *GameScene*.

- **Update en la Clase Game:**

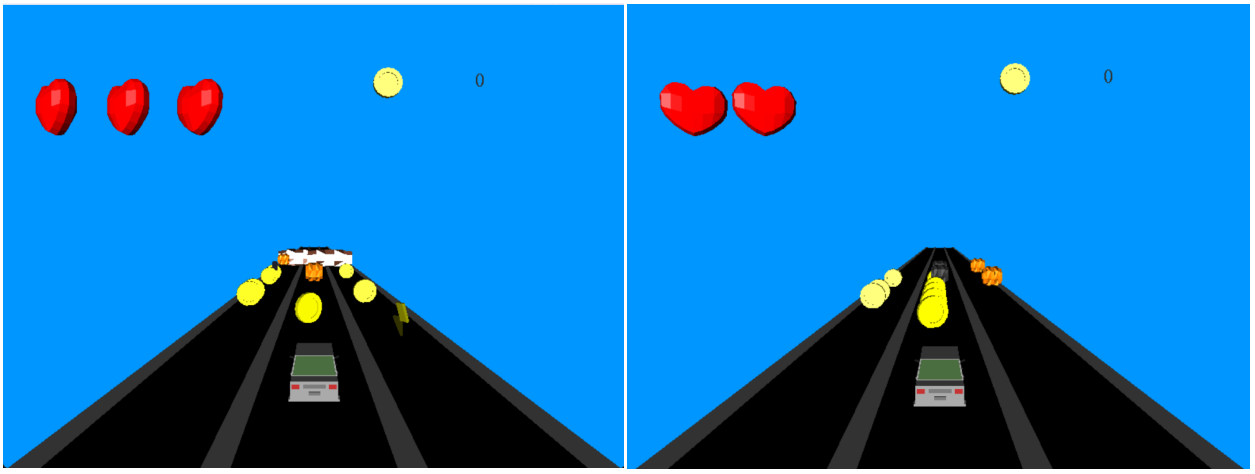
Cuando se quiere cambiar de escena, se llama al método endScene(true) en la escena que se ha terminado, y posteriormente al getter de *hasEndedScene()* heredado en todas las escenas. En el caso de que sea una escena de juego habría que comprobar la condición de victoria para saber si se ha ganado o perdido. En caso de haber ganado, el jugador es llevado al siguiente nivel, así hasta completar todos los niveles y llegar a *WinScene*, donde se finaliza el juego, podremos escoger si pasar al siguiente nivel, o saldremos del juego mediante `exit(0)`, finalizando la ejecución del programa.



8. RESULTADO FINAL Y CONCLUSIONES

El resultado final del juego es una experiencia de gameplay que puede ser entretenida dependiendo del estilo del jugador. La ejecución del programa es estable, sin errores, y se

destaca por una excelente jerarquía de clases, lo que contribuye a una estructura sólida y eficiente. La única autocrítica que se realiza es que los errores presentes podrían estar relacionados exclusivamente con el proceso de depuración del juego.



