PROGRAMACIÓN AVANZADA: PRACTICA 3

Grado en Diseño y Desarrollo de Videojuegos URJC

CURSO 21/22

GONZALO BARRANCO CASTRO
Sede de Quintana

Contenido

INTRODUCCIÓN

La memoria de esta tercera parte del Proyecto de Programación Avanzada tiene el objetivo de explicar que nuevas clases y métodos se han implementado en el juego, a fin de que funcione correctamente.

Los detalles sobre el videojuego han sido especificados en el Documento de Descripción del Concepto del Juego, aun así, se hará aquí un pequeño resumen:

Thorn Mayhem es un juego del genero arcade, sin historia y donde no existe una condición de victoria. El jugador debe esquivar obstáculos, que aparecen en 5 posibles carriles por los que el personaje que se controla puede desplazarse, y conseguir tanta puntuación como le sea posible. Si el personaje choca con un obstáculo, la partida termina (externo al documento se ha dejado una partida completa para ver).

El juego cuenta con un sistema de guardado de la puntuación, colisiones y varias entidades que interactúan dentro del videojuego cuyo funcionamiento se describe en el apartado de entidades.

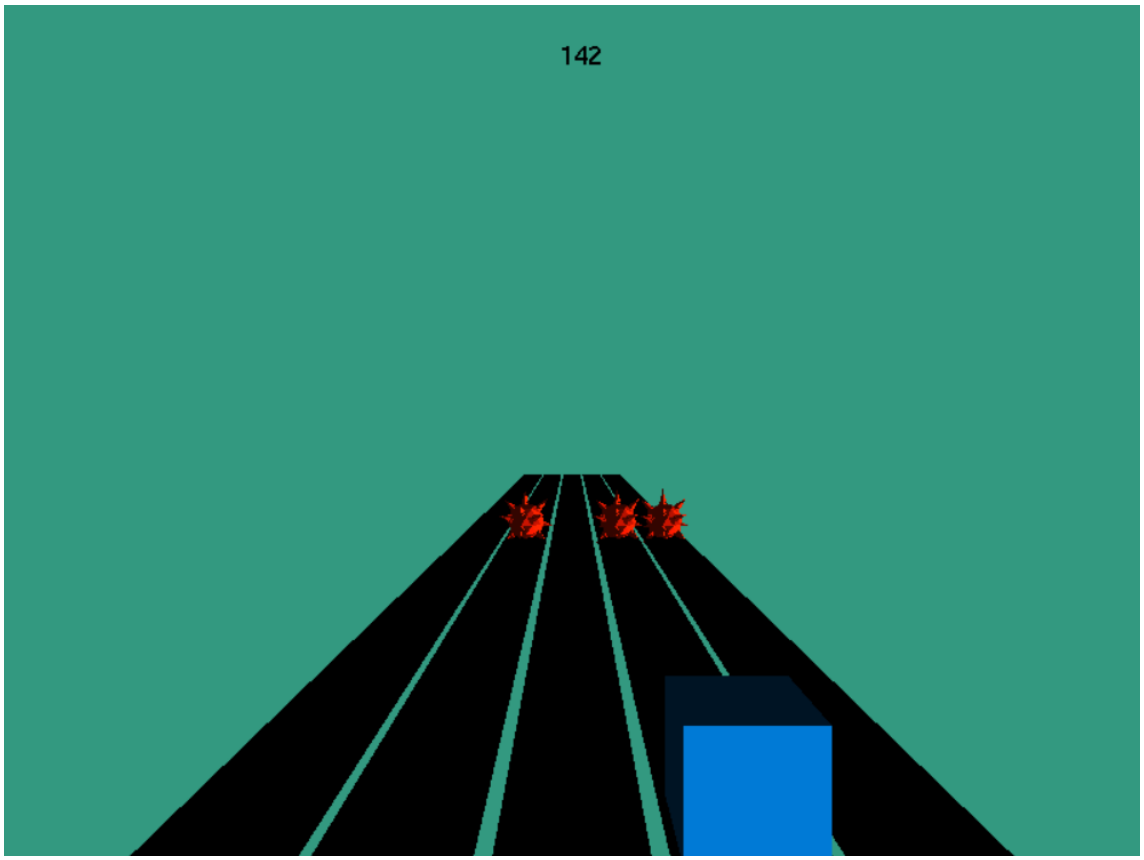
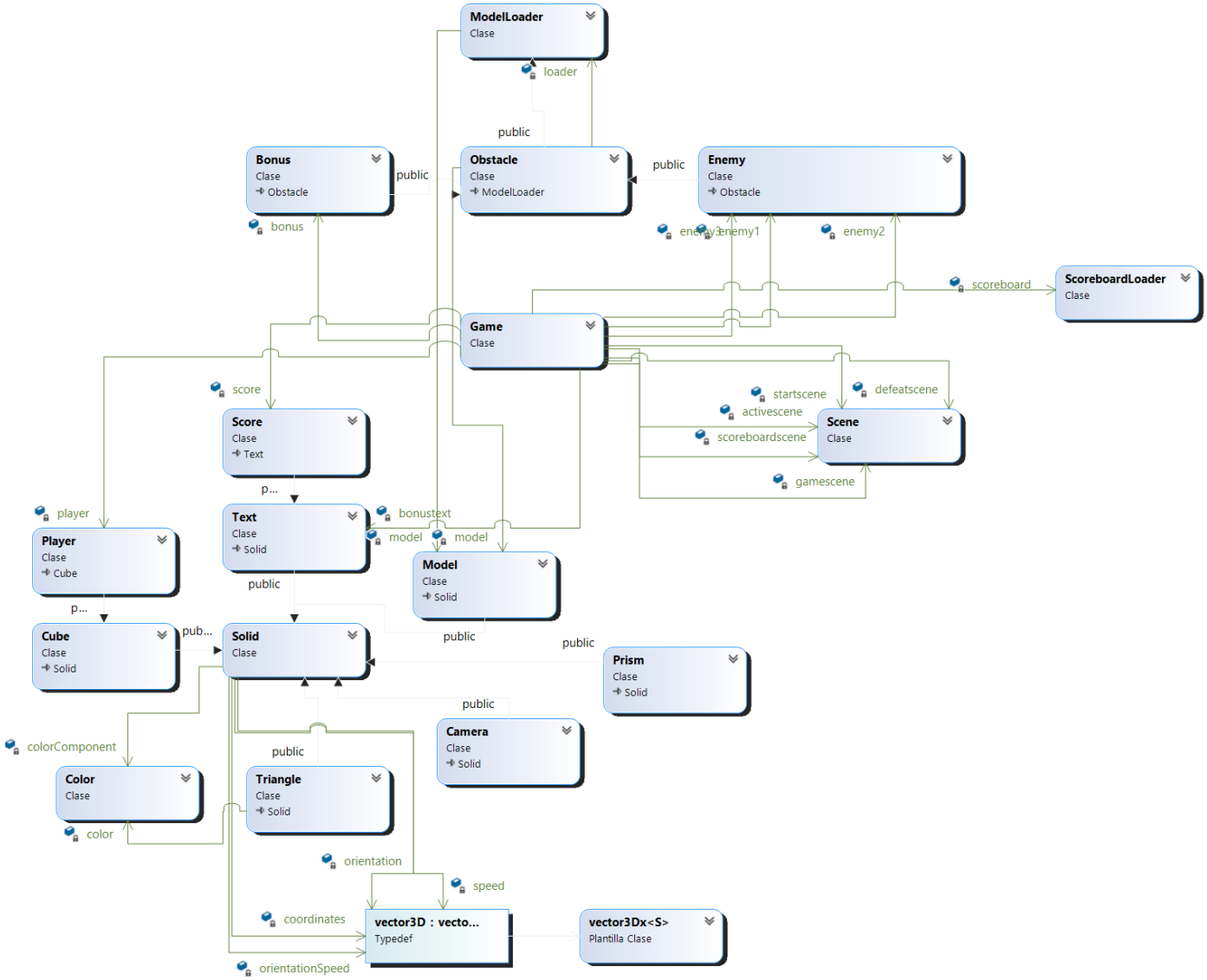


Ilustración 1. Imagen de una partida de Thorn Mayhem

DIAGRAMA UML



ENTIDADES

A la hora de elaborar el juego se implementaron diferentes clases que pueden englobarse dentro de una misma categoría denominada: “Entidad“.

Bajo esta definición se agrupan todos los objetos que tienen colisiones, requieren una actualización continua o tienen un movimiento.

PLAYER

Este es el objeto que ha de controlar el jugador. Hereda de la clase Cube ya que el modelo del jugador es un cubo de color azul.

```
class Player : public Cube
{
private:
	float finalCoord;
	bool move;

public:
	Player() : Cube(), finalCoord(0.0), move(false) {};

	inline const float GetFinalCoord() { return this->finalCoord; };
	inline const bool GetMove() { return this->move; };

	inline void SetFinalCoord(const float& dtime) { this->finalCoord =
dtime; }
	inline void SetMove(const bool& dmove) { this->move = dmove; }

	void Update(const float& time);

};
```

Para conseguir que el jugador se mueva cuando el jugador pulsa una tecla se hace mano de un método Update modificado que al heredar de Cube y por tanto de Solid será invocado por el objeto “scene” de Game.h.

```
void Player::Update(const float& time) {

	if (this->getcoordinateX() - this->GetFinalCoord() != 0) {
		this->SetMove(true);
		Solid::Update(time);
	}
	else { this->SetMove(false); }
}
```

finalCoord será una variable que indica la coordenada a la que ha de llegar el jugador. Entonces el método Update invoca el Update de Solid encargado del movimiento hasta que llega al destino.

Dentro de Game.cpp el programa detecta cuando el jugador pulsa “a” o “d” para mover el personaje, primero detecta si está en el borde y le da un valor a finalCoord para que se activa el movimiento junto a indicarle la dirección de la velocidad, es decir, si quiere desplazarse a la izquierda o la derecha.

OBSTACLE

Obstacle es una clase muy simple que será padre de las entidades que se comportan como un obstáculo del juego, es decir, que aparezca aleatoriamente sobre los caminos moviéndose hacia el jugador.

```
class Obstacle : public ModelLoader
{
private:
    Model* model;
    ModelLoader* loader;

public:
    Obstacle(string ruta, float size) {
        loader = new ModelLoader();
        loader->setSize(size);
        model = new Model;
        loader->LoadModel(ruta);
        *model = loader->getModel();
    }

    inline Model* GetModel() { return this->model; }
    inline void ClearLoader() { loader->Clear(); }
    virtual bool CheckCollision(Player* p, Solid* b) = 0;
};
```

Hereda de la clase Model ya que los obstáculos tienen un modelo importado a través de una carpeta. Por ello en el constructor se introduce su tamaño y la ruta hacia el modelo. También es una clase abstracta gracias al método virtual CheckCollision(). En las clases heredadas se definirá este método.

ENEMY Y BONUS

Ambas clases heredan de Obstacle y son bastante idénticas. Enemy es la clase a la que pertenecen los obstáculos dañinos del juego que hacen que el jugador pierda mientras que Bonus es la clase que pertenece a los obstáculos que dan puntuación extra al jugador.

Lo único que añaden es definir el método CheckCollision():

```
bool Enemy::CheckCollision(Player* p, Solid* b) {
	if ((p->getcoordinateX() > (b->getcoordinateX() + 2) || p-
>getcoordinateX() < (b->getcoordinateX() - 2)) ||
		(p->getcoordinateY() > (b->getcoordinateY() + 2) || p-
>getcoordinateY() < (b->getcoordinateY() - 2)) ||
		(p->getcoordinateZ() > (b->getcoordinateZ() + 2) || p-
>getcoordinateZ() < (b->getcoordinateZ() - 2)))
	{
		return false;
	}
	else {
		b->setcoordinateX(100.0);
		return true;
	}

}
```

Lo que hace este método es comprobar si el jugador y un objeto hijo de Solid que recibe estén a menos de cierta distancia en las coordenadas (x,y,z), si esta entre esas

coordenadas mueve el objeto fuera de la visión de la cámara para que la colisión no ocurra mas de 1 vez consecutiva.