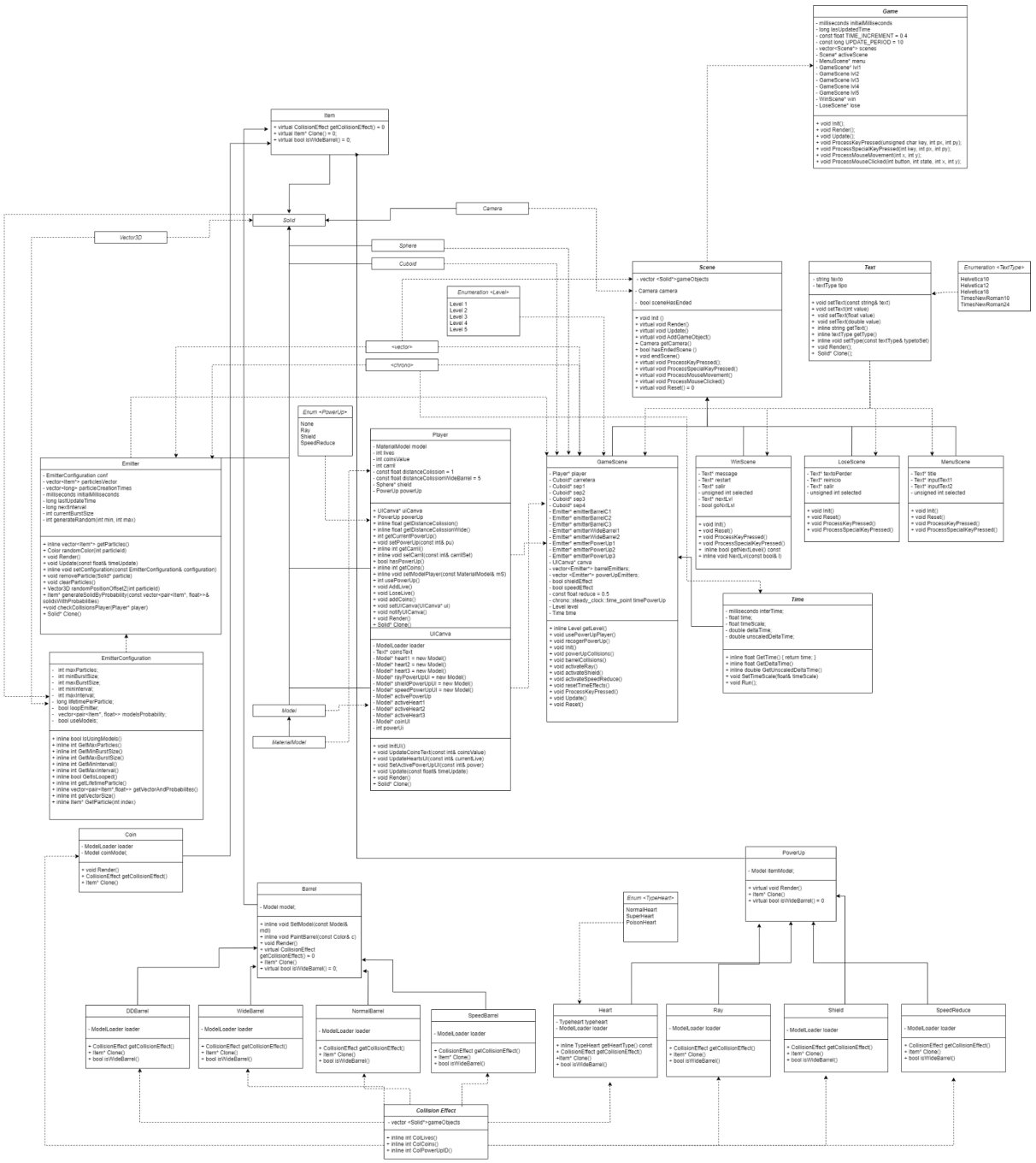
Se considera que durante todo el desarrollo se ha trabajado de forma correcta y respetando las especificaciones pedidas para este trabajo.

9. ARCHIVOS EXTERNOS

(VER CARPETA DONDE SE ADJUNTA IMAGEN Y ARCHIVO DEL DIAGRAMA UML)

(VER CARPETA DONDE SE ADJUNTA EL DOCUMENTO DE CONCEPTO DE JUEGO)

(VER CARPETA DONDE SE ADJUNTA EL VIDEO DE EJECUCIÓN)



10. BIBLIOGRAFÍA

Importación de modelos obtenidos de Internet mediante la página de Sketchfab:

- **Moneda:**
  - https://sketchfab.com/3d-models/lowpoly-gold-coin-34794c00e9d140f6b86e930fef18b009

- **Corazón:**
  - https://sketchfab.com/3d-models/low-poly-spinning-heart-17cf0dbe4435434eb6e04394fd5bf7ae
- **Escudo:**
  - https://sketchfab.com/3d-models/shield-a718744443134eab8c5e009060f5f564
- **Barril:**
  - Proporcionado por el equipo docente.
- **Player:**
  - https://sketchfab.com/3d-models/simple-car-low-poly-d2d1af06374141279ab68ab4b1444ba8

El resto de modelos utilizados en el proyecto son de creación propia.

Otros enlaces:

- **Programación C++:**
  - https://www.w3schools.com/cpp/default.asp