SCORE

Este es el objeto encargado de llevar la puntación dentro del juego y de mostrarla por pantalla. La clase contiene una variable entera “points” que controla la cantidad de puntos que lleva el jugador.

```
class Score : public Text
{
private:
	int points;
	vector3D ogCoords = this->getCoordinates();

	inline void UpScore() { this->points++; }
	inline int CountDigits(const int n) { return int(log10(n)); }

public:
	Score() : Text(string("Score: "), vector3D(4.0, 11.0, -0.2)), points(0)
{};

	inline const int GetScore() { return this->points; }

	inline void SetScore(const string& text) { this->setText(text); }

	inline void Reset() { this->points = 0; }

	inline void ScoreBonus(const float& time) { this->points = this->points
+ 200 + time / 5; }


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

Dentro de la clase sus métodos se pueden dividir entre aquellos dedicados a modificar la puntación. Además de un clásico getter y setter, existen tres métodos adicionales: El primero es UpScore, que será invocado por el método Update para aumentar la puntuación cada ciclo. El segundo es ScoreBonus, que es llamado cuando el motor de colisiones detecta que el jugador ha chocado con un obstáculo de las lase Bonus dando una puntuación al jugador que aumenta según el tiempo que haya estado jugando. Por último, Reset se encarga de establecer la puntuación a 0.

Dentro de Update se encuentra lo siguiente:

```
void Score::Update(const float& time) {
	this->UpScore();

	this->SetScore(to_string(this->GetScore()));

	//ALIGN
	this->setCoordinates(ogCoords + vector3D(-0.1,0,0) *
CountDigits(points));

}
```

Lo que hace es aumentar la puntuación con el método UpScore y luego actualizar la clase Text dentro de Score para que muestre la nueva puntuación.

El fragmento comentado: “ALIGN” se encarga de centrar el texto gracias a la función CountDigits, que recibe la puntuación y devuelve cuantas cifras tiene, luego multiplica ese valor por -0.1 en el eje x para que el texto se centre al tiempo que aumenta de tamaño.

SISTEMAS DEL GAMEPLAY

En este índice se expondrán los sistemas acoplados dentro del Game.cpp con el fin de que el juego funcione con éxito.

COLISIONES

Tal y como se ha mostrado en las clases Enemy y Bonus ambas presentan un método para calcular sus colisiones, pero el resultado de dichas colisiones se dictaminará dentro de la clase Game.

Para comenzar y crear un código más adaptable y flexible se crean dos nuevos vectores: obs_vector y bonus_vector, cada uno almacena objetos de la clase Obstacle. A pesar de esto el programa se sirve del Polimorfismo para permitir almacenar objetos de la clase Bonus y Enemy gracias a que Obstacle es una clase abstracta. También se declaran la cantidad de enemigos y bonus deseada.

```
//OBSTACULOS
Enemy* enemy1;
Enemy* enemy2;
Enemy* enemy3;
Bonus* bonus;

//VARIABLES PARA LA GENERACION DE OBSTACULOS Y LAS COLISIONES
vector<Obstacle*> obs_vector;
vector<Obstacle*> bonus_vector;
```

Tras inicializarlos en Game::Init() se inserta cada objeto enemy o bonus en su vector correspondiente.

Posteriormente dentro del Update de Game se llaman a los siguientes métodos para comprobar las colisiones.

BonusCollision recorre todo el bonus_vector gracias a un “for each”, para cada uno de ellos comprueba llamando al método CheckCollision si efectivamente están colisionando. Si es el caso llama al método ScoreBonus() para añadir la puntuación correspondiente y mueve un objeto Text cerca del jugador llamado “bonus text” para que el jugador vea claramente cuanta puntuación ha ganado al recoger el Bonus.

```
void Game::BonusCollision(){
	for (Obstacle* var : bonus_vector) {
		if (var->CheckCollision(this->player, var->GetModel()) == true) {
			this->score->ScoreBonus(time_passed);
			this->bonustext->setText("+" + to_string((int)(200 +
time_passed / 5)));
			this->bonustext->setcoordinateX(this->player-
>getcoordinateX());
			this->bonustext->setcoordinateY(2.0);
		}
	}
}
```

Para la colisión con los enemigos el funcionamiento es idéntico. Pero el resultado es distinto. Una vez se choca con el enemigo todos los obstáculos vuelven a su posición inicial (En Z=20 ya que asi están fuera de la cámara, se profundizará más en el apartado de Generación procedural), luego coloca al jugador en la posición en la que empieza la partida, guarda la puntuación en un fichero gracias a GenerateScoreboard (Más sobre

esta función en el apartado de Scoreboard), resetea la puntuación y cambia de escena a la de derrota.

```
void Game::EnemyCollision(){
	for (Obstacle* var : obs_vector) {
		if (var->CheckCollision(this->player, var->GetModel()) == true) {
			time_passed = 0;
			for (Obstacle* var : obs_vector) { var->GetModel()-
>setcoordinateZ(20.0); }
			for (Obstacle* var : bonus_vector) { var->GetModel()-
>setcoordinateZ(20.0); }
			this->player->setcoordinateX(4.0);
			GenerateScoreboard();
			this->score->Reset();
			activescene = defeatscene;
		}
	}
}
```

GENERACION PROCEDURAL DE OBSTACULOS

El juego debe ser capaz de generar un desafío infinito para el jugador. Con ese fin se ha implementado un método llamado ObsGenerate() encargado de colocar de 1 a 5 enemigos y bonus en un carril aleatorio, y de aumentar su velocidad y cadencia progresivamente.

Primero es importante saber que el programa no crea ni destruye nuevos obstáculos, la cantidad deseada se declara, tal y como se ha descrito con anterioridad, en el Game.h. Para dar la sensación de desaparecer los objetos simplemente salen de la cámara del jugador. A continuación, se presenta una representación gráfica: El obstáculo viaja de z=-70 hasta z=50 donde se le devuelve a -70 para que continue el proceso.

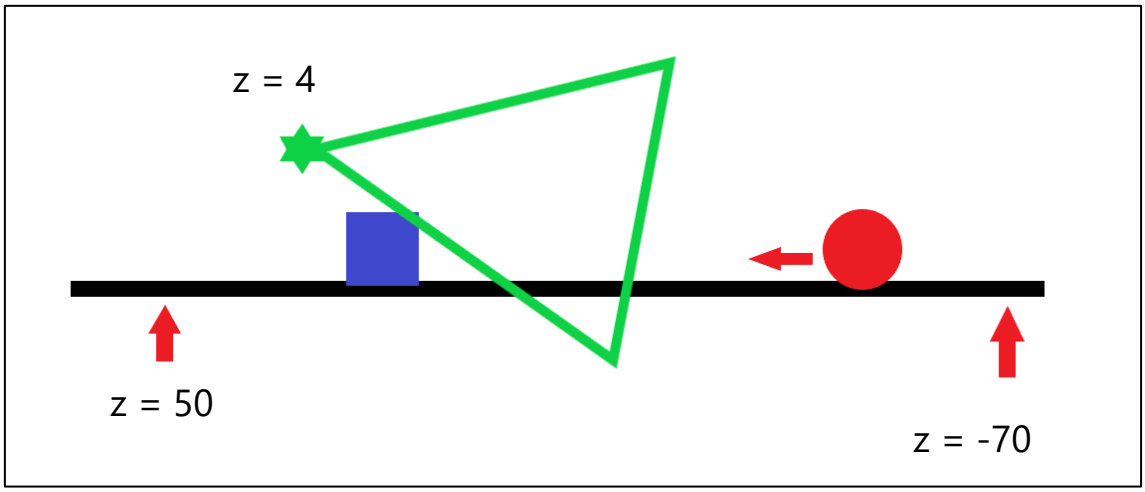


Ilustración 2. Diagrama sobre el funcionamiento de la generación de obstáculos

Gracias a que el número de obstáculos no puede subir de 5 este método no resulta demasiado ineficiente y es más útil a la hora de evitar errores gracias a su simpleza. Asimismo, gracias a que los obstáculos solo dependen de estar en z=50 para repetir el proceso de generación solo con aumentar la velocidad de los obstáculos también se aumenta la cadencia con la que se generan.

Para aumentar la velocidad se ha decidido emplear un sencillo algoritmo, siendo Speed la velocidad del obstáculo y t el tiempo que ha pasado de juego: $Speed = 1 + \frac{t}{4000}$. El tiempo que ha pasado también se calcula en el método Update y acumula los milisegundos ocurridos entre ciclo y ciclo de actualización. Adicionalmente es importante destacar que la velocidad tiene un limitador igual a 2.2, que se aplica al obstáculo si el resultado de la ecuación es mayor a ese valor.

A la hora de decidir donde ira cada obstáculo se utiliza dos algoritmos muy similares en Bonus y Enemy: Se declara en Game.h un array de 5 espacios, llamado "random_pos", y en el constructor se inicializa con la coordenada x de cada carril: {-1, 1.5, 4, 6.5, 9}. Cada vez que sea necesario volver a llevar los obstáculos a z = -70 el método desordena el array (empleando el método "random_shuffle" de la librería <algorithm>). Posteriormente un for each se encarga de pasar por cada objeto del vector que guarda los enemigos o los bonus para aplicarles una velocidad, la posición z=-70 y una posición en x determinada por su correspondiente índice en el random_pos (Es decir, el obstáculo en la posición 0 del vector obtiene la coordenada guardada en random_pos[0]).

Gracias a que en ocasiones Bonus y Enemies se van a superponer da la sensación de que no aparece un bonus en cada ciclo, a pesar de que lo hace, pero dentro de un enemigo.

```
void Game::ObsGenerate(const float& time, const float& time_passed) {
	if (this->enemy1->GetModel()->getcoordinateZ() >= 50) {

		//Aumentar velocidad
		float speed = 1.0 + time_passed / 4000;

		//Indice para ir pasando por los vectores
		int i;

		//ENEMIGO
		random_shuffle(random_pos, random_pos + 5);
		i = 0;
		for (Obstacle* var : obs_vector) {
			if (speed <= 2.2) var->GetModel()->setSpeedZ(speed);
			var->GetModel()->setcoordinateZ(-70);
			var->GetModel()->setcoordinateX(random_pos[i]);
			i++;
		}

		//BONUS
		random_shuffle(random_pos, random_pos + 5);
		i = 0;
		for (Obstacle* var : bonus_vector) {
			if (speed <= 2.2) var->GetModel()->setSpeedZ(speed);
			var->GetModel()->setcoordinateZ(-70);
			var->GetModel()->setcoordinateX(random_pos[i]);
			bonustext->setText("");
			i++;
		}
	}
}
```

ESCENAS

El último sistema que habilita el gameplay del que se hablará son la administración de Escenas. El juego presenta un total de 4, que se declaran en el Game.h, junto a otro objeto de la clase Scene que será igual a la escena activa en ese momento:

```
Scene* activescene;

//LAS ESCENAS DEL JUEGO
Scene* startscene = new(nothrow) Scene();
Scene* gamescene = new(nothrow) Scene();
Scene* defeatscene = new(nothrow) Scene();
Scene* scoreboardscene = new(nothrow) Scene();
```

De esta forma se permite al jugador moverse a través de las escenas cuando se cumplen ciertas condiciones o pulsa una tecla. A continuación, se muestra un diagrama con todas las escenas y como se conectan:

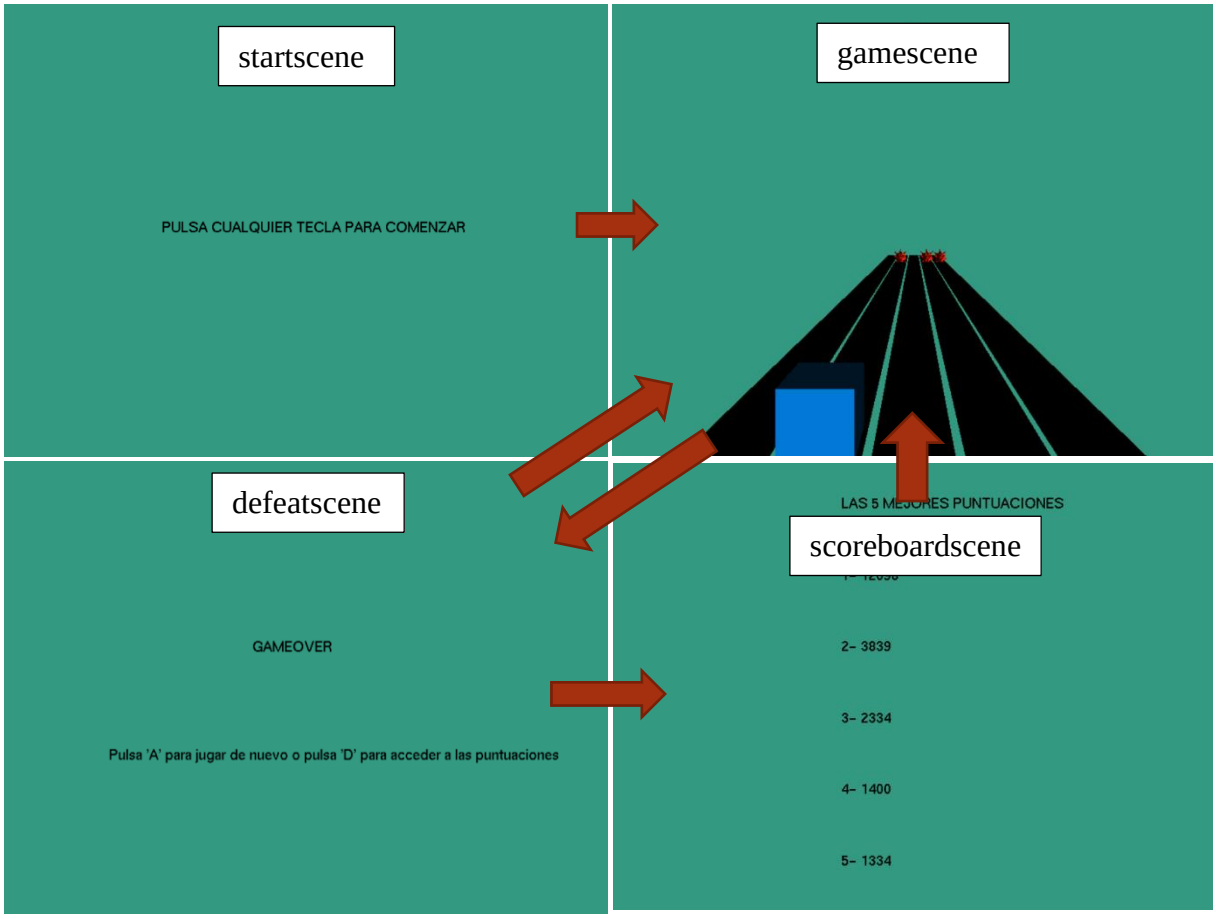


Ilustración 3. Diagrama con cada escena y a cuál se puede mover el jugador

SCOREBOARD

Dado a que este es un juego del genero arcade es recomendable guardar las puntuaciones del jugador de otras partidas para que pueda ver si en esta ha conseguido un nuevo record. Para esto el programa se sirve de un manejo de ficheros y funciones que se describirán a continuación en la clase ScoreboardLoader:

```
using namespace std;

class ScoreboardLoader
{
private:
	vector<int> scores;

public:
	void LoadScore(const int& score, string filePath);

	void OrderScore(string filePath);

	void InitScore(string filePath);

	int GetAnScore(const int& index);

	int GetSize() { return this->scores.size(); };
};
```

La clase guardará un vector de enteros que almacena todas las puntuaciones del jugador. Por la parte de los métodos LoadScore es el encargado de insertar cualquier nueva puntuación al fichero. Para ello inserta la nueva puntuación dentro del vector de enteros y luego el método se encarga de escribir todo el contenido del vector en el archivo .txt que se le índique a través de un String.

```
void ScoreboardLoader::LoadScore(const int& score, string filePath) {
	try {
		ofstream objFile(filePath);
		if (objFile.is_open()) {
			this->scores.push_back(score);
			for (int& var : this->scores) {
				objFile << to_string(var) << "\n";
			}
		}
		else {
			cout << "No se ha podido abrir el archivo desde: " <<
filePath;
		}
		objFile.close();
	}
	catch (exception& ex) {
		cout << "Excepcion al abrir: " << filePath;
		cout << ex.what();
	}
}
```

InitScore se encarga de insertar, gracias a un ifstream y un getLine(), los valores del archivo .txt al vector de enteros para poder trabajar con ello. Otro método que también emplea el ifstream de forma muy similar es OrderScore que inserta los valores del archivo .txt al vector score y luego los ordena de mayor a menor gracias a un método de la librería <algorithm> llamado “sort()”.

```
void ScoreboardLoader::OrderScore(string filePath) {
	try {
		this->scores.clear();
		ifstream objFile(filePath);
		if (objFile.is_open()) {
			string line;
			while (getline(objFile, line)) {
				this->scores.push_back(stoi(line));
			}
		}
		else {
			cout << "No se ha podido abrir el archivo desde: " <<
filePath;
		}
		sort(this->scores.begin(), this->scores.end(), greater<int>());
		objFile.close();
	}
	catch (exception& ex) {
		cout << "Excepcion al abrir: " << filePath;
		cout << ex.what();
	}
}
```

Los últimos métodos GetAnScore() y GetSize() simplemente devuelven un valor del vector dado un índice o su tamaño, respectivamente.

Una vez explicada la Clase solo falta su implementación dentro de Game.h. Primero se declaran 2 nuevas variables, un objeto ScoreboardLoader para administrar sus métodos y un vector de la clase Text que servirá para mostrar las mejores puntaciones en la escena “scoreboardscene”.

```
//SCOREBOARD
ScoreboardLoader* scoreboard;
vector<Text*> bestscores;
```

Dentro del Game::Init() se inicializa “scoreboard” y llama al método InitScore(). Posteriormente se inicializan también 5 objetos de la clase Text que compondrán el vector de “bestscores”.

El ultimo paso por tanto es dar un valor a cada uno de estos objetos para que se muestren las 5 mejores puntuaciones por pantalla mediante un método llamado “GenerateScoreboard()”:

```
void Game::GenerateScoreboard(){
	this->scoreboard->LoadScore(score->GetScore(), "..\\Scores\\Score.txt");
	this->scoreboard->OrderScore("..\\Scores\\Score.txt");
	for (int i = 0; i < 5; i++) {
		if (i<scoreboard->GetSize()) this->bestscores[i]-
>setText(to_string(i + 1) + "- " + to_string(this->scoreboard->GetAnScore(i)));
		else this->bestscores[i]->setText(to_string(i + 1) + "-
_________");
	}
}
```

Primero se guarda la nueva puntuación dentro del fichero y el vector de puntuaciones de “scoreboard”, posteriormente se ordena y luego entra en un bucle for de 5 iteraciones que pasa por cada uno de los índices y le da un valor a cada objeto Text para mostrar las 5 mejores puntuaciones. Para evitar errores en este ultimo algoritmo se establece la condición de que si la variable “i” es menor al tamaño del vector (es decir, hay menos de 5 puntuaciones guardadas) rellene ese objeto Text con un texto por defecto.

CONCLUSIONES

Siendo este uno de los primeros trabajos donde realmente se realiza un juego de toda la carrera lo he encontrado bastante satisfactorio. Considero que la forma en la que he organizado el código y me he tirado de los pelos durante horas tratando de buscar una solución a un problema son lecciones que extrapolaré a futuros proyectos.

También considero que engloba muy bien todos los contenidos del curso, dándonos una base sobre la que construir y lanzándonos posteriormente a crear algo con ello (con poca experiencia y recursos limitados) a un proyecto en el que en un comienzo me he encontrado muy perdido.

En conclusión, he disfrutado mucho creando este juego y el resultado final me hace sentirme orgulloso del esfuerzo que he invertido en el pensando, investigando y programando. Sin duda estas experiencias serán útiles en un futuro